Matplotlib

Release 1.5.3

John Hunter, Darren Dale, Eric Firing, Michael Droettboom and the m

December 05, 2016

CONTENTS

Part I

Matplotlib v2.0

Changing the default colors and style of matplotlib has proven to be a much larger task than initially thought, but we are finally approaching the release.

For the full details of what has been changed see a draft of the release notes.

The current pre-release is v2.0.0rc1

You can install pre-releases via

```
pip install --pre matplotlib
```

which has source + wheels for Mac, Win, and manylinux or

using

```
conda install -c conda-forge/label/rc -c conda-forge matplotlib
```

which has binaries for Mac, Win, and linux. You can also install from `source` from git

```
git clone https://github.com/matplotlib/matplotlib.git
cd matplotlib
git checkout v2.0.0rc1
```

or tarball

```
wget https://github.com/matplotlib/matplotlib/archive/v2.0.0rc1.tar.gz -O matplotlib-v2.
↪0.0rc1.tar.gz
tar -xzvf matplotlib-v2.0.0rc1.tar.gz
cd matplotlib-v2.0.0rc1
```

via

```
pip install -v .
```

Part II

User's Guide

CHAPTER

ONE

INTRODUCTION

matplotlib is a library for making 2D plots of arrays in Python. Although it has its origins in emulating the MATLAB® [1] graphics commands, it is independent of MATLAB, and can be used in a Pythonic, object oriented way. Although matplotlib is written primarily in pure Python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays.

matplotlib is designed with the philosophy that you should be able to create simple plots with just a few commands, or just one! If you want to see a histogram of your data, you shouldn't need to instantiate objects, call methods, set properties, and so on; it should just work.

For years, I used to use MATLAB exclusively for data analysis and visualization. MATLAB excels at making nice looking plots easy. When I began working with EEG data, I found that I needed to write applications to interact with my data, and developed an EEG analysis application in MATLAB. As the application grew in complexity, interacting with databases, http servers, manipulating complex data structures, I began to strain against the limitations of MATLAB as a programming language, and decided to start over in Python. Python more than makes up for all of MATLAB's deficiencies as a programming language, but I was having difficulty finding a 2D plotting package (for 3D VTK more than exceeds all of my needs).

When I went searching for a Python plotting package, I had several requirements:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc.)
- Postscript output for inclusion with TeX documents
- Embeddable in a graphical user interface for application development
- Code should be easy enough that I can understand it and extend it
- Making plots should be easy

Finding no package that suited me just right, I did what any self-respecting Python programmer would do: rolled up my sleeves and dived in. Not having any real experience with computer graphics, I decided to emulate MATLAB's plotting capabilities because that is something MATLAB does very well. This had the added advantage that many people have a lot of MATLAB experience, and thus they can quickly get up to steam plotting in python. From a developer's perspective, having a fixed user interface (the pylab interface) has been very useful, because the guts of the code base can be redesigned without affecting user code.

The matplotlib code is conceptually divided into three parts: the *pylab interface* is the set of functions provided by `matplotlib.pylab` which allow the user to create plots with code quite similar to MATLAB

[1] MATLAB is a registered trademark of The MathWorks, Inc.

figure generating code (*Pyplot tutorial*). The *matplotlib frontend* or *matplotlib API* is the set of classes that do the heavy lifting, creating and managing figures, text, lines, plots and so on (*Artist tutorial*). This is an abstract interface that knows nothing about output. The *backends* are device-dependent drawing devices, aka renderers, that transform the frontend representation to hardcopy or a display device (*What is a backend?*). Example backends: PS creates PostScript® hardcopy, SVG creates Scalable Vector Graphics hardcopy, Agg creates PNG output using the high quality Anti-Grain Geometry library that ships with matplotlib, GTK embeds matplotlib in a Gtk+ application, GTKAgg uses the Anti-Grain renderer to create a figure and embed it in a Gtk+ application, and so on for PDF, WxWidgets, Tkinter, etc.

matplotlib is used by many people in many different contexts. Some people want to automatically generate PostScript files to send to a printer or publishers. Others deploy matplotlib on a web application server to generate PNG output for inclusion in dynamically-generated web pages. Some use matplotlib interactively from the Python shell in Tkinter on Windows™. My primary use is to embed matplotlib in a Gtk+ EEG application that runs on Windows, Linux and Macintosh OS X.

CHAPTER

TWO

CONFIGURATION GUIDE

2.1 Installing

There are many different ways to install matplotlib, and the best way depends on what operating system you are using, what you already have installed, and how you want to use it. To avoid wading through all the details (and potential complications) on this page, there are several convenient options.

2.1.1 Installing pre-built packages

Most platforms : scientific Python distributions

The first option is to use one of the pre-packaged python distributions that already provide matplotlib built-in. The Continuum.io Python distribution (Anaconda or miniconda) and the Enthought distribution (Canopy) are both excellent choices that "just work" out of the box for Windows, OSX and common Linux platforms. Both of these distributions include matplotlib and *lots* of other useful tools.

Linux : using your package manager

If you are on Linux, you might prefer to use your package manager. matplotlib is packaged for almost every major Linux distribution.

- Debian / Ubuntu : `sudo apt-get install python-matplotlib`
- Fedora / Redhat : `sudo yum install python-matplotlib`

Mac OSX : using pip

If you are on Mac OSX you can probably install matplotlib binaries using the standard Python installation program pip. See *Installing OSX binary wheels*.

Windows

If you don't already have Python installed, we recommend using one of the scipy-stack compatible Python distributions such as WinPython, Python(x,y), Enthought Canopy, or Continuum Anaconda, which have matplotlib and many of its dependencies, plus other useful packages, preinstalled.

For standard Python installations, install matplotlib using pip:

```
python -m pip install -U pip setuptools
python -m pip install matplotlib
```

In case Python 2.7 or 3.4 are not installed for all users, the Microsoft Visual C++ 2008 (64 bit or 32 bit for Python 2.7) or Microsoft Visual C++ 2010 (64 bit or 32 bit for Python 3.4) redistributable packages need to be installed.

Matplotlib depends on Pillow for reading and saving JPEG, BMP, and TIFF image files. Matplotlib requires MiKTeX and GhostScript for rendering text with LaTeX. FFmpeg, avconv, mencoder, or ImageMagick are required for the animation module.

The following backends should work out of the box: agg, tkagg, ps, pdf and svg. For other backends you may need to install pycairo, PyQt4, PyQt5, PySide, wxPython, PyGTK, Tornado, or GhostScript.

TkAgg is probably the best backend for interactive use from the standard Python shell or IPython. It is enabled as the default backend for the official binaries. GTK3 is not supported on Windows.

The Windows wheels (`*.whl`) on the PyPI download page do not contain test data or example code. If you want to try the many demos that come in the matplotlib source distribution, download the `*.tar.gz` file and look in the `examples` subdirectory. To run the test suite, copy the `lib\matplotlib\tests` and `lib\mpl_toolkits\tests` directories from the source distribution to `sys.prefix\Lib\site-packages\matplotlib` and `sys.prefix\Lib\site-packages\mpl_toolkits` respectively, and install nose, mock, Pillow, MiKTeX, GhostScript, ffmpeg, avconv, mencoder, ImageMagick, and Inkscape.

2.1.2 Installing from source

If you are interested in contributing to matplotlib development, running the latest source code, or just like to build everything yourself, it is not difficult to build matplotlib from source. Grab the latest *tar.gz* release file from the PyPI files page, or if you want to develop matplotlib or just need the latest bugfixed version, grab the latest git version *Source install from git*.

The standard environment variables `CC`, `CXX`, `PKG_CONFIG` are respected. This means you can set them if your toolchain is prefixed. This may be used for cross compiling.

> export CC=x86_64-pc-linux-gnu-gcc export CXX=x86_64-pc-linux-gnu-g++ export PKG_CONFIG=x86_64-pc-linux-gnu-pkg-config

Once you have satisfied the requirements detailed below (mainly python, numpy, libpng and freetype), you can build matplotlib:

```
cd matplotlib
python setup.py build
python setup.py install
```

We provide a setup.cfg file that goes with `setup.py` which you can use to customize the build process. For example, which default backend to use, whether some of the optional libraries that matplotlib ships with are installed, and so on. This file will be particularly useful to those packaging matplotlib.

If you have installed prerequisites to nonstandard places and need to inform matplotlib where they are, edit `setupext.py` and add the base dirs to the `basedir` dictionary entry for your `sys.platform`. e.g., if the header to some required library is in `/some/path/include/someheader.h`, put `/some/path` in the `basedir` list for your platform.

Build requirements

These are external packages which you will need to install before installing matplotlib. If you are building on OSX, see *Building on OSX*. If you are building on Windows, see *Building on Windows*. If you are installing dependencies with a package manager on Linux, you may need to install the development packages (look for a "-dev" postfix) in addition to the libraries themselves.

Required Dependencies

***python* 2.7, 3.4, or 3.5** Download python.

***numpy* 1.6 (or later)** array support for python (download numpy)

setuptools Setuptools provides extensions for python package installation.

***dateutil* 1.1 or later** Provides extensions to python datetime handling. If using pip, easy_install or installing from source, the installer will attempt to download and install `python_dateutil` from PyPI.

pyparsing Required for matplotlib's mathtext math rendering support. If using pip, easy_install or installing from source, the installer will attempt to download and install `pyparsing` from PyPI.

libpng 1.2 (or later) library for loading and saving *PNG* files (download). libpng requires zlib.

pytz Used to manipulate time-zone aware datetimes. http://pypi.python.org/pypi/pytz

***freetype* 2.3 or later** library for reading true type font files.

cycler 0.9 or later Composable cycle class used for constructing style-cycles

Optional GUI framework

These are optional packages which you may want to install to use matplotlib with a user interface toolkit. See *What is a backend?* for more details on the optional matplotlib backends and the capabilities they provide.

***tk* 8.3 or later, not 8.6.0 or 8.6.1** The TCL/Tk widgets library used by the TkAgg backend.

Versions 8.6.0 and 8.6.1 are known to have issues that may result in segfaults when closing multiple windows in the wrong order.

***pyqt* 4.0 or later** The Qt4 widgets library python wrappers for the Qt4Agg backend

***pygtk* 2.4 or later** The python wrappers for the GTK widgets library for use with the GTK or GTKAgg backend

***wxpython* 2.8 or later** The python wrappers for the wx widgets library for use with the WX or WXAgg backend

Optional external programs

ffmpeg/avconv or mencoder Required for the animation module to be save out put to movie formats.

ImageMagick Required for the animation module to be able to save to animated gif.

Optional dependencies

Pillow If Pillow is installed, matplotlib can read and write a larger selection of image file formats.

pkg-config A tool used to find required non-python libraries. This is not strictly required, but can make installation go more smoothly if the libraries and headers are not in the expected locations.

Required libraries that ship with matplotlib

***agg* 2.4** The antigrain C++ rendering engine. matplotlib links against the agg template source statically, so it will not affect anything on your system outside of matplotlib.

`qhull` 2012.1 A library for computing Delaunay triangulations.

`ttconv` truetype font utility

six 1.9.0 Python 2/3 compatibility library. Do not use this in third-party code.

Building on Linux

It is easiest to use your system package manager to install the dependencies.

If you are on Debian/Ubuntu, you can get all the dependencies required to build matplotlib with:

```
sudo apt-get build-dep python-matplotlib
```

If you are on Fedora/RedHat, you can get all the dependencies required to build matplotlib by first installing `yum-builddep` and then running:

```
su -c "yum-builddep python-matplotlib"
```

This does not build matplotlib, but it does get the install the build dependencies, which will make building from source easier.

Building on OSX

The build situation on OSX is complicated by the various places one can get the libpng and freetype requirements (darwinports, fink, /usr/X11R6) and the different architectures (e.g., x86, ppc, universal) and the different OSX version (e.g., 10.4 and 10.5). We recommend that you build the way we do for the OSX release: get the source from the tarball or the git repository and follow the instruction in `README.osx`.

Building on Windows

The Python shipped from http://www.python.org is compiled with Visual Studio 2008 for versions before 3.3, Visual Studio 2010 for 3.3 and 3.4, and Visual Studio 2015 for 3.5. Python extensions are recommended to be compiled with the same compiler.

Since there is no canonical Windows package manager, the methods for building freetype, zlib, and libpng from source code are documented as a build script at matplotlib-winbuild.

2.2 Customizing matplotlib

2.2.1 Using style sheets

Style sheets provide a means for more specific and/or temporary configuration modifications, but in a repeatable and well-ordered manner. A style sheet is a file with the same syntax as the `matplotlibrc` file, and when applied, it will override the `matplotlibrc`.

For more information and examples, see *Customizing plots with style sheets*.

2.2.2 Dynamic rc settings

You can also dynamically change the default rc settings in a python script or interactively from the python shell. All of the rc settings are stored in a dictionary-like variable called *matplotlib.rcParams*, which is global to the matplotlib package. rcParams can be modified directly, for example:

```
import matplotlib as mpl
mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.color'] = 'r'
```

Matplotlib also provides a couple of convenience functions for modifying rc settings. The *matplotlib.rc()* command can be used to modify multiple settings in a single group at once, using keyword arguments:

```
import matplotlib as mpl
mpl.rc('lines', linewidth=2, color='r')
```

The `matplotlib.rcdefaults()` command will restore the standard matplotlib default settings.

There is some degree of validation when setting the values of rcParams, see `matplotlib.rcsetup` for details.

2.2.3 The `matplotlibrc` file

matplotlib uses `matplotlibrc` configuration files to customize all kinds of properties, which we call `rc settings` or `rc parameters`. You can control the defaults of almost every property in matplotlib: figure size and dpi, line width, color and style, axes, axis and grid properties, text and font properties and so on. matplotlib looks for `matplotlibrc` in four locations, in the following order:

1. matplotlibrc in the current working directory, usually used for specific customizations that you do not want to apply elsewhere.
2. $MATPLOTLIBRC/matplotlibrc.
3. It next looks in a user-specific place, depending on your platform:
   - On Linux, it looks in .config/matplotlib/matplotlibrc (or $XDG_CONFIG_HOME/matplotlib/matplotlibrc) if you've customized your environment.
   - On other platforms, it looks in .matplotlib/matplotlibrc.

   See *matplotlib configuration and cache directory locations*.
4. *INSTALL*/matplotlib/mpl-data/matplotlibrc, where *INSTALL* is something like /usr/lib/python2.5/site-packages on Linux, and maybe C:\Python25\Lib\site-packages on Windows. Every time you install matplotlib, this file will be overwritten, so if you want your customizations to be saved, please move this file to your user-specific matplotlib directory.

To display where the currently active matplotlibrc file was loaded from, one can do the following:

```
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
'/home/foo/.config/matplotlib/matplotlibrc'
```

See below for a sample *matplotlibrc file*.

A sample matplotlibrc file

```
### MATPLOTLIBRC FORMAT

# This is a sample matplotlib configuration file - you can find a copy
# of it on your system in
# site-packages/matplotlib/mpl-data/matplotlibrc.  If you edit it
# there, please note that it will be overwritten in your next install.
# If you want to keep a permanent local copy that will not be
# overwritten, place it in the following location:
# unix/linux:
#     $HOME/.config/matplotlib/matplotlibrc or
#     $XDG_CONFIG_HOME/matplotlib/matplotlibrc (if $XDG_CONFIG_HOME is set)
# other platforms:
#     $HOME/.matplotlib/matplotlibrc
#
# See http://matplotlib.org/users/customizing.html#the-matplotlibrc-file for
# more details on the paths which are checked for the configuration file.
#
# This file is best viewed in a editor which supports python mode
# syntax highlighting. Blank lines, or lines starting with a comment
# symbol, are ignored, as are trailing comments.  Other lines must
# have the format
#    key : val # optional comment
```

```
#
# Colors: for the color values below, you can either use - a
# matplotlib color string, such as r, k, or b - an rgb tuple, such as
# (1.0, 0.5, 0.0) - a hex string, such as ff00ff or #ff00ff - a scalar
# grayscale intensity such as 0.75 - a legal html color name, e.g., red,
# blue, darkslategray

#### CONFIGURATION BEGINS HERE

# The default backend; one of GTK GTKAgg GTKCairo GTK3Agg GTK3Cairo
# CocoaAgg MacOSX Qt4Agg Qt5Agg TkAgg WX WXAgg Agg Cairo GDK PS PDF SVG
# Template.
# You can also deploy your own backend outside of matplotlib by
# referring to the module name (which must be in the PYTHONPATH) as
# 'module://my_backend'.
backend      : qt4agg

# If you are using the Qt4Agg backend, you can choose here
# to use the PyQt4 bindings or the newer PySide bindings to
# the underlying Qt4 toolkit.
#backend.qt4 : PyQt4        # PyQt4 | PySide

# Note that this can be overridden by the environment variable
# QT_API used by Enthought Tool Suite (ETS); valid values are
# "pyqt" and "pyside".  The "pyqt" setting has the side effect of
# forcing the use of Version 2 API for QString and QVariant.

# The port to use for the web server in the WebAgg backend.
# webagg.port : 8888

# If webagg.port is unavailable, a number of other random ports will
# be tried until one that is available is found.
# webagg.port_retries : 50

# When True, open the webbrowser to the plot that is shown
# webagg.open_in_browser : True

# When True, the figures rendered in the nbagg backend are created with
# a transparent background.
# nbagg.transparent : True

# if you are running pyplot inside a GUI and your backend choice
# conflicts, we will automatically try to find a compatible one for
# you if backend_fallback is True
#backend_fallback: True

#interactive  : False
#toolbar      : toolbar2   # None | toolbar2  ("classic" is deprecated)
#timezone     : UTC        # a pytz timezone string, e.g., US/Central or Europe/Paris

# Where your matplotlib data lives if you installed to a non-default
# location.  This is where the matplotlib fonts, bitmaps, etc reside
#datapath : /home/jdhunter/mpldata
```

```
### LINES
# See http://matplotlib.org/api/artist_api.html#module-matplotlib.lines for more
# information on line properties.
#lines.linewidth   : 1.0     # line width in points
#lines.linestyle   : -       # solid line
#lines.color       : blue    # has no affect on plot(); see axes.prop_cycle
#lines.marker      : None    # the default marker
#lines.markeredgewidth  : 0.5     # the line width around the marker symbol
#lines.markersize  : 6            # markersize, in points
#lines.dash_joinstyle : miter        # miter|round|bevel
#lines.dash_capstyle : butt          # butt|round|projecting
#lines.solid_joinstyle : miter       # miter|round|bevel
#lines.solid_capstyle : projecting   # butt|round|projecting
#lines.antialiased : True         # render lines in antialiased (no jaggies)

#markers.fillstyle: full # full|left|right|bottom|top|none

### PATCHES
# Patches are graphical objects that fill 2D space, like polygons or
# circles.  See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.patches
# information on patch properties
#patch.linewidth        : 1.0     # edge width in points
#patch.facecolor        : blue
#patch.edgecolor        : black
#patch.antialiased      : True    # render patches in antialiased (no jaggies)

### FONT
#
# font properties used by text.Text.  See
# http://matplotlib.org/api/font_manager_api.html for more
# information on font properties.  The 6 font properties used for font
# matching are given below with their default values.
#
# The font.family property has five values: 'serif' (e.g., Times),
# 'sans-serif' (e.g., Helvetica), 'cursive' (e.g., Zapf-Chancery),
# 'fantasy' (e.g., Western), and 'monospace' (e.g., Courier).  Each of
# these font families has a default list of font names in decreasing
# order of priority associated with them.  When text.usetex is False,
# font.family may also be one or more concrete font names.
#
# The font.style property has three values: normal (or roman), italic
# or oblique.  The oblique style will be used for italic, if it is not
# present.
#
# The font.variant property has two values: normal or small-caps.  For
# TrueType fonts, which are scalable fonts, small-caps is equivalent
# to using a font size of 'smaller', or about 83% of the current font
# size.
#
# The font.weight property has effectively 13 values: normal, bold,
```

```
# bolder, lighter, 100, 200, 300, ..., 900.  Normal is the same as
# 400, and bold is 700.  bolder and lighter are relative values with
# respect to the current weight.
#
# The font.stretch property has 11 values: ultra-condensed,
# extra-condensed, condensed, semi-condensed, normal, semi-expanded,
# expanded, extra-expanded, ultra-expanded, wider, and narrower.  This
# property is not currently implemented.
#
# The font.size property is the default font size for text, given in pts.
# 12pt is the standard value.
#
#font.family         : sans-serif
#font.style          : normal
#font.variant        : normal
#font.weight         : medium
#font.stretch        : normal
# note that font.size controls default text sizes.  To configure
# special text sizes tick labels, axes, labels, title, etc, see the rc
# settings for axes and ticks. Special text sizes can be defined
# relative to font.size, using the following values: xx-small, x-small,
# small, medium, large, x-large, xx-large, larger, or smaller
#font.size           : 12.0
#font.serif          : Bitstream Vera Serif, New Century Schoolbook, Century Schoolbook␣
↪L, Utopia, ITC Bookman, Bookman, Nimbus Roman No9 L, Times New Roman, Times,␣
↪Palatino, Charter, serif
#font.sans-serif     : Bitstream Vera Sans, Lucida Grande, Verdana, Geneva, Lucid,␣
↪Arial, Helvetica, Avant Garde, sans-serif
#font.cursive        : Apple Chancery, Textile, Zapf Chancery, Sand, Script MT, Felipa,␣
↪cursive
#font.fantasy        : Comic Sans MS, Chicago, Charcoal, Impact, Western, Humor Sans,␣
↪fantasy
#font.monospace      : Bitstream Vera Sans Mono, Andale Mono, Nimbus Mono L, Courier␣
↪New, Courier, Fixed, Terminal, monospace

### TEXT
# text properties used by text.Text.  See
# http://matplotlib.org/api/artist_api.html#module-matplotlib.text for more
# information on text properties

#text.color          : black

### LaTeX customizations. See http://wiki.scipy.org/Cookbook/Matplotlib/UsingTex
#text.usetex         : False  # use latex for all text handling. The following fonts
                              # are supported through the usual rc parameter settings:
                              # new century schoolbook, bookman, times, palatino,
                              # zapf chancery, charter, serif, sans-serif, helvetica,
                              # avant garde, courier, monospace, computer modern roman,
                              # computer modern sans serif, computer modern typewriter
                              # If another font is desired which can loaded using the
                              # LaTeX \usepackage command, please inquire at the
                              # matplotlib mailing list
#text.latex.unicode : False # use "ucs" and "inputenc" LaTeX packages for handling
```

```
                            # unicode strings.
#text.latex.preamble :  # IMPROPER USE OF THIS FEATURE WILL LEAD TO LATEX FAILURES
                            # AND IS THEREFORE UNSUPPORTED. PLEASE DO NOT ASK FOR HELP
                            # IF THIS FEATURE DOES NOT DO WHAT YOU EXPECT IT TO.
                            # preamble is a comma separated list of LaTeX statements
                            # that are included in the LaTeX document preamble.
                            # An example:
                            # text.latex.preamble : \usepackage{bm},\usepackage{euler}
                            # The following packages are always loaded with usetex, so
                            # beware of package collisions: color, geometry, graphicx,
                            # type1cm, textcomp. Adobe Postscript (PSSNFS) font packages
                            # may also be loaded, depending on your font settings

#text.dvipnghack : None      # some versions of dvipng don't handle alpha
                             # channel properly.  Use True to correct
                             # and flush ~/.matplotlib/tex.cache
                             # before testing and False to force
                             # correction off.  None will try and
                             # guess based on your dvipng version

#text.hinting : auto   # May be one of the following:
                       #   'none': Perform no hinting
                       #   'auto': Use freetype's autohinter
                       #   'native': Use the hinting information in the
                       #             font file, if available, and if your
                       #             freetype library supports it
                       #   'either': Use the native hinting information,
                       #             or the autohinter if none is available.
                       # For backward compatibility, this value may also be
                       # True === 'auto' or False === 'none'.
#text.hinting_factor : 8 # Specifies the amount of softness for hinting in the
                         # horizontal direction.  A value of 1 will hint to full
                         # pixels.  A value of 2 will hint to half pixels etc.

#text.antialiased : True # If True (default), the text will be antialiased.
                         # This only affects the Agg backend.

# The following settings allow you to select the fonts in math mode.
# They map from a TeX font name to a fontconfig font pattern.
# These settings are only used if mathtext.fontset is 'custom'.
# Note that this "custom" mode is unsupported and may go away in the
# future.
#mathtext.cal : cursive
#mathtext.rm  : serif
#mathtext.tt  : monospace
#mathtext.it  : serif:italic
#mathtext.bf  : serif:bold
#mathtext.sf  : sans
#mathtext.fontset : cm # Should be 'cm' (Computer Modern), 'stix',
                       # 'stixsans' or 'custom'
#mathtext.fallback_to_cm : True  # When True, use symbols from the Computer Modern
                                 # fonts when a symbol can not be found in one of
                                 # the custom math fonts.
```

```
#mathtext.default : it # The default font to use for math.
                       # Can be any of the LaTeX font names, including
                       # the special name "regular" for the same font
                       # used in regular text.

### AXES
# default face and edge color, default tick sizes,
# default fontsizes for ticklabels, and so on.  See
# http://matplotlib.org/api/axes_api.html#module-matplotlib.axes
#axes.hold           : True    # whether to clear the axes by default on
#axes.facecolor      : white   # axes background color
#axes.edgecolor      : black   # axes edge color
#axes.linewidth      : 1.0     # edge linewidth
#axes.grid           : False   # display grid or not
#axes.titlesize      : large   # fontsize of the axes title
#axes.labelsize      : medium  # fontsize of the x any y labels
#axes.labelpad       : 5.0     # space between label and axis
#axes.labelweight    : normal  # weight of the x and y labels
#axes.labelcolor     : black
#axes.axisbelow      : False   # whether axis gridlines and ticks are below
                               # the axes elements (lines, text, etc)

#axes.formatter.limits : -7, 7 # use scientific notation if log10
                               # of the axis range is smaller than the
                               # first or larger than the second
#axes.formatter.use_locale : False # When True, format tick labels
                                   # according to the user's locale.
                                   # For example, use ',' as a decimal
                                   # separator in the fr_FR locale.
#axes.formatter.use_mathtext : False # When True, use mathtext for scientific
                                     # notation.
#axes.formatter.useoffset      : True    # If True, the tick label formatter
                                         # will default to labeling ticks relative
                                         # to an offset when the data range is very
                                         # small compared to the minimum absolute
                                         # value of the data.

#axes.unicode_minus  : True    # use unicode for the minus symbol
                               # rather than hyphen.  See
                               # http://en.wikipedia.org/wiki/Plus_and_minus_signs
 ↪#Character_codes
#axes.prop_cycle    : cycler('color', 'bgrcmyk')
                                            # color cycle for plot lines
                                            # as list of string colorspecs:
                                            # single letter, long name, or
                                            # web-style hex
#axes.xmargin        : 0  # x margin.  See `axes.Axes.margins`
#axes.ymargin        : 0  # y margin See `axes.Axes.margins`

#polaraxes.grid      : True    # display grid on polar axes
#axes3d.grid         : True    # display grid on 3d axes
```

```
### TICKS
# see http://matplotlib.org/api/axis_api.html#matplotlib.axis.Tick
#xtick.major.size     : 4      # major tick size in points
#xtick.minor.size     : 2      # minor tick size in points
#xtick.major.width    : 0.5    # major tick width in points
#xtick.minor.width    : 0.5    # minor tick width in points
#xtick.major.pad      : 4      # distance to major tick label in points
#xtick.minor.pad      : 4      # distance to the minor tick label in points
#xtick.color          : k      # color of the tick labels
#xtick.labelsize      : medium # fontsize of the tick labels
#xtick.direction      : in     # direction: in, out, or inout

#ytick.major.size     : 4      # major tick size in points
#ytick.minor.size     : 2      # minor tick size in points
#ytick.major.width    : 0.5    # major tick width in points
#ytick.minor.width    : 0.5    # minor tick width in points
#ytick.major.pad      : 4      # distance to major tick label in points
#ytick.minor.pad      : 4      # distance to the minor tick label in points
#ytick.color          : k      # color of the tick labels
#ytick.labelsize      : medium # fontsize of the tick labels
#ytick.direction      : in     # direction: in, out, or inout


### GRIDS
#grid.color       :   black   # grid color
#grid.linestyle   :   :       # dotted
#grid.linewidth   :   0.5     # in points
#grid.alpha       :   1.0     # transparency, between 0.0 and 1.0

### Legend
#legend.fancybox      : False  # if True, use a rounded box for the
                               # legend, else a rectangle
#legend.isaxes        : True
#legend.numpoints     : 2      # the number of points in the legend line
#legend.fontsize      : large
#legend.borderpad     : 0.5    # border whitespace in fontsize units
#legend.markerscale   : 1.0    # the relative size of legend markers vs. original
# the following dimensions are in axes coords
#legend.labelspacing  : 0.5    # the vertical space between the legend entries in␣
 ↪fraction of fontsize
#legend.handlelength  : 2.     # the length of the legend lines in fraction of fontsize
#legend.handleheight  : 0.7     # the height of the legend handle in fraction of fontsize
#legend.handletextpad : 0.8    # the space between the legend line and legend text in␣
 ↪fraction of fontsize
#legend.borderaxespad : 0.5   # the border between the axes and legend edge in fraction␣
 ↪of fontsize
#legend.columnspacing : 2.    # the border between the axes and legend edge in fraction␣
 ↪of fontsize
#legend.shadow        : False
#legend.frameon       : True   # whether or not to draw a frame around legend
#legend.framealpha    : None    # opacity of of legend frame
#legend.scatterpoints : 3 # number of scatter points
```

```
### FIGURE
# See http://matplotlib.org/api/figure_api.html#matplotlib.figure.Figure
#figure.titlesize : medium     # size of the figure title
#figure.titleweight : normal   # weight of the figure title
#figure.figsize   : 8, 6    # figure size in inches
#figure.dpi       : 80      # figure dots per inch
#figure.facecolor : 0.75    # figure facecolor; 0.75 is scalar gray
#figure.edgecolor : white   # figure edgecolor
#figure.autolayout : False  # When True, automatically adjust subplot
                            # parameters to make the plot fit the figure
#figure.max_open_warning : 20  # The maximum number of figures to open through
                               # the pyplot interface before emitting a warning.
                               # If less than one this feature is disabled.

# The figure subplot parameters.  All dimensions are a fraction of the
# figure width or height
#figure.subplot.left    : 0.125  # the left side of the subplots of the figure
#figure.subplot.right   : 0.9    # the right side of the subplots of the figure
#figure.subplot.bottom  : 0.1    # the bottom of the subplots of the figure
#figure.subplot.top     : 0.9    # the top of the subplots of the figure
#figure.subplot.wspace  : 0.2    # the amount of width reserved for blank space between␣
 ↪subplots
#figure.subplot.hspace  : 0.2    # the amount of height reserved for white space between␣
 ↪subplots

### IMAGES
#image.aspect : equal             # equal | auto | a number
#image.interpolation  : bilinear  # see help(imshow) for options
#image.cmap   : jet               # gray | jet etc...
#image.lut    : 256               # the size of the colormap lookup table
#image.origin : upper             # lower | upper
#image.resample  : False
#image.composite_image : True     # When True, all the images on a set of axes are
                                  # combined into a single composite image before
                                  # saving a figure as a vector graphics file,
                                  # such as a PDF.

### CONTOUR PLOTS
#contour.negative_linestyle : dashed # dashed | solid
#contour.corner_mask        : True   # True | False | legacy

### ERRORBAR PLOTS
#errorbar.capsize : 3             # length of end cap on error bars in pixels

### Agg rendering
### Warning: experimental, 2008/10/10
#agg.path.chunksize : 0           # 0 to disable; values in the range
                                  # 10000 to 100000 can improve speed slightly
                                  # and prevent an Agg rendering failure
                                  # when plotting very large data sets,
                                  # especially if they are very gappy.
                                  # It may cause minor artifacts, though.
                                  # A value of 20000 is probably a good
```

```
                                  # starting point.
### SAVING FIGURES
#path.simplify : True   # When True, simplify paths by removing "invisible"
                        # points to reduce file size and increase rendering
                        # speed
#path.simplify_threshold : 0.1  # The threshold of similarity below which
                                # vertices will be removed in the simplification
                                # process
#path.snap : True # When True, rectilinear axis-aligned paths will be snapped to
                  # the nearest pixel when certain criteria are met.  When False,
                  # paths will never be snapped.
#path.sketch : None # May be none, or a 3-tuple of the form (scale, length,
                    # randomness).
                    # *scale* is the amplitude of the wiggle
                    # perpendicular to the line (in pixels).  *length*
                    # is the length of the wiggle along the line (in
                    # pixels).  *randomness* is the factor by which
                    # the length is randomly scaled.

# the default savefig params can be different from the display params
# e.g., you may want a higher resolution, or to make the figure
# background white
#savefig.dpi         : 100      # figure dots per inch
#savefig.facecolor   : white    # figure facecolor when saving
#savefig.edgecolor   : white    # figure edgecolor when saving
#savefig.format      : png      # png, ps, pdf, svg
#savefig.bbox        : standard # 'tight' or 'standard'.
                                # 'tight' is incompatible with pipe-based animation
                                # backends but will workd with temporary file based ones:
                                # e.g. setting animation.writer to ffmpeg will not work,
                                # use ffmpeg_file instead
#savefig.pad_inches  : 0.1      # Padding to be used when bbox is set to 'tight'
#savefig.jpeg_quality: 95       # when a jpeg is saved, the default quality parameter.
#savefig.directory   : ~        # default directory in savefig dialog box,
                                # leave empty to always use current working directory
#savefig.transparent : False    # setting that controls whether figures are saved with a
                                # transparent background by default

# tk backend params
#tk.window_focus   : False    # Maintain shell focus for TkAgg

# ps backend params
#ps.papersize      : letter   # auto, letter, legal, ledger, A0-A10, B0-B10
#ps.useafm         : False    # use of afm fonts, results in small files
#ps.usedistiller   : False    # can be: None, ghostscript or xpdf
                                          # Experimental: may produce smaller files.
                                          # xpdf intended for production of publication␣
 ↪quality files,
                                          # but requires ghostscript, xpdf and ps2eps
#ps.distiller.res  : 6000      # dpi
#ps.fonttype       : 3         # Output Type 3 (Type3) or Type 42 (TrueType)

# pdf backend params
```

```
#pdf.compression   : 6 # integer from 0 to 9
                       # 0 disables compression (good for debugging)
#pdf.fonttype       : 3         # Output Type 3 (Type3) or Type 42 (TrueType)

# svg backend params
#svg.image_inline : True       # write raster image data directly into the svg file
#svg.image_noscale : False     # suppress scaling of raster data embedded in SVG
#svg.fonttype : 'path'         # How to handle SVG fonts:
#    'none': Assume fonts are installed on the machine where the SVG will be viewed.
#    'path': Embed characters as paths -- supported by most SVG renderers
#    'svgfont': Embed characters as SVG fonts -- supported only by Chrome,
#               Opera and Safari

# docstring params
#docstring.hardcopy = False  # set this when you want to generate hardcopy docstring

# Set the verbose flags.  This controls how much information
# matplotlib gives you at runtime and where it goes.  The verbosity
# levels are: silent, helpful, debug, debug-annoying.  Any level is
# inclusive of all the levels below it.  If your setting is "debug",
# you'll get all the debug and helpful messages.  When submitting
# problems to the mailing-list, please set verbose to "helpful" or "debug"
# and paste the output into your report.
#
# The "fileo" gives the destination for any calls to verbose.report.
# These objects can a filename, or a filehandle like sys.stdout.
#
# You can override the rc default verbosity from the command line by
# giving the flags --verbose-LEVEL where LEVEL is one of the legal
# levels, e.g., --verbose-helpful.
#
# You can access the verbose instance in your code
#   from matplotlib import verbose.
#verbose.level  : silent      # one of silent, helpful, debug, debug-annoying
#verbose.fileo  : sys.stdout  # a log filename, sys.stdout or sys.stderr

# Event keys to interact with figures/plots via keyboard.
# Customize these settings according to your needs.
# Leave the field(s) empty if you don't need a key-map. (i.e., fullscreen : '')

#keymap.fullscreen : f               # toggling
#keymap.home : h, r, home            # home or reset mnemonic
#keymap.back : left, c, backspace    # forward / backward keys to enable
#keymap.forward : right, v           #   left handed quick navigation
#keymap.pan : p                      # pan mnemonic
#keymap.zoom : o                     # zoom mnemonic
#keymap.save : s                     # saving current figure
#keymap.quit : ctrl+w, cmd+w         # close the current figure
#keymap.grid : g                     # switching on/off a grid in current axes
#keymap.yscale : l                   # toggle scaling of y-axes ('log'/'linear')
#keymap.xscale : L, k                # toggle scaling of x-axes ('log'/'linear')
#keymap.all_axes : a                 # enable all axes
```

```
# Control location of examples data files
#examples.directory : ''   # directory to look in for custom installation

###ANIMATION settings
#animation.html : 'none'          # How to display the animation as HTML in
                                   # the IPython notebook. 'html5' uses
                                   # HTML5 video tag.
#animation.writer : ffmpeg         # MovieWriter 'backend' to use
#animation.codec : mpeg4           # Codec to use for writing movie
#animation.bitrate: -1             # Controls size/quality tradeoff for movie.
                                   # -1 implies let utility auto-determine
#animation.frame_format: 'png'    # Controls frame format used by temp files
#animation.ffmpeg_path: 'ffmpeg'  # Path to ffmpeg binary. Without full path
                                   # $PATH is searched
#animation.ffmpeg_args: ''        # Additional arguments to pass to ffmpeg
#animation.avconv_path: 'avconv'  # Path to avconv binary. Without full path
                                   # $PATH is searched
#animation.avconv_args: ''        # Additional arguments to pass to avconv
#animation.mencoder_path: 'mencoder'
                                   # Path to mencoder binary. Without full path
                                   # $PATH is searched
#animation.mencoder_args: ''      # Additional arguments to pass to mencoder
#animation.convert_path: 'convert' # Path to ImageMagick's convert binary.
                                   # On Windows use the full path since convert
                                   # is also the name of a system tool.
```

2.3 Using matplotlib in a python shell

By default, matplotlib defers drawing until the end of the script because drawing can be an expensive operation, and you may not want to update the plot every time a single property is changed, only once after all the properties have changed.

But when working from the python shell, you usually do want to update the plot with every command, e.g., after changing the `xlabel()`, or the marker style of a line. While this is simple in concept, in practice it can be tricky, because matplotlib is a graphical user interface application under the hood, and there are some tricks to make the applications work right in a python shell.

2.3.1 IPython to the rescue

Note: The mode described here still exists for historical reasons, but it is highly advised not to use. It pollutes namespaces with functions that will shadow python built-in and can lead to hard to track bugs. To get IPython integration without imports the use of the `%matplotlib` magic is preferred. See ipython documentation .

Fortunately, ipython, an enhanced interactive python shell, has figured out all of these tricks, and is matplotlib aware, so when you start ipython in the *pylab* mode.

```
johnh@flag:~> ipython
Python 2.4.5 (#4, Apr 12 2008, 09:09:16)
IPython 0.9.0 -- An enhanced Interactive Python.

In [1]: %pylab

  Welcome to pylab, a matplotlib-based Python environment.
  For more information, type 'help(pylab)'.

In [2]: x = randn(10000)

In [3]: hist(x, 100)
```

it sets everything up for you so interactive plotting works as you would expect it to. Call `figure()` and a figure window pops up, call `plot()` and your data appears in the figure window.

Note in the example above that we did not import any matplotlib names because in pylab mode, ipython will import them automatically. ipython also turns on *interactive* mode for you, which causes every pyplot command to trigger a figure update, and also provides a matplotlib aware `run` command to run matplotlib scripts efficiently. ipython will turn off interactive mode during a `run` command, and then restore the interactive state at the end of the run so you can continue tweaking the figure manually.

There has been a lot of recent work to embed ipython, with pylab support, into various GUI applications, so check on the ipython mailing list for the latest status.

2.3.2 Other python interpreters

If you can't use ipython, and still want to use matplotlib/pylab from an interactive python shell, e.g., the plain-ole standard python interactive interpreter, you are going to need to understand what a matplotlib backend is *What is a backend?*.

With the TkAgg backend, which uses the Tkinter user interface toolkit, you can use matplotlib from an arbitrary non-gui python shell. Just set your `backend : TkAgg` and `interactive : True` in your `matplotlibrc` file (see *Customizing matplotlib*) and fire up python. Then:

```
>>> from pylab import *
>>> plot([1,2,3])
>>> xlabel('hi mom')
```

should work out of the box. This is also likely to work with recent versions of the qt4agg and gtkagg backends, and with the macosx backend on the Macintosh. Note, in batch mode, i.e. when making figures from scripts, interactive mode can be slow since it redraws the figure with each command. So you may want to think carefully before making this the default behavior via the `matplotlibrc` file instead of using the functions listed in the next section.

Gui shells are at best problematic, because they have to run a mainloop, but interactive plotting also involves a mainloop. Ipython has sorted all this out for the primary matplotlib backends. There may be other shells and IDEs that also work with matplotlib in interactive mode, but one obvious candidate does not: the python IDLE IDE is a Tkinter gui app that does not support pylab interactive mode, regardless of backend.

2.3.3 Controlling interactive updating

The *interactive* property of the pyplot interface controls whether a figure canvas is drawn on every pyplot command. If *interactive* is *False*, then the figure state is updated on every plot command, but will only be drawn on explicit calls to `draw()`. When *interactive* is *True*, then every pyplot command triggers a draw.

The pyplot interface provides 4 commands that are useful for interactive control.

`isinteractive()` returns the interactive setting *True|False*

`ion()` turns interactive mode on

`ioff()` turns interactive mode off

`draw()` forces a figure redraw

When working with a big figure in which drawing is expensive, you may want to turn matplotlib's interactive setting off temporarily to avoid the performance hit:

```
>>> #create big-expensive-figure
>>> ioff()      # turn updates off
>>> title('now how much would you pay?')
>>> xticklabels(fontsize=20, color='green')
>>> draw()      # force a draw
>>> savefig('alldone', dpi=300)
>>> close()
>>> ion()      # turn updating back on
>>> plot(rand(20), mfc='g', mec='r', ms=40, mew=4, ls='--', lw=3)
```

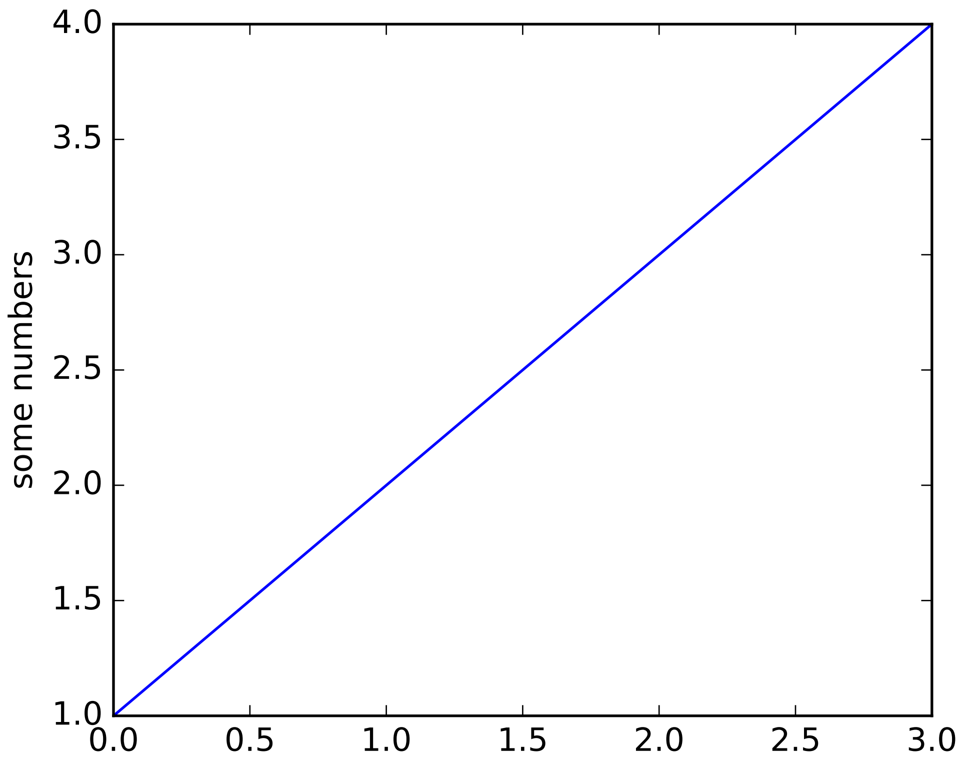
CHAPTER THREE

BEGINNER'S GUIDE

3.1 Pyplot tutorial

matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. In *matplotlib.pyplot* various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that "axes" here and in most places in the documentation refers to the *axes* part of a figure and not the strict mathematical term for more than one axis).

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```



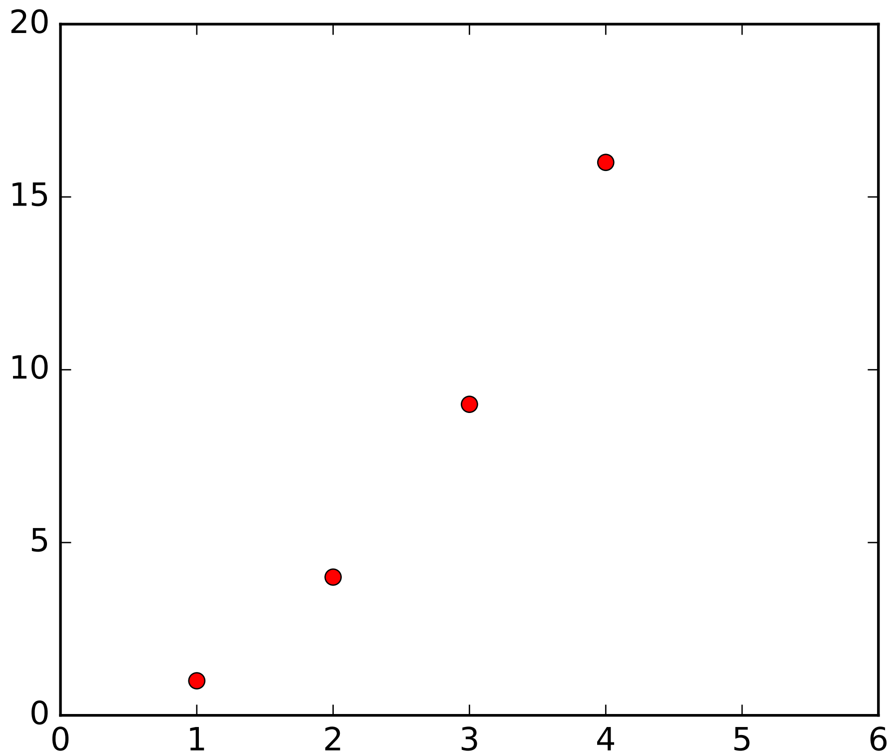
You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the *plot()* command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are `[0,1,2,3]`.

plot() is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue

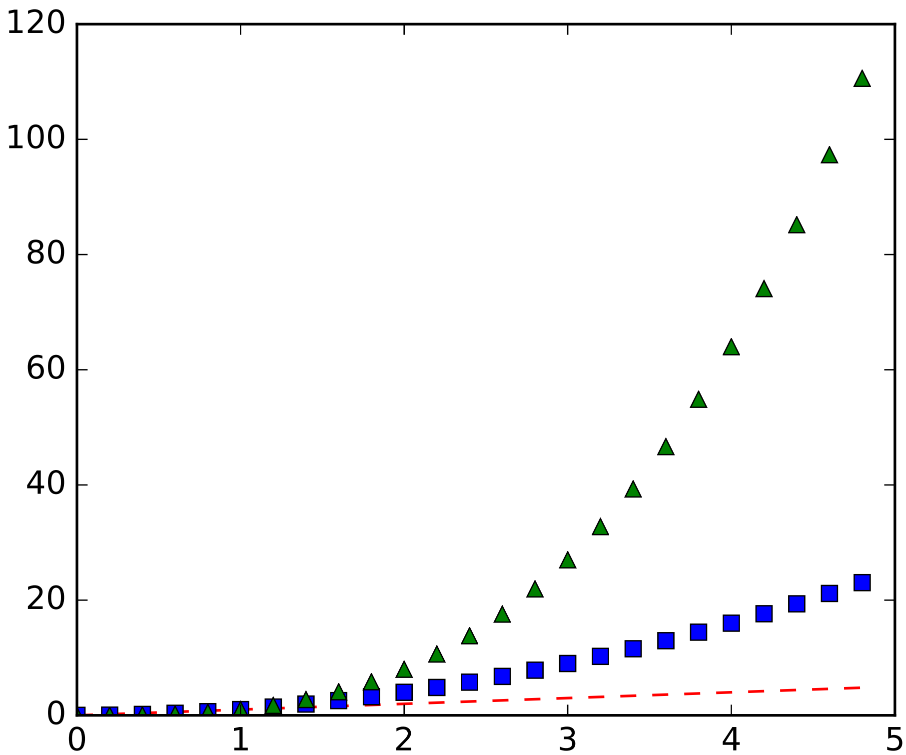
```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```



See the *plot()* documentation for a complete list of line styles and format strings. The *axis()* command in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the axes.

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates a plotting several lines with different format styles in one command using arrays.

```
import numpy as np
import matplotlib.pyplot as plt

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



3.1.1 Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see *matplotlib.lines.Line2D*. There are several ways to set line properties

- Use keyword args:

```
plt.plot(x, y, linewidth=2.0)
```

- Use the setter methods of a `Line2D` instance. `plot` returns a list of `Line2D` objects; e.g., `line1, line2 = plot(x1, y1, x2, y2)`. In the code below we will suppose that we have only one line so that the list returned is of length 1. We use tuple unpacking with `line,` to get the first element of that list:

```
line, = plt.plot(x, y, '-')
line.set_antialiased(False) # turn off antialising
```

- Use the *setp()* command. The example below uses a MATLAB-style command to set multiple properties on a list of lines. `setp` works transparently with a list of objects or a single object. You can either use python keyword arguments or MATLAB-style string/value pairs:

```
lines = plt.plot(x1, y1, x2, y2)
# use keyword args
```

```
plt.setp(lines, color='r', linewidth=2.0)
# or MATLAB style string value pairs
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
```

Here are the available *Line2D* properties.

Property	Value Type
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a Transform instance, a Patch
color or c	any matplotlib color
contains	the hit testing function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
data	(np.array xdata, np.array ydata)
figure	a matplotlib.figure.Figure instance
label	any string
linestyle or ls	['-' | '--' | '-.' | ':' | 'steps' | ...]
linewidth or lw	float value in points
lod	[True | False]
marker	['+' | ',' | '.' | '1' | '2' | '3' | '4']
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markersize or ms	float
markevery	[None | integer | (startind, stride)]
picker	used in interactive line selection
pickradius	the line pick selection radius
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
visible	[True | False]
xdata	np.array
ydata	np.array
zorder	any number

To get a list of settable line properties, call the *setp()* function with a line or lines as argument

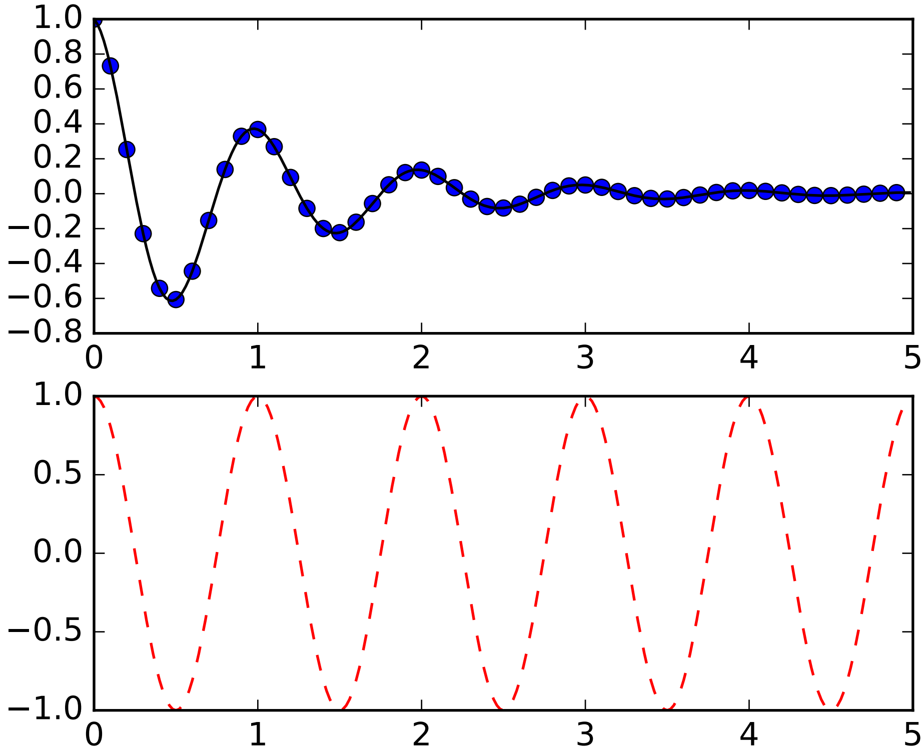
```
In [69]: lines = plt.plot([1, 2, 3])

In [70]: plt.setp(lines)
  alpha: float
  animated: [True | False]
```

```
antialiased or aa: [True | False]
...snip
```

3.1.2 Working with multiple figures and axes

MATLAB, and *pyplot*, have the concept of the current figure and the current axes. All plotting commands apply to the current axes. The function *gca()* returns the current axes (a *matplotlib.axes.Axes* instance), and *gcf()* returns the current figure (*matplotlib.figure.Figure* instance). Normally, you don't have to worry about this, because it is all taken care of behind the scenes. Below is a script to create two subplots.

```python
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```



The *figure()* command here is optional because `figure(1)` will be created by default, just as a `subplot(111)` will be created by default if you don't manually specify any axes. The *subplot()* command specifies `numrows, numcols, fignum` where `fignum` ranges from 1 to `numrows*numcols`. The commas in the `subplot` command are optional if `numrows*numcols<10`. So `subplot(211)` is identical to `subplot(2, 1, 1)`. You can create an arbitrary number of subplots and axes. If you want to place an axes manually, i.e., not on a rectangular grid, use the *axes()* command, which allows you to specify the location as `axes([left, bottom, width, height])` where all values are in fractional (0 to 1) coordinates. See *pylab_examples example code: axes_demo.py* for an example of placing axes manually and *pylab_examples example code: subplots_demo.py* for an example with lots of subplots.

You can create multiple figures by using multiple *figure()* calls with an increasing figure number. Of course, each figure can contain as many axes and subplots as your heart desires:

```
import matplotlib.pyplot as plt
plt.figure(1)                # the first figure
plt.subplot(211)             # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212)             # the second subplot in the first figure
plt.plot([4, 5, 6])


plt.figure(2)                # a second figure
plt.plot([4, 5, 6])          # creates a subplot(111) by default

plt.figure(1)                # figure 1 current; subplot(212) still current
```

```
plt.subplot(211)             # make subplot(211) in figure1 current
plt.title('Easy as 1, 2, 3') # subplot 211 title
```

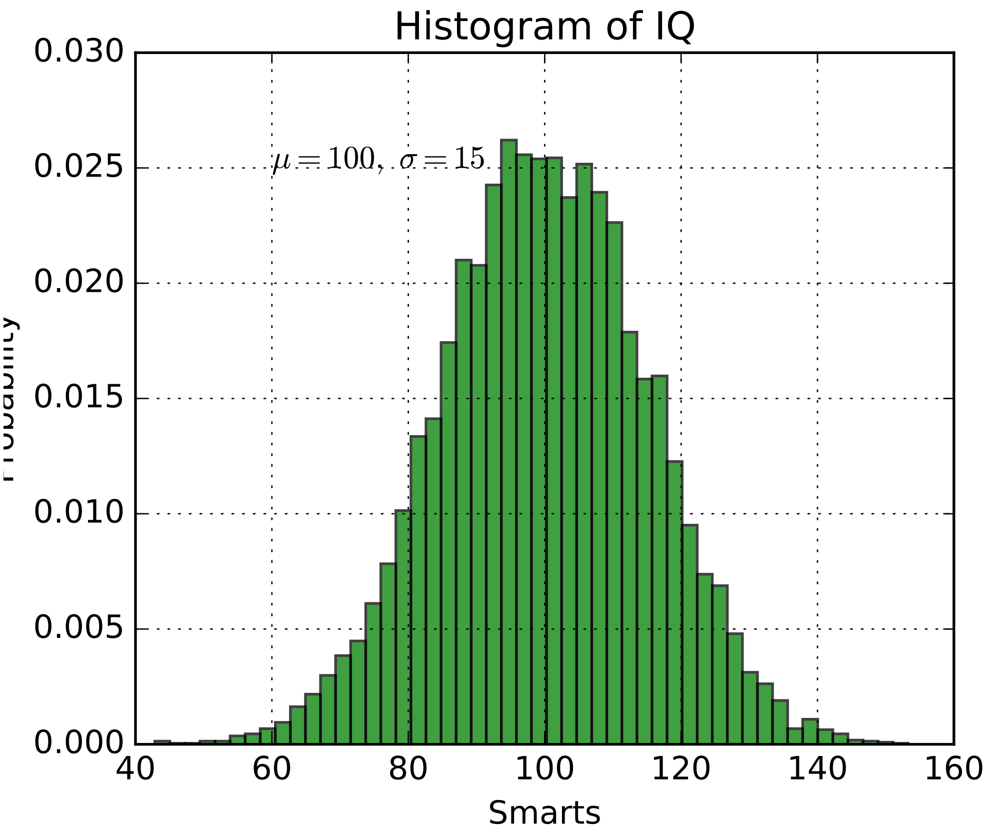
You can clear the current figure with `clf()` and the current axes with `cla()`. If you find it annoying that states (specifically the current image, figure and axes) are being maintained for you behind the scenes, don't despair: this is just a thin stateful wrapper around an object oriented API, which you can use instead (see *Artist tutorial*)

If you are making lots of figures, you need to be aware of one more thing: the memory required for a figure is not completely released until the figure is explicitly closed with `close()`. Deleting all references to the figure, and/or using the window manager to kill the window in which the figure appears on the screen, is not enough, because pyplot maintains internal references until `close()` is called.

3.1.3 Working with text

The `text()` command can be used to add text in an arbitrary location, and the `xlabel()`, `ylabel()` and `title()` are used to add text in the indicated locations (see *Text introduction* for a more detailed example)

```
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)


plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```



All of the `text()` commands return an `matplotlib.text.Text` instance. Just as with with lines above, you can customize the properties by passing keyword arguments into the text functions or using `setp()`:

```
t = plt.xlabel('my data', fontsize=14, color='red')
```

These properties are covered in more detail in *Text properties and layout*.

Using mathematical expressions in text

matplotlib accepts TeX equation expressions in any text expression. For example to write the expression $\sigma_i = 15$ in the title, you can write a TeX expression surrounded by dollar signs:

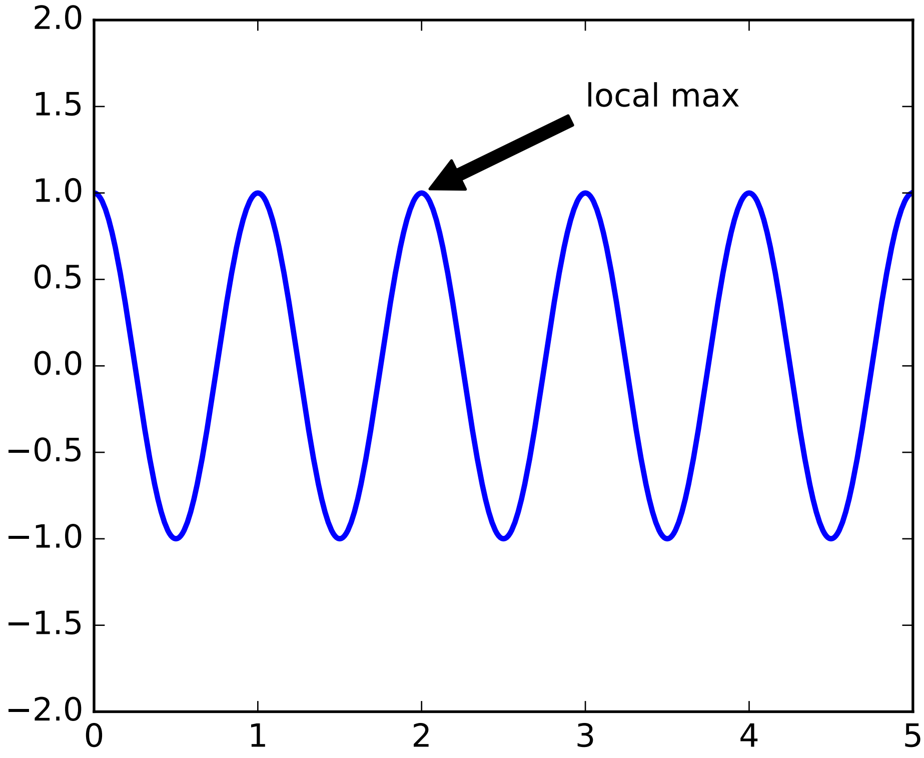
```
plt.title(r'$\sigma_i=15$')
```

The `r` preceding the title string is important – it signifies that the string is a *raw* string and not to treat backslashes as python escapes. matplotlib has a built-in TeX expression parser and layout engine, and ships its own math fonts – for details see *Writing mathematical expressions*. Thus you can use mathematical text across platforms without requiring a TeX installation. For those who have LaTeX and dvipng installed, you can also use LaTeX to format your text and incorporate the output directly into your display figures or saved postscript – see *Text rendering With LaTeX*.

Annotating text

The uses of the basic `text()` command above place text at an arbitrary position on the Axes. A common use for text is to annotate some feature of the plot, and the `annotate()` method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x,y)` tuples.

```
import numpy as np
import matplotlib.pyplot as plt

ax = plt.subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05),
            )

plt.ylim(-2,2)
plt.show()
```



In this basic example, both the `xy` (arrow tip) and `xytext` locations (text location) are in data coordinates.

There are a variety of other coordinate systems one can choose – see *Annotating text* and *Annotating Axes* for details. More examples can be found in *pylab_examples example code: annotation_demo.py*.

3.1.4 Logarithmic and other nonlinear axis

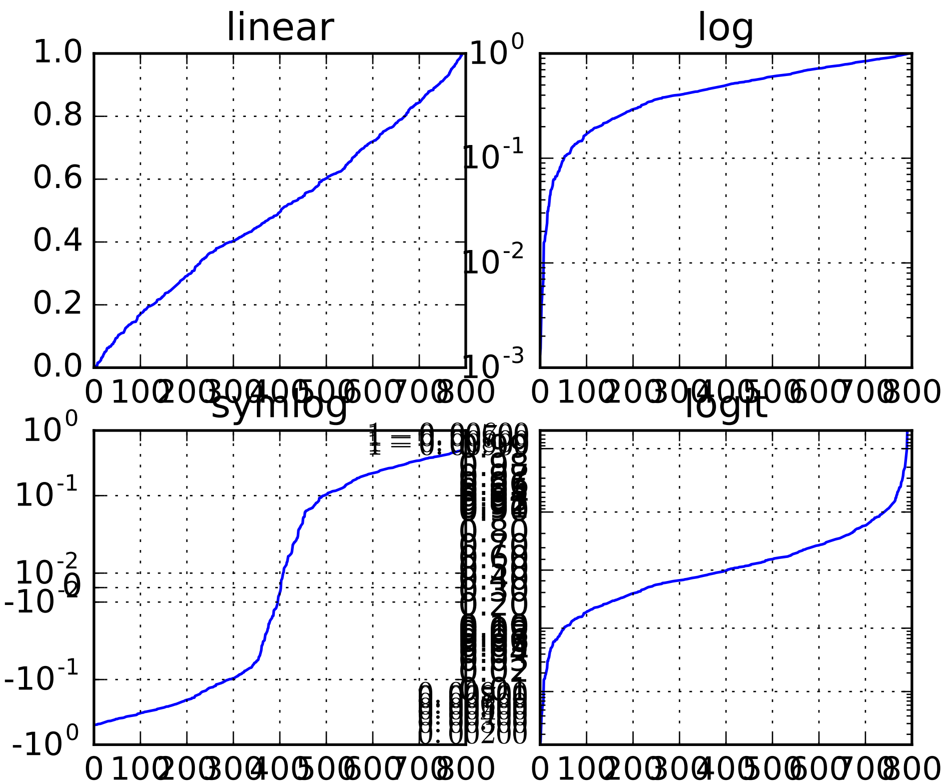
matplotlib.pyplot supports not only linear axis scales, but also logarithmic and logit scales. This is commonly used if data spans many orders of magnitude. Changing the scale of an axis is easy:

> plt.xscale('log')

An example of four plots with the same data and different scales for the y axis is shown below.

```
import numpy as np
import matplotlib.pyplot as plt

# make up some data in the interval ]0, 1[
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
plt.figure(1)

# linear
plt.subplot(221)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)


# log
plt.subplot(222)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)


# symmetric log
plt.subplot(223)
plt.plot(x, y - y.mean())
plt.yscale('symlog', linthreshy=0.05)
plt.title('symlog')
plt.grid(True)

# logit
plt.subplot(224)
plt.plot(x, y)
plt.yscale('logit')
plt.title('logit')
plt.grid(True)
```

```
plt.show()
```



It is also possible to add your own scale, see *Adding new scales and projections to matplotlib* for details.

3.2 Customizing plots with style sheets

The `style` package adds support for easy-to-switch plotting "styles" with the same parameters as a matplotlibrc file.

There are a number of pre-defined styles provided by matplotlib. For example, there's a pre-defined style called "ggplot", which emulates the aesthetics of ggplot (a popular plotting package for R). To use this style, just add:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use('ggplot')
```

To list all available styles, use:

```
>>> print(plt.style.available)
```

3.2.1 Defining your own style

You can create custom styles and use them by calling `style.use` with the path or URL to the style sheet. Alternatively, if you add your `<style-name>.mplstyle` file to `mpl_configdir/stylelib`, you can reuse your custom style sheet with a call to `style.use(<style-name>)`. By default `mpl_configdir` should be `~/.config/matplotlib`, but you can check where yours is with `matplotlib.get_configdir()`, you may need to create this directory. Note that a custom style sheet in `mpl_configdir/stylelib` will override a style sheet defined by matplotlib if the styles have the same name.

For example, you might want to create `mpl_configdir/stylelib/presentation.mplstyle` with the following:

```
axes.titlesize : 24
axes.labelsize : 20
lines.linewidth : 3
lines.markersize : 10
xtick.labelsize : 16
ytick.labelsize : 16
```

Then, when you want to adapt a plot designed for a paper to one that looks good in a presentation, you can just add:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use('presentation')
```

3.2.2 Composing styles

Style sheets are designed to be composed together. So you can have a style sheet that customizes colors and a separate style sheet that alters element sizes for presentations. These styles can easily be combined by passing a list of styles:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use(['dark_background', 'presentation'])
```

Note that styles further to the right will overwrite values that are already defined by styles on the left.

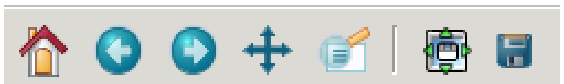
3.2.3 Temporary styling

If you only want to use a style for a specific block of code but don't want to change the global styling, the style package provides a context manager for limiting your changes to a specific scope. To isolate the your styling changes, you can write something like the following:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>>
>>> with plt.style.context(('dark_background')):
>>>     plt.plot(np.sin(np.linspace(0, 2 * np.pi)), 'r-o')
>>>
>>> # Some plotting code with the default style
```

```
>>>
>>> plt.show()
```

3.3 Interactive navigation



All figure windows come with a navigation toolbar, which can be used to navigate through the data set. Here is a description of each of the buttons at the bottom of the toolbar

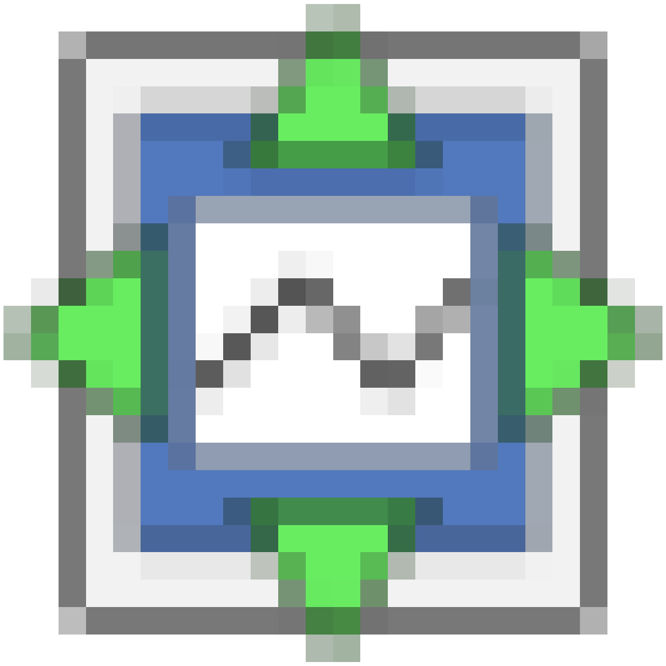
The `Forward` and `Back` buttons These are akin to the web browser forward and back buttons. They are used to navigate back and forth between previously defined views. They have no meaning unless you have already navigated somewhere else using the pan and zoom buttons. This is analogous to trying to click `Back` on your web browser before visiting a new page –nothing happens. `Home` always takes you to the first, default view of your data. For `Home`, `Forward` and `Back`, think web browser where data views are web pages. Use the pan and zoom to rectangle to define new views.

The `Pan/Zoom` button This button has two modes: pan and zoom. Click the toolbar button to activate panning and zooming, then put your mouse somewhere over an axes. Press the left mouse button and hold it to pan the figure, dragging it to a new position. When you release it, the data under the point where you pressed will be moved to the point where you released. If you press 'x' or 'y' while panning the motion will be constrained to the x or y axis, respectively. Press the right mouse button to zoom, dragging it to a new position. The x axis will be zoomed in proportionate to the rightward movement and zoomed out proportionate to the leftward movement. Ditto for the y axis and up/down motions. The point under your mouse when you begin the zoom remains stationary, allowing you to zoom to an arbitrary point in the figure. You can use the modifier keys 'x', 'y' or 'CONTROL' to constrain the zoom to the x axis, the y axis, or aspect ratio preserve, respectively.

With polar plots, the pan and zoom functionality behaves differently. The radius axis labels can be dragged using the left mouse button. The radius scale can be zoomed in and out using the right mouse button.

The `Zoom-to-rectangle` button Click this toolbar button to activate this mode. Put your mouse somewhere over and axes and press the left mouse button. Drag the mouse while holding the button to a new location and release. The axes view limits will be zoomed to the rectangle you have defined.

There is also an experimental 'zoom out to rectangle' in this mode with the right button, which will place your entire axes in the region defined by the zoom out rectangle.



The `Subplot-configuration` button Use this tool to configure the parameters of the subplot: the left, right, top, bottom, space between the rows and space between the columns.

The Save button Click this button to launch a file save dialog. You can save files with the following extensions: `png`, `ps`, `eps`, `svg` and `pdf`.

3.3.1 Navigation Keyboard Shortcuts

The following table holds all the default keys, which can be overwritten by use of your matplotlibrc (#keymap.*).

Command	Keyboard Shortcut(s)
Home/Reset	**h** or **r** or **home**
Back	**c** or **left arrow** or **backspace**
Forward	**v** or **right arrow**
Pan/Zoom	**p**
Zoom-to-rect	**o**
Save	**ctrl** + **s**
Toggle fullscreen	**ctrl** + **f**
Close plot	**ctrl** + **w**
Constrain pan/zoom to x axis	hold **x** when panning/zooming with mouse
Constrain pan/zoom to y axis	hold **y** when panning/zooming with mouse
Preserve aspect ratio	hold **CONTROL** when panning/zooming with mouse
Toggle grid	**g** when mouse is over an axes
Toggle x axis scale (log/linear)	**L** or **k** when mouse is over an axes
Toggle y axis scale (log/linear)	**l** when mouse is over an axes

If you are using *matplotlib.pyplot* the toolbar will be created automatically for every figure. If you are writing your own user interface code, you can add the toolbar as a widget. The exact syntax depends on your UI, but we have examples for every supported UI in the `matplotlib/examples/user_interfaces` directory. Here is some example code for GTK:

```
import gtk

from matplotlib.figure import Figure
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as␣
↪NavigationToolbar

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400,300)
win.set_title("Embedding in GTK")

vbox = gtk.VBox()
win.add(vbox)

fig = Figure(figsize=(5,4), dpi=100)
ax = fig.add_subplot(111)
ax.plot([1,2,3])

canvas = FigureCanvas(fig)  # a gtk.DrawingArea
vbox.pack_start(canvas)
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False)

win.show_all()
gtk.main()
```

3.4 Working with text

3.4.1 Text introduction

matplotlib has excellent text support, including mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support. Because we embed the fonts directly in the output documents, e.g., for postscript or PDF, what you see on the screen is what you get in the hardcopy. freetype2 support produces very nice, antialiased fonts, that look good even at small raster sizes. matplotlib includes its own *matplotlib.font_manager*, thanks to Paul Barrett, which implements a cross platform, W3C compliant font finding algorithm.

You have total control over every text property (font size, font weight, text location and color, etc) with sensible defaults set in the rc file. And significantly for those interested in mathematical or scientific figures, matplotlib implements a large number of TeX math symbols and commands, to support *mathematical expressions* anywhere in your figure.

3.4.2 Basic text commands

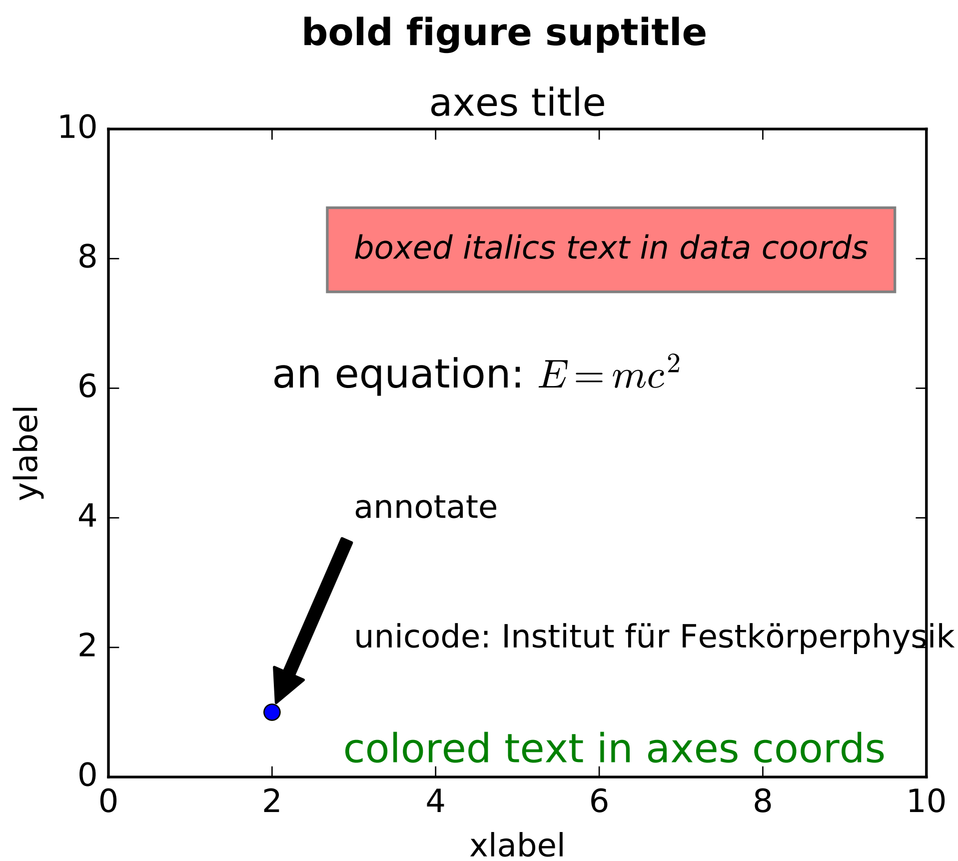
The following commands are used to create text in the pyplot interface

- *text()* - add text at an arbitrary location to the `Axes`; *matplotlib.axes.Axes.text()* in the API.
- *xlabel()* - add an axis label to the x-axis; *matplotlib.axes.Axes.set_xlabel()* in the API.
- *ylabel()* - add an axis label to the y-axis; *matplotlib.axes.Axes.set_ylabel()* in the API.
- *title()* - add a title to the `Axes`; *matplotlib.axes.Axes.set_title()* in the API.
- *figtext()* - add text at an arbitrary location to the `Figure`; *matplotlib.figure.Figure.text()* in the API.
- *suptitle()* - add a title to the `Figure`; *matplotlib.figure.Figure.suptitle()* in the API.
- ***annotate()* - add an annotation, with** optional arrow, to the `Axes` ; *matplotlib.axes.Axes.annotate()* in the API.

All of these functions create and return a *matplotlib.text.Text()* instance, which can be configured with a variety of font and other properties. The example below shows all of these commands in action.

```
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt

fig = plt.figure()
fig.suptitle('bold figure suptitle', fontsize=14, fontweight='bold')

ax = fig.add_subplot(111)
fig.subplots_adjust(top=0.85)
ax.set_title('axes title')

ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')
```

```
ax.text(3, 8, 'boxed italics text in data coords', style='italic',
        bbox={'facecolor':'red', 'alpha':0.5, 'pad':10})

ax.text(2, 6, r'an equation: $E=mc^2$', fontsize=15)

ax.text(3, 2, u'unicode: Institut f\374r Festk\366rperphysik')

ax.text(0.95, 0.01, 'colored text in axes coords',
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='green', fontsize=15)


ax.plot([2], [1], 'o')
ax.annotate('annotate', xy=(2, 1), xytext=(3, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.axis([0, 10, 0, 10])

plt.show()
```



3.4.3 Text properties and layout

The *matplotlib.text.Text* instances have a variety of properties which can be configured via keyword arguments to the text commands (e.g., *title()*, *xlabel()* and *text()*).

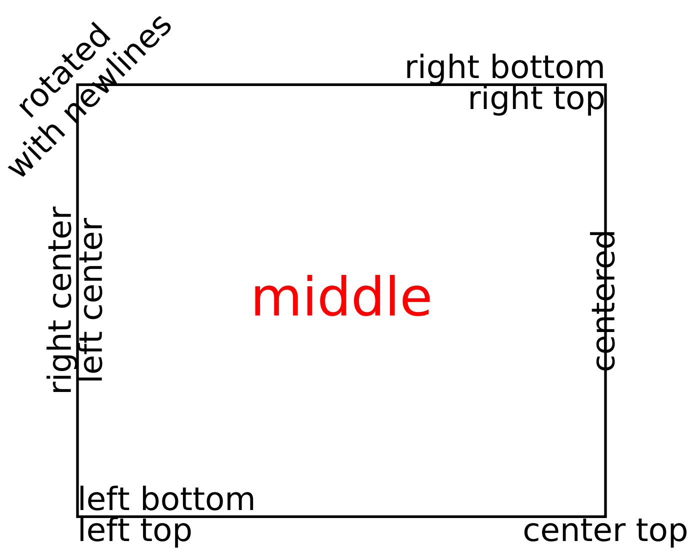
Property	Value Type
alpha	float
backgroundcolor	any matplotlib color
bbox	rectangle prop dict plus key 'pad' which is a pad in points
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a Transform instance, a Patch
color	any matplotlib color
family	['serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
fontproperties	a matplotlib.font_manager.FontProperties instance
horizontalalignment or ha	['center' | 'right' | 'left']
label	any string
linespacing	float
multialignment	['left' | 'right' | 'center']
name or fontname	string e.g., ['Sans' | 'Courier' | 'Helvetica' ...]
picker	[None|float|boolean|callable]
position	(x,y)
rotation	[angle in degrees 'vertical' | 'horizontal'
size or fontsize	[size in points | relative size, e.g., 'smaller', 'x-large']
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion
transform	a matplotlib.transform transformation instance
variant	['normal' | 'small-caps']
verticalalignment or va	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	['normal' | 'bold' | 'heavy' | 'light' | 'ultrabold' | 'ultralight']
x	float
y	float
zorder	any number

You can layout text with the alignment arguments `horizontalalignment`, `verticalalignment`, and `multialignment`. `horizontalalignment` controls whether the x positional argument for the text indicates the left, center or right side of the text bounding box. `verticalalignment` controls whether the y positional argument for the text indicates the bottom, center or top side of the text bounding box. `multialignment`, for newline separated strings only, controls whether the different lines are left, center or right justified. Here is an example which uses the *text()* command to show the various alignment possibilities. The use of `transform=ax.transAxes` throughout the code indicates that the coordinates are given relative to the axes bounding box, with 0,0 being the lower left of the axes and 1,1 the upper right.

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
```

```
# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
top = bottom + height

fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

# axes coordinates are 0,0 is bottom left and 1,1 is upper right
p = patches.Rectangle(
    (left, bottom), width, height,
    fill=False, transform=ax.transAxes, clip_on=False
    )

ax.add_patch(p)

ax.text(left, bottom, 'left top',
        horizontalalignment='left',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
        horizontalalignment='left',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right bottom',
        horizontalalignment='right',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right top',
        horizontalalignment='right',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(right, bottom, 'center top',
        horizontalalignment='center',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'right center',
        horizontalalignment='right',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom+top), 'left center',
        horizontalalignment='left',
        verticalalignment='center',
        rotation='vertical',
```

```
        transform=ax.transAxes)

ax.text(0.5*(left+right), 0.5*(bottom+top), 'middle',
        horizontalalignment='center',
        verticalalignment='center',
        fontsize=20, color='red',
        transform=ax.transAxes)

ax.text(right, 0.5*(bottom+top), 'centered',
        horizontalalignment='center',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, top, 'rotated\nwith newlines',
        horizontalalignment='center',
        verticalalignment='center',
        rotation=45,
        transform=ax.transAxes)

ax.set_axis_off()
plt.show()
```



3.4.4 Writing mathematical expressions

You can use a subset TeX markup in any matplotlib text string by placing it inside a pair of dollar signs ($).

Note that you do not need to have TeX installed, since matplotlib ships its own TeX expression parser, layout engine and fonts. The layout engine is a fairly direct adaptation of the layout algorithms in Donald Knuth's TeX, so the quality is quite good (matplotlib also provides a `usetex` option for those who do want to call out to TeX to generate their text (see *Text rendering With LaTeX*).

Any text element can use math text. You should use raw strings (precede the quotes with an `'r'`), and surround the math text with dollar signs ($), as in TeX. Regular text and mathtext can be interleaved within the same string. Mathtext can use the Computer Modern fonts (from (La)TeX), STIX fonts (with are designed to blend well with Times) or a Unicode font that you provide. The mathtext font can be selected with the customization variable `mathtext.fontset` (see *Customizing matplotlib*)

Note: On "narrow" builds of Python, if you use the STIX fonts you should also set `ps.fonttype` and `pdf.fonttype` to 3 (the default), not 42. Otherwise some characters will not be visible.

Here is a simple example:

```
# plain text
plt.title('alpha > beta')
```

produces "alpha > beta".

Whereas this:

```
# math text
plt.title(r'$\alpha > \beta$')
```

produces "$\alpha > \beta$".

Note: Mathtext should be placed between a pair of dollar signs ($). To make it easy to display monetary values, e.g., "$100.00", if a single dollar sign is present in the entire string, it will be displayed verbatim as a dollar sign. This is a small change from regular TeX, where the dollar sign in non-math text would have to be escaped ('$').

Note: While the syntax inside the pair of dollar signs ($) aims to be TeX-like, the text outside does not. In particular, characters such as:

```
# $ % & ~ _ ^ \ { } \( \) \[ \]
```

have special meaning outside of math mode in TeX. Therefore, these characters will behave differently depending on the rcParam `text.usetex` flag. See the *usetex tutorial* for more information.

Subscripts and superscripts

To make subscripts and superscripts, use the '_' and '^' symbols:

```
r'$\alpha_i > \beta_i$'
```

$$\alpha_i > \beta_i \tag{3.1}$$

Some symbols automatically put their sub/superscripts under and over the operator. For example, to write the sum of x_i from 0 to ∞, you could do:

```
r'$\sum_{i=0}^\infty x_i$'
```

$$\sum_{i=0}^{\infty} x_i \tag{3.2}$$

Fractions, binomials and stacked numbers

Fractions, binomials and stacked numbers can be created with the \frac{}{}, \binom{}{} and \stackrel{}{} commands, respectively:

```
r'$\frac{3}{4} \binom{3}{4} \stackrel{3}{4}$'
```

produces

$$\frac{3}{4}\binom{3}{4}\stackrel{3}{4} \tag{3.3}$$

Fractions can be arbitrarily nested:

```
r'$\frac{5 - \frac{1}{x}}{4}$'
```

produces

$$\frac{5 - \frac{1}{x}}{4} \tag{3.4}$$

Note that special care needs to be taken to place parentheses and brackets around fractions. Doing things the obvious way produces brackets that are too small:

```
r'$(\frac{5 - \frac{1}{x}}{4})$'
```

$$(\frac{5 - \frac{1}{x}}{4}) \tag{3.5}$$

The solution is to precede the bracket with \left and \right to inform the parser that those brackets encompass the entire object:

```
r'$\left(\frac{5 - \frac{1}{x}}{4}\right)$'
```

$$\left(\frac{5 - \frac{1}{x}}{4}\right) \tag{3.6}$$

Radicals

Radicals can be produced with the `\sqrt[]{}` command. For example:

```
r'$\sqrt{2}$'
```

$$\sqrt{2} \tag{3.7}$$

Any base can (optionally) be provided inside square brackets. Note that the base must be a simple expression, and can not contain layout commands such as fractions or sub/superscripts:

```
r'$\sqrt[3]{x}$'
```

$$\sqrt[3]{x} \tag{3.8}$$

Fonts

The default font is *italics* for mathematical symbols.

Note: This default can be changed using the `mathtext.default` rcParam. This is useful, for example, to use the same font as regular non-math text for math text, by setting it to `regular`.

To change fonts, e.g., to write "sin" in a Roman font, enclose the text in a font command:

```
r'$s(t) = \mathcal{A}\mathrm{sin}(2 \omega t)$'
```

$$s(t) = \mathcal{A}\mathrm{sin}(2\omega t) \tag{3.9}$$

More conveniently, many commonly used function names that are typeset in a Roman font have shortcuts. So the expression above could be written as follows:

```
r'$s(t) = \mathcal{A}\sin(2 \omega t)$'
```

$$s(t) = \mathcal{A}\sin(2\omega t) \tag{3.10}$$

Here "s" and "t" are variable in italics font (default), "sin" is in Roman font, and the amplitude "A" is in calligraphy font. Note in the example above the caligraphy `A` is squished into the `sin`. You can use a spacing command to add a little whitespace between them:

```
s(t) = \mathcal{A}\/\sin(2 \omega t)
```

$$s(t) = \mathcal{A}\/\sin(2\omega t) \tag{3.11}$$

The choices available with all fonts are:

Command	Result
`\mathrm{Roman}`	Roman
`\mathit{Italic}`	Italic
`\mathtt{Typewriter}`	$\mathtt{Typewriter}$
`\mathcal{CALLIGRAPHY}`	$\mathcal{CALLIGRAPHY}$

When using the STIX fonts, you also have the choice of:

Command	Result
`\mathbb{blackboard}`	⋖Ↄ⅂⋊Ↄ⟍
`\mathrm{\mathbb{blackboard}}`	⋖Ↄ⅂⋊Ↄ⟍
`\mathfrak{Fraktur}`	$\mathfrak{Fraktur}$
`\mathsf{sansserif}`	$\mathsf{sansserif}$
`\mathrm{\mathsf{sansserif}}`	$\mathsf{sansserif}$

There are also three global "font sets" to choose from, which are selected using the `mathtext.fontset` parameter in *matplotlibrc*.

`cm`: **Computer Modern (TeX)**

$$\mathcal{R}\prod_{i=\alpha_{i+1}}^{\infty} a_i \sin(2\pi f x_i)$$

`stix`: **STIX** (designed to blend well with Times)

$$\mathcal{R}\prod_{i=\alpha_{i+1}}^{\infty} a_i \sin(2\pi f x_i)$$

`stixsans`: **STIX sans-serif**

$$\mathcal{R}\prod_{i=\alpha_{i+1}}^{\infty} a_i \sin(2\pi f x_i)$$

Additionally, you can use `\mathdefault{...}` or its alias `\mathregular{...}` to use the font used for regular text outside of mathtext. There are a number of limitations to this approach, most notably that far fewer symbols will be available, but it can be useful to make math expressions blend well with other text in the plot.

Custom fonts

mathtext also provides a way to use custom fonts for math. This method is fairly tricky to use, and should be considered an experimental feature for patient users only. By setting the rcParam `mathtext.fontset` to `custom`, you can then set the following parameters, which control which font file to use for a particular set of math characters.

Parameter	Corresponds to
`mathtext.it`	`\mathit{}` or default italic
`mathtext.rm`	`\mathrm{}` Roman (upright)
`mathtext.tt`	`\mathtt{}` Typewriter (monospace)
`mathtext.bf`	`\mathbf{}` bold italic
`mathtext.cal`	`\mathcal{}` calligraphic
`mathtext.sf`	`\mathsf{}` sans-serif

Each parameter should be set to a fontconfig font descriptor (as defined in the yet-to-be-written font chapter).

The fonts used should have a Unicode mapping in order to find any non-Latin characters, such as Greek. If you want to use a math symbol that is not contained in your custom fonts, you can set the rcParam `mathtext.fallback_to_cm` to `True` which will cause the mathtext system to use characters from the default Computer Modern fonts whenever a particular character can not be found in the custom font.

Note that the math glyphs specified in Unicode have evolved over time, and many fonts may not have glyphs in the correct place for mathtext.

Accents

An accent command may precede any symbol to add an accent above it. There are long and short forms for some of them.

Command	Result
`\acute a` or `\'a`	$\acute{a}$
`\bar a`	$\bar{a}$
`\breve a`	$\breve{a}$
`\ddot a` or `\"a`	$\ddot{a}$
`\dot a` or `\.a`	$\dot{a}$
`\grave a` or `` \`a ``	$\grave{a}$
`\hat a` or `\^a`	$\hat{a}$
`\tilde a` or `\~a`	$\tilde{a}$
`\vec a`	$\vec{a}$
`\overline{abc}`	$\overline{abc}$

In addition, there are two special accents that automatically adjust to the width of the symbols below:

Command	Result
`\widehat{xyz}`	$\widehat{xyz}$
`\widetilde{xyz}`	$\widetilde{xyz}$

Care should be taken when putting accents on lower-case i's and j's. Note that in the following `\imath` is used to avoid the extra dot over the i:

```
r"$\hat i\ \ \hat \imath$"
```

$$\hat{i}\ \ \hat{\imath} \tag{3.12}$$

Symbols

You can also use a large number of the TeX symbols, as in `\infty`, `\leftarrow`, `\sum`, `\int`.

Lower-case Greek

α \alpha	β \beta	χ \chi	δ \delta	$\digamma$ \digamma
ϵ \epsilon	η \eta	γ \gamma	ι \iota	κ \kappa
λ \lambda	μ \mu	ν \nu	ω \omega	ϕ \phi
π \pi	ψ \psi	ρ \rho	σ \sigma	τ \tau
θ \theta	υ \upsilon	ε \varepsilon	$\varkappa$ \varkappa	φ \varphi
ϖ \varpi	ϱ \varrho	ς \varsigma	ϑ \vartheta	ξ \xi
ζ \zeta				

Upper-case Greek

Δ \Delta	Γ \Gamma	Λ \Lambda	Ω \Omega	Φ \Phi	Π \Pi
Ψ \Psi	Σ \Sigma	Θ \Theta	Υ \Upsilon	Ξ \Xi	$\mho$ \mho
∇ \nabla					

Hebrew

$\aleph$ \aleph	$\beth$ \beth	$\daleth$ \daleth	$\gimel$ \gimel

Delimiters

<table>
<tr><td>/ /</td><td>[[</td><td>⇓ \Downarrow</td><td>⇑ \Uparrow</td><td>‖ \Vert</td><td>\ \backslash</td></tr>
<tr><td>↓ \downarrow</td><td>⟨ \langle</td><td>⌈ \lceil</td><td>⌊ \lfloor</td><td>⌞ \llcorner</td><td>⌟ \lrcorner</td></tr>
<tr><td>⟩ \rangle</td><td>⌉ \rceil</td><td>⌋ \rfloor</td><td>⌜ \ulcorner</td><td>↑ \uparrow</td><td>⌝ \urcorner</td></tr>
<tr><td>| \vert</td><td>{ \{</td><td>‖ \|</td><td>} \}</td><td>]]</td><td>| |</td></tr>
</table>

Big symbols

$\bigcap$ \bigcap	$\bigcup$ \bigcup	$\bigodot$ \bigodot	$\bigoplus$ \bigoplus	$\bigotimes$ \bigotimes
$\biguplus$ \biguplus	$\bigvee$ \bigvee	$\bigwedge$ \bigwedge	$\coprod$ \coprod	$\int$ \int
$\oint$ \oint	$\prod$ \prod	$\sum$ \sum		

Standard function names

Pr \Pr	arccos \arccos	arcsin \arcsin	arctan \arctan
arg \arg	cos \cos	cosh \cosh	cot \cot
coth \coth	csc \csc	deg \deg	det \det
dim \dim	exp \exp	gcd \gcd	hom \hom
inf \inf	ker \ker	lg \lg	lim \lim
lim inf \liminf	lim sup \limsup	ln \ln	log \log
max \max	min \min	sec \sec	sin \sin
sinh \sinh	sup \sup	tan \tan	tanh \tanh

Binary operation and relation symbols

≏ \Bumpeq	⋒ \Cap	⋓ \Cup
≑ \Doteq	⋈ \Join	⋐ \Subset
⋑ \Supset	⊩ \Vdash	⊪ \Vvdash
≈ \approx	≊ \approxeq	∗ \ast
≍ \asymp	϶ \backepsilon	∽ \backsim
⋍ \backsimeq	⊼ \barwedge	∵ \because
≬ \between	◯ \bigcirc	▽ \bigtriangledown
△ \bigtriangleup	◀ \blacktriangleleft	▶ \blacktriangleright
⊥ \bot	⋈ \bowtie	⊡ \boxdot
⊟ \boxminus	⊞ \boxplus	⊠ \boxtimes
• \bullet	≏ \bumpeq	∩ \cap
· \cdot	∘ \circ	≗ \circeq
≔ \coloneq	≅ \cong	∪ \cup
⋞ \curlyeqprec	⋟ \curlyeqsucc	⋎ \curlyvee
⋏ \curlywedge	† \dag	⊣ \dashv
‡ \ddag	⋄ \diamond	÷ \div
⋇ \divideontimes	≐ \doteq	≑ \doteqdot
∔ \dotplus	⩞ \doublebarwedge	≖ \eqcirc
≕ \eqcolon	≂ \eqsim	⪖ \eqslantgtr
⪕ \eqslantless	≡ \equiv	≒ \fallingdotseq

⌢ \frown	≥ \geq	≧ \geqq
⩾ \geqslant	≫ \gg	⋙ \ggg
⪊ \gnapprox	≩ \gneqq	⋧ \gnsim
⪆ \gtrapprox	⋗ \gtrdot	⋛ \gtreqless
⪌ \gtreqqless	≷ \gtrless	≳ \gtrsim
∈ \in	⊺ \intercal	⋋ \leftthreetimes
≤ \leq	≦ \leqq	⩽ \leqslant
⪅ \lessapprox	⋖ \lessdot	⋚ \lesseqgtr
⪋ \lesseqqgtr	≶ \lessgtr	≲ \lesssim
≪ \ll	⋘ \lll	⪉ \lnapprox
≨ \lneqq	⋦ \lnsim	⋉ \ltimes
| \mid	⊧ \models	∓ \mp
⊯ \nVDash	⊮ \nVdash	≉ \napprox
≇ \ncong	≠ \ne	≠ \neq
≠ \neq	≢ \nequiv	≱ \ngeq
≯ \ngtr	∋ \ni	≰ \nleq
≮ \nless	∤ \nmid	∉ \notin
∦ \nparallel	⊀ \nprec	≁ \nsim
⊄ \nsubset	⊈ \nsubseteq	⊁ \nsucc
⊅ \nsupset	⊉ \nsupseteq	⋪ \ntriangleleft

⋬ \ntrianglelefteq	⋫ \ntriangleright	⋭ \ntrianglerighteq
⊮ \nvDash	⊬ \nvdash	⊙ \odot
⊖ \ominus	⊕ \oplus	⊘ \oslash
⊗ \otimes	∥ \parallel	⊥ \perp
⋔ \pitchfork	± \pm	≺ \prec
⪷ \precapprox	≼ \preccurlyeq	⪯ \preceq
⪹ \precnapprox	⋨ \precnsim	≾ \precsim
∝ \propto	⋌ \rightthreetimes	≓ \risingdotseq
⋊ \rtimes	∼ \sim	≃ \simeq
/ \slash	⌣ \smile	⊓ \sqcap
⊔ \sqcup	⊏ \sqsubset	⊏ \sqsubset
⊑ \sqsubseteq	⊐ \sqsupset	⊐ \sqsupset
⊒ \sqsupseteq	⋆ \star	⊂ \subset
⊆ \subseteq	⫅ \subseteqq	⊊ \subsetneq
⫋ \subsetneqq	≻ \succ	⪸ \succapprox
≽ \succcurlyeq	⪰ \succeq	⪺ \succnapprox
⋩ \succnsim	≿ \succsim	⊃ \supset
⊇ \supseteq	⫆ \supseteqq	⊋ \supsetneq
⫌ \supsetneqq	∴ \therefore	× \times
⊤ \top	◂ \triangleleft	⊴ \trianglelefteq
≜ \triangleq	▸ \triangleright	⊵ \trianglerighteq
⊎ \uplus	⊨ \vDash	∝ \varpropto
⊲ \vartriangleleft	⊳ \vartriangleright	⊢ \vdash
∨ \vee	⊻ \veebar	∧ \wedge
≀ \wr		

Arrow symbols

$\Downarrow$ \Downarrow	$\Leftarrow$ \Leftarrow
$\Leftrightarrow$ \Leftrightarrow	$\Lleftarrow$ \Lleftarrow
$\Longleftarrow$ \Longleftarrow	$\Longleftrightarrow$ \Longleftrightarrow
$\Longrightarrow$ \Longrightarrow	$\Lsh$ \Lsh
$\Nearrow$ \Nearrow	$\Nwarrow$ \Nwarrow
$\Rightarrow$ \Rightarrow	$\Rrightarrow$ \Rrightarrow
$\Rsh$ \Rsh	$\Searrow$ \Searrow
$\Swarrow$ \Swarrow	$\Uparrow$ \Uparrow
$\Updownarrow$ \Updownarrow	$\circlearrowleft$ \circlearrowleft
$\circlearrowright$ \circlearrowright	$\curvearrowleft$ \curvearrowleft
$\curvearrowright$ \curvearrowright	$\dashleftarrow$ \dashleftarrow
$\dashrightarrow$ \dashrightarrow	$\downarrow$ \downarrow
$\downdownarrows$ \downdownarrows	$\downharpoonleft$ \downharpoonleft
$\downharpoonright$ \downharpoonright	$\hookleftarrow$ \hookleftarrow
$\hookrightarrow$ \hookrightarrow	$\leadsto$ \leadsto
$\leftarrow$ \leftarrow	$\leftarrowtail$ \leftarrowtail
$\leftharpoondown$ \leftharpoondown	$\leftharpoonup$ \leftharpoonup
$\leftleftarrows$ \leftleftarrows	$\leftrightarrow$ \leftrightarrow
$\leftrightarrows$ \leftrightarrows	$\leftrightharpoons$ \leftrightharpoons
$\leftrightsquigarrow$ \leftrightsquigarrow	$\leftsquigarrow$ \leftsquigarrow

$\longleftarrow$ \longleftarrow	$\longleftrightarrow$ \longleftrightarrow
$\longmapsto$ \longmapsto	$\longrightarrow$ \longrightarrow
$\looparrowleft$ \looparrowleft	$\looparrowright$ \looparrowright
$\mapsto$ \mapsto	$\multimap$ \multimap
$\nLeftarrow$ \nLeftarrow	$\nLeftrightarrow$ \nLeftrightarrow
$\nRightarrow$ \nRightarrow	$\nearrow$ \nearrow
$\nleftarrow$ \nleftarrow	$\nleftrightarrow$ \nleftrightarrow
$\nrightarrow$ \nrightarrow	$\nwarrow$ \nwarrow
$\rightarrow$ \rightarrow	$\rightarrowtail$ \rightarrowtail
$\rightharpoondown$ \rightharpoondown	$\rightharpoonup$ \rightharpoonup
$\rightleftarrows$ \rightleftarrows	$\rightleftarrows$ \rightleftarrows
$\rightleftharpoons$ \rightleftharpoons	$\rightleftharpoons$ \rightleftharpoons
$\rightrightarrows$ \rightrightarrows	$\rightrightarrows$ \rightrightarrows
$\rightsquigarrow$ \rightsquigarrow	$\searrow$ \searrow
$\swarrow$ \swarrow	$\to$ \to
$\twoheadleftarrow$ \twoheadleftarrow	$\twoheadrightarrow$ \twoheadrightarrow
$\uparrow$ \uparrow	$\updownarrow$ \updownarrow
$\updownarrow$ \updownarrow	$\upharpoonleft$ \upharpoonleft
$\upharpoonright$ \upharpoonright	$\upuparrows$ \upuparrows

Miscellaneous symbols

$ \$	Å \AA	Ⅎ \Finv
⅁ \Game	ℑ \Im	¶ \P
ℜ \Re	§ \S	∠ \angle
‵ \backprime	★ \bigstar	■ \blacksquare
▲ \blacktriangle	▼ \blacktriangledown	⋯ \cdots
✓ \checkmark	® \circledR	Ⓢ \circledS
♣ \clubsuit	∁ \complement	© \copyright
⋱ \ddots	♢ \diamondsuit	ℓ \ell
∅ \emptyset	ð \eth	∃ \exists
♭ \flat	∀ \forall	ħ \hbar
♡ \heartsuit	ℏ \hslash	∭ \iiint
∬ \iint	∬ \iint	ı \imath
∞ \infty	ȷ \jmath	… \ldots
∡ \measuredangle	♮ \natural	¬ \neg
∄ \nexists	∰ \oiiint	∂ \partial
′ \prime	♯ \sharp	♠ \spadesuit
∢ \sphericalangle	\ss	▿ \triangledown
∅ \varnothing	▵ \vartriangle	⋮ \vdots
℘ \wp	¥ \yen	

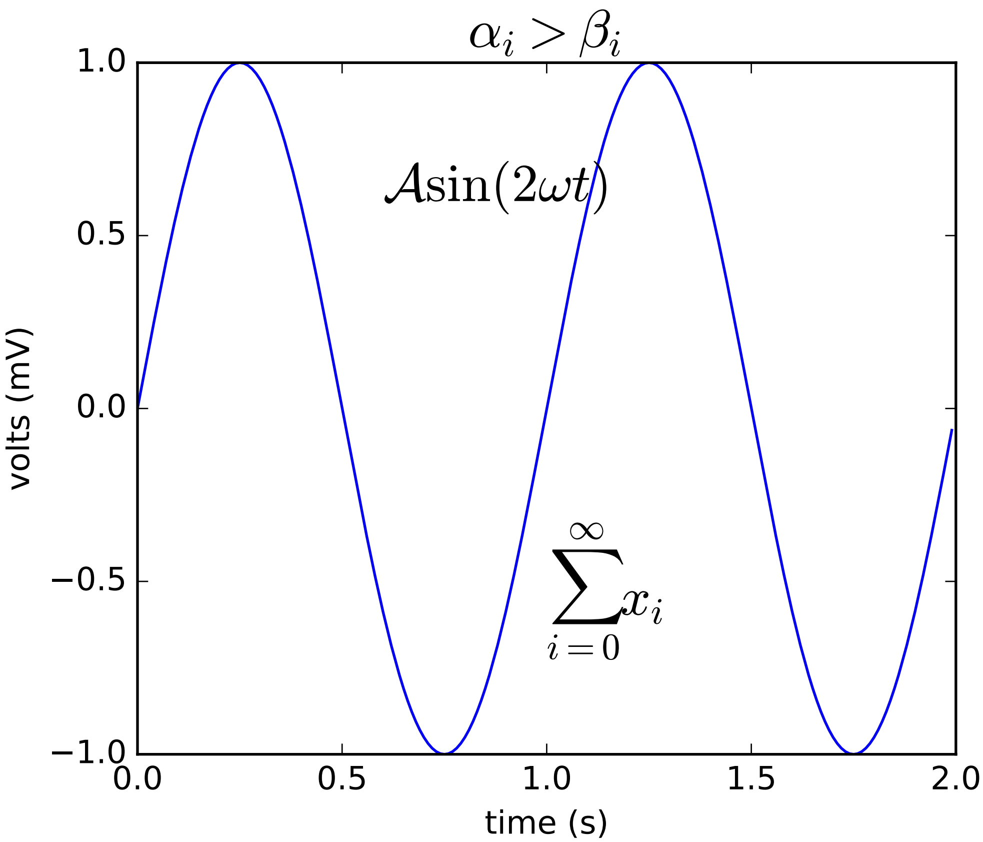
If a particular symbol does not have a name (as is true of many of the more obscure symbols in the STIX fonts), Unicode characters can also be used:

```
ur'$\u23ce$'
```

Example

Here is an example illustrating many of these features in context.

```
import numpy as np
import matplotlib.pyplot as plt
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)

plt.plot(t,s)
plt.title(r'$\alpha_i > \beta_i$', fontsize=20)
plt.text(1, -0.6, r'$\sum_{i=0}^\infty x_i$', fontsize=20)
plt.text(0.6, 0.6, r'$\mathcal{A}\mathrm{sin}(2 \omega t)$',
         fontsize=20)
plt.xlabel('time (s)')
plt.ylabel('volts (mV)')
plt.show()
```



3.4.5 Typesetting With XeLaTeX/LuaLaTeX

Using the `pgf` backend, matplotlib can export figures as pgf drawing commands that can be processed with pdflatex, xelatex or lualatex. XeLaTeX and LuaLaTeX have full unicode support and can use any font that is installed in the operating system, making use of advanced typographic features of OpenType, AAT and Graphite. Pgf pictures created by `plt.savefig('figure.pgf')` can be embedded as raw commands in LaTeX documents. Figures can also be directly compiled and saved to PDF with `plt.savefig('figure.pdf')` by either switching to the backend

```
matplotlib.use('pgf')
```

or registering it for handling pdf output

```
from matplotlib.backends.backend_pgf import FigureCanvasPgf
matplotlib.backend_bases.register_backend('pdf', FigureCanvasPgf)
```

The second method allows you to keep using regular interactive backends and to save xelatex, lualatex or pdflatex compiled PDF files from the graphical user interface.

Matplotlib's pgf support requires a recent LaTeX installation that includes the TikZ/PGF packages (such as TeXLive), preferably with XeLaTeX or LuaLaTeX installed. If either pdftocairo or ghostscript is present on your system, figures can optionally be saved to PNG images as well. The executables for all applications must be located on your *PATH*.

Rc parameters that control the behavior of the pgf backend:

Parameter	Documentation
pgf.preamble	Lines to be included in the LaTeX preamble
pgf.rcfonts	Setup fonts from rc params using the fontspec package
pgf.texsystem	Either "xelatex" (default), "lualatex" or "pdflatex"

Note: TeX defines a set of special characters, such as:

```
# $ % & ~ _ ^ \ { }
```

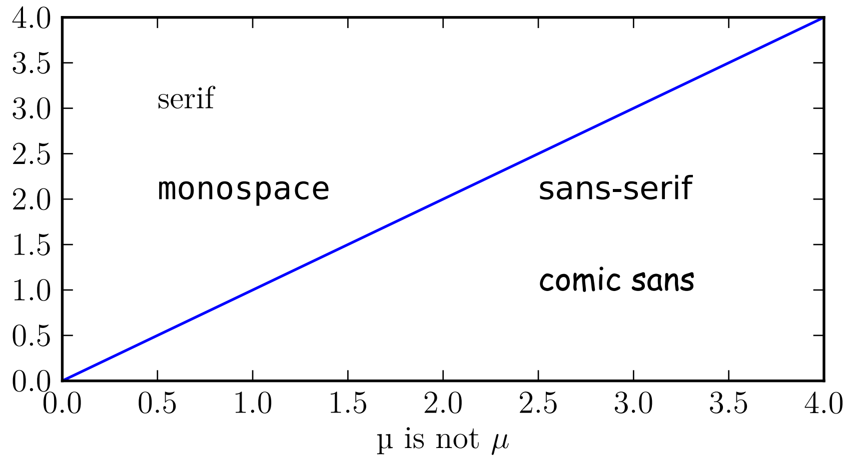
Generally, these characters must be escaped correctly. For convenience, some characters (_,^,%) are automatically escaped outside of math environments.

Font specification

The fonts used for obtaining the size of text elements or when compiling figures to PDF are usually defined in the matplotlib rc parameters. You can also use the LaTeX default Computer Modern fonts by clearing the lists for `font.serif`, `font.sans-serif` or `font.monospace`. Please note that the glyph coverage of these fonts is very limited. If you want to keep the Computer Modern font face but require extended unicode support, consider installing the Computer Modern Unicode fonts *CMU Serif*, *CMU Sans Serif*, etc.

When saving to `.pgf`, the font configuration matplotlib used for the layout of the figure is included in the header of the text file.

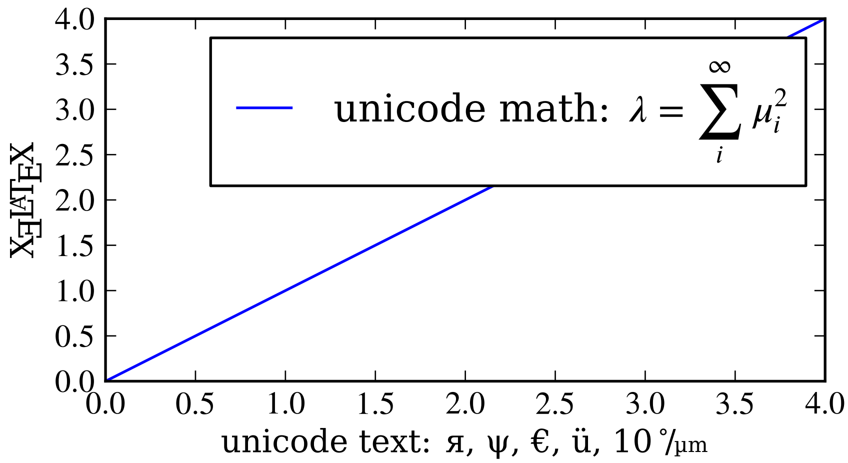
```
# -*- coding: utf-8 -*-

import matplotlib as mpl
mpl.use("pgf")
pgf_with_rc_fonts = {
    "font.family": "serif",
    "font.serif": [],                   # use latex default serif font
    "font.sans-serif": ["DejaVu Sans"], # use a specific sans-serif font
}
mpl.rcParams.update(pgf_with_rc_fonts)

import matplotlib.pyplot as plt
plt.figure(figsize=(4.5,2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.text(2.5, 1., "comic sans", family="Comic Sans MS")
plt.xlabel(u"µ is not $\\mu$")
plt.tight_layout(.5)
```



Custom preamble

Full customization is possible by adding your own commands to the preamble. Use the `pgf.preamble` parameter if you want to configure the math fonts, using `unicode-math` for example, or for loading additional packages. Also, if you want to do the font configuration yourself instead of using the fonts specified in the rc parameters, make sure to disable `pgf.rcfonts`.

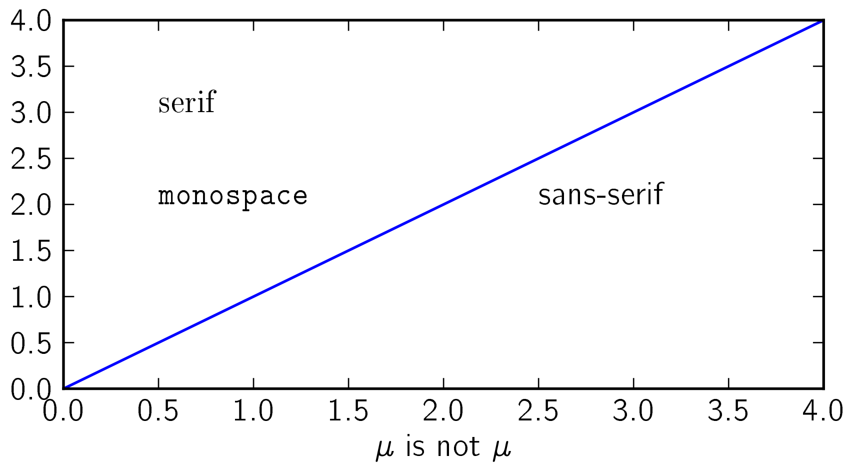
```
# -*- coding: utf-8 -*-
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

from matplotlib.externals import six

import matplotlib as mpl
mpl.use("pgf")
pgf_with_custom_preamble = {
    "font.family": "serif", # use serif/main font for text elements
    "text.usetex": True,    # use inline math for ticks
    "pgf.rcfonts": False,   # don't setup fonts from rc parameters
    "pgf.preamble": [
         "\\usepackage{units}",          # load additional packages
         "\\usepackage{metalogo}",
         "\\usepackage{unicode-math}",  # unicode math setup
         r"\setmathfont{xits-math.otf}",
         r"\setmainfont{DejaVu Serif}", # serif font via preamble
         ]
}
mpl.rcParams.update(pgf_with_custom_preamble)
```



Choosing the TeX system

The TeX system to be used by matplotlib is chosen by the `pgf.texsystem` parameter. Possible values are `'xelatex'` (default), `'lualatex'` and `'pdflatex'`. Please note that when selecting pdflatex the fonts and unicode handling must be configured in the preamble.

```
# -*- coding: utf-8 -*-

import matplotlib as mpl
mpl.use("pgf")
pgf_with_pdflatex = {
    "pgf.texsystem": "pdflatex",
    "pgf.preamble": [
         r"\usepackage[utf8x]{inputenc}",
         r"\usepackage[T1]{fontenc}",
         r"\usepackage{cmbright}",
         ]
}
mpl.rcParams.update(pgf_with_pdflatex)

import matplotlib.pyplot as plt
plt.figure(figsize=(4.5,2.5))
plt.plot(range(5))
plt.text(0.5, 3., "serif", family="serif")
plt.text(0.5, 2., "monospace", family="monospace")
plt.text(2.5, 2., "sans-serif", family="sans-serif")
plt.xlabel(u"µ is not $\\mu$")
plt.tight_layout(.5)
```



Troubleshooting

- Please note that the TeX packages found in some Linux distributions and MiKTeX installations are dramatically outdated. Make sure to update your package catalog and upgrade or install a recent TeX distribution.
- On Windows, the *PATH* environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See *Environment Variables* and *Setting environment variables in windows* for details.
- A limitation on Windows causes the backend to keep file handles that have been opened by your application open. As a result, it may not be possible to delete the corresponding files until the application closes (see #1324).
- Sometimes the font rendering in figures that are saved to png images is very bad. This happens when the pdftocairo tool is not available and ghostscript is used for the pdf to png conversion.
- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.
- The `pgf.preamble` rc setting provides lots of flexibility, and lots of ways to cause problems. When experiencing problems, try to minimalize or disable the custom preamble.
- Configuring an `unicode-math` environment can be a bit tricky. The TeXLive distribution for example provides a set of math fonts which are usually not installed system-wide. XeTeX, unlike LuaLatex, cannot find these fonts by their name, which is why you might have to specify `\setmathfont{xits-math.otf}` instead of `\setmathfont{XITS Math}` or alternatively make the fonts available to your OS. See this tex.stackexchange.com question for more details.
- If the font configuration used by matplotlib differs from the font setting in yout LaTeX document, the alignment of text elements in imported figures may be off. Check the header of your `.pgf` file if you are unsure about the fonts matplotlib used for the layout.
- Vector images and hence `.pgf` files can become bloated if there are a lot of objects in the graph. This can be the case for image processing or very big scatter graphs. In an extreme case this can cause TeX to run out of memory: "TeX capacity exceeded, sorry" You can configure latex to increase the amount

of memory available to generate the `.pdf` image as discussed on tex.stackexchange.com. Another way would be to "rasterize" parts of the graph causing problems using either the `rasterized=True` keyword, or `.set_rasterized(True)` as per this example.

- If you still need help, please see *Getting help*

3.4.6 Text rendering With LaTeX

Matplotlib has the option to use LaTeX to manage all text layout. This option is available with the following backends:

- Agg
- PS
- PDF

The LaTeX option is activated by setting `text.usetex : True` in your rc settings. Text handling with matplotlib's LaTeX support is slower than matplotlib's very capable *mathtext*, but is more flexible, since different LaTeX packages (font packages, math packages, etc.) can be used. The results can be striking, especially when you take care to use the same fonts in your figures as in the main document.

Matplotlib's LaTeX support requires a working LaTeX installation, dvipng (which may be included with your LaTeX installation), and Ghostscript (GPL Ghostscript 8.60 or later is recommended). The executables for these external dependencies must all be located on your *PATH*.

There are a couple of options to mention, which can be changed using *rc settings*. Here is an example matplotlibrc file:

```
font.family        : serif
font.serif         : Times, Palatino, New Century Schoolbook, Bookman, Computer Modern␣
 ↪Roman
font.sans-serif    : Helvetica, Avant Garde, Computer Modern Sans serif
font.cursive       : Zapf Chancery
font.monospace     : Courier, Computer Modern Typewriter

text.usetex        : true
```

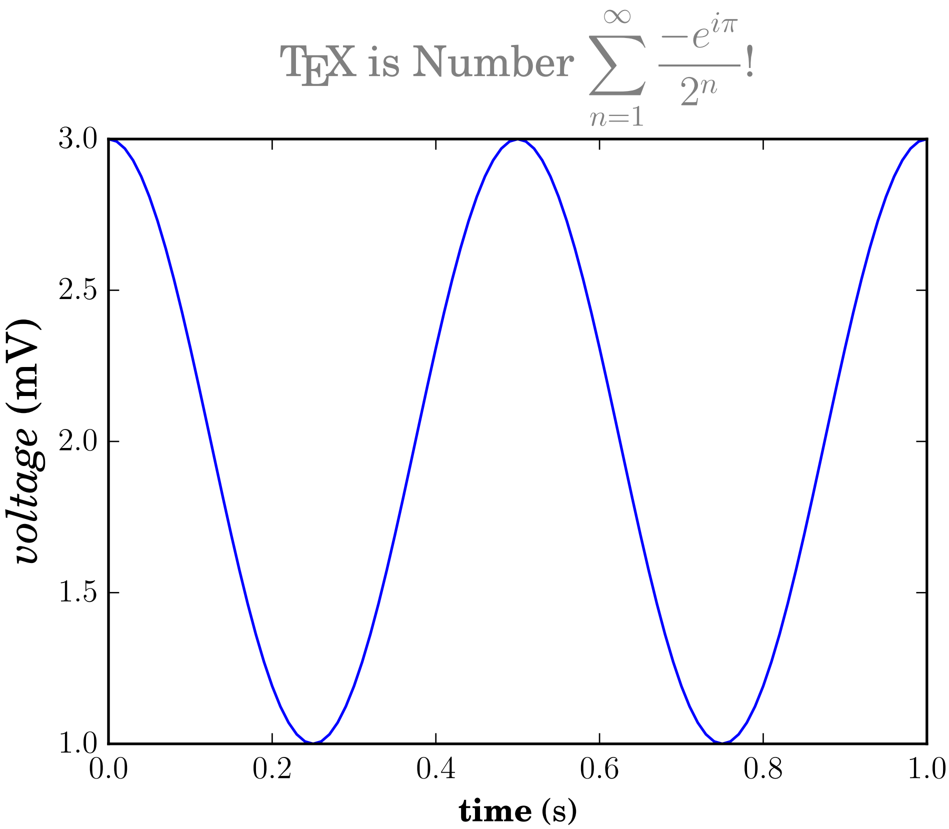
The first valid font in each family is the one that will be loaded. If the fonts are not specified, the Computer Modern fonts are used by default. All of the other fonts are Adobe fonts. Times and Palatino each have their own accompanying math fonts, while the other Adobe serif fonts make use of the Computer Modern math fonts. See the PSNFSS documentation for more details.

To use LaTeX and select Helvetica as the default font, without editing matplotlibrc use:

```
from matplotlib import rc
rc('font',**{'family':'sans-serif','sans-serif':['Helvetica']})
## for Palatino and other serif fonts use:
#rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)
```

Here is the standard example, `tex_demo.py`:

```python
"""
Demo of TeX rendering.

You can use TeX to render all of your matplotlib text if the rc
parameter text.usetex is set.  This works currently on the agg and ps
backends, and requires that you have tex and the other dependencies
described at http://matplotlib.org/users/usetex.html
properly installed on your system.  The first time you run a script
you will see a lot of output from tex and associated tools.  The next
time, the run may be silent, as a lot of the information is cached in
~/.tex.cache

"""
import numpy as np
import matplotlib.pyplot as plt


# Example data
t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(4 * np.pi * t) + 2

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.plot(t, s)

plt.xlabel(r'\textbf{time} (s)')
plt.ylabel(r'\textit{voltage} (mV)',fontsize=16)
plt.title(r"\TeX\ is Number "
          r"$\displaystyle\sum_{n=1}^\infty\frac{-e^{i\pi}}{2^n}$!",
          fontsize=16, color='gray')
# Make room for the ridiculously large title.
plt.subplots_adjust(top=0.8)

plt.savefig('tex_demo')
plt.show()
```



Note that display math mode (`$$ e=mc^2 $$`) is not supported, but adding the command `\displaystyle`, as in `tex_demo.py`, will produce the same results.

Note: Certain characters require special escaping in TeX, such as:

```
# $ % & ~ _ ^ \ { } \( \) \[ \]
```

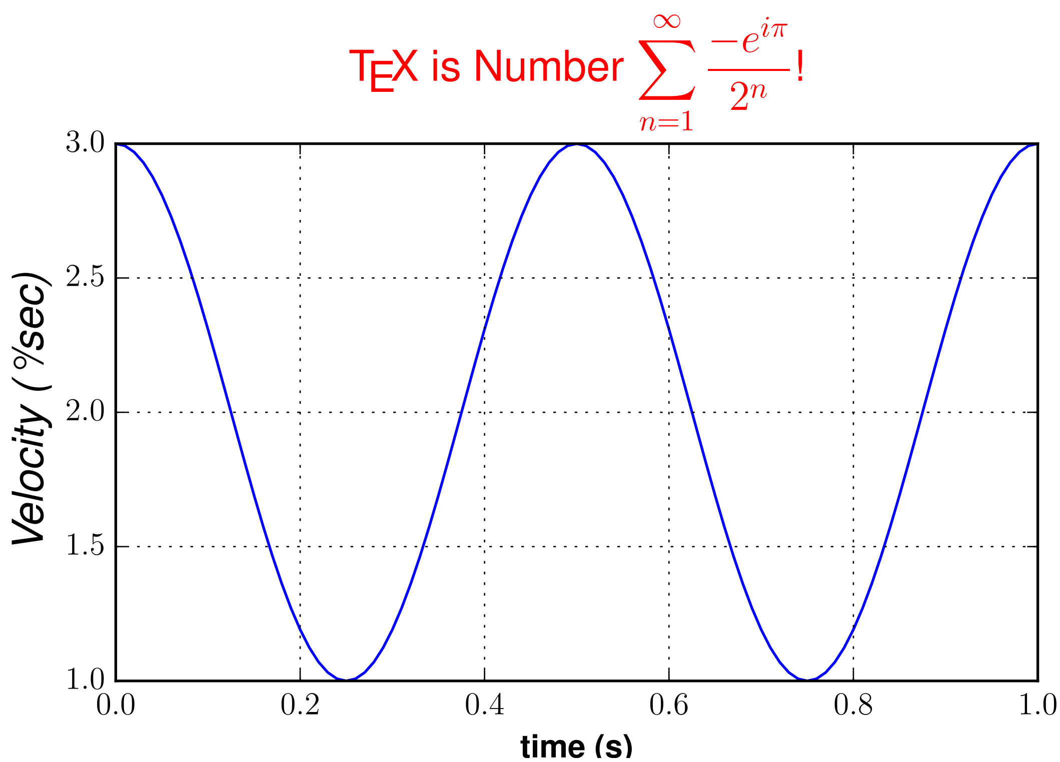
Therefore, these characters will behave differently depending on the rcParam `text.usetex` flag.

usetex with unicode

It is also possible to use unicode strings with the LaTeX text manager, here is an example taken from `tex_unicode_demo.py`:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
This demo is tex_demo.py modified to have unicode. See that file for
more information.
"""

from __future__ import unicode_literals
```

```python
import numpy as np
import matplotlib
matplotlib.rcParams['text.usetex'] = True
matplotlib.rcParams['text.latex.unicode'] = True
import matplotlib.pyplot as plt

plt.figure(1, figsize=(6, 4))
ax = plt.axes([0.1, 0.1, 0.8, 0.7])
t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(2*2*np.pi*t) + 2
plt.plot(t, s)

plt.xlabel(r'\textbf{time (s)}')
plt.ylabel('\\textit{Velocity (\u00B0/sec)}', fontsize=16)
plt.title(r'\TeX\ is Number $\displaystyle\sum_{n=1}^\infty'
          r'\frac{-e^{i\pi}}{2^n}$!', fontsize=16, color='r')
plt.grid(True)
plt.show()
```



Postscript options

In order to produce encapsulated postscript files that can be embedded in a new LaTeX document, the default behavior of matplotlib is to distill the output, which removes some postscript operators used by LaTeX that are illegal in an eps file. This step produces results which may be unacceptable to some users, because the text is coarsely rasterized and converted to bitmaps, which are not scalable like standard postscript, and the text is not searchable. One workaround is to to set `ps.distiller.res` to a higher value (perhaps

6000) in your rc settings, which will produce larger files but may look better and scale reasonably. A better workaround, which requires Poppler or Xpdf, can be activated by changing the `ps.usedistiller` rc setting to `xpdf`. This alternative produces postscript without rasterizing text, so it scales properly, can be edited in Adobe Illustrator, and searched text in pdf documents.

Possible hangups

- On Windows, the *PATH* environment variable may need to be modified to include the directories containing the latex, dvipng and ghostscript executables. See *Environment Variables* and *Setting environment variables in windows* for details.
- Using MiKTeX with Computer Modern fonts, if you get odd *Agg and PNG results, go to MiKTeX/Options and update your format files
- The fonts look terrible on screen. You are probably running Mac OS, and there is some funny business with older versions of dvipng on the mac. Set `text.dvipnghack : True` in your matplotlibrc file.
- On Ubuntu and Gentoo, the base texlive install does not ship with the type1cm package. You may need to install some of the extra packages to get all the goodies that come bundled with other latex distributions.
- Some progress has been made so matplotlib uses the dvi files directly for text layout. This allows latex to be used for text layout with the pdf and svg backends, as well as the *Agg and PS backends. In the future, a latex installation may be the only external dependency.

Troubleshooting

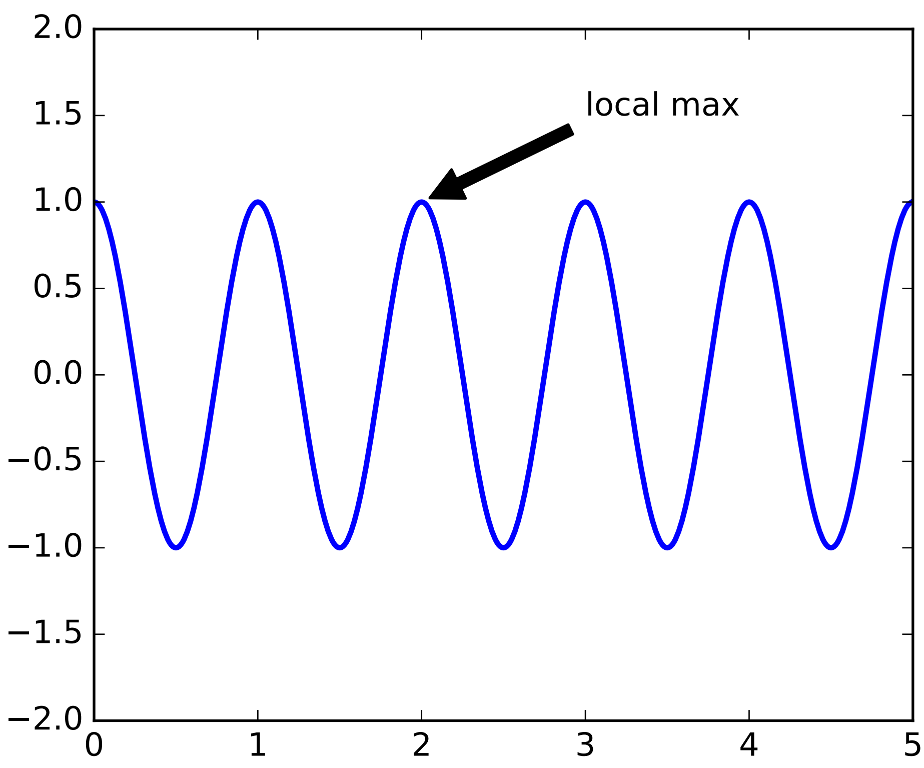
- Try deleting your `.matplotlib/tex.cache` directory. If you don't know where to find `.matplotlib`, see *matplotlib configuration and cache directory locations*.
- Make sure LaTeX, dvipng and ghostscript are each working and on your *PATH*.
- Make sure what you are trying to do is possible in a LaTeX document, that your LaTeX syntax is valid and that you are using raw strings if necessary to avoid unintended escape sequences.
- Most problems reported on the mailing list have been cleared up by upgrading Ghostscript. If possible, please try upgrading to the latest release before reporting problems to the list.
- The `text.latex.preamble` rc setting is not officially supported. This option provides lots of flexibility, and lots of ways to cause problems. Please disable this option before reporting problems to the mailing list.
- If you still need help, please see *Getting help*

3.4.7 Annotating text

For a more detailed introduction to annotations, see *Annotating Axes*.

The uses of the basic `text()` command above place text at an arbitrary position on the Axes. A common use case of text is to annotate some feature of the plot, and the `annotate()` method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument `xy` and the location of the text `xytext`. Both of these arguments are `(x,y)` tuples.

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05),
            )

ax.set_ylim(-2,2)
plt.show()
```



In this example, both the `xy` (arrow tip) and `xytext` locations (text location) are in data coordinates. There are a variety of other coordinate systems one can choose – you can specify the coordinate system of `xy` and `xytext` with one of the following strings for `xycoords` and `textcoords` (default is 'data')

argument	coordinate system
'figure points'	points from the lower left corner of the figure
'figure pixels'	pixels from the lower left corner of the figure
'figure fraction'	0,0 is lower left of figure and 1,1 is upper right
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	0,0 is lower left of axes and 1,1 is upper right
'data'	use the axes data coordinate system

For example to place the text coordinates in fractional axes coordinates, one could do:

```
ax.annotate('local max', xy=(3, 1),  xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top',
            )
```

For physical coordinate systems (points or pixels) the origin is the (bottom, left) of the figure or axes. If the value is negative, however, the origin is from the (right, top) of the figure or axes, analogous to negative indexing of sequences.

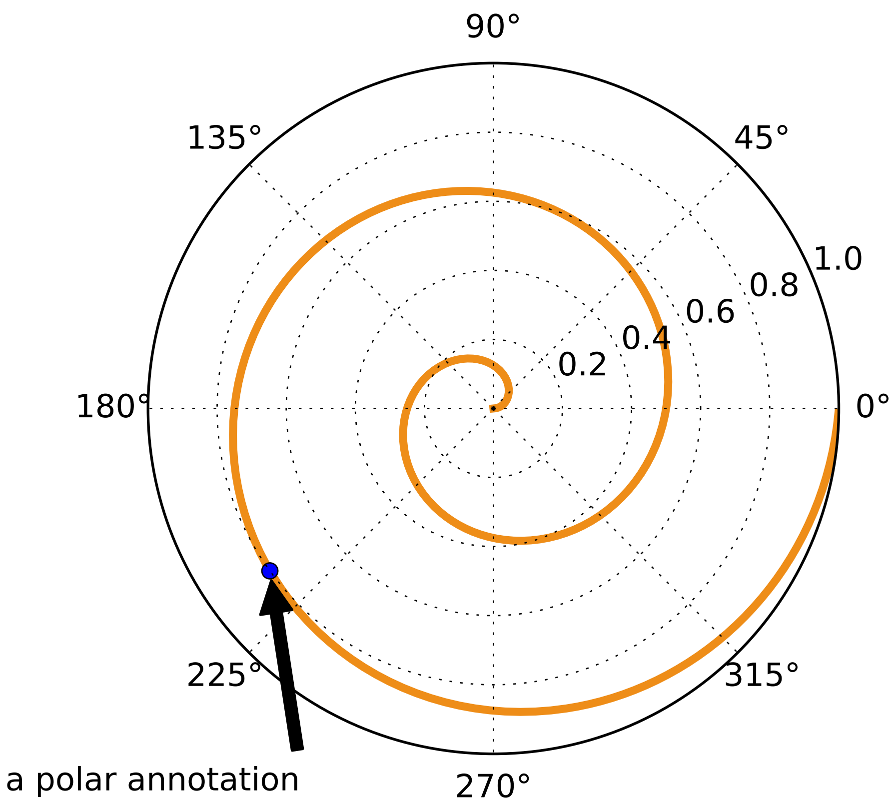
Optionally, you can specify arrow properties which draws an arrow from the text to the annotated point by giving a dictionary of arrow properties in the optional keyword argument `arrowprops`.

arrowprops key	description
width	the width of the arrow in points
frac	the fraction of the arrow length occupied by the head
headwidth	the width of the base of the arrow head in points
shrink	move the tip and base some percent away from the annotated point and text
**kwargs	any key for *matplotlib.patches.Polygon*, e.g., `facecolor`

In the example below, the `xy` point is in native coordinates (`xycoords` defaults to 'data'). For a polar axes, this is in (theta, radius) space. The text in this example is placed in the fractional figure coordinate system. *matplotlib.text.Text* keyword args like `horizontalalignment`, `verticalalignment` and `fontsize are passed from the '~matplotlib.Axes.annotate' to the ''Text` instance

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, polar=True)
r = np.arange(0,1,0.001)
theta = 2*2*np.pi*r
line, = ax.plot(theta, r, color='#ee8d18', lw=3)

ind = 800
thisr, thistheta = r[ind], theta[ind]
ax.plot([thistheta], [thisr], 'o')
ax.annotate('a polar annotation',
            xy=(thistheta, thisr),  # theta, radius
            xytext=(0.05, 0.05),    # fraction, fraction
            textcoords='figure fraction',
```

```
             arrowprops=dict(facecolor='black', shrink=0.05),
             horizontalalignment='left',
             verticalalignment='bottom',
             )
plt.show()
```



For more on all the wild and wonderful things you can do with annotations, including fancy arrows, see *Annotating Axes* and *pylab_examples example code: annotation_demo.py*.

3.5 Image tutorial

3.5.1 Startup commands

First, let's start IPython. It is a most excellent enhancement to the standard Python prompt, and it ties in especially well with Matplotlib. Start IPython either at a shell, or the IPython Notebook now.

With IPython started, we now need to connect to a GUI event loop. This tells IPython where (and how) to display plots. To connect to a GUI loop, execute the **%matplotlib** magic at your IPython prompt. There's more detail on exactly what this does at IPython's documentation on GUI event loops.

If you're using IPython Notebook, the same commands are available, but people commonly use a specific argument to the %matplotlib magic:

```
In [1]: %matplotlib inline
```

This turns on inline plotting, where plot graphics will appear in your notebook. This has important implications for interactivity. For inline plotting, commands in cells below the cell that outputs a plot will not affect the plot. For example, changing the color map is not possible from cells below the cell that creates a plot. However, for other backends, such as qt4, that open a separate window, cells below those that create the plot will change the plot - it is a live object in memory.

This tutorial will use matplotlib's imperative-style plotting interface, pyplot. This interface maintains global state, and is very useful for quickly and easily experimenting with various plot settings. The alternative is the object-oriented interface, which is also very powerful, and generally more suitable for large application development. If you'd like to learn about the object-oriented interface, a great place to start is our FAQ on usage. For now, let's get on with the imperative-style approach:

```
In [2]: import matplotlib.pyplot as plt
In [3]: import matplotlib.image as mpimg
In [4]: import numpy as np
```

3.5.2 Importing image data into Numpy arrays

Loading image data is supported by the Pillow library. Natively, matplotlib only supports PNG images. The commands shown below fall back on Pillow if the native read fails.

The image used in this example is a PNG file, but keep that Pillow requirement in mind for your own data.

Here's the image we're going to play with:

It's a 24-bit RGB PNG image (8 bits for each of R, G, B). Depending on where you get your data, the other kinds of image that you'll most likely encounter are RGBA images, which allow for transparency, or single-channel grayscale (luminosity) images. You can right click on it and choose "Save image as" to download it to your computer for the rest of this tutorial.

And here we go...

```
In [5]: img=mpimg.imread('stinkbug.png')
Out[5]:
array([[[ 0.40784314,  0.40784314,  0.40784314],
        [ 0.40784314,  0.40784314,  0.40784314],
        [ 0.40784314,  0.40784314,  0.40784314],
        ...,
        [ 0.42745098,  0.42745098,  0.42745098],
        [ 0.42745098,  0.42745098,  0.42745098],
        [ 0.42745098,  0.42745098,  0.42745098]],


       ...,
       [[ 0.44313726,  0.44313726,  0.44313726],
        [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
        [ 0.4509804 ,  0.4509804 ,  0.4509804 ],
        ...,
        [ 0.44705883,  0.44705883,  0.44705883],
```

```
        [ 0.44705883,  0.44705883,  0.44705883],
        [ 0.44313726,  0.44313726,  0.44313726]]], dtype=float32)
```

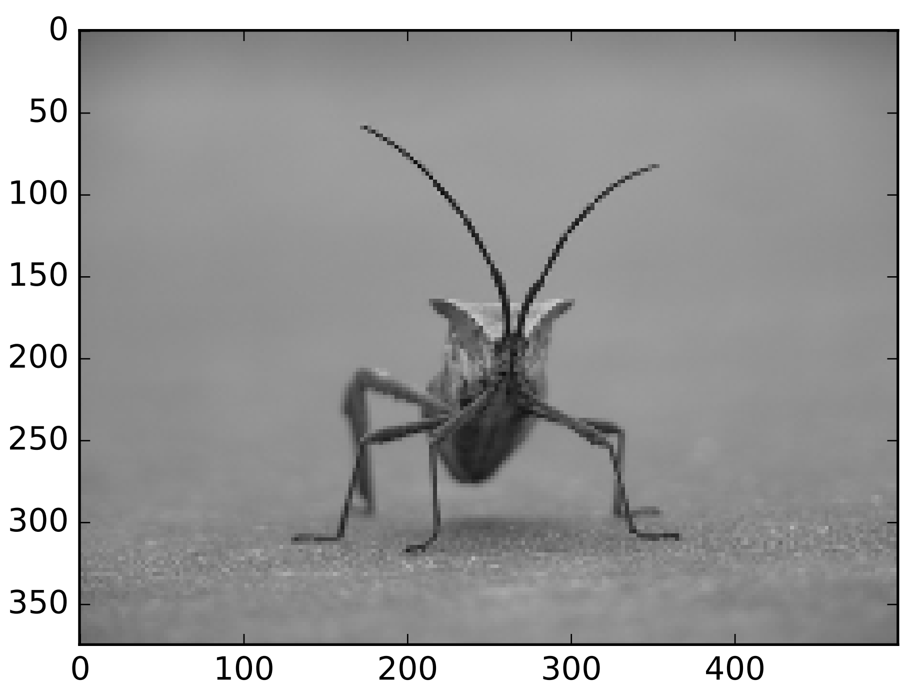
Note the dtype there - float32. Matplotlib has rescaled the 8 bit data from each channel to floating point data between 0.0 and 1.0. As a side note, the only datatype that Pillow can work with is uint8. Matplotlib plotting can handle float32 and uint8, but image reading/writing for any format other than PNG is limited to uint8 data. Why 8 bits? Most displays can only render 8 bits per channel worth of color gradation. Why can they only render 8 bits/channel? Because that's about all the human eye can see. More here (from a photography standpoint): Luminous Landscape bit depth tutorial.

Each inner list represents a pixel. Here, with an RGB image, there are 3 values. Since it's a black and white image, R, G, and B are all similar. An RGBA (where A is alpha, or transparency), has 4 values per inner list, and a simple luminance image just has one value (and is thus only a 2-D array, not a 3-D array). For RGB and RGBA images, matplotlib supports float32 and uint8 data types. For grayscale, matplotlib supports only float32. If your array data does not meet one of these descriptions, you need to rescale it.

3.5.3 Plotting numpy arrays as images

So, you have your data in a numpy array (either by importing it, or by generating it). Let's render it. In Matplotlib, this is performed using the `imshow()` function. Here we'll grab the plot object. This object gives you an easy way to manipulate the plot from the prompt.

```
In [6]: imgplot = plt.imshow(img)
```



You can also plot any numpy array.

Applying pseudocolor schemes to image plots

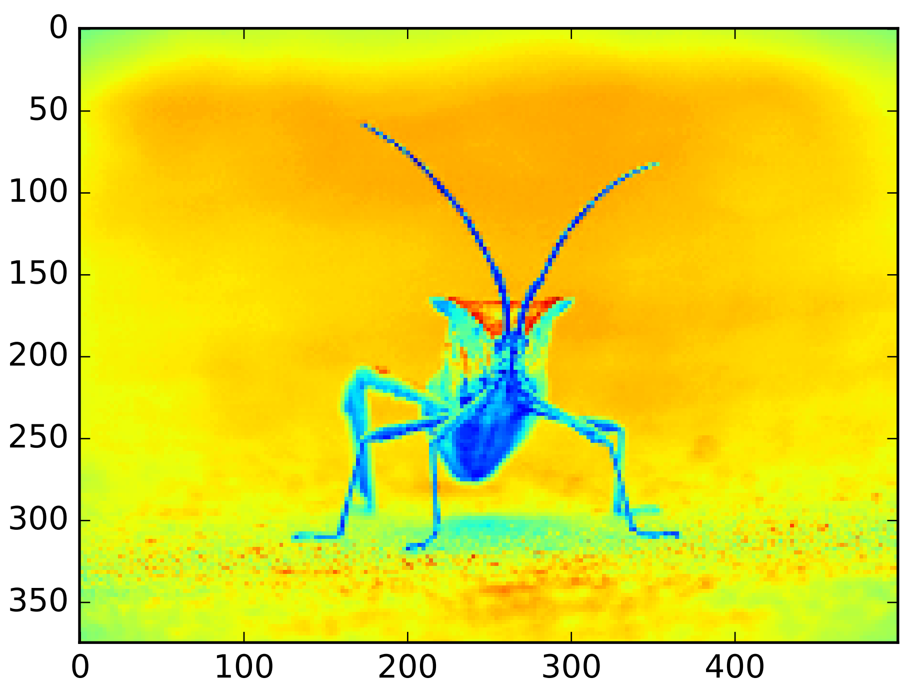
Pseudocolor can be a useful tool for enhancing contrast and visualizing your data more easily. This is especially useful when making presentations of your data using projectors - their contrast is typically quite poor.

Pseudocolor is only relevant to single-channel, grayscale, luminosity images. We currently have an RGB image. Since R, G, and B are all similar (see for yourself above or in your data), we can just pick one channel of our data:

```
In [7]: lum_img = img[:,:,0]
```

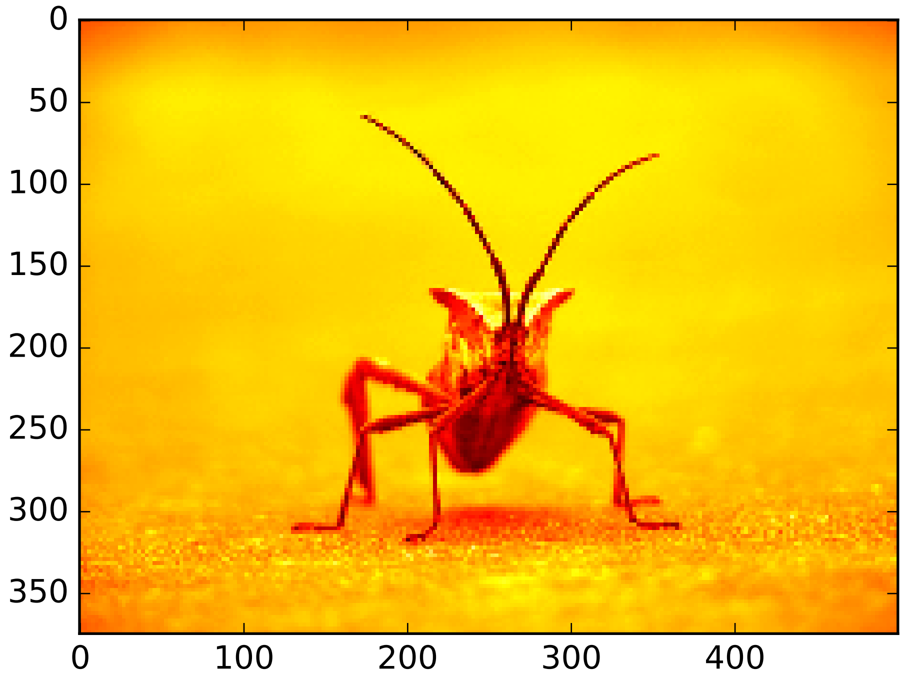
This is array slicing. You can read more in the Numpy tutorial.

```
In [8]: plt.imshow(lum_img)
```



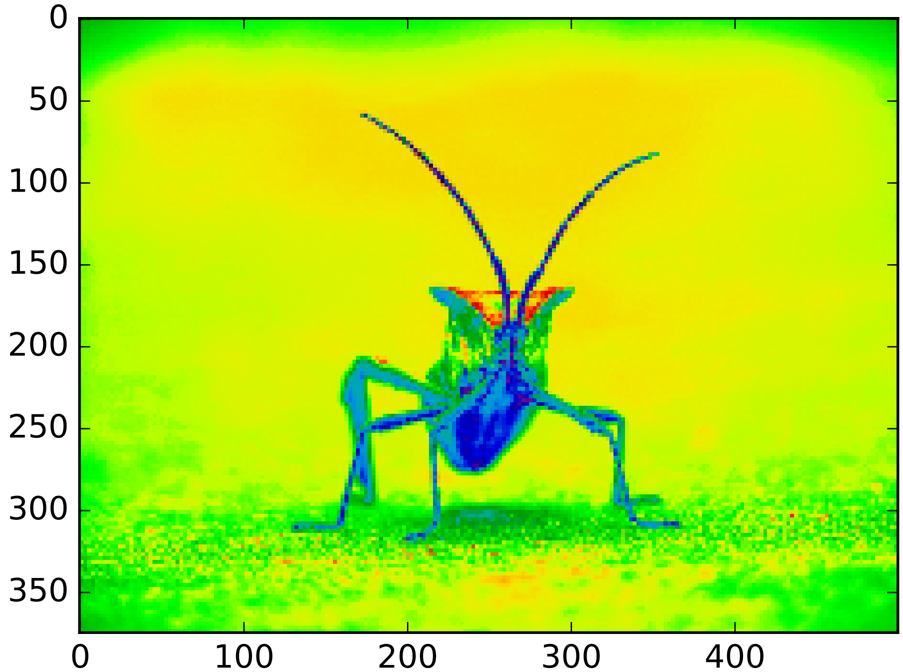
Now, with a luminosity (2D, no color) image, the default colormap (aka lookup table, LUT), is applied. The default is called jet. There are plenty of others to choose from.

```
In [9]: plt.imshow(lum_img, cmap="hot")
```



Note that you can also change colormaps on existing plot objects using the `set_cmap()` method:

```
In [10]: imgplot = plt.imshow(lum_img)
In [11]: imgplot.set_cmap('spectral')
```



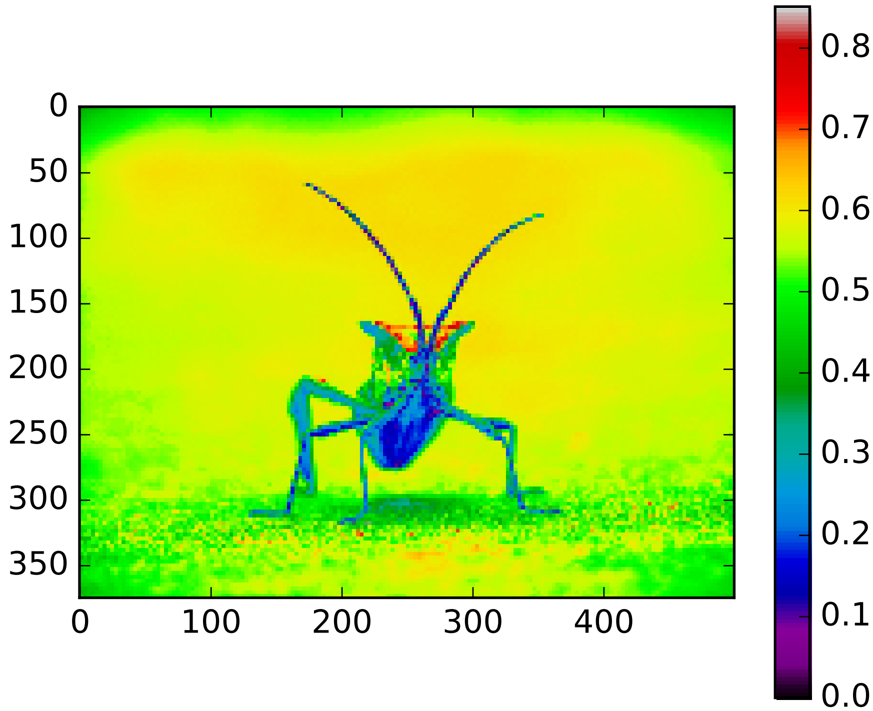
Note: However, remember that in the IPython notebook with the inline backend, you can't make changes to plots that have already been rendered. If you create imgplot here in one cell, you cannot call set_cmap() on it in a later cell and expect the earlier plot to change. Make sure that you enter these commands together in one cell. plt commands will not change plots from earlier cells.

There are many other colormap schemes available. See the list and images of the colormaps.

Color scale reference

It's helpful to have an idea of what value a color represents. We can do that by adding color bars.

```
In [12]: imgplot = plt.imshow(lum_img)
In [13]: plt.colorbar()
```

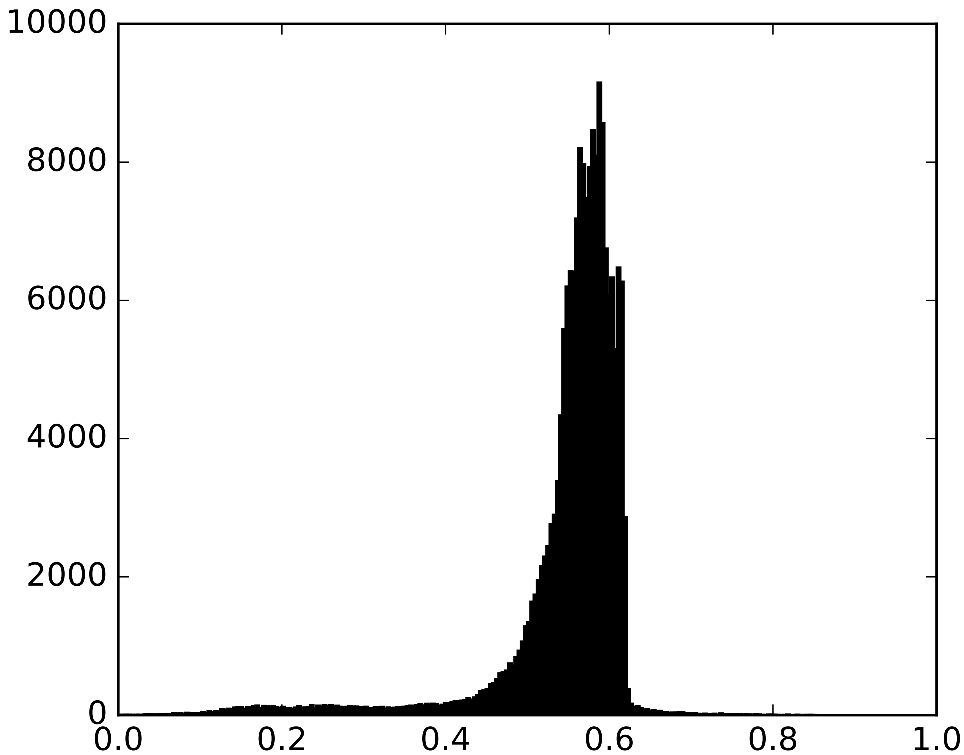


This adds a colorbar to your existing figure. This won't automatically change if you change you switch to a different colormap - you have to re-create your plot, and add in the colorbar again.

Examining a specific data range

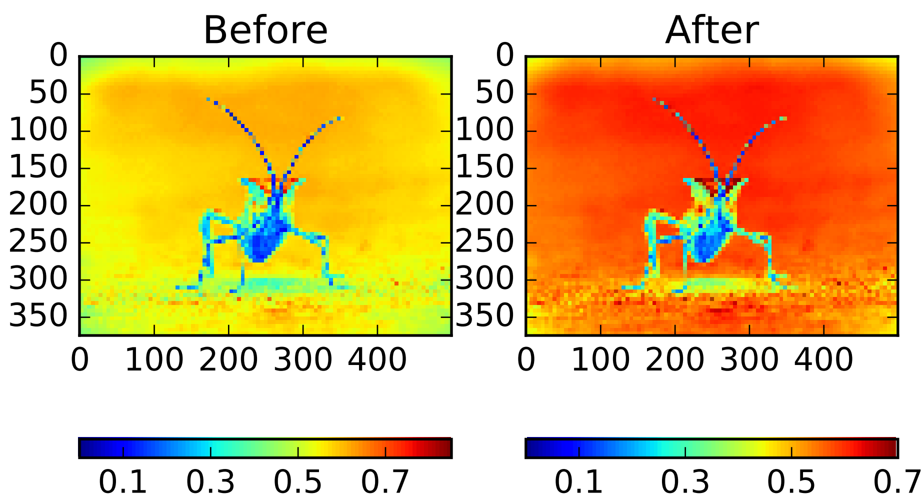
Sometimes you want to enhance the contrast in your image, or expand the contrast in a particular region while sacrificing the detail in colors that don't vary much, or don't matter. A good tool to find interesting regions is the histogram. To create a histogram of our image data, we use the `hist()` function.

```
In [14]: plt.hist(lum_img.ravel(), bins=256, range=(0.0, 1.0), fc='k', ec='k')
```



Most often, the "interesting" part of the image is around the peak, and you can get extra contrast by clipping the regions above and/or below the peak. In our histogram, it looks like there's not much useful information in the high end (not many white things in the image). Let's adjust the upper limit, so that we effectively "zoom in on" part of the histogram. We do this by passing the clim argument to imshow. You could also do this by calling the `set_clim()` method of the image plot object, but make sure that you do so in the same cell as your plot command when working with the IPython Notebook - it will not change plots from earlier cells.

```
In [15]: imgplot = plt.imshow(lum_img, clim=(0.0, 0.7))
```

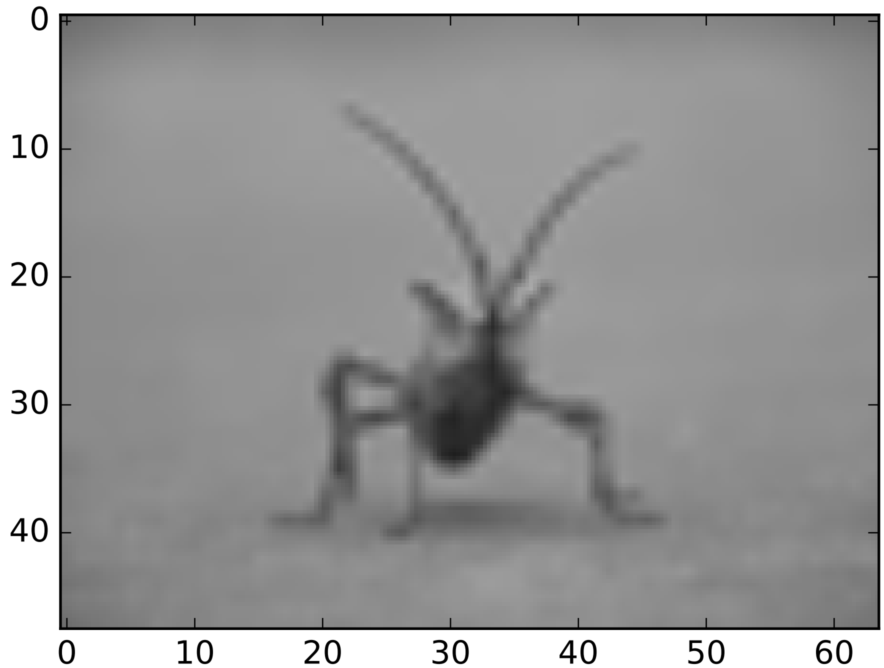


Array Interpolation schemes

Interpolation calculates what the color or value of a pixel "should" be, according to different mathematical schemes. One common place that this happens is when you resize an image. The number of pixels change, but you want the same information. Since pixels are discrete, there's missing space. Interpolation is how you fill that space. This is why your images sometimes come out looking pixelated when you blow them up. The effect is more pronounced when the difference between the original image and the expanded image is greater. Let's take our image and shrink it. We're effectively discarding pixels, only keeping a select few. Now when we plot it, that data gets blown up to the size on your screen. The old pixels aren't there anymore, and the computer has to draw in pixels to fill that space.

We'll use the Pillow library that we used to load the image also to resize the image.

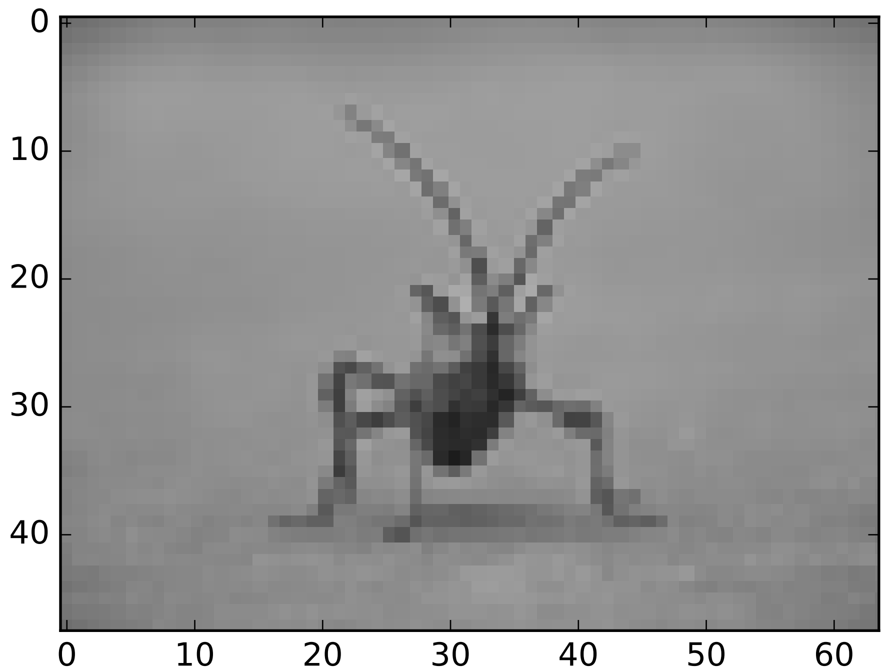
```
In [16]: from PIL import Image
In [17]: img = Image.open('../_static/stinkbug.png')
In [18]: img.thumbnail((64, 64), Image.ANTIALIAS) # resizes image in-place
In [19]: imgplot = plt.imshow(img)
```



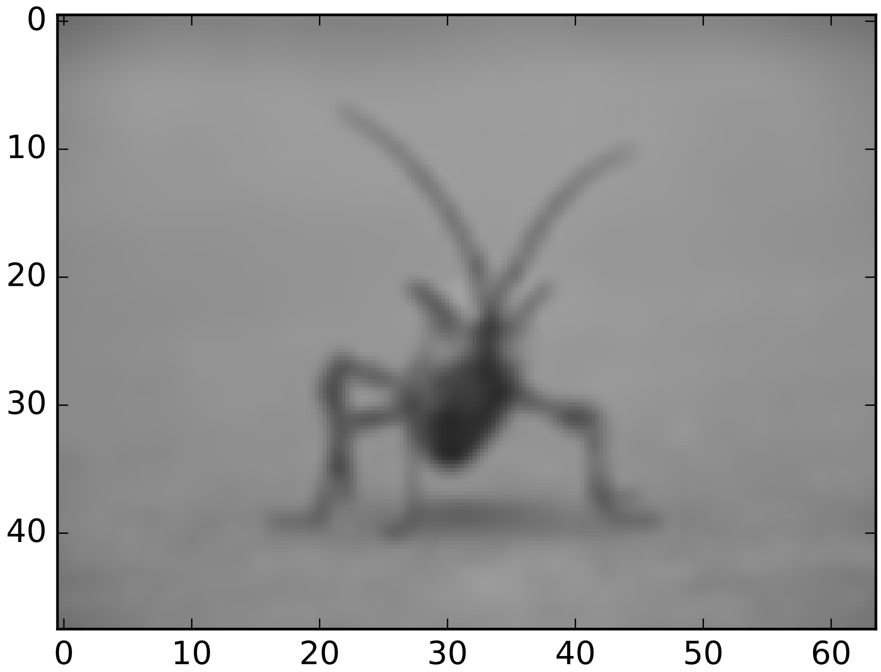
Here we have the default interpolation, bilinear, since we did not give *imshow()* any interpolation argument.

Let's try some others:

```
In [20]: imgplot = plt.imshow(img, interpolation="nearest")
```




```
In [21]: imgplot = plt.imshow(img, interpolation="bicubic")
```



Bicubic interpolation is often used when blowing up photos - people tend to prefer blurry over pixelated.

3.6 Legend guide

This legend guide is an extension of the documentation available at `legend()` - please ensure you are familiar with contents of that documentation before proceeding with this guide.

This guide makes use of some common terms, which are documented here for clarity:

legend entry A legend is made up of one or more legend entries. An entry is made up of exactly one key and one label.

legend key The colored/patterned marker to the left of each legend label.

legend label The text which describes the handle represented by the key.

legend handle The original object which is used to generate an appropriate entry in the legend.

3.6.1 Controlling the legend entries

Calling `legend()` with no arguments automatically fetches the legend handles and their associated labels. This functionality is equivalent to:

```
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)
```

The `get_legend_handles_labels()` function returns a list of handles/artists which exist on the Axes which can be used to generate entries for the resulting legend - it is worth noting however that not all artists can be added to a legend, at which point a "proxy" will have to be created (see *Creating artists specifically for adding to the legend (aka. Proxy artists)* for further details).

For full control of what is being added to the legend, it is common to pass the appropriate handles directly to `legend()`:

```
line_up, = plt.plot([1,2,3], label='Line 2')
line_down, = plt.plot([3,2,1], label='Line 1')
plt.legend(handles=[line_up, line_down])
```

In some cases, it is not possible to set the label of the handle, so it is possible to pass through the list of labels to `legend()`:

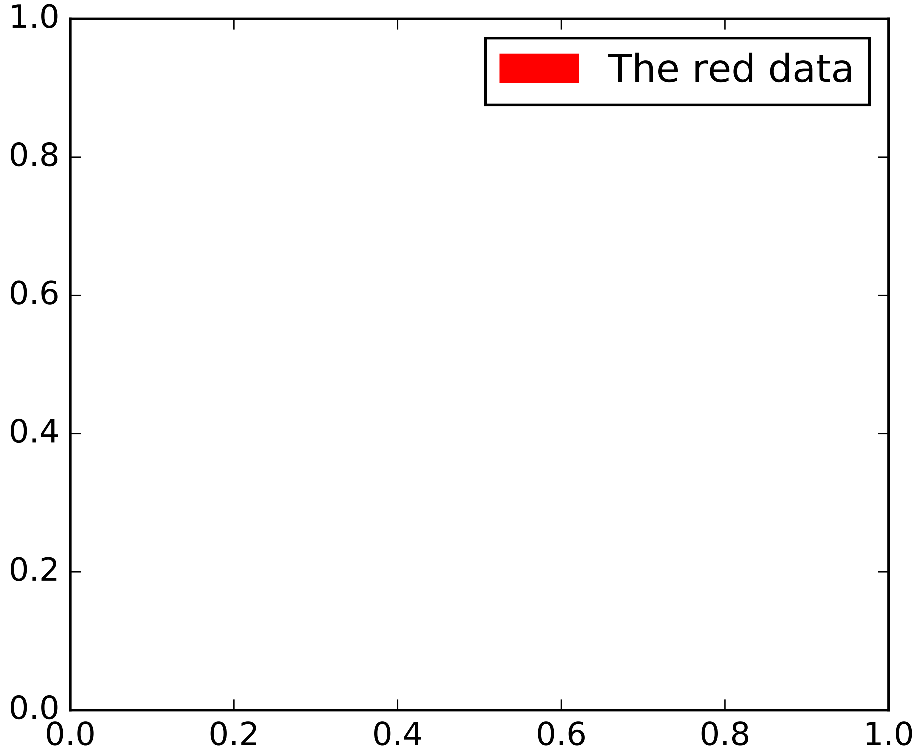
```
line_up, = plt.plot([1,2,3], label='Line 2')
line_down, = plt.plot([3,2,1], label='Line 1')
plt.legend([line_up, line_down], ['Line Up', 'Line Down'])
```

3.6.2 Creating artists specifically for adding to the legend (aka. Proxy artists)

Not all handles can be turned into legend entries automatically, so it is often necessary to create an artist which *can*. Legend handles don't have to exists on the Figure or Axes in order to be used.

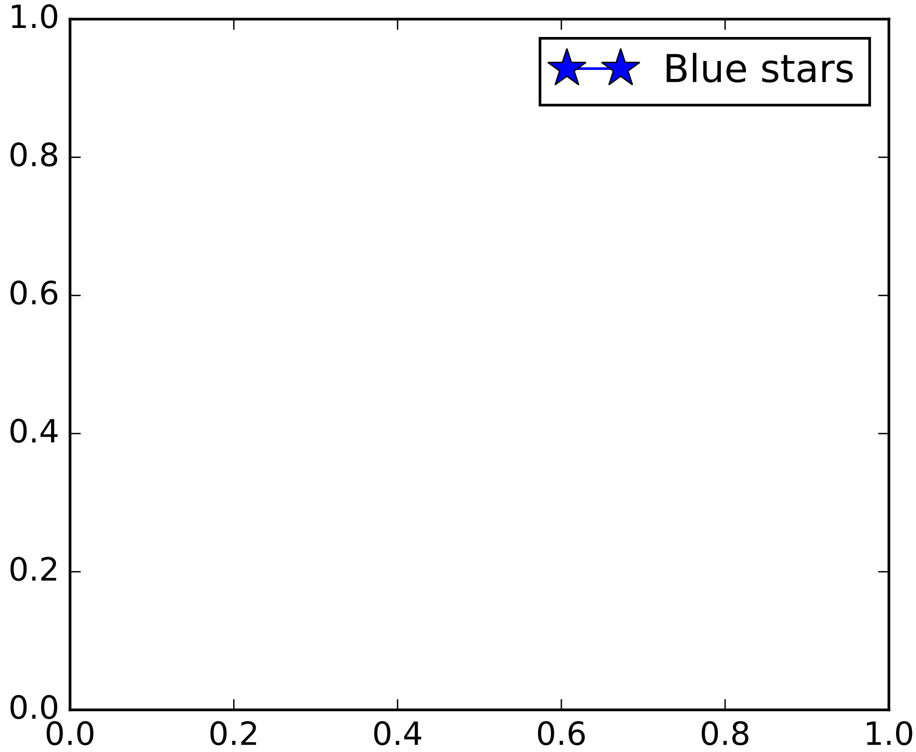
Suppose we wanted to create a legend which has an entry for some data which is represented by a red color:

```
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

red_patch = mpatches.Patch(color='red', label='The red data')
plt.legend(handles=[red_patch])

plt.show()
```



There are many supported legend handles, instead of creating a patch of color we could have created a line with a marker:

```
import matplotlib.lines as mlines
import matplotlib.pyplot as plt

blue_line = mlines.Line2D([], [], color='blue', marker='*',
                          markersize=15, label='Blue stars')
plt.legend(handles=[blue_line])

plt.show()
```



3.6.3 Legend location

The location of the legend can be specified by the keyword argument *loc*. Please see the documentation at `legend()` for more details.

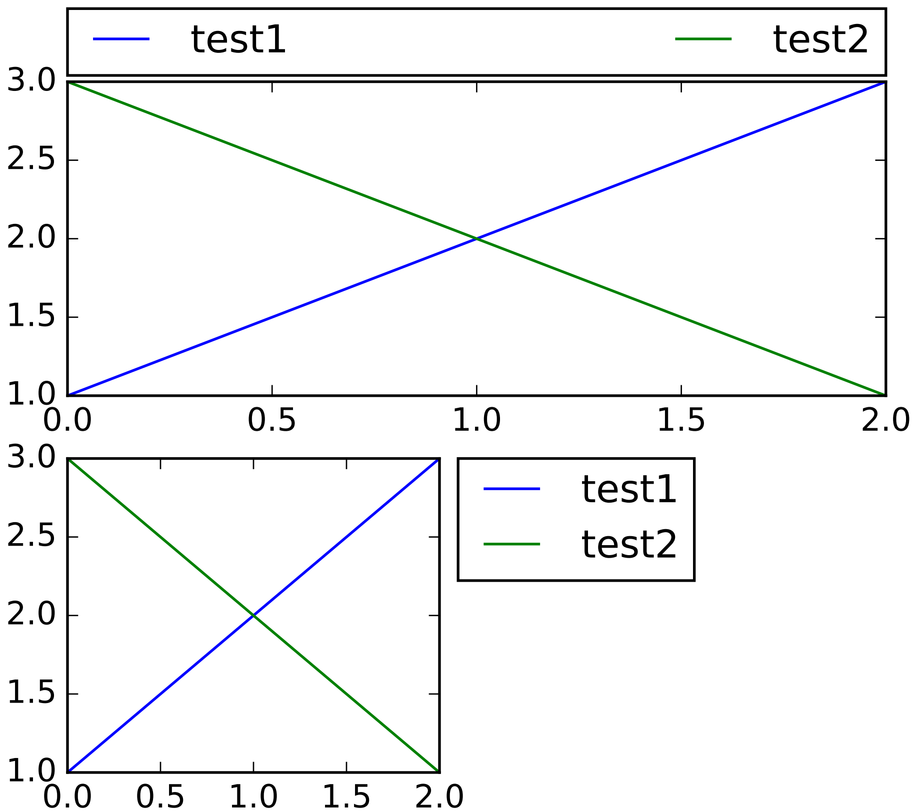
The `bbox_to_anchor` keyword gives a great degree of control for manual legend placement. For example, if you want your axes legend located at the figure's top right-hand corner instead of the axes' corner, simply specify the corner's location, and the coordinate system of that location:

```
plt.legend(bbox_to_anchor=(1, 1),
           bbox_transform=plt.gcf().transFigure)
```

More examples of custom legend placement:

```
import matplotlib.pyplot as plt


plt.subplot(211)
plt.plot([1,2,3], label="test1")
plt.plot([3,2,1], label="test2")
# Place a legend above this subplot, expanding itself to
# fully use the given bounding box.
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
           ncol=2, mode="expand", borderaxespad=0.)
```

```
plt.subplot(223)
plt.plot([1,2,3], label="test1")
plt.plot([3,2,1], label="test2")
# Place a legend to the right of this smaller subplot.
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

plt.show()
```

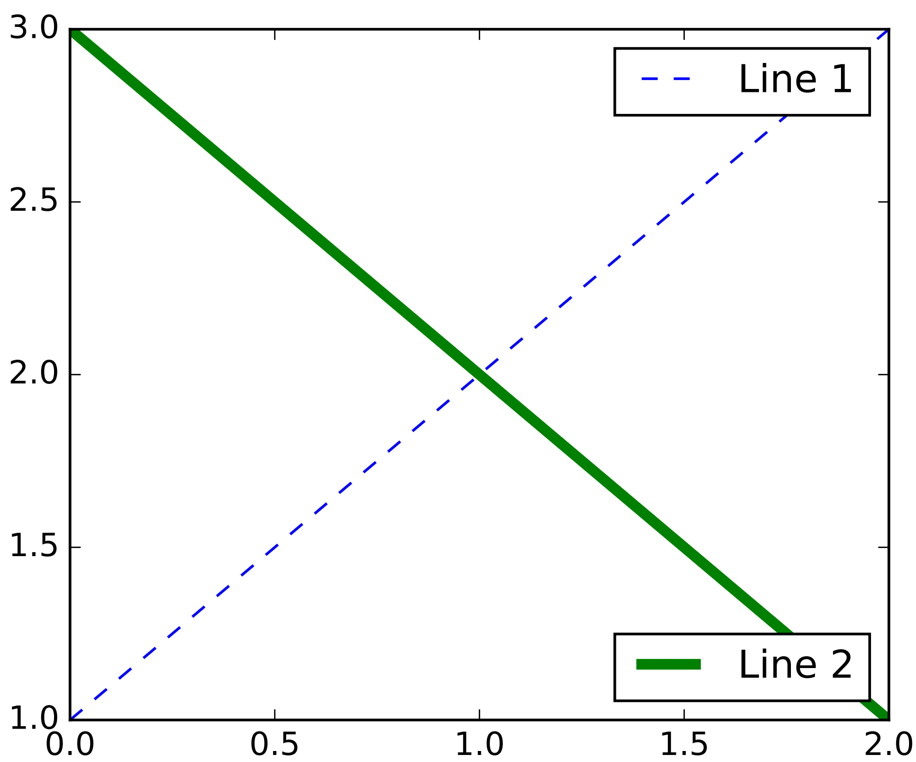


3.6.4 Multiple legends on the same Axes

Sometimes it is more clear to split legend entries across multiple legends. Whilst the instinctive approach to doing this might be to call the *legend()* function multiple times, you will find that only one legend ever exists on the Axes. This has been done so that it is possible to call *legend()* repeatedly to update the legend to the latest handles on the Axes, so to persist old legend instances, we must add them manually to the Axes:

```
import matplotlib.pyplot as plt

line1, = plt.plot([1,2,3], label="Line 1", linestyle='--')
line2, = plt.plot([3,2,1], label="Line 2", linewidth=4)

# Create a legend for the first line.
first_legend = plt.legend(handles=[line1], loc=1)
```

```
# Add the legend manually to the current Axes.
ax = plt.gca().add_artist(first_legend)

# Create another legend for the second line.
plt.legend(handles=[line2], loc=4)

plt.show()
```



3.6.5 Legend Handlers

In order to create legend entries, handles are given as an argument to an appropriate *HandlerBase* subclass. The choice of handler subclass is determined by the following rules:

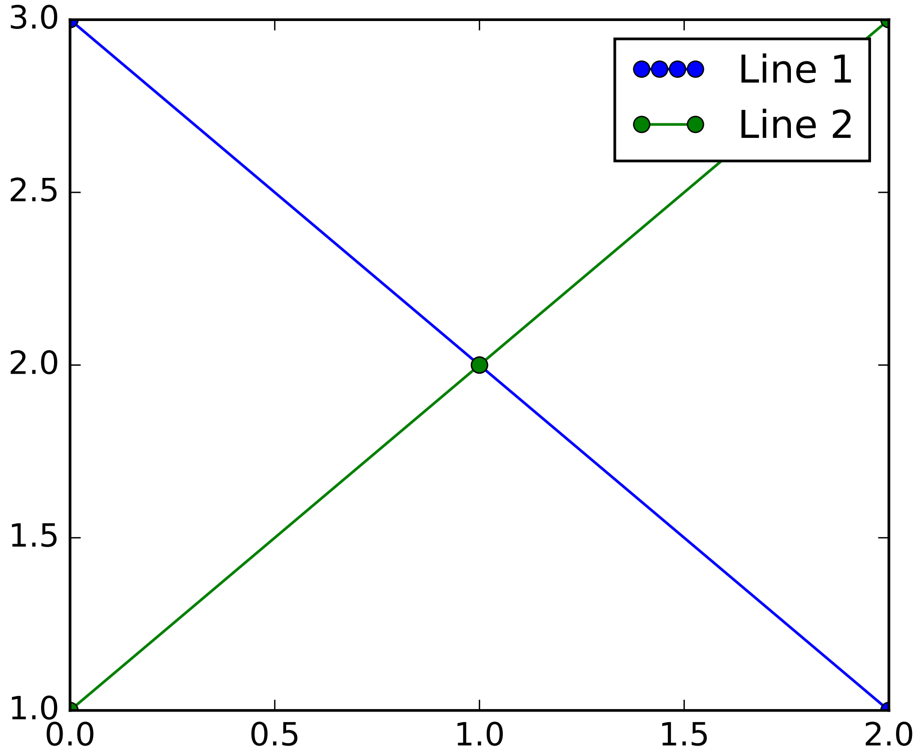
1. Update *get_legend_handler_map()* with the value in the `handler_map` keyword.
2. Check if the `handle` is in the newly created `handler_map`.
3. Check if the type of `handle` is in the newly created `handler_map`.
4. Check if any of the types in the `handle`'s mro is in the newly created `handler_map`.

For completeness, this logic is mostly implemented in *get_legend_handler()*.

All of this flexibility means that we have the necessary hooks to implement custom handlers for our own type of legend key.

The simplest example of using custom handlers is to instantiate one of the existing `HandlerBase` subclasses. For the sake of simplicity, let's choose `matplotlib.legend_handler.HandlerLine2D` which accepts a `numpoints` argument (note numpoints is a keyword on the `legend()` function for convenience). We can then pass the mapping of instance to Handler as a keyword to legend.

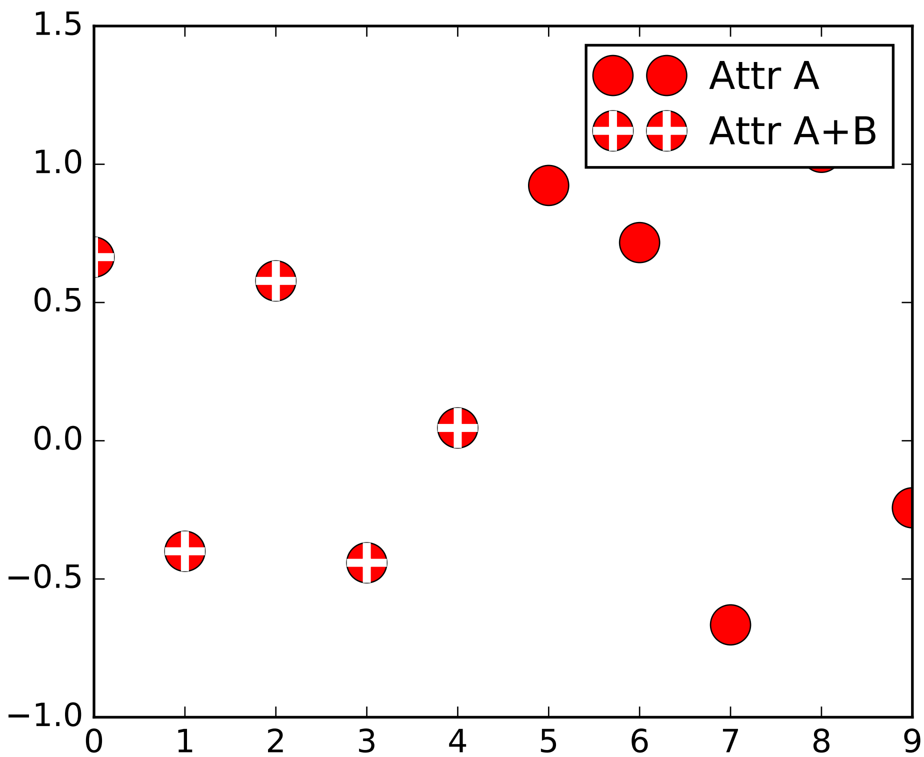
```
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerLine2D

line1, = plt.plot([3,2,1], marker='o', label='Line 1')
line2, = plt.plot([1,2,3], marker='o', label='Line 2')

plt.legend(handler_map={line1: HandlerLine2D(numpoints=4)})
```



As you can see, "Line 1" now has 4 marker points, where "Line 2" has 2 (the default). Try the above code, only change the map's key from `line1` to `type(line1)`. Notice how now both `Line2D` instances get 4 markers.

Along with handlers for complex plot types such as errorbars, stem plots and histograms, the default `handler_map` has a special `tuple` handler (`HandlerTuple`) which simply plots the handles on top of one another for each item in the given tuple. The following example demonstrates combining two legend keys on top of one another:

```
import matplotlib.pyplot as plt
from numpy.random import randn

z = randn(10)

red_dot, = plt.plot(z, "ro", markersize=15)
# Put a white cross over some of the data.
white_cross, = plt.plot(z[:5], "w+", markeredgewidth=3, markersize=15)

plt.legend([red_dot, (red_dot, white_cross)], ["Attr A", "Attr A+B"])
```

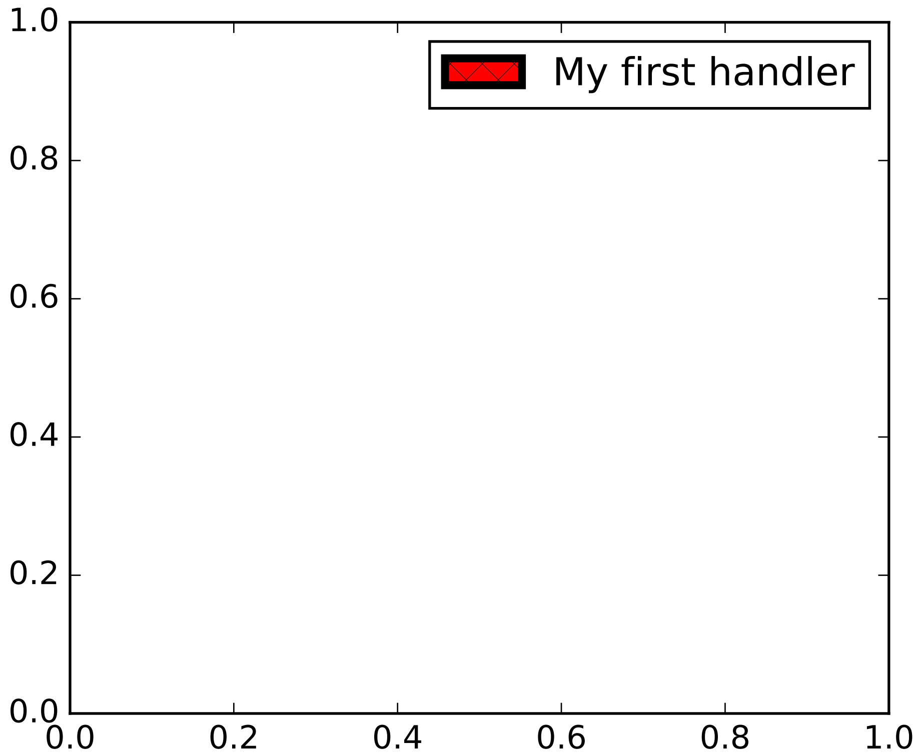


Implementing a custom legend handler

A custom handler can be implemented to turn any handle into a legend key (handles don't necessarily need to be matplotlib artists). The handler must implement a "legend_artist" method which returns a single artist for the legend to use. Signature details about the "legend_artist" are documented at *legend_artist()*.

```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

class AnyObject(object):
    pass
```

```
class AnyObjectHandler(object):
    def legend_artist(self, legend, orig_handle, fontsize, handlebox):
        x0, y0 = handlebox.xdescent, handlebox.ydescent
        width, height = handlebox.width, handlebox.height
        patch = mpatches.Rectangle([x0, y0], width, height, facecolor='red',
                                   edgecolor='black', hatch='xx', lw=3,
                                   transform=handlebox.get_transform())
        handlebox.add_artist(patch)
        return patch

plt.legend([AnyObject()], ['My first handler'],
           handler_map={AnyObject: AnyObjectHandler()})
```



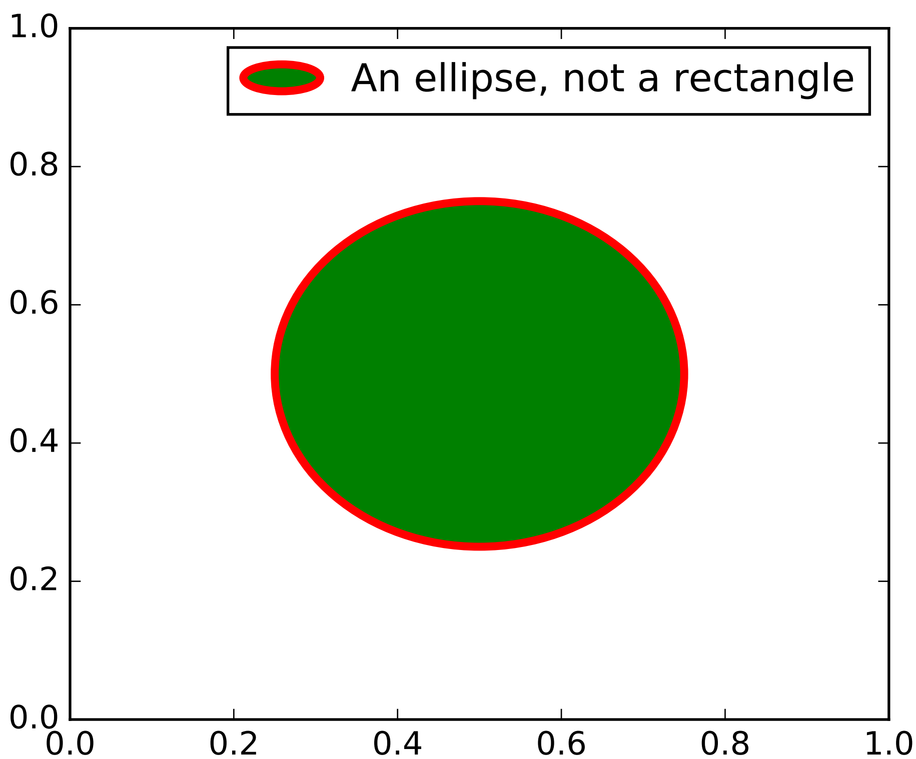
Alternatively, had we wanted to globally accept `AnyObject` instances without needing to manually set the `handler_map` keyword all the time, we could have registered the new handler with:

```
from matplotlib.legend import Legend
Legend.update_default_handler_map({AnyObject: AnyObjectHandler()})
```

Whilst the power here is clear, remember that there are already many handlers implemented and what you want to achieve may already be easily possible with existing classes. For example, to produce elliptical legend keys, rather than rectangular ones:

```
from matplotlib.legend_handler import HandlerPatch
import matplotlib.pyplot as plt
```

```
import matplotlib.patches as mpatches


class HandlerEllipse(HandlerPatch):
    def create_artists(self, legend, orig_handle,
                       xdescent, ydescent, width, height, fontsize, trans):
        center = 0.5 * width - 0.5 * xdescent, 0.5 * height - 0.5 * ydescent
        p = mpatches.Ellipse(xy=center, width=width + xdescent,
                             height=height + ydescent)
        self.update_prop(p, orig_handle, legend)
        p.set_transform(trans)
        return [p]


c = mpatches.Circle((0.5, 0.5), 0.25, facecolor="green",
                    edgecolor="red", linewidth=3)
plt.gca().add_patch(c)

plt.legend([c], ["An ellipse, not a rectangle"],
           handler_map={mpatches.Circle: HandlerEllipse()})
```



3.6.6 Known examples of using legend

Here is a non-exhaustive list of the examples available involving legend being used in various ways:

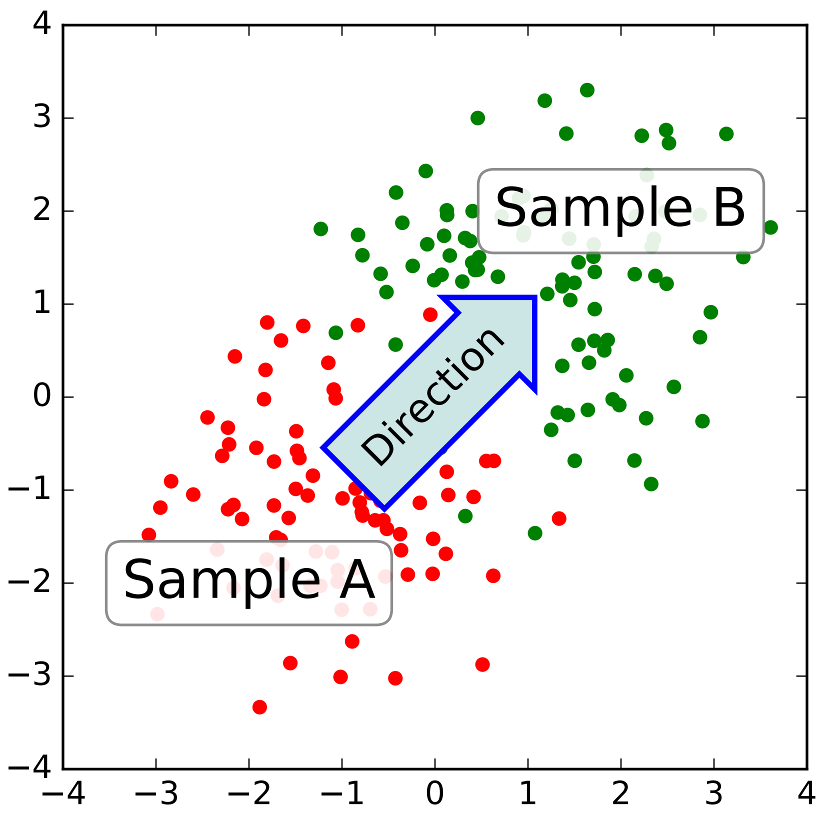
- *lines_bars_and_markers example code: scatter_with_legend.py*
- *api example code: legend_demo.py*
- *pylab_examples example code: contourf_hatching.py*
- *pylab_examples example code: figlegend_demo.py*
- *pylab_examples example code: finance_work2.py*
- *pylab_examples example code: scatter_symbol.py*

3.7 Annotating Axes

Do not proceed unless you already have read *Annotating text*, `text()` and `annotate()`!

3.7.1 Annotating with Text with Box

Let's start with a simple example.



The `text()` function in the pyplot module (or text method of the Axes class) takes bbox keyword argument, and when given, a box around the text is drawn.

```
bbox_props = dict(boxstyle="rarrow,pad=0.3", fc="cyan", ec="b", lw=2)
t = ax.text(0, 0, "Direction", ha="center", va="center", rotation=45,
            size=15,
            bbox=bbox_props)
```

The patch object associated with the text can be accessed by:

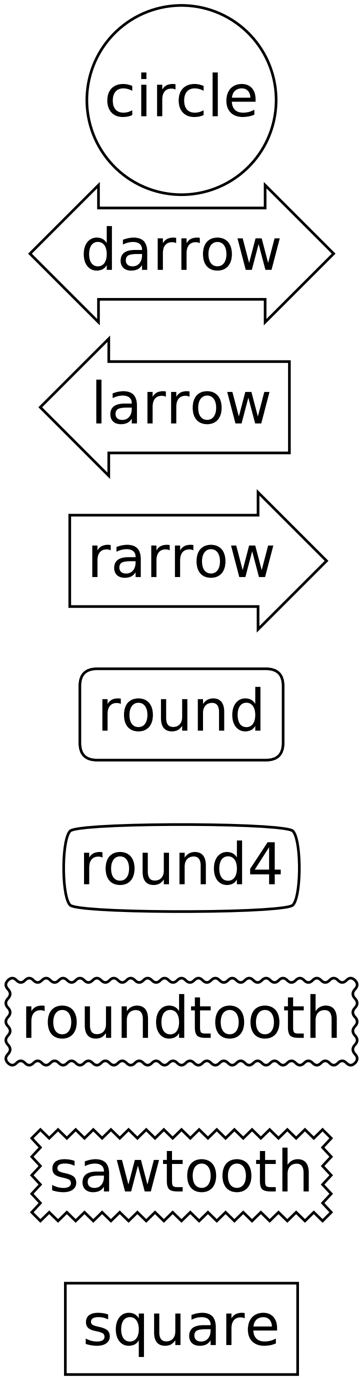
```
bb = t.get_bbox_patch()
```

The return value is an instance of FancyBboxPatch and the patch properties like facecolor, edgewidth, etc. can be accessed and modified as usual. To change the shape of the box, use *set_boxstyle* method.

```
bb.set_boxstyle("rarrow", pad=0.6)
```

The arguments are the name of the box style with its attributes as keyword arguments. Currently, following box styles are implemented.

Class	Name	Attrs
Circle	`circle`	pad=0.3
DArrow	`darrow`	pad=0.3
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3,rounding_size=None
Round4	`round4`	pad=0.3,rounding_size=None
Roundtooth	`roundtooth`	pad=0.3,tooth_size=None
Sawtooth	`sawtooth`	pad=0.3,tooth_size=None
Square	`square`	pad=0.3



Note that the attributes arguments can be specified within the style name with separating comma (this form can be used as "boxstyle" value of bbox argument when initializing the text instance)

```
bb.set_boxstyle("rarrow,pad=0.6")
```

3.7.2 Annotating with Arrow

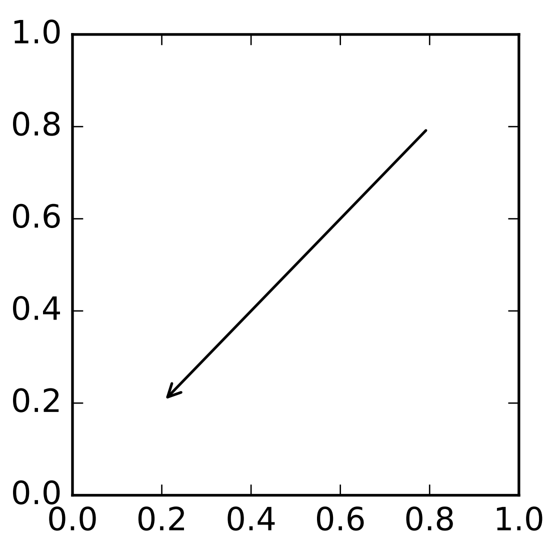
The `annotate()` function in the pyplot module (or annotate method of the Axes class) is used to draw an arrow connecting two points on the plot.

```
ax.annotate("Annotation",
            xy=(x1, y1), xycoords='data',
            xytext=(x2, y2), textcoords='offset points',
            )
```

This annotates a point at `xy` in the given coordinate (`xycoords`) with the text at `xytext` given in `textcoords`. Often, the annotated point is specified in the *data* coordinate and the annotating text in *offset points*. See `annotate()` for available coordinate systems.

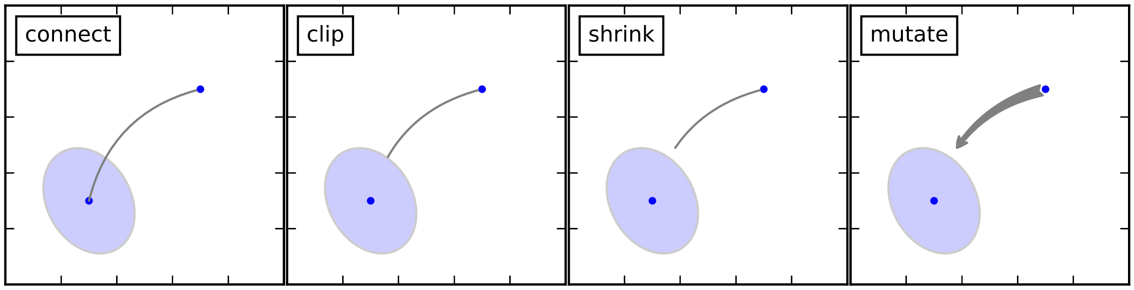
An arrow connecting two point (xy & xytext) can be optionally drawn by specifying the `arrowprops` argument. To draw only an arrow, use empty string as the first argument.

```
ax.annotate("",
            xy=(0.2, 0.2), xycoords='data',
            xytext=(0.8, 0.8), textcoords='data',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3"),
            )
```



The arrow drawing takes a few steps.

1. a connecting path between two points are created. This is controlled by `connectionstyle` key value.
2. If patch object is given (*patchA* & *patchB*), the path is clipped to avoid the patch.
3. The path is further shrunk by given amount of pixels (*shirnkA* & *shrinkB*)
4. The path is transmuted to arrow patch, which is controlled by the `arrowstyle` key value.

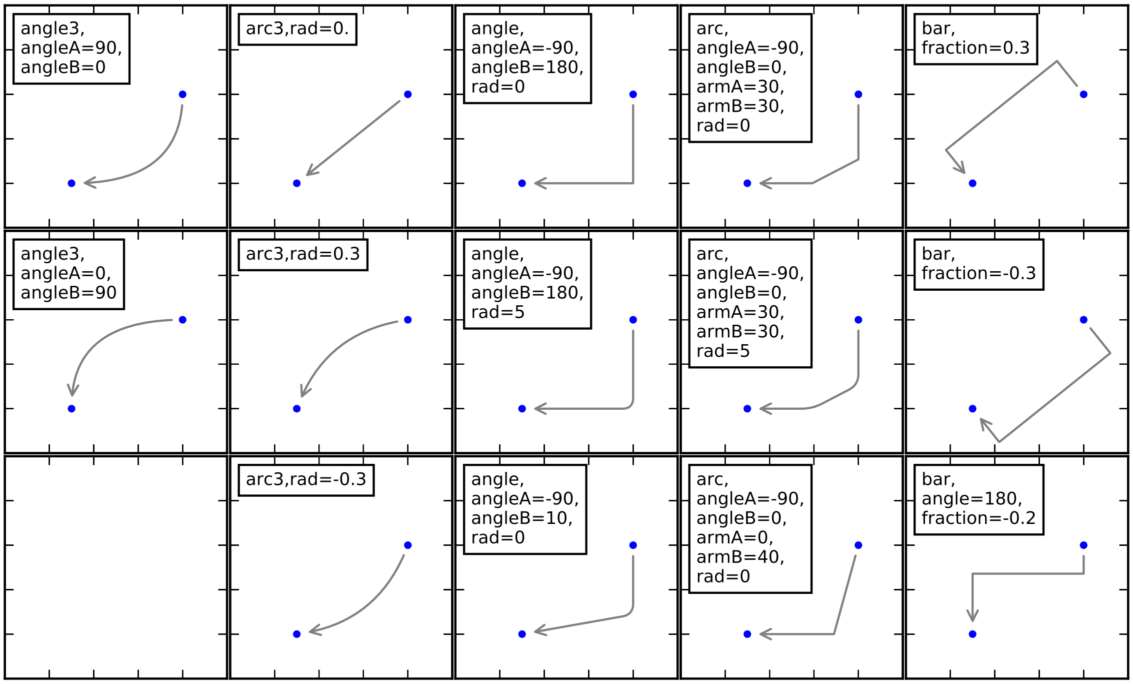


The creation of the connecting path between two points is controlled by `connectionstyle` key and following styles are available.

Name	Attrs
`angle`	angleA=90,angleB=0,rad=0.0
`angle3`	angleA=90,angleB=0
`arc`	angleA=0,angleB=0,armA=None,armB=None,rad=0.0
`arc3`	rad=0.0
`bar`	armA=0.0,armB=0.0,fraction=0.3,angle=None

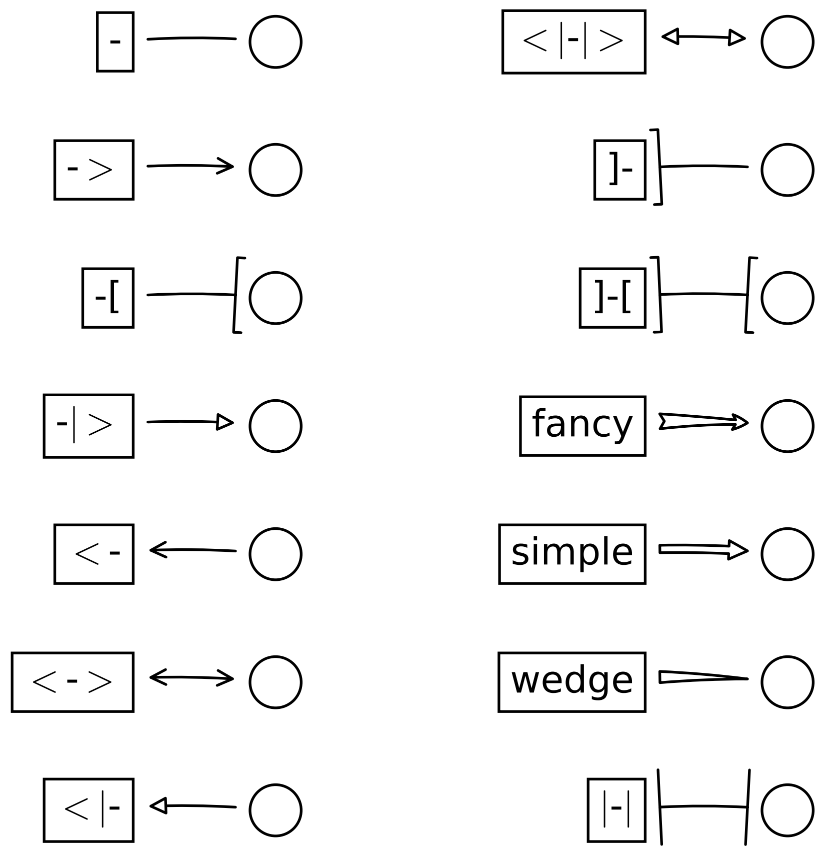
Note that "3" in `angle3` and `arc3` is meant to indicate that the resulting path is a quadratic spline segment (three control points). As will be discussed below, some arrow style option only can be used when the connecting path is a quadratic spline.

The behavior of each connection style is (limitedly) demonstrated in the example below. (Warning : The behavior of the `bar` style is currently not well defined, it may be changed in the future).



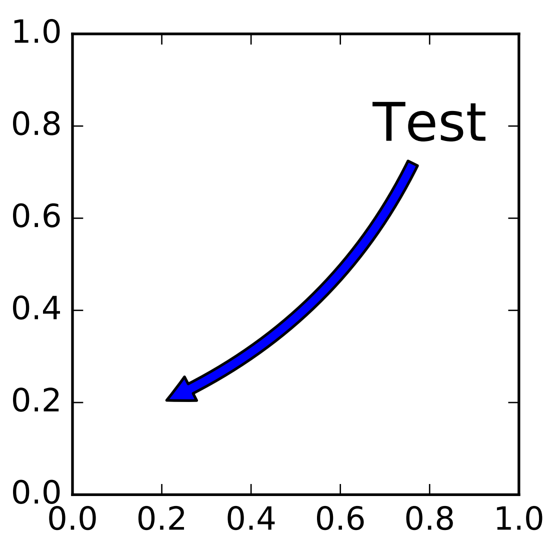
The connecting path (after clipping and shrinking) is then mutated to an arrow patch, according to the given `arrowstyle`.

Name	Attrs
`-`	None
`->`	head_length=0.4,head_width=0.2
`-[`	widthB=1.0,lengthB=0.2,angleB=None
`|-|`	widthA=1.0,widthB=1.0
`-|>`	head_length=0.4,head_width=0.2
`<-`	head_length=0.4,head_width=0.2
`<->`	head_length=0.4,head_width=0.2
`<|-`	head_length=0.4,head_width=0.2
`<|-|>`	head_length=0.4,head_width=0.2
`fancy`	head_length=0.4,head_width=0.4,tail_width=0.4
`simple`	head_length=0.5,head_width=0.5,tail_width=0.2
`wedge`	tail_width=0.3,shrink_factor=0.5

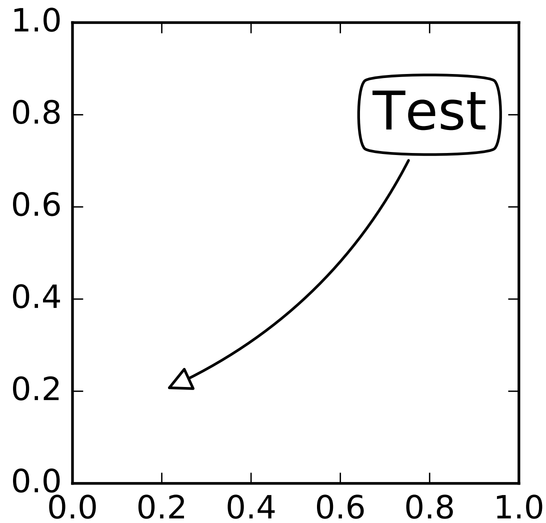


Some arrowstyles only work with connection style that generates a quadratic-spline segment. They are `fancy`, `simple`, and `wedge`. For these arrow styles, you must use "angle3" or "arc3" connection style.

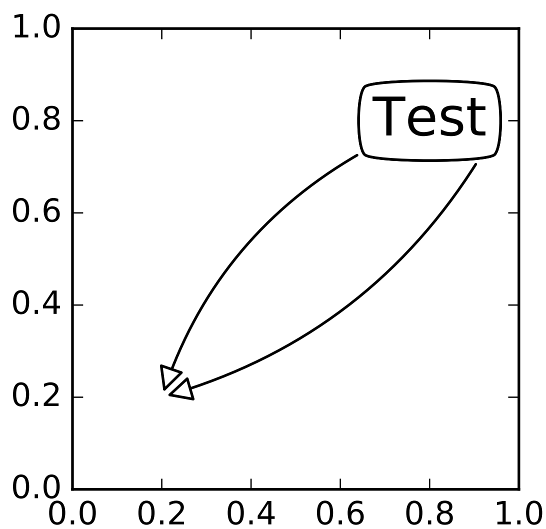
If the annotation string is given, the patchA is set to the bbox patch of the text by default.



As in the text command, a box around the text can be drawn using the bbox argument.



By default, the starting point is set to the center of the text extent. This can be adjusted with relpos key value. The values are normalized to the extent of the text. For example, (0,0) means lower-left corner and (1,1) means top-right.

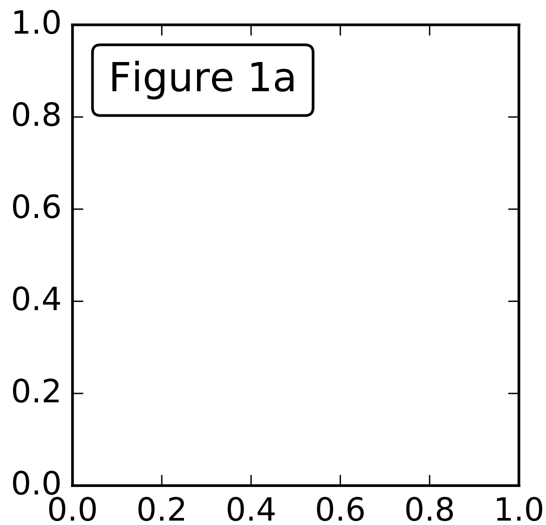


3.7.3 Placing Artist at the anchored location of the Axes

There are class of artist that can be placed at the anchored location of the Axes. A common example is the legend. This type of artists can be created by using the OffsetBox class. A few predefined classes are available in mpl_toolkits.axes_grid.anchored_artists.

```
from mpl_toolkits.axes_grid.anchored_artists import AnchoredText
at = AnchoredText("Figure 1a",
```

```
                  prop=dict(size=8), frameon=True,
                  loc=2,
                  )
at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
ax.add_artist(at)
```

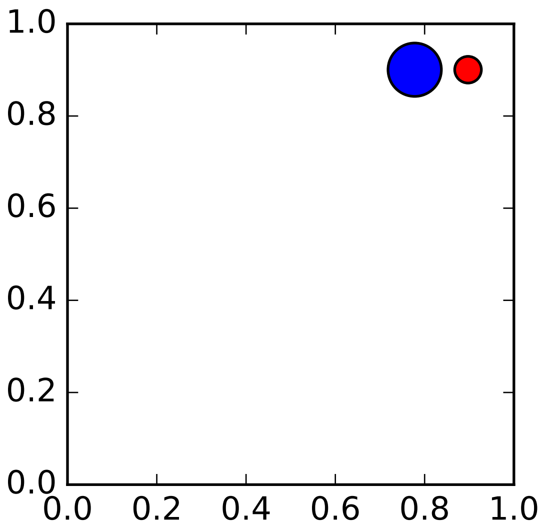


The *loc* keyword has same meaning as in the legend command.

A simple application is when the size of the artist (or collection of artists) is known in pixel size during the time of creation. For example, If you want to draw a circle with fixed size of 20 pixel x 20 pixel (radius = 10 pixel), you can utilize `AnchoredDrawingArea`. The instance is created with a size of the drawing area (in pixel). And user can add arbitrary artist to the drawing area. Note that the extents of the artists that are added to the drawing area has nothing to do with the placement of the drawing area itself. The initial size only matters.

```
from mpl_toolkits.axes_grid.anchored_artists import AnchoredDrawingArea

ada = AnchoredDrawingArea(20, 20, 0, 0,
                          loc=1, pad=0., frameon=False)
p1 = Circle((10, 10), 10)
ada.drawing_area.add_artist(p1)
p2 = Circle((30, 10), 5, fc="r")
ada.drawing_area.add_artist(p2)
```

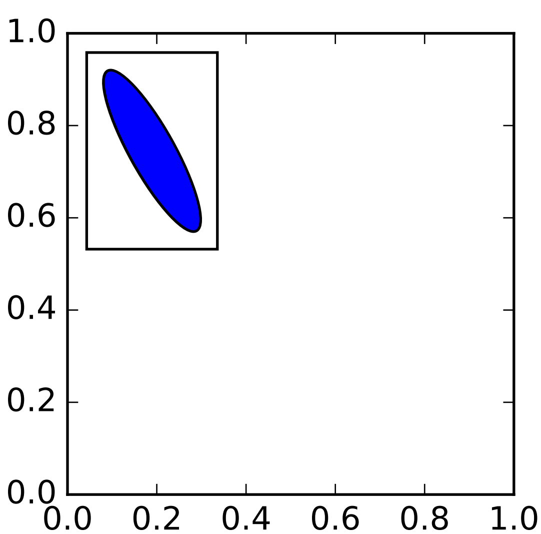
The artists that are added to the drawing area should not have transform set (they will be overridden) and the dimension of those artists are interpreted as a pixel coordinate, i.e., the radius of the circles in above example are 10 pixel and 5 pixel, respectively.



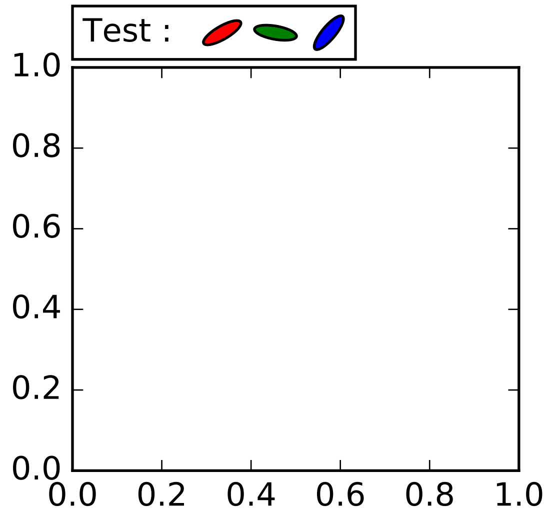
Sometimes, you want to your artists scale with data coordinate (or other coordinate than canvas pixel). You can use `AnchoredAuxTransformBox` class. This is similar to `AnchoredDrawingArea` except that the extent of the artist is determined during the drawing time respecting the specified transform.

```
from mpl_toolkits.axes_grid.anchored_artists import AnchoredAuxTransformBox

box = AnchoredAuxTransformBox(ax.transData, loc=2)
el = Ellipse((0,0), width=0.1, height=0.4, angle=30) # in data coordinates!
box.drawing_area.add_artist(el)
```

The ellipse in the above example will have width and height corresponds to 0.1 and 0.4 in data coordinate and will be automatically scaled when the view limits of the axes change.



As in the legend, the bbox_to_anchor argument can be set. Using the HPacker and VPacker, you can have an arrangement(?) of artist as in the legend (as a matter of fact, this is how the legend is created).



Note that unlike the legend, the `bbox_transform` is set to IdentityTransform by default.

3.7.4 Using Complex Coordinate with Annotation

The Annotation in matplotlib support several types of coordinate as described in *Annotating text*. For an advanced user who wants more control, it supports a few other options.

1. `Transform` instance. For example,

```
ax.annotate("Test", xy=(0.5, 0.5), xycoords=ax.transAxes)
```

   is identical to

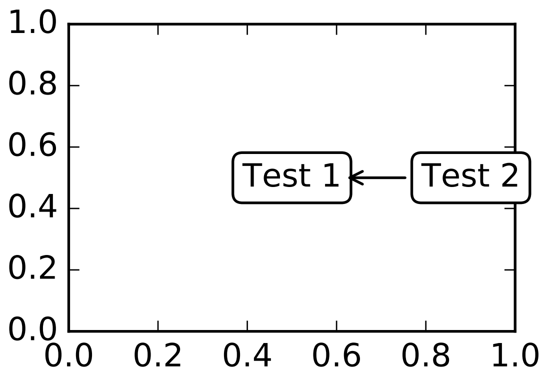
```
ax.annotate("Test", xy=(0.5, 0.5), xycoords="axes fraction")
```

   With this, you can annotate a point in other axes.

```
ax1, ax2 = subplot(121), subplot(122)
ax2.annotate("Test", xy=(0.5, 0.5), xycoords=ax1.transData,
             xytext=(0.5, 0.5), textcoords=ax2.transData,
             arrowprops=dict(arrowstyle="->"))
```

2. `Artist` instance. The xy value (or xytext) is interpreted as a fractional coordinate of the bbox (return value of *get_window_extent*) of the artist.

```
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1, # (1,0.5) of the an1's bbox
                  xytext=(30,0), textcoords="offset points",
                  va="center", ha="left",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))
```



Note that it is your responsibility that the extent of the coordinate artist (*an1* in above example) is determined before *an2* gets drawn. In most cases, it means that an2 needs to be drawn later than *an1*.

3. A callable object that returns an instance of either `BboxBase` or `Transform`. If a transform is returned, it is same as 1 and if bbox is returned, it is same as 2. The callable object should take a single argument of renderer instance. For example, following two commands give identical results

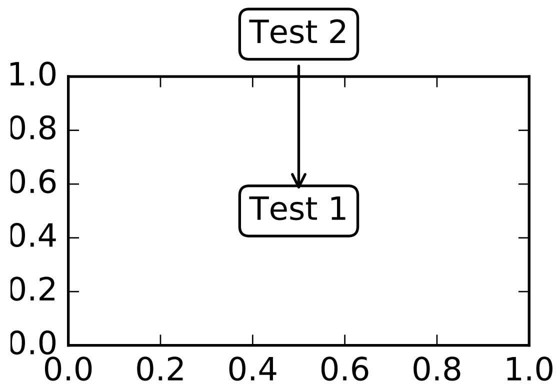
```
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1,
                  xytext=(30,0), textcoords="offset points")
an2 = ax.annotate("Test 2", xy=(1, 0.5), xycoords=an1.get_window_extent,
                  xytext=(30,0), textcoords="offset points")
```

4. A tuple of two coordinate specification. The first item is for x-coordinate and the second is for y-coordinate. For example,

```
annotate("Test", xy=(0.5, 1), xycoords=("data", "axes fraction"))
```

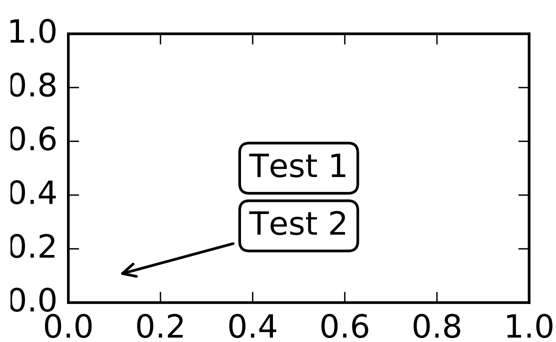
0.5 is in data coordinate, and 1 is in normalized axes coordinate. You may use an artist or transform as with a tuple. For example,

```
import matplotlib.pyplot as plt

plt.figure(figsize=(3,2))
ax=plt.axes([0.1, 0.1, 0.8, 0.7])
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))

an2 = ax.annotate("Test 2", xy=(0.5, 1.), xycoords=an1,
                  xytext=(0.5,1.1), textcoords=(an1, "axes fraction"),
                  va="bottom", ha="center",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))
plt.show()
```



5. Sometimes, you want your annotation with some "offset points", but not from the annotated point but from other point. `OffsetFrom` is a helper class for such case.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(3,2))
ax=plt.axes([0.1, 0.1, 0.8, 0.7])
an1 = ax.annotate("Test 1", xy=(0.5, 0.5), xycoords="data",
                  va="center", ha="center",
                  bbox=dict(boxstyle="round", fc="w"))

from matplotlib.text import OffsetFrom
offset_from = OffsetFrom(an1, (0.5, 0))
an2 = ax.annotate("Test 2", xy=(0.1, 0.1), xycoords="data",
                  xytext=(0, -10), textcoords=offset_from,
                  # xytext is offset points from "xy=(0.5, 0), xycoords=an1"
                  va="top", ha="center",
                  bbox=dict(boxstyle="round", fc="w"),
                  arrowprops=dict(arrowstyle="->"))
plt.show()
```



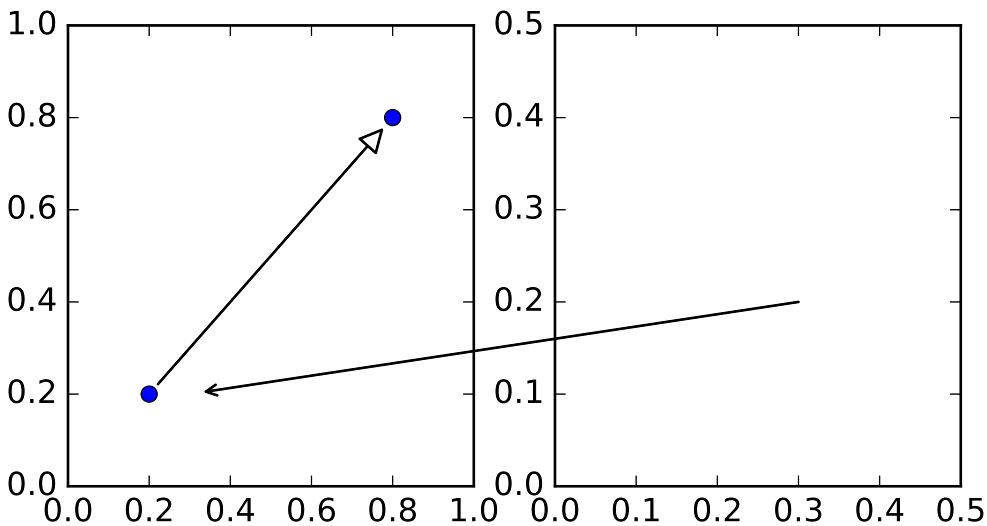
You may take a look at this example *pylab_examples example code: annotation_demo3.py*.

3.7.5 Using ConnectorPatch

The ConnectorPatch is like an annotation without a text. While the annotate function is recommended in most of situation, the ConnectorPatch is useful when you want to connect points in different axes.

```
from matplotlib.patches import ConnectionPatch
xy = (0.2, 0.2)
con = ConnectionPatch(xyA=xy, xyB=xy, coordsA="data", coordsB="data",
                      axesA=ax1, axesB=ax2)
ax2.add_artist(con)
```

The above code connects point xy in data coordinate of `ax1` to point xy int data coordinate of `ax2`. Here is a simple example.

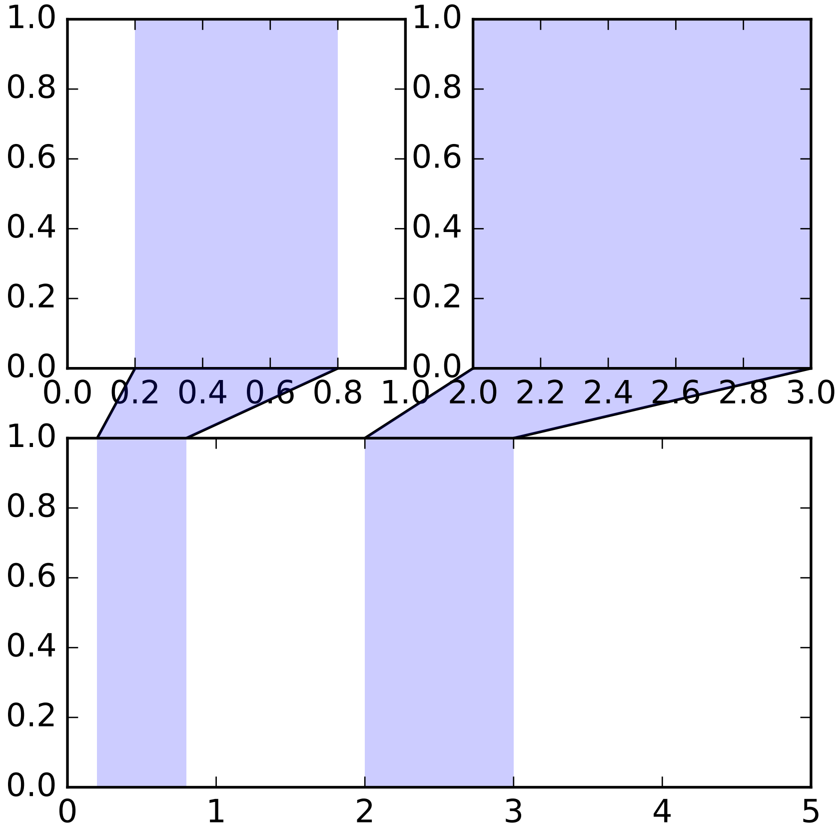


While the ConnectorPatch instance can be added to any axes, but you may want it to be added to the axes in the latter (?) of the axes drawing order to prevent overlap (?) by other axes.

Advanced Topics

3.7.6 Zoom effect between Axes

mpl_toolkits.axes_grid.inset_locator defines some patch classes useful for interconnect two axes. Understanding the code requires some knowledge of how mpl's transform works. But, utilizing it will be straight forward.

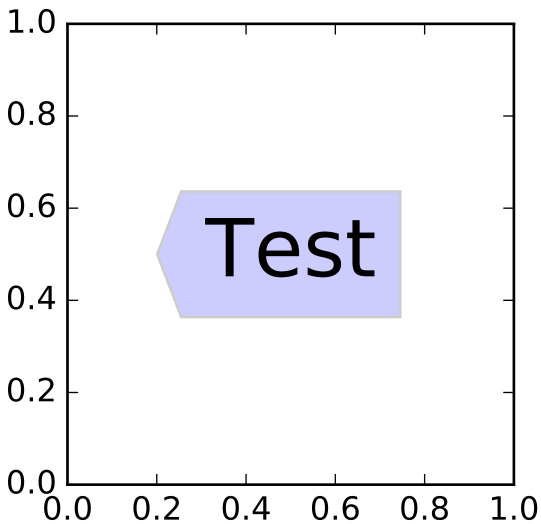


3.7.7 Define Custom BoxStyle

You can use a custom box style. The value for the `boxstyle` can be a callable object in following forms.:

```
def __call__(self, x0, y0, width, height, mutation_size,
             aspect_ratio=1.):
    """
    Given the location and size of the box, return the path of
    the box around it.

      - *x0*, *y0*, *width*, *height* : location and size of the box
      - *mutation_size* : a reference scale for the mutation.
      - *aspect_ratio* : aspect-ration for the mutation.
    """
    path = ...
    return path
```

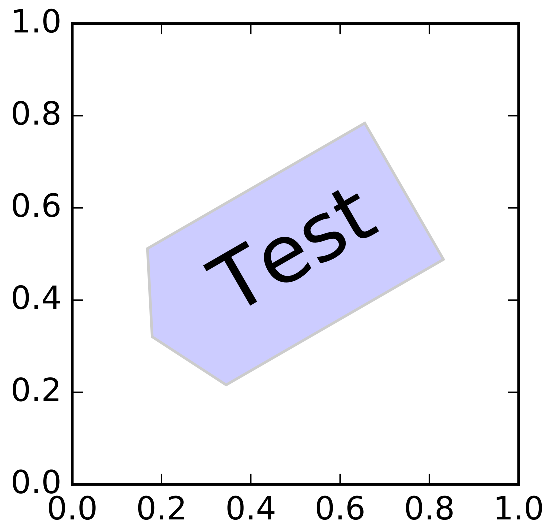
Here is a complete example.



However, it is recommended that you derive from the matplotlib.patches.BoxStyle._Base as demonstrated below.

```
from matplotlib.path import Path
from matplotlib.patches import BoxStyle
import matplotlib.pyplot as plt

# we may derive from matplotlib.patches.BoxStyle._Base class.
# You need to overide transmute method in this case.

class MyStyle(BoxStyle._Base):
    """
    A simple box.
    """

    def __init__(self, pad=0.3):
        """
        The arguments need to be floating numbers and need to have
        default values.

         *pad*
            amount of padding
        """

        self.pad = pad
        super(MyStyle, self).__init__()

    def transmute(self, x0, y0, width, height, mutation_size):
        """
        Given the location and size of the box, return the path of
        the box around it.

          - *x0*, *y0*, *width*, *height* : location and size of the box
          - *mutation_size* : a reference scale for the mutation.
```

```
        Often, the *mutation_size* is the font size of the text.
        You don't need to worry about the rotation as it is
        automatically taken care of.
        """

        # padding
        pad = mutation_size * self.pad

        # width and height with padding added.
        width, height = width + 2.*pad, \
                        height + 2.*pad,

        # boundary of the padded box
        x0, y0 = x0-pad, y0-pad,
        x1, y1 = x0+width, y0 + height

        cp = [(x0, y0),
              (x1, y0), (x1, y1), (x0, y1),
              (x0-pad, (y0+y1)/2.), (x0, y0),
              (x0, y0)]

        com = [Path.MOVETO,
               Path.LINETO, Path.LINETO, Path.LINETO,
               Path.LINETO, Path.LINETO,
               Path.CLOSEPOLY]

        path = Path(cp, com)

        return path


# register the custom style
BoxStyle._style_list["angled"] = MyStyle

plt.figure(1, figsize=(3,3))
ax = plt.subplot(111)
ax.text(0.5, 0.5, "Test", size=30, va="center", ha="center", rotation=30,
        bbox=dict(boxstyle="angled,pad=0.5", alpha=0.2))

del BoxStyle._style_list["angled"]

plt.show()
```



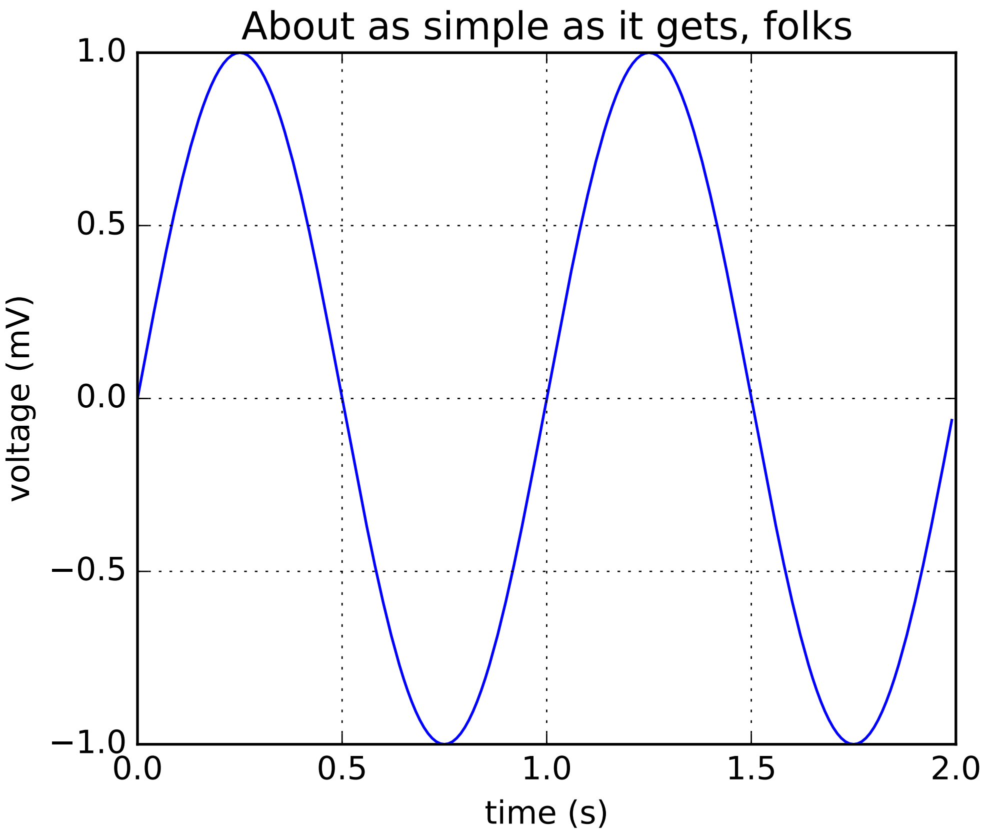
Similarly, you can define custom ConnectionStyle and custom ArrowStyle. See the source code of `lib/matplotlib/patches.py` and check how each style class is defined.

3.8 Screenshots

Here you'll find a host of example plots with the code that generated them.

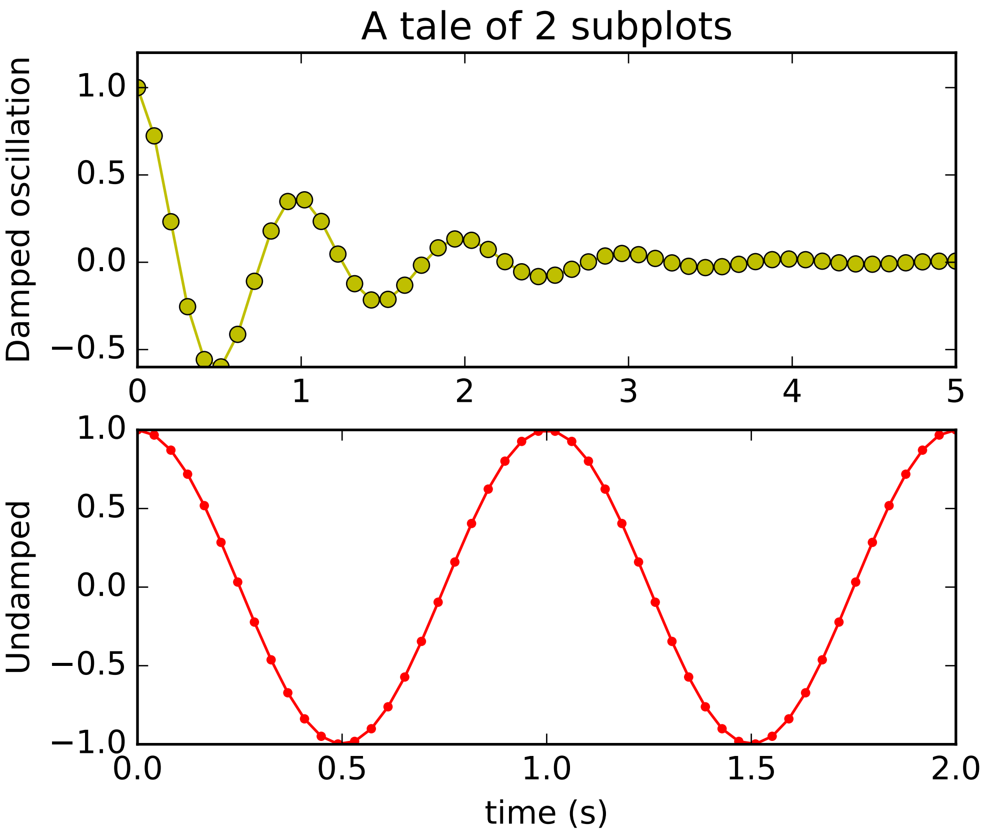
3.8.1 Simple Plot

Here's a very basic `plot()` with text labels:



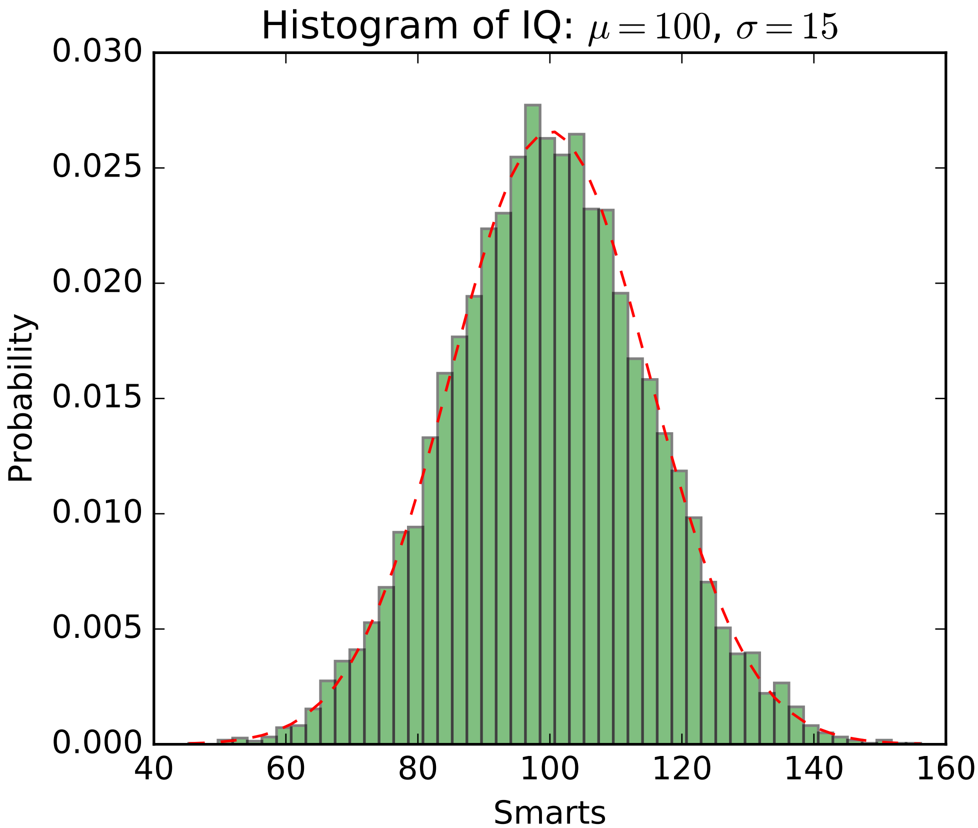
3.8.2 Subplot demo

Multiple axes (i.e. subplots) are created with the `subplot()` command:



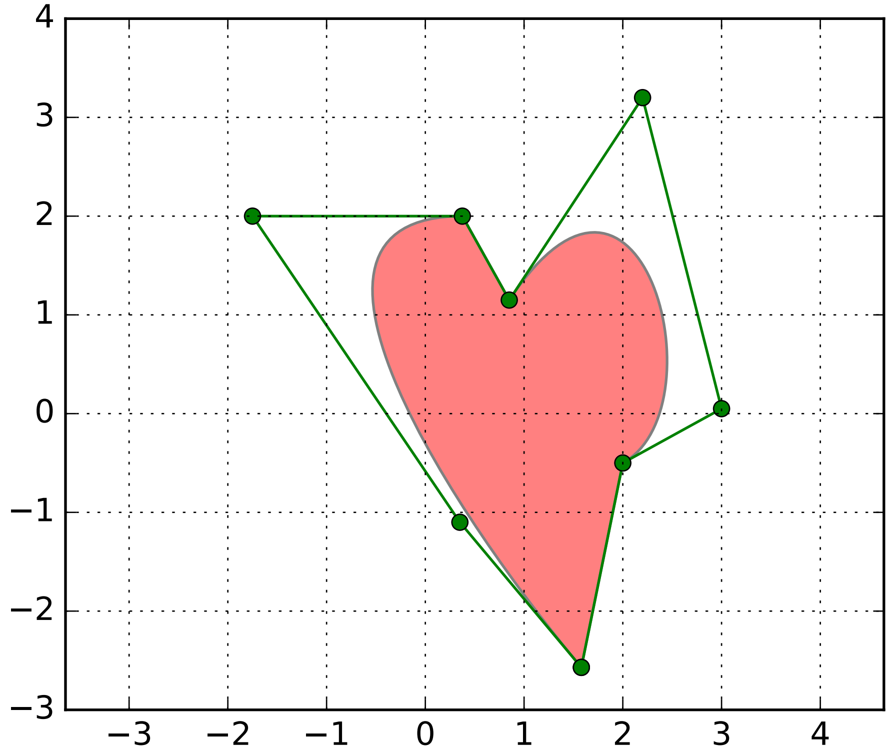
3.8.3 Histograms

The `hist()` command automatically generates histograms and returns the bin counts or probabilities:



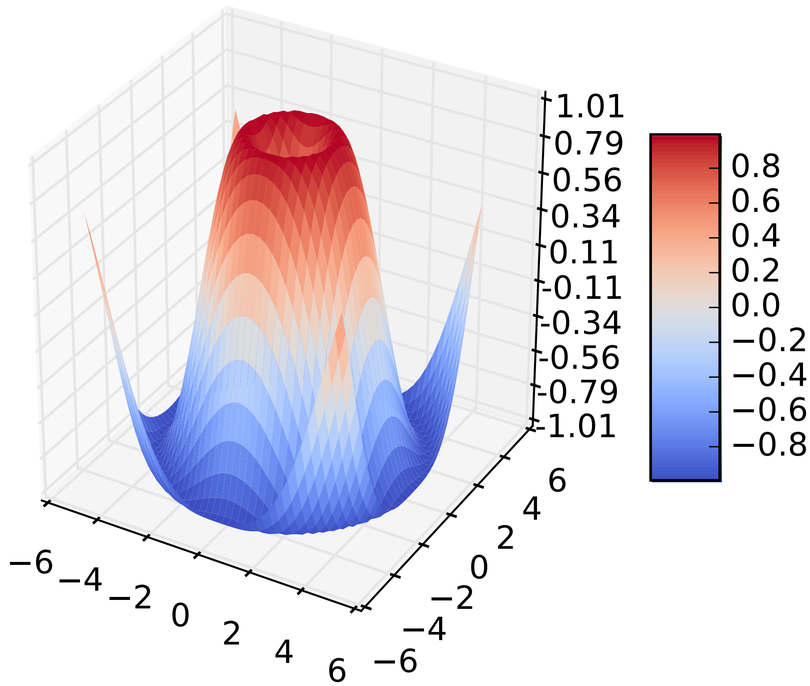
3.8.4 Path demo

You can add arbitrary paths in matplotlib using the `matplotlib.path` module:



3.8.5 mplot3d

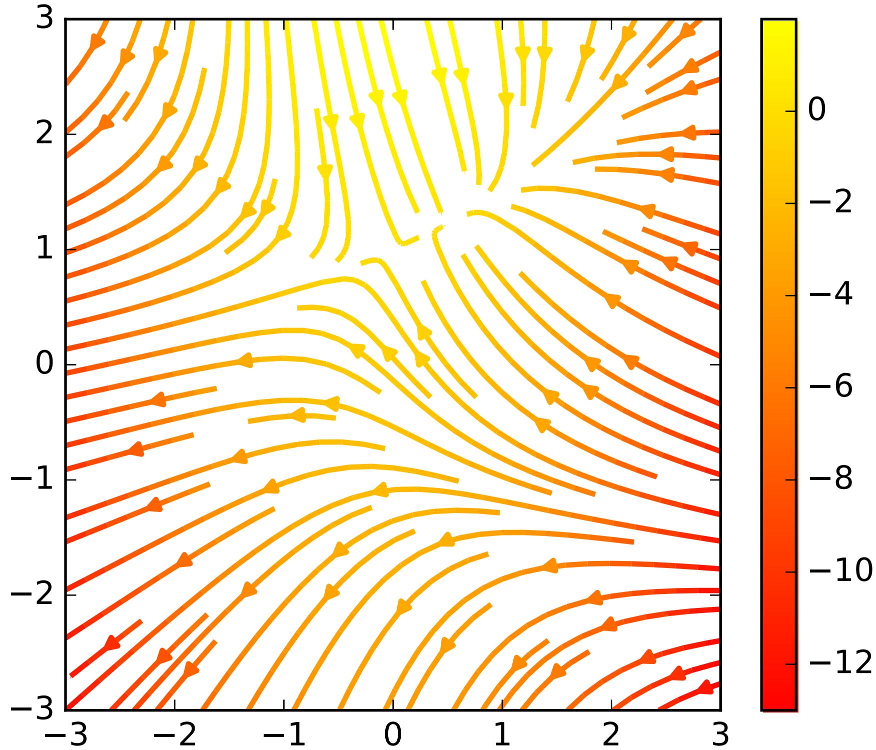
The mplot3d toolkit (see *mplot3d tutorial* and *mplot3d Examples*) has support for simple 3d graphs including surface, wireframe, scatter, and bar charts.

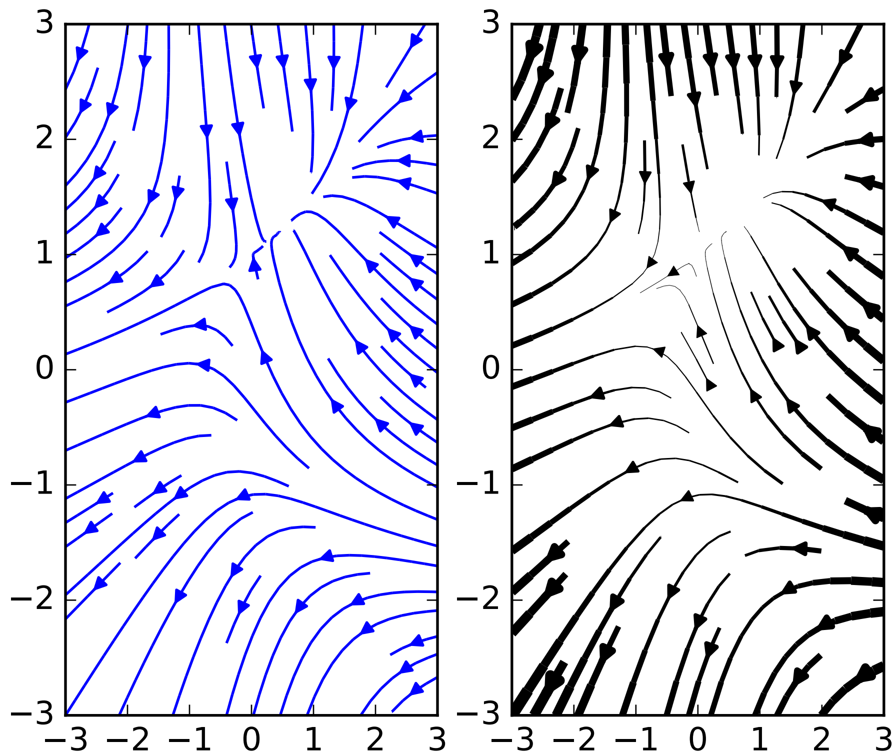


Thanks to John Porter, Jonathon Taylor, Reinier Heeres, and Ben Root for the `mplot3d` toolkit. This toolkit is included with all standard matplotlib installs.

3.8.6 Streamplot

The *streamplot()* function plots the streamlines of a vector field. In addition to simply plotting the streamlines, it allows you to map the colors and/or line widths of streamlines to a separate parameter, such as the speed or local intensity of the vector field.

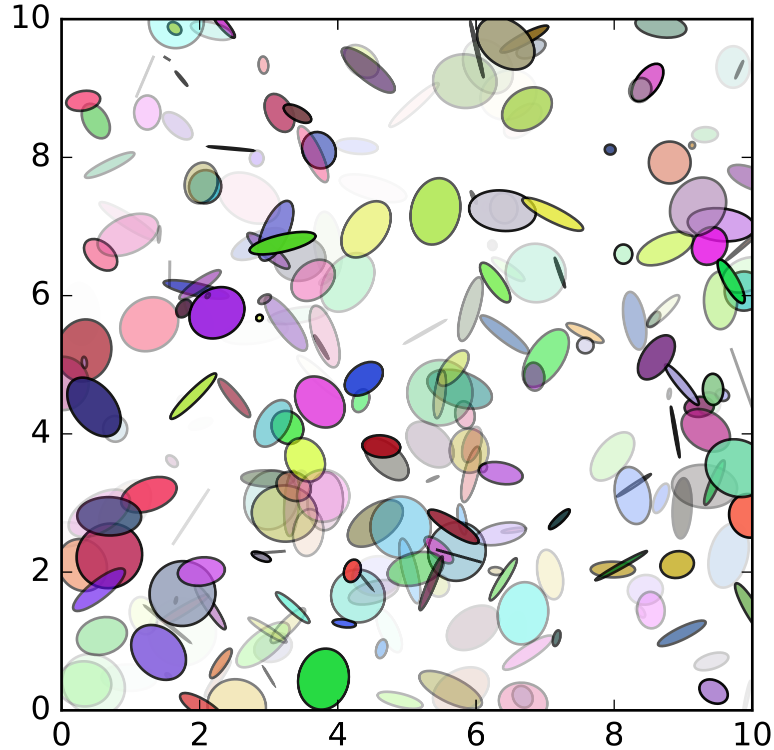

3
2
1
0
−1
−2
−3
−3 −2 −1 0 1 2 3
0
−2
−4
−6
−8
−10
−12



This feature complements the `quiver()` function for plotting vector fields. Thanks to Tom Flannaghan and Tony Yu for adding the streamplot function.

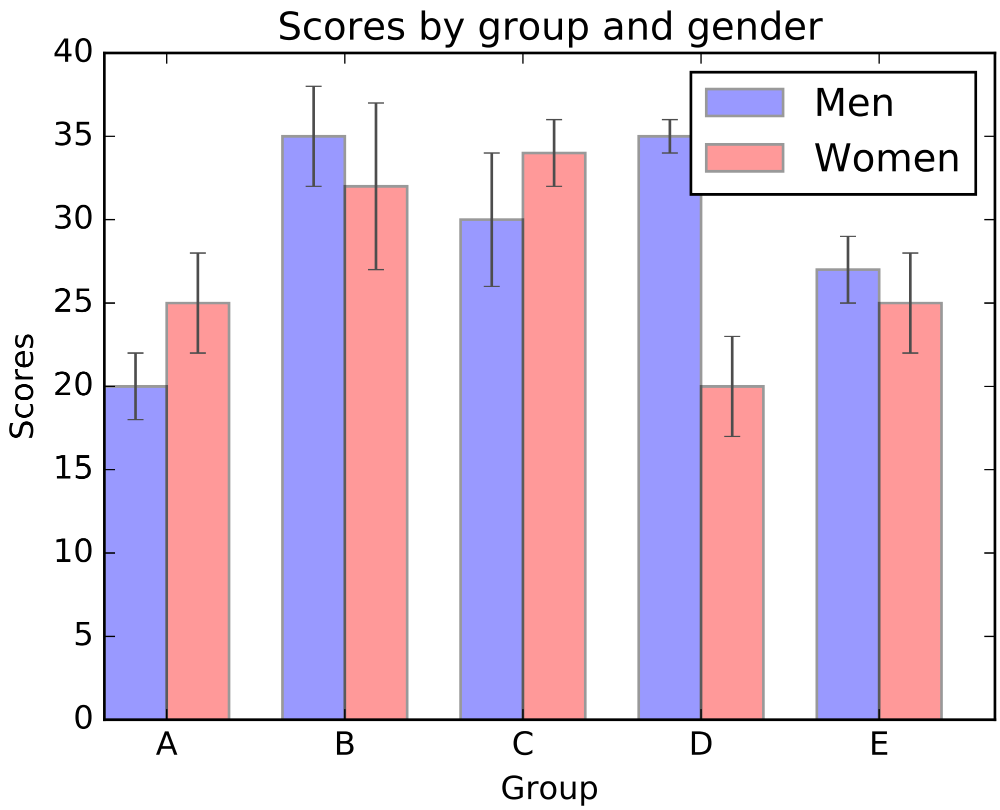
3.8.7 Ellipses

In support of the Phoenix mission to Mars (which used matplotlib to display ground tracking of spacecraft), Michael Droettboom built on work by Charlie Moad to provide an extremely accurate 8-spline approximation to elliptical arcs (see `Arc`), which are insensitive to zoom level.



3.8.8 Bar charts

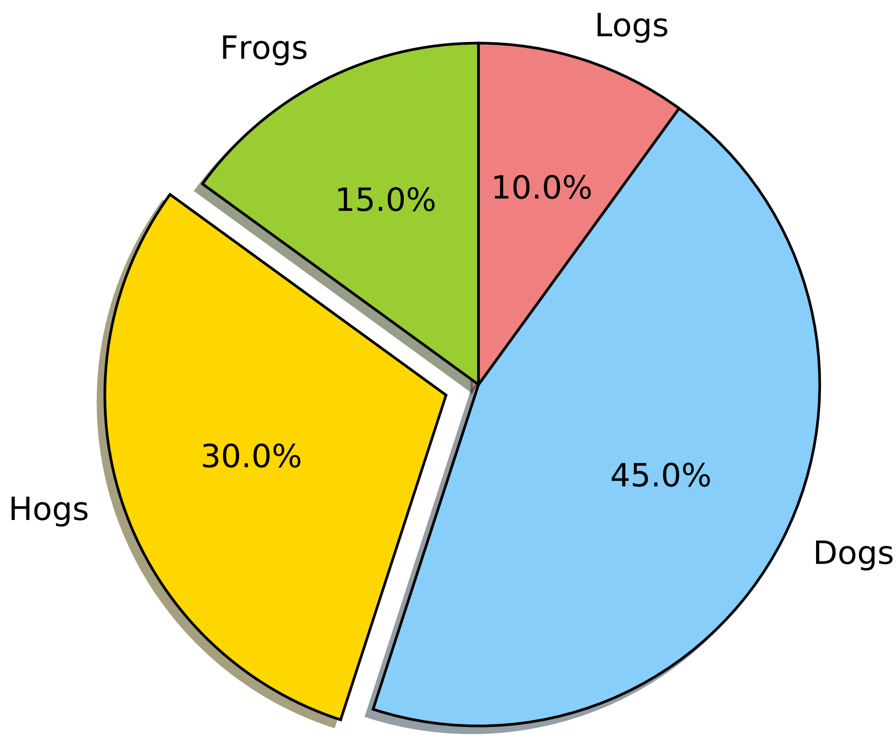
Bar charts are simple to create using the `bar()` command, which includes customizations such as error bars:

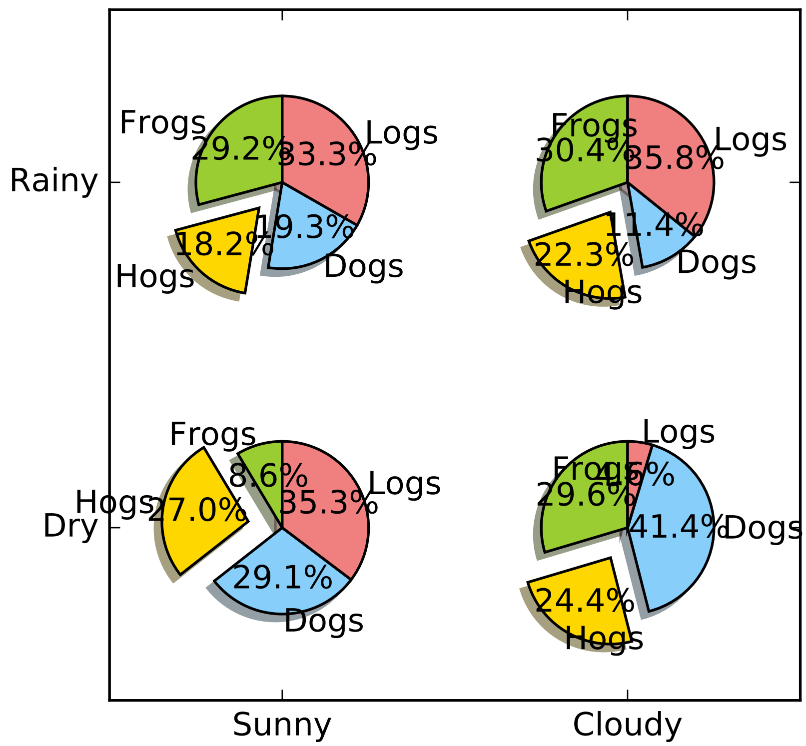


It's also simple to create stacked bars (bar_stacked.py), candlestick bars (finance_demo.py), and horizontal bar charts (barh_demo.py).

3.8.9 Pie charts

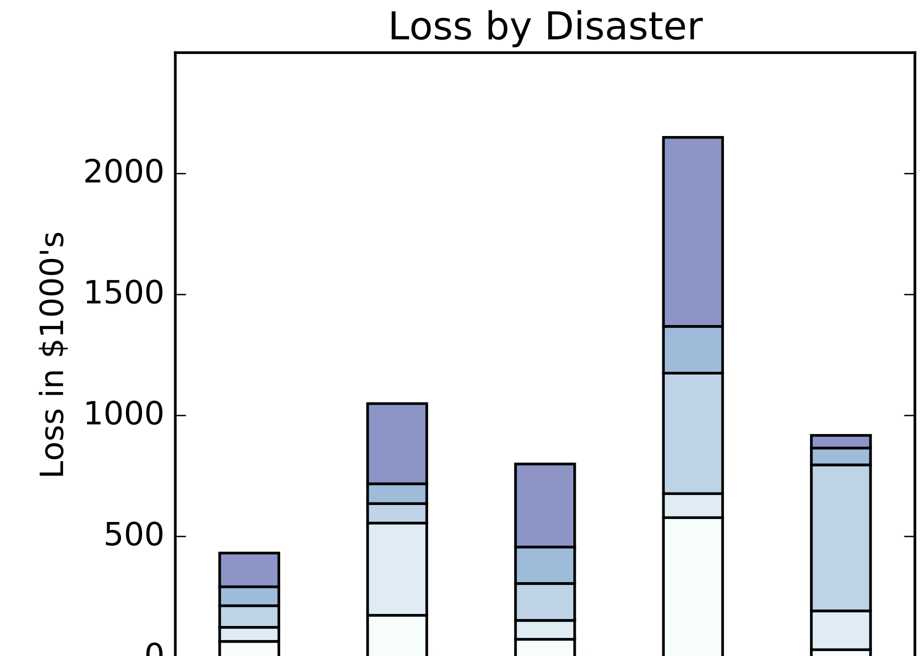
The `pie()` command allows you to easily create pie charts. Optional features include auto-labeling the percentage of area, exploding one or more wedges from the center of the pie, and a shadow effect. Take a close look at the attached code, which generates this figure in just a few lines of code.


Frogs
Logs
15.0%
10.0%
30.0%
Hogs
45.0%
Dogs



3.8.10 Table demo

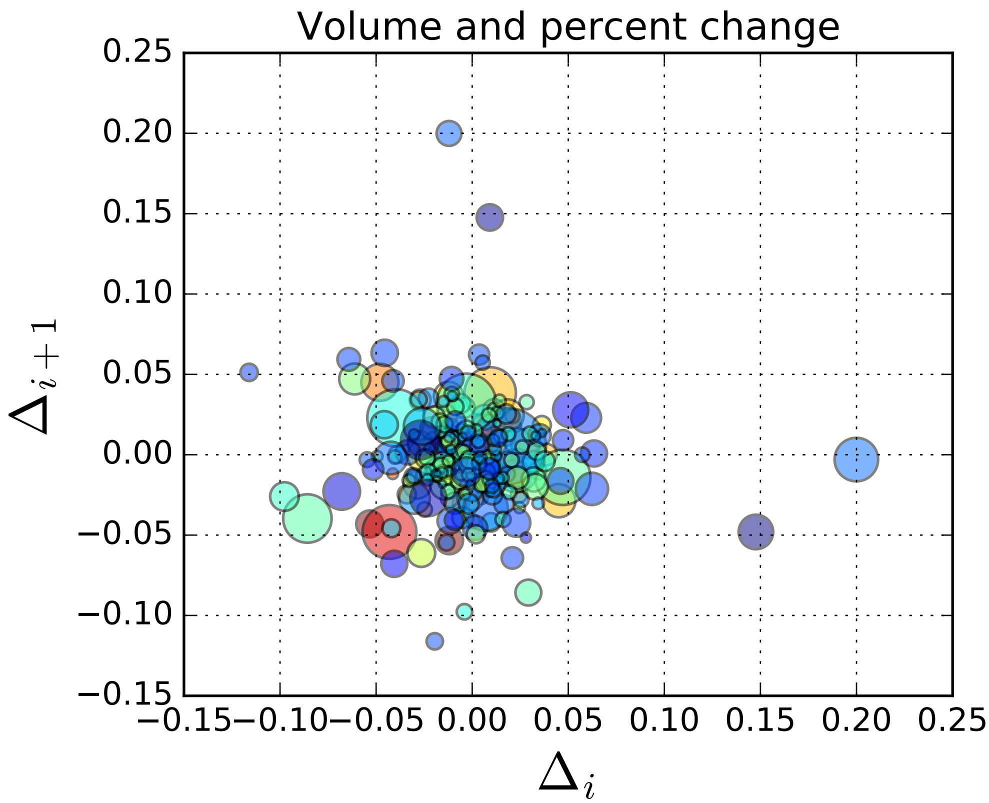
The `table()` command adds a text table to an axes.



	Freeze	Wind	Flood	Quake	Hail
100 year	431.5	1049.4	799.6	2149.8	917.9
50 year	292.2	717.8	456.4	1368.5	865.6
20 year	213.8	636.0	305.7	1175.2	796.0
10 year	124.6	555.4	153.2	677.2	192.5
5 year	66.4	174.3	75.1	577.9	32.0

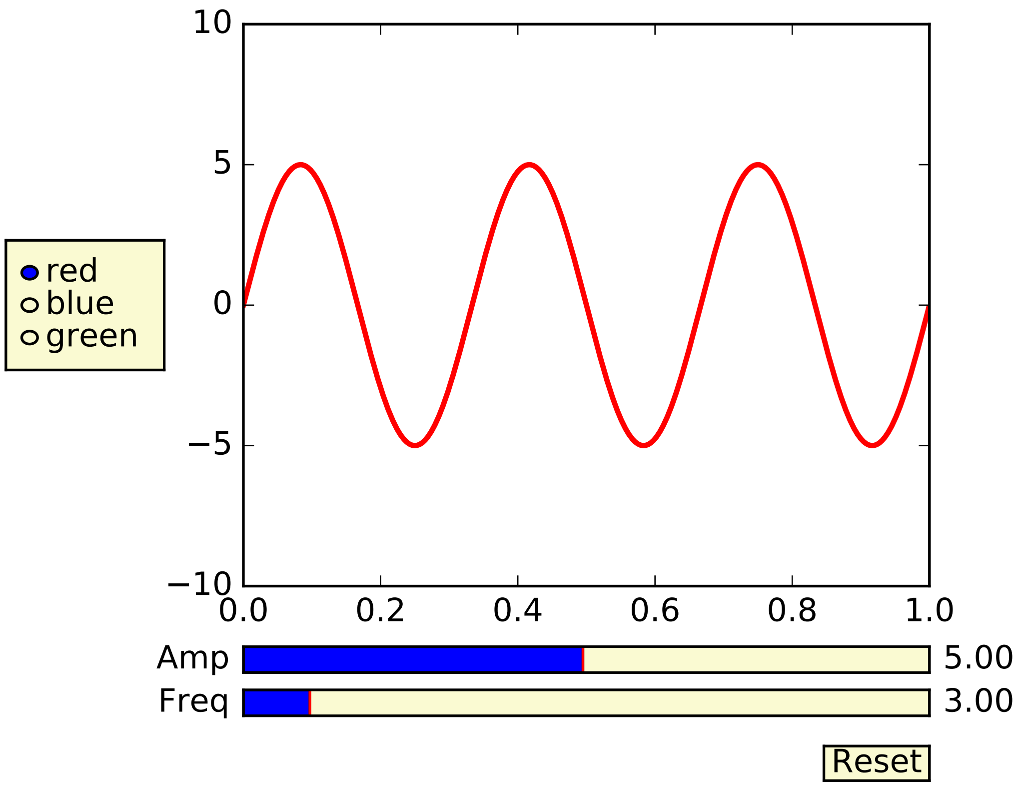
3.8.11 Scatter demo

The `scatter()` command makes a scatter plot with (optional) size and color arguments. This example plots changes in Google's stock price, with marker sizes reflecting the trading volume and colors varying with time. Here, the alpha attribute is used to make semitransparent circle markers.



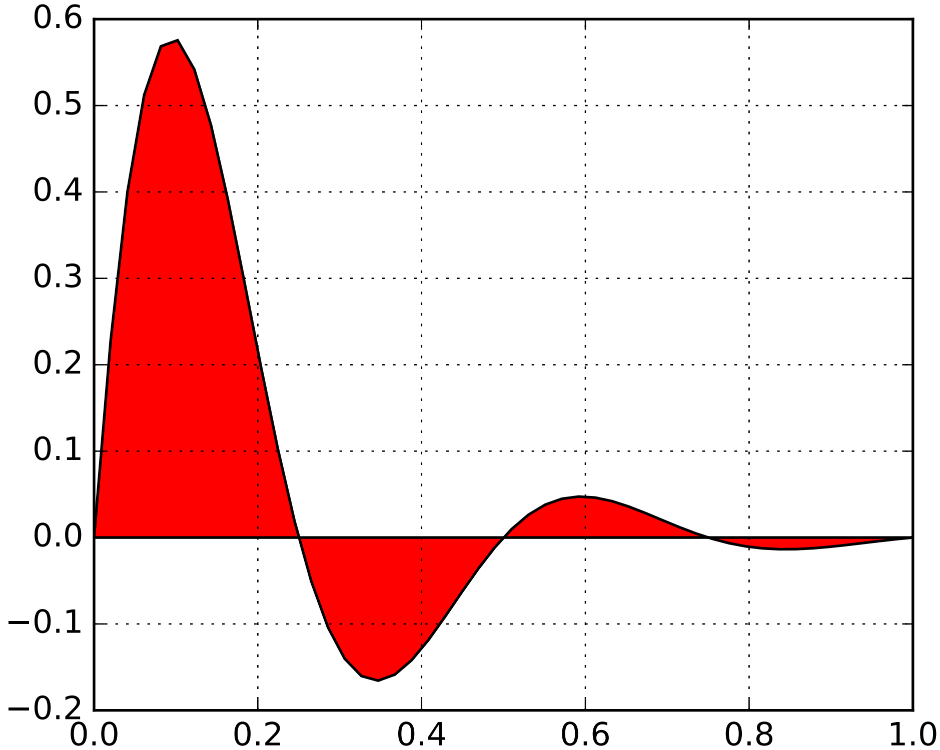
3.8.12 Slider demo

Matplotlib has basic GUI widgets that are independent of the graphical user interface you are using, allowing you to write cross GUI figures and widgets. See `matplotlib.widgets` and the widget examples.



3.8.13 Fill demo

The *fill()* command lets you plot filled curves and polygons:



Thanks to Andrew Straw for adding this function.

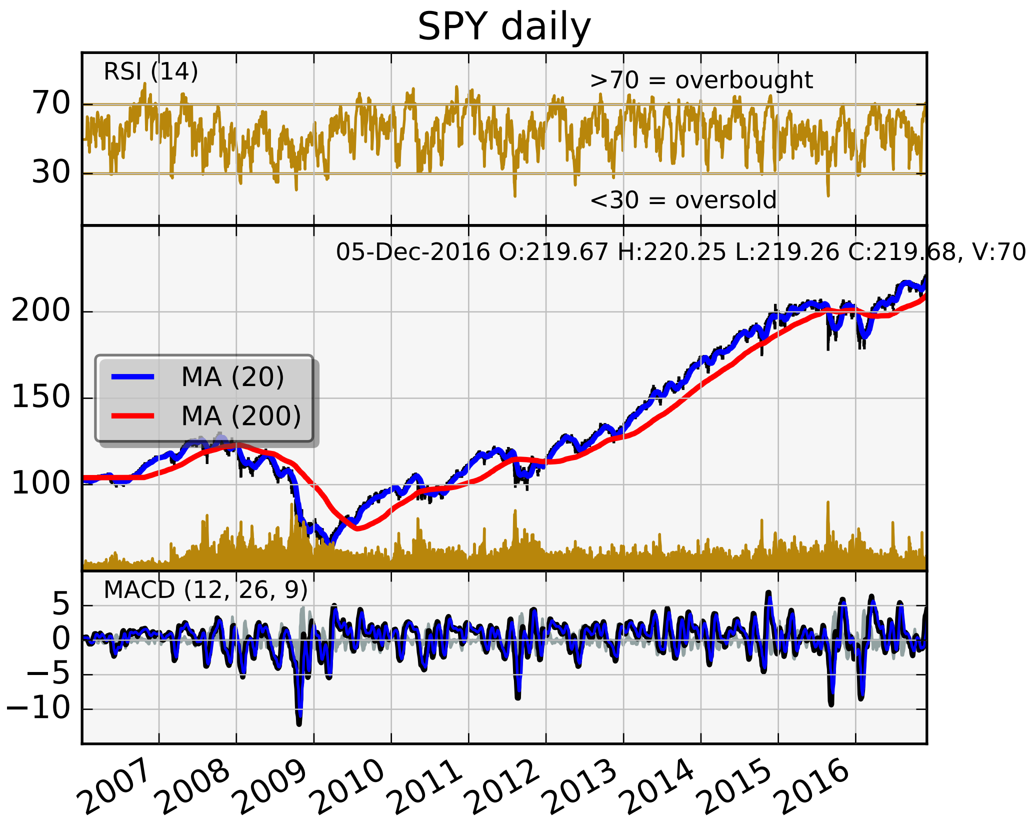
3.8.14 Date demo

You can plot date data with major and minor ticks and custom tick formatters for both.

See *matplotlib.ticker* and *matplotlib.dates* for details and usage.

3.8.15 Financial charts

You can make sophisticated financial plots by combining the various plot functions, layout commands, and labeling tools provided by matplotlib. The following example emulates one of the financial plots in ChartDirector:



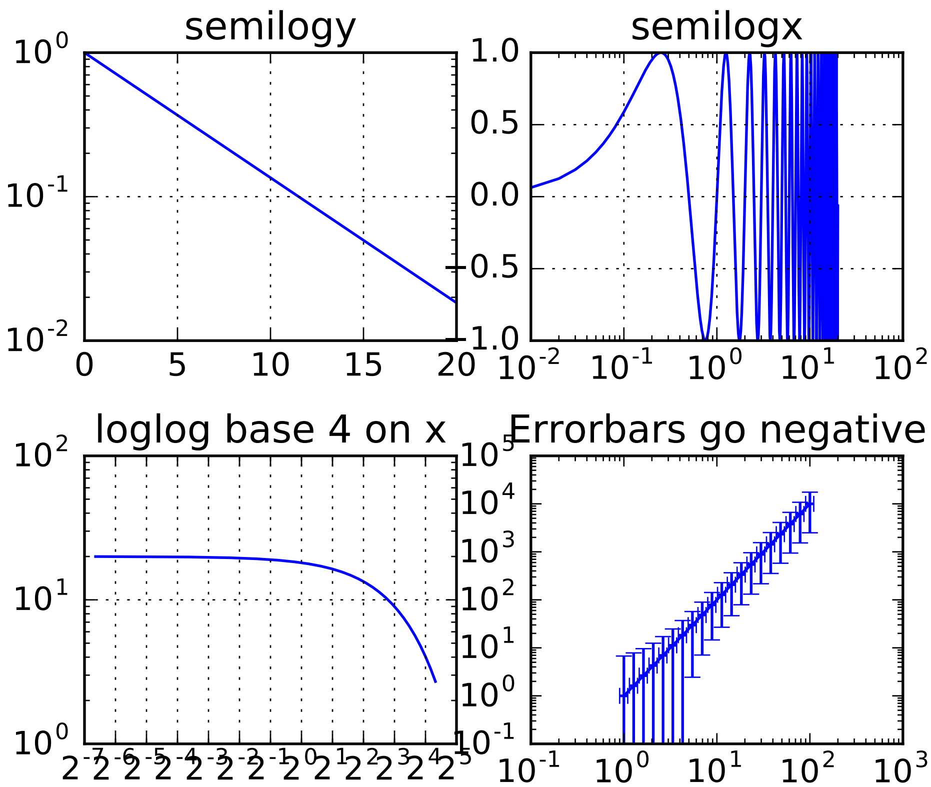
3.8.16 Basemap demo

Jeff Whitaker's *Basemap* add-on toolkit makes it possible to plot data on many different map projections. This example shows how to plot contours, markers and text on an orthographic projection, with NASA's "blue marble" satellite image as a background.

Sorry, could not import Basemap

3.8.17 Log plots

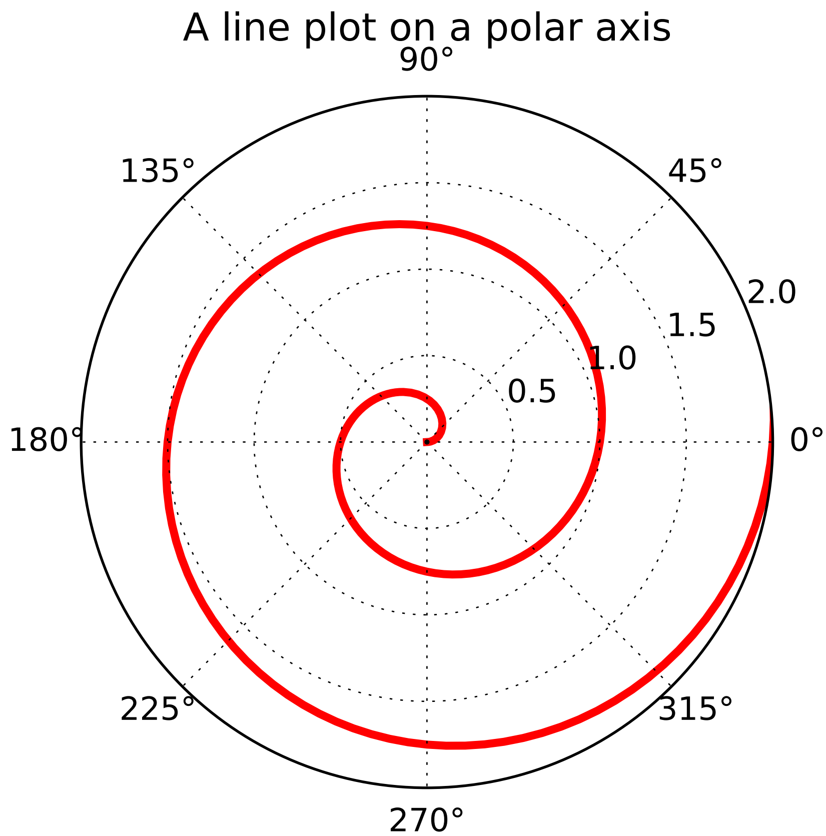
The `semilogx()`, `semilogy()` and `loglog()` functions simplify the creation of logarithmic plots.



Thanks to Andrew Straw, Darren Dale and Gregory Lielens for contributions log-scaling infrastructure.

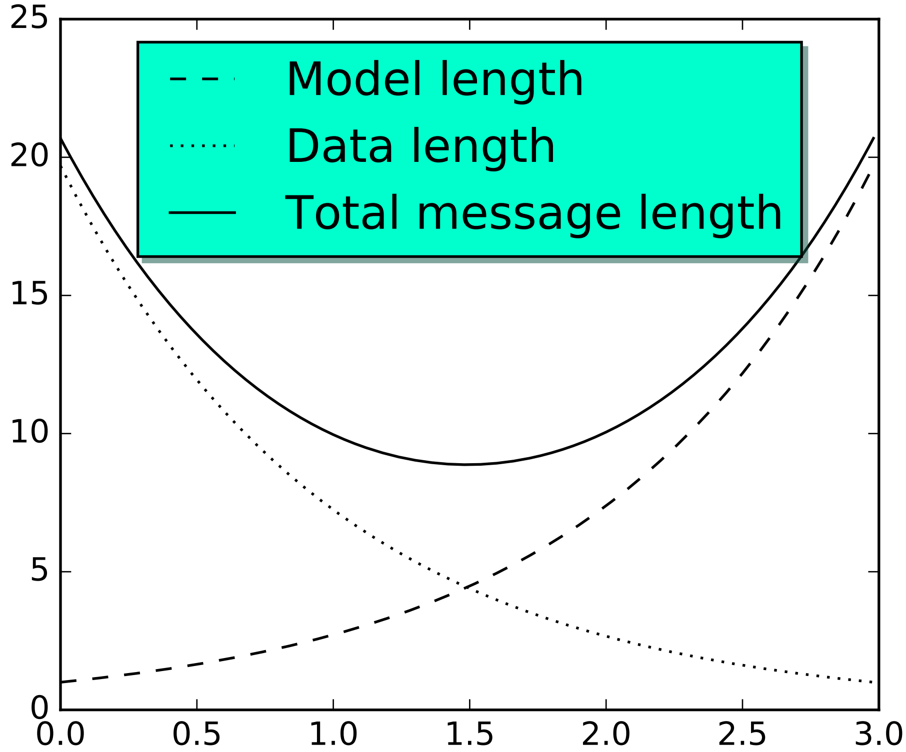
3.8.18 Polar plots

The `polar()` command generates polar plots.



3.8.19 Legends

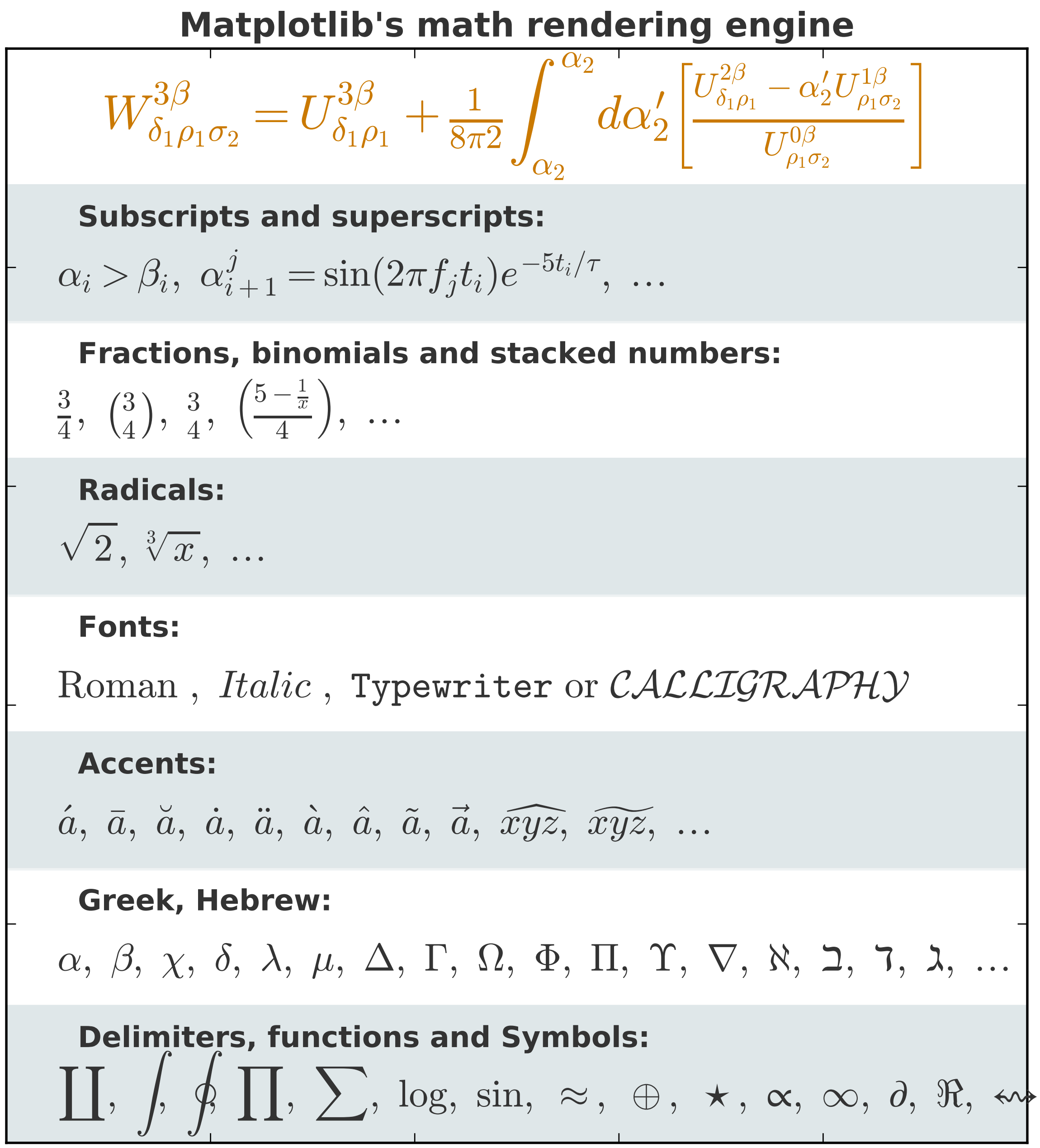
The *legend()* command automatically generates figure legends, with MATLAB-compatible legend placement commands.



Thanks to Charles Twardy for input on the legend command.

3.8.20 Mathtext_examples

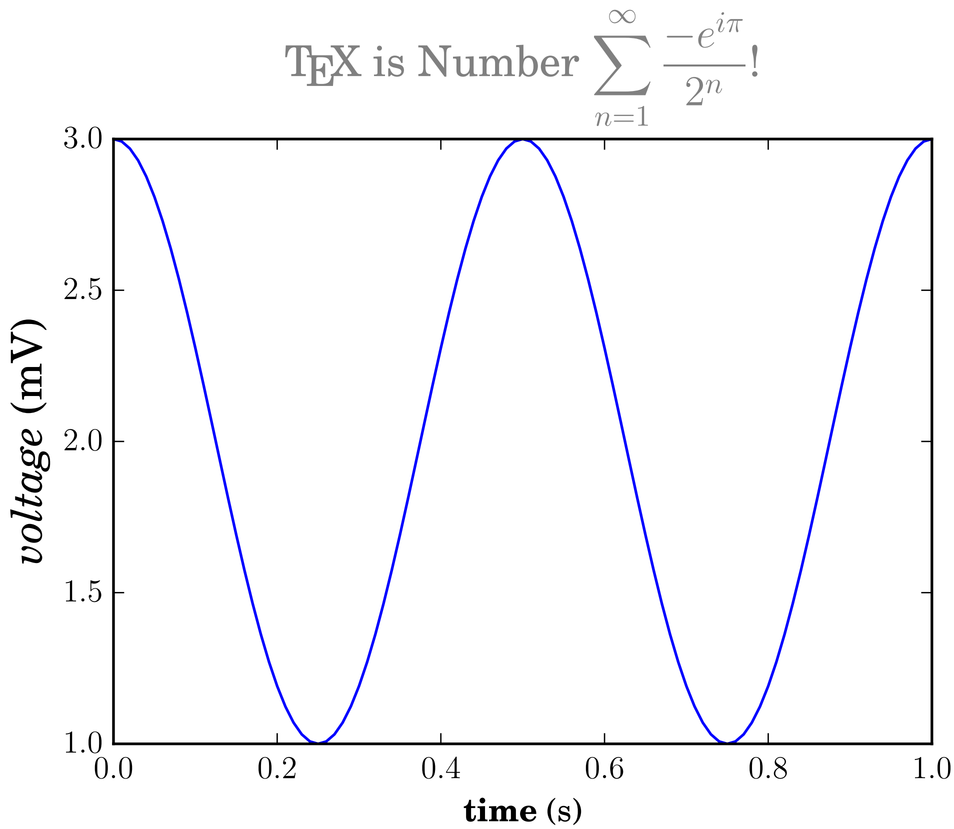
Below is a sampling of the many TeX expressions now supported by matplotlib's internal mathtext engine. The mathtext module provides TeX style mathematical expressions using freetype2 and the BaKoMa computer modern or STIX fonts. See the `matplotlib.mathtext` module for additional details.



Matplotlib's mathtext infrastructure is an independent implementation and does not require TeX or any external packages installed on your computer. See the tutorial at *Writing mathematical expressions*.

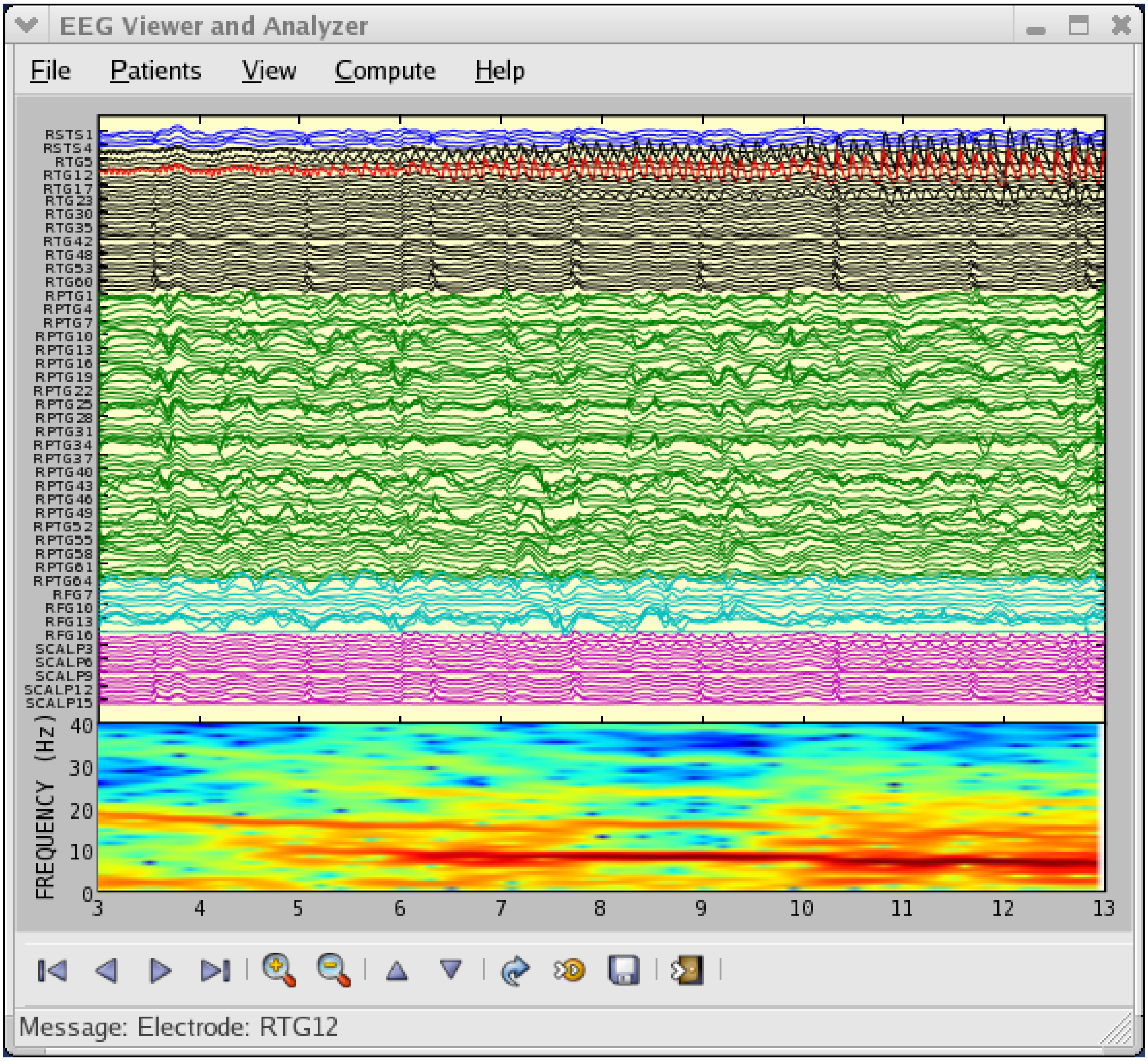
3.8.21 Native TeX rendering

Although matplotlib's internal math rendering engine is quite powerful, sometimes you need TeX. Matplotlib supports external TeX rendering of strings with the *usetex* option.



3.8.22 EEG demo

You can embed matplotlib into pygtk, wx, Tk, FLTK, or Qt applications. Here is a screenshot of an EEG viewer called pbrain.



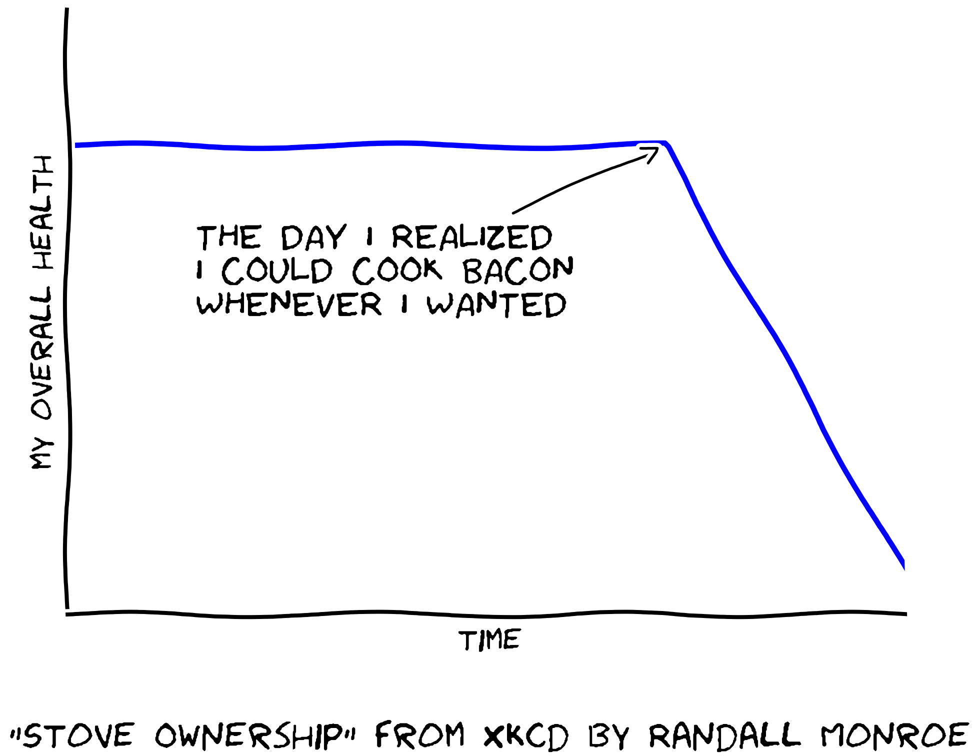
The lower axes uses *specgram()* to plot the spectrogram of one of the EEG channels.

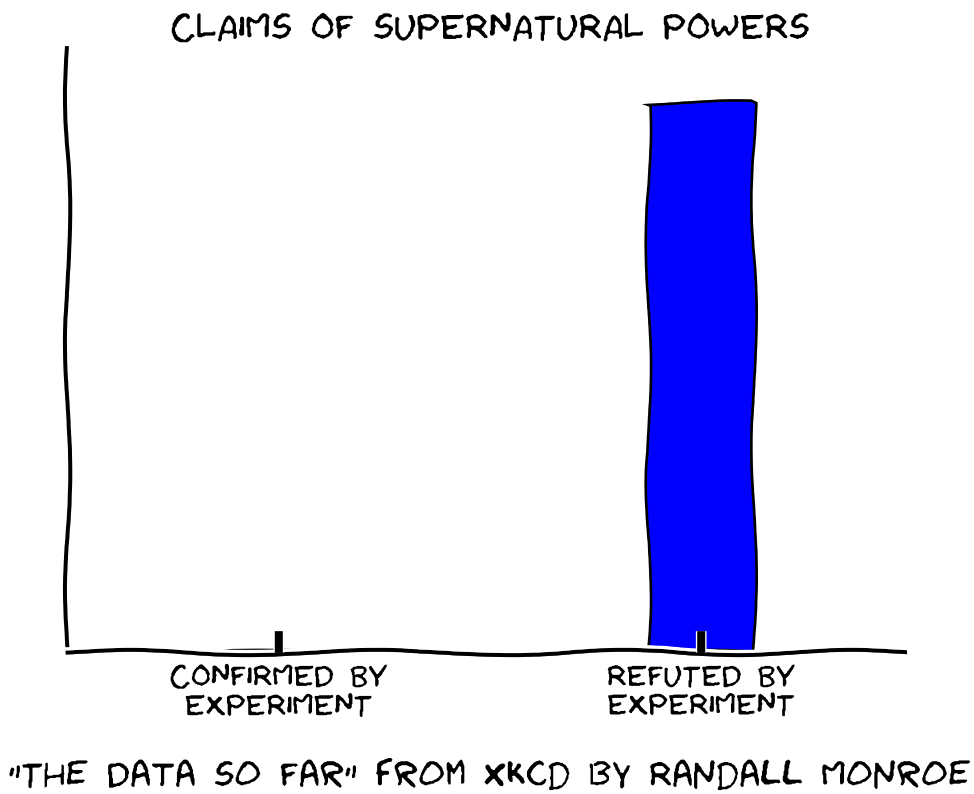
For examples of how to embed matplotlib in different toolkits, see:

- *user_interfaces example code: embedding_in_gtk2.py*
- *user_interfaces example code: embedding_in_wx2.py*
- *user_interfaces example code: mpl_with_glade.py*
- *user_interfaces example code: embedding_in_qt4.py*
- *user_interfaces example code: embedding_in_tk.py*

3.8.23 XKCD-style sketch plots

matplotlib supports plotting in the style of `xkcd`.


THE DAY I REALIZED
I COULD COOK BACON
WHENEVER I WANTED
MY OVERALL HEALTH
TIME
"STOVE OWNERSHIP" FROM XKCD BY RANDALL MONROE



3.9 Choosing Colormaps

3.9.1 Overview

The idea behind choosing a good colormap is to find a good representation in 3D colorspace for your data set. The best colormap for any given data set depends on many things including:

- Whether representing form or metric data (*[Ware]*)
- Your knowledge of the data set (*e.g.*, is there a critical value from which the other values deviate?)
- If there is an intuitive color scheme for the parameter you are plotting
- If there is a standard in the field the audience may be expecting

For many applications, a perceptually uniform colormap is the best choice — one in which equal steps in data are perceived as equal steps in the color space. Researchers have found that the human brain perceives changes in the lightness parameter as changes in the data much better than, for example, changes in hue. Therefore, colormaps which have monotonically increasing lightness through the colormap will be better interpreted by the viewer.

Color can be represented in 3D space in various ways. One way to represent color is using CIELAB. In CIELAB, color space is represented by lightness, L^*; red-green, a^*; and yellow-blue, b^*. The lightness parameter L^* can then be used to learn more about how the matplotlib colormaps will be perceived by viewers.

An excellent starting resource for learning about human perception of colormaps is from *[IBM]*.

3.9.2 Classes of colormaps

Colormaps are often split into several categories based on their function (see, *e.g.*, *[Moreland]*):

1. Sequential: change in lightness and often saturation of color incrementally, often using a single hue; should be used for representing information that has ordering.
2. Diverging: change in lightness and possibly saturation of two different colors that meet in the middle at an unsaturated color; should be used when the information being plotted has a critical middle value, such as topography or when the data deviates around zero.
3. Qualitative: often are miscellaneous colors; should be used to represent information which does not have ordering or relationships.

3.9.3 Lightness of matplotlib colormaps

Here we examine the lightness values of the matplotlib colormaps. Note that some documentation on the colormaps is available (*[list-colormaps]*).

Sequential

For the Sequential plots, the lightness value increases monotonically through the colormaps. This is good. Some of the L^* values in the colormaps span from 0 to 100 (binary and the other grayscale), and others start around $L^* = 20$. Those that have a smaller range of L^* will accordingly have a smaller perceptual range. Note also that the L^* function varies amongst the colormaps: some are approximately linear in L^* and others are more curved.

Sequential2

Many of the L^* values from the Sequential2 plots are monotonically increasing, but some (autumn, cool, spring, and winter) plateau or even go both up and down in L^* space. Others (afmhot, copper, gist_heat, and hot) have kinks in the L^* functions. Data that is being represented in a region of the colormap that is at a plateau or kink will lead to a perception of banding of the data in those values in the colormap (see *[mycarta-banding]* for an excellent example of this).

Diverging

For the Diverging maps, we want to have monotonically increasing L^* values up to a maximum, which should be close to $L^* = 100$, followed by monotonically decreasing L^* values. We are looking for approximately equal minimum L^* values at opposite ends of the colormap. By these measures, BrBG and RdBu are good options. coolwarm is a good option, but it doesn't span a wide range of L^* values (see grayscale section below).

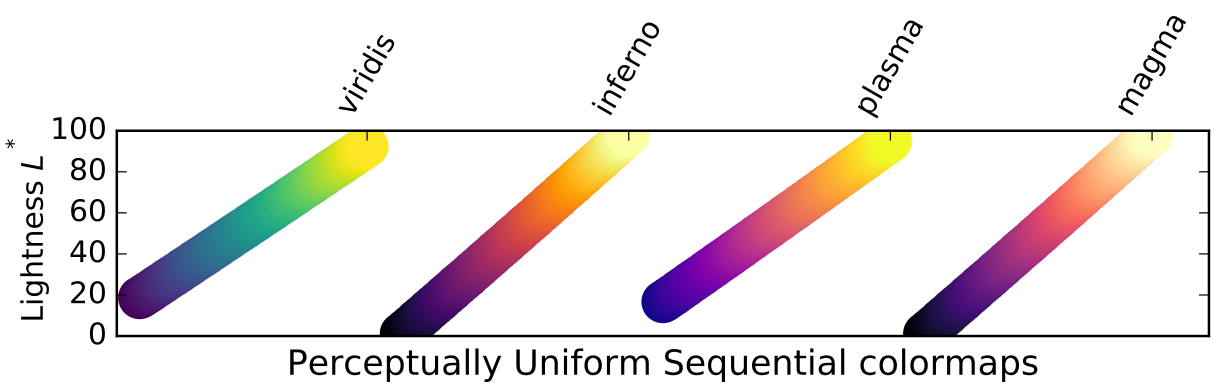
Qualitative

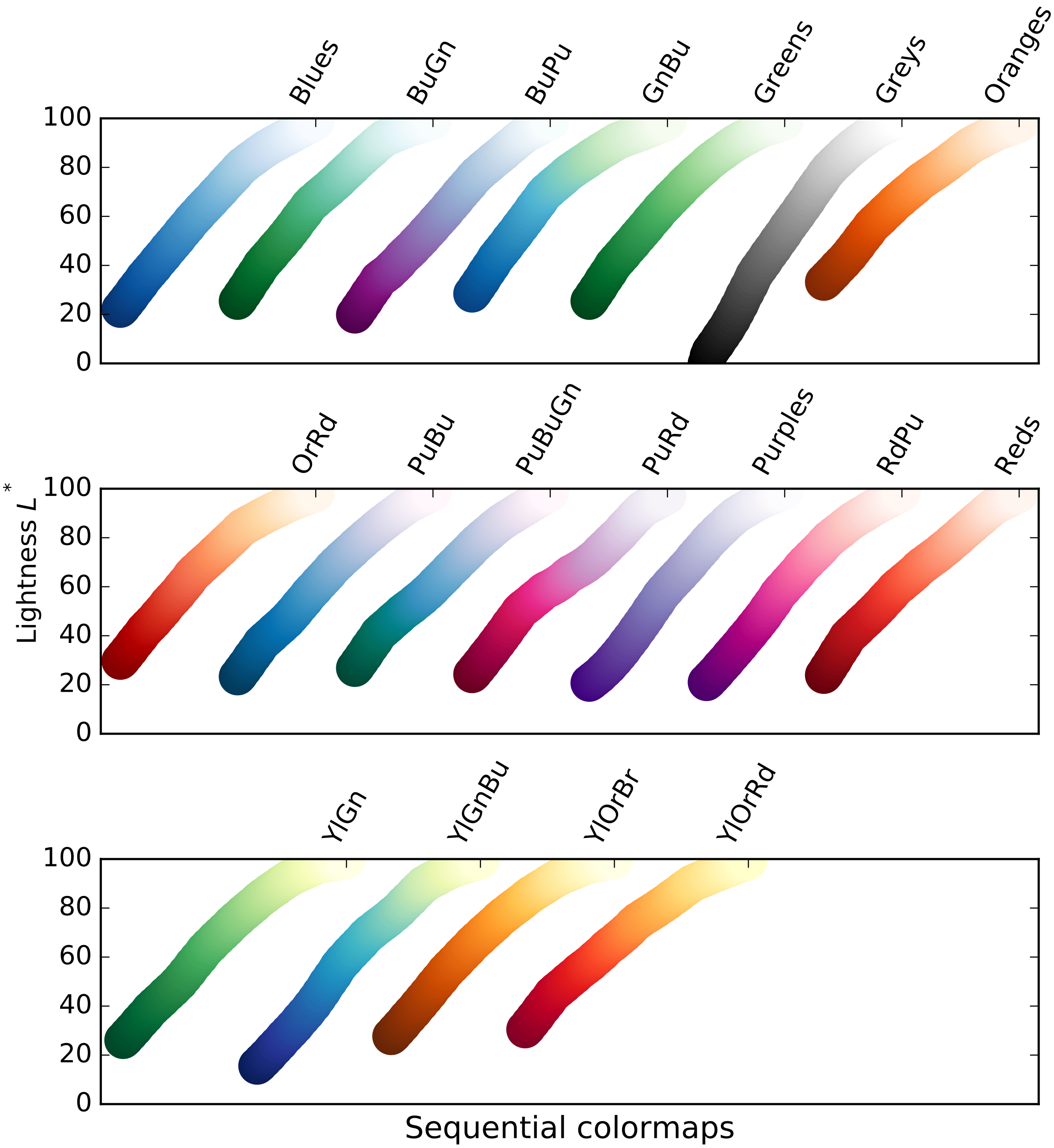
Qualitative colormaps are not aimed at being perceptual maps, but looking at the lightness parameter can verify that for us. The L^* values move all over the place throughout the colormap, and are clearly not monotonically increasing. These would not be good options for use as perceptual colormaps.

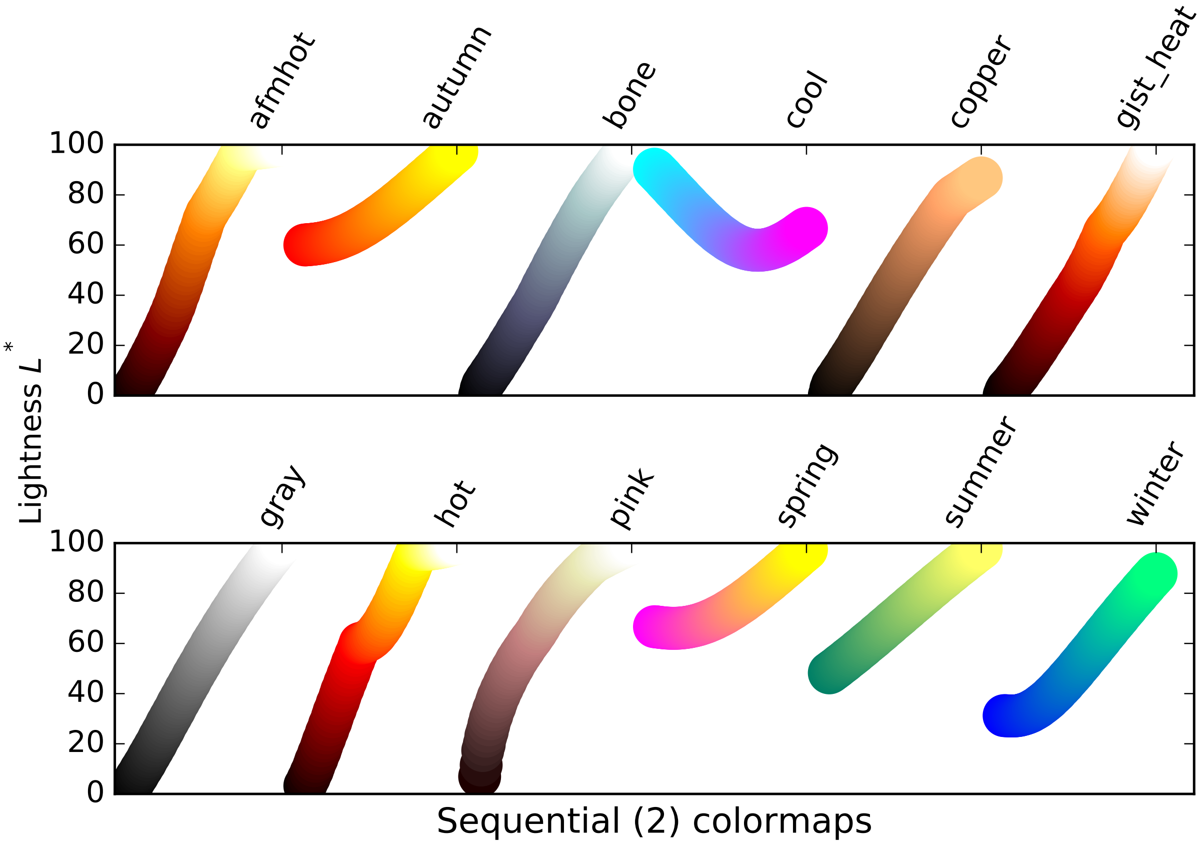
Miscellaneous

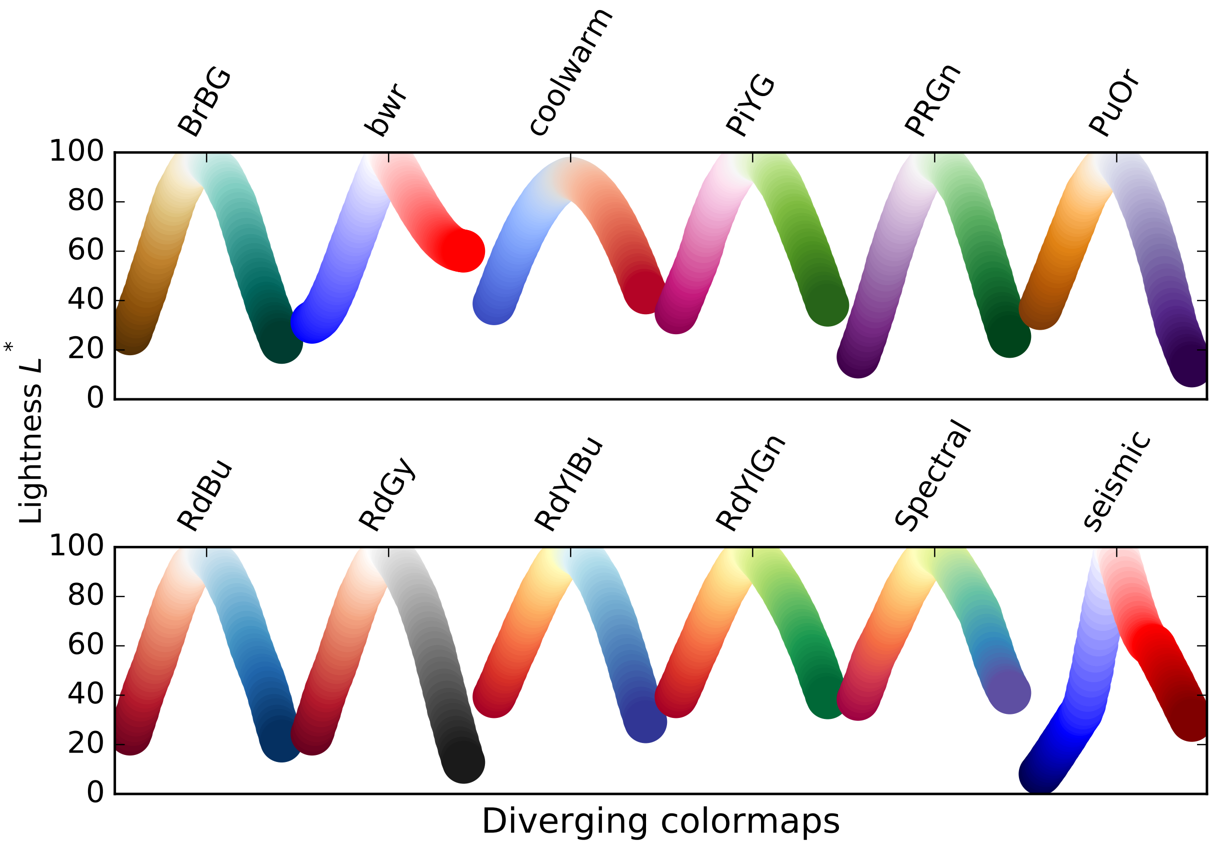
Some of the miscellaneous colormaps have particular uses for which they have been created. For example, gist_earth, ocean, and terrain all seem to be created for plotting topography (green/brown) and water depths (blue) together. We would expect to see a divergence in these colormaps, then, but multiple kinks may not be ideal, such as in gist_earth and terrain. CMRmap was created to convert well to grayscale, though it does appear to have some small kinks in L^*. cubehelix was created to vary smoothly in both lightness and hue, but appears to have a small hump in the green hue area.

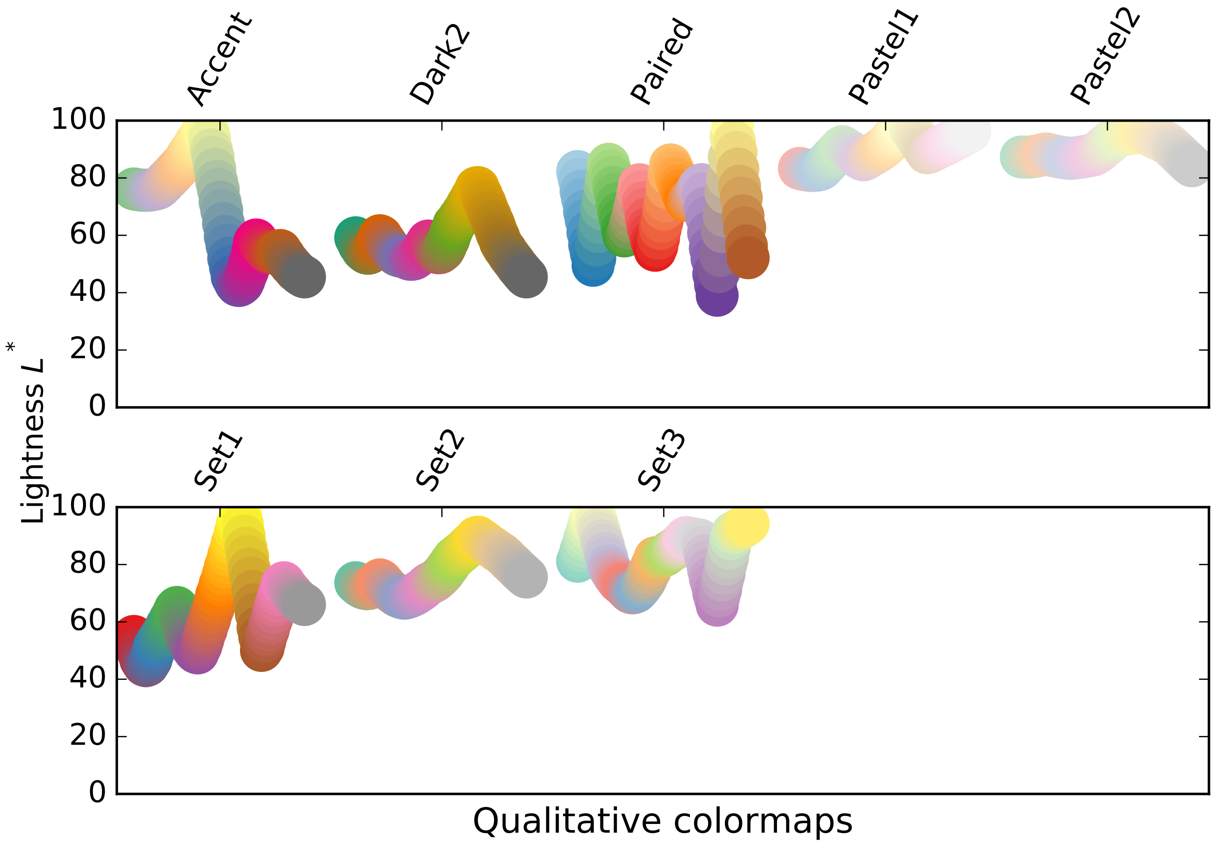
The often-used jet colormap is included in this set of colormaps. We can see that the L^* values vary widely throughout the colormap, making it a poor choice for representing data for viewers to see perceptually. See an extension on this idea at *[mycarta-jet]*.

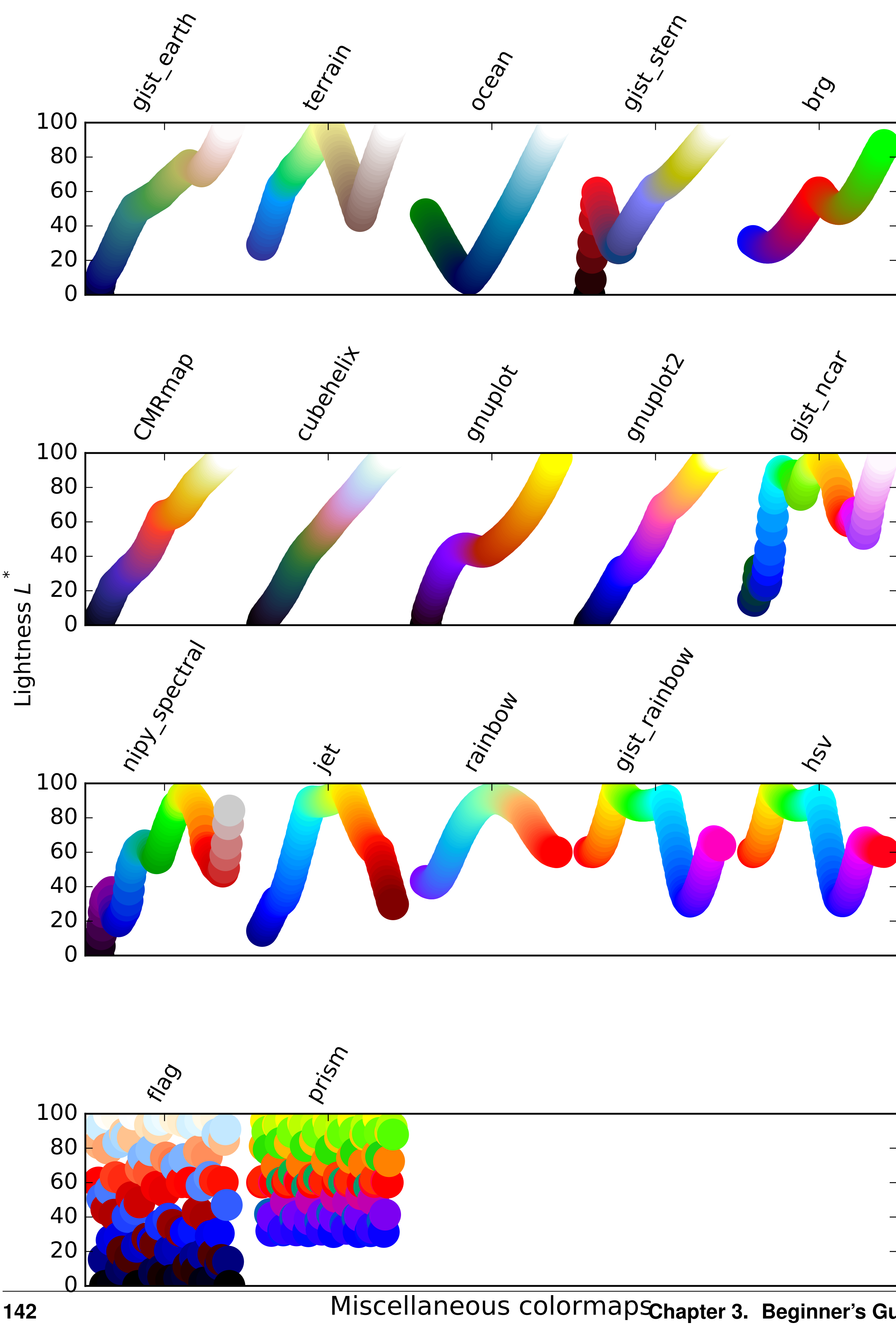



Blues
BuGn
BuPu
GnBu
Greens
Greys
Oranges
OrRd
PuBu
PuBuGn
PuRd
Purples
RdPu
Reds
YlGn
YlGnBu
YlOrBr
YlOrRd
Lightness L^*
100
80
60
40
20
0
Sequential colormaps


afmhot
autumn
bone
cool
copper
gist_heat
gray
hot
pink
spring
summer
winter
Lightness L^*
100
80
60
40
20
0
Sequential (2) colormaps


BrBG
bwr
coolwarm
PiYG
PRGn
PuOr
RdBu
RdGy
RdYlBu
RdYlGn
Spectral
seismic
Lightness L^*
100
80
60
40
20
0
Diverging colormaps


Accent
Dark2
Paired
Pastel1
Pastel2
Set1
Set2
Set3
Lightness L^*
0
20
40
60
80
100
Qualitative colormaps

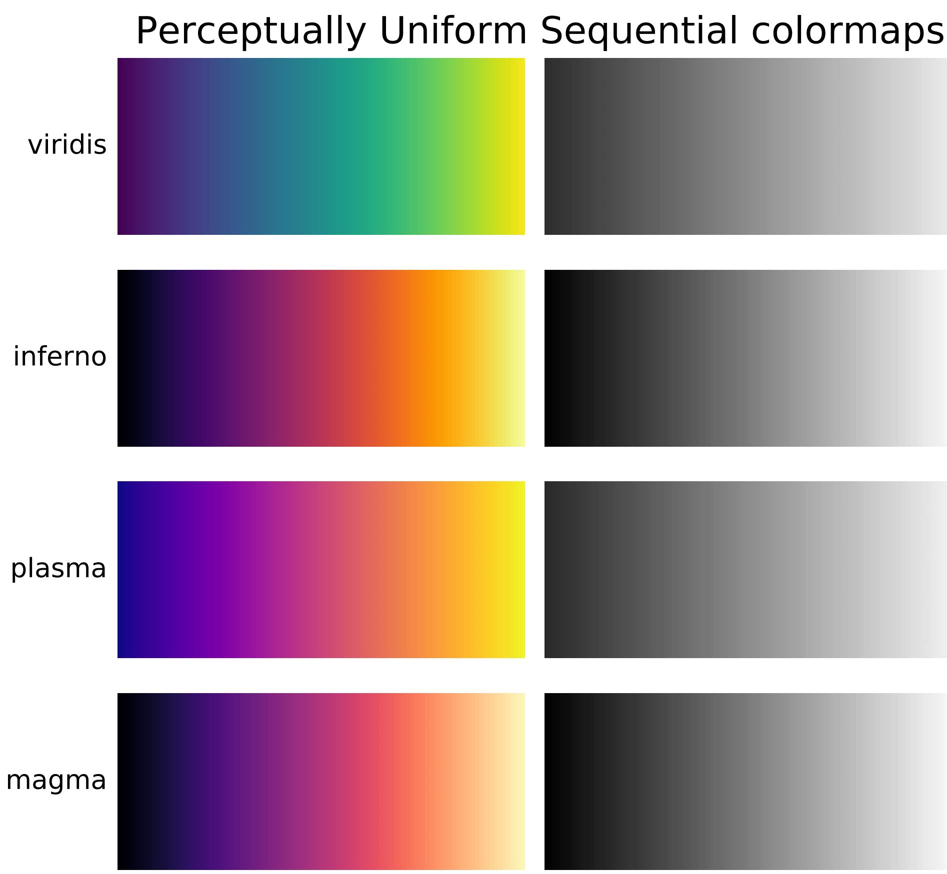

gist_earth
terrain
ocean
gist_stern
brg
CMRmap
cubehelix
gnuplot
gnuplot2
gist_ncar
nipy_spectral
jet
rainbow
gist_rainbow
hsv
flag
prism
Lightness L^*
0
20
40
60
80
100
Miscellaneous colormaps

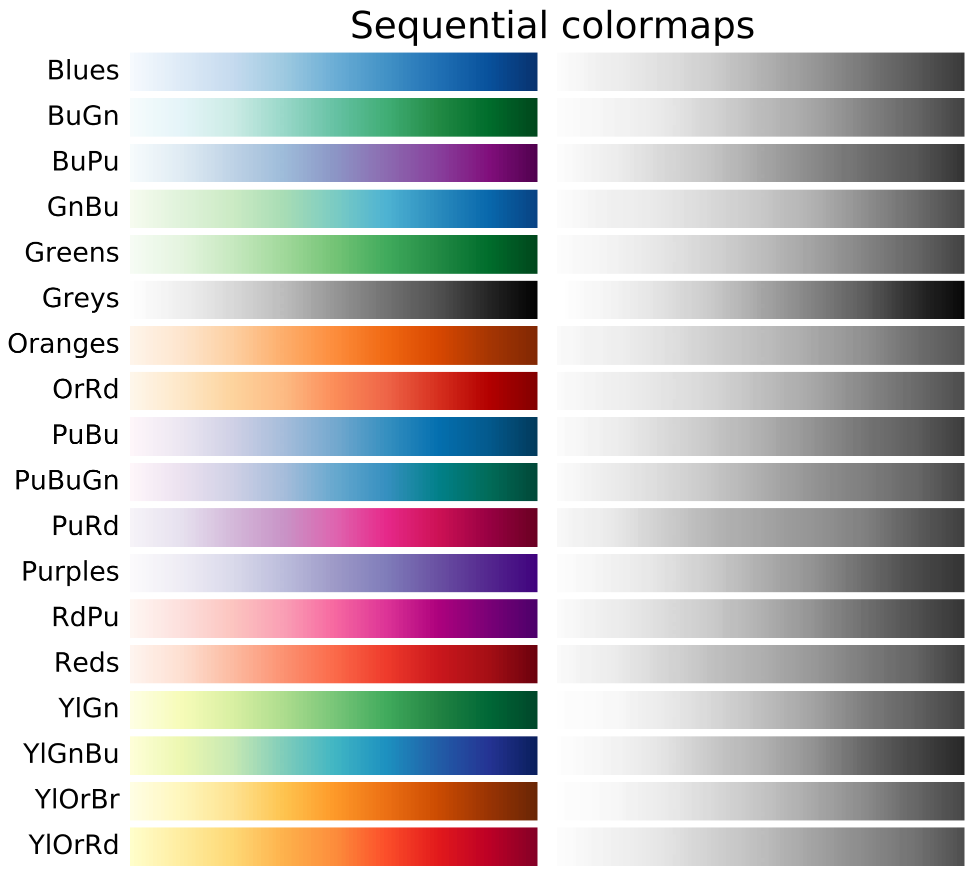
3.9.4 Grayscale conversion

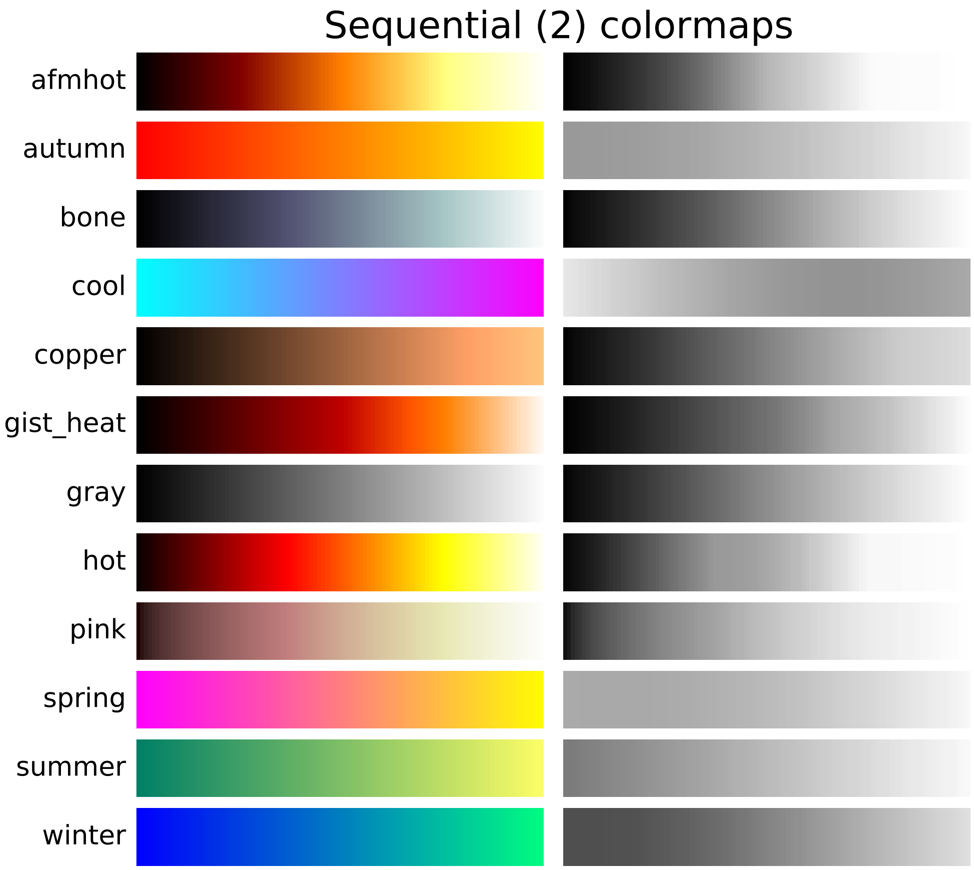
It is important to pay attention to conversion to grayscale for color plots, since they may be printed on black and white printers. If not carefully considered, your readers may end up with indecipherable plots because the grayscale changes unpredictably through the colormap.

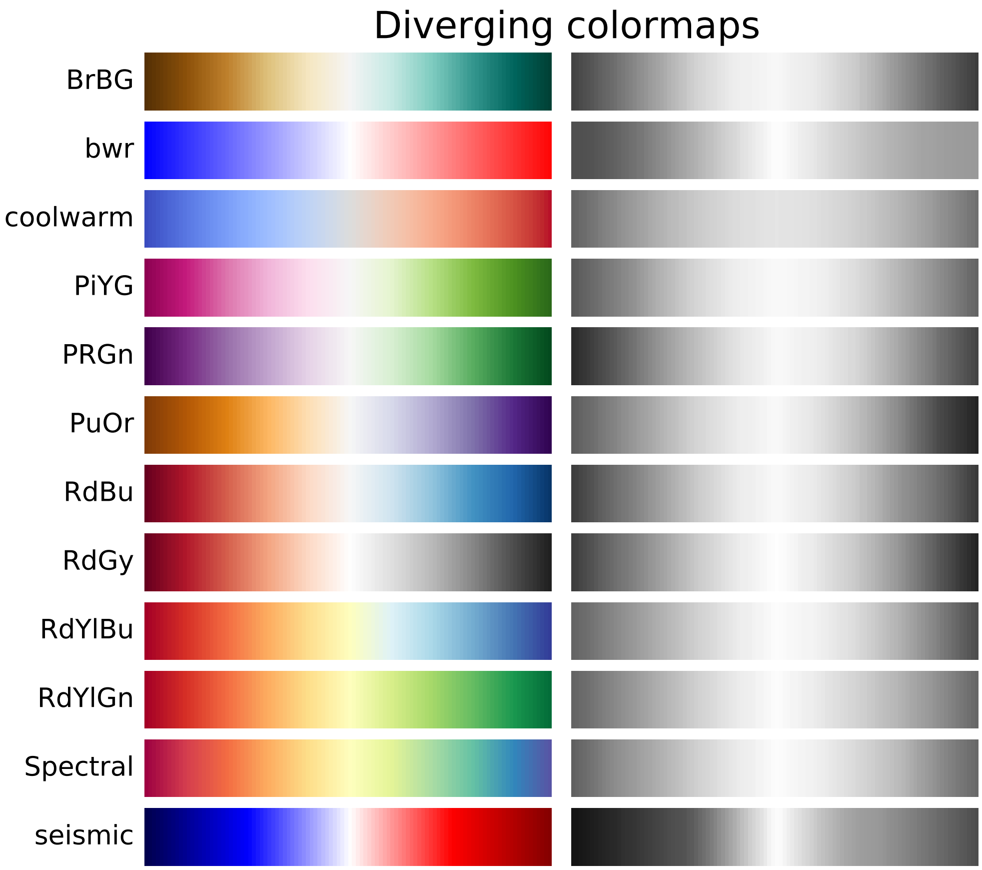
Conversion to grayscale is done in many different ways *[bw]*. Some of the better ones use a linear combination of the rgb values of a pixel, but weighted according to how we perceive color intensity. A nonlinear method of conversion to grayscale is to use the L^* values of the pixels. In general, similar principles apply for this question as they do for presenting one's information perceptually; that is, if a colormap is chosen that is monotonically increasing in L^* values, it will print in a reasonable manner to grayscale.

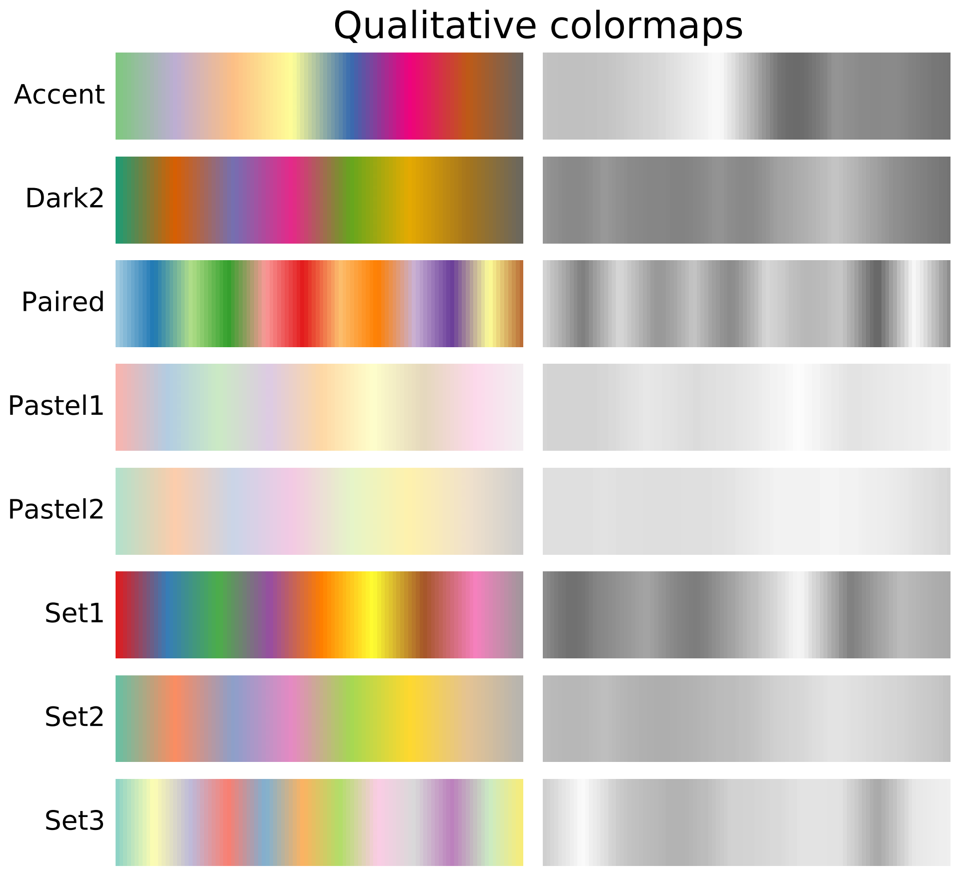
With this in mind, we see that the Sequential colormaps have reasonable representations in grayscale. Some of the Sequential2 colormaps have decent enough grayscale representations, though some (autumn, spring, summer, winter) have very little grayscale change. If a colormap like this was used in a plot and then the plot was printed to grayscale, a lot of the information may map to the same gray values. The Diverging colormaps mostly vary from darker gray on the outer edges to white in the middle. Some (PuOr and seismic) have noticably darker gray on one side than the other and therefore are not very symmetric. coolwarm has little range of gray scale and would print to a more uniform plot, losing a lot of detail. Note that overlaid, labeled contours could help differentiate between one side of the colormap vs. the other since color cannot be used once a plot is printed to grayscale. Many of the Qualitative and Miscellaneous colormaps, such as Accent, hsv, and jet, change from darker to lighter and back to darker gray throughout the colormap. This would make it impossible for a viewer to interpret the information in a plot once it is printed in grayscale.

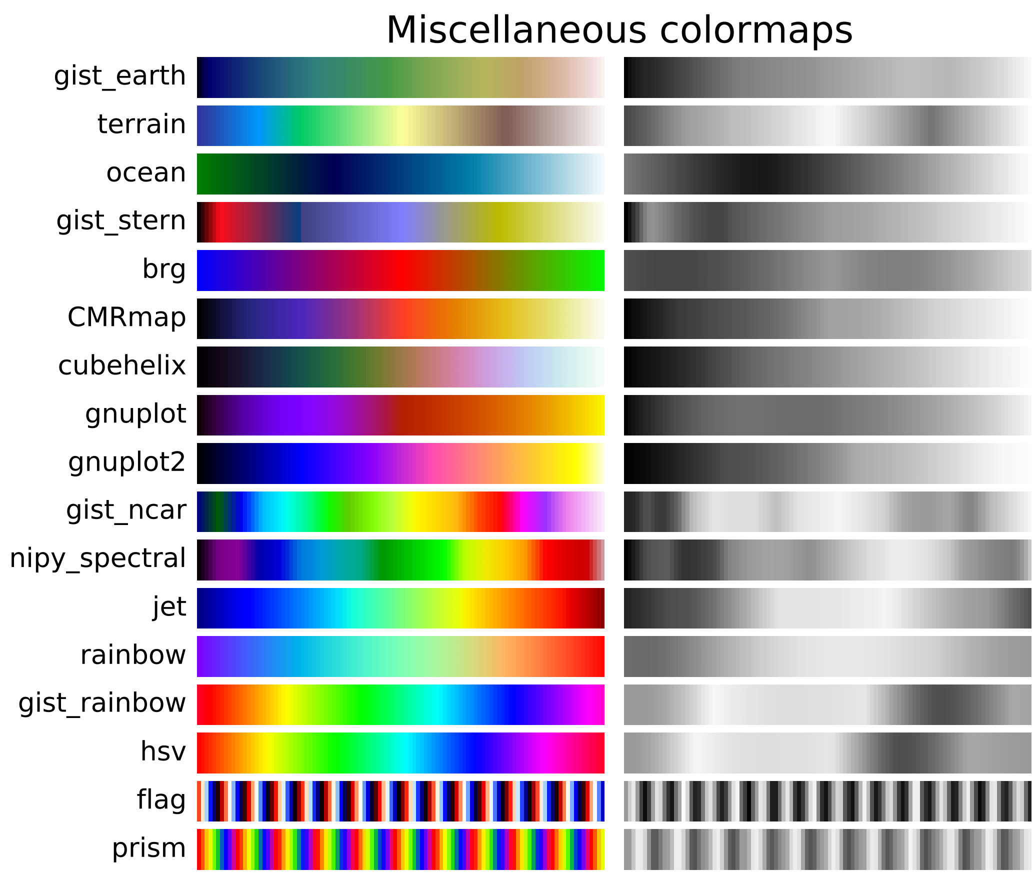



Sequential colormaps
Blues
BuGn
BuPu
GnBu
Greens
Greys
Oranges
OrRd
PuBu
PuBuGn
PuRd
Purples
RdPu
Reds
YlGn
YlGnBu
YlOrBr
YlOrRd


Sequential (2) colormaps
afmhot
autumn
bone
cool
copper
gist_heat
gray
hot
pink
spring
summer
winter


Diverging colormaps
BrBG
bwr
coolwarm
PiYG
PRGn
PuOr
RdBu
RdGy
RdYlBu
RdYlGn
Spectral
seismic


Qualitative colormaps
Accent
Dark2
Paired
Pastel1
Pastel2
Set1
Set2
Set3



3.9.5 Color vision deficiencies

There is a lot of information available about color blindness (*e.g.*, *[colorblindness]*). Additionally, there are tools available to convert images to how they look for different types of color vision deficiencies (*e.g.*, *[asp]*).

The most common form of color vision deficiency involves differentiating between red and green. Thus, avoiding colormaps with both red and green will avoid many problems in general.

3.9.6 References

3.10 Colormap Normalization

Objects that use colormaps by default linearly map the colors in the colormap from data values *vmin* to *vmax*. For example:

```
pcm = ax.pcolormesh(x, y, Z, vmin=-1., vmax=1., cmap='RdBu_r')
```

will map the data in *Z* linearly from -1 to +1, so *Z=0* will give a color at the center of the colormap *RdBu_r* (white in this case).

Matplotlib does this mapping in two steps, with a normalization from [0,1] occurring first, and then mapping onto the indices in the colormap. Normalizations are classes defined in the *matplotlib.colors()* module. The default, linear normalization is *matplotlib.colors.Normalize()*.

Artists that map data to color pass the arguments *vmin* and *vmax* to construct a *matplotlib.colors.Normalize()* instance, then call it:

```
In [1]: import matplotlib as mpl

In [2]: norm = mpl.colors.Normalize(vmin=-1.,vmax=1.)

In [3]: norm(0.)
Out[3]: 0.5
```

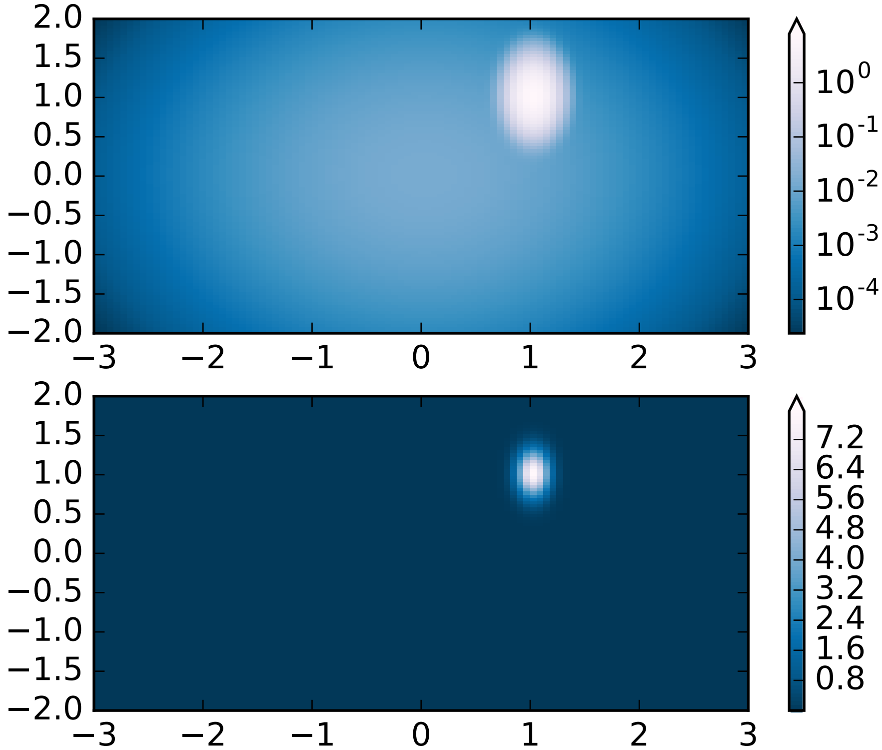
However, there are sometimes cases where it is useful to map data to colormaps in a non-linear fashion.

3.10.1 Logarithmic

One of the most common transformations is to plot data by taking its logarithm (to the base-10). This transformation is useful to display changes across disparate scales. Using `colors.LogNorm()` normalizes the data via log_{10}. In the example below, there are two bumps, one much smaller than the other. Using `colors.LogNorm()`, the shape and location of each bump can clearly be seen:

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

'''
Lognorm: Instead of pcolor log10(Z1) you can have colorbars that have
the exponential labels using a norm.
'''
N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right.  Needs to have
# z/colour axis on a log scale so we see both hump and spike.  linear
# scale only shows the spike.
Z1 = bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0) +  \
    0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolor(X, Y, Z1,
                   norm=colors.LogNorm(vmin=Z1.min(), vmax=Z1.max()),
                   cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[0], extend='max')
```

```
pcm = ax[1].pcolor(X, Y, Z1, cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[1], extend='max')
fig.show()
```



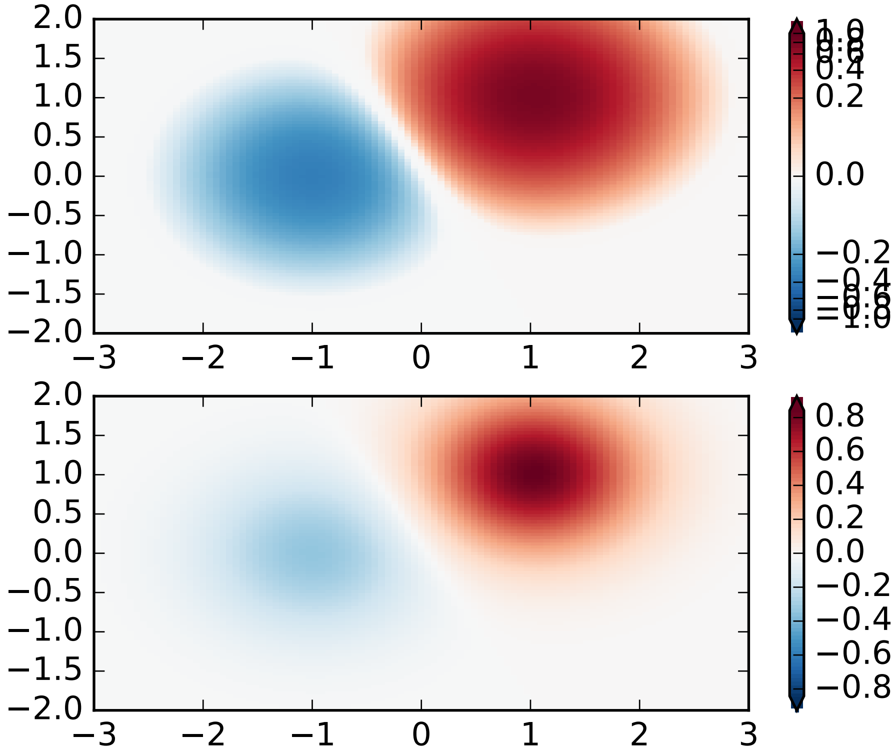
3.10.2 Symmetric logarithmic

Similarly, it sometimes happens that there is data that is positive and negative, but we would still like a logarithmic scaling applied to both. In this case, the negative numbers are also scaled logarithmically, and mapped to smaller numbers; e.g., if `vmin=-vmax`, then they the negative numbers are mapped from 0 to 0.5 and the positive from 0.5 to 1.

Since the logarithm of values close to zero tends toward infinity, a small range around zero needs to be mapped linearly. The parameter *linthresh* allows the user to specify the size of this range (-*linthresh*, *linthresh*). The size of this range in the colormap is set by *linscale*. When *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

import numpy as np
import matplotlib.pyplot as plt
```

```python
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

"""
SymLogNorm: two humps, one negative and one positive, The positive
with 5-times the amplitude. Linearly, you cannot see detail in the
negative hump.  Here we logarithmically scale the positive and
negative data separately.

Note that colorbar labels do not come out looking very good.
"""
N=100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = (bivariate_normal(X, Y, 1., 1., 1.0, 1.0))**2  \
    - 0.4 * (bivariate_normal(X, Y, 1.0, 1.0, -1.0, 0.0))**2
Z1 = Z1/0.03

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z1,
                       norm=colors.SymLogNorm(linthresh=0.03, linscale=0.03,
                                              vmin=-1.0, vmax=1.0),
                       cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], extend='both')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='RdBu_r', vmin=-np.max(Z1))
fig.colorbar(pcm, ax=ax[1], extend='both')
fig.show()
```



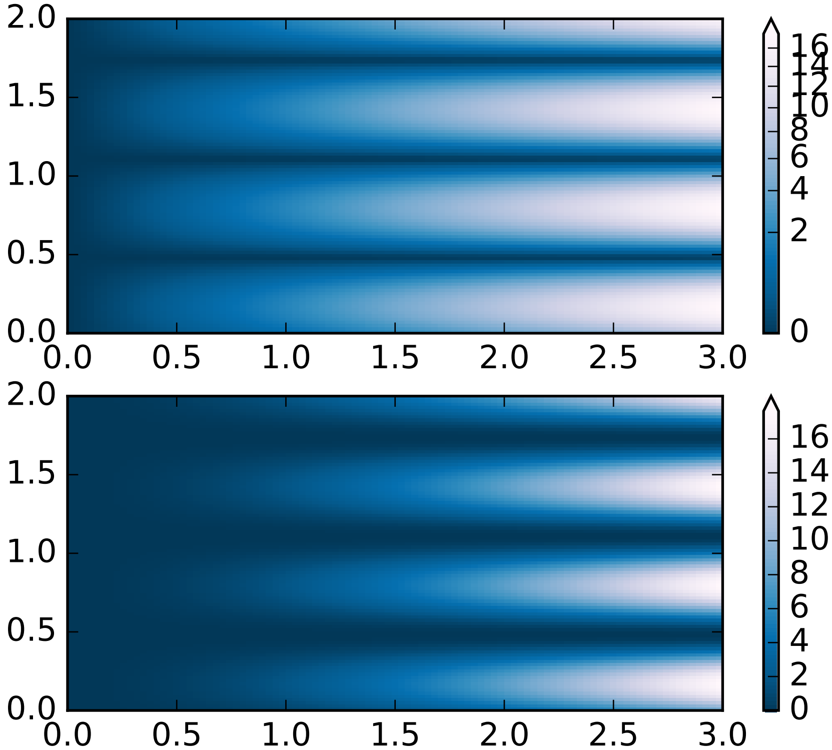
3.10.3 Power-law

Sometimes it is useful to remap the colors onto a power-law relationship (i.e. $y = x^{\gamma}$, where γ is the power). For this we use the `colors.PowerNorm()`. It takes as an argument *gamma* (*gamma* == 1.0 will just yield the default linear normalization):

Note: There should probably be a good reason for plotting the data using this type of transformation. Technical viewers are used to linear and logarithmic axes and data transformations. Power laws are less common, and viewers should explicitly be made aware that they have been used.

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
```

```
'''
PowerNorm: Here a power-law trend in X partially obscures a rectified
sine wave in Y. We can remove the power law using a PowerNorm.
'''
X, Y = np.mgrid[0:3:complex(0, N), 0:2:complex(0, N)]
Z1 = (1 + np.sin(Y * 10.)) * X**(2.)

fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z1, norm=colors.PowerNorm(gamma=1./2.),
                       cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[0], extend='max')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='PuBu_r')
fig.colorbar(pcm, ax=ax[1], extend='max')
fig.show()
```



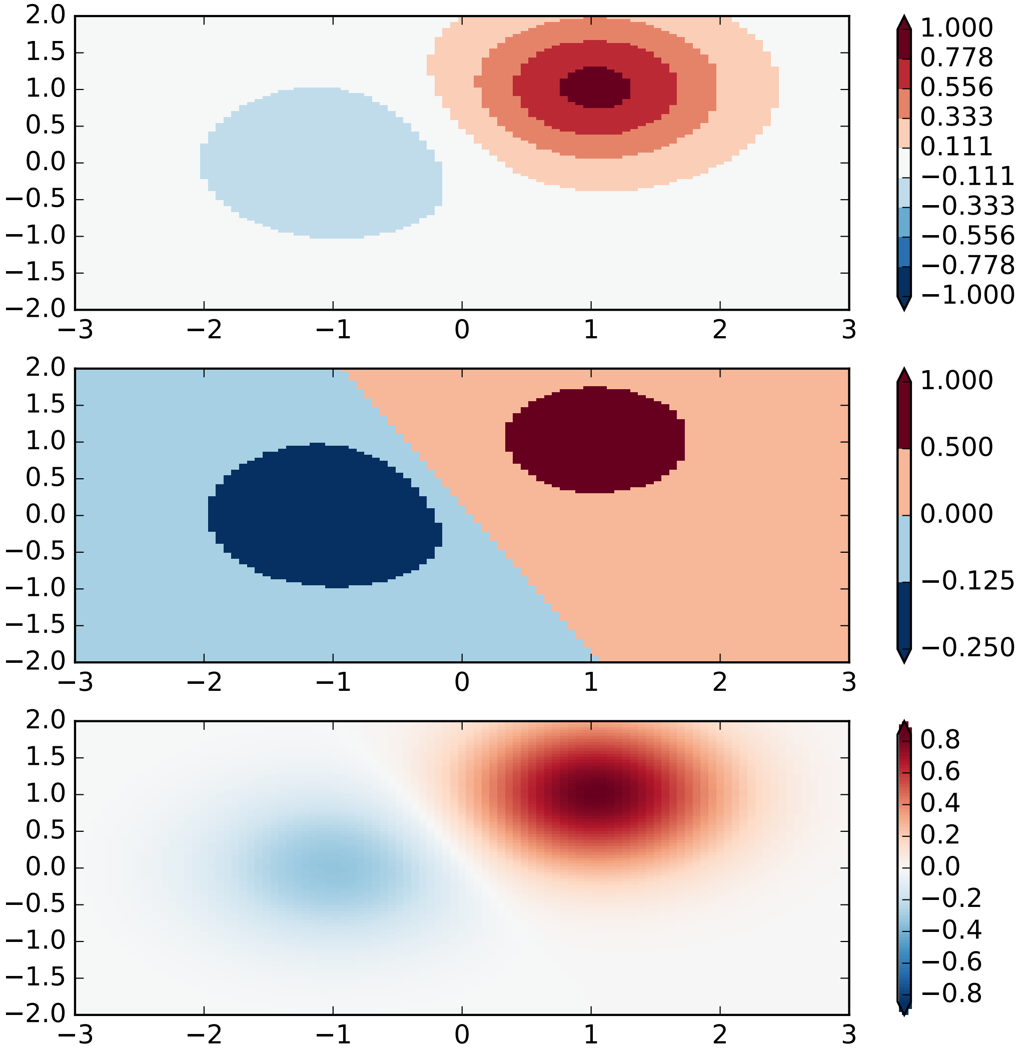
3.10.4 Discrete bounds

Another normaization that comes with matplolib is `colors.BoundaryNorm()`. In addition to *vmin* and *vmax*, this takes as arguments boundaries between which data is to be mapped. The colors are then linearly distributed between these "bounds". For instance:

```
In [4]: import matplotlib.colors as colors

In [5]: bounds = np.array([-0.25, -0.125, 0, 0.5, 1])

In [6]: norm = colors.BoundaryNorm(boundaries=bounds, ncolors=4)

In [7]: print(norm([-0.2,-0.15,-0.02, 0.3, 0.8, 0.99]))
[0 0 1 2 3 3]
```

Note unlike the other norms, this norm returns values from 0 to *ncolors*-1.

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = (bivariate_normal(X, Y, 1., 1., 1.0, 1.0))**2  \
    - 0.4 * (bivariate_normal(X, Y, 1.0, 1.0, -1.0, 0.0))**2
Z1 = Z1/0.03

'''
BoundaryNorm: For this one you provide the boundaries for your colors,
and the Norm puts the first color in between the first pair, the
second color between the second pair, etc.
'''

fig, ax = plt.subplots(3, 1, figsize=(8, 8))
ax = ax.flatten()
# even bounds gives a contour-like effect
bounds = np.linspace(-1, 1, 10)
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[0].pcolormesh(X, Y, Z1,
                       norm=norm,
                       cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], extend='both', orientation='vertical')

# uneven bounds changes the colormapping:
bounds = np.array([-0.25, -0.125, 0, 0.5, 1])
norm = colors.BoundaryNorm(boundaries=bounds, ncolors=256)
pcm = ax[1].pcolormesh(X, Y, Z1, norm=norm, cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[1], extend='both', orientation='vertical')

pcm = ax[2].pcolormesh(X, Y, Z1, cmap='RdBu_r', vmin=-np.max(Z1))
fig.colorbar(pcm, ax=ax[2], extend='both', orientation='vertical')
fig.show()
```



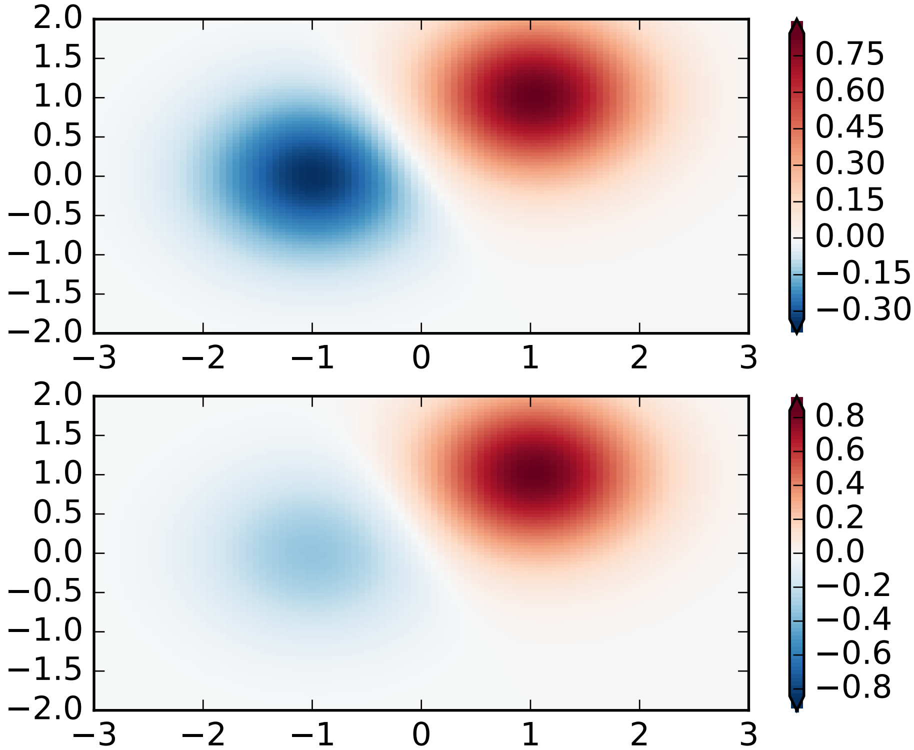
3.10.5 Custom normalization: Two linear ranges

It is possible to define your own normalization. In the following example, we modify `colors:SymLogNorm()` to use different linear maps for the negative data values and the positive. (Note that this example is simple, and does not validate inputs or account for complex cases such as masked data)

Note: This may appear soon as `colors.OffsetNorm()`.

As above, non-symmetric mapping of data to color is non-standard practice for quantitative data, and should only be used advisedly. A practical example is having an ocean/land colormap where the land and ocean data span different ranges.

```
"""
Demonstration of using norm to map colormaps onto data in non-linear ways.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from matplotlib.mlab import bivariate_normal

N = 100
'''
Custom Norm: An example with a customized normalization.  This one
uses the example above, and normalizes the negative data differently
from the positive.
'''
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]
Z1 = (bivariate_normal(X, Y, 1., 1., 1.0, 1.0))**2  \
    - 0.4 * (bivariate_normal(X, Y, 1.0, 1.0, -1.0, 0.0))**2
Z1 = Z1/0.03

# Example of making your own norm.  Also see matplotlib.colors.
# From Joe Kington: This one gives two different linear ramps:


class MidpointNormalize(colors.Normalize):
    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        colors.Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        # I'm ignoring masked values and all kinds of edge cases to make a
        # simple example...
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
        return np.ma.masked_array(np.interp(value, x, y))
#####
fig, ax = plt.subplots(2, 1)

pcm = ax[0].pcolormesh(X, Y, Z1,
                       norm=MidpointNormalize(midpoint=0.),
                       cmap='RdBu_r')
fig.colorbar(pcm, ax=ax[0], extend='both')

pcm = ax[1].pcolormesh(X, Y, Z1, cmap='RdBu_r', vmin=-np.max(Z1))
fig.colorbar(pcm, ax=ax[1], extend='both')
fig.show()
```



2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
3
0.75
0.60
0.45
0.30
0.15
0.00
−0.15
−0.30
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8

CHAPTER

FOUR

ADVANCED GUIDE

4.1 Artist tutorial

There are three layers to the matplotlib API. The `matplotlib.backend_bases.FigureCanvas` is the area onto which the figure is drawn, the `matplotlib.backend_bases.Renderer` is the object which knows how to draw on the `FigureCanvas`, and the `matplotlib.artist.Artist` is the object that knows how to use a renderer to paint onto the canvas. The `FigureCanvas` and `Renderer` handle all the details of talking to user interface toolkits like wxPython or drawing languages like PostScript®, and the `Artist` handles all the high level constructs like representing and laying out the figure, text, and lines. The typical user will spend 95% of his time working with the `Artists`.

There are two types of `Artists`: primitives and containers. The primitives represent the standard graphical objects we want to paint onto our canvas: `Line2D`, `Rectangle`, `Text`, `AxesImage`, etc., and the containers are places to put them (`Axis`, `Axes` and `Figure`). The standard use is to create a `Figure` instance, use the `Figure` to create one or more `Axes` or `Subplot` instances, and use the `Axes` instance helper methods to create the primitives. In the example below, we create a `Figure` instance using `matplotlib.pyplot.figure()`, which is a convenience method for instantiating `Figure` instances and connecting them with your user interface or drawing toolkit `FigureCanvas`. As we will discuss below, this is not necessary – you can work directly with PostScript, PDF Gtk+, or wxPython `FigureCanvas` instances, instantiate your `Figures` directly and connect them yourselves – but since we are focusing here on the `Artist` API we'll let `pyplot` handle some of those details for us:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(2,1,1) # two rows, one column, first plot
```

The `Axes` is probably the most important class in the matplotlib API, and the one you will be working with most of the time. This is because the `Axes` is the plotting area into which most of the objects go, and the `Axes` has many special helper methods (`plot()`, `text()`, `hist()`, `imshow()`) to create the most common graphics primitives (`Line2D`, `Text`, `Rectangle`, `Image`, respectively). These helper methods will take your data (e.g., `numpy` arrays and strings) and create primitive `Artist` instances as needed (e.g., `Line2D`), add them to the relevant containers, and draw them when requested. Most of you are probably familiar with the `Subplot`, which is just a special case of an `Axes` that lives on a regular rows by columns grid of `Subplot` instances. If you want to create an `Axes` at an arbitrary location, simply use the `add_axes()` method which takes a list of `[left, bottom, width, height]` values in 0-1 relative figure coordinates:

```
fig2 = plt.figure()
ax2 = fig2.add_axes([0.15, 0.1, 0.7, 0.3])
```

Continuing with our example:

```
import numpy as np
t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax.plot(t, s, color='blue', lw=2)
```

In this example, `ax` is the `Axes` instance created by the `fig.add_subplot` call above (remember `Subplot` is just a subclass of `Axes`) and when you call `ax.plot`, it creates a `Line2D` instance and adds it to the `Axes.lines` list. In the interactive ipython session below, you can see that the `Axes.lines` list is length one and contains the same line that was returned by the `line, = ax.plot...` call:

```
In [101]: ax.lines[0]
Out[101]: <matplotlib.lines.Line2D instance at 0x19a95710>

In [102]: line
Out[102]: <matplotlib.lines.Line2D instance at 0x19a95710>
```

If you make subsequent calls to `ax.plot` (and the hold state is "on" which is the default) then additional lines will be added to the list. You can remove lines later simply by calling the list methods; either of these will work:

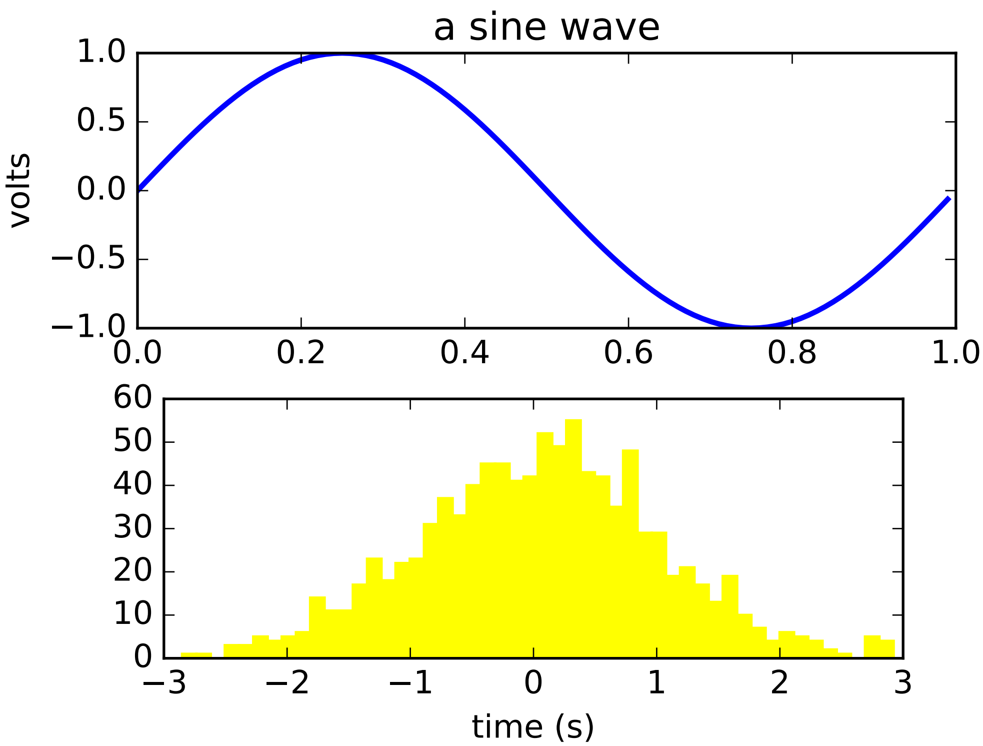
```
del ax.lines[0]
ax.lines.remove(line)  # one or the other, not both!
```

The Axes also has helper methods to configure and decorate the x-axis and y-axis tick, tick labels and axis labels:

```
xtext = ax.set_xlabel('my xdata') # returns a Text instance
ytext = ax.set_ylabel('my ydata')
```

When you call *`ax.set_xlabel`*, it passes the information on the *`Text`* instance of the *`XAxis`*. Each `Axes` instance contains an *`XAxis`* and a *`YAxis`* instance, which handle the layout and drawing of the ticks, tick labels and axis labels.

Try creating the figure below.



4.1.1 Customizing your objects

Every element in the figure is represented by a matplotlib *Artist*, and each has an extensive list of properties to configure its appearance. The figure itself contains a *Rectangle* exactly the size of the figure, which you can use to set the background color and transparency of the figures. Likewise, each *Axes* bounding box (the standard white box with black edges in the typical matplotlib plot, has a `Rectangle` instance that determines the color, transparency, and other properties of the Axes. These instances are stored as member variables `Figure.patch` and `Axes.patch` ("Patch" is a name inherited from MATLAB, and is a 2D "patch" of color on the figure, e.g., rectangles, circles and polygons). Every matplotlib `Artist` has the following properties

Property	Description
alpha	The transparency - a scalar from 0-1
animated	A boolean that is used to facilitate animated drawing
axes	The axes that the Artist lives in, possibly None
clip_box	The bounding box that clips the Artist
clip_on	Whether clipping is enabled
clip_path	The path the artist is clipped to
contains	A picking function to test whether the artist contains the pick point
figure	The figure instance the artist lives in, possibly None
label	A text label (e.g., for auto-labeling)
picker	A python object that controls object picking
transform	The transformation
visible	A boolean whether the artist should be drawn
zorder	A number which determines the drawing order
rasterized	Boolean; Turns vectors into rastergraphics: (for compression & eps transparency)

Each of the properties is accessed with an old-fashioned setter or getter (yes we know this irritates Pythonistas and we plan to support direct access via properties or traits but it hasn't been done yet). For example, to multiply the current alpha by a half:

```
a = o.get_alpha()
o.set_alpha(0.5*a)
```

If you want to set a number of properties at once, you can also use the `set` method with keyword arguments. For example:

```
o.set(alpha=0.5, zorder=2)
```

If you are working interactively at the python shell, a handy way to inspect the `Artist` properties is to use the *matplotlib.artist.getp()* function (simply `getp()` in pylab), which lists the properties and their values. This works for classes derived from `Artist` as well, e.g., `Figure` and `Rectangle`. Here are the `Figure` rectangle properties mentioned above:

```
In [149]: matplotlib.artist.getp(fig.patch)
    alpha = 1.0
    animated = False
    antialiased or aa = True
    axes = None
    clip_box = None
    clip_on = False
    clip_path = None
    contains = None
    edgecolor or ec = w
    facecolor or fc = 0.75
    figure = Figure(8.125x6.125)
    fill = 1
    hatch = None
    height = 1
    label =
    linewidth or lw = 1.0
    picker = None
```

```
    transform = <Affine object at 0x134cca84>
    verts = ((0, 0), (0, 1), (1, 1), (1, 0))
    visible = True
    width = 1
    window_extent = <Bbox object at 0x134acbcc>
    x = 0
    y = 0
    zorder = 1
```

The docstrings for all of the classes also contain the `Artist` properties, so you can consult the interactive "help" or the *artists* for a listing of properties for a given object.

4.1.2 Object containers

Now that we know how to inspect and set the properties of a given object we want to configure, we need to now how to get at that object. As mentioned in the introduction, there are two kinds of objects: primitives and containers. The primitives are usually the things you want to configure (the font of a `Text` instance, the width of a `Line2D`) although the containers also have some properties as well – for example the `Axes Artist` is a container that contains many of the primitives in your plot, but it also has properties like the `xscale` to control whether the xaxis is 'linear' or 'log'. In this section we'll review where the various container objects store the `Artists` that you want to get at.

4.1.3 Figure container

The top level container `Artist` is the `matplotlib.figure.Figure`, and it contains everything in the figure. The background of the figure is a `Rectangle` which is stored in `Figure.patch`. As you add subplots (`add_subplot()`) and axes (`add_axes()`) to the figure these will be appended to the `Figure.axes`. These are also returned by the methods that create them:

```
In [156]: fig = plt.figure()

In [157]: ax1 = fig.add_subplot(211)

In [158]: ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])

In [159]: ax1
Out[159]: <matplotlib.axes.Subplot instance at 0xd54b26c>

In [160]: print fig.axes
[<matplotlib.axes.Subplot instance at 0xd54b26c>, <matplotlib.axes.Axes instance at␣
↪0xd3f0b2c>]
```

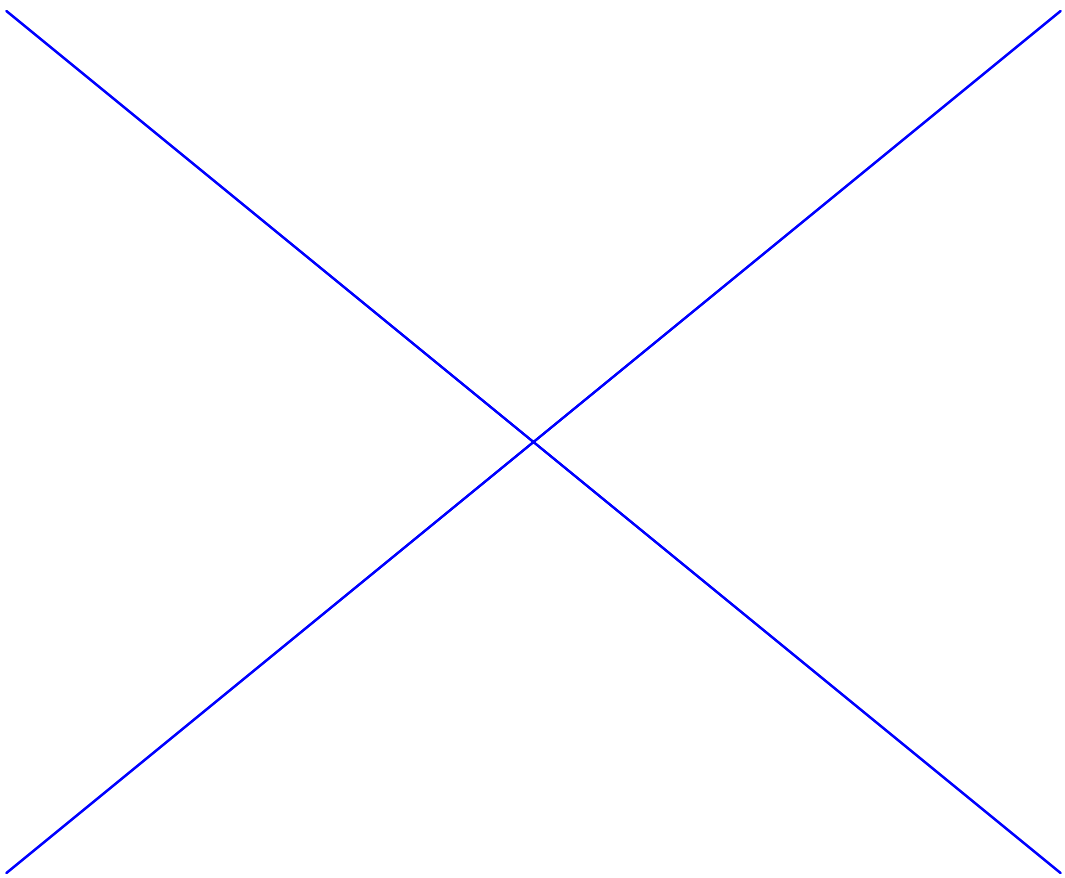
Because the figure maintains the concept of the "current axes" (see `Figure.gca` and `Figure.sca`) to support the pylab/pyplot state machine, you should not insert or remove axes directly from the axes list, but rather use the `add_subplot()` and `add_axes()` methods to insert, and the `delaxes()` method to delete. You are free however, to iterate over the list of axes or index into it to get access to `Axes` instances you want to customize. Here is an example which turns all the axes grids on:

```
for ax in fig.axes:
    ax.grid(True)
```

The figure also has its own text, lines, patches and images, which you can use to add primitives directly. The default coordinate system for the `Figure` will simply be in pixels (which is not usually what you want) but you can control this by setting the transform property of the `Artist` you are adding to the figure.

More useful is "figure coordinates" where (0, 0) is the bottom-left of the figure and (1, 1) is the top-right of the figure which you can obtain by setting the `Artist` transform to `fig.transFigure`:

```
In [191]: fig = plt.figure()

In [192]: l1 = matplotlib.lines.Line2D([0, 1], [0, 1],
           transform=fig.transFigure, figure=fig)

In [193]: l2 = matplotlib.lines.Line2D([0, 1], [1, 0],
           transform=fig.transFigure, figure=fig)

In [194]: fig.lines.extend([l1, l2])

In [195]: fig.canvas.draw()
```



Here is a summary of the Artists the figure contains

Figure attribute	Description
axes	A list of Axes instances (includes Subplot)
patch	The Rectangle background
images	A list of FigureImages patches - useful for raw pixel display
legends	A list of Figure Legend instances (different from Axes.legends)
lines	A list of Figure Line2D instances (rarely used, see Axes.lines)
patches	A list of Figure patches (rarely used, see Axes.patches)
texts	A list Figure Text instances

4.1.4 Axes container

The *matplotlib.axes.Axes* is the center of the matplotlib universe – it contains the vast majority of all the `Artists` used in a figure with many helper methods to create and add these `Artists` to itself, as well as helper methods to access and customize the `Artists` it contains. Like the *Figure*, it contains a *Patch* `patch` which is a *Rectangle* for Cartesian coordinates and a *Circle* for polar coordinates; this patch determines the shape, background and border of the plotting region:

```
ax = fig.add_subplot(111)
rect = ax.patch  # a Rectangle instance
rect.set_facecolor('green')
```

When you call a plotting method, e.g., the canonical *plot()* and pass in arrays or lists of values, the method will create a *matplotlib.lines.Line2D()* instance, update the line with all the `Line2D` properties passed as keyword arguments, add the line to the `Axes.lines` container, and returns it to you:

```
In [213]: x, y = np.random.rand(2, 100)

In [214]: line, = ax.plot(x, y, '-', color='blue', linewidth=2)
```

`plot` returns a list of lines because you can pass in multiple x, y pairs to plot, and we are unpacking the first element of the length one list into the line variable. The line has been added to the `Axes.lines` list:

```
In [229]: print ax.lines
[<matplotlib.lines.Line2D instance at 0xd378b0c>]
```

Similarly, methods that create patches, like *bar()* creates a list of rectangles, will add the patches to the `Axes.patches` list:

```
In [233]: n, bins, rectangles = ax.hist(np.random.randn(1000), 50, facecolor='yellow')

In [234]: rectangles
Out[234]: <a list of 50 Patch objects>

In [235]: print len(ax.patches)
```

You should not add objects directly to the `Axes.lines` or `Axes.patches` lists unless you know exactly what you are doing, because the `Axes` needs to do a few things when it creates and adds an object. It sets the figure and axes property of the `Artist`, as well as the default `Axes` transformation (unless a transformation is set). It also inspects the data contained in the `Artist` to update the data structures controlling auto-scaling, so that the view limits can be adjusted to contain the plotted data. You can, nonetheless, create objects yourself and add them directly to the Axes using helper methods like *add_line()* and *add_patch()*. Here is an annotated interactive session illustrating what is going on:

```
In [261]: fig = plt.figure()

In [262]: ax = fig.add_subplot(111)

# create a rectangle instance
In [263]: rect = matplotlib.patches.Rectangle( (1,1), width=5, height=12)

# by default the axes instance is None
In [264]: print rect.get_axes()
None

# and the transformation instance is set to the "identity transform"
In [265]: print rect.get_transform()
<Affine object at 0x13695544>

# now we add the Rectangle to the Axes
In [266]: ax.add_patch(rect)

# and notice that the ax.add_patch method has set the axes
# instance
In [267]: print rect.get_axes()
Axes(0.125,0.1;0.775x0.8)

# and the transformation has been set too
In [268]: print rect.get_transform()
<Affine object at 0x15009ca4>

# the default axes transformation is ax.transData
In [269]: print ax.transData
<Affine object at 0x15009ca4>

# notice that the xlimits of the Axes have not been changed
In [270]: print ax.get_xlim()
(0.0, 1.0)

# but the data limits have been updated to encompass the rectangle
In [271]: print ax.dataLim.bounds
(1.0, 1.0, 5.0, 12.0)

# we can manually invoke the auto-scaling machinery
In [272]: ax.autoscale_view()

# and now the xlim are updated to encompass the rectangle
In [273]: print ax.get_xlim()
(1.0, 6.0)

# we have to manually force a figure draw
In [274]: ax.figure.canvas.draw()
```

There are many, many Axes helper methods for creating primitive Artists and adding them to their respective containers. The table below summarizes a small sampling of them, the kinds of Artist they create, and where they store them

Helper method	Artist	Container
ax.annotate - text annotations	Annotate	ax.texts
ax.bar - bar charts	Rectangle	ax.patches
ax.errorbar - error bar plots	Line2D and Rectangle	ax.lines and ax.patches
ax.fill - shared area	Polygon	ax.patches
ax.hist - histograms	Rectangle	ax.patches
ax.imshow - image data	AxesImage	ax.images
ax.legend - axes legends	Legend	ax.legends
ax.plot - xy plots	Line2D	ax.lines
ax.scatter - scatter charts	PolygonCollection	ax.collections
ax.text - text	Text	ax.texts

In addition to all of these Artists, the Axes contains two important Artist containers: the *XAxis* and *YAxis*, which handle the drawing of the ticks and labels. These are stored as instance variables xaxis and yaxis. The XAxis and YAxis containers will be detailed below, but note that the Axes contains many helper methods which forward calls on to the *Axis* instances so you often do not need to work with them directly unless you want to. For example, you can set the font size of the XAxis ticklabels using the Axes helper method:

```
for label in ax.get_xticklabels():
    label.set_color('orange')
```

Below is a summary of the Artists that the Axes contains

Axes attribute	Description
artists	A list of Artist instances
patch	Rectangle instance for Axes background
collections	A list of Collection instances
images	A list of AxesImage
legends	A list of Legend instances
lines	A list of Line2D instances
patches	A list of Patch instances
texts	A list of Text instances
xaxis	matplotlib.axis.XAxis instance
yaxis	matplotlib.axis.YAxis instance

4.1.5 Axis containers

The *matplotlib.axis.Axis* instances handle the drawing of the tick lines, the grid lines, the tick labels and the axis label. You can configure the left and right ticks separately for the y-axis, and the upper and lower ticks separately for the x-axis. The Axis also stores the data and view intervals used in auto-scaling, panning and zooming, as well as the *Locator* and *Formatter* instances which control where the ticks are placed and how they are represented as strings.

Each Axis object contains a label attribute (this is what pylab modifies in calls to xlabel() and ylabel()) as well as a list of major and minor ticks. The ticks are *XTick* and *YTick* instances, which contain the actual line and text primitives that render the ticks and ticklabels. Because the ticks are dynamically created as needed (e.g., when panning and zooming), you should access the lists of major and minor ticks through their accessor methods *get_major_ticks()* and *get_minor_ticks()*. Although the ticks contain all the primitives and will be covered below, the Axis methods contain accessor methods to return the tick lines, tick labels, tick locations etc.:

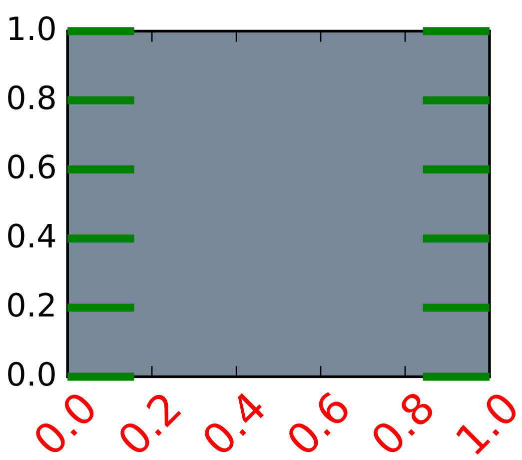
```
In [285]: axis = ax.xaxis

In [286]: axis.get_ticklocs()
Out[286]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

In [287]: axis.get_ticklabels()
Out[287]: <a list of 10 Text major ticklabel objects>

# note there are twice as many ticklines as labels because by
#  default there are tick lines at the top and bottom but only tick
#  labels below the xaxis; this can be customized
In [288]: axis.get_ticklines()
Out[288]: <a list of 20 Line2D ticklines objects>

# by default you get the major ticks back
In [291]: axis.get_ticklines()
Out[291]: <a list of 20 Line2D ticklines objects>

# but you can also ask for the minor ticks
In [292]: axis.get_ticklines(minor=True)
Out[292]: <a list of 0 Line2D ticklines objects>
```

Here is a summary of some of the useful accessor methods of the Axis (these have corresponding setters where useful, such as set_major_formatter)

Accessor method	Description
get_scale	The scale of the axis, e.g., 'log' or 'linear'
get_view_interval	The interval instance of the axis view limits
get_data_interval	The interval instance of the axis data limits
get_gridlines	A list of grid lines for the Axis
get_label	The axis label - a Text instance
get_ticklabels	A list of Text instances - keyword minor=True|False
get_ticklines	A list of Line2D instances - keyword minor=True|False
get_ticklocs	A list of Tick locations - keyword minor=True|False
get_major_locator	The matplotlib.ticker.Locator instance for major ticks
get_major_formatter	The matplotlib.ticker.Formatter instance for major ticks
get_minor_locator	The matplotlib.ticker.Locator instance for minor ticks
get_minor_formatter	The matplotlib.ticker.Formatter instance for minor ticks
get_major_ticks	A list of Tick instances for major ticks
get_minor_ticks	A list of Tick instances for minor ticks
grid	Turn the grid on or off for the major or minor ticks

Here is an example, not recommended for its beauty, which customizes the axes and tick properties

```python
import numpy as np
import matplotlib.pyplot as plt

# plt.figure creates a matplotlib.figure.Figure instance
fig = plt.figure()
rect = fig.patch # a rectangle instance
rect.set_facecolor('lightgoldenrodyellow')

ax1 = fig.add_axes([0.1, 0.3, 0.4, 0.4])
rect = ax1.patch
rect.set_facecolor('lightslategray')


for label in ax1.xaxis.get_ticklabels():
    # label is a Text instance
    label.set_color('red')
    label.set_rotation(45)
    label.set_fontsize(16)

for line in ax1.yaxis.get_ticklines():
    # line is a Line2D instance
    line.set_color('green')
    line.set_markersize(25)
    line.set_markeredgewidth(3)
```



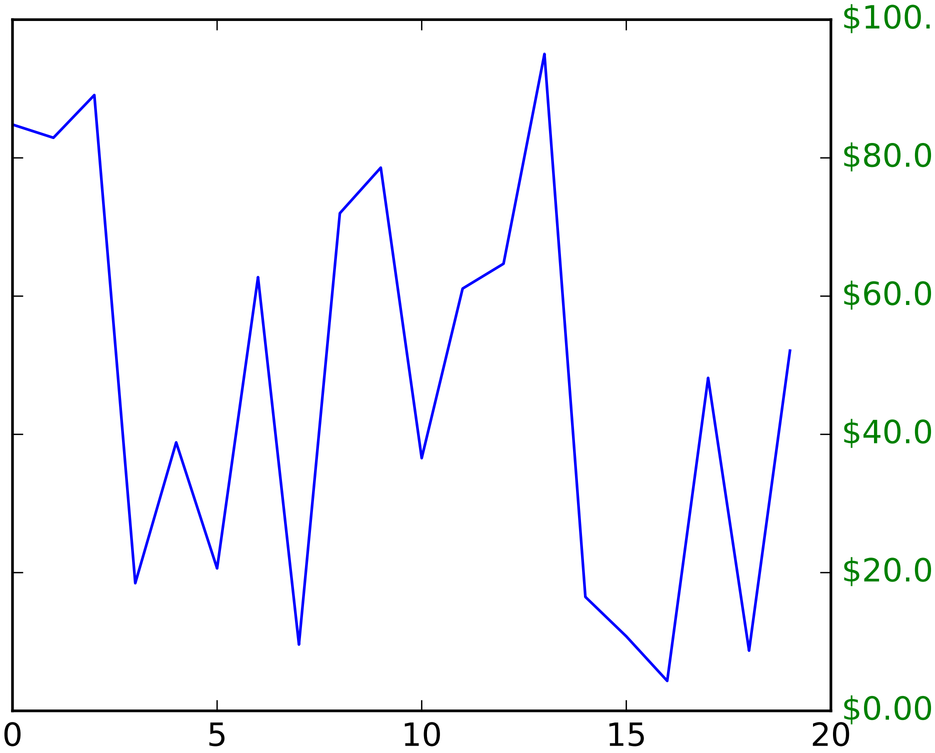
4.1.6 Tick containers

The *matplotlib.axis.Tick* is the final container object in our descent from the *Figure* to the *Axes* to the *Axis* to the *Tick*. The `Tick` contains the tick and grid line instances, as well as the label instances for the upper and lower ticks. Each of these is accessible directly as an attribute of the `Tick`. In addition, there are boolean variables that determine whether the upper labels and ticks are on for the x-axis and whether the right labels and ticks are on for the y-axis.

Tick attribute	Description
tick1line	Line2D instance
tick2line	Line2D instance
gridline	Line2D instance
label1	Text instance
label2	Text instance
gridOn	boolean which determines whether to draw the tickline
tick1On	boolean which determines whether to draw the 1st tickline
tick2On	boolean which determines whether to draw the 2nd tickline
label1On	boolean which determines whether to draw tick label
label2On	boolean which determines whether to draw tick label

Here is an example which sets the formatter for the right side ticks with dollar signs and colors them green on the right side of the yaxis

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(100*np.random.rand(20))

formatter = ticker.FormatStrFormatter('$%1.2f')
ax.yaxis.set_major_formatter(formatter)

for tick in ax.yaxis.get_major_ticks():
    tick.label1On = False
    tick.label2On = True
    tick.label2.set_color('green')
```



4.2 Customizing Location of Subplot Using GridSpec

GridSpec specifies the geometry of the grid that a subplot will be placed. The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

SubplotSpec specifies the location of the subplot in the given *GridSpec*.

subplot2grid a helper function that is similar to "pyplot.subplot" but uses 0-based indexing and let subplot to occupy multiple cells.

4.2.1 Basic Example of using subplot2grid

To use subplot2grid, you provide geometry of the grid and the location of the subplot in the grid. For a simple single-cell subplot:

```
ax = plt.subplot2grid((2,2),(0, 0))
```

is identical to

```
ax = plt.subplot(2,2,1)
```

Note that, unlike matplotlib's subplot, the index starts from 0 in gridspec.

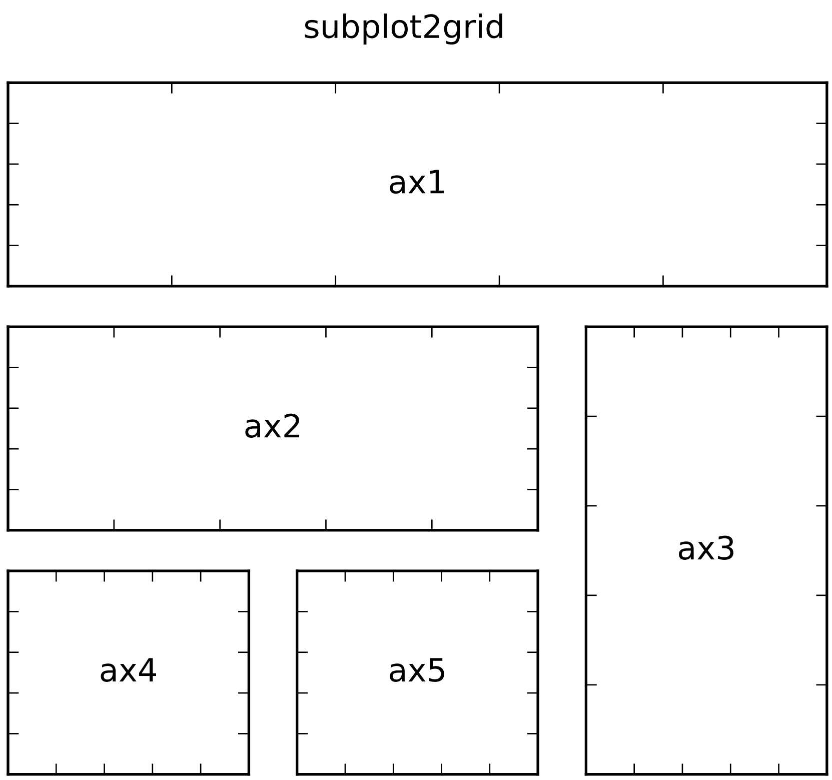
To create a subplot that spans multiple cells,

```
ax2 = plt.subplot2grid((3,3), (1, 0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1, 2), rowspan=2)
```

For example, the following commands

```
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1, 2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2, 0))
ax5 = plt.subplot2grid((3,3), (2, 1))
```

creates



4.2.2 GridSpec and SubplotSpec

You can create GridSpec explicitly and use them to create a Subplot.

For example,

```
ax = plt.subplot2grid((2,2),(0, 0))
```

is equal to

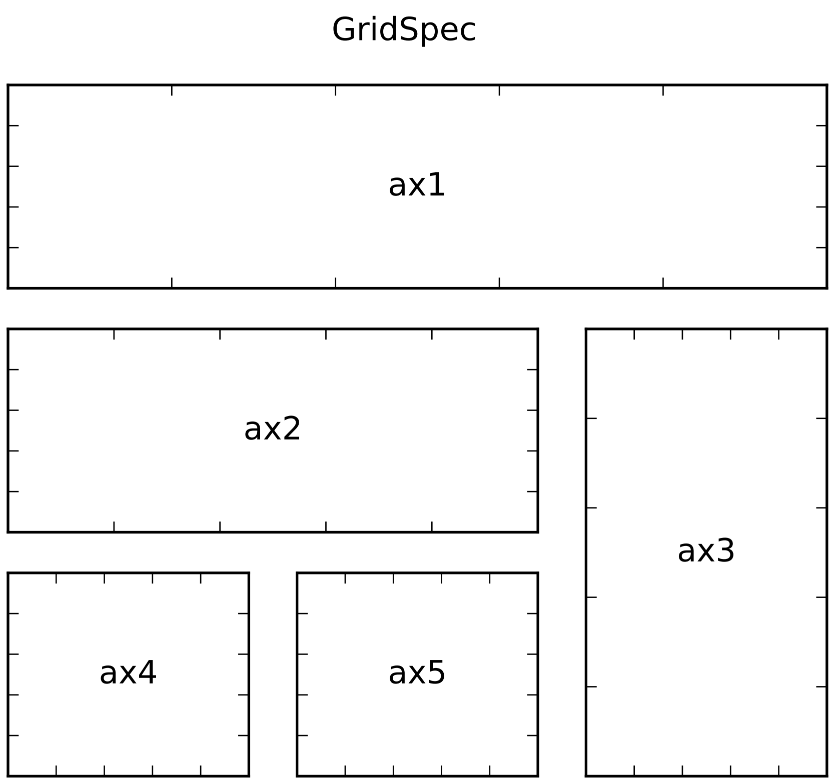
```
import matplotlib.gridspec as gridspec
gs = gridspec.GridSpec(2, 2)
ax = plt.subplot(gs[0, 0])
```

A gridspec instance provides array-like (2d or 1d) indexing that returns the SubplotSpec instance. For, SubplotSpec that spans multiple cells, use slice.

```
ax2 = plt.subplot(gs[1,:-1])
ax3 = plt.subplot(gs[1:, -1])
```

The above example becomes

```
gs = gridspec.GridSpec(3, 3)
ax1 = plt.subplot(gs[0, :])
ax2 = plt.subplot(gs[1,:-1])
ax3 = plt.subplot(gs[1:, -1])
ax4 = plt.subplot(gs[-1,0])
ax5 = plt.subplot(gs[-1,-2])
```



4.2.3 Adjust GridSpec layout

When a GridSpec is explicitly used, you can adjust the layout parameters of subplots that are created from the gridspec.

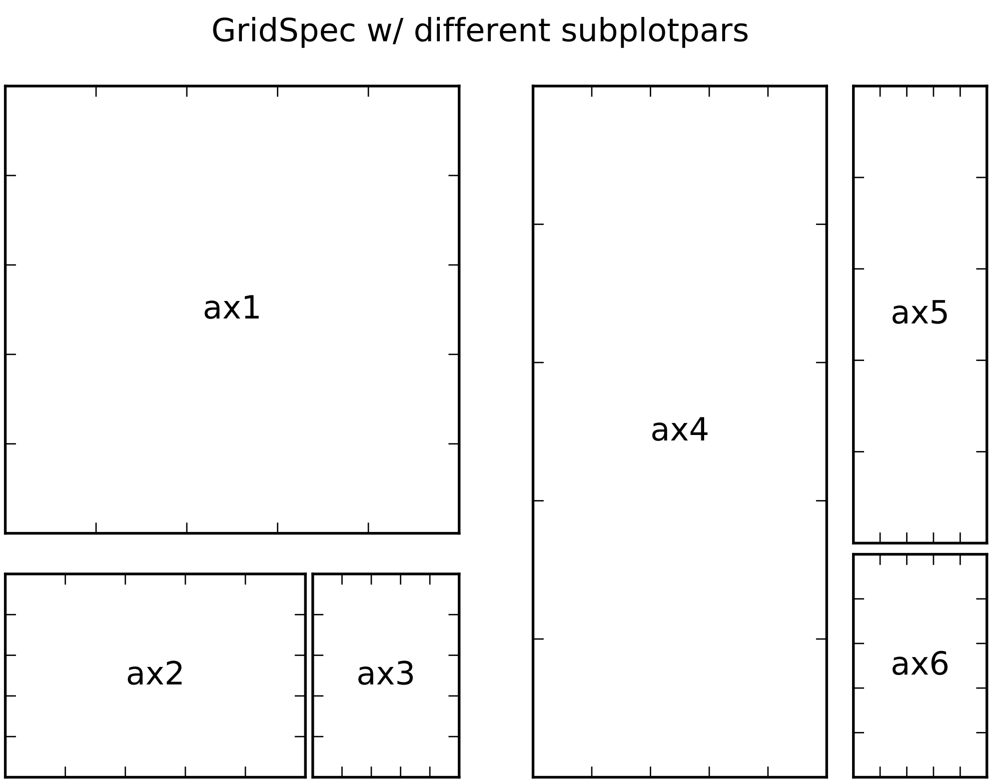
```
gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)
```

This is similar to *subplots_adjust*, but it only affects the subplots that are created from the given GridSpec.

The code below

```
gs1 = gridspec.GridSpec(3, 3)
gs1.update(left=0.05, right=0.48, wspace=0.05)
ax1 = plt.subplot(gs1[:-1, :])
ax2 = plt.subplot(gs1[-1, :-1])
ax3 = plt.subplot(gs1[-1, -1])

gs2 = gridspec.GridSpec(3, 3)
gs2.update(left=0.55, right=0.98, hspace=0.05)
ax4 = plt.subplot(gs2[:, :-1])
ax5 = plt.subplot(gs2[:-1, -1])
ax6 = plt.subplot(gs2[-1, -1])
```

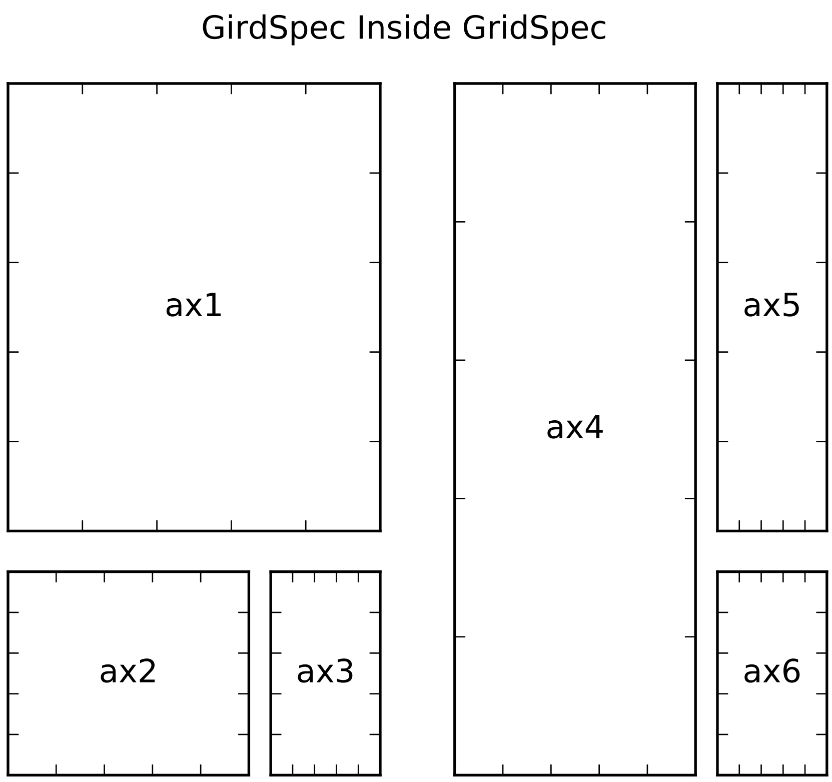
creates



4.2.4 GridSpec using SubplotSpec

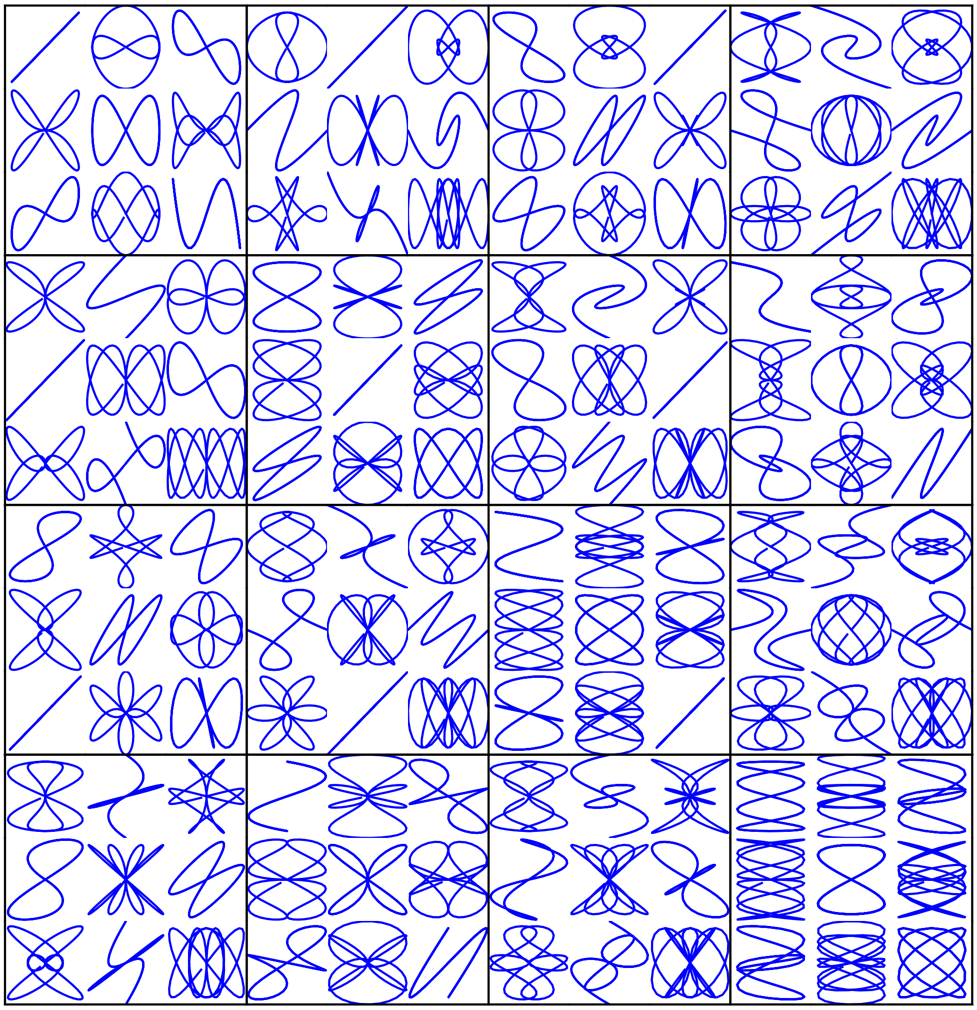
You can create GridSpec from the SubplotSpec, in which case its layout parameters are set to that of the location of the given SubplotSpec.

```
gs0 = gridspec.GridSpec(1, 2)

gs00 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[0])
gs01 = gridspec.GridSpecFromSubplotSpec(3, 3, subplot_spec=gs0[1])
```



4.2.5 A Complex Nested GridSpec using SubplotSpec

Here's a more sophisticated example of nested gridspec where we put a box around each cell of the outer 4x4 grid, by hiding appropriate spines in each of the inner 3x3 grids.

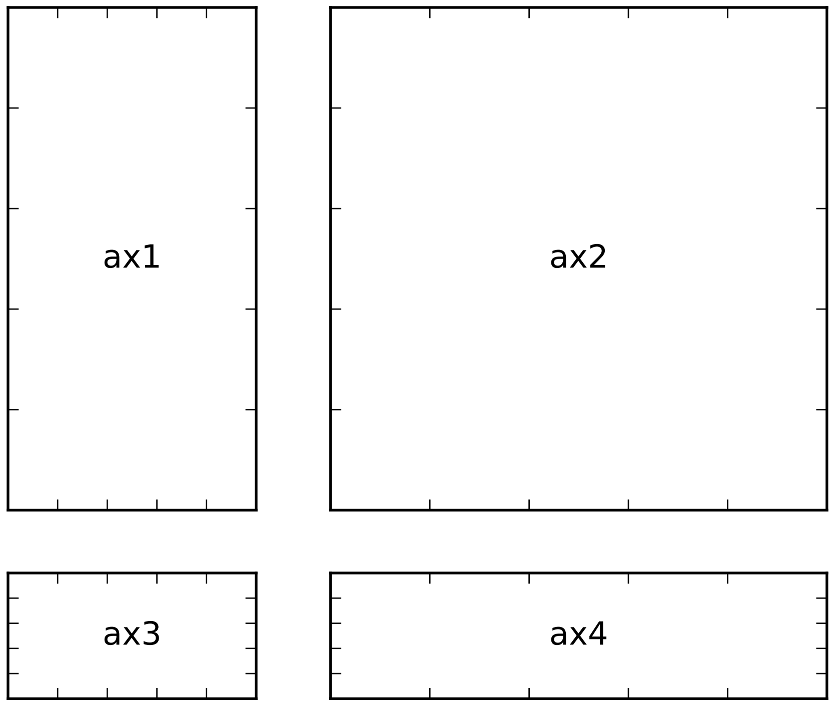


4.2.6 GridSpec with Varying Cell Sizes

By default, GridSpec creates cells of equal sizes. You can adjust relative heights and widths of rows and columns. Note that absolute values are meaningless, only their relative ratios matter.

```
gs = gridspec.GridSpec(2, 2,
                       width_ratios=[1,2],
                       height_ratios=[4,1]
                       )

ax1 = plt.subplot(gs[0])
ax2 = plt.subplot(gs[1])
```

```
ax3 = plt.subplot(gs[2])
ax4 = plt.subplot(gs[3])
```



4.3 Tight Layout guide

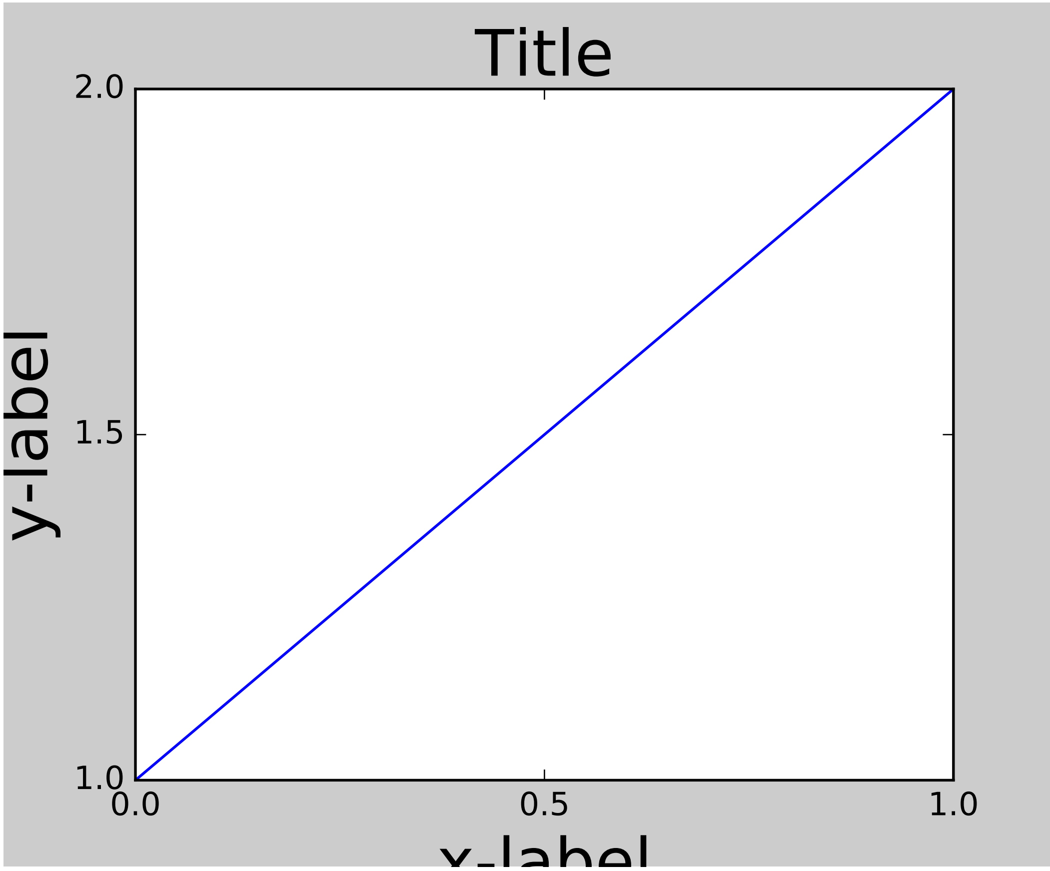
tight_layout automatically adjusts subplot params so that the subplot(s) fits in to the figure area. This is an experimental feature and may not work for some cases. It only checks the extents of ticklabels, axis labels, and titles.

4.3.1 Simple Example

In matplotlib, the location of axes (including subplots) are specified in normalized figure coordinates. It can happen that your axis labels or titles (or sometimes even ticklabels) go outside the figure area, and are thus clipped.

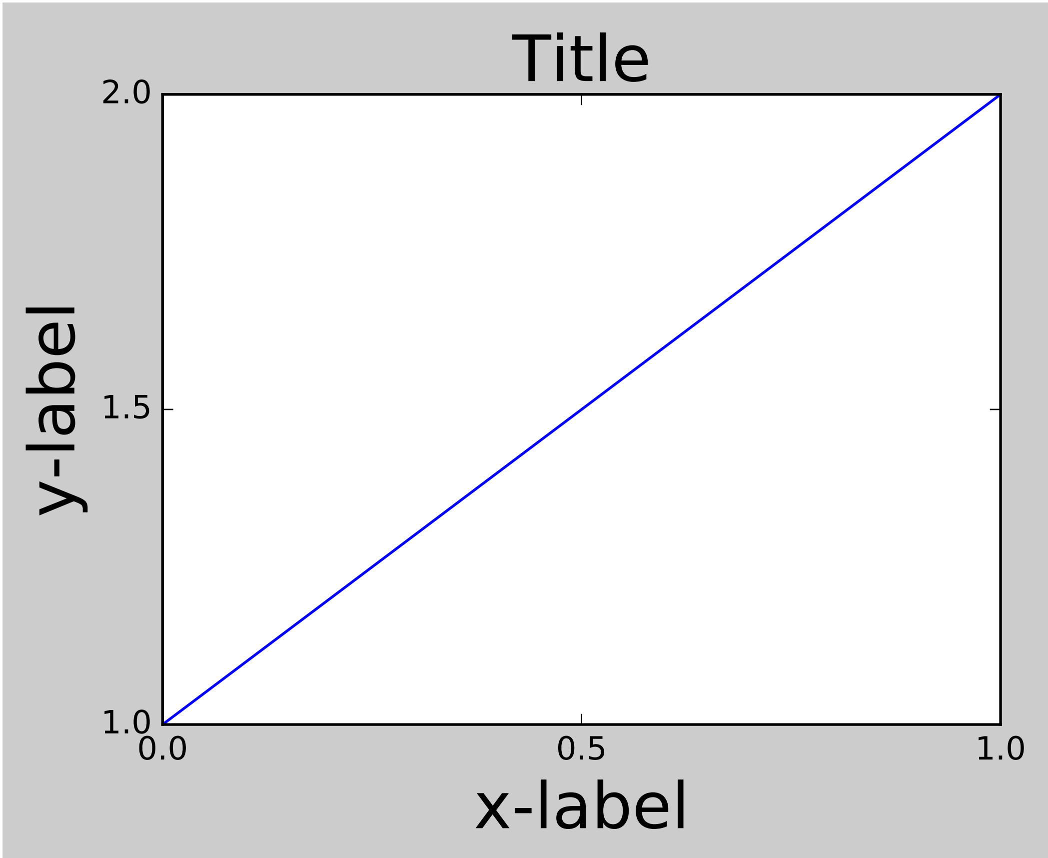
```
plt.rcParams['savefig.facecolor'] = "0.8"

def example_plot(ax, fontsize=12):
    ax.plot([1, 2])
    ax.locator_params(nbins=3)
    ax.set_xlabel('x-label', fontsize=fontsize)
```

```
    ax.set_ylabel('y-label', fontsize=fontsize)
    ax.set_title('Title', fontsize=fontsize)

plt.close('all')
fig, ax = plt.subplots()
example_plot(ax, fontsize=24)
```



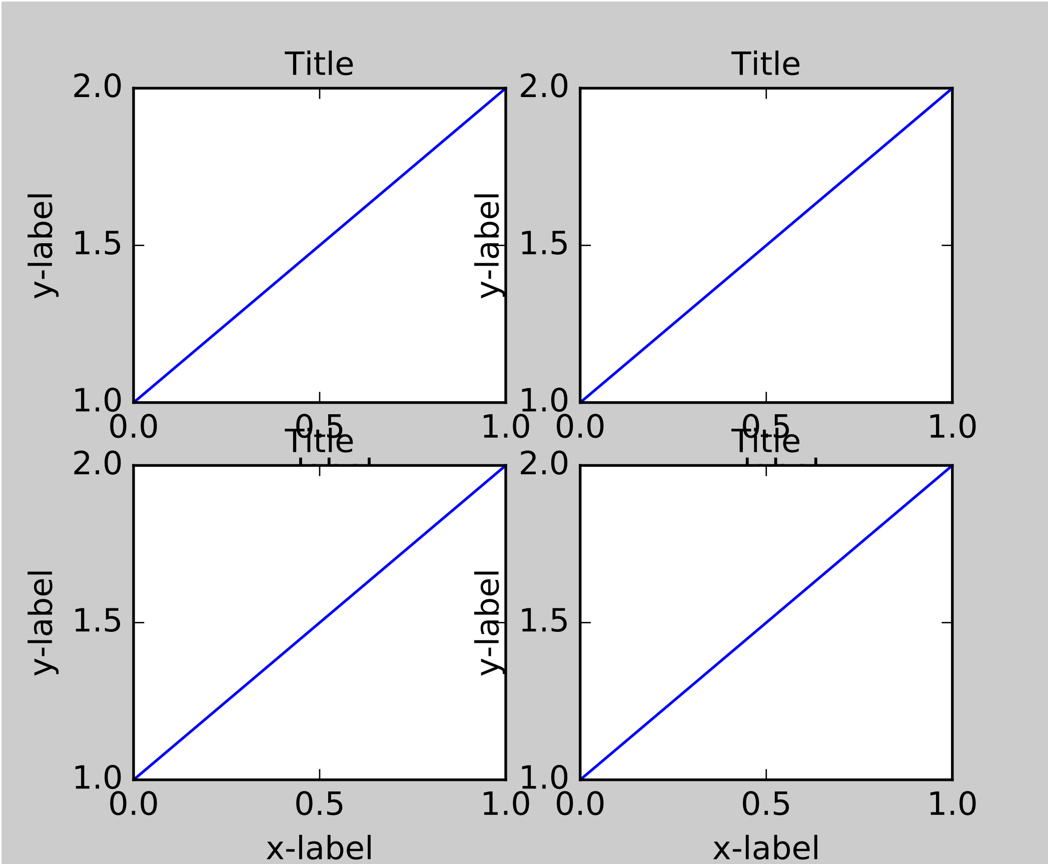
To prevent this, the location of axes needs to be adjusted. For subplots, this can be done by adjusting the subplot params (*Move the edge of an axes to make room for tick labels*). Matplotlib v1.1 introduces a new command `tight_layout()` that does this automatically for you.

```
plt.tight_layout()
```



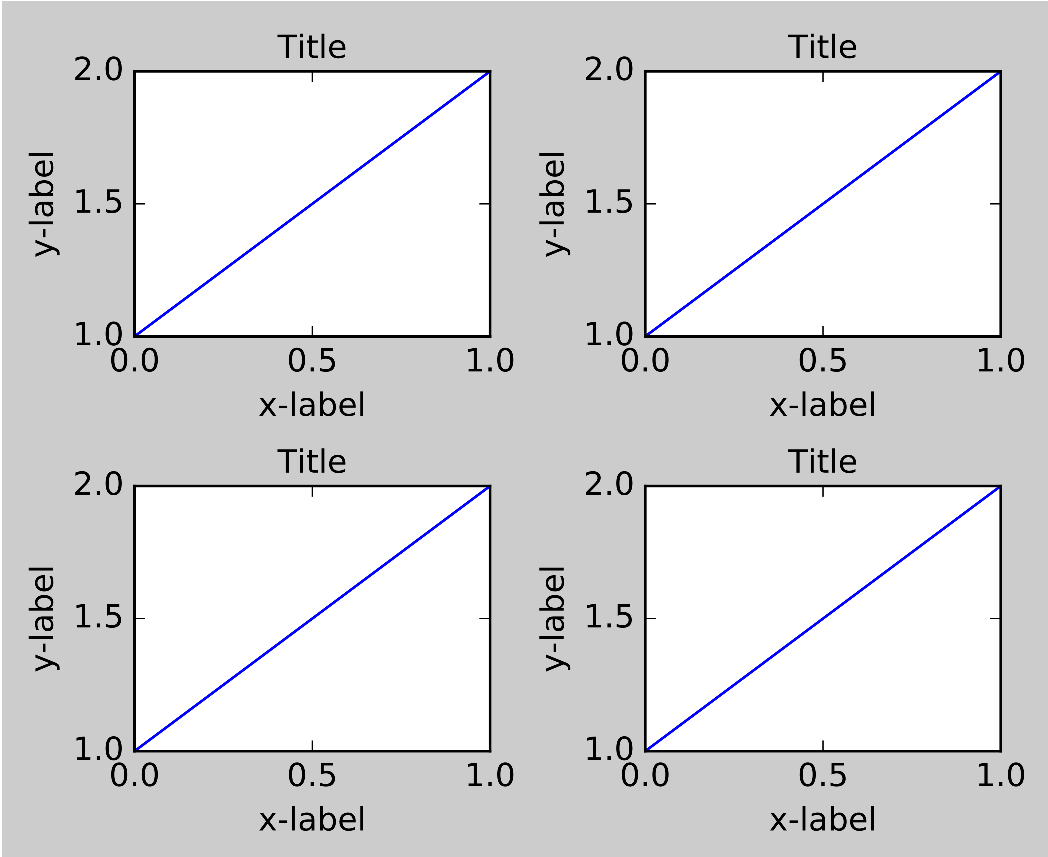
When you have multiple subplots, often you see labels of different axes overlapping each other.

```
plt.close('all')
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
```



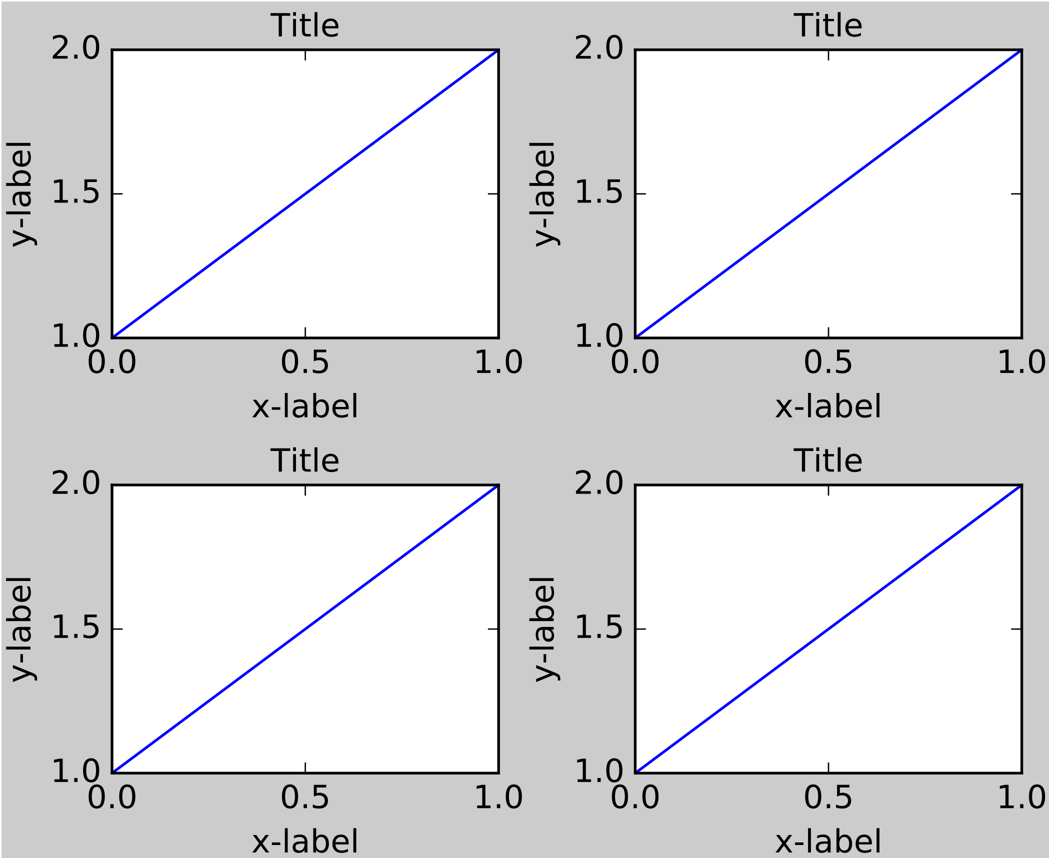
tight_layout() will also adjust spacing between subplots to minimize the overlaps.

```
plt.tight_layout()
```



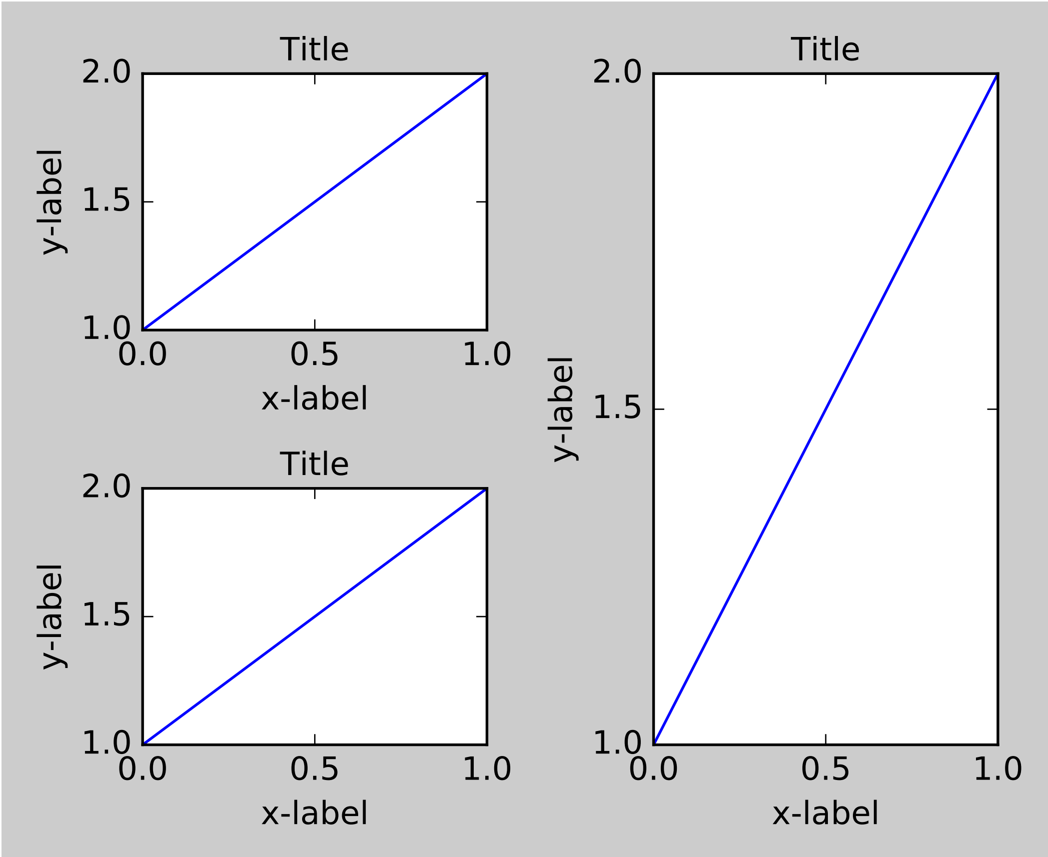
tight_layout() can take keyword arguments of *pad*, *w_pad* and *h_pad*. These control the extra padding around the figure border and between subplots. The pads are specified in fraction of fontsize.

```
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
```



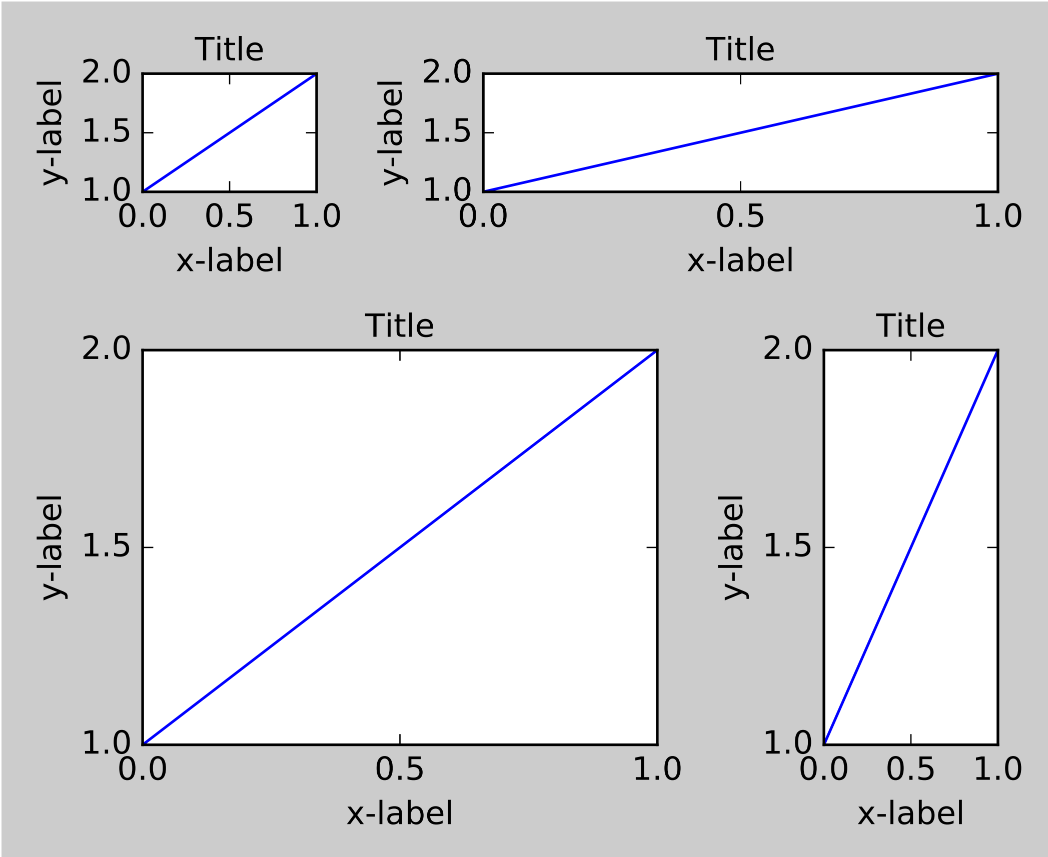
tight_layout() will work even if the sizes of subplots are different as far as their grid specification is compatible. In the example below, *ax1* and *ax2* are subplots of a 2x2 grid, while *ax3* is of a 1x2 grid.

```
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot(221)
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)

plt.tight_layout()
```



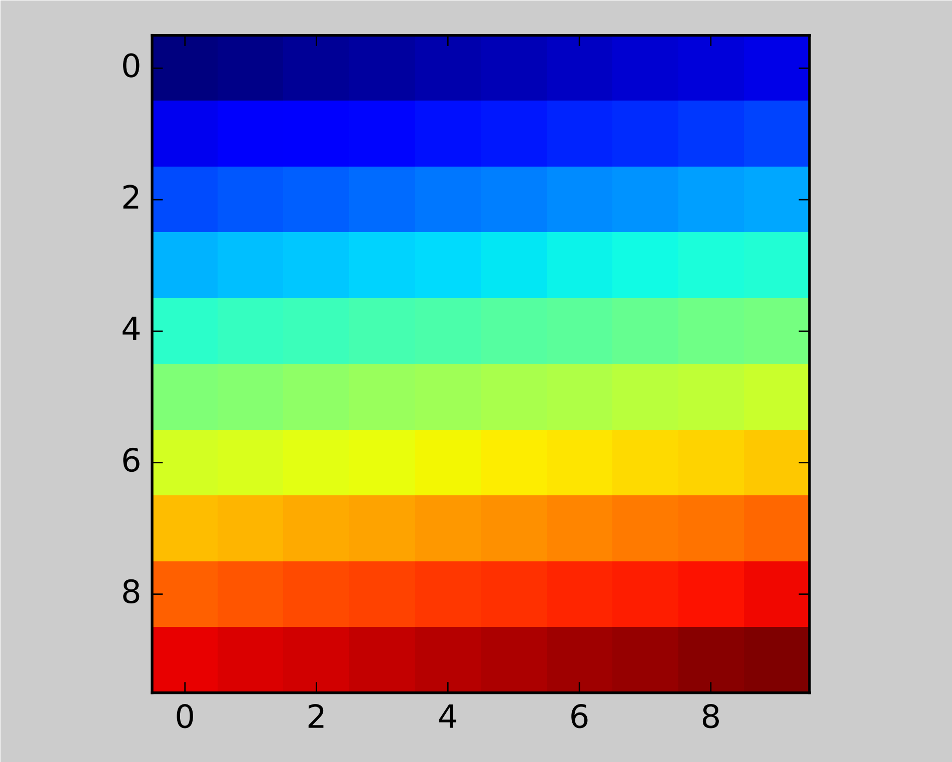
It works with subplots created with *subplot2grid()*. In general, subplots created from the gridspec (*Customizing Location of Subplot Using GridSpec*) will work.

```
plt.close('all')
fig = plt.figure()

ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)

plt.tight_layout()
```



Although not thoroughly tested, it seems to work for subplots with aspect != "auto" (e.g., axes with images).

```
arr = np.arange(100).reshape((10,10))

plt.close('all')
fig = plt.figure(figsize=(5,4))

ax = plt.subplot(111)
im = ax.imshow(arr, interpolation="none")

plt.tight_layout()
```



Caveats

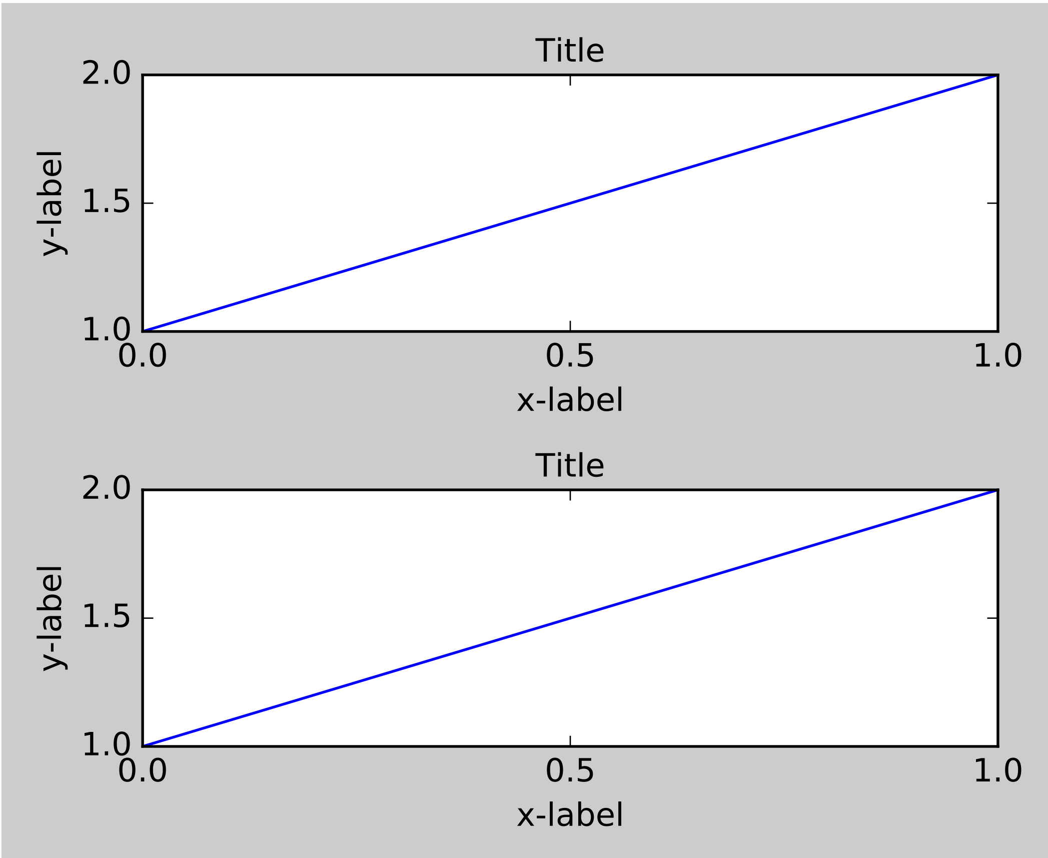
- *tight_layout()* only considers ticklabels, axis labels, and titles. Thus, other artists may be clipped and also may overlap.
- It assumes that the extra space needed for ticklabels, axis labels, and titles is independent of original location of axes. This is often true, but there are rare cases where it is not.
- pad=0 clips some of the texts by a few pixels. This may be a bug or a limitation of the current algorithm and it is not clear why it happens. Meanwhile, use of pad at least larger than 0.3 is recommended.

Use with GridSpec

GridSpec has its own *tight_layout()* method (the pyplot api *tight_layout()* also works).

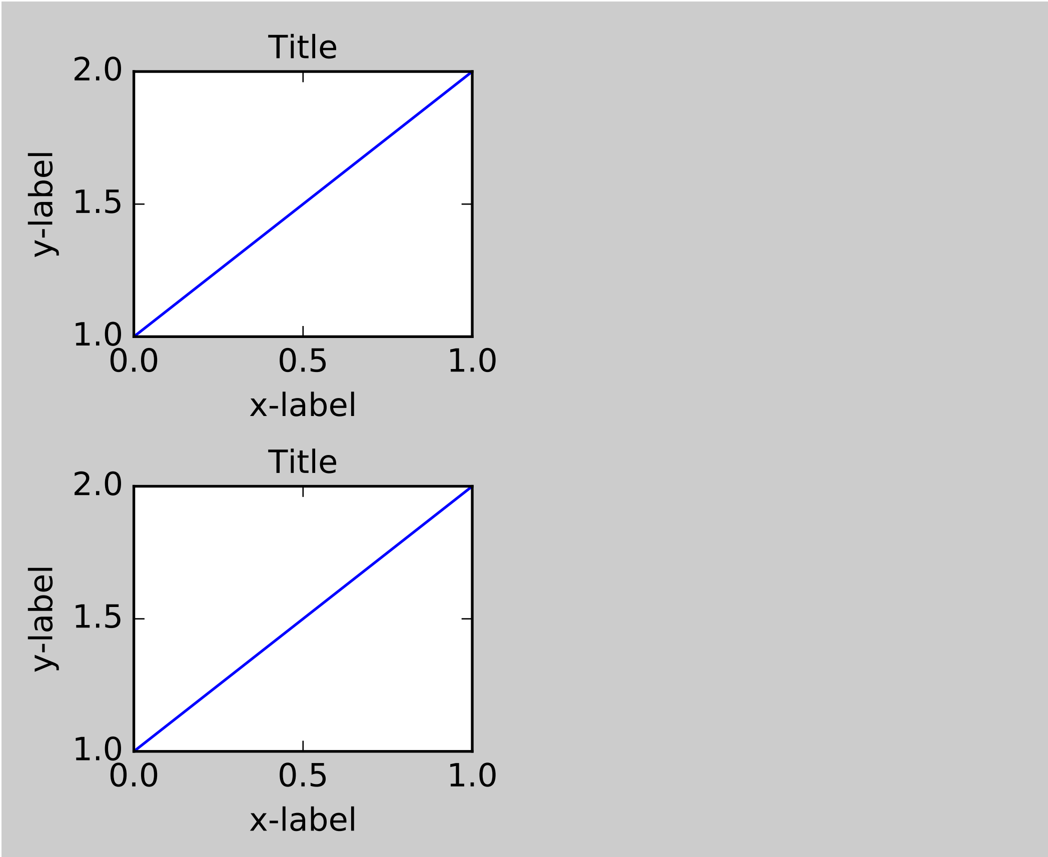
```
plt.close('all')
fig = plt.figure()

import matplotlib.gridspec as gridspec

gs1 = gridspec.GridSpec(2, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])

example_plot(ax1)
```

```
example_plot(ax2)

gs1.tight_layout(fig)
```



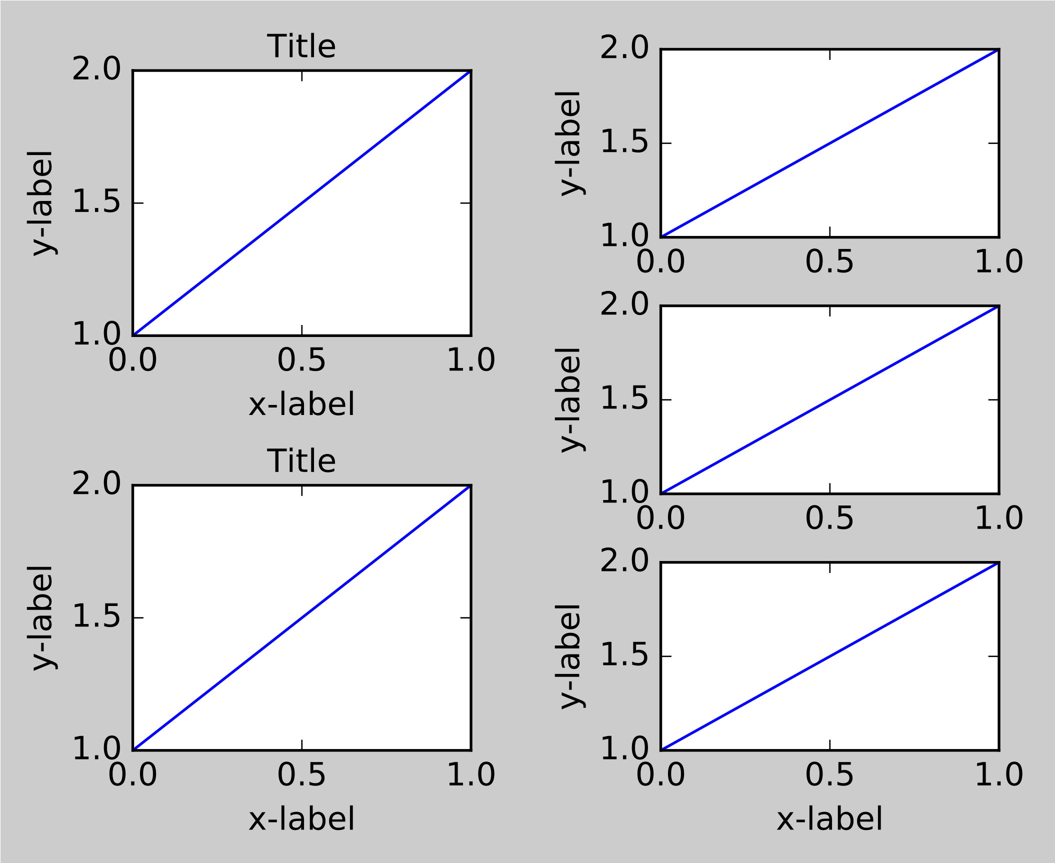
You may provide an optional *rect* parameter, which specifies the bounding box that the subplots will be fit inside. The coordinates must be in normalized figure coordinates and the default is (0, 0, 1, 1).

```
gs1.tight_layout(fig, rect=[0, 0, 0.5, 1])
```



For example, this can be used for a figure with multiple gridspecs.

```
gs2 = gridspec.GridSpec(3, 1)

for ss in gs2:
    ax = fig.add_subplot(ss)
    example_plot(ax)
    ax.set_title("")
    ax.set_xlabel("")

ax.set_xlabel("x-label", fontsize=12)

gs2.tight_layout(fig, rect=[0.5, 0, 1, 1], h_pad=0.5)
```

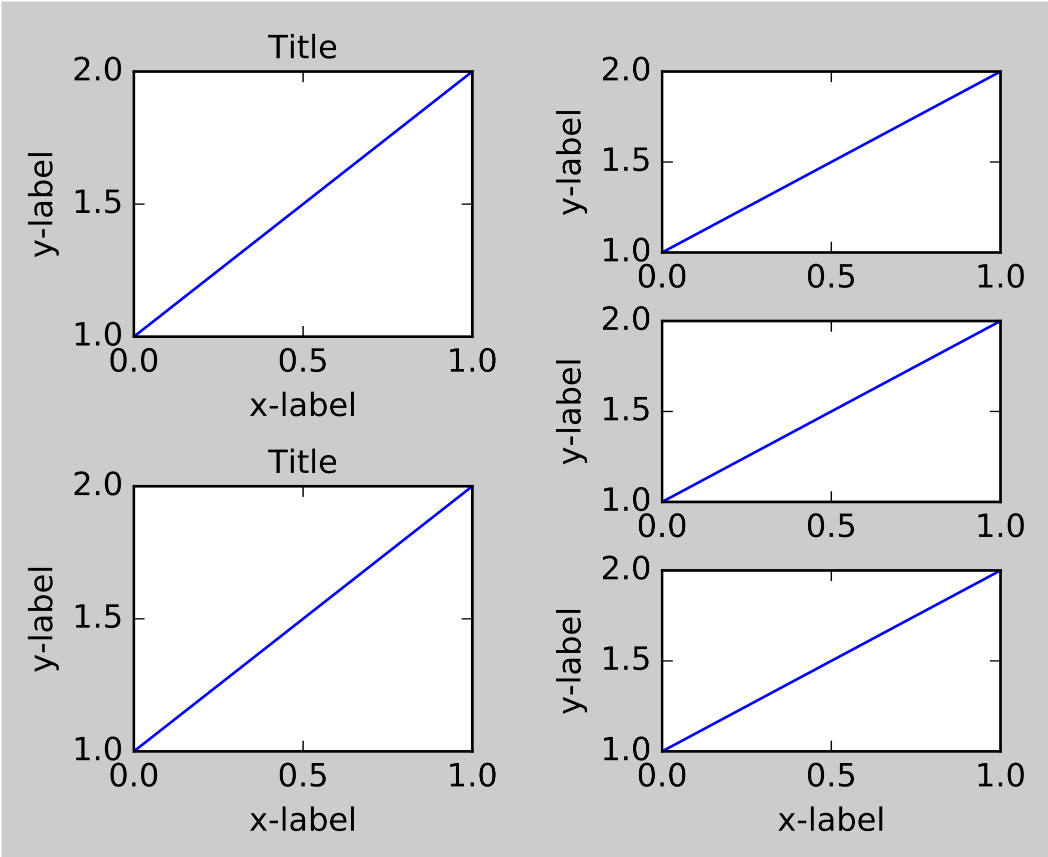


We may try to match the top and bottom of two grids

```
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)
```

While this should be mostly good enough, adjusting top and bottom may require adjustment of hspace also. To update hspace & vspace, we call `tight_layout()` again with updated rect argument. Note that the rect argument specifies the area including the ticklabels, etc. Thus, we will increase the bottom (which is 0 for the normal case) by the difference between the *bottom* from above and the bottom of each gridspec. Same thing for the top.

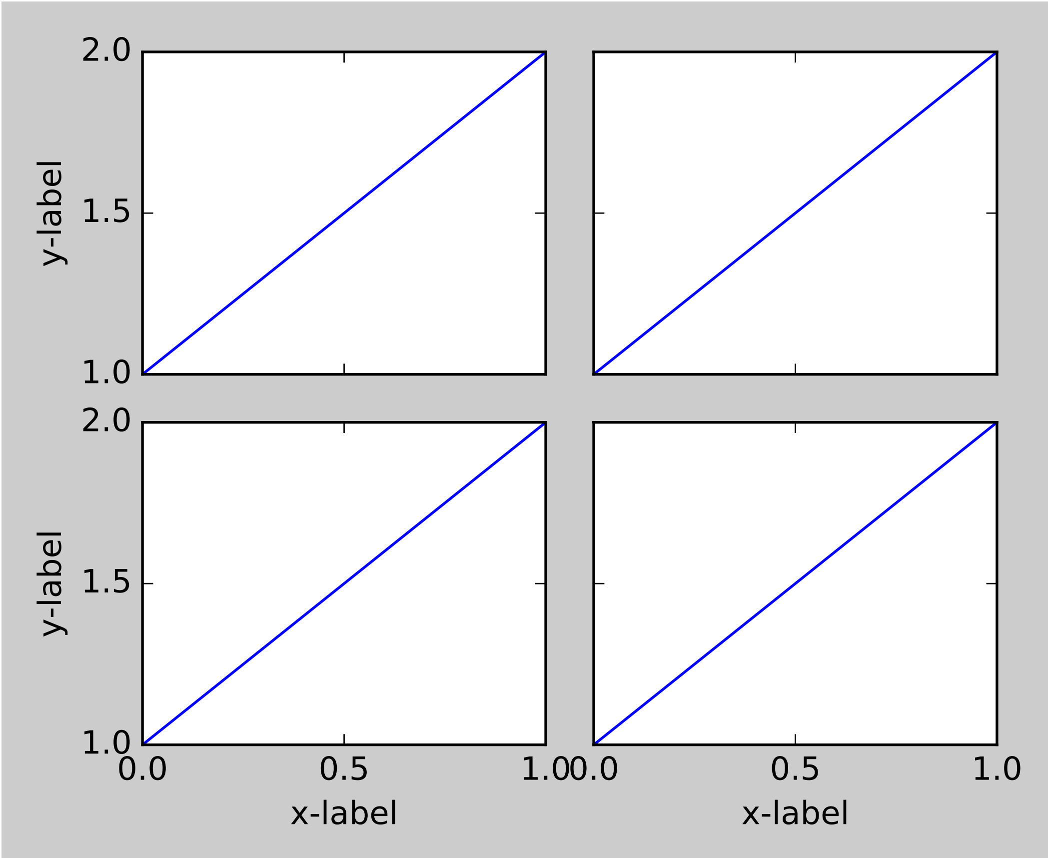
```
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.tight_layout(fig, rect=[None, 0 + (bottom-gs1.bottom),
                            0.5, 1 - (gs1.top-top)])
gs2.tight_layout(fig, rect=[0.5, 0 + (bottom-gs2.bottom),
                            None, 1 - (gs2.top-top)],
                 h_pad=0.5)
```



Use with AxesGrid1

While limited, the axes_grid1 toolkit is also supported.

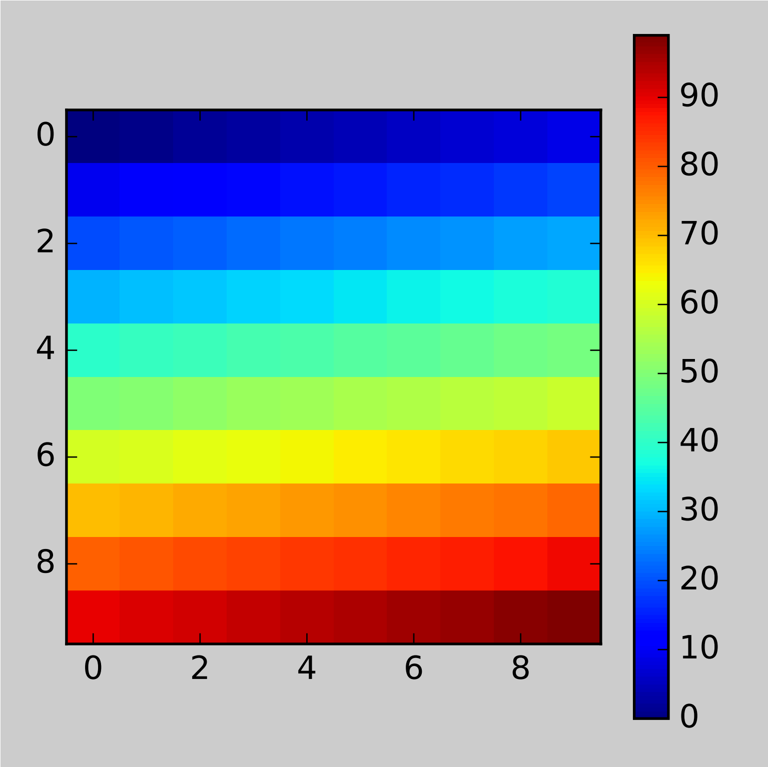
```
plt.close('all')
fig = plt.figure()

from mpl_toolkits.axes_grid1 import Grid
grid = Grid(fig, rect=111, nrows_ncols=(2,2),
            axes_pad=0.25, label_mode='L',
            )

for ax in grid:
    example_plot(ax)
    ax.title.set_visible(False)

plt.tight_layout()
```



Colorbar

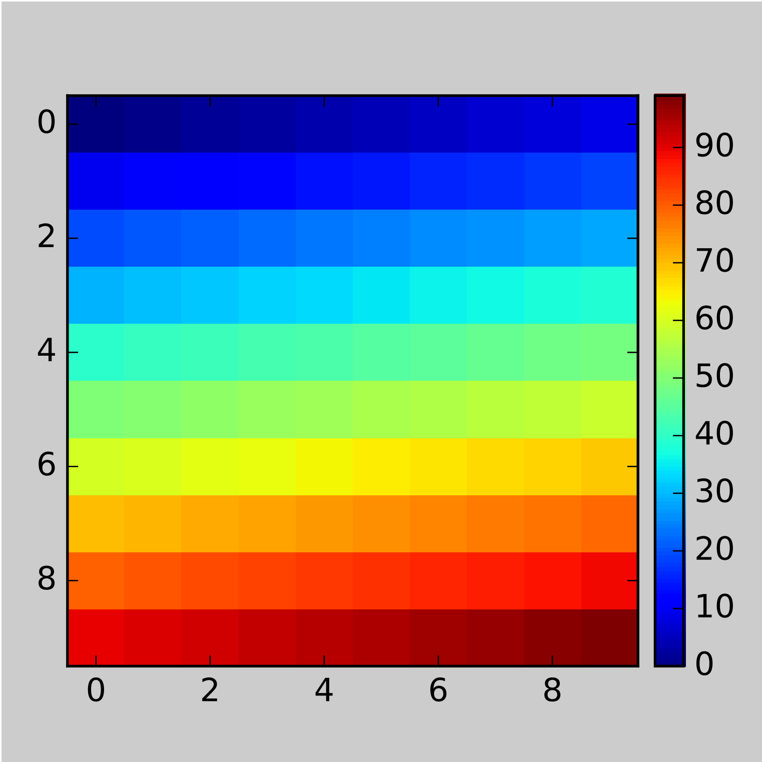
If you create a colorbar with the `colorbar()` command, the created colorbar is an instance of Axes, *not* Subplot, so tight_layout does not work. With Matplotlib v1.1, you may create a colorbar as a subplot using the gridspec.

```
plt.close('all')
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

plt.colorbar(im, use_gridspec=True)

plt.tight_layout()
```



Another option is to use AxesGrid1 toolkit to explicitly create an axes for colorbar.

```
plt.close('all')
fig = plt.figure(figsize=(4, 4))
im = plt.imshow(arr, interpolation="none")

from mpl_toolkits.axes_grid1 import make_axes_locatable
divider = make_axes_locatable(plt.gca())
cax = divider.append_axes("right", "5%", pad="3%")
plt.colorbar(im, cax=cax)

plt.tight_layout()
```



4.4 Event handling and picking

matplotlib works with a number of user interface toolkits (wxpython, tkinter, qt4, gtk, and macosx) and in order to support features like interactive panning and zooming of figures, it is helpful to the developers to have an API for interacting with the figure via key presses and mouse movements that is "GUI neutral" so we don't have to repeat a lot of code across the different user interfaces. Although the event handling API is GUI neutral, it is based on the GTK model, which was the first user interface matplotlib supported. The events that are triggered are also a bit richer vis-a-vis matplotlib than standard GUI events, including information like which *matplotlib.axes.Axes* the event occurred in. The events also understand the matplotlib coordinate system, and report event locations in both pixel and data coordinates.

4.4.1 Event connections

To receive events, you need to write a callback function and then connect your function to the event manager, which is part of the *FigureCanvasBase*. Here is a simple example that prints the location of the mouse click and which button was pressed:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(np.random.rand(10))

def onclick(event):
    print('button=%d, x=%d, y=%d, xdata=%f, ydata=%f' %
          (event.button, event.x, event.y, event.xdata, event.ydata))
```

```
cid = fig.canvas.mpl_connect('button_press_event', onclick)
```

The `FigureCanvas` method *mpl_connect()* returns a connection id which is simply an integer. When you want to disconnect the callback, just call:

```
fig.canvas.mpl_disconnect(cid)
```

Note: The canvas retains only weak references to the callbacks. Therefore if a callback is a method of a class instance, you need to retain a reference to that instance. Otherwise the instance will be garbage-collected and the callback will vanish.

Here are the events that you can connect to, the class instances that are sent back to you when the event occurs, and the event descriptions

Event name	Class and description
'button_press_event'	*MouseEvent* - mouse button is pressed
'button_release_event'	*MouseEvent* - mouse button is released
'draw_event'	*DrawEvent* - canvas draw
'key_press_event'	*KeyEvent* - key is pressed
'key_release_event'	*KeyEvent* - key is released
'motion_notify_event'	*MouseEvent* - mouse motion
'pick_event'	*PickEvent* - an object in the canvas is selected
'resize_event'	*ResizeEvent* - figure canvas is resized
'scroll_event'	*MouseEvent* - mouse scroll wheel is rolled
'figure_enter_event'	*LocationEvent* - mouse enters a new figure
'figure_leave_event'	*LocationEvent* - mouse leaves a figure
'axes_enter_event'	*LocationEvent* - mouse enters a new axes
'axes_leave_event'	*LocationEvent* - mouse leaves an axes

4.4.2 Event attributes

All matplotlib events inherit from the base class *matplotlib.backend_bases.Event*, which store the attributes:

name the event name

canvas the FigureCanvas instance generating the event

guiEvent the GUI event that triggered the matplotlib event

The most common events that are the bread and butter of event handling are key press/release events and mouse press/release and movement events. The *KeyEvent* and *MouseEvent* classes that handle these events are both derived from the LocationEvent, which has the following attributes

x x position - pixels from left of canvas

y y position - pixels from bottom of canvas

inaxes the *Axes* instance if mouse is over axes

xdata x coord of mouse in data coords

ydata y coord of mouse in data coords

Let's look a simple example of a canvas, where a simple line segment is created every time a mouse is pressed:

```
from matplotlib import pyplot as plt

class LineBuilder:
    def __init__(self, line):
        self.line = line
        self.xs = list(line.get_xdata())
        self.ys = list(line.get_ydata())
        self.cid = line.figure.canvas.mpl_connect('button_press_event', self)

    def __call__(self, event):
        print('click', event)
        if event.inaxes!=self.line.axes: return
        self.xs.append(event.xdata)
        self.ys.append(event.ydata)
        self.line.set_data(self.xs, self.ys)
        self.line.figure.canvas.draw()

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click to build line segments')
line, = ax.plot([0], [0])  # empty line
linebuilder = LineBuilder(line)

plt.show()
```

The *MouseEvent* that we just used is a *LocationEvent*, so we have access to the data and pixel coordinates in event.x and event.xdata. In addition to the `LocationEvent` attributes, it has

button button pressed None, 1, 2, 3, 'up', 'down' (up and down are used for scroll events)

key the key pressed: None, any character, 'shift', 'win', or 'control'

Draggable rectangle exercise

Write draggable rectangle class that is initialized with a *Rectangle* instance but will move its x,y location when dragged. Hint: you will need to store the original `xy` location of the rectangle which is stored as rect.xy and connect to the press, motion and release mouse events. When the mouse is pressed, check to see if the click occurs over your rectangle (see `matplotlib.patches.Rectangle.contains()`) and if it does, store the rectangle xy and the location of the mouse click in data coords. In the motion event callback, compute the deltax and deltay of the mouse movement, and add those deltas to the origin of the rectangle you stored. The redraw the figure. On the button release event, just reset all the button press data you stored as None.

Here is the solution:

```python
import numpy as np
import matplotlib.pyplot as plt

class DraggableRectangle:
    def __init__(self, rect):
        self.rect = rect
        self.press = None

    def connect(self):
        'connect to all the events we need'
        self.cidpress = self.rect.figure.canvas.mpl_connect(
            'button_press_event', self.on_press)
        self.cidrelease = self.rect.figure.canvas.mpl_connect(
            'button_release_event', self.on_release)
        self.cidmotion = self.rect.figure.canvas.mpl_connect(
            'motion_notify_event', self.on_motion)

    def on_press(self, event):
        'on button press we will see if the mouse is over us and store some data'
        if event.inaxes != self.rect.axes: return

        contains, attrd = self.rect.contains(event)
        if not contains: return
        print('event contains', self.rect.xy)
        x0, y0 = self.rect.xy
        self.press = x0, y0, event.xdata, event.ydata

    def on_motion(self, event):
        'on motion we will move the rect if the mouse is over us'
        if self.press is None: return
        if event.inaxes != self.rect.axes: return
        x0, y0, xpress, ypress = self.press
        dx = event.xdata - xpress
        dy = event.ydata - ypress
        #print('x0=%f, xpress=%f, event.xdata=%f, dx=%f, x0+dx=%f' %
        #      (x0, xpress, event.xdata, dx, x0+dx))
        self.rect.set_x(x0+dx)
        self.rect.set_y(y0+dy)

        self.rect.figure.canvas.draw()


    def on_release(self, event):
        'on release we reset the press data'
        self.press = None
        self.rect.figure.canvas.draw()

    def disconnect(self):
        'disconnect all the stored connection ids'
        self.rect.figure.canvas.mpl_disconnect(self.cidpress)
        self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
        self.rect.figure.canvas.mpl_disconnect(self.cidmotion)
```

```
fig = plt.figure()
ax = fig.add_subplot(111)
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()
```

Extra credit: use the animation blit techniques discussed in the animations recipe to make the animated drawing faster and smoother.

Extra credit solution:

```
# draggable rectangle with the animation blit techniques; see
# http://www.scipy.org/Cookbook/Matplotlib/Animations
import numpy as np
import matplotlib.pyplot as plt

class DraggableRectangle:
    lock = None  # only one can be animated at a time
    def __init__(self, rect):
        self.rect = rect
        self.press = None
        self.background = None

    def connect(self):
        'connect to all the events we need'
        self.cidpress = self.rect.figure.canvas.mpl_connect(
            'button_press_event', self.on_press)
        self.cidrelease = self.rect.figure.canvas.mpl_connect(
            'button_release_event', self.on_release)
        self.cidmotion = self.rect.figure.canvas.mpl_connect(
            'motion_notify_event', self.on_motion)

    def on_press(self, event):
        'on button press we will see if the mouse is over us and store some data'
        if event.inaxes != self.rect.axes: return
        if DraggableRectangle.lock is not None: return
        contains, attrd = self.rect.contains(event)
        if not contains: return
        print('event contains', self.rect.xy)
        x0, y0 = self.rect.xy
        self.press = x0, y0, event.xdata, event.ydata
        DraggableRectangle.lock = self

        # draw everything but the selected rectangle and store the pixel buffer
        canvas = self.rect.figure.canvas
        axes = self.rect.axes
        self.rect.set_animated(True)
        canvas.draw()
```

```
        self.background = canvas.copy_from_bbox(self.rect.axes.bbox)

        # now redraw just the rectangle
        axes.draw_artist(self.rect)

        # and blit just the redrawn area
        canvas.blit(axes.bbox)

    def on_motion(self, event):
        'on motion we will move the rect if the mouse is over us'
        if DraggableRectangle.lock is not self:
            return
        if event.inaxes != self.rect.axes: return
        x0, y0, xpress, ypress = self.press
        dx = event.xdata - xpress
        dy = event.ydata - ypress
        self.rect.set_x(x0+dx)
        self.rect.set_y(y0+dy)

        canvas = self.rect.figure.canvas
        axes = self.rect.axes
        # restore the background region
        canvas.restore_region(self.background)

        # redraw just the current rectangle
        axes.draw_artist(self.rect)

        # blit just the redrawn area
        canvas.blit(axes.bbox)

    def on_release(self, event):
        'on release we reset the press data'
        if DraggableRectangle.lock is not self:
            return

        self.press = None
        DraggableRectangle.lock = None

        # turn off the rect animation property and reset the background
        self.rect.set_animated(False)
        self.background = None

        # redraw the full figure
        self.rect.figure.canvas.draw()

    def disconnect(self):
        'disconnect all the stored connection ids'
        self.rect.figure.canvas.mpl_disconnect(self.cidpress)
        self.rect.figure.canvas.mpl_disconnect(self.cidrelease)
        self.rect.figure.canvas.mpl_disconnect(self.cidmotion)

fig = plt.figure()
ax = fig.add_subplot(111)
```

```
rects = ax.bar(range(10), 20*np.random.rand(10))
drs = []
for rect in rects:
    dr = DraggableRectangle(rect)
    dr.connect()
    drs.append(dr)

plt.show()
```

4.4.3 Mouse enter and leave

If you want to be notified when the mouse enters or leaves a figure or axes, you can connect to the figure/axes enter/leave events. Here is a simple example that changes the colors of the axes and figure background that the mouse is over:

```
"""
Illustrate the figure and axes enter and leave events by changing the
frame colors on enter and leave
"""
import matplotlib.pyplot as plt

def enter_axes(event):
    print('enter_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('yellow')
    event.canvas.draw()

def leave_axes(event):
    print('leave_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('white')
    event.canvas.draw()

def enter_figure(event):
    print('enter_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('red')
    event.canvas.draw()

def leave_figure(event):
    print('leave_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('grey')
    event.canvas.draw()

fig1 = plt.figure()
fig1.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig1.add_subplot(211)
ax2 = fig1.add_subplot(212)

fig1.canvas.mpl_connect('figure_enter_event', enter_figure)
fig1.canvas.mpl_connect('figure_leave_event', leave_figure)
fig1.canvas.mpl_connect('axes_enter_event', enter_axes)
fig1.canvas.mpl_connect('axes_leave_event', leave_axes)
```

```
fig2 = plt.figure()
fig2.suptitle('mouse hover over figure or axes to trigger events')
ax1 = fig2.add_subplot(211)
ax2 = fig2.add_subplot(212)

fig2.canvas.mpl_connect('figure_enter_event', enter_figure)
fig2.canvas.mpl_connect('figure_leave_event', leave_figure)
fig2.canvas.mpl_connect('axes_enter_event', enter_axes)
fig2.canvas.mpl_connect('axes_leave_event', leave_axes)

plt.show()
```

4.4.4 Object picking

You can enable picking by setting the `picker` property of an *Artist* (e.g., a matplotlib *Line2D*, *Text*, *Patch*, *Polygon*, `AxesImage`, etc...)

There are a variety of meanings of the `picker` property:

None picking is disabled for this artist (default)

boolean if True then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist

float if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if its data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event.

function if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event. The signature is `hit, props = picker(artist, mouseevent)` to determine the hit test. If the mouse event is over the artist, return `hit=True` and props is a dictionary of properties you want added to the *PickEvent* attributes

After you have enabled an artist for picking by setting the `picker` property, you need to connect to the figure canvas pick_event to get pick callbacks on mouse press events. e.g.:

```
def pick_handler(event):
    mouseevent = event.mouseevent
    artist = event.artist
    # now do something with this...
```

The *PickEvent* which is passed to your callback is always fired with two attributes:

mouseevent the mouse event that generate the pick event. The mouse event in turn has attributes like `x` and `y` (the coords in display space, e.g., pixels from left, bottom) and xdata, ydata (the coords in data space). Additionally, you can get information about which buttons were pressed, which keys were pressed, which *Axes* the mouse is over, etc. See *matplotlib.backend_bases.MouseEvent* for details.

artist the *Artist* that generated the pick event.

Additionally, certain artists like *Line2D* and *PatchCollection* may attach additional meta data like the indices into the data that meet the picker criteria (e.g., all the points in the line that are within the specified epsilon tolerance)

Simple picking example

In the example below, we set the line picker property to a scalar, so it represents a tolerance in points (72 points per inch). The onpick callback function will be called when the pick event it within the tolerance distance from the line, and has the indices of the data vertices that are within the pick distance tolerance. Our onpick callback function simply prints the data that are under the pick location. Different matplotlib Artists can attach different data to the PickEvent. For example, `Line2D` attaches the ind property, which are the indices into the line data under the pick point. See `pick()` for details on the `PickEvent` properties of the line. Here is the code:

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on points')

line, = ax.plot(np.random.rand(100), 'o', picker=5)  # 5 points tolerance

def onpick(event):
    thisline = event.artist
    xdata = thisline.get_xdata()
    ydata = thisline.get_ydata()
    ind = event.ind
    points = tuple(zip(xdata[ind], ydata[ind]))
    print('onpick points:', points)

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

Picking exercise

Create a data set of 100 arrays of 1000 Gaussian random numbers and compute the sample mean and standard deviation of each of them (hint: numpy arrays have a mean and std method) and make a xy marker plot of the 100 means vs the 100 standard deviations. Connect the line created by the plot command to the pick event, and plot the original time series of the data that generated the clicked on points. If more than one point is within the tolerance of the clicked on point, you can use multiple subplots to plot the multiple time series.

Exercise solution:

```
"""
compute the mean and stddev of 100 data sets and plot mean vs stddev.
When you click on one of the mu, sigma points, plot the raw data from
```

```
the dataset that generated the mean and stddev
"""
import numpy as np
import matplotlib.pyplot as plt

X = np.random.rand(100, 1000)
xs = np.mean(X, axis=1)
ys = np.std(X, axis=1)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('click on point to plot time series')
line, = ax.plot(xs, ys, 'o', picker=5)  # 5 points tolerance


def onpick(event):

    if event.artist!=line: return True

    N = len(event.ind)
    if not N: return True


    figi = plt.figure()
    for subplotnum, dataind in enumerate(event.ind):
        ax = figi.add_subplot(N,1,subplotnum+1)
        ax.plot(X[dataind])
        ax.text(0.05, 0.9, 'mu=%1.3f\nsigma=%1.3f'%(xs[dataind], ys[dataind]),
                transform=ax.transAxes, va='top')
        ax.set_ylim(-0.5, 1.5)
    figi.show()
    return True

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

4.5 Transformations Tutorial

Like any graphics packages, matplotlib is built on top of a transformation framework to easily move between coordinate systems, the userland `data` coordinate system, the `axes` coordinate system, the `figure` coordinate system, and the `display` coordinate system. In 95% of your plotting, you won't need to think about this, as it happens under the hood, but as you push the limits of custom figure generation, it helps to have an understanding of these objects so you can reuse the existing transformations matplotlib makes available to you, or create your own (see *`matplotlib.transforms`*). The table below summarizes the existing coordinate systems, the transformation object you should use to work in that coordinate system, and the description of that system. In the `Transformation Object` column, `ax` is a *`Axes`* instance, and `fig` is a *`Figure`* instance.

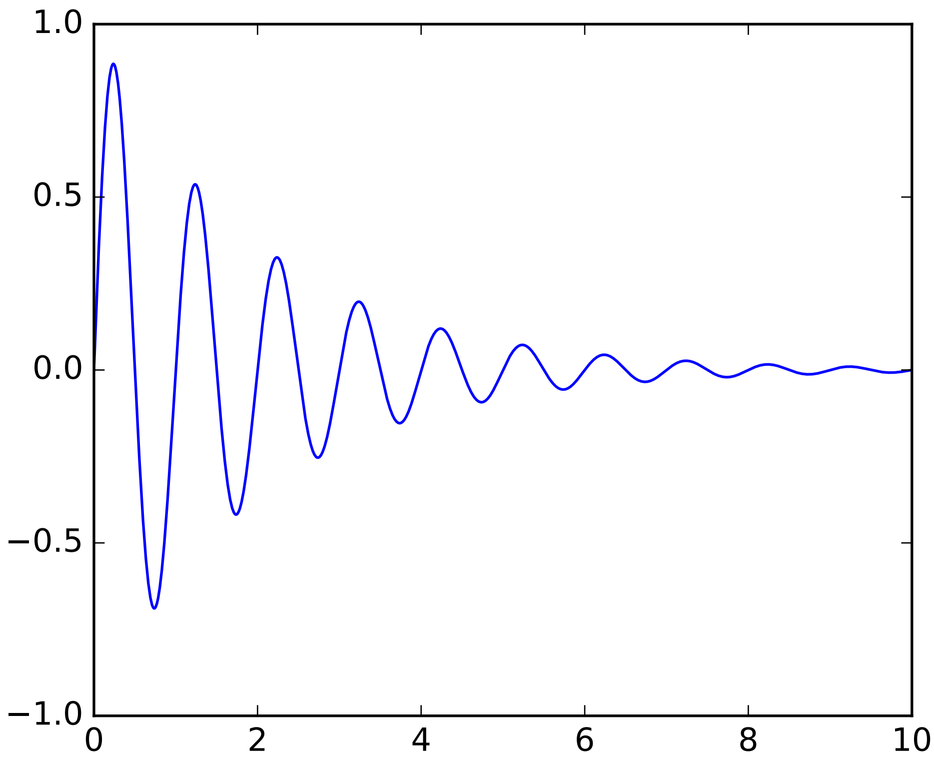
Coordinate	Transformation Object	Description
`data`	`ax.transData`	The userland data coordinate system, controlled by the xlim and ylim
`axes`	`ax.transAxes`	The coordinate system of the *Axes*; (0,0) is bottom left of the axes, and (1,1) is top right of the axes.
`figure`	`fig.transFigure`	The coordinate system of the *Figure*; (0,0) is bottom left of the figure, and (1,1) is top right of the figure.
`display`	`None`	This is the pixel coordinate system of the display; (0,0) is the bottom left of the display, and (width, height) is the top right of the display in pixels. Alternatively, the identity transform (*matplotlib.transforms.IdentityTransform()*) may be used instead of None.

All of the transformation objects in the table above take inputs in their coordinate system, and transform the input to the `display` coordinate system. That is why the `display` coordinate system has `None` for the `Transformation Object` column – it already is in display coordinates. The transformations also know how to invert themselves, to go from `display` back to the native coordinate system. This is particularly useful when processing events from the user interface, which typically occur in display space, and you want to know where the mouse click or key-press occurred in your data coordinate system.

4.5.1 Data coordinates

Let's start with the most commonly used coordinate, the `data` coordinate system. Whenever you add data to the axes, matplotlib updates the datalimits, most commonly updated with the *set_xlim()* and *set_ylim()* methods. For example, in the figure below, the data limits stretch from 0 to 10 on the x-axis, and -1 to 1 on the y-axis.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 10, 0.005)
y = np.exp(-x/2.) * np.sin(2*np.pi*x)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
ax.set_xlim(0, 10)
ax.set_ylim(-1, 1)

plt.show()
```



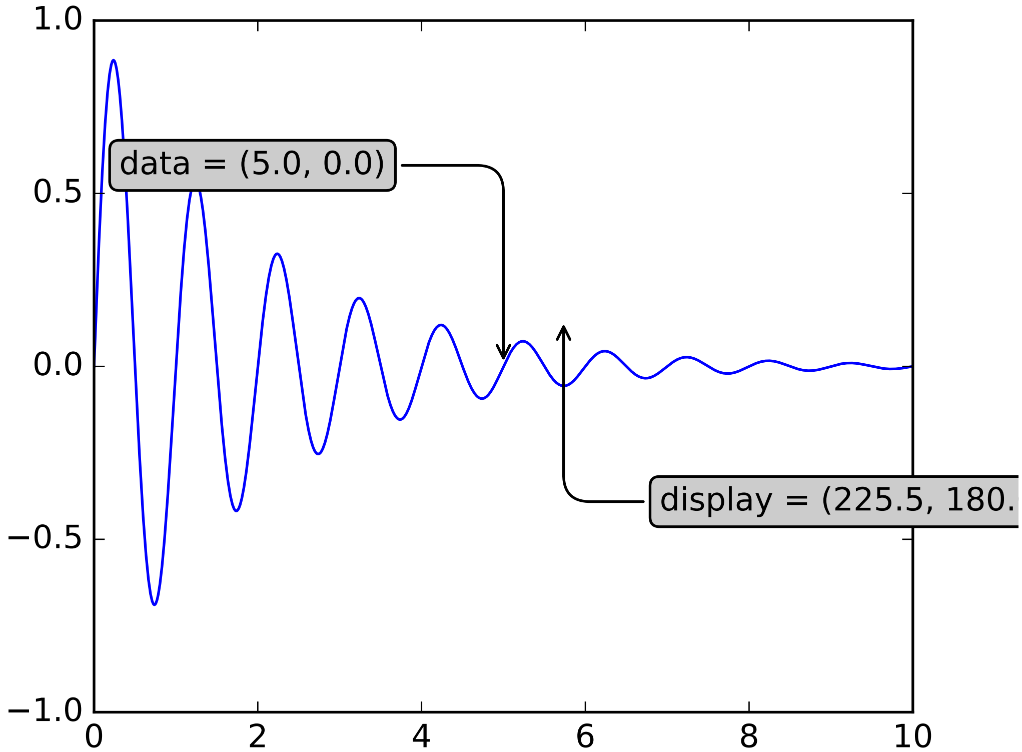
You can use the `ax.transData` instance to transform from your `data` to your `display` coordinate system, either a single point or a sequence of points as shown below:

```
In [14]: type(ax.transData)
Out[14]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [15]: ax.transData.transform((5, 0))
Out[15]: array([ 335.175,  247.   ])

In [16]: ax.transData.transform([(5, 0), (1,2)])
Out[16]:
array([[ 335.175,  247.   ],
       [ 132.435,  642.2  ]])
```

You can use the *`inverted()`* method to create a transform which will take you from display to data coordinates:

```
In [41]: inv = ax.transData.inverted()

In [42]: type(inv)
Out[42]: <class 'matplotlib.transforms.CompositeGenericTransform'>

In [43]: inv.transform((335.175,  247.))
Out[43]: array([ 5.,  0.])
```

If your are typing along with this tutorial, the exact values of the display coordinates may differ if you have a different window size or dpi setting. Likewise, in the figure below, the display labeled points are probably not the same as in the ipython session because the documentation figure size defaults are different.



Note: If you run the source code in the example above in a GUI backend, you may also find that the two arrows for the `data` and `display` annotations do not point to exactly the same point. This is because the display point was computed before the figure was displayed, and the GUI backend may slightly resize the figure when it is created. The effect is more pronounced if you resize the figure yourself. This is one good reason why you rarely want to work in display space, but you can connect to the `'on_draw'` *Event* to update figure coordinates on figure draws; see *Event handling and picking*.

When you change the x or y limits of your axes, the data limits are updated so the transformation yields a new display point. Note that when we just change the ylim, only the y-display coordinate is altered, and when we change the xlim too, both are altered. More on this later when we talk about the *Bbox*.

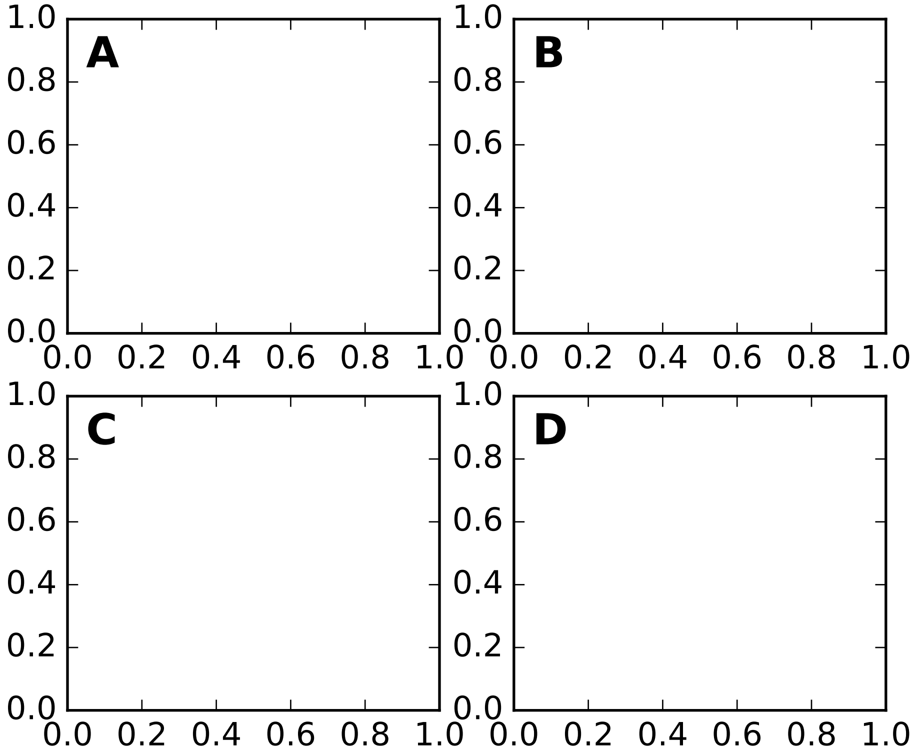
```
In [54]: ax.transData.transform((5, 0))
Out[54]: array([ 335.175,  247.   ])

In [55]: ax.set_ylim(-1,2)
Out[55]: (-1, 2)

In [56]: ax.transData.transform((5, 0))
Out[56]: array([ 335.175     ,  181.13333333])
```

```
In [57]: ax.set_xlim(10,20)
Out[57]: (10, 20)

In [58]: ax.transData.transform((5, 0))
Out[58]: array([-171.675     ,  181.13333333])
```

4.5.2 Axes coordinates

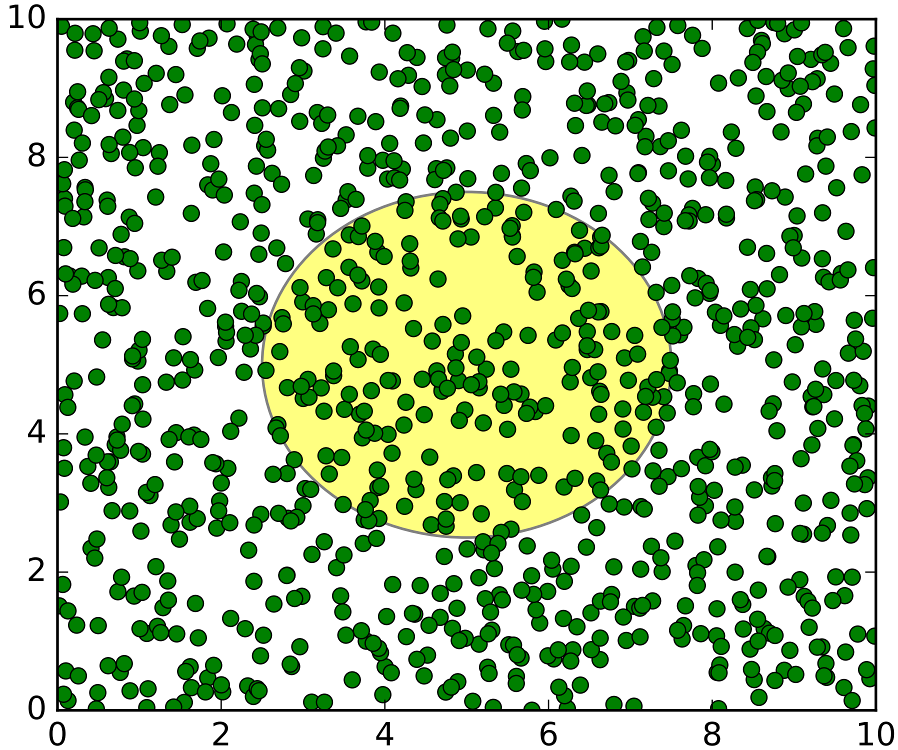
After the `data` coordinate system, `axes` is probably the second most useful coordinate system. Here the point (0,0) is the bottom left of your axes or subplot, (0.5, 0.5) is the center, and (1.0, 1.0) is the top right. You can also refer to points outside the range, so (-0.1, 1.1) is to the left and above your axes. This coordinate system is extremely useful when placing text in your axes, because you often want a text bubble in a fixed, location, e.g., the upper left of the axes pane, and have that location remain fixed when you pan or zoom. Here is a simple example that creates four panels and labels them 'A', 'B', 'C', 'D' as you often see in journals.

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
for i, label in enumerate(('A', 'B', 'C', 'D')):
    ax = fig.add_subplot(2,2,i+1)
    ax.text(0.05, 0.95, label, transform=ax.transAxes,
      fontsize=16, fontweight='bold', va='top')

plt.show()
```



You can also make lines or patches in the axes coordinate system, but this is less useful in my experience than using `ax.transAxes` for placing text. Nonetheless, here is a silly example which plots some random dots in `data` space, and overlays a semi-transparent *Circle* centered in the middle of the axes with a radius one quarter of the axes – if your axes does not preserve aspect ratio (see *set_aspect()*), this will look like an ellipse. Use the pan/zoom tool to move around, or manually change the data xlim and ylim, and you will see the data move, but the circle will remain fixed because it is not in `data` coordinates and will always remain at the center of the axes.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
fig = plt.figure()
ax = fig.add_subplot(111)
x, y = 10*np.random.rand(2, 1000)
ax.plot(x, y, 'go')  # plot some data in data coordinates

circ = patches.Circle((0.5, 0.5), 0.25, transform=ax.transAxes,
                      facecolor='yellow', alpha=0.5)
ax.add_patch(circ)

plt.show()
```

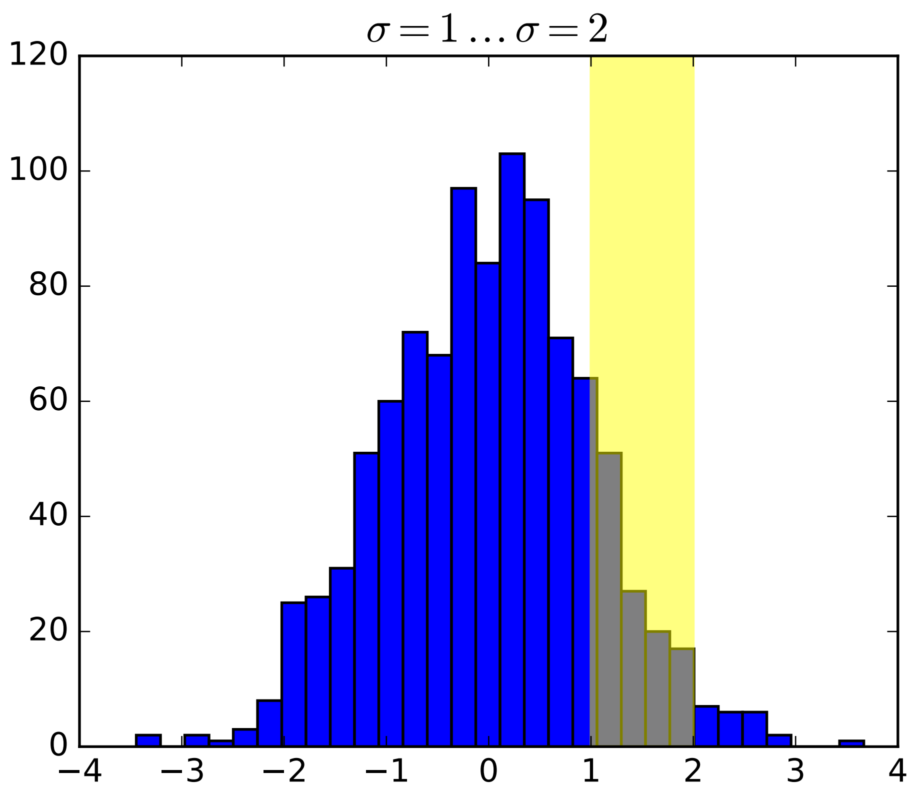


4.5.3 Blended transformations

Drawing in `blended` coordinate spaces which mix `axes` with `data` coordinates is extremely useful, for example to create a horizontal span which highlights some region of the y-data but spans across the x-axis regardless of the data limits, pan or zoom level, etc. In fact these blended lines and spans are so useful, we have built in functions to make them easy to plot (see `axhline()`, `axvline()`, `axhspan()`, `axvspan()`) but for didactic purposes we will implement the horizontal span here using a blended transformation. This trick only works for separable transformations, like you see in normal Cartesian coordinate systems, but not on inseparable transformations like the `PolarTransform`.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.transforms as transforms

fig = plt.figure()
ax = fig.add_subplot(111)

x = np.random.randn(1000)

ax.hist(x, 30)
ax.set_title(r'$\sigma=1 \/ \dots \/ \sigma=2$', fontsize=16)
```

```
# the x coords of this transformation are data, and the
# y coord are axes
trans = transforms.blended_transform_factory(
    ax.transData, ax.transAxes)

# highlight the 1..2 stddev region with a span.
# We want x to be in data coordinates and y to
# span from 0..1 in axes coords
rect = patches.Rectangle((1,0), width=1, height=1,
                         transform=trans, color='yellow',
                         alpha=0.5)

ax.add_patch(rect)

plt.show()
```



Note: The blended transformations where x is in data coords and y in axes coordinates is so useful that we have helper methods to return the versions mpl uses internally for drawing ticks, ticklabels, etc. The methods are *matplotlib.axes.Axes.get_xaxis_transform()* and *matplotlib.axes.Axes.get_yaxis_transform()*. So in the example above, the call to *blended_transform_factory()* can be replaced by `get_xaxis_transform`:

```
trans = ax.get_xaxis_transform()
```

4.5.4 Using offset transforms to create a shadow effect

One use of transformations is to create a new transformation that is offset from another transformation, e.g., to place one object shifted a bit relative to another object. Typically you want the shift to be in some physical dimension, like points or inches rather than in data coordinates, so that the shift effect is constant at different zoom levels and dpi settings.

One use for an offset is to create a shadow effect, where you draw one object identical to the first just to the right of it, and just below it, adjusting the zorder to make sure the shadow is drawn first and then the object it is shadowing above it. The transforms module has a helper transformation *ScaledTranslation*. It is instantiated with:

```
trans = ScaledTranslation(xt, yt, scale_trans)
```

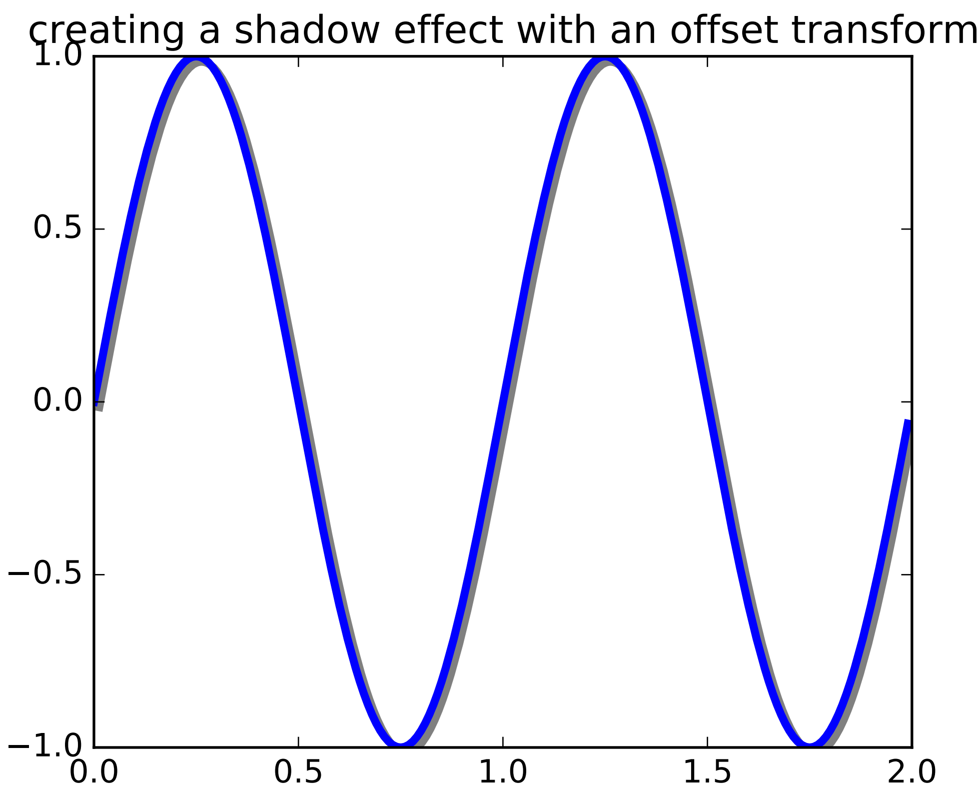
where `xt` and `yt` are the translation offsets, and `scale_trans` is a transformation which scales `xt` and `yt` at transformation time before applying the offsets. A typical use case is to use the figure `fig.dpi_scale_trans` transformation for the `scale_trans` argument, to first scale `xt` and `yt` specified in points to `display` space before doing the final offset. The dpi and inches offset is a common-enough use case that we have a special helper function to create it in `matplotlib.transforms.offset_copy()`, which returns a new transform with an added offset. But in the example below, we'll create the offset transform ourselves. Note the use of the plus operator in:

```
offset = transforms.ScaledTranslation(dx, dy,
  fig.dpi_scale_trans)
shadow_transform = ax.transData + offset
```

showing that can chain transformations using the addition operator. This code says: first apply the data transformation `ax.transData` and then translate the data by `dx` and `dy` points. In typography, a'point <http://en.wikipedia.org/wiki/Point_%28typography%29>'_ is 1/72 inches, and by specifying your offsets in points, your figure will look the same regardless of the dpi resolution it is saved in.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.transforms as transforms

fig = plt.figure()
ax = fig.add_subplot(111)

# make a simple sine wave
x = np.arange(0., 2., 0.01)
y = np.sin(2*np.pi*x)
line, = ax.plot(x, y, lw=3, color='blue')

# shift the object over 2 points, and down 2 points
dx, dy = 2/72., -2/72.
```

```
offset = transforms.ScaledTranslation(dx, dy,
  fig.dpi_scale_trans)
shadow_transform = ax.transData + offset

# now plot the same data with our offset transform;
# use the zorder to make sure we are below the line
ax.plot(x, y, lw=3, color='gray',
  transform=shadow_transform,
  zorder=0.5*line.get_zorder())

ax.set_title('creating a shadow effect with an offset transform')
plt.show()
```



4.5.5 The transformation pipeline

The `ax.transData` transform we have been working with in this tutorial is a composite of three different transformations that comprise the transformation pipeline from `data -> display` coordinates. Michael Droettboom implemented the transformations framework, taking care to provide a clean API that segregated the nonlinear projections and scales that happen in polar and logarithmic plots, from the linear affine transformations that happen when you pan and zoom. There is an efficiency here, because you can pan and zoom in your axes which affects the affine transformation, but you may not need to compute the potentially expensive nonlinear scales or projections on simple navigation events. It is also possible to multiply affine transformation matrices together, and then apply them to coordinates in one step. This is not true of all possible transformations.

Here is how the `ax.transData` instance is defined in the basic separable axis *Axes* class:

```
self.transData = self.transScale + (self.transLimits + self.transAxes)
```

We've been introduced to the `transAxes` instance above in *Axes coordinates*, which maps the (0,0), (1,1) corners of the axes or subplot bounding box to `display` space, so let's look at these other two pieces.

`self.transLimits` is the transformation that takes you from `data` to `axes` coordinates; i.e., it maps your view xlim and ylim to the unit space of the axes (and `transAxes` then takes that unit space to display space). We can see this in action here

```
In [80]: ax = subplot(111)

In [81]: ax.set_xlim(0, 10)
Out[81]: (0, 10)

In [82]: ax.set_ylim(-1,1)
Out[82]: (-1, 1)

In [84]: ax.transLimits.transform((0,-1))
Out[84]: array([ 0.,  0.])

In [85]: ax.transLimits.transform((10,-1))
Out[85]: array([ 1.,  0.])

In [86]: ax.transLimits.transform((10,1))
Out[86]: array([ 1.,  1.])

In [87]: ax.transLimits.transform((5,0))
Out[87]: array([ 0.5,  0.5])
```

and we can use this same inverted transformation to go from the unit `axes` coordinates back to `data` coordinates.

```
In [90]: inv.transform((0.25, 0.25))
Out[90]: array([ 2.5, -0.5])
```

The final piece is the `self.transScale` attribute, which is responsible for the optional non-linear scaling of the data, e.g., for logarithmic axes. When an Axes is initially setup, this is just set to the identity transform, since the basic matplotlib axes has linear scale, but when you call a logarithmic scaling function like *semilogx()* or explicitly set the scale to logarithmic with *set_xscale()*, then the `ax.transScale` attribute is set to handle the nonlinear projection. The scales transforms are properties of the respective `xaxis` and `yaxis` *Axis* instances. For example, when you call `ax.set_xscale('log')`, the xaxis updates its scale to a *matplotlib.scale.LogScale* instance.

For non-separable axes the PolarAxes, there is one more piece to consider, the projection transformation. The `transData` *matplotlib.projections.polar.PolarAxes* is similar to that for the typical separable matplotlib Axes, with one additional piece `transProjection`:

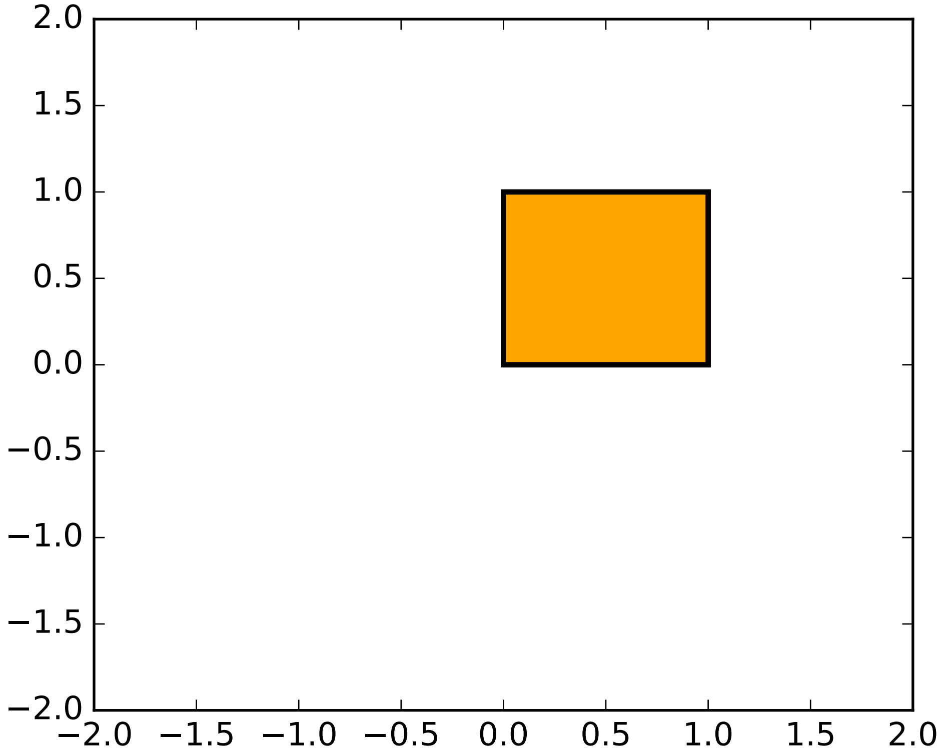
```
self.transData = self.transScale + self.transProjection + \
    (self.transProjectionAffine + self.transAxes)
```

`transProjection` handles the projection from the space, e.g., latitude and longitude for map data, or radius and theta for polar data, to a separable Cartesian coordinate system. There are several projection examples in the `matplotlib.projections` package, and the best way to learn more is to open the source for those packages and see how to make your own, since matplotlib supports extensible axes and projections. Michael Droettboom has provided a nice tutorial example of creating a hammer projection axes; see *api example code: custom_projection_example.py*.

4.6 Path Tutorial

The object underlying all of the `matplotlib.patch` objects is the *Path*, which supports the standard set of moveto, lineto, curveto commands to draw simple and compound outlines consisting of line segments and splines. The `Path` is instantiated with a (N,2) array of (x,y) vertices, and a N-length array of path codes. For example to draw the unit rectangle from (0,0) to (1,1), we could use this code

```
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches

verts = [
    (0., 0.), # left, bottom
    (0., 1.), # left, top
    (1., 1.), # right, top
    (1., 0.), # right, bottom
    (0., 0.), # ignored
    ]

codes = [Path.MOVETO,
         Path.LINETO,
         Path.LINETO,
         Path.LINETO,
         Path.CLOSEPOLY,
         ]

path = Path(verts, codes)

fig = plt.figure()
ax = fig.add_subplot(111)
patch = patches.PathPatch(path, facecolor='orange', lw=2)
ax.add_patch(patch)
ax.set_xlim(-2,2)
ax.set_ylim(-2,2)
plt.show()
```



The following path codes are recognized

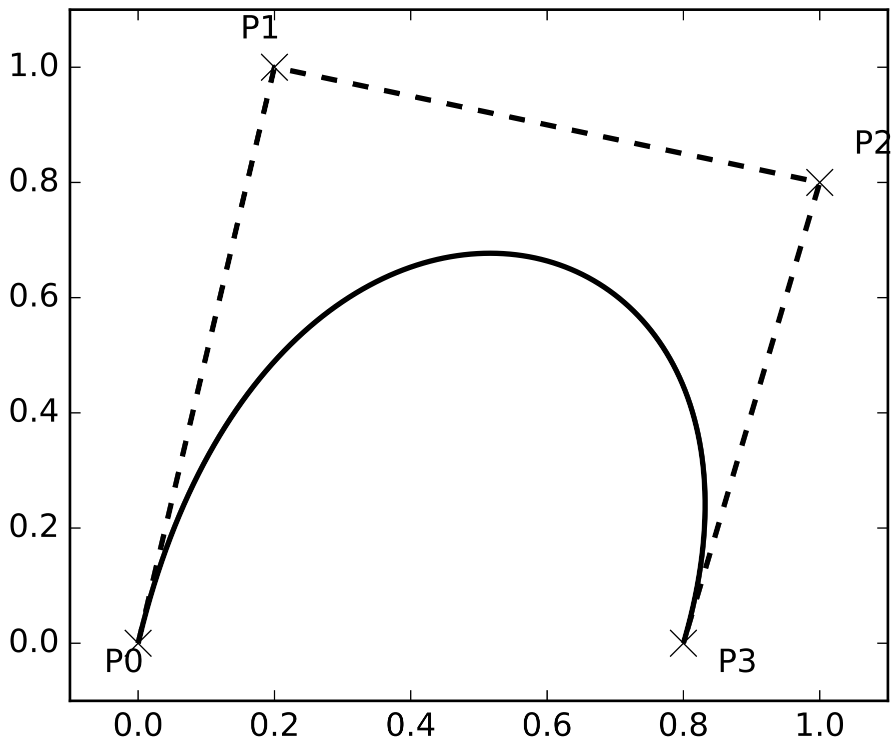
Code	Vertices	Description
STOP	1 (ignored)	A marker for the end of the entire path (currently not required and ignored)
MOVETO	1	Pick up the pen and move to the given vertex.
LINETO	1	Draw a line from the current position to the given vertex.
CURVE3	2 (1 control point, 1 endpoint)	Draw a quadratic Bézier curve from the current position, with the given control point, to the given end point.
CURVE4	3 (2 control points, 1 endpoint)	Draw a cubic Bézier curve from the current position, with the given control points, to the given end point.
CLOSEPOLY	1 (point itself is ignored)	Draw a line segment to the start point of the current polyline.

4.6.1 Bézier example

Some of the path components require multiple vertices to specify them: for example CURVE 3 is a bézier curve with one control point and one end point, and CURVE4 has three vertices for the two control points and the end point. The example below shows a CURVE4 Bézier spline – the bézier curve will be contained in the convex hull of the start point, the two control points, and the end point

```
import matplotlib.pyplot as plt
from matplotlib.path import Path
```

```
import matplotlib.patches as patches

verts = [
    (0., 0.),  # P0
    (0.2, 1.), # P1
    (1., 0.8), # P2
    (0.8, 0.), # P3
    ]

codes = [Path.MOVETO,
         Path.CURVE4,
         Path.CURVE4,
         Path.CURVE4,
         ]

path = Path(verts, codes)

fig = plt.figure()
ax = fig.add_subplot(111)
patch = patches.PathPatch(path, facecolor='none', lw=2)
ax.add_patch(patch)

xs, ys = zip(*verts)
ax.plot(xs, ys, 'x--', lw=2, color='black', ms=10)

ax.text(-0.05, -0.05, 'P0')
ax.text(0.15, 1.05, 'P1')
ax.text(1.05, 0.85, 'P2')
ax.text(0.85, -0.05, 'P3')

ax.set_xlim(-0.1, 1.1)
ax.set_ylim(-0.1, 1.1)
plt.show()
```



4.6.2 Compound paths

All of the simple patch primitives in matplotlib, Rectangle, Circle, Polygon, etc, are implemented with simple path. Plotting functions like *hist()* and *bar()*, which create a number of primitives, e.g., a bunch of Rectangles, can usually be implemented more efficiently using a compound path. The reason `bar` creates a list of rectangles and not a compound path is largely historical: the *Path* code is comparatively new and `bar` predates it. While we could change it now, it would break old code, so here we will cover how to create compound paths, replacing the functionality in bar, in case you need to do so in your own code for efficiency reasons, e.g., you are creating an animated bar plot.

We will make the histogram chart by creating a series of rectangles for each histogram bar: the rectangle width is the bin width and the rectangle height is the number of datapoints in that bin. First we'll create some random normally distributed data and compute the histogram. Because numpy returns the bin edges and not centers, the length of `bins` is 1 greater than the length of `n` in the example below:

```
# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)
```

We'll now extract the corners of the rectangles. Each of the `left`, `bottom`, etc, arrays below is `len(n)`, where `n` is the array of counts for each histogram bar:

```
# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n
```

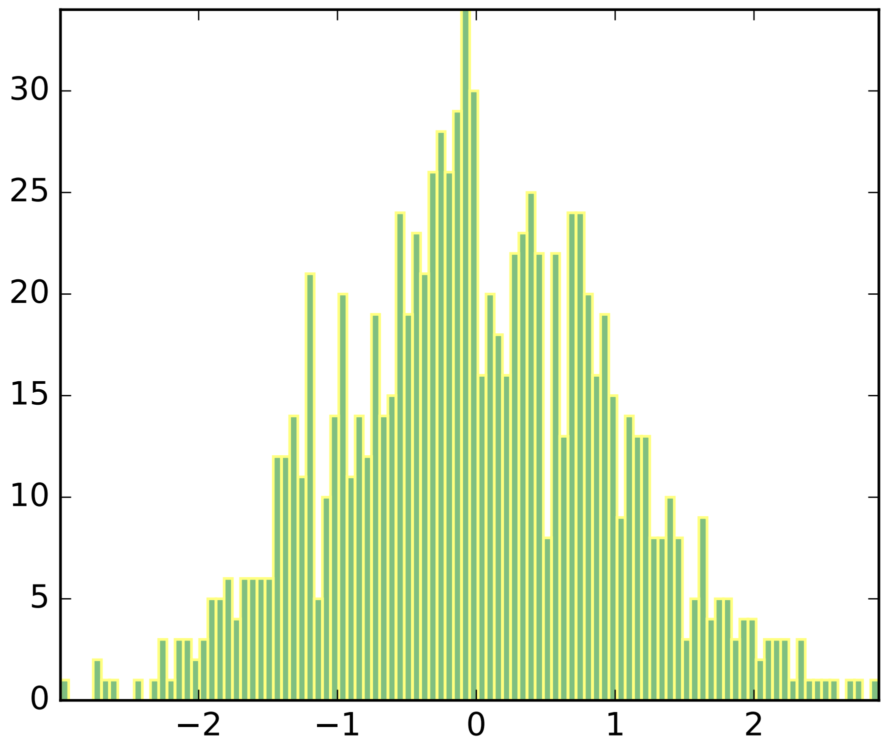
Now we have to construct our compound path, which will consist of a series of `MOVETO`, `LINETO` and `CLOSEPOLY` for each rectangle. For each rectangle, we need 5 vertices: 1 for the `MOVETO`, 3 for the `LINETO`, and 1 for the `CLOSEPOLY`. As indicated in the table above, the vertex for the closepoly is ignored but we still need it to keep the codes aligned with the vertices:

```
nverts = nrects*(1+3+1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5,0] = left
verts[0::5,1] = bottom
verts[1::5,0] = left
verts[1::5,1] = top
verts[2::5,0] = right
verts[2::5,1] = top
verts[3::5,0] = right
verts[3::5,1] = bottom
```

All that remains is to create the path, attach it to a `PathPatch`, and add it to our axes:

```
barpath = path.Path(verts, codes)
patch = patches.PathPatch(barpath, facecolor='green',
  edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)
```

Here is the result



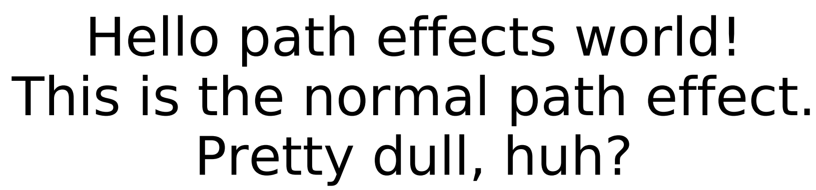
4.7 Path effects guide

Matplotlib's *patheffects* module provides functionality to apply a multiple draw stage to any Artist which can be rendered via a *Path*.

Artists which can have a path effect applied to them include *Patch*, *Line2D*, *Collection* and even *Text*. Each artist's path effects can be controlled via the `set_path_effects` method (*set_path_effects*), which takes an iterable of *AbstractPathEffect* instances.

The simplest path effect is the *Normal* effect, which simply draws the artist without any effect:

```
import matplotlib.pyplot as plt
import matplotlib.patheffects as path_effects

fig = plt.figure(figsize=(5, 1.5))
text = fig.text(0.5, 0.5, 'Hello path effects world!\nThis is the normal '
                          'path effect.\nPretty dull, huh?',
                ha='center', va='center', size=20)
text.set_path_effects([path_effects.Normal()])
plt.show()
```

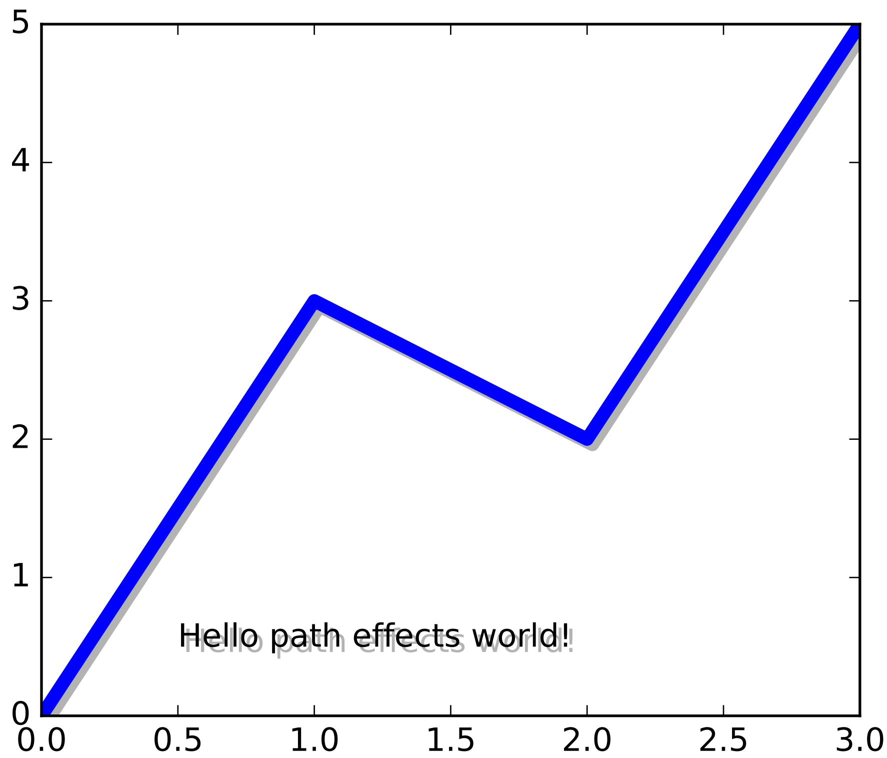


Whilst the plot doesn't look any different to what you would expect without any path effects, the drawing of the text now been changed to use the path effects framework, opening up the possibilities for more interesting examples.

4.7.1 Adding a shadow

A far more interesting path effect than *Normal* is the drop-shadow, which we can apply to any of our path based artists. The classes *SimplePatchShadow* and *SimpleLineShadow* do precisely this by drawing either a filled patch or a line patch below the original artist:

```
import matplotlib.pyplot as plt
import matplotlib.patheffects as path_effects

text = plt.text(0.5, 0.5, 'Hello path effects world!',
                path_effects=[path_effects.withSimplePatchShadow()])

plt.plot([0, 3, 2, 5], linewidth=5, color='blue',
         path_effects=[path_effects.SimpleLineShadow(),
                       path_effects.Normal()])
plt.show()
```

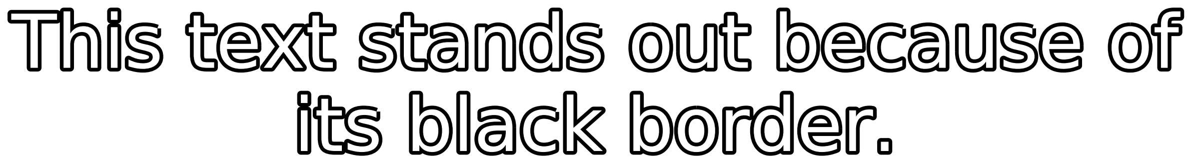


Notice the two approaches to setting the path effects in this example. The first uses the `with*` classes to include the desired functionality automatically followed with the "normal" effect, whereas the latter explicitly defines the two path effects to draw.

4.7.2 Making an artist stand out

One nice way of making artists visually stand out is to draw an outline in a bold color below the actual artist. The *Stroke* path effect makes this a relatively simple task:

```
import matplotlib.pyplot as plt
import matplotlib.patheffects as path_effects

fig = plt.figure(figsize=(7, 1))
text = fig.text(0.5, 0.5, 'This text stands out because of\n'
                          'its black border.', color='white',
                          ha='center', va='center', size=30)
text.set_path_effects([path_effects.Stroke(linewidth=3, foreground='black'),
                       path_effects.Normal()])
plt.show()
```



It is important to note that this effect only works because we have drawn the text path twice; once with a thick black line, and then once with the original text path on top.

You may have noticed that the keywords to *Stroke* and *SimplePatchShadow* and *SimpleLineShadow* are not the usual Artist keywords (such as `facecolor` and `edgecolor` etc.). This is because with these path effects we are operating at lower level of matplotlib. In fact, the keywords which are accepted are those for a *matplotlib.backend_bases.GraphicsContextBase* instance, which have been designed for making it easy to create new backends - and not for its user interface.

4.7.3 Greater control of the path effect artist

As already mentioned, some of the path effects operate at a lower level than most users will be used to, meaning that setting keywords such as `facecolor` and `edgecolor` raise an AttributeError. Luckily there is a generic *PathPatchEffect* path effect which creates a *PathPatch* class with the original path. The keywords to this effect are identical to those of *PathPatch*:

```
import matplotlib.pyplot as plt
import matplotlib.patheffects as path_effects

fig = plt.figure(figsize=(8, 1))
t = fig.text(0.02, 0.5, 'Hatch shadow', fontsize=75, weight=1000, va='center')
t.set_path_effects([path_effects.PathPatchEffect(offset=(4, -4), hatch='xxxx',
                                                  facecolor='gray'),
                    path_effects.PathPatchEffect(edgecolor='white', linewidth=1.1,
                                                 facecolor='black')])
plt.show()
```

4.8 Our Favorite Recipes

Here is a collection of short tutorials, examples and code snippets that illustrate some of the useful idioms and tricks to make snazzier figures and overcome some matplotlib warts.

4.8.1 Sharing axis limits and views

It's common to make two or more plots which share an axis, e.g., two subplots with time as a common axis. When you pan and zoom around on one, you want the other to move around with you. To facilitate this, matplotlib Axes support a `sharex` and `sharey` attribute. When you create a *subplot()* or *axes()* instance, you can pass in a keyword indicating what axes you want to share with

```
In [96]: t = np.arange(0, 10, 0.01)

In [97]: ax1 = plt.subplot(211)

In [98]: ax1.plot(t, np.sin(2*np.pi*t))
Out[98]: [<matplotlib.lines.Line2D object at 0x98719ec>]

In [99]: ax2 = plt.subplot(212, sharex=ax1)

In [100]: ax2.plot(t, np.sin(4*np.pi*t))
Out[100]: [<matplotlib.lines.Line2D object at 0xb7d8fec>]
```

4.8.2 Easily creating subplots

In early versions of matplotlib, if you wanted to use the pythonic API and create a figure instance and from that create a grid of subplots, possibly with shared axes, it involved a fair amount of boilerplate code. e.g.

```
# old style
fig = plt.figure()
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(223, sharex=ax1, sharey=ax1)
ax3 = fig.add_subplot(224, sharex=ax1, sharey=ax1)
```

Fernando Perez has provided a nice top level method to create in `subplots()` (note the "s" at the end) everything at once, and turn off x and y sharing for the whole bunch. You can either unpack the axes individually:

```
# new style method 1; unpack the axes
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True, sharey=True)
ax1.plot(x)
```

or get them back as a numrows x numcolumns object array which supports numpy indexing:

```
# new style method 2; use an axes array
fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)
axs[0,0].plot(x)
```

4.8.3 Fixing common date annoyances

matplotlib allows you to natively plots python datetime instances, and for the most part does a good job picking tick locations and string formats. There are a couple of things it does not handle so gracefully, and here are some tricks to help you work around them. We'll load up some sample date data which contains datetime.date objects in a numpy record array:

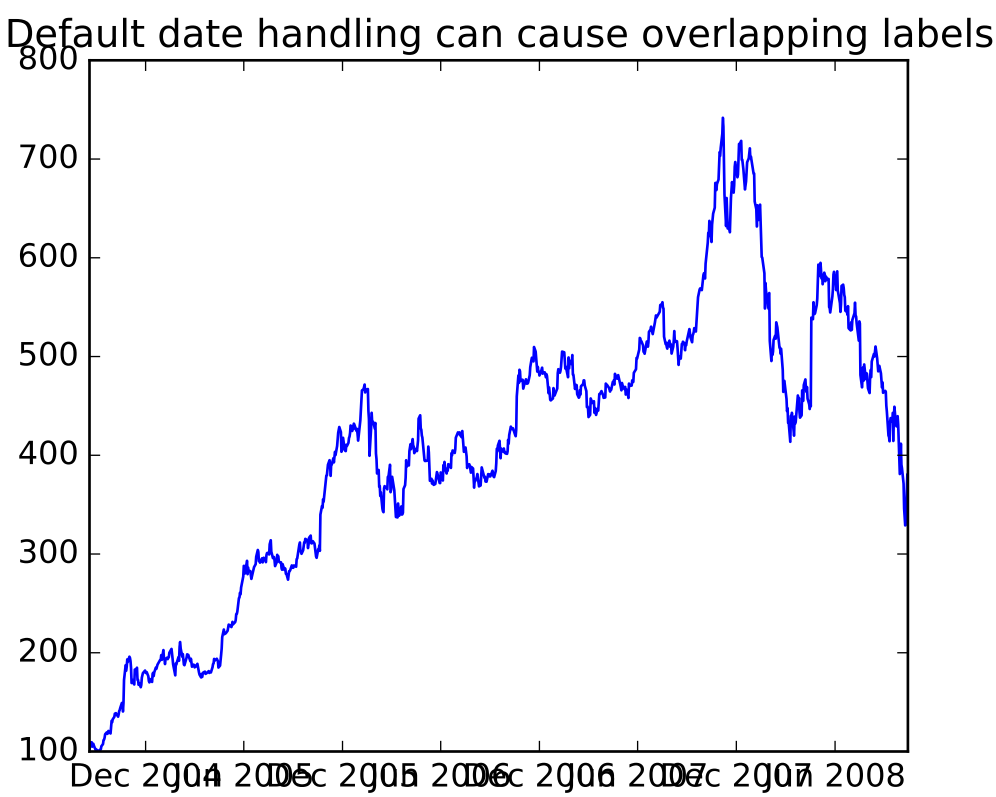
```
In [63]: datafile = cbook.get_sample_data('goog.npy')

In [64]: r = np.load(datafile).view(np.recarray)

In [65]: r.dtype
Out[65]: dtype([('date', '|O4'), ('', '|V4'), ('open', '<f8'),
                ('high', '<f8'), ('low', '<f8'), ('close', '<f8'),
                ('volume', '<i8'),  ('adj_close', '<f8')])

In [66]: r.date
Out[66]:
array([2004-08-19, 2004-08-20, 2004-08-23, ..., 2008-10-10, 2008-10-13,
       2008-10-14], dtype=object)
```

The dtype of the numpy record array for the field `date` is `|O4` which means it is a 4-byte python object pointer; in this case the objects are datetime.date instances, which we can see when we print some samples in the ipython terminal window.

If you plot the data,

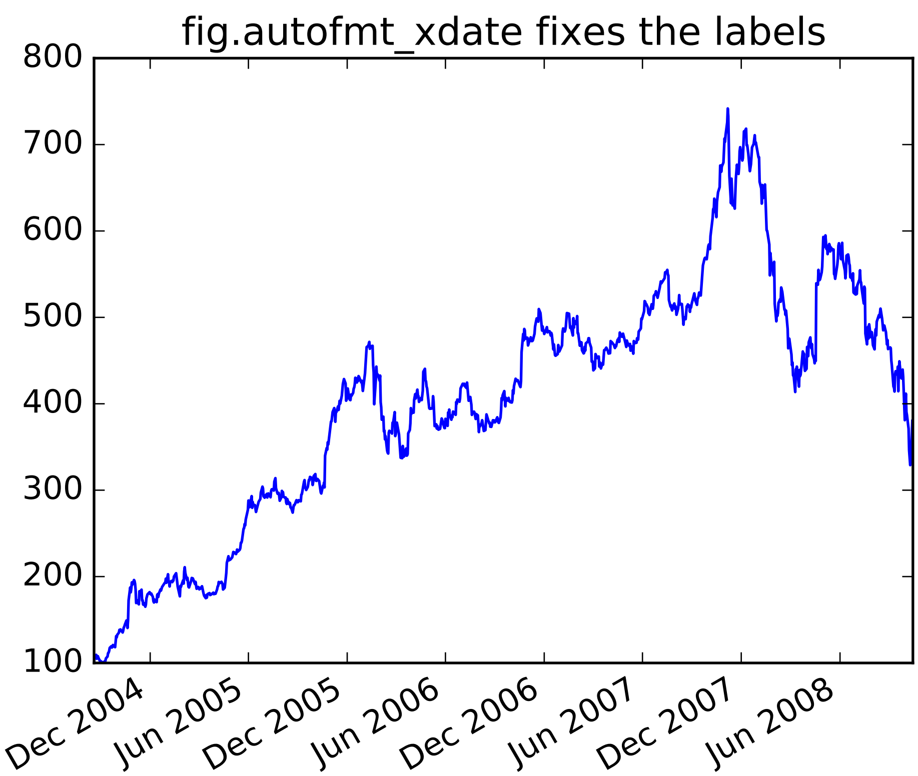
```
In [67]: plot(r.date, r.close)
Out[67]: [<matplotlib.lines.Line2D object at 0x92a6b6c>]
```

you will see that the x tick labels are all squashed together.



Another annoyance is that if you hover the mouse over the window and look in the lower right corner of the matplotlib toolbar (*Interactive navigation*) at the x and y coordinates, you see that the x locations are formatted the same way the tick labels are, e.g., "Dec 2004". What we'd like is for the location in the toolbar to have a higher degree of precision, e.g., giving us the exact date out mouse is hovering over. To fix the first problem, we can use `matplotlib.figure.Figure.autofmt_xdate()` and to fix the second problem we can use the `ax.fmt_xdata` attribute which can be set to any function that takes a scalar and returns a string. matplotlib has a number of date formatters built in, so we'll use one of those.

```
plt.close('all')
fig, ax = plt.subplots(1)
ax.plot(r.date, r.close)

# rotate and align the tick labels so they look better
fig.autofmt_xdate()

# use a more precise date string for the x axis locations in the
# toolbar
import matplotlib.dates as mdates
ax.fmt_xdata = mdates.DateFormatter('%Y-%m-%d')
plt.title('fig.autofmt_xdate fixes the labels')
```



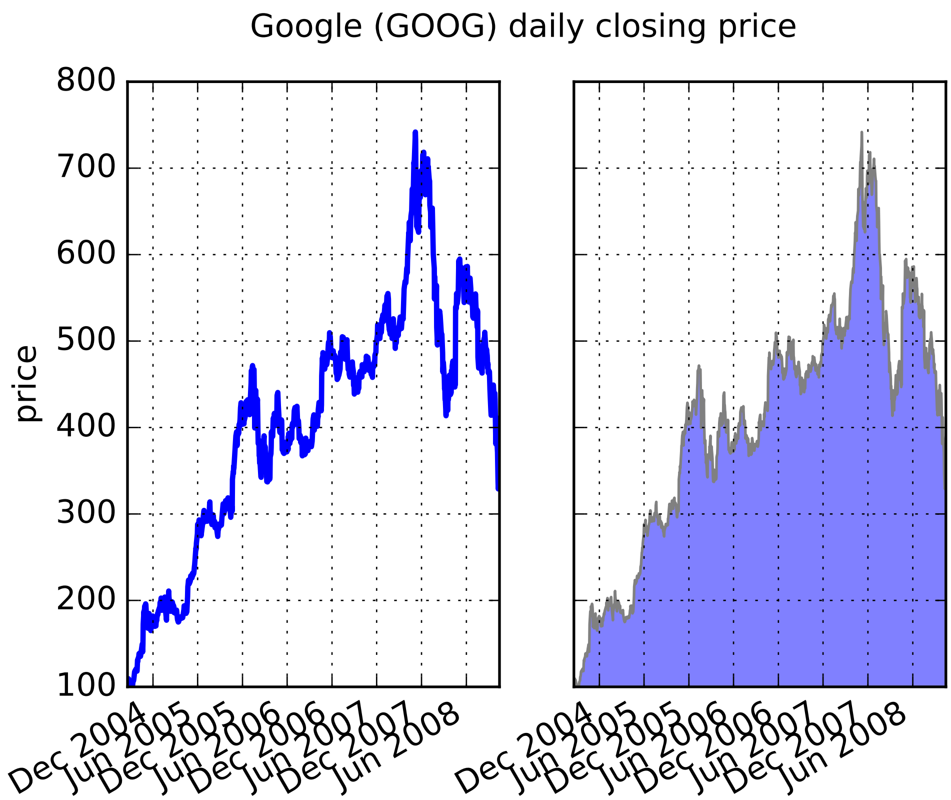
Now when you hover your mouse over the plotted data, you'll see date format strings like 2004-12-01 in the toolbar.

4.8.4 Fill Between and Alpha

The *fill_between()* function generates a shaded region between a min and max boundary that is useful for illustrating ranges. It has a very handy `where` argument to combine filling with logical ranges, e.g., to just fill in a curve over some threshold value.

At its most basic level, `fill_between` can be use to enhance a graphs visual appearance. Let's compare two graphs of a financial times with a simple line plot on the left and a filled line on the right.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook

# load up some sample financial data
datafile = cbook.get_sample_data('goog.npy')
try:
    # Python3 cannot load python2 .npy files with datetime(object) arrays
    # unless the encoding is set to bytes. Hovever this option was
    # not added until numpy 1.10 so this example will only work with
    # python 2 or with numpy 1.10 and later.
    r = np.load(datafile, encoding='bytes').view(np.recarray)
except TypeError:
    r = np.load(datafile).view(np.recarray)
# create two subplots with the shared x and y axes
fig, (ax1, ax2) = plt.subplots(1,2, sharex=True, sharey=True)

pricemin = r.close.min()

ax1.plot(r.date, r.close, lw=2)
ax2.fill_between(r.date, pricemin, r.close, facecolor='blue', alpha=0.5)

for ax in ax1, ax2:
    ax.grid(True)

ax1.set_ylabel('price')
for label in ax2.get_yticklabels():
    label.set_visible(False)

fig.suptitle('Google (GOOG) daily closing price')
fig.autofmt_xdate()
```

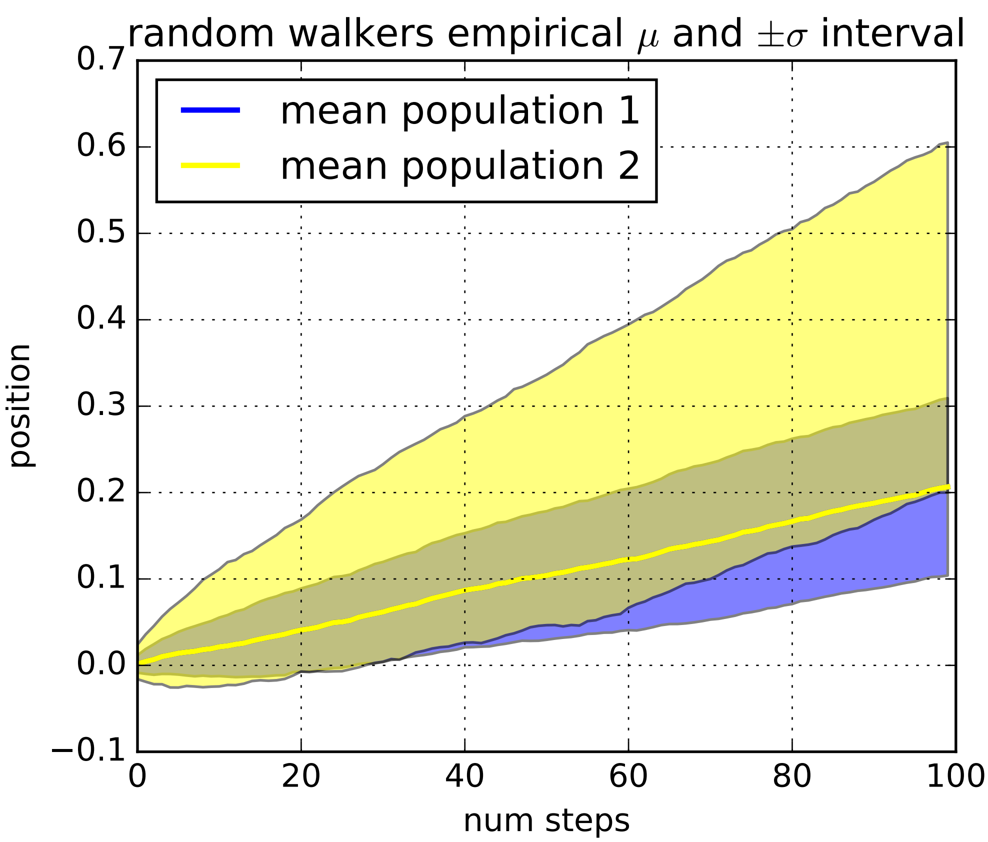


The alpha channel is not necessary here, but it can be used to soften colors for more visually appealing plots. In other examples, as we'll see below, the alpha channel is functionally useful as the shaded regions can overlap and alpha allows you to see both. Note that the postscript format does not support alpha (this is a postscript limitation, not a matplotlib limitation), so when using alpha save your figures in PNG, PDF or SVG.

Our next example computes two populations of random walkers with a different mean and standard deviation of the normal distributions from which the steps are drawn. We use shared regions to plot +/- one standard deviation of the mean position of the population. Here the alpha channel is useful, not just aesthetic.

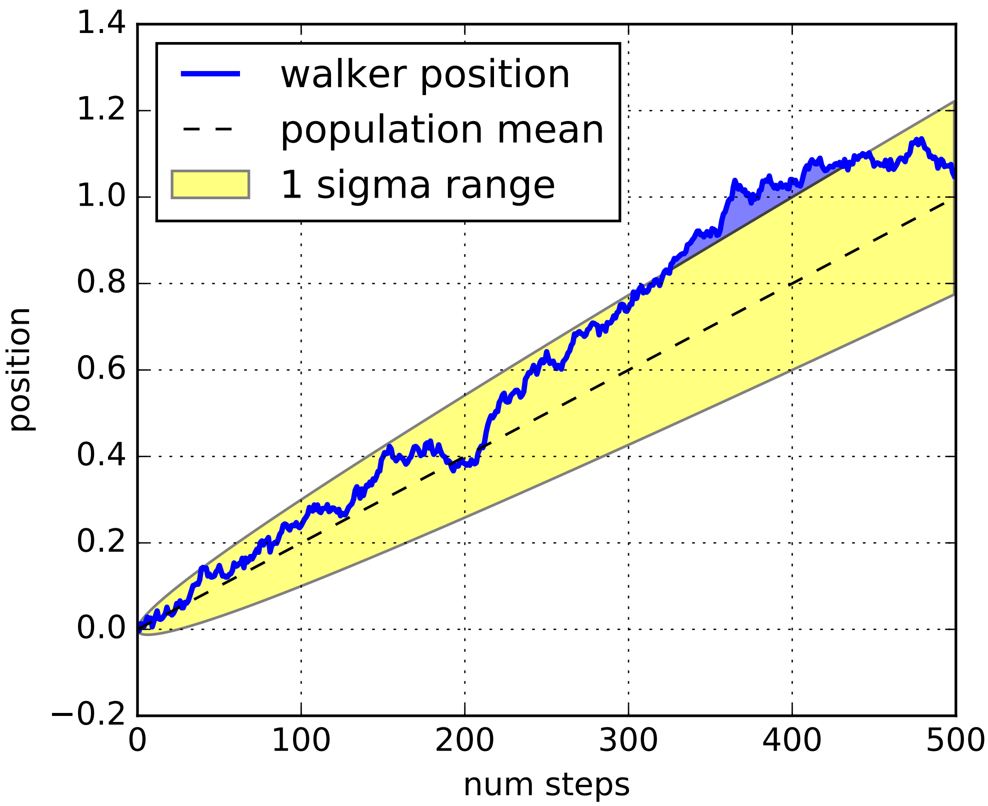
```
import matplotlib.pyplot as plt
import numpy as np

Nsteps, Nwalkers = 100, 250
t = np.arange(Nsteps)

# an (Nsteps x Nwalkers) array of random walk steps
S1 = 0.002 + 0.01*np.random.randn(Nsteps, Nwalkers)
S2 = 0.004 + 0.02*np.random.randn(Nsteps, Nwalkers)

# an (Nsteps x Nwalkers) array of random walker positions
X1 = S1.cumsum(axis=0)
X2 = S2.cumsum(axis=0)
```

```
# Nsteps length arrays empirical means and standard deviations of both
# populations over time
mu1 = X1.mean(axis=1)
sigma1 = X1.std(axis=1)
mu2 = X2.mean(axis=1)
sigma2 = X2.std(axis=1)

# plot it!
fig, ax = plt.subplots(1)
ax.plot(t, mu1, lw=2, label='mean population 1', color='blue')
ax.plot(t, mu1, lw=2, label='mean population 2', color='yellow')
ax.fill_between(t, mu1+sigma1, mu1-sigma1, facecolor='blue', alpha=0.5)
ax.fill_between(t, mu2+sigma2, mu2-sigma2, facecolor='yellow', alpha=0.5)
ax.set_title('random walkers empirical $\mu$ and $\pm \sigma$ interval')
ax.legend(loc='upper left')
ax.set_xlabel('num steps')
ax.set_ylabel('position')
ax.grid()
```



The `where` keyword argument is very handy for highlighting certain regions of the graph. `where` takes a boolean mask the same length as the x, ymin and ymax arguments, and only fills in the region where the boolean mask is True. In the example below, we simulate a single random walker and compute the analytic mean and standard deviation of the population positions. The population mean is shown as the black dashed line, and the plus/minus one sigma deviation from the mean is shown as the yellow filled region. We use the where mask `X>upper_bound` to find the region where the walker is above the one sigma boundary, and shade that region blue.

```
np.random.seed(1234)

Nsteps = 500
t = np.arange(Nsteps)

mu = 0.002
sigma = 0.01

# the steps and position
S = mu + sigma*np.random.randn(Nsteps)
X = S.cumsum()

# the 1 sigma upper and lower analytic population bounds
lower_bound = mu*t - sigma*np.sqrt(t)
upper_bound = mu*t + sigma*np.sqrt(t)

fig, ax = plt.subplots(1)
ax.plot(t, X, lw=2, label='walker position', color='blue')
ax.plot(t, mu*t, lw=1, label='population mean', color='black', ls='--')
ax.fill_between(t, lower_bound, upper_bound, facecolor='yellow', alpha=0.5,
                label='1 sigma range')
ax.legend(loc='upper left')

# here we use the where argument to only fill the region where the
# walker is above the population 1 sigma boundary
ax.fill_between(t, upper_bound, X, where=X>upper_bound, facecolor='blue', alpha=0.5)
ax.set_xlabel('num steps')
ax.set_ylabel('position')
ax.grid()
```



Another handy use of filled regions is to highlight horizontal or vertical spans of an axes – for that matplotlib has some helper functions `axhspan()` and `axvspan()` and example *pylab_examples example code: axhspan_demo.py*.

4.8.5 Transparent, fancy legends

Sometimes you know what your data looks like before you plot it, and may know for instance that there won't be much data in the upper right hand corner. Then you can safely create a legend that doesn't overlay your data:

```
ax.legend(loc='upper right')
```

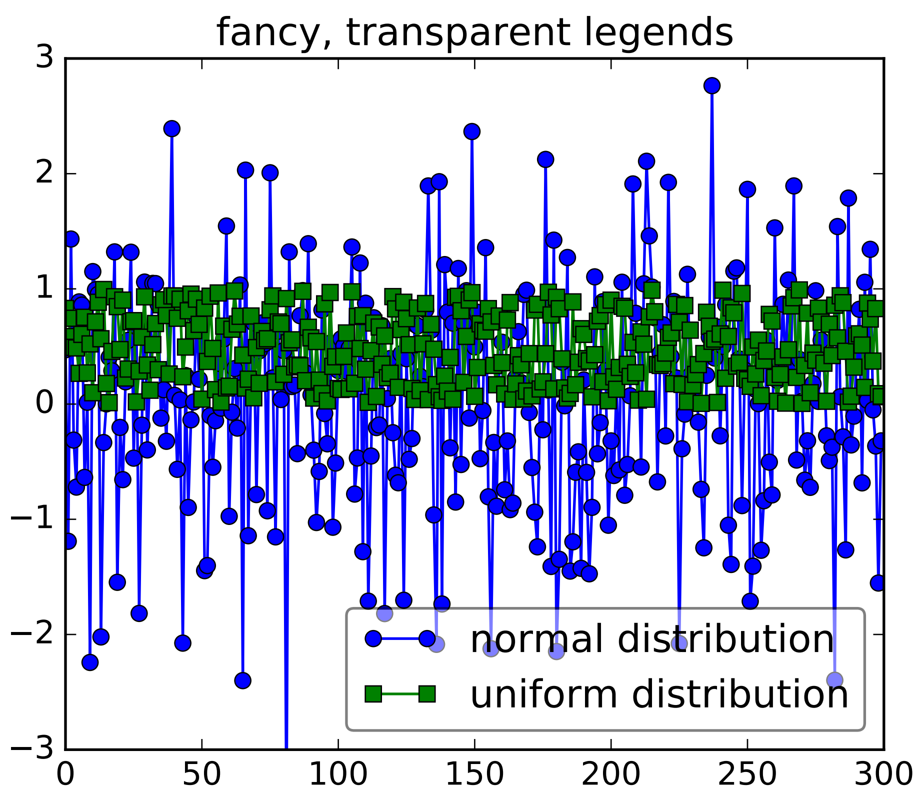
Other times you don't know where your data is, and loc='best' will try and place the legend:

```
ax.legend(loc='best')
```

but still, your legend may overlap your data, and in these cases it's nice to make the legend frame transparent.

```
np.random.seed(1234)
fig, ax = plt.subplots(1)
ax.plot(np.random.randn(300), 'o-', label='normal distribution')
ax.plot(np.random.rand(300), 's-', label='uniform distribution')
ax.set_ylim(-3, 3)
ax.legend(loc='best', fancybox=True, framealpha=0.5)
```

```
ax.set_title('fancy, transparent legends')
```

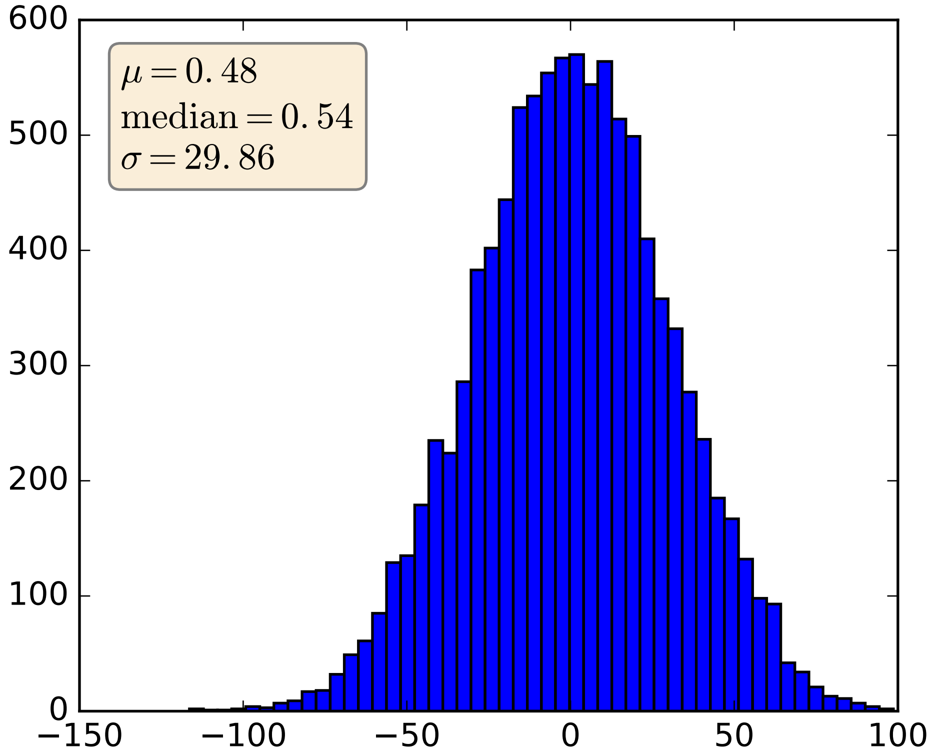


4.8.6 Placing text boxes

When decorating axes with text boxes, two useful tricks are to place the text in axes coordinates (see *Transformations Tutorial*), so the text doesn't move around with changes in x or y limits. You can also use the `bbox` property of text to surround the text with a `Patch` instance – the `bbox` keyword argument takes a dictionary with keys that are Patch properties.

```
np.random.seed(1234)
fig, ax = plt.subplots(1)
x = 30*np.random.randn(10000)
mu = x.mean()
median = np.median(x)
sigma = x.std()
textstr = '$\mu=%.2f$\n$\mathrm{median}=%.2f$\n$\sigma=%.2f$'%(mu, median, sigma)

ax.hist(x, 50)
# these are matplotlib.patch.Patch properties
props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)

# place a text box in upper left in axes coords
```

```
ax.text(0.05, 0.95, textstr, transform=ax.transAxes, fontsize=14,
        verticalalignment='top', bbox=props)
```



CHAPTER

FIVE

WHAT'S NEW IN MATPLOTLIB

For a list of all of the issues and pull requests since the last revision, see the *Github Stats*.

Note: matplotlib 1.5 supports Python 2.7, 3.4, and 3.5

matplotlib 1.4 supports Python 2.6, 2.7, 3.3, and 3.4

matplotlib 1.3 supports Python 2.6, 2.7, 3.2, and 3.3

matplotlib 1.2 supports Python 2.6, 2.7, and 3.1

matplotlib 1.1 supports Python 2.4 to 2.7

Table of Contents

5.1 new in matplotlib-1.5

5.1.1 Interactive OO usage

All `Artists` now keep track of if their internal state has been changed but not reflected in the display ('stale') by a call to `draw`. It is thus possible to pragmatically determine if a given `Figure` needs to be re-drawn in an interactive session.

To facilitate interactive usage a `draw_all` method has been added to `pyplot` which will redraw all of the figures which are 'stale'.

To make this convenient for interactive use matplotlib now registers a function either with IPython's 'post_execute' event or with the displayhook in the standard python REPL to automatically call `plt.draw_all` just before control is returned to the REPL. This ensures that the draw command is deferred and only called once.

The upshot of this is that for interactive backends (including `%matplotlib notebook`) in interactive mode (with `plt.ion()`)

```
In [1]: import matplotlib.pyplot as plt

In [2]: fig, ax = plt.subplots()
```

```
In [3]: ln, = ax.plot([0, 1, 4, 9, 16])

In [4]: plt.show()

In [5]: ln.set_color('g')
```

will automatically update the plot to be green. Any subsequent modifications to the `Artist` objects will do likewise.

This is the first step of a larger consolidation and simplification of the pyplot internals.

5.1.2 Working with labeled data like pandas DataFrames

Plot methods which take arrays as inputs can now also work with labeled data and unpack such data.

This means that the following two examples produce the same plot:

Example

```
df = pandas.DataFrame({"var1":[1,2,3,4,5,6], "var2":[1,2,3,4,5,6]})
plt.plot(df["var1"], df["var2"])
```

Example

```
plt.plot("var1", "var2", data=df)
```

This works for most plotting methods, which expect arrays/sequences as inputs. `data` can be anything which supports `__getitem__` (`dict`, `pandas.DataFrame`, `h5py`, ...) to access `array` like values with string keys.

In addition to this, some other changes were made, which makes working with labeled data (ex `pandas.Series`) easier:

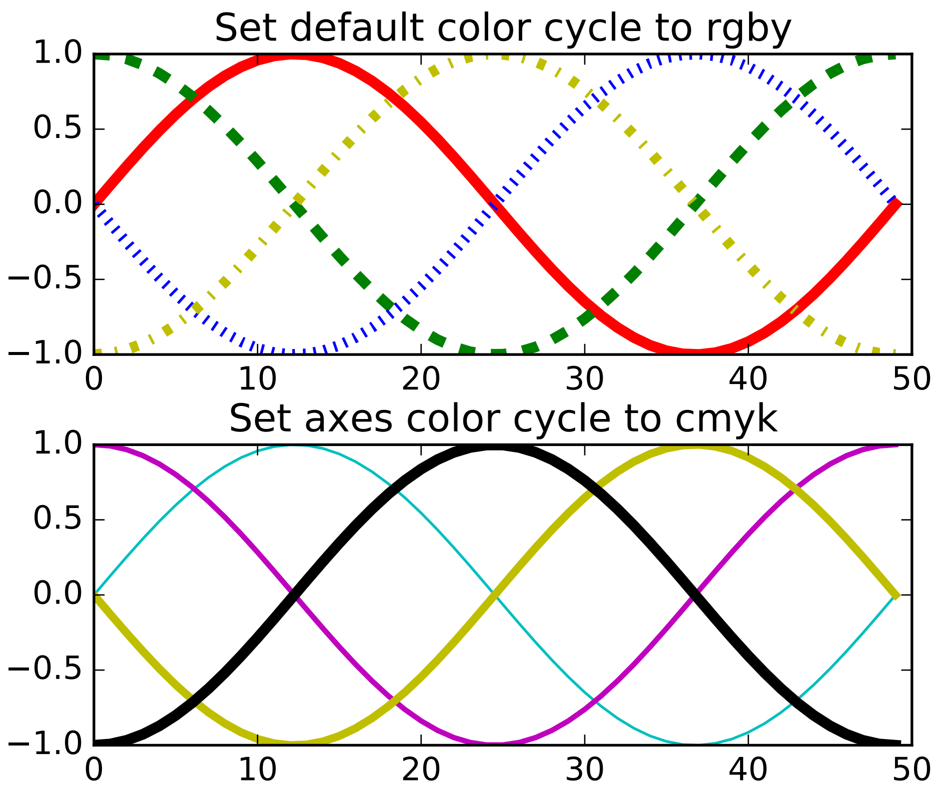
- For plotting methods with `label` keyword argument, one of the data inputs is designated as the label source. If the user does not supply a `label` that value object will be introspected for a label, currently by looking for a `name` attribute. If the value object does not have a `name` attribute but was specified by as a key into the `data` kwarg, then the key is used. In the above examples, this results in an implicit `label="var2"` for both cases.
- `plot()` now uses the index of a `Series` instead of `np.arange(len(y))`, if no `x` argument is supplied.

5.1.3 Added `axes.prop_cycle` key to rcParams

This is a more generic form of the now-deprecated `axes.color_cycle` param. Now, we can cycle more than just colors, but also linestyles, hatches, and just about any other artist property. Cycler notation is used for defining property cycles. Adding cyclers together will be like you are `zip()`-ing together two or more property cycles together:

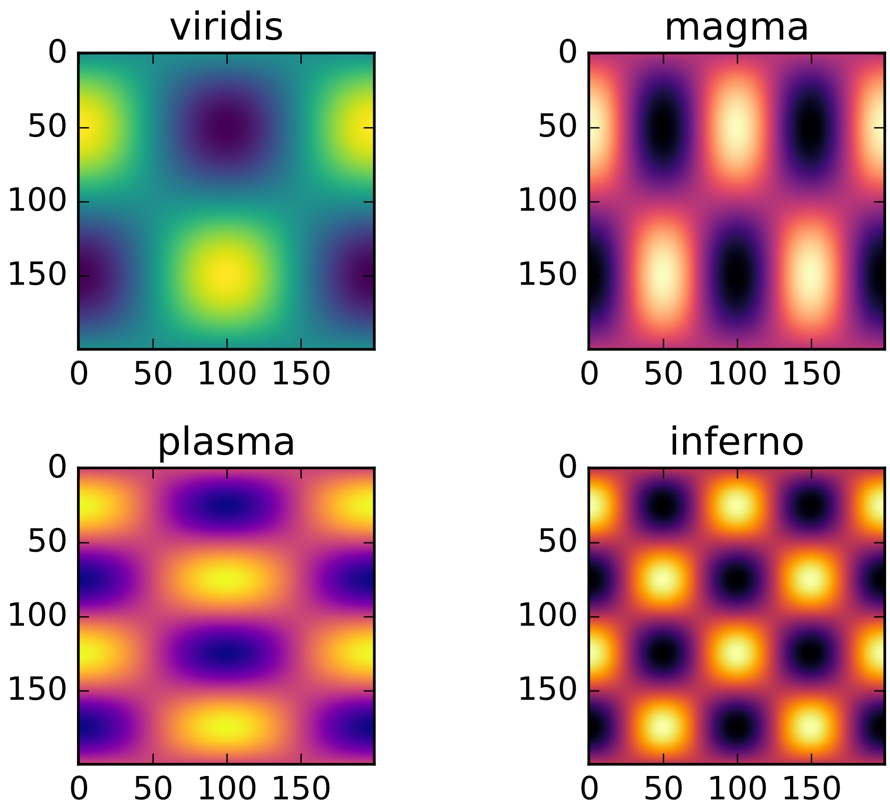
```
axes.prop_cycle: cycler('color', 'rgb') + cycler('lw', [1, 2, 3])
```

You can even multiply cyclers, which is like using `itertools.product()` on two or more property cycles. Remember to use parentheses if writing a multi-line `prop_cycle` parameter.



5.1.4 New Colormaps

All four of the colormaps proposed as the new default are available as `'viridis'` (the new default in 2.0), `'magma'`, `'plasma'`, and `'inferno'`



5.1.5 Styles

Several new styles have been added, including many styles from the Seaborn project. Additionally, in order to prep for the upcoming 2.0 style-change release, a 'classic' and 'default' style has been added. For this release, the 'default' and 'classic' styles are identical. By using them now in your scripts, you can help ensure a smooth transition during future upgrades of matplotlib, so that you can upgrade to the snazzy new defaults when you are ready!

```
import matplotlib.style
matplotlib.style.use('classic')
```

The 'default' style will give you matplotlib's latest plotting styles:

```
matplotlib.style.use('default')
```

5.1.6 Backends

New backend selection

The environment variable *MPLBACKEND* can now be used to set the matplotlib backend.

wx backend has been updated

The wx backend can now be used with both wxPython classic and Phoenix.

wxPython classic has to be at least version 2.8.12 and works on Python 2.x. As of May 2015 no official release of wxPython Phoenix is available but a current snapshot will work on Python 2.7+ and 3.4+.

If you have multiple versions of wxPython installed, then the user code is responsible setting the wxPython version. How to do this is explained in the comment at the beginning of the example `examplesuser_interfacesembedding_in_wx2.py`.

5.1.7 Configuration (rcParams)

Some parameters have been added, others have been improved.

Parameter	Description
`{x,y}axis.labelpad`	mplot3d now respects these parameters
`axes.labelpad`	Default space between the axis and the label
`errorbar.capsize`	Default length of end caps on error bars
`{x,y}tick.minor.visible`	Default visibility of minor x/y ticks
`legend.framealpha`	Default transparency of the legend frame box
`legend.facecolor`	Default facecolor of legend frame box (or `'inherit'` from `axes.facecolor`)
`legend.edgecolor`	Default edgecolor of legend frame box (or `'inherit'` from `axes.edgecolor`)
`figure.titlesize`	Default font size for figure suptitles
`figure.titleweight`	Default font weight for figure suptitles
`image.composite_image`	Whether a vector graphics backend should composite several images into a single image or not when saving. Useful when needing to edit the files further in Inkscape or other programs.
`markers.fillstyle`	Default fillstyle of markers. Possible values are `'full'` (the default), `'left'`, `'right'`, `'bottom'`, `'top'` and `'none'`
`toolbar`	Added `'toolmanager'` as a valid value, enabling the experimental `ToolManager` feature.

5.1.8 Widgets

Active state of Selectors

All selectors now implement `set_active` and `get_active` methods (also called when accessing the `active` property) to properly update and query whether they are active.

Moved `ignore`, `set_active`, and `get_active` methods to base class `Widget`

Pushes up duplicate methods in child class to parent class to avoid duplication of code.

Adds enable/disable feature to MultiCursor

A MultiCursor object can be disabled (and enabled) after it has been created without destroying the object. Example:

```
multi_cursor.active = False
```

Improved RectangleSelector and new EllipseSelector Widget

Adds an `interactive` keyword which enables visible handles for manipulating the shape after it has been drawn.

Adds keyboard modifiers for:

- Moving the existing shape (default key = 'space')
- Making the shape square (default 'shift')
- Make the initial point the center of the shape (default 'control')
- Square and center can be combined

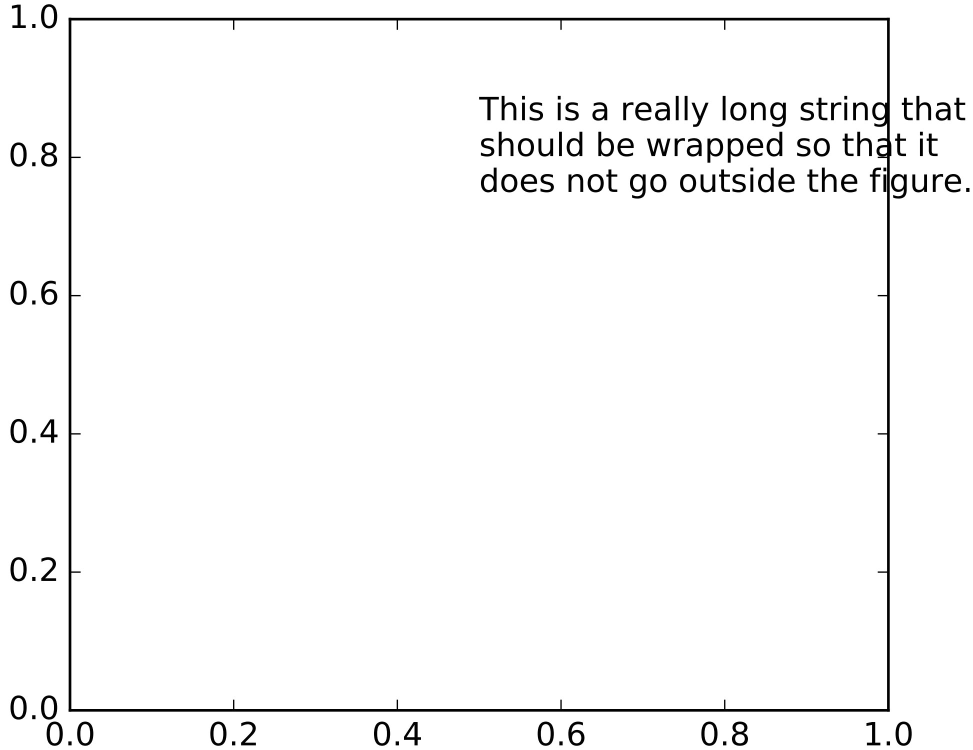
Allow Artists to Display Pixel Data in Cursor

Adds `get_pixel_data` and `format_pixel_data` methods to artists which can be used to add zdata to the cursor display in the status bar. Also adds an implementation for Images.

5.1.9 New plotting features

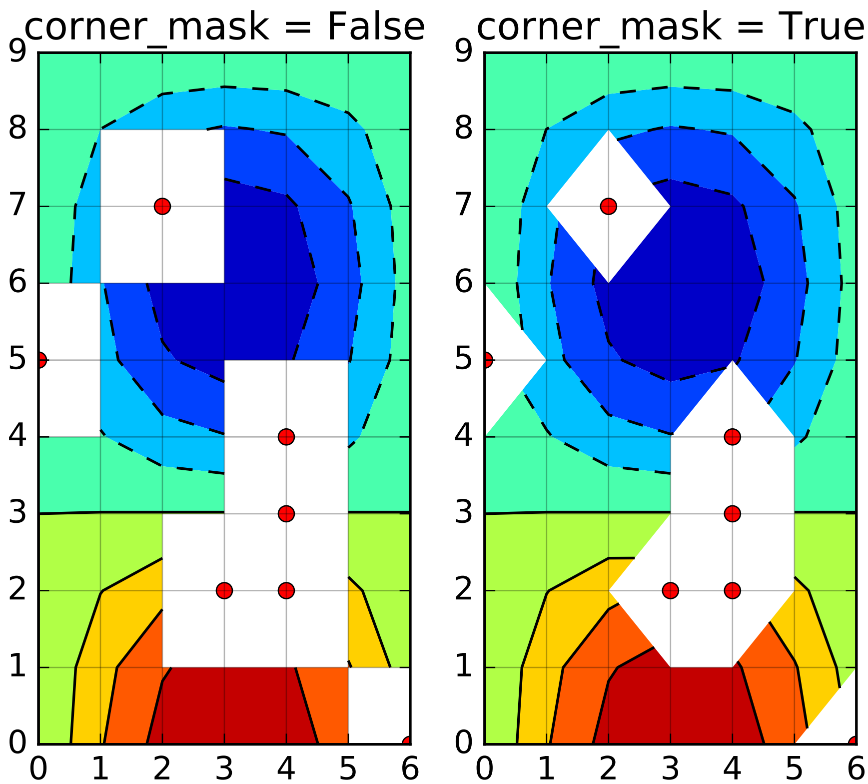
Auto-wrapping Text

Added the keyword argument "wrap" to Text, which automatically breaks long lines of text when being drawn. Works for any rotated text, different modes of alignment, and for text that are either labels or titles. This breaks at the `Figure`, not `Axes` edge.



Contour plot corner masking

Ian Thomas rewrote the C++ code that calculates contours to add support for corner masking. This is controlled by a new keyword argument `corner_mask` in the functions *contour()* and *contourf()*. The previous behaviour, which is now obtained using `corner_mask=False`, was for a single masked point to completely mask out all four quads touching that point. The new behaviour, obtained using `corner_mask=True`, only masks the corners of those quads touching the point; any triangular corners comprising three unmasked points are contoured as usual. If the `corner_mask` keyword argument is not specified, the default value is taken from rcParams.



Mostly unified linestyles for `Line2D`, `Patch` and `Collection`

The handling of linestyles for Lines, Patches and Collections has been unified. Now they all support defining linestyles with short symbols, like `"--"`, as well as with full names, like `"dashed"`. Also the definition using a dash pattern (`(0., [3., 3.])`) is supported for all methods using `Line2D`, `Patch` or `Collection`.

Legend marker order

Added ability to place the label before the marker in a legend box with `markerfirst` keyword

Support for legend for PolyCollection and stackplot

Added a `legend_handler` for *`PolyCollection`* as well as a `labels` argument to *`stackplot()`*.

Support for alternate pivots in mplot3d quiver plot

Added a `pivot` kwarg to *`quiver()`* that controls the pivot point around which the quiver line rotates. This also determines the placement of the arrow head along the quiver line.

Logit Scale

Added support for the 'logit' axis scale, a nonlinear transformation

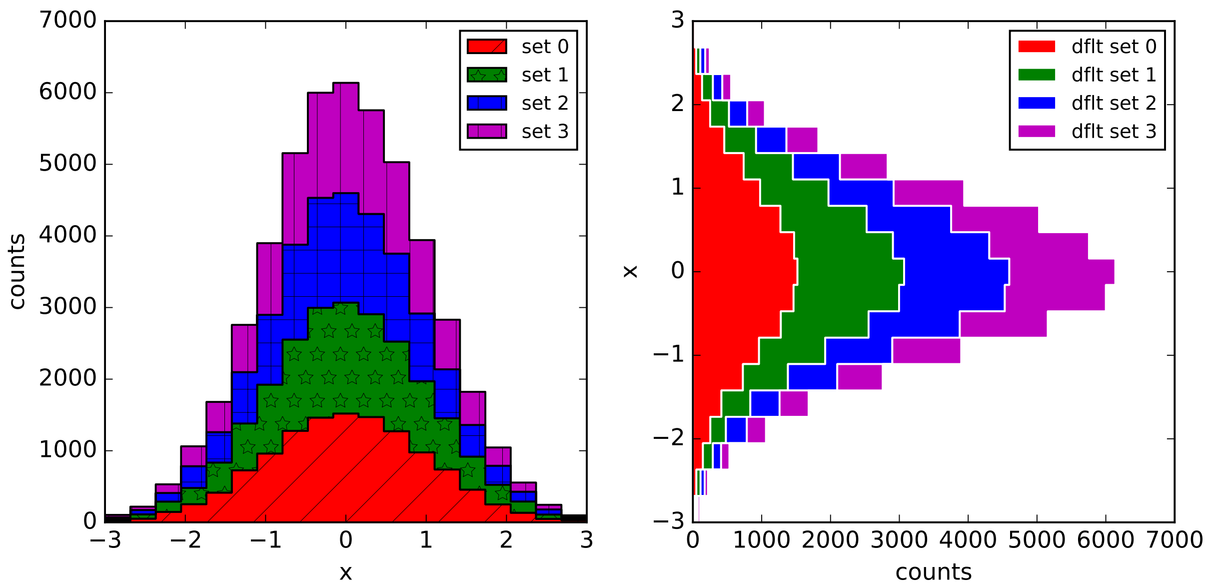
$$x- > \log 10(x/(1-x)) \tag{5.1}$$

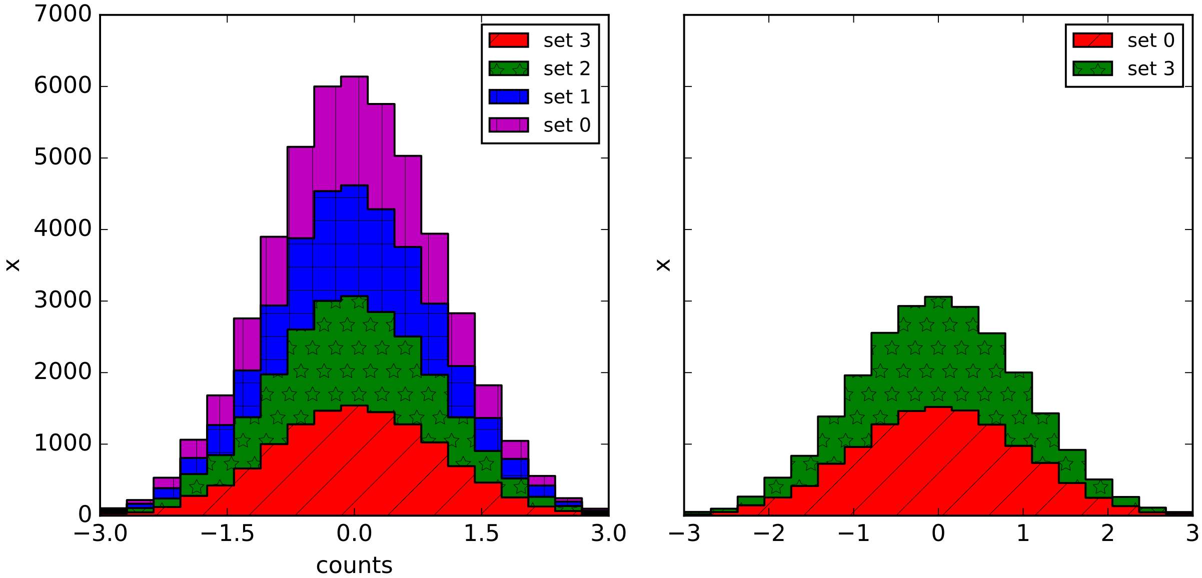
for data between 0 and 1 excluded.

Add step kwargs to fill_between

Added `step` kwarg to `Axes.fill_between` to allow to fill between lines drawn using the 'step' draw style. The values of `step` match those of the `where` kwarg of `Axes.step`. The asymmetry of of the kwargs names is not ideal, but `Axes.fill_between` already has a `where` kwarg.

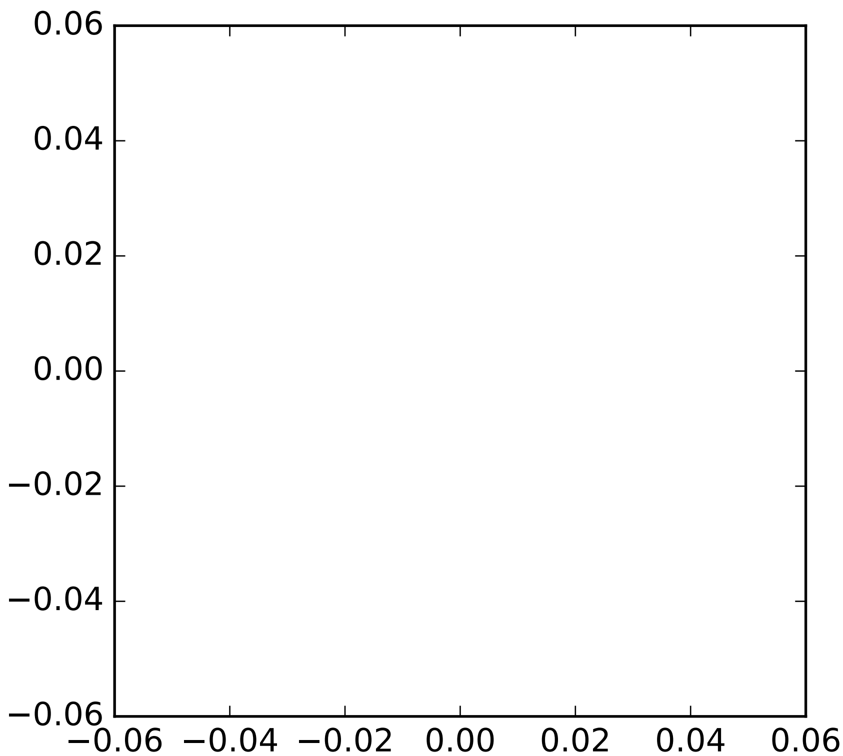
This is particularly useful for plotting pre-binned histograms.





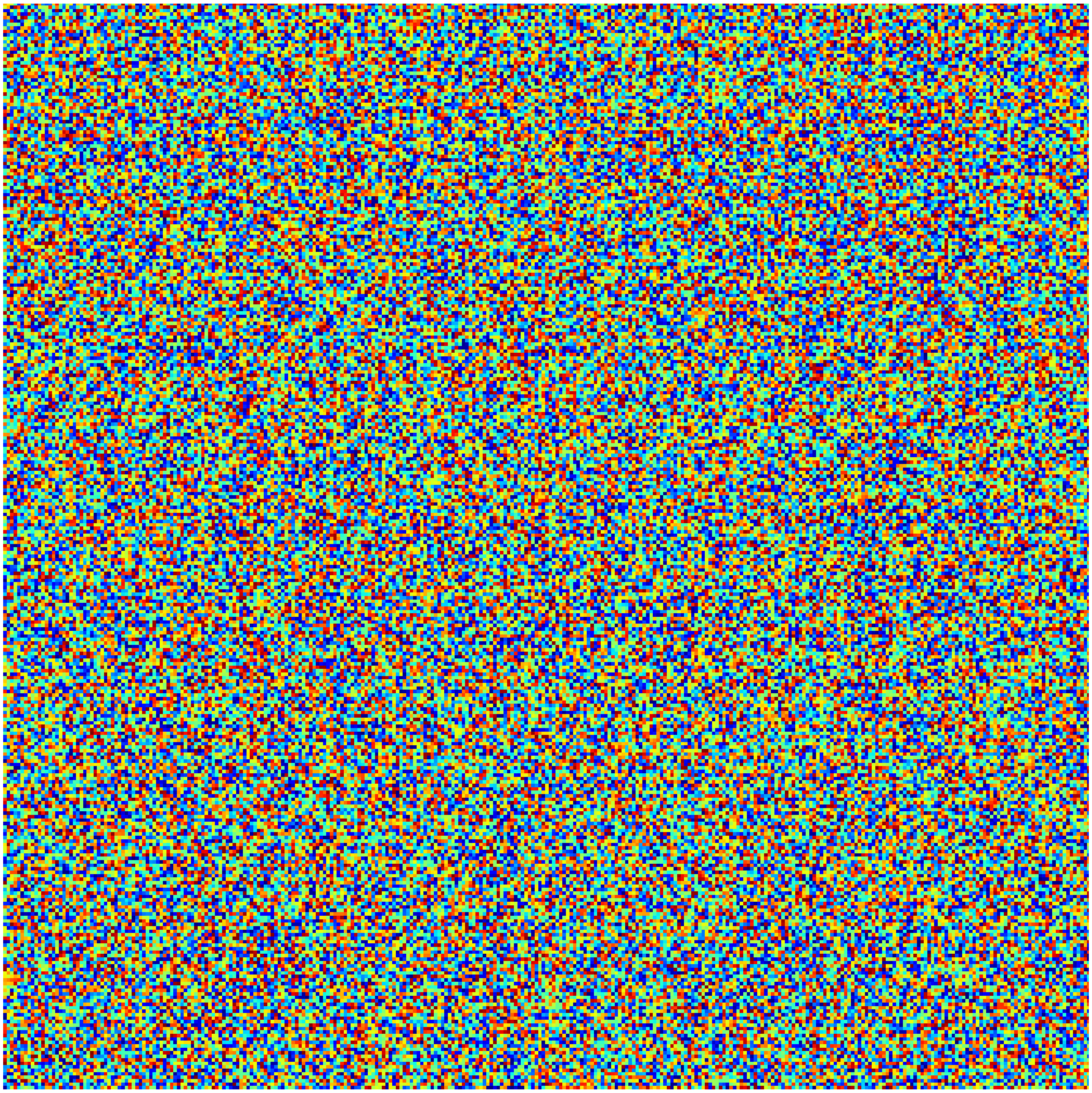
Square Plot

Implemented square plots feature as a new parameter in the axis function. When argument 'square' is specified, equal scaling is set, and the limits are set such that `xmax-xmin == ymax-ymin`.



Updated figimage to take optional resize parameter

Added the ability to plot simple 2D-Array using `plt.figimage(X, resize=True)`. This is useful for plotting simple 2D-Array without the Axes or whitespacing around the image.



Updated Figure.savefig() can now use figure's dpi

Added support to save the figure with the same dpi as the figure on the screen using `dpi='figure'`.

Example:

```
f = plt.figure(dpi=25)              # dpi set to 25
S = plt.scatter([1,2,3],[4,5,6])
f.savefig('output.png', dpi='figure')    # output savefig dpi set to 25 (same as figure)
```

Updated Table to control edge visibility

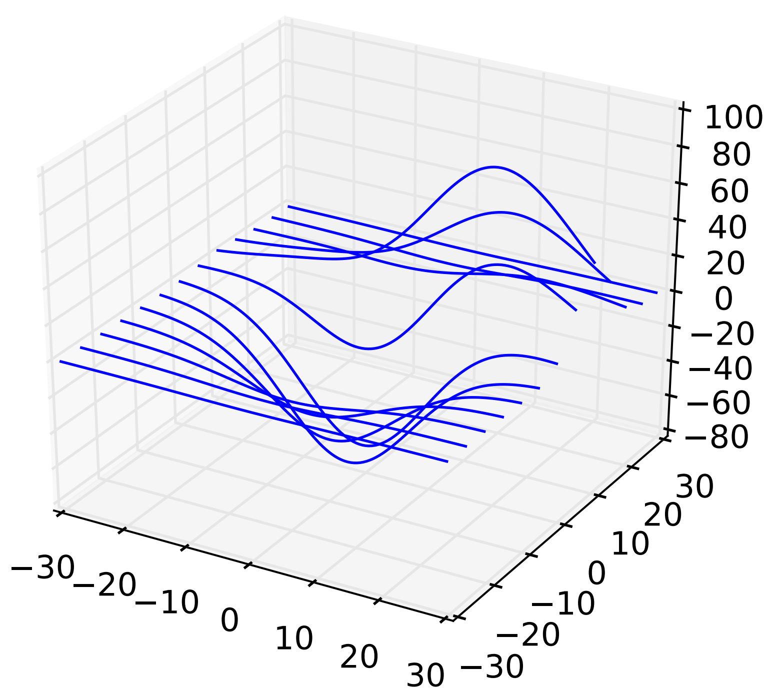
Added the ability to toggle the visibility of lines in Tables. Functionality added to the `pyplot.table()` factory function under the keyword argument "edges". Values can be the strings "open", "closed", "horizontal", "vertical" or combinations of the letters "L", "R", "T", "B" which represent left, right, top, and bottom respectively.

Example:

```
table(..., edges="open")  # No line visible
table(..., edges="closed")  # All lines visible
table(..., edges="horizontal")  # Only top and bottom lines visible
table(..., edges="LT")  # Only left and top lines visible.
```

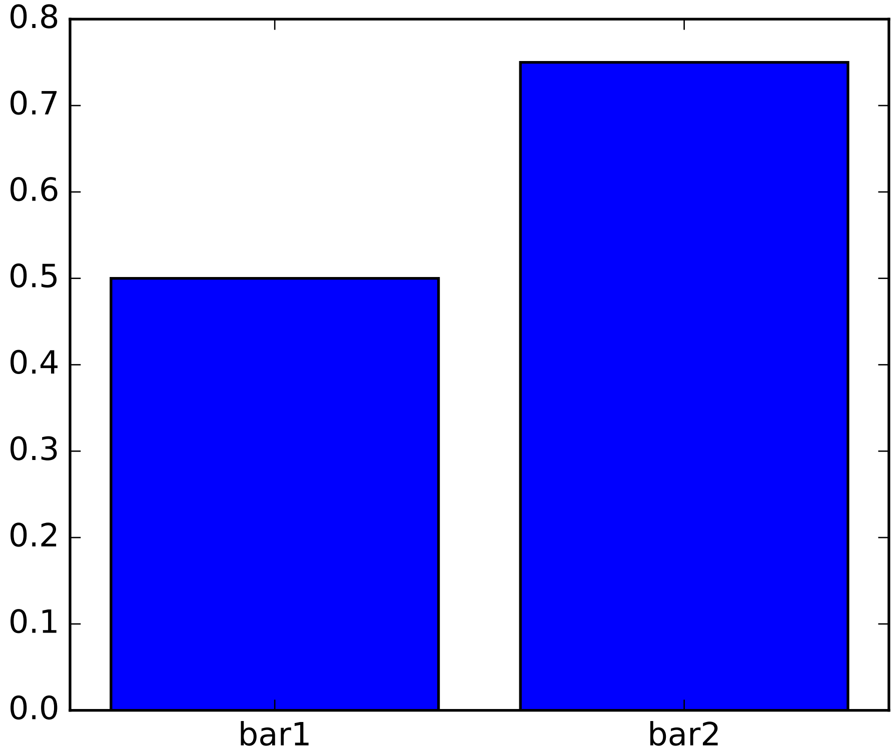
Zero r/cstride support in plot_wireframe

Adam Hughes added support to mplot3d's plot_wireframe to draw only row or column line plots.



Plot bar and barh with labels

Added kwarg `"tick_label"` to `bar` and `barh` to support plotting bar graphs with a text label for each bar.



Added center and frame kwargs to pie

These control where the center of the pie graph are and if the Axes frame is shown.

Fixed 3D filled contour plot polygon rendering

Certain cases of 3D filled contour plots that produce polygons with multiple holes produced improper rendering due to a loss of path information between `PolyCollection` and `Poly3DCollection`. A function `set_verts_and_codes()` was added to allow path information to be retained for proper rendering.

Dense colorbars are rasterized

Vector file formats (pdf, ps, svg) are efficient for many types of plot element, but for some they can yield excessive file size and even rendering artifacts, depending on the renderer used for screen display. This is a problem for colorbars that show a large number of shades, as is most commonly the case. Now, if a colorbar is showing 50 or more colors, it will be rasterized in vector backends.

DateFormatter strftime

`strftime` method will format a `datetime.datetime` object with the format string passed to the formatter's constructor. This method accepts datetimes with years before 1900, unlike `datetime.datetime.strftime()`.

Artist-level {get,set}_usetex for text

Add `{get,set}_usetex` methods to *Text* objects which allow artist-level control of LaTeX rendering vs the internal mathtex rendering.

`ax.remove()` works as expected

As with artists added to an *Axes*, `Axes` objects can be removed from their figure via *remove()*.

API Consistency fix within Locators set_params() function

set_params() function, which sets parameters within a *Locator* type instance, is now available to all `Locator` types. The implementation also prevents unsafe usage by strictly defining the parameters that a user can set.

To use, call `set_params()` on a `Locator` instance with desired arguments:

```
loc = matplotlib.ticker.LogLocator()
# Set given attributes for loc.
loc.set_params(numticks=8, numdecs=8, subs=[2.0], base=8)
# The below will error, as there is no such parameter for LogLocator
# named foo
# loc.set_params(foo='bar')
```

Date Locators

Date Locators (derived from *DateLocator*) now implement the `tick_values()` method. This is expected of all Locators derived from `Locator`.

The Date Locators can now be used easily without creating axes

```
from datetime import datetime
from matplotlib.dates import YearLocator
t0 = datetime(2002, 10, 9, 12, 10)
tf = datetime(2005, 10, 9, 12, 15)
loc = YearLocator()
values = loc.tick_values(t0, tf)
```

OffsetBoxes now support clipping

`Artists` draw onto objects of type `OffsetBox` through `DrawingArea` and `TextArea`. The `TextArea` calculates the required space for the text and so the text is always within the bounds, for this nothing has changed.

However, `DrawingArea` acts as a parent for zero or more `Artists` that draw on it and may do so beyond the bounds. Now child `Artists` can be clipped to the bounds of the `DrawingArea`.

OffsetBoxes now considered by tight_layout

When *`tight_layout()`* or `Figure.tight_layout()` or `GridSpec.tight_layout()` is called, `OffsetBoxes` that are anchored outside the axes will not get chopped out. The `OffsetBoxes` will also not get overlapped by other axes in case of multiple subplots.

Per-page pdf notes in multi-page pdfs (PdfPages)

Add a new method *`attach_note()`* to the PdfPages class, allowing the attachment of simple text notes to pages in a multi-page pdf of figures. The new note is visible in the list of pdf annotations in a viewer that has this facility (Adobe Reader, OSX Preview, Skim, etc.). Per default the note itself is kept off-page to prevent it to appear in print-outs.

`PdfPages.attach_note` needs to be called before `savefig()` in order to be added to the correct figure.

Updated fignum_exists to take figure name

Added the ability to check the existence of a figure using its name instead of just the figure number. Example:

```
figure('figure')
fignum_exists('figure') #true
```

5.1.10 ToolManager

Federico Ariza wrote the new *`ToolManager`* that comes as replacement for `NavigationToolbar2`

`ToolManager` offers a new way of looking at the user interactions with the figures. Before we had the `NavigationToolbar2` with its own tools like `zoom/pan/home/save/...` and also we had the shortcuts like `yscale/grid/quit/....` `Toolmanager` relocate all those actions as `Tools` (located in *`backend_tools`*), and defines a way to `access/trigger/reconfigure` them.

The `Toolbars` are replaced for `ToolContainers` that are just GUI interfaces to `trigger` the tools. But don't worry the default backends include a `ToolContainer` called `toolbar`

Note: At the moment, we release this primarily for feedback purposes and should be treated as experimental until further notice as API changes will occur. For the moment the `ToolManager` works only with the `GTK3` and `Tk` backends. Make sure you use one of those. Port for the rest of the backends is comming soon.

To activate the ToolManager include the following at the top of your file

```
>>> matplotlib.rcParams['toolbar'] = 'toolmanager'
```

Interact with the ToolContainer

The most important feature is the ability to easily reconfigure the ToolContainer (aka toolbar). For example, if we want to remove the "forward" button we would just do.

```
>>> fig.canvas.manager.toolmanager.remove_tool('forward')
```

Now if you want to programmatically trigger the "home" button

```
>>> fig.canvas.manager.toolmanager.trigger_tool('home')
```

New Tools for ToolManager

It is possible to add new tools to the ToolManager

A very simple tool that prints "You're awesome" would be:

```
from matplotlib.backend_tools import ToolBase
class AwesomeTool(ToolBase):
    def trigger(self, *args, **kwargs):
        print("You're awesome")
```

To add this tool to ToolManager

```
>>> fig.canvas.manager.toolmanager.add_tool('Awesome', AwesomeTool)
```

If we want to add a shortcut ("d") for the tool

```
>>> fig.canvas.manager.toolmanager.update_keymap('Awesome', 'd')
```

To add it to the toolbar inside the group 'foo'

```
>>> fig.canvas.manager.toolbar.add_tool('Awesome', 'foo')
```

There is a second class of tools, "Toggleable Tools", this are almost the same as our basic tools, just that belong to a group, and are mutually exclusive inside that group. For tools derived from ToolToggleBase there are two basic methods enable and disable that are called automatically whenever it is toggled.

A full example is located in *user_interfaces example code: toolmanager.py*

5.1.11 cbook.is_sequence_of_strings recognizes string objects

This is primarily how pandas stores a sequence of strings

```
import pandas as pd
import matplotlib.cbook as cbook

a = np.array(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(a))  # True

a = np.array(['a', 'b', 'c'], dtype=object)
print(cbook.is_sequence_of_strings(a))  # True

s = pd.Series(['a', 'b', 'c'])
print(cbook.is_sequence_of_strings(s))  # True
```

Previously, the last two prints returned false.

5.1.12 New `close-figs` argument for plot directive

Matplotlib has a sphinx extension `plot_directive` that creates plots for inclusion in sphinx documents. Matplotlib 1.5 adds a new option to the plot directive - `close-figs` - that closes any previous figure windows before creating the plots. This can help avoid some surprising duplicates of plots when using `plot_directive`.

5.1.13 Support for URL string arguments to `imread`

The *imread()* function now accepts URL strings that point to remote PNG files. This circumvents the generation of a HTTPResponse object directly.

5.1.14 Display hook for animations in the IPython notebook

Animation instances gained a `_repr_html_` method to support inline display of animations in the notebook. The method used to display is controlled by the `animation.html` rc parameter, which currently supports values of `none` and `html5`. `none` is the default, performing no display. `html5` converts the animation to an h264 encoded video, which is embedded directly in the notebook.

Users not wishing to use the `_repr_html_` display hook can also manually call the `to_html5_video` method to get the HTML and display using IPython's `HTML` display class:

```
from IPython.display import HTML
HTML(anim.to_html5_video())
```

5.1.15 Prefixed pkg-config for building

Handling of `pkg-config` has been fixed in so far as it is now possible to set it using the environment variable `PKG_CONFIG`. This is important if your toolchain is prefixed. This is done in a simpilar way as setting `CC` or `CXX` before building. An example follows.

> export PKG_CONFIG=x86_64-pc-linux-gnu-pkg-config

5.2 new in matplotlib-1.4

Thomas A. Caswell served as the release manager for the 1.4 release.

5.2.1 New colormap

In heatmaps, a green-to-red spectrum is often used to indicate intensity of activity, but this can be problematic for the red/green colorblind. A new, colorblind-friendly colormap is now available at `matplotlib.cm.Wistia`. This colormap maintains the red/green symbolism while achieving deuteranopic legibility through brightness variations. See here for more information.

5.2.2 The nbagg backend

Phil Elson added a new backend, named "nbagg", which enables interactive figures in a live IPython notebook session. The backend makes use of the infrastructure developed for the webagg backend, which itself gives standalone server backed interactive figures in the browser, however nbagg does not require a dedicated matplotlib server as all communications are handled through the IPython Comm machinery.

As with other backends nbagg can be enabled inside the IPython notebook with:

```
import matplotlib
matplotlib.use('nbagg')
```

Once figures are created and then subsequently shown, they will placed in an interactive widget inside the notebook allowing panning and zooming in the same way as any other matplotlib backend. Because figures require a connection to the IPython notebook server for their interactivity, once the notebook is saved, each figure will be rendered as a static image - thus allowing non-interactive viewing of figures on services such as nbviewer.

5.2.3 New plotting features

Power-law normalization

Ben Gamari added a power-law normalization method, *PowerNorm*. This class maps a range of values to the interval [0,1] with power-law scaling with the exponent provided by the constructor's `gamma` argument. Power law normalization can be useful for, e.g., emphasizing small populations in a histogram.

Fully customizable boxplots

Paul Hobson overhauled the *boxplot()* method such that it is now completely customizable in terms of the styles and positions of the individual artists. Under the hood, *boxplot()* relies on a new function (*boxplot_stats()*), which accepts any data structure currently compatible with *boxplot()*, and returns a list of dictionaries containing the positions for each element of the boxplots. Then a second method, `bxp()` is called to draw the boxplots based on the stats.

The *boxplot()* function can be used as before to generate boxplots from data in one step. But now the user has the flexibility to generate the statistics independently, or to modify the output of *boxplot_stats()* prior to plotting with bxp().

Lastly, each artist (e.g., the box, outliers, cap, notches) can now be toggled on or off and their styles can be passed in through individual kwargs. See the examples: *statistics example code: boxplot_demo.py* and *statistics example code: bxp_demo.py*

Added a bool kwarg, manage_xticks, which if False disables the management of the ticks and limits on the x-axis by *bxp()*.

Support for datetime axes in 2d plots

Andrew Dawson added support for datetime axes to *contour()*, *contourf()*, *pcolormesh()* and *pcolor()*.

Support for additional spectrum types

Todd Jennings added support for new types of frequency spectrum plots: *magnitude_spectrum()*, *phase_spectrum()*, and *angle_spectrum()*, as well as corresponding functions in mlab.

He also added these spectrum types to *specgram()*, as well as adding support for linear scaling there (in addition to the existing dB scaling). Support for additional spectrum types was also added to *specgram()*.

He also increased the performance for all of these functions and plot types.

Support for detrending and windowing 2D arrays in mlab

Todd Jennings added support for 2D arrays in the *detrend_mean()*, *detrend_none()*, and *detrend()*, as well as adding *apply_window()* which support windowing 2D arrays.

Support for strides in mlab

Todd Jennings added some functions to mlab to make it easier to use numpy strides to create memory-efficient 2D arrays. This includes *stride_repeat()*, which repeats an array to create a 2D array, and *stride_windows()*, which uses a moving window to create a 2D array from a 1D array.

Formatter for new-style formatting strings

Added FormatStrFormatterNewStyle which does the same job as FormatStrFormatter, but accepts new-style formatting strings instead of printf-style formatting strings

Consistent grid sizes in streamplots

streamplot() uses a base grid size of 30x30 for both density=1 and density=(1, 1). Previously a grid size of 30x30 was used for density=1, but a grid size of 25x25 was used for density=(1, 1).

Get a list of all tick labels (major and minor)

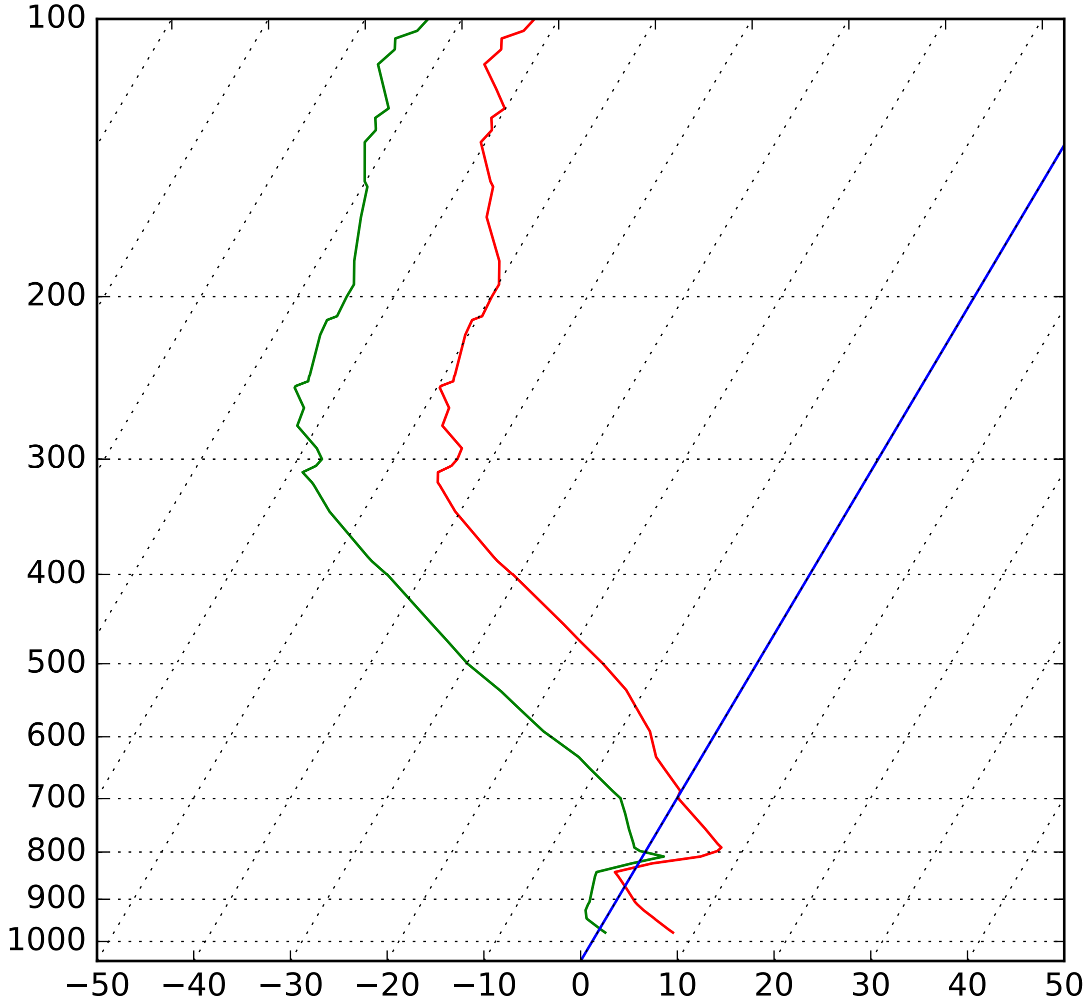
Added the `kwarg` 'which' to `get_xticklabels()`, `get_yticklabels()` and `get_ticklabels()`. 'which' can be 'major', 'minor', or 'both' select which ticks to return, like `set_ticks_position()`. If 'which' is `None` then the old behaviour (controlled by the bool `minor`).

Separate horizontal/vertical axes padding support in ImageGrid

The `kwarg` 'axes_pad' to `mpl_toolkits.axes_grid1.ImageGrid` can now be a tuple if separate horizontal/vertical padding is needed. This is supposed to be very helpful when you have a labelled legend next to every subplot and you need to make some space for legend's labels.

Support for skewed transformations

The *`Affine2D`* gained additional methods `skew` and `skew_deg` to create skewed transformations. Additionally, matplotlib internals were cleaned up to support using such transforms in `Axes`. This transform is important for some plot types, specifically the Skew-T used in meteorology.



Support for specifying properties of wedge and text in pie charts.

Added the `kwargs` 'wedgeprops' and 'textprops' to `pie()` to accept properties for wedge and text objects in a pie. For example, one can specify wedgeprops = {'linewidth':3} to specify the width of the borders of the wedges in the pie. For more properties that the user can specify, look at the docs for the wedge and text objects.

Fixed the direction of errorbar upper/lower limits

Larry Bradley fixed the `errorbar()` method such that the upper and lower limits (*lolims*, *uplims*, *xlolims*, *xuplims*) now point in the correct direction.

More consistent add-object API for Axes

Added the Axes method *add_image* to put image handling on a par with artists, collections, containers, lines, patches, and tables.

Violin Plots

Per Parker, Gregory Kelsie, Adam Ortiz, Kevin Chan, Geoffrey Lee, Deokjae Donald Seo, and Taesu Terry Lim added a basic implementation for violin plots. Violin plots can be used to represent the distribution of sample data. They are similar to box plots, but use a kernel density estimation function to present a smooth approximation of the data sample used. The added features are:

`violin()` - Renders a violin plot from a collection of statistics. *violin_stats()* - Produces a collection of statistics suitable for rendering a violin plot. *violinplot()* - Creates a violin plot from a set of sample data. This method makes use of *violin_stats()* to process the input data, and *violin_stats()* to do the actual rendering. Users are also free to modify or replace the output of *violin_stats()* in order to customize the violin plots to their liking.

This feature was implemented for a software engineering course at the University of Toronto, Scarborough, run in Winter 2014 by Anya Tafliovich.

More `markevery` options to show only a subset of markers

Rohan Walker extended the `markevery` property in *Line2D*. You can now specify a subset of markers to show with an int, slice object, numpy fancy indexing, or float. Using a float shows markers at approximately equal display-coordinate-distances along the line.

Added size related functions to specialized `Collections`

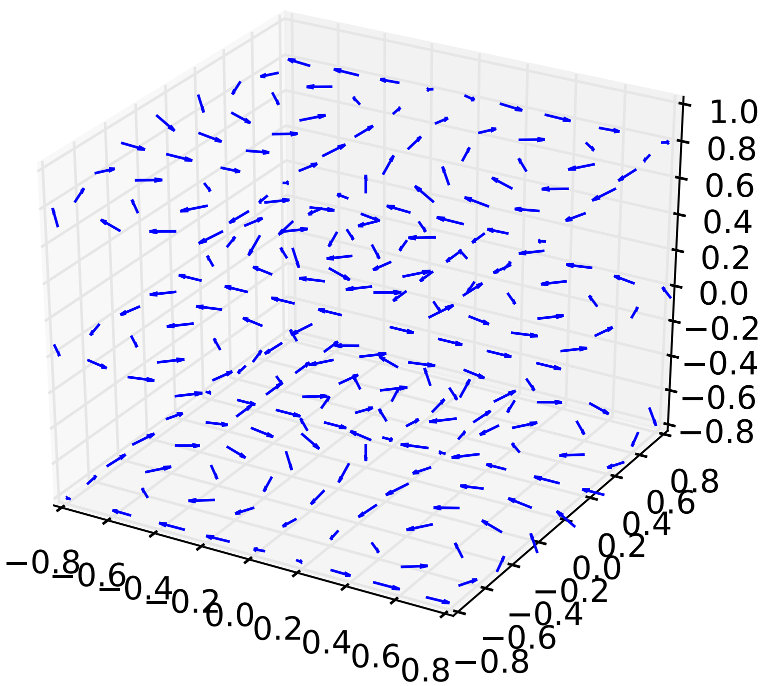
Added the `get_size` and `set_size` functions to control the size of elements of specialized collections (*AsteriskPolygonCollection* *BrokenBarHCollection* *CircleCollection* *PathCollection* *PolyCollection* *RegularPolyCollection* *StarPolygonCollection*).

Fixed the mouse coordinates giving the wrong theta value in Polar graph

Added code to `transform_non_affine()` to ensure that the calculated theta value was between the range of 0 and 2 * pi since the problem was that the value can become negative after applying the direction and rotation to the theta calculation.

Simple quiver plot for mplot3d toolkit

A team of students in an *Engineering Large Software Systems* course, taught by Prof. Anya Tafliovich at the University of Toronto, implemented a simple version of a quiver plot in 3D space for the mplot3d toolkit as one of their term project. This feature is documented in *quiver()*. The team members are: Ryan Steve D'Souza, Victor B, xbtsw, Yang Wang, David, Caradec Bisesar and Vlad Vassilovski.



polar-plot r-tick locations

Added the ability to control the angular position of the r-tick labels on a polar plot via `set_rlabel_position()`.

5.2.4 Date handling

n-d array support for date conversion

Andrew Dawson added support for n-d array handling to *matplotlib.dates.num2date()*, *matplotlib.dates.date2num()* and `matplotlib.dates.datestr2num()`. Support is also added to the unit conversion interfaces `matplotlib.dates.DateConverter` and *matplotlib.units.Registry*.

5.2.5 Configuration (rcParams)

`savefig.transparent` added

Controls whether figures are saved with a transparent background by default. Previously `savefig` always defaulted to a non-transparent background.

`axes.titleweight`

Added rcParam to control the weight of the title

`axes.formatter.useoffset` added

Controls the default value of `useOffset` in `ScalarFormatter`. If `True` and the data range is much smaller than the data average, then an offset will be determined such that the tick labels are meaningful. If `False` then the full number will be formatted in all conditions.

`nbagg.transparent` added

Controls whether nbagg figures have a transparent background. `nbagg.transparent` is `True` by default.

XDG compliance

Matplotlib now looks for configuration files (both rcparams and style) in XDG compliant locations.

5.2.6 `style` package added

You can now easily switch between different styles using the new `style` package:

```
>>> from matplotlib import style
>>> style.use('dark_background')
```

Subsequent plots will use updated colors, sizes, etc. To list all available styles, use:

```
>>> print style.available
```

You can add your own custom `<style name>.mplstyle` files to `~/.matplotlib/stylelib` or call `use` with a URL pointing to a file with `matplotlibrc` settings.

Note that this is an experimental feature, and the interface may change as users test out this new feature.

5.2.7 Backends

Qt5 backend

Martin Fitzpatrick and Tom Badran implemented a Qt5 backend. The differences in namespace locations between Qt4 and Qt5 was dealt with by shimming Qt4 to look like Qt5, thus the Qt5 implementation is the primary implementation. Backwards compatibility for Qt4 is maintained by wrapping the Qt5 implementation.

The Qt5Agg backend currently does not work with IPython's %matplotlib magic.

The 1.4.0 release has a known bug where the toolbar is broken. This can be fixed by:

```
cd path/to/installed/matplotlib
wget https://github.com/matplotlib/matplotlib/pull/3322.diff
# unix2dos 3322.diff (if on windows to fix line endings)
patch -p2 < 3322.diff
```

Qt4 backend

Rudolf Höfler changed the appearance of the subplottool. All sliders are vertically arranged now, buttons for tight layout and reset were added. Furthermore, the subplottool is now implemented as a modal dialog. It was previously a QMainWindow, leaving the SPT open if one closed the plot window.

In the figure options dialog one can now choose to (re-)generate a simple automatic legend. Any explicitly set legend entries will be lost, but changes to the curves' label, linestyle, et cetera will now be updated in the legend.

Interactive performance of the Qt4 backend has been dramatically improved under windows.

The mapping of key-signals from Qt to values matplotlib understands was greatly improved (For both Qt4 and Qt5).

Cairo backends

The Cairo backends are now able to use the cairocffi bindings which are more actively maintained than the pycairo bindings.

Gtk3Agg backend

The Gtk3Agg backend now works on Python 3.x, if the cairocffi bindings are installed.

PDF backend

Added context manager for saving to multi-page PDFs.

5.2.8 Text

Text URLs supported by SVG backend

The `svg` backend will now render *`Text`* objects' url as a link in output SVGs. This allows one to make clickable text in saved figures using the url kwarg of the *`Text`* class.

Anchored sizebar font

Added the `fontproperties` kwarg to `AnchoredSizeBar` to control the font properties.

5.2.9 Sphinx extensions

The `:context:` directive in the *plot_directive* Sphinx extension can now accept an optional `reset` setting, which will cause the context to be reset. This allows more than one distinct context to be present in documentation. To enable this option, use `:context: reset` instead of `:context:` any time you want to reset the context.

5.2.10 Legend and PathEffects documentation

The *Legend guide* and *Path effects guide* have both been updated to better reflect the full potential of each of these powerful features.

5.2.11 Widgets

Span Selector

Added an option `span_stays` to the *SpanSelector* which makes the selector rectangle stay on the axes after you release the mouse.

5.2.12 GAE integration

Matplotlib will now run on google app engine.

5.3 new in matplotlib-1.3

5.3.1 New in 1.3.1

1.3.1 is a bugfix release, primarily dealing with improved setup and handling of dependencies, and correcting and enhancing the documentation.

The following changes were made in 1.3.1 since 1.3.0.

Enhancements

- Added a context manager for creating multi-page pdfs (see *matplotlib.backends.backend_pdf.PdfPages*).
- The WebAgg backend should now have lower latency over heterogeneous Internet connections.

Bug fixes

- Histogram plots now contain the endline.
- Fixes to the Molleweide projection.

- Handling recent fonts from Microsoft and Macintosh-style fonts with non-ascii metadata is improved.
- Hatching of fill between plots now works correctly in the PDF backend.
- Tight bounding box support now works in the PGF backend.
- Transparent figures now display correctly in the Qt4Agg backend.
- Drawing lines from one subplot to another now works.
- Unit handling on masked arrays has been improved.

Setup and dependencies

- Now works with any version of pyparsing 1.5.6 or later, without displaying hundreds of warnings.
- Now works with 64-bit versions of Ghostscript on MS-Windows.
- When installing from source into an environment without Numpy, Numpy will first be downloaded and built and then used to build matplotlib.
- Externally installed backends are now always imported using a fully-qualified path to the module.
- Works with newer version of wxPython.
- Can now build with a PyCXX installed globally on the system from source.
- Better detection of Gtk3 dependencies.

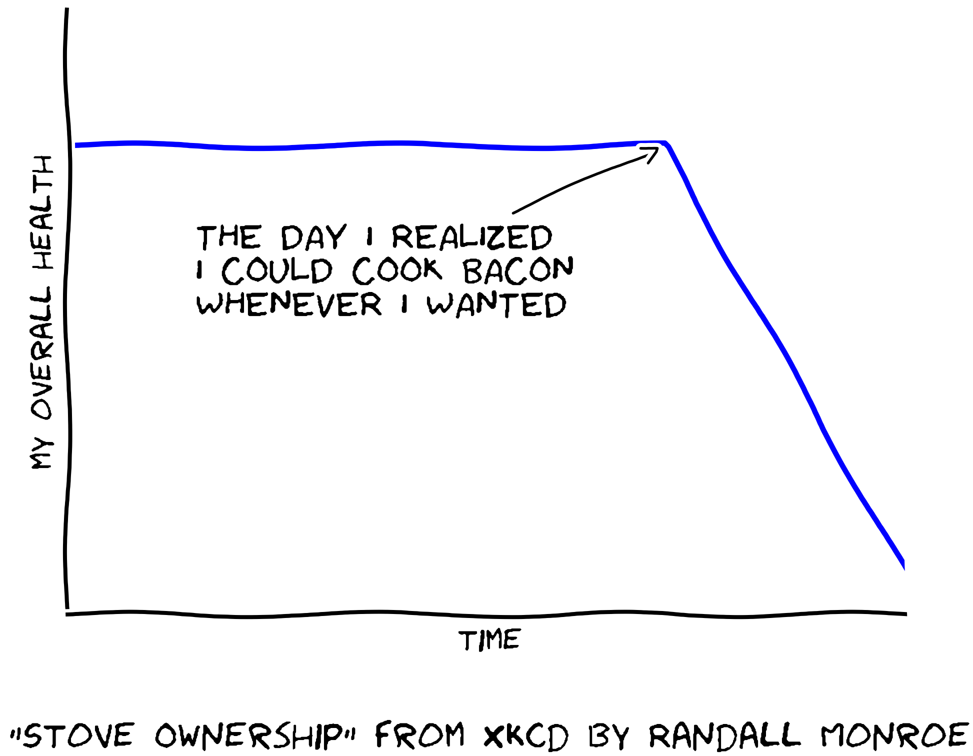
Testing

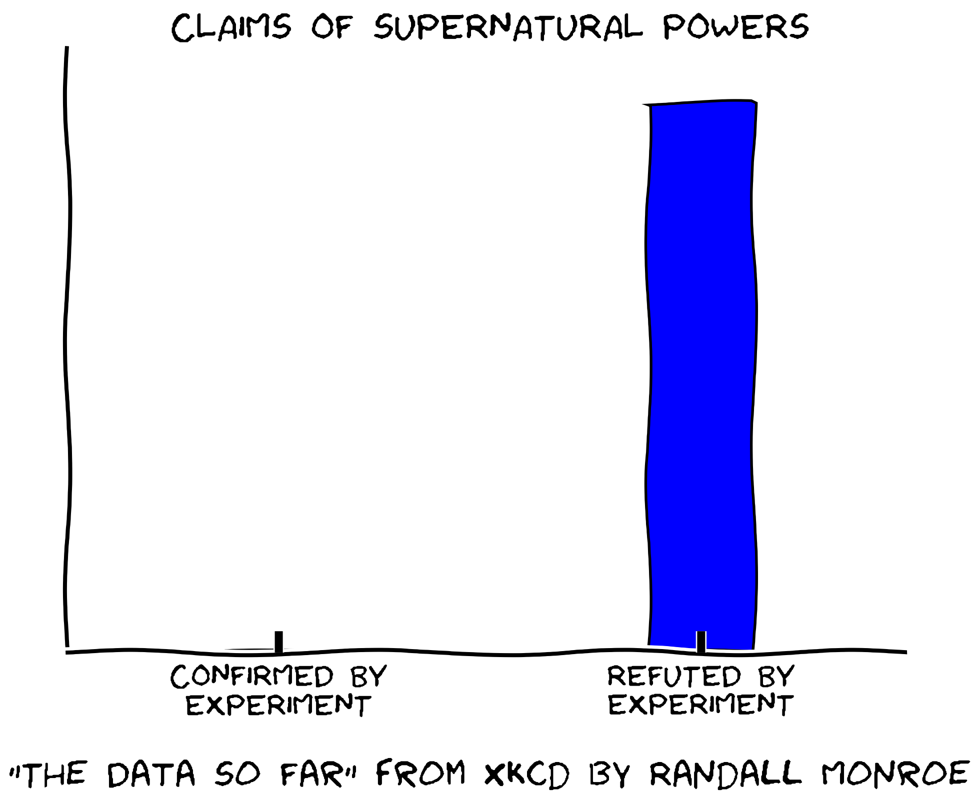
- Tests should now work in non-English locales.
- PEP8 conformance tests now report on locations of issues.

5.3.2 New plotting features

`xkcd`-style sketch plotting

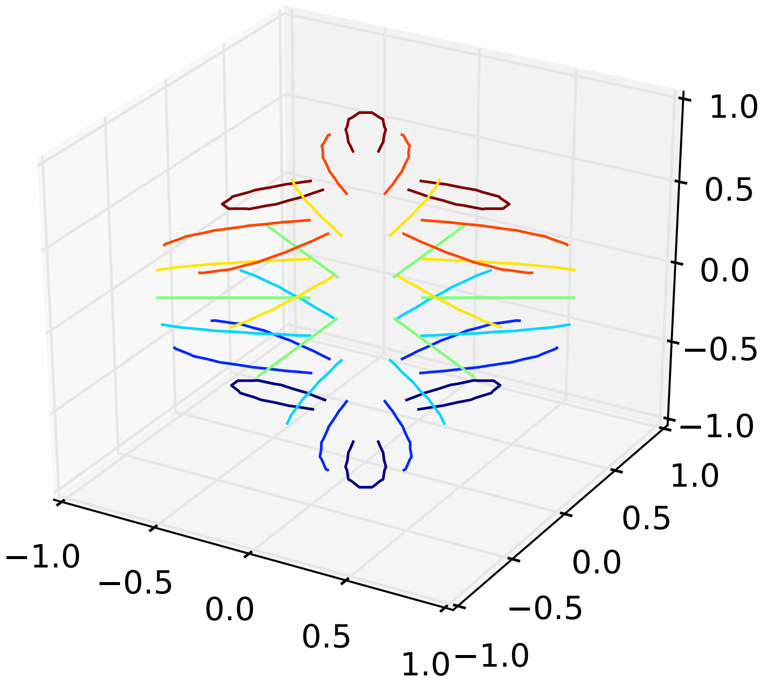
To give your plots a sense of authority that they may be missing, Michael Droettboom (inspired by the work of many others in PR #1329) has added an xkcd-style sketch plotting mode. To use it, simply call *`matplotlib.pyplot.xkcd()`* before creating your plot. For really fine control, it is also possible to modify each artist's sketch parameters individually with *`matplotlib.artist.Artist.set_sketch_params()`*.


THE DAY I REALIZED
I COULD COOK BACON
WHENEVER I WANTED
MY OVERALL HEALTH
TIME
"STOVE OWNERSHIP" FROM XKCD BY RANDALL MONROE



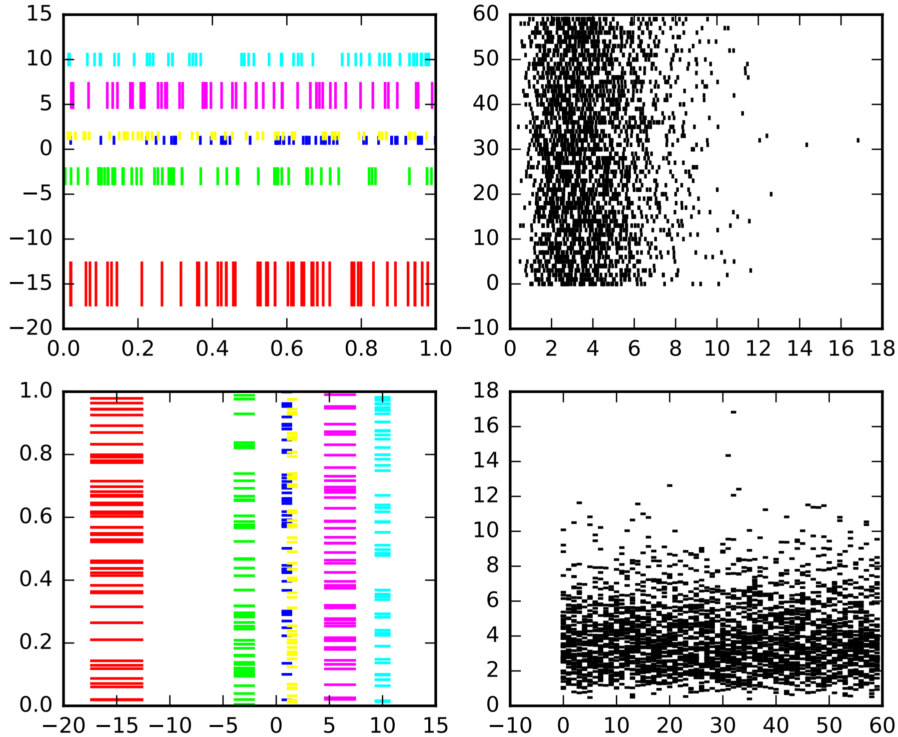
5.3.3 Updated Axes3D.contour methods

Damon McDougall updated the *tricontour()* and *tricontourf()* methods to allow 3D contour plots on abitrary unstructured user-specified triangulations.



New eventplot plot type

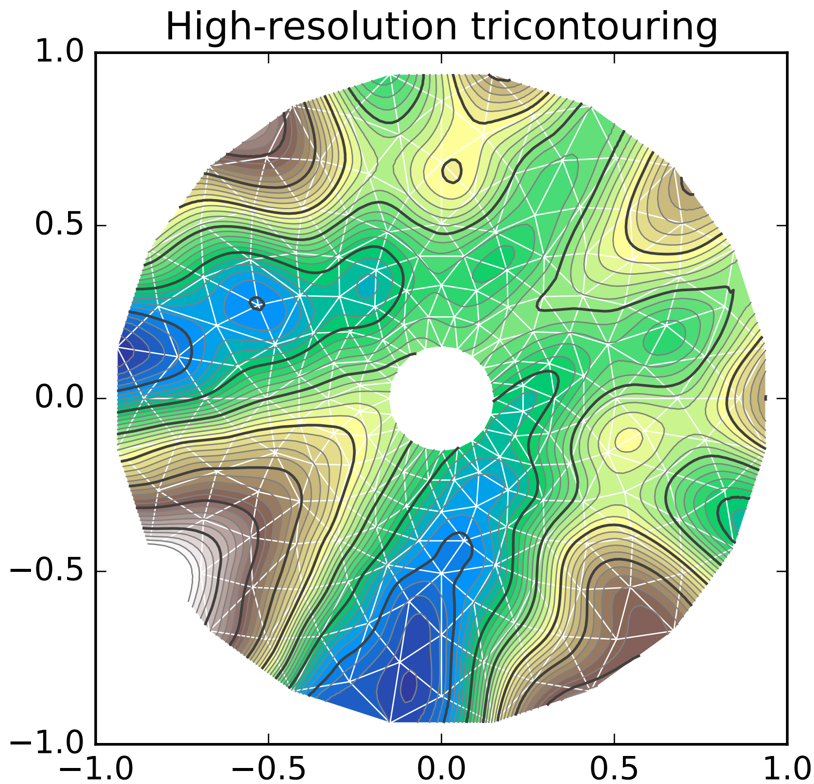
Todd Jennings added a *eventplot()* function to create multiple rows or columns of identical line segments



As part of this feature, there is a new `EventCollection` class that allows for plotting and manipulating rows or columns of identical line segments.

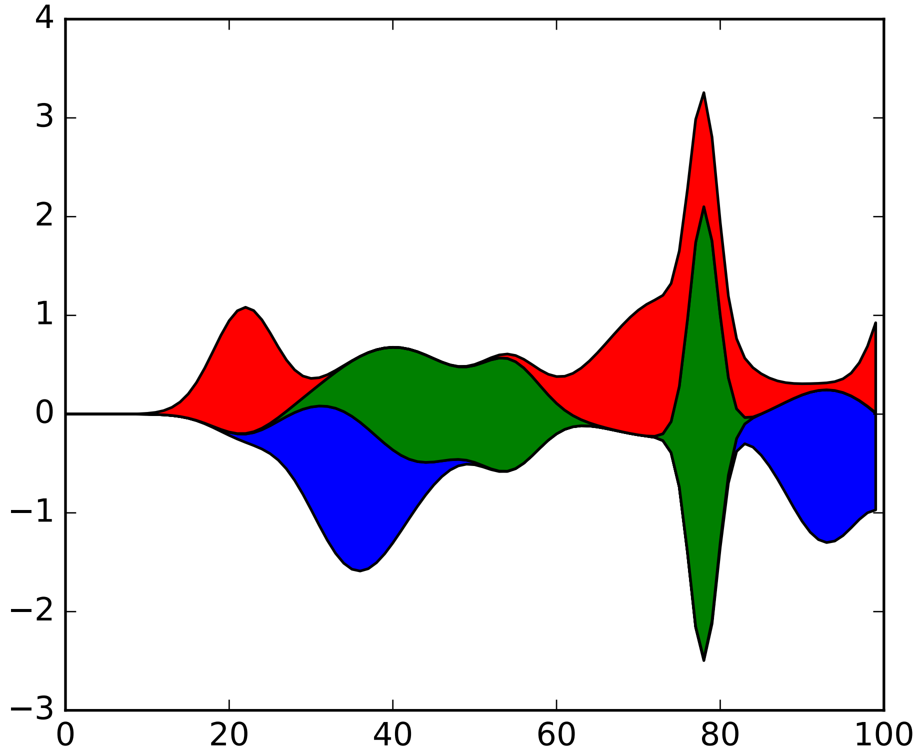
Triangular grid interpolation

Geoffroy Billotey and Ian Thomas added classes to perform interpolation within triangular grids: (`LinearTriInterpolator` and `CubicTriInterpolator`) and a utility class to find the triangles in which points lie (`TrapezoidMapTriFinder`). A helper class to perform mesh refinement and smooth contouring was also added (`UniformTriRefiner`). Finally, a class implementing some basic tools for triangular mesh improvement was added (`TriAnalyzer`).



Baselines for stackplot

Till Stensitzki added non-zero baselines to `stackplot()`. They may be symmetric or weighted.



Rectangular colorbar extensions

Andrew Dawson added a new keyword argument *extendrect* to `colorbar()` to optionally make colorbar extensions rectangular instead of triangular.

More robust boxplots

Paul Hobson provided a fix to the `boxplot()` method that prevent whiskers from being drawn inside the box for oddly distributed data sets.

Calling subplot() without arguments

A call to `subplot()` without any arguments now acts the same as `subplot(111)` or `subplot(1,1,1)` – it creates one axes for the whole figure. This was already the behavior for both `axes()` and `subplots()`, and now this consistency is shared with `subplot()`.

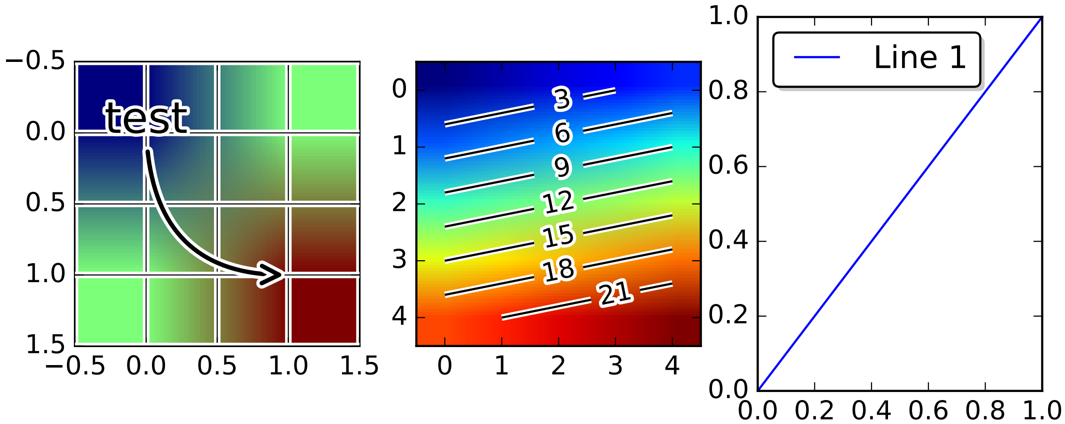
5.3.4 Drawing

Independent alpha values for face and edge colors

Wes Campaigne modified how *Patch* objects are drawn such that (for backends supporting transparency) you can set different alpha values for faces and edges, by specifying their colors in RGBA format. Note that if you set the alpha attribute for the patch object (e.g. using *set_alpha()* or the `alpha` keyword argument), that value will override the alpha components set in both the face and edge colors.

Path effects on lines

Thanks to Jae-Joon Lee, path effects now also work on plot lines.



Easier creation of colormap and normalizer for levels with colors

Phil Elson added the *matplotlib.colors.from_levels_and_colors()* function to easily create a colormap and normalizer for representation of discrete colors for plot types such as *matplotlib.pyplot.pcolormesh()*, with a similar interface to that of `contourf()`.

Full control of the background color

Wes Campaigne and Phil Elson fixed the Agg backend such that PNGs are now saved with the correct background color when `fig.patch.get_alpha()` is not 1.

Improved `bbox_inches="tight"` functionality

Passing `bbox_inches="tight"` through to `plt.save()` now takes into account *all* artists on a figure - this was previously not the case and led to several corner cases which did not function as expected.

Initialize a rotated rectangle

Damon McDougall extended the *Rectangle* constructor to accept an `angle` kwarg, specifying the rotation of a rectangle in degrees.

5.3.5 Text

Anchored text support

The svg and pgf backends are now able to save text alignment information to their output formats. This allows to edit text elements in saved figures, using Inkscape for example, while preserving their intended position. For svg please note that you'll have to disable the default text-to-path conversion (`mpl.rc('svg', fonttype='none')`).

Better vertical text alignment and multi-line text

The vertical alignment of text is now consistent across backends. You may see small differences in text placement, particularly with rotated text.

If you are using a custom backend, note that the `draw_text` renderer method is now passed the location of the baseline, not the location of the bottom of the text bounding box.

Multi-line text will now leave enough room for the height of very tall or very low text, such as superscripts and subscripts.

Left and right side axes titles

Andrew Dawson added the ability to add axes titles flush with the left and right sides of the top of the axes using a new keyword argument `loc` to *title()*.

Improved manual contour plot label positioning

Brian Mattern modified the manual contour plot label positioning code to interpolate along line segments and find the actual closest point on a contour to the requested position. Previously, the closest path vertex was used, which, in the case of straight contours was sometimes quite distant from the requested location. Much more precise label positioning is now possible.

5.3.6 Configuration (rcParams)

Quickly find rcParams

Phil Elson made it easier to search for rcParameters by passing a valid regular expression to `matplotlib.RcParams.find_all()`. *matplotlib.RcParams* now also has a pretty repr and str representation so that search results are printed prettily:

```
>>> import matplotlib
>>> print(matplotlib.rcParams.find_all('\.size'))
RcParams({'font.size': 12,
          'xtick.major.size': 4,
          'xtick.minor.size': 2,
          'ytick.major.size': 4,
          'ytick.minor.size': 2})
```

`axes.xmargin` and `axes.ymargin` added to rcParams

`rcParam` values (`axes.xmargin` and `axes.ymargin`) were added to configure the default margins used. Previously they were hard-coded to default to 0, default value of both rcParam values is 0.

Changes to font rcParams

The `font.*` rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.

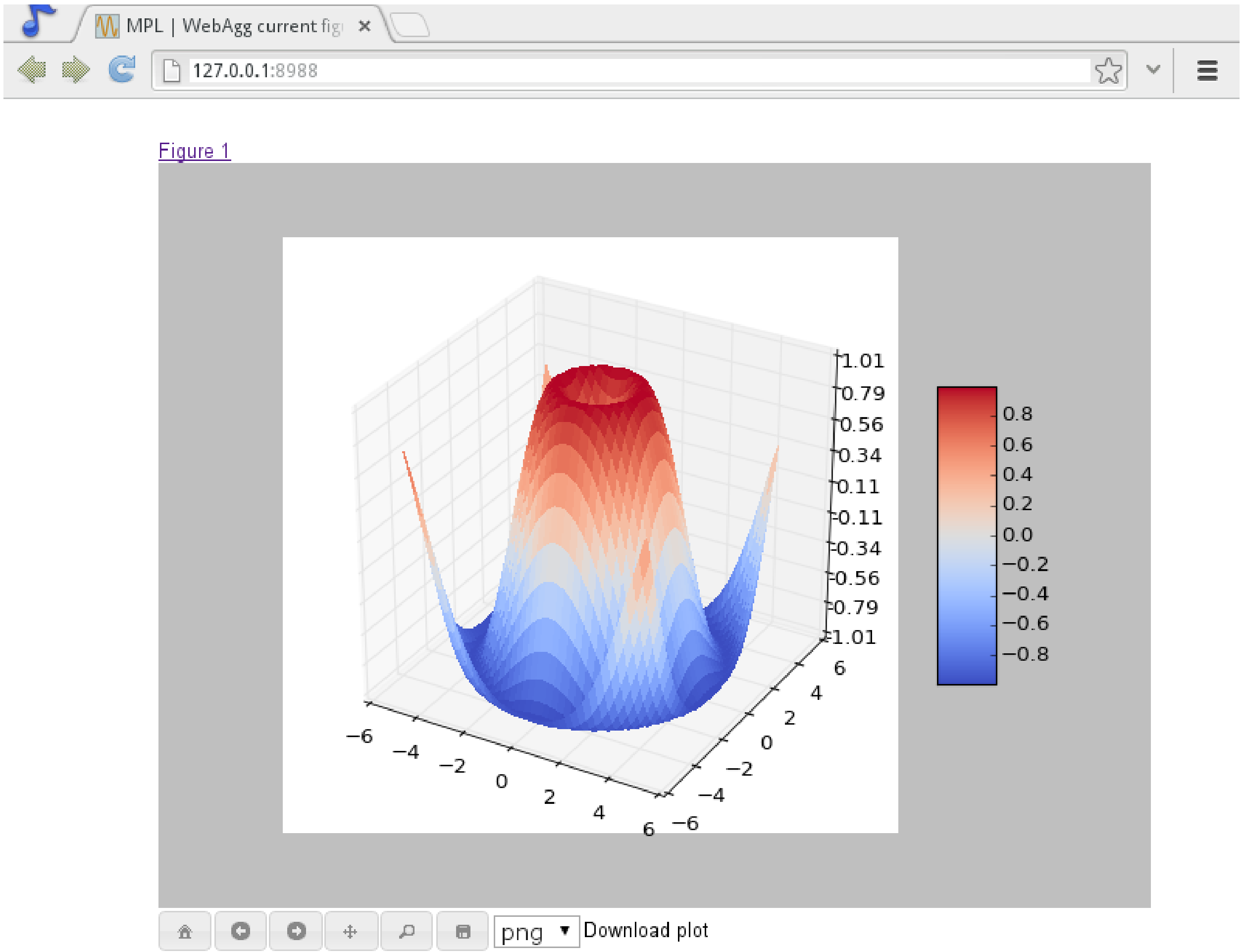
`savefig.jpeg_quality` added to rcParams

rcParam value `savefig.jpeg_quality` was added so that the user can configure the default quality used when a figure is written as a JPEG. The default quality is 95; previously, the default quality was 75. This change minimizes the artifacting inherent in JPEG images, particularly with images that have sharp changes in color as plots often do.

5.3.7 Backends

WebAgg backend

Michael Droettboom, Phil Elson and others have developed a new backend, WebAgg, to display figures in a web browser. It works with animations as well as being fully interactive.



Future versions of matplotlib will integrate this backend with the IPython notebook for a fully web browser based plotting frontend.

Remember save directory

Martin Spacek made the save figure dialog remember the last directory saved to. The default is configurable with the new `savefig.directory` rcParam in `matplotlibrc`.

5.3.8 Documentation and examples

Numpydoc docstrings

Nelle Varoquaux has started an ongoing project to convert matplotlib's docstrings to numpydoc format. See MEP10 for more information.

Example reorganization

Tony Yu has begun work reorganizing the examples into more meaningful categories. The new gallery page is the fruit of this ongoing work. See MEP12 for more information.

Examples now use subplots()

For the sake of brevity and clarity, most of the *examples* now use the newer `subplots()`, which creates a figure and one (or multiple) axes object(s) in one call. The old way involved a call to `figure()`, followed by one (or multiple) `subplot()` calls.

5.3.9 Infrastructure

Housecleaning

A number of features that were deprecated in 1.2 or earlier, or have not been in a working state for a long time have been removed. Highlights include removing the Qt version 3 backends, and the FltkAgg and Emf backends. See *Changes in 1.3.x* for a complete list.

New setup script

matplotlib 1.3 includes an entirely rewritten setup script. We now ship fewer dependencies with the tarballs and installers themselves. Notably, `pytz`, `dateutil`, `pyparsing` and `six` are no longer included with matplotlib. You can either install them manually first, or let `pip` install them as dependencies along with matplotlib. It is now possible to not include certain subcomponents, such as the unit test data, in the install. See `setup.cfg.template` for more information.

XDG base directory support

On Linux, matplotlib now uses the `XDG base directory specification` to find the `matplotlibrc` configuration file. `matplotlibrc` should now be kept in `config/matplotlib`, rather than `matplotlib`. If your configuration is found in the old location, it will still be used, but a warning will be displayed.

Catch opening too many figures using pyplot

Figures created through `pyplot.figure` are retained until they are explicitly closed. It is therefore common for new users of matplotlib to run out of memory when creating a large series of figures in a loop without closing them.

matplotlib will now display a `RuntimeWarning` when too many figures have been opened at once. By default, this is displayed for 20 or more figures, but the exact number may be controlled using the `figure.max_open_warning` rcParam.

5.4 new in matplotlib 1.2.2

5.4.1 Improved collections

The individual items of a collection may now have different alpha values and be rendered correctly. This also fixes a bug where collections were always filled in the PDF backend.

5.4.2 Multiple images on same axes are correctly transparent

When putting multiple images onto the same axes, the background color of the axes will now show through correctly.

5.5 new in matplotlib-1.2

5.5.1 Python 3.x support

Matplotlib 1.2 is the first version to support Python 3.x, specifically Python 3.1 and 3.2. To make this happen in a reasonable way, we also had to drop support for Python versions earlier than 2.6.

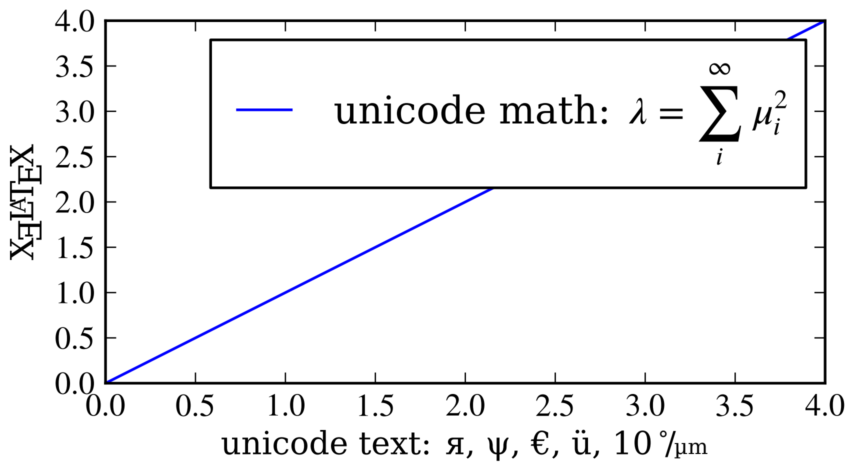
This work was done by Michael Droettboom, the Cape Town Python Users' Group, many others and supported financially in part by the SAGE project.

The following GUI backends work under Python 3.x: Gtk3Cairo, Qt4Agg, TkAgg and MacOSX. The other GUI backends do not yet have adequate bindings for Python 3.x, but continue to work on Python 2.6 and 2.7, particularly the Qt and QtAgg backends (which have been deprecated). The non-GUI backends, such as PDF, PS and SVG, work on both Python 2.x and 3.x.

Features that depend on the Python Imaging Library, such as JPEG handling, do not work, since the version of PIL for Python 3.x is not sufficiently mature.

5.5.2 PGF/TikZ backend

Peter Würtz wrote a backend that allows matplotlib to export figures as drawing commands for LaTeX. These can be processed by PdfLaTeX, XeLaTeX or LuaLaTeX using the PGF/TikZ package. Usage examples and documentation are found in *Typesetting With XeLaTeX/LuaLaTeX*.



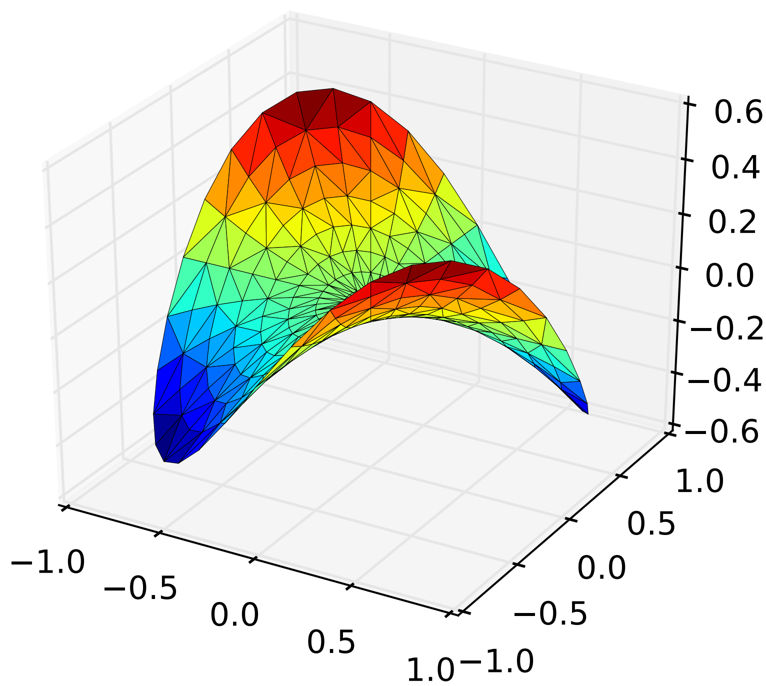
5.5.3 Locator interface

Philip Elson exposed the intelligence behind the tick Locator classes with a simple interface. For instance, to get no more than 5 sensible steps which span the values 10 and 19.5:

```
>>> import matplotlib.ticker as mticker
>>> locator = mticker.MaxNLocator(nbins=5)
>>> print(locator.tick_values(10, 19.5))
[ 10.  12.  14.  16.  18.  20.]
```

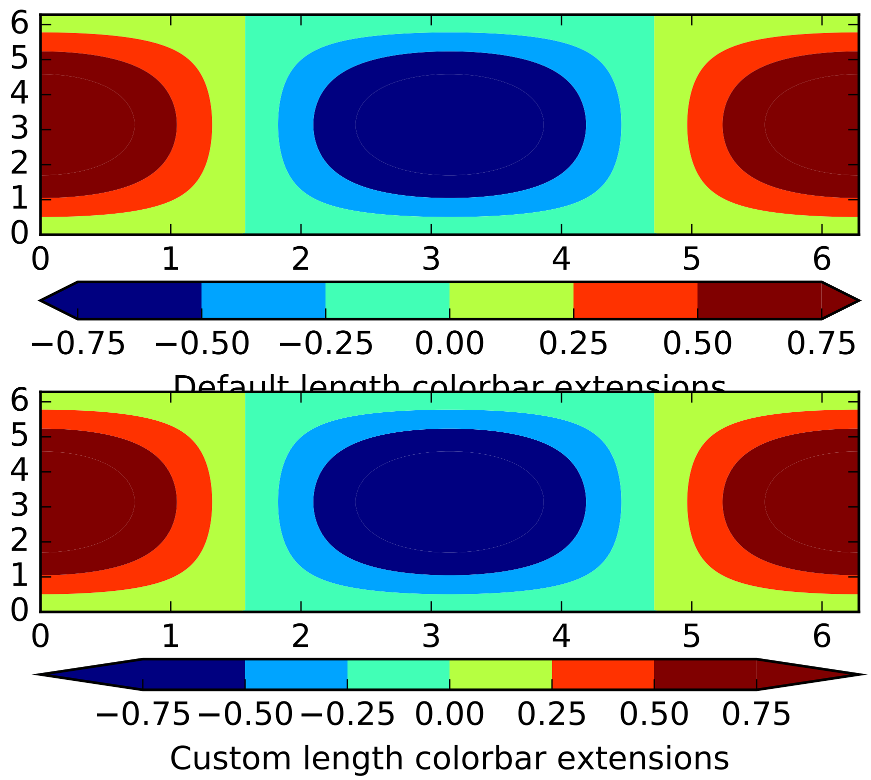
5.5.4 Tri-Surface Plots

Damon McDougall added a new plotting method for the `mplot3d` toolkit called *plot_trisurf()*.



5.5.5 Control the lengths of colorbar extensions

Andrew Dawson added a new keyword argument *extendfrac* to `colorbar()` to control the length of minimum and maximum colorbar extensions.



5.5.6 Figures are picklable

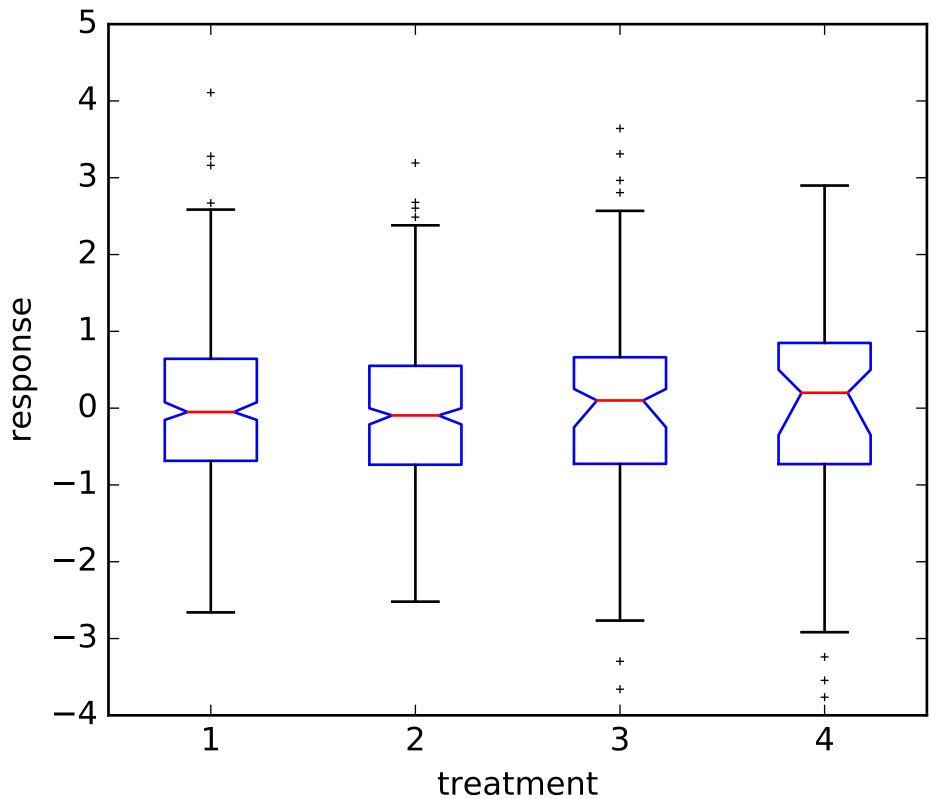
Philip Elson added an experimental feature to make figures picklable for quick and easy short-term storage of plots. Pickle files are not designed for long term storage, are unsupported when restoring a pickle saved in another matplotlib version and are insecure when restoring a pickle from an untrusted source. Having said this, they are useful for short term storage for later modification inside matplotlib.

5.5.7 Set default bounding box in matplotlibrc

Two new defaults are available in the matplotlibrc configuration file: `savefig.bbox`, which can be set to 'standard' or 'tight', and `savefig.pad_inches`, which controls the bounding box padding.

5.5.8 New Boxplot Functionality

Users can now incorporate their own methods for computing the median and its confidence intervals into the `boxplot()` method. For every column of data passed to boxplot, the user can specify an accompanying median and confidence interval.



5.5.9 New RC parameter functionality

Matthew Emmett added a function and a context manager to help manage RC parameters: `rc_file()` and `rc_context`. To load RC parameters from a file:

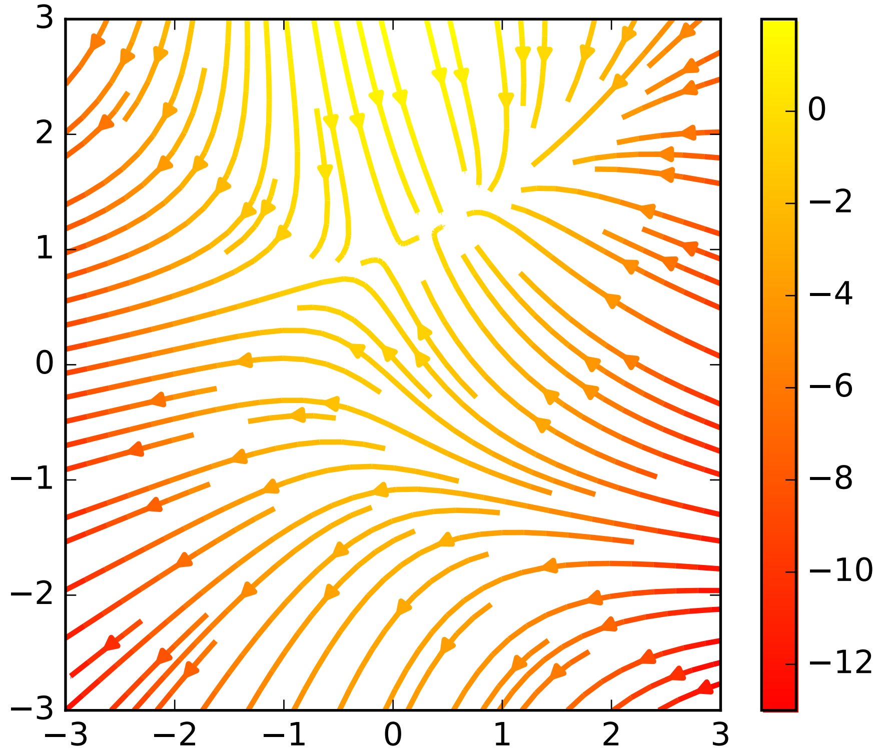
```
>>> mpl.rc_file('mpl.rc')
```

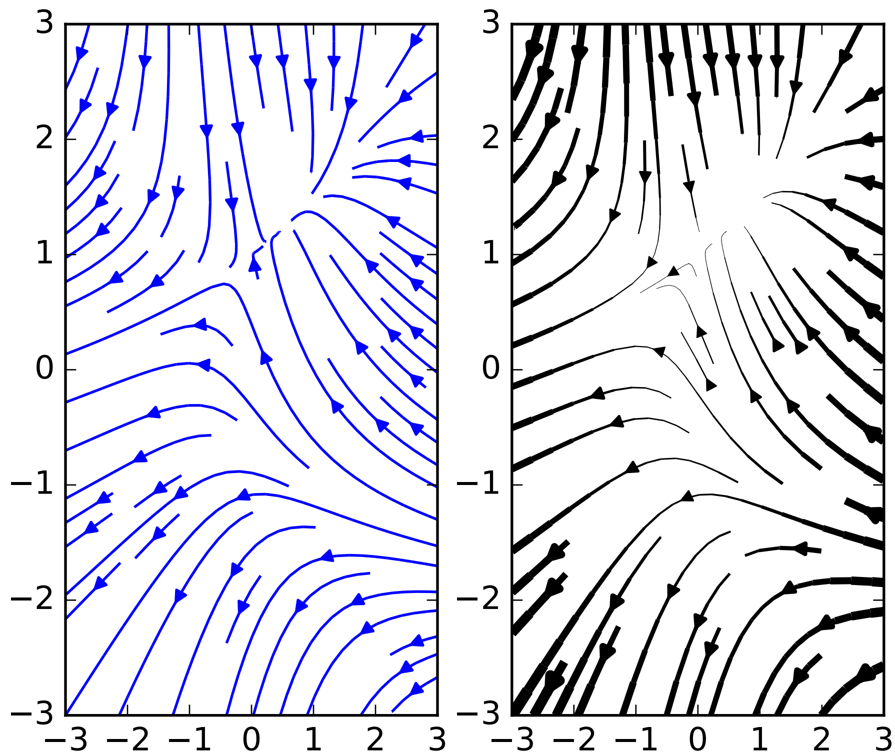
To temporarily use RC parameters:

```
>>> with mpl.rc_context(fname='mpl.rc', rc={'text.usetex': True}):
>>>     ...
```

5.5.10 Streamplot

Tom Flannaghan and Tony Yu have added a new `streamplot()` function to plot the streamlines of a vector field. This has been a long-requested feature and complements the existing `quiver()` function for plotting vector fields. In addition to simply plotting the streamlines of the vector field, `streamplot()` allows users to map the colors and/or line widths of the streamlines to a separate parameter, such as the speed or local intensity of the vector field.


3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
0
−2
−4
−6
−8
−10
−12



5.5.11 New hist functionality

Nic Eggert added a new `stacked` kwarg to *hist()* that allows creation of stacked histograms using any of the histogram types. Previously, this functionality was only available by using the `barstacked` histogram type. Now, when `stacked=True` is passed to the function, any of the histogram types can be stacked. The `barstacked` histogram type retains its previous functionality for backwards compatibility.

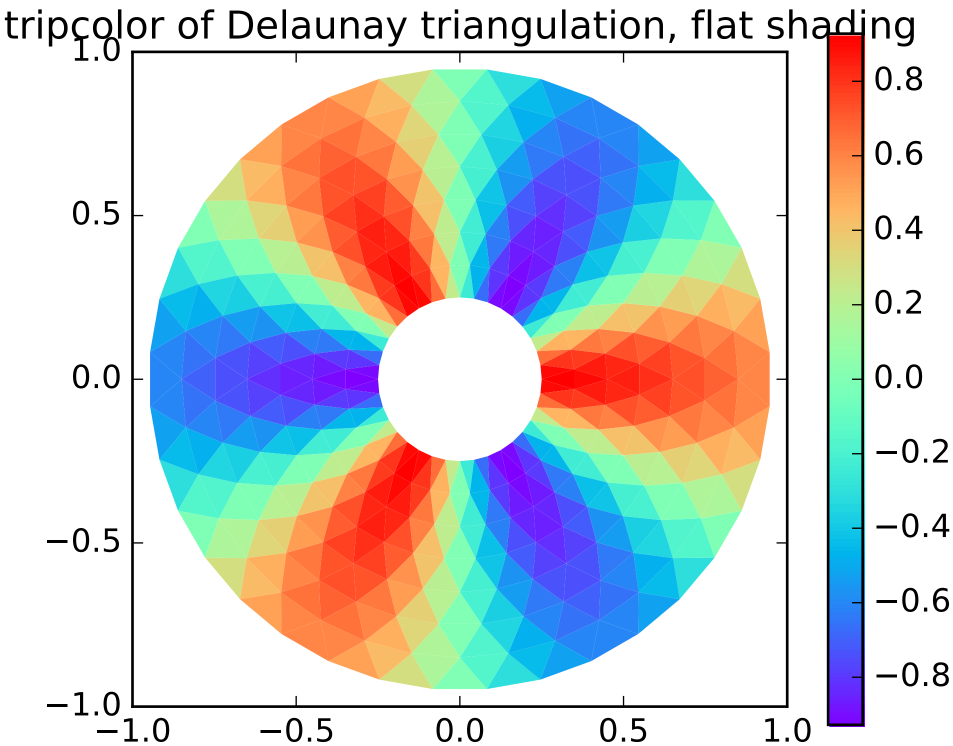
5.5.12 Updated shipped dependencies

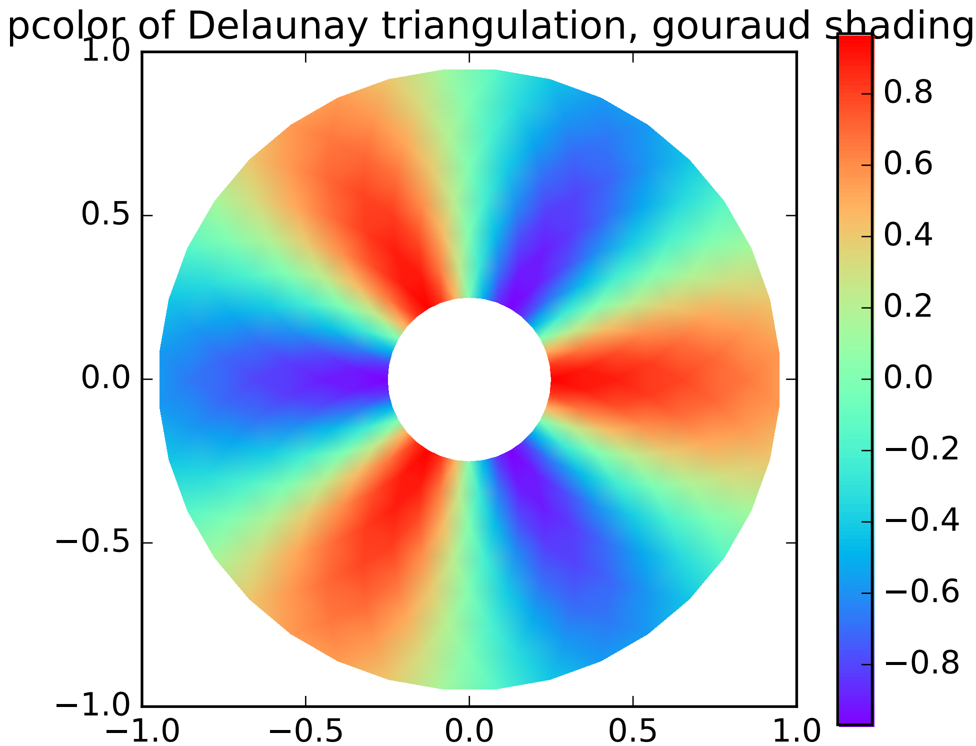
The following dependencies that ship with matplotlib and are optionally installed alongside it have been updated:

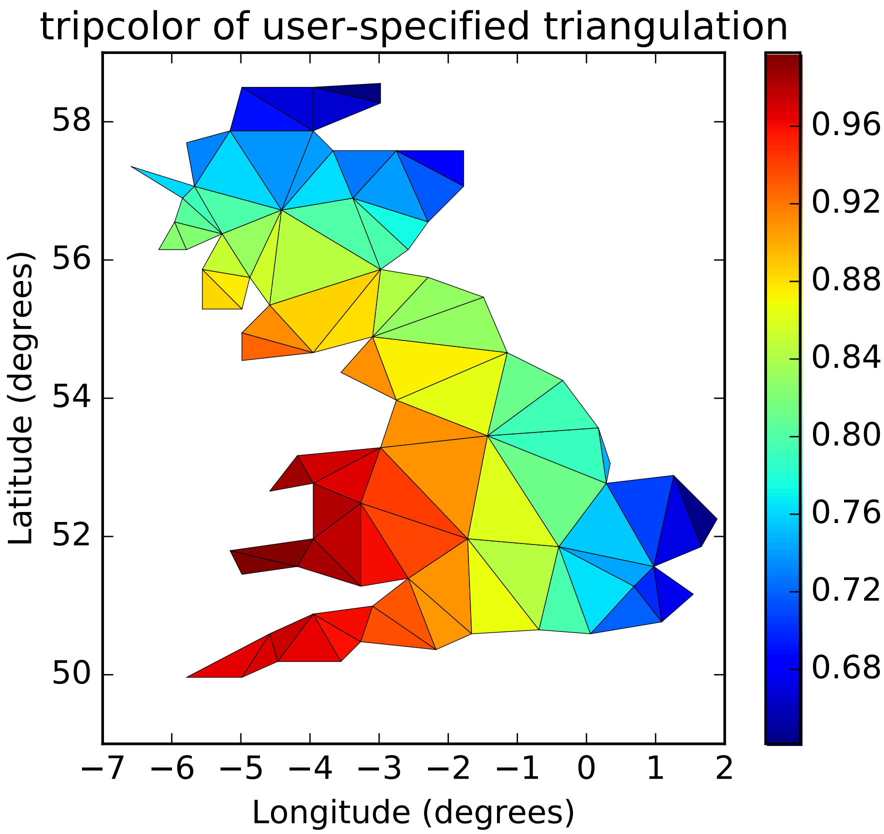
- `pytz` 2012d
- `dateutil` 1.5 on Python 2.x, and 2.1 on Python 3.x

5.5.13 Face-centred colors in tripcolor plots

Ian Thomas extended *tripcolor()* to allow one color value to be specified for each triangular face rather than for each point in a triangulation.

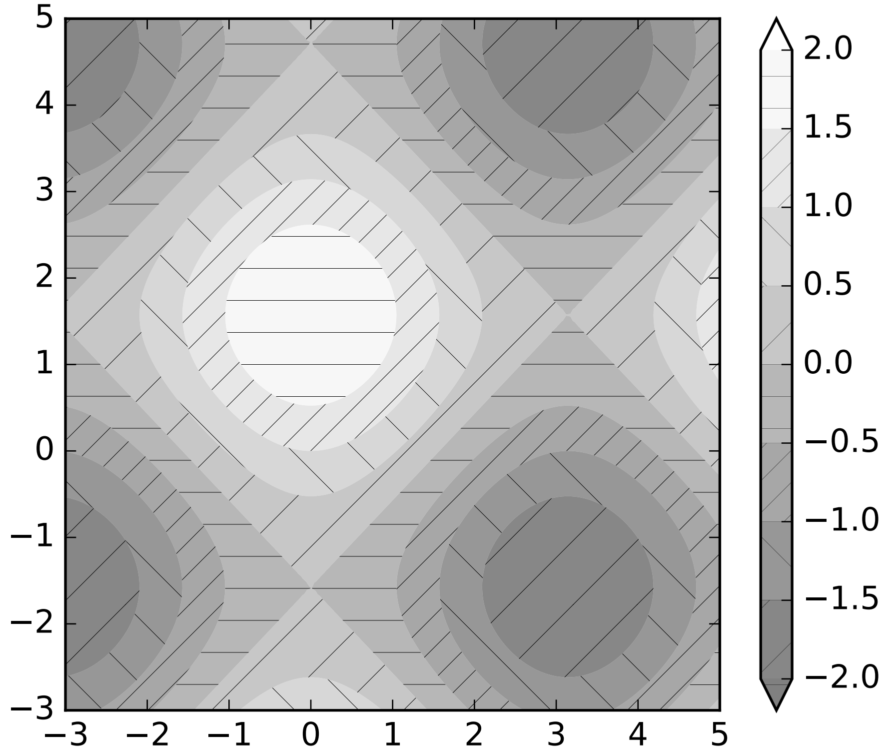

tripcolor of Delaunay triangulation, flat shading
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8

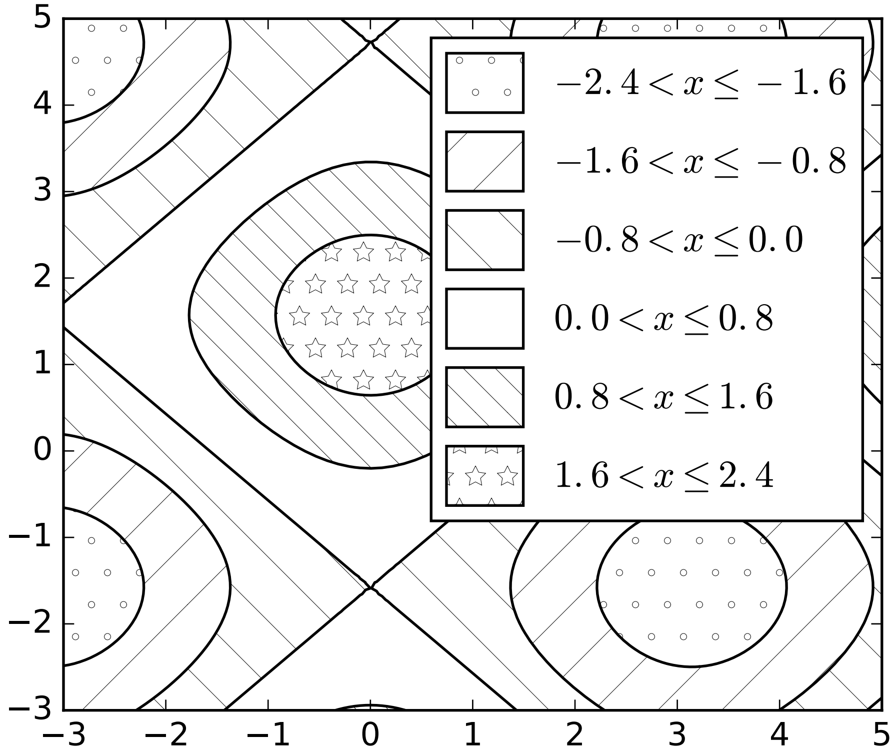

pcolor of Delaunay triangulation, gouraud shading
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8



5.5.14 Hatching patterns in filled contour plots, with legends

Phil Elson added support for hatching to `contourf()`, together with the ability to use a legend to identify contoured ranges.


5
4
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
4
5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0



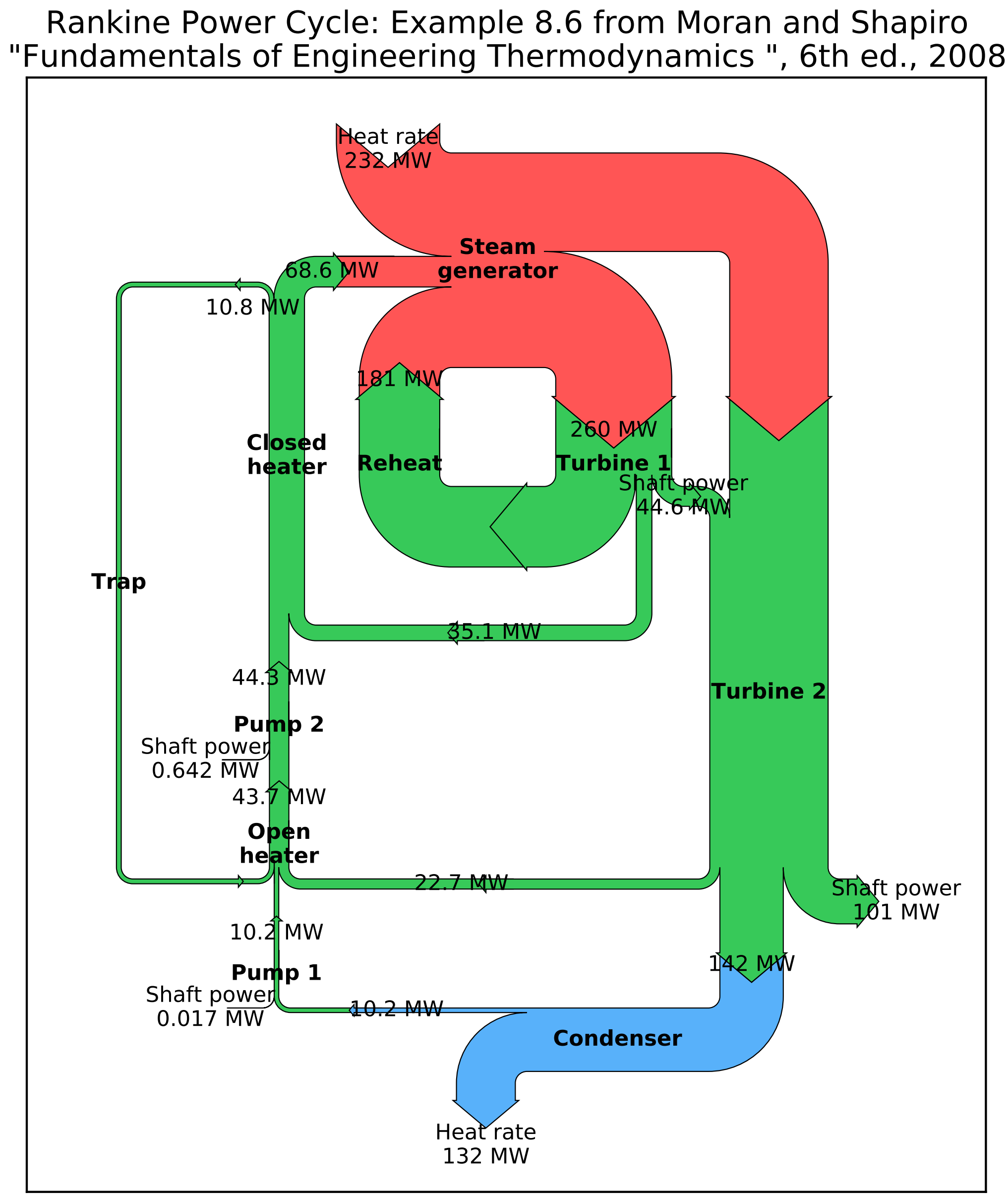
5.5.15 Known issues in the matplotlib-1.2 release

- When using the Qt4Agg backend with IPython 0.11 or later, the save dialog will not display. This should be fixed in a future version of IPython.

5.6 new in matplotlib-1.1

5.6.1 Sankey Diagrams

Kevin Davies has extended Yannick Copin's original Sankey example into a module (`sankey`) and provided new examples (*api example code: sankey_demo_basics.py*, *api example code: sankey_demo_links.py*, *api example code: sankey_demo_rankine.py*).



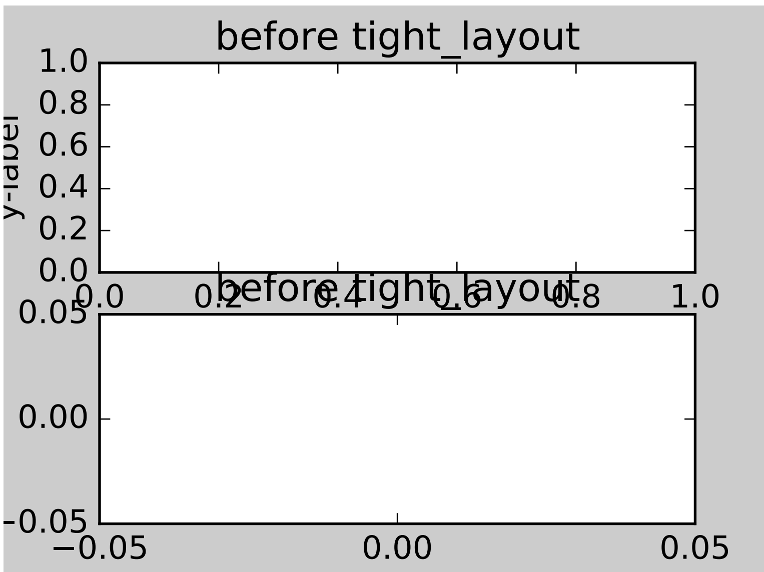
5.6.2 Animation

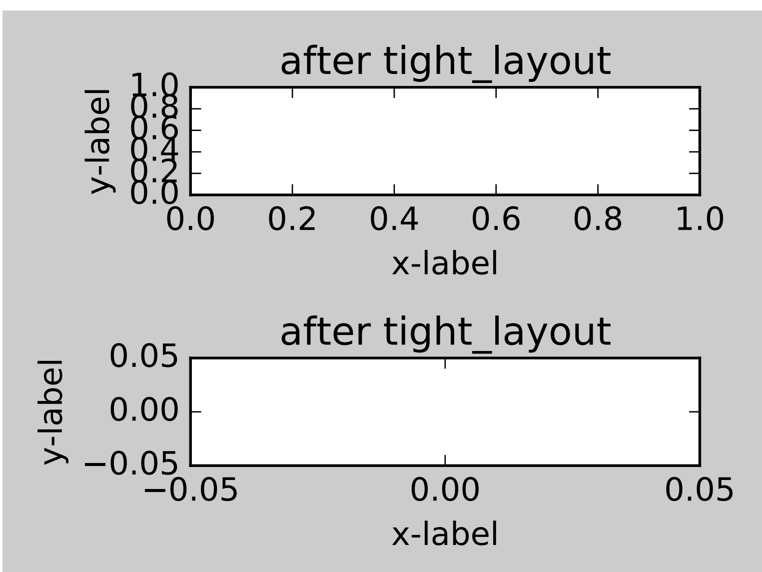
Ryan May has written a backend-independent framework for creating animated figures. The `animation` module is intended to replace the backend-specific examples formerly in the *Matplotlib Examples* listings. Examples using the new framework are in *animation Examples*; see the entrancing *double pendulum* which uses `matplotlib.animation.Animation.save()` to create the movie below.

This should be considered as a beta release of the framework; please try it and provide feedback.

5.6.3 Tight Layout

A frequent issue raised by users of matplotlib is the lack of a layout engine to nicely space out elements of the plots. While matplotlib still adheres to the philosophy of giving users complete control over the placement of plot elements, Jae-Joon Lee created the *tight_layout* module and introduced a new command *tight_layout()* to address the most common layout issues.





The usage of this functionality can be as simple as

```
plt.tight_layout()
```

and it will adjust the spacing between subplots so that the axis labels do not overlap with neighboring subplots. A *Tight Layout guide* has been created to show how to use this new tool.

5.6.4 PyQT4, PySide, and IPython

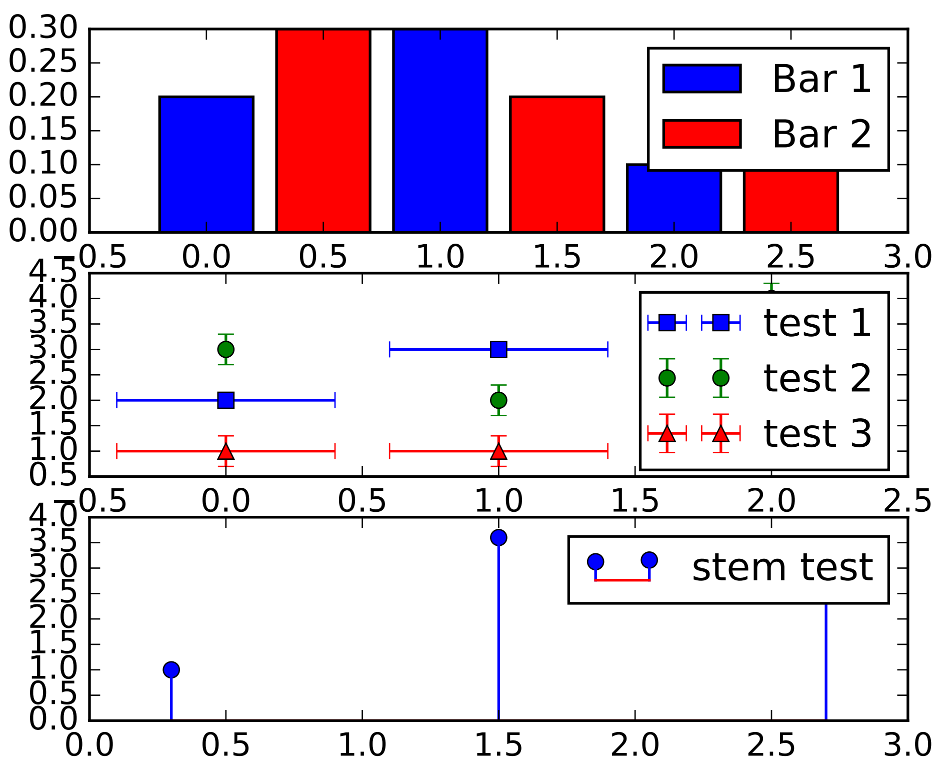
Gerald Storer made the Qt4 backend compatible with PySide as well as PyQT4. At present, however, PySide does not support the PyOS_InputHook mechanism for handling gui events while waiting for text input, so it cannot be used with the new version 0.11 of IPython. Until this feature appears in PySide, IPython users should use the PyQT4 wrapper for QT4, which remains the matplotlib default.

An rcParam entry, "backend.qt4", has been added to allow users to select PyQt4, PyQt4v2, or PySide. The latter two use the Version 2 Qt API. In most cases, users can ignore this rcParam variable; it is available to aid in testing, and to provide control for users who are embedding matplotlib in a PyQt4 or PySide app.

5.6.5 Legend

Jae-Joon Lee has improved plot legends. First, legends for complex plots such as `stem()` plots will now display correctly. Second, the 'best' placement of a legend has been improved in the presence of NANs.

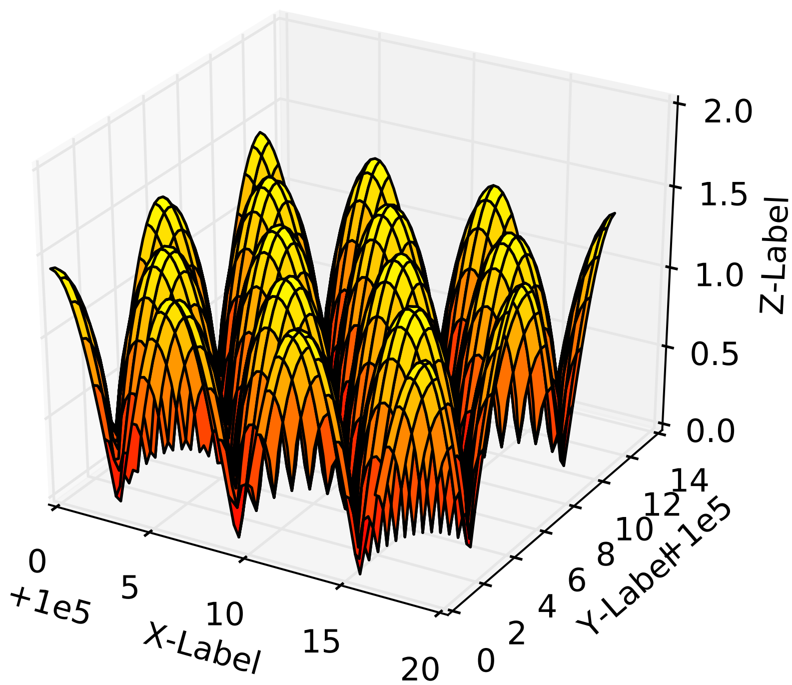
See the *Legend guide* for more detailed explanation and examples.



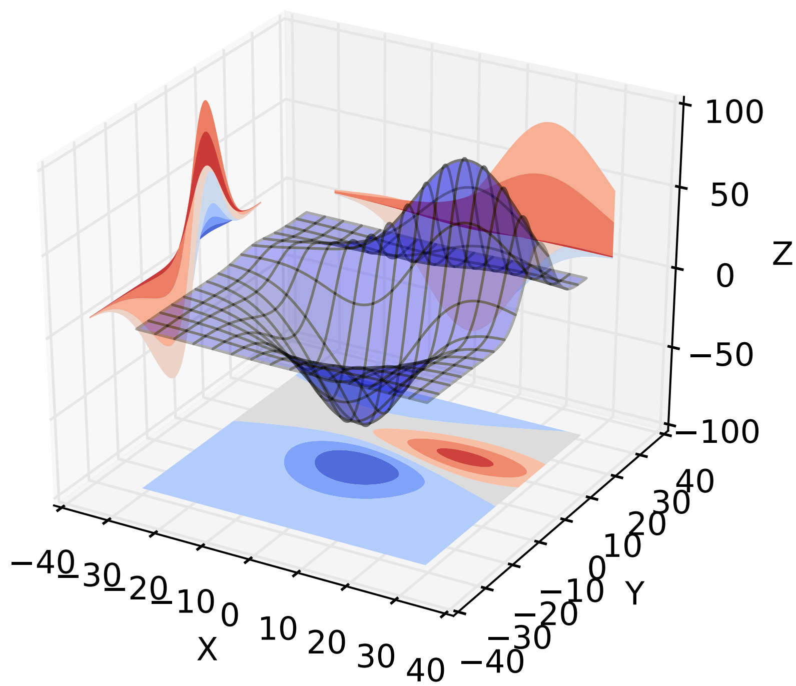
5.6.6 mplot3d

In continuing the efforts to make 3D plotting in matplotlib just as easy as 2D plotting, Ben Root has made several improvements to the `mplot3d` module.

- `Axes3D` has been improved to bring the class towards feature-parity with regular Axes objects
- Documentation for *mplot3d* was significantly expanded
- Axis labels and orientation improved
- Most 3D plotting functions now support empty inputs
- Ticker offset display added:



- `contourf()` gains *zdir* and *offset* kwargs. You can now do this:



5.6.7 Numerix support removed

After more than two years of deprecation warnings, Numerix support has now been completely removed from matplotlib.

5.6.8 Markers

The list of available markers for *plot()* and *scatter()* has now been merged. While they were mostly similar, some markers existed for one function, but not the other. This merge did result in a conflict for the 'd' diamond marker. Now, 'd' will be interpreted to always mean "thin" diamond while 'D' will mean "regular" diamond.

Thanks to Michael Droettboom for this effort.

5.6.9 Other improvements

- Unit support for polar axes and *arrow()*
- *PolarAxes* gains getters and setters for "theta_direction", and "theta_offset" to allow for theta to go in either the clock-wise or counter-clockwise direction and to specify where zero degrees should be placed. *set_theta_zero_location()* is an added convenience function.

- Fixed error in argument handling for tri-functions such as `tripcolor()`
- `axes.labelweight` parameter added to rcParams.
- For `imshow()`, *interpolation='nearest'* will now always perform an interpolation. A "none" option has been added to indicate no interpolation at all.
- An error in the Hammer projection has been fixed.
- *clabel* for `contour()` now accepts a callable. Thanks to Daniel Hyams for the original patch.
- Jae-Joon Lee added the `HBox` and `VBox` classes.
- Christoph Gohlke reduced memory usage in `imshow()`.
- `scatter()` now accepts empty inputs.
- The behavior for 'symlog' scale has been fixed, but this may result in some minor changes to existing plots. This work was refined by ssyr.
- Peter Butterworth added named figure support to `figure()`.
- Michiel de Hoon has modified the MacOSX backend to make its interactive behavior consistent with the other backends.
- Pim Schellart added a new colormap called "cubehelix". Sameer Grover also added a colormap called "coolwarm". See it and all other colormaps *here*.
- Many bug fixes and documentation improvements.

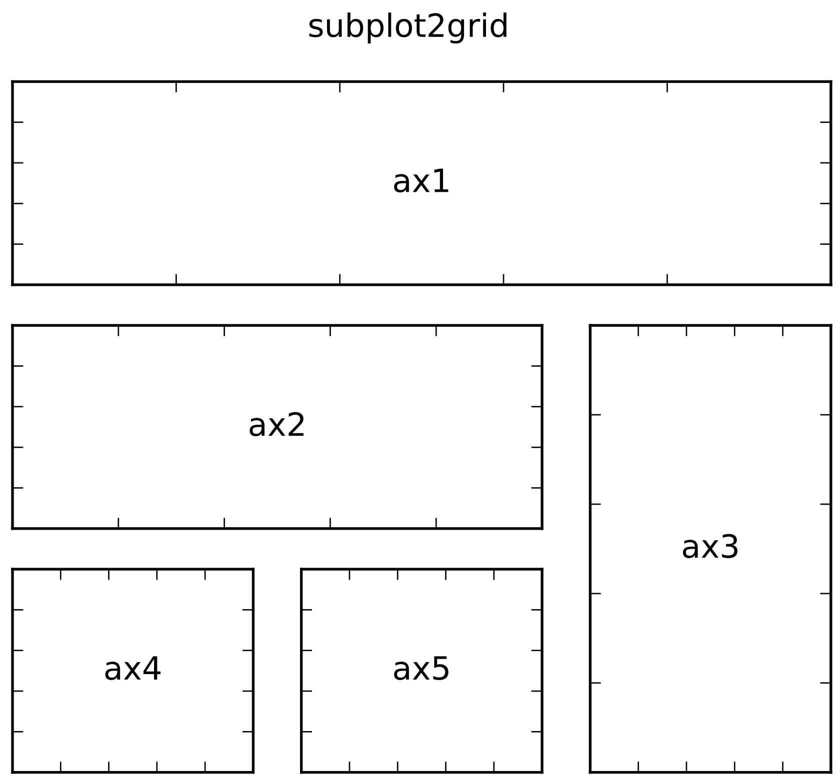
5.7 new in matplotlib-1.0

5.7.1 HTML5/Canvas backend

Simon Ratcliffe and Ludwig Schwardt have released an HTML5/Canvas backend for matplotlib. The backend is almost feature complete, and they have done a lot of work comparing their html5 rendered images with our core renderer Agg. The backend features client/server interactive navigation of matplotlib figures in an html5 compliant browser.

5.7.2 Sophisticated subplot grid layout

Jae-Joon Lee has written `gridspec`, a new module for doing complex subplot layouts, featuring row and column spans and more. See *Customizing Location of Subplot Using GridSpec* for a tutorial overview.



5.7.3 Easy pythonic subplots

Fernando Perez got tired of all the boilerplate code needed to create a figure and multiple subplots when using the matplotlib API, and wrote a *subplots()* helper function. Basic usage allows you to create the figure and an array of subplots with numpy indexing (starts with 0). e.g.:

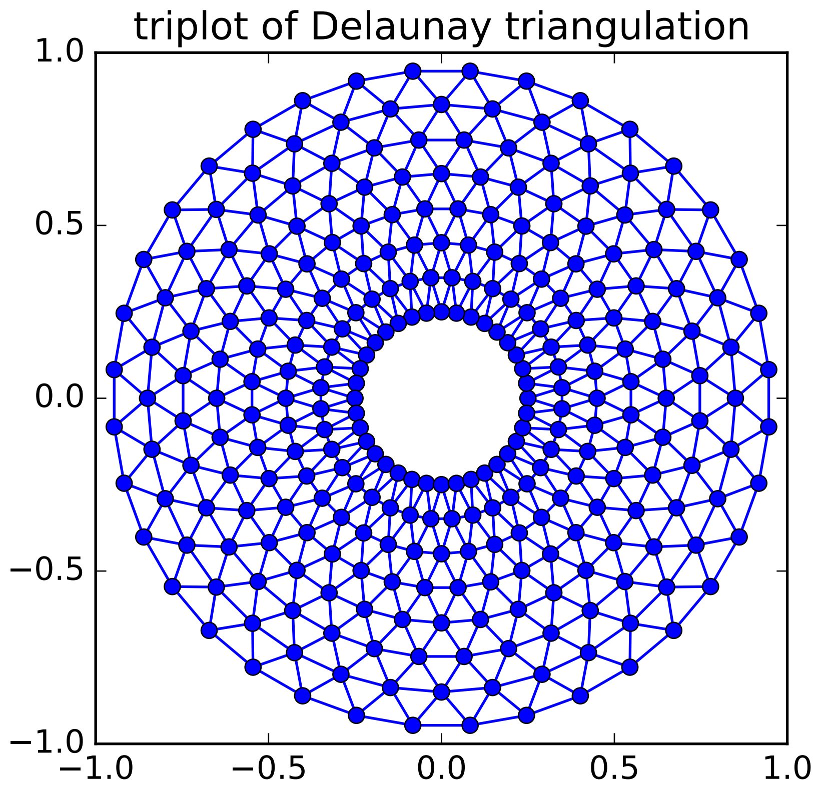
```
fig, axarr = plt.subplots(2, 2)
axarr[0,0].plot([1,2,3])   # upper, left
```

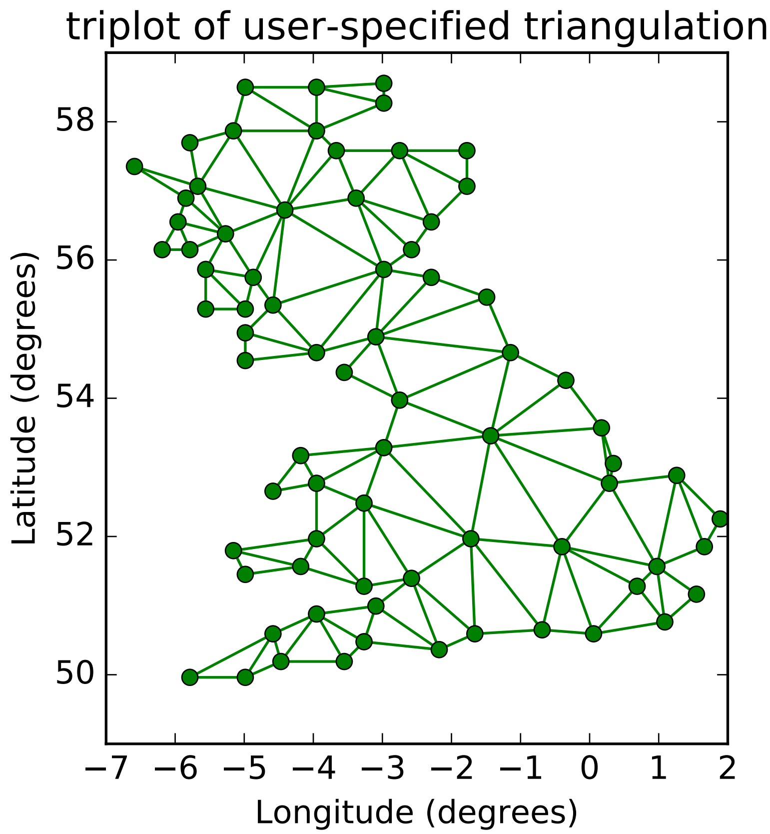
See *pylab_examples example code: subplots_demo.py* for several code examples.

5.7.4 Contour fixes and and triplot

Ian Thomas has fixed a long-standing bug that has vexed our most talented developers for years. *contourf()* now handles interior masked regions, and the boundaries of line and filled contours coincide.

Additionally, he has contributed a new module *tri* and helper function *triplot()* for creating and plotting unstructured triangular grids.


triplot of Delaunay triangulation
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

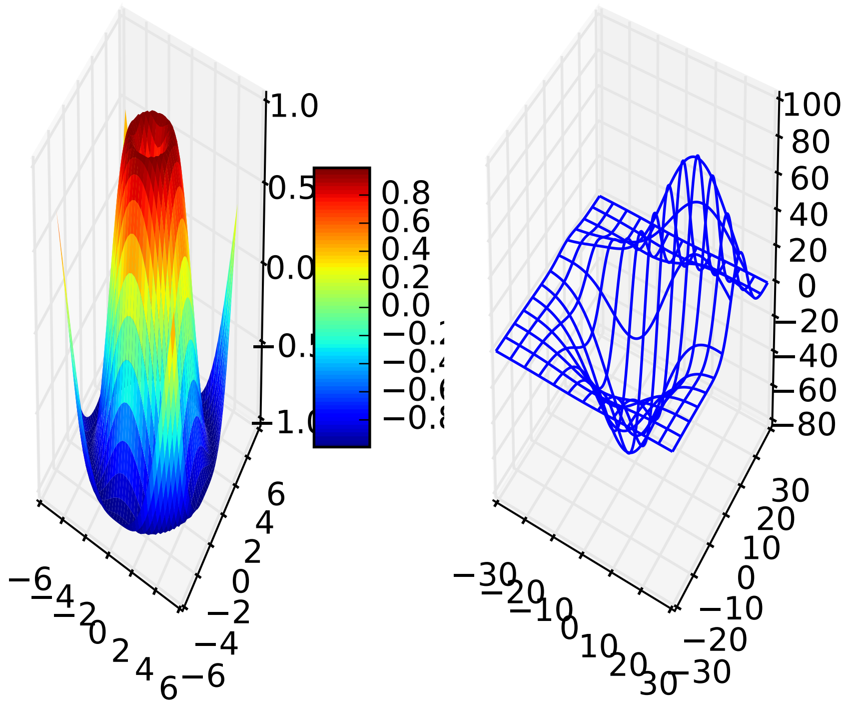


5.7.5 multiple calls to show supported

A long standing request is to support multiple calls to *show()*. This has been difficult because it is hard to get consistent behavior across operating systems, user interface toolkits and versions. Eric Firing has done a lot of work on rationalizing show across backends, with the desired behavior to make show raise all newly created figures and block execution until they are closed. Repeated calls to show should raise newly created figures since the last call. Eric has done a lot of testing on the user interface toolkits and versions and platforms he has access to, but it is not possible to test them all, so please report problems to the mailing list and bug tracker.

5.7.6 mplot3d graphs can be embedded in arbitrary axes

You can now place an mplot3d graph into an arbitrary axes location, supporting mixing of 2D and 3D graphs in the same figure, and/or multiple 3D graphs in a single figure, using the "projection" keyword argument to add_axes or add_subplot. Thanks Ben Root.



5.7.7 tick_params

Eric Firing wrote tick_params, a convenience method for changing the appearance of ticks and tick labels. See pyplot function `tick_params()` and associated Axes method `tick_params()`.

5.7.8 Lots of performance and feature enhancements

- Faster magnification of large images, and the ability to zoom in to a single pixel
- Local installs of documentation work better
- Improved "widgets" – mouse grabbing is supported
- More accurate snapping of lines to pixel boundaries
- More consistent handling of color, particularly the alpha channel, throughout the API

5.7.9 Much improved software carpentry

The matplotlib trunk is probably in as good a shape as it has ever been, thanks to improved software carpentry. We now have a buildbot which runs a suite of nose regression tests on every svn commit, auto-generating a set of images and comparing them against a set of known-goods, sending emails to developers on failures with a pixel-by-pixel image comparison. Releases and release bugfixes happen in branches, allowing active new feature development to happen in the trunk while keeping the release branches stable. Thanks to Andrew Straw, Michael Droettboom and other matplotlib developers for the heavy lifting.

5.7.10 Bugfix marathon

Eric Firing went on a bug fixing and closing marathon, closing over 100 bugs on the bug tracker with help from Jae-Joon Lee, Michael Droettboom, Christoph Gohlke and Michiel de Hoon.

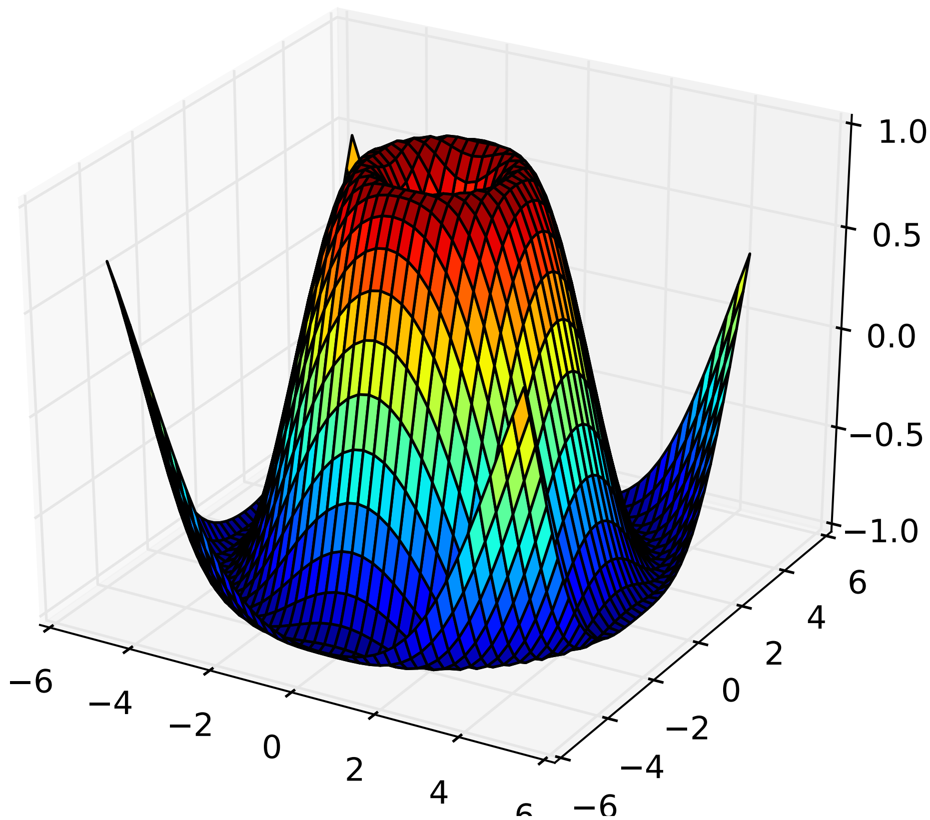
5.8 new in matplotlib-0.99

5.8.1 New documentation

Jae-Joon Lee has written two new guides *Legend guide* and *Annotating Axes*. Michael Sarahan has written *Image tutorial*. John Hunter has written two new tutorials on working with paths and transformations: *Path Tutorial* and *Transformations Tutorial*.

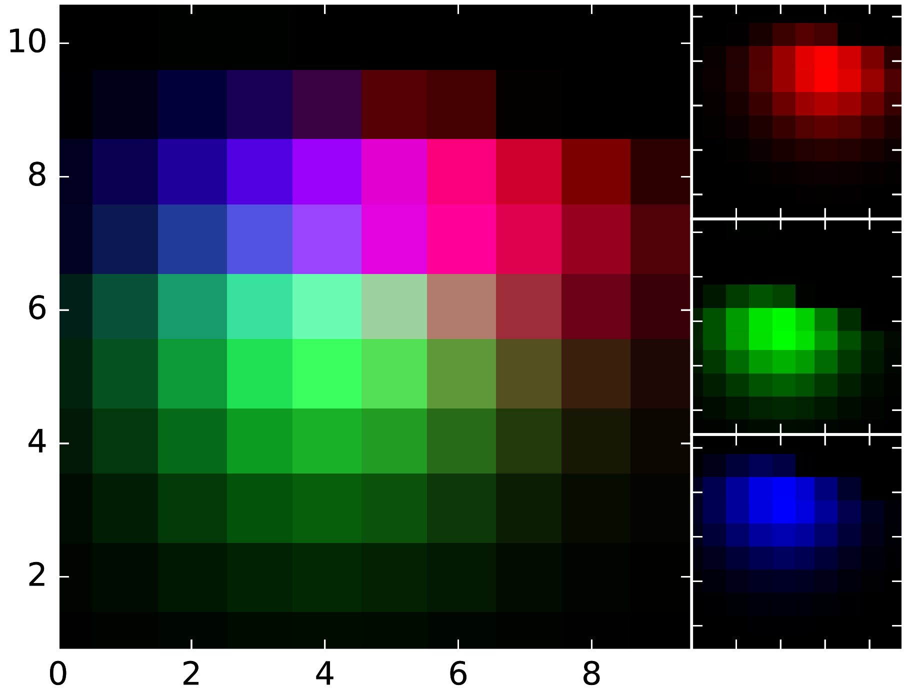
5.8.2 mplot3d

Reinier Heeres has ported John Porter's mplot3d over to the new matplotlib transformations framework, and it is now available as a toolkit mpl_toolkits.mplot3d (which now comes standard with all mpl installs). See *mplot3d Examples* and *mplot3d tutorial*



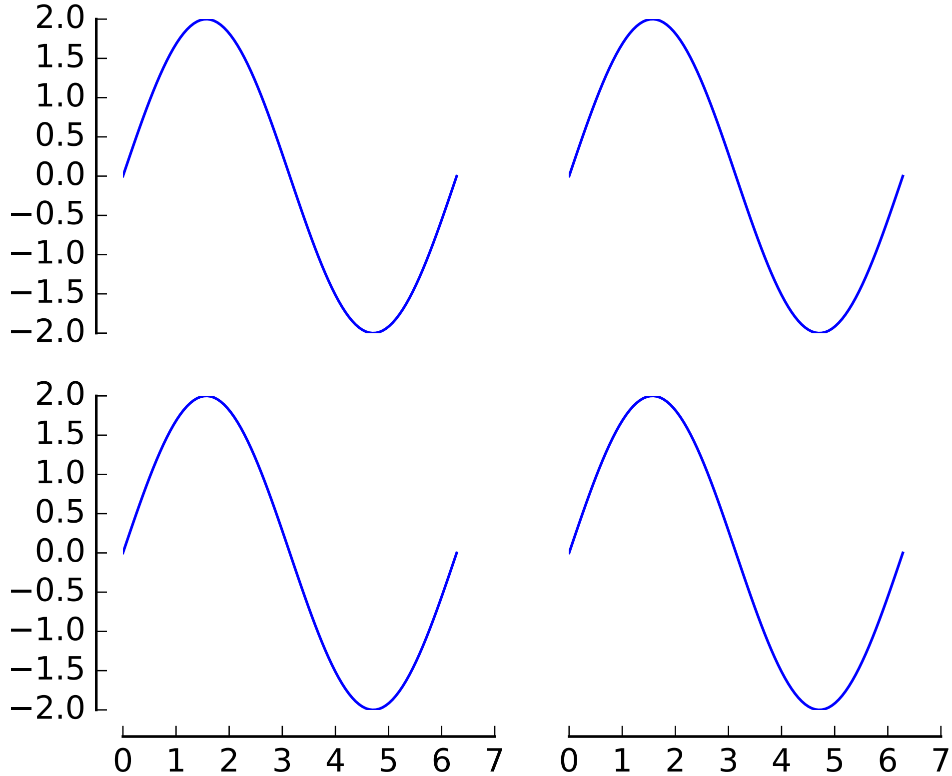
5.8.3 axes grid toolkit

Jae-Joon Lee has added a new toolkit to ease displaying multiple images in matplotlib, as well as some support for curvilinear grids to support the world coordinate system. The toolkit is included standard with all new mpl installs. See *axes_grid Examples* and *The Matplotlib AxesGrid Toolkit User's Guide*.



5.8.4 Axis spine placement

Andrew Straw has added the ability to place "axis spines" – the lines that denote the data limits – in various arbitrary locations. No longer are your axis lines constrained to be a simple rectangle around the figure – you can turn on or off left, bottom, right and top, as well as "detach" the spine to offset it away from the data. See *pylab_examples example code: spine_placement_demo.py* and `matplotlib.spines.Spine`.



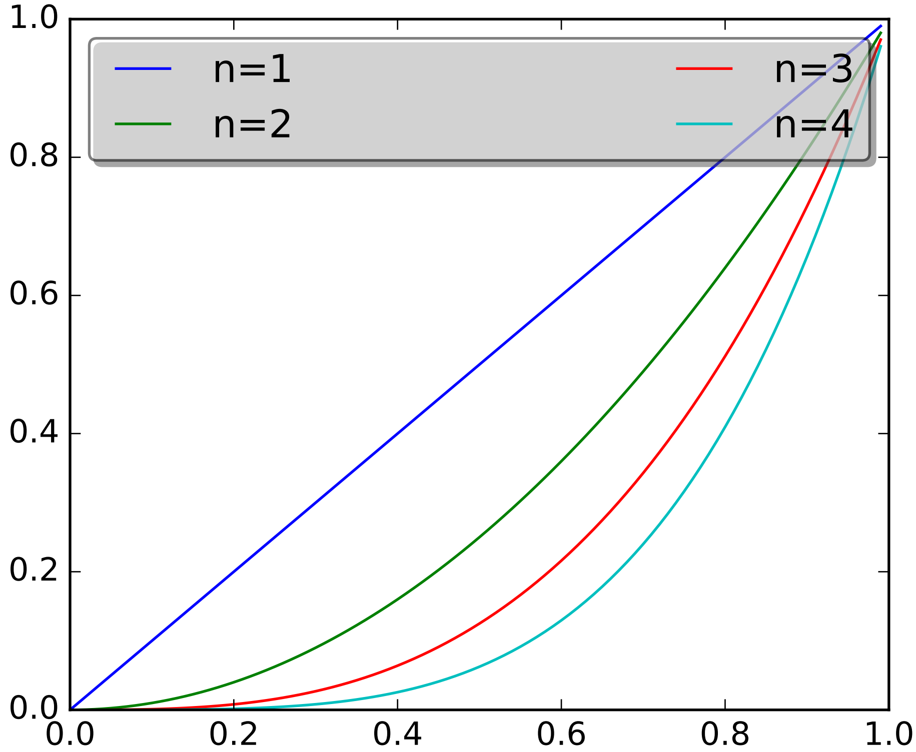
5.9 new in 0.98.4

It's been four months since the last matplotlib release, and there are a lot of new features and bug-fixes.

Thanks to Charlie Moad for testing and preparing the source release, including binaries for OS X and Windows for python 2.4 and 2.5 (2.6 and 3.0 will not be available until numpy is available on those releases). Thanks to the many developers who contributed to this release, with contributions from Jae-Joon Lee, Michael Droettboom, Ryan May, Eric Firing, Manuel Metz, Jouni K. Seppänen, Jeff Whitaker, Darren Dale, David Kaplan, Michiel de Hoon and many others who submitted patches

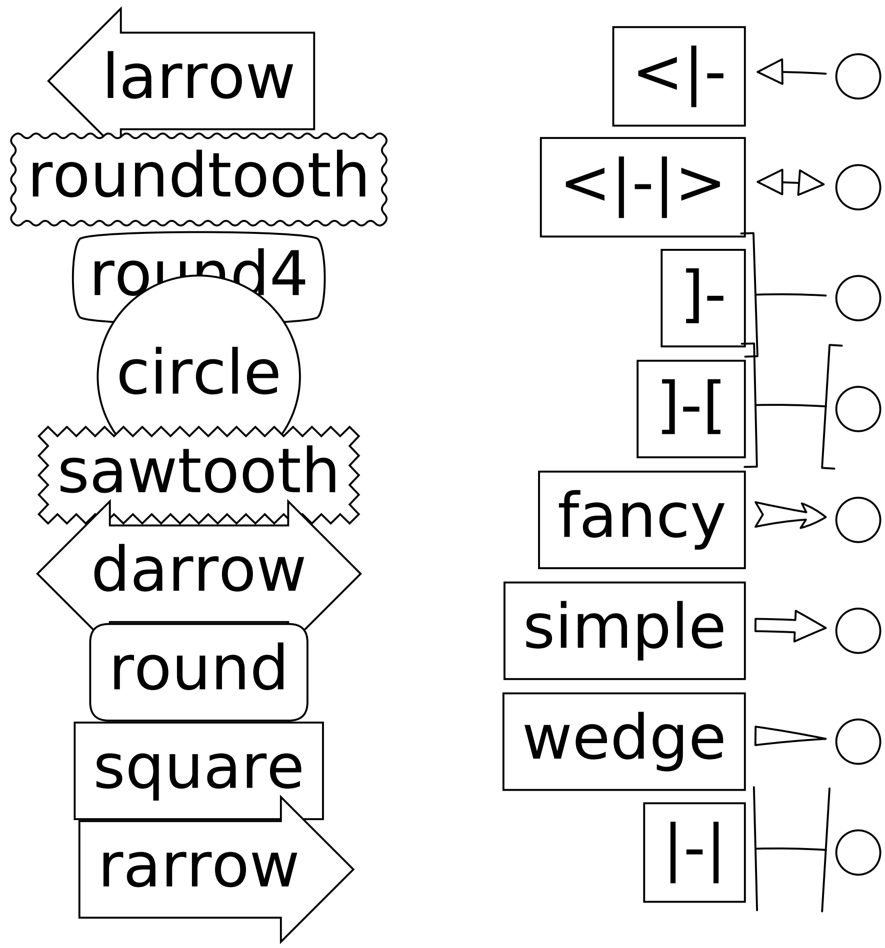
5.9.1 Legend enhancements

Jae-Joon has rewritten the legend class, and added support for multiple columns and rows, as well as fancy box drawing. See `legend()` and `matplotlib.legend.Legend`.



5.9.2 Fancy annotations and arrows

Jae-Joon has added lots of support to annotations for drawing fancy boxes and connectors in annotations. See *annotate()* and *BoxStyle*, *ArrowStyle*, and *ConnectionStyle*.



5.9.3 Native OS X backend

Michiel de Hoon has provided a native Mac OSX backend that is almost completely implemented in C. The backend can therefore use Quartz directly and, depending on the application, can be orders of magnitude faster than the existing backends. In addition, no third-party libraries are needed other than Python and NumPy. The backend is interactive from the usual terminal application on Mac using regular Python. It hasn't been tested with ipython yet, but in principle it should to work there as well. Set 'backend : macosx' in your matplotlibrc file, or run your script with:

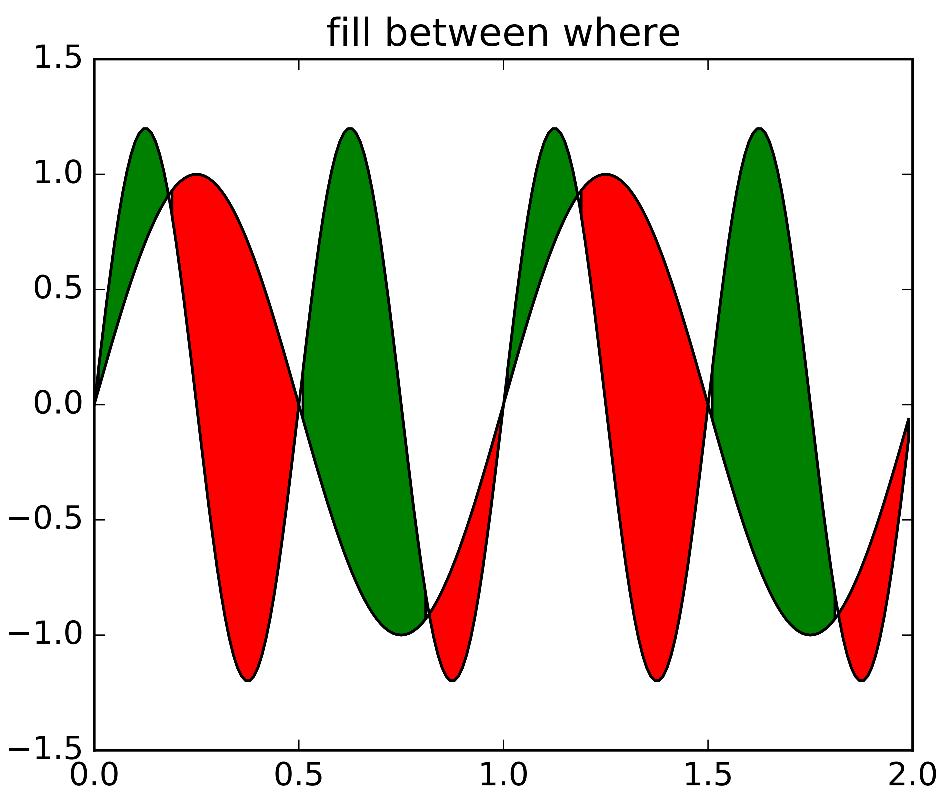
```
> python myfile.py -dmacosx
```

5.9.4 psd amplitude scaling

Ryan May did a lot of work to rationalize the amplitude scaling of `psd()` and friends. See *pylab_examples example code: psd_demo2.py*. and *pylab_examples example code: psd_demo3.py*. The changes should increase MATLAB compatibility and increase scaling options.

5.9.5 Fill between

Added a `fill_between()` function to make it easier to do shaded region plots in the presence of masked data. You can pass an *x* array and a *ylower* and *yupper* array to fill between, and an optional *where* argument which is a logical mask where you want to do the filling.



5.9.6 Lots more

Here are the 0.98.4 notes from the CHANGELOG:

```
Added mdehoon's native macosx backend from sf patch 2179017 - JDH

Removed the prints in the set_*style commands.  Return the list of
pretty-printed strings instead - JDH

Some of the changes Michael made to improve the output of the
```

```
property tables in the rest docs broke of made difficult to use
some of the interactive doc helpers, e.g., setp and getp.  Having all
the rest markup in the ipython shell also confused the docstrings.
I added a new rc param docstring.harcopy, to format the docstrings
differently for hardcopy and other use.  The ArtistInspector
could use a little refactoring now since there is duplication of
effort between the rest out put and the non-rest output - JDH

Updated spectral methods (psd, csd, etc.) to scale one-sided
densities by a factor of 2 and, optionally, scale all densities by
the sampling frequency.  This gives better MATLAB
compatibility. -RM

Fixed alignment of ticks in colorbars. -MGD

drop the deprecated "new" keyword of np.histogram() for numpy 1.2
or later.  -JJL

Fixed a bug in svg backend that new_figure_manager() ignores
keywords arguments such as figsize, etc. -JJL

Fixed a bug that the handlelength of the new legend class set too
short when numpoints=1 -JJL

Added support for data with units (e.g., dates) to
Axes.fill_between. -RM

Added fancybox keyword to legend. Also applied some changes for
better look, including baseline adjustment of the multiline texts
so that it is center aligned. -JJL

The transmuter classes in the patches.py are reorganized as
subclasses of the Style classes. A few more box and arrow styles
are added. -JJL

Fixed a bug in the new legend class that didn't allowed a tuple of
coordinate values as loc. -JJL

Improve checks for external dependencies, using subprocess
(instead of deprecated popen*) and distutils (for version
checking) - DSD

Reimplementation of the legend which supports baseline alignment,
multi-column, and expand mode. - JJL

Fixed histogram autoscaling bug when bins or range are given
explicitly (fixes Debian bug 503148) - MM

Added rcParam axes.unicode_minus which allows plain hyphen for
minus when False - JDH

Added scatterpoints support in Legend. patch by Erik Tollerud -
JJL
```

```
Fix crash in log ticking. - MGD

Added static helper method BrokenHBarCollection.span_where and
Axes/pyplot method fill_between.  See
examples/pylab/fill_between.py - JDH

Add x_isdata and y_isdata attributes to Artist instances, and use
them to determine whether either or both coordinates are used when
updating dataLim.  This is used to fix autoscaling problems that
had been triggered by axhline, axhspan, axvline, axvspan. - EF

Update the psd(), csd(), cohere(), and specgram() methods of Axes
and the csd() cohere(), and specgram() functions in mlab to be in
sync with the changes to psd().  In fact, under the hood, these
all call the same core to do computations. - RM

Add 'pad_to' and 'sides' parameters to mlab.psd() to allow
controlling of zero padding and returning of negative frequency
components, respectively.  These are added in a way that does not
change the API. - RM

Fix handling of c kwarg by scatter; generalize is_string_like to
accept numpy and numpy.ma string array scalars. - RM and EF

Fix a possible EINTR problem in dviread, which might help when
saving pdf files from the qt backend. - JKS

Fix bug with zoom to rectangle and twin axes - MGD

Added Jae Joon's fancy arrow, box and annotation enhancements --
see examples/pylab_examples/annotation_demo2.py

Autoscaling is now supported with shared axes - EF

Fixed exception in dviread that happened with Minion - JKS

set_xlim, ylim now return a copy of the viewlim array to avoid
modify inplace surprises

Added image thumbnail generating function
matplotlib.image.thumbnail.  See examples/misc/image_thumbnail.py
- JDH

Applied scatleg patch based on ideas and work by Erik Tollerud and
Jae-Joon Lee. - MM

Fixed bug in pdf backend: if you pass a file object for output
instead of a filename, e.g., in a wep app, we now flush the object
at the end. - JKS

Add path simplification support to paths with gaps. - EF
```

```
Fix problem with AFM files that don't specify the font's full name
or family name. - JKS

Added 'scilimits' kwarg to Axes.ticklabel_format() method, for
easy access to the set_powerlimits method of the major
ScalarFormatter. - EF

Experimental new kwarg borderpad to replace pad in legend, based
on suggestion by Jae-Joon Lee.  - EF

Allow spy to ignore zero values in sparse arrays, based on patch
by Tony Yu.  Also fixed plot to handle empty data arrays, and
fixed handling of markers in figlegend. - EF

Introduce drawstyles for lines. Transparently split linestyles
like 'steps--' into drawstyle 'steps' and linestyle '--'.  Legends
always use drawstyle 'default'. - MM

Fixed quiver and quiverkey bugs (failure to scale properly when
resizing) and added additional methods for determining the arrow
angles - EF

Fix polar interpolation to handle negative values of theta - MGD

Reorganized cbook and mlab methods related to numerical
calculations that have little to do with the goals of those two
modules into a separate module numerical_methods.py Also, added
ability to select points and stop point selection with keyboard in
ginput and manual contour labeling code.  Finally, fixed contour
labeling bug. - DMK

Fix backtick in Postscript output. - MGD

[ 2089958 ] Path simplification for vector output backends
Leverage the simplification code exposed through path_to_polygons
to simplify certain well-behaved paths in the vector backends
(PDF, PS and SVG).  "path.simplify" must be set to True in
matplotlibrc for this to work.  - MGD

Add "filled" kwarg to Path.intersects_path and
Path.intersects_bbox. - MGD

Changed full arrows slightly to avoid an xpdf rendering problem
reported by Friedrich Hagedorn. - JKS

Fix conversion of quadratic to cubic Bezier curves in PDF and PS
backends. Patch by Jae-Joon Lee. - JKS

Added 5-point star marker to plot command q- EF

Fix hatching in PS backend - MGD

Fix log with base 2 - MGD
```

```
Added support for bilinear interpolation in
NonUniformImage; patch by Gregory Lielens. - EF

Added support for multiple histograms with data of
different length - MM

Fix step plots with log scale - MGD

Fix masked arrays with markers in non-Agg backends - MGD

Fix clip_on kwarg so it actually works correctly - MGD

Fix locale problems in SVG backend - MGD

fix quiver so masked values are not plotted - JSW

improve interactive pan/zoom in qt4 backend on windows - DSD

Fix more bugs in NaN/inf handling.  In particular, path
simplification (which does not handle NaNs or infs) will be turned
off automatically when infs or NaNs are present.  Also masked
arrays are now converted to arrays with NaNs for consistent
handling of masks and NaNs - MGD and EF

Added support for arbitrary rasterization resolutions to the SVG
backend. - MW
```

CHAPTER

SIX

GITHUB STATS

GitHub stats for 2014/08/26 - 2016/09/08 (tag: v1.4.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 1189 issues and merged 1214 pull requests.

The following 360 authors contributed 6335 commits.

- AbdealiJK
- Acanthostega
- Adrien Chardon
- Adrien F. Vincent
- Adrien VINCENT
- Alan Du
- Alberto
- alex
- Alex Rothberg
- Alexander Taylor
- Alexei Colin
- Ali Mehdi
- Ali Uneri
- Alistair Muldal
- Allan Haldane
- AmyTeegarden
- Andreas Hilboll
- Andreas Mayer
- Andy Zhu
- Anton Akhmerov

- Antony Lee
- Arie
- Ariel Hernán Curiale
- Arnaud Gardelein
- Arpad Horvath
- basharovV
- bcongdon
- Behram Mistree
- Ben Congdon
- Ben Root
- Benjamin Berg
- Benjamin Congdon
- Benjamin Reedlunn
- BHT
- Brett Cannon
- Brian McLaughlin
- Bruno Beltran
- Cameron Davidson-Pilon
- cammil
- Casey Webster
- Casper van der Wel
- caspervdw
- chadawagner
- chebee7i
- Chen Karako
- Chris Holdgraf
- Christian Brueffer
- Christoph Gohlke
- Cimarron Mittelsteadt
- CJ Carey
- curiale
- DaCoEx

- Damon McDougall
- danhickstein
- DanHickstein
- Danhickstein
- David
- David Haberthür
- David Kua
- David Stansby
- deeenes
- Devashish Deshpande
- Diego Mora Cespedes
- dlmccaffrey
- domspad
- Dora Fraeman
- dsquareindia
- Duncan Macleod
- Dylan Evans
- e-q
- 5. (a) Patrick Bos
- Elena Glassman
- Elias Pipping
- Elliott Sales de Andrade
- elpres
- Elvis Stansvik
- Emil Mikulic
- endolith
- Endolith
- Eric Dill
- Eric Firing
- Eric Ma
- Eric O. LEBIGOT (EOL)
- Erik Bray

- Eugen Beck
- Eugene Yurtsev
- Fabien Maussion
- Fabio Zanini
- Federico Ariza
- ffteja
- fibersnet
- Florian LB
- Florian Rhiem
- flothesof
- Francesco Montesano
- Francis Colas
- Francoise Provencher
- Frank Yu
- François Magimel
- frenchwr
- fvgoto
- Gaute Hope
- gcallah
- gepcel
- Giovanni
- gluap
- Gregory Ashton
- Gregory R. Lee
- Grillard
- hamogu
- hannah
- Hans Dembinski
- Hans Moritz Günther
- Hassan Kibirige
- Hastings Greer
- Heiko Oberdiek

- Henning Pohl
- Herbert Kruitbosch
- Holger Peters
- hugadams
- Ian Thomas
- Ilia Kurenkov
- ImSoErgodic
- insertroar
- Ioannis Filippidis
- Isaac Schwabacher
- Isaac Slavitt
- Ismo Toijala
- itziakos
- 10. Goutin
- Jacob McDonald
- Jae-Joon Lee
- Jaime Fernandez
- jaimefrio
- Jake VanderPlas
- James A. Bednar
- James Pallister
- James R. Evans
- Jan Schlüter
- Jan Schulz
- Jan-willem De Bleser
- Jascha Ulrich
- Jason King
- Jason Liw Yan Chong
- Jason Miller
- Jason Zheng
- JayP16
- jbbrokaw

- Jeff Lutgen
- Jeffrey Hokanson @ Loki
- Jens Hedegaard Nielsen
- Jeremy Fix
- Jessica B. Hamrick
- JGoutin
- jlutgen
- Jody Klymak
- Joe Kington
- Joel B. Mohler
- John Vandenberg
- jonchar
- Jorrit Wronski
- Josef Heinen
- Joseph Fox-Rabinovitz
- Joseph Jon Booker
- Jouni K. Seppänen
- jowr
- Julian Mehne
- Julian V. Modesto
- Julien Lhermitte
- Julien Schueller
- Julien-Charles Lévesque
- Katy Huff
- Kevin Keating
- khyox
- kikocorreoso
- Kimmo Palin
- Kjartan Myrdal
- klaus
- klonuo
- Konrad Förstner

- Konstantin Tretyakov
- Kristen M. Thyng
- Kristen Thyng
- Kyle Bridgemohansingh
- Kyler Brown
- Leeonadoh
- leeonadoh
- Lennart Fricke
- Leo Singer
- Levi Kilcher
- lichri12
- LindyBalboa
- Lori J
- Loïc Estève
- Luis Pedro Coelho
- Mad Physicist
- Magnus Nord
- Majid alDosari
- maluethi
- mamrehn
- Marcos Duarte
- Marek Rudnicki
- Marin Gilles
- Markus Rothe
- Martin Fitzpatrick
- Martin Thoma
- masamson
- Masud Rahman
- Mathieu Duponchelle
- Matt Giuca
- Matt Hancock
- Matt Li

- Matt Shen
- Matthew Brett
- matthias
- Matthias Bussonnier
- Maximilian Albert
- Maximilian Maahn
- mbyt
- mdehoon
- mdipierro
- Mellissa Cross
- Michael Droettboom
- Michael Sarahan
- Michiel de Hoon
- Min RK
- Minty Zhang
- MirandaXM
- mlub
- mrkrd
- muahah
- Muhammad Mehdi
- myyc
- Nathan Goldbaum
- Nathan Musoke
- Neil Crighton
- Nelle Varoquaux
- Niall Robinson
- Nicholas Devenish
- nickystringer
- Nico Schlömer
- Nicolas P. Rougier
- Nicolas Tessore
- Nikita Kniazev

- Niklas Koep
- Nils Werner
- none
- nwin
- Ocean Wolf
- OceanWolf
- ocefpaf
- Oleg Selivanov
- Olga Botvinnik
- Orso Meneghini
- Pankaj Pandey
- Parfenov Sergey
- patchen
- Patrick Chen
- paul
- Paul G
- Paul Ganssle
- Paul Hobson
- Paul Kirow
- Paul Romano
- Pete Bachant
- Pete Peterson
- Peter Mortensen
- Peter St. John
- Peter Würtz
- Phil Elson
- Pierre de Buyl
- productivememberofsociety666
- Przemysław Dąbek
- pupssman
- Qingpeng "Q.P." Zhang
- Ramiro Gómez

- Randy Olson
- rasbt
- Remi Rampin
- Robin Dunn
- Robin Wilson
- rsnape
- ryan
- Ryan May
- Ryan Morshead
- Ryan Nelson
- ryanbelt
- RyanPan
- s9w
- Samuel St-Jean
- Sander
- scls19fr
- Scott Howard
- Scott Lawrence
- scott-vsi
- sdementen
- Sebastián Vanrell
- Simon Gibbons
- Skelpdar
- Slav
- sohero
- Spencer McIntyre
- Stanley, Simon
- Stefan Lehmann
- Stefan Pfenninger
- Stefan van der Walt
- Stephen Horst
- Sterling Smith

- Steven Silvester
- story645
- Stuart Mumford
- switham
- Tamas Gal
- Terrence J. Katzenbaer
- The Gitter Badger
- Thomas A Caswell
- Thomas Hisch
- Thomas Lake
- Thomas Robitaille
- Thomas Spura
- Thorsten Liebig
- Till Stensitzki
- tmdavison
- Tobias Megies
- Tomas Kazmar
- tomoemon
- Trish Gillett-Kawamoto
- u55
- ugurthemaster
- Ulrich Dobramysl
- ultra-andy
- Umair Idris
- Vadim Markovtsev
- Valentin Schmidt
- Víctor Zabalza
- Warren Weckesser
- Wen Li
- Wendell Smith
- Werner F Bruhin
- wernerfb

- Wieland Hoffmann
- William Manley
- Xiaowen Tang
- xuanyuansen
- Xufeng Wang
- Yu Feng
- Yunfei Yang
- Yuri D'Elia
- Yuval Langer
- Zair Mubashar
- Zbigniew Jędrzejewski-Szmek

GitHub issues and pull requests:

Pull Requests (1214):

- PR #7037: DOC change axhspan to numpydoc format
- PR #7047: DOC - SpanSelector widget documentation
- PR #7049: [MRG] Documentated dependencies to the doc and remove unecessary dependencies.
- PR #7063: Tweek tol for test_hist_steplog to fix tests on appveyor
- PR #7055: FIX: testings.nose was not installed
- PR #7058: Minor animation fixes
- PR #7057: FIX: Removed financial demos that stalled because of yahoo requests
- PR #7052: Uncaught exns are fatal for PyQt5, so catch them.
- PR #7048: FIX: remove unused variable
- PR #7042: FIX: ticks filtered by Axis, not in Tick.draw
- PR #7026: Merge 2.x to master
- PR #6988: Text box widget, take over of PR5375
- PR #6957: DOC: clearing out some instances of using pylab in the docs
- PR #7012: Don't blacklist test_usetex using pytest
- PR #7011: TST: Fixed `skip_if_command_unavailable` decorator problem
- PR #6918: enable previously leftout test_usetex
- PR #7006: FIX: sphinx 1.4.0 details
- PR #6900: Enh: break website screenshot banner into 4 pieces and introduce a responsive layout
- PR #6997: FIX: slow plots of pandas objects (Second try)

- PR #6792: PGF Backend: Support interpolation='none'
- PR #6983: Catch invalid interactive switch to log scale.
- PR #6491: Don't warn in Collections.contains if picker is not numlike.
- PR #6978: Add link to O'Reilly video course covering matplotlib
- PR #6930: BUG: PcolorImage handles non-contiguous arrays, provides data readout
- PR #6889: support for updating axis ticks for categorical data
- PR #6974: Fixed wrong expression
- PR #6730: Add Py.test testing framework support
- PR #6904: Use edgecolor rather than linewidth to control edge display.
- PR #6919: Rework MaxNLocator, eliminating infinite loop; closes #6849
- PR #6955: Add parameter checks to DayLocator initiator
- PR #5161: [WIP] Proposed change to default log scale tick formatting
- PR #6875: Add keymap (default: G) to toggle minor grid.
- PR #6920: Prepare for cross-framework test suite
- PR #6944: Restore cbook.report_memory, which was deleted in d063dee.
- PR #6961: remove extra "a"
- PR #6947: Changed error message. Issue #6933
- PR #6923: Make sure nose is only imported when needed
- PR #6851: Do not restrict coverage to `matplotlib` module only
- PR #6938: Image interpolation selector in Qt figure options.
- PR #6787: Python3.5 dictview support
- PR #6407: adding default toggled state for toggle tools
- PR #6898: Fix read mode when loading cached AFM fonts
- PR #6892: Don't force anncoords to fig coords upon dragging.
- PR #6895: Prevent forced alpha in figureoptions.
- PR #6877: Fix Path deepcopy signature
- PR #6822: Use travis native cache
- PR #6821: Break reference cycle Line2D <-> Line2D._lineFunc.
- PR #6879: Delete font cache in one of the configurations
- PR #6832: Fix for ylabel title in example tex_unicode_demo.py
- PR #6848: `test_tinypages`: pytest compatible module level setup
- PR #6881: add doi to bibtex entry for Hunter (2007)

- PR #6842: Clarify Axes.hexbin *extent* docstring
- PR #6861: Update ggplot URLs
- PR #6878: DOC: use venv instead of virtualenv on python 3
- PR #6837: Fix Normalize(<signed integer array>).
- PR #6874: Update bachelors_degree_by_gender example.
- PR #6867: Mark `make_all_2d_testfuncs` as not a test
- PR #6854: Fix for PyQt5.7 support.
- PR #6862: Change default doc image format to png and pdf
- PR #6819: Add mpl_toolkits to coveragerc.
- PR #6840: Fixed broken `test_pickle.test_complete` test
- PR #6841: DOC: Switch to OO code style & ensure fixed y-range in `psd_demo3`
- PR #6843: DOC: Fix `psd_demo_complex` similarly to `psd_demo3`
- PR #6829: Tick label rotation via `set_tick_params`
- PR #6799: Allow creating annotation arrows w/ default props.
- PR #6262: Properly handle UTC conversion in date2num.
- PR #6777: Raise lock timeout as actual exception
- PR #6817: DOC: Fix a few typos and formulations
- PR #6826: Clarify doc for "norm" kwarg to `imshow`.
- PR #6807: Deprecate `{get,set}_cursorprops`.
- PR #6811: Add xkcd font as one of the options
- PR #6815: Rename tests in `test_mlab.py`
- PR #6808: Don't forget to disconnect callbacks for dragging.
- PR #6803: better freetype version checking
- PR #6778: Added contribute information to readme
- PR #6786: 2.0 Examples fixes. See #6762
- PR #6774: Appveyor: use newer conda packages and only run all tests on one platform
- PR #6779: Fix tutorial pyplot scales (issue #6775)
- PR #6768: Takeover #6535
- PR #6763: Invalidate test cache on gs/inkscape version
- PR #6765: Get more rcParams for 3d
- PR #6764: Support returning polylines from to_polygons
- PR #6760: DOC: clean up of demo_annotation_box.py

- PR #6735: Added missing side tick rcParams
- PR #6761: Fixed warnings catching and counting with `warnings.catch_warnings`
- PR #5349: Add a Gitter chat badge to README.rst
- PR #6755: PEP: fix minor formatting issues
- PR #6699: Warn if MPLBACKEND is invalid.
- PR #6754: Fixed error handling in `ImageComparisonTest.setup_class`
- PR #6734: register IPython's eventloop integration in plt.install_repl_displayhook
- PR #6745: DOC: typo in broken_axis pylab example
- PR #6747: Also output the actual error on svg backend tests using subprocess
- PR #6744: Add workaround for failures due to newer miktex
- PR #6741: Missing `cleanup` decorator in `test_subplots.test_exceptions`
- PR #6736: doc: fix unescaped backslash
- PR #6733: Mergev2.x to master
- PR #6729: Fix crash if byte-compiled level 2
- PR #6575: setup.py: Recommend installation command for pkgs
- PR #6645: Fix containment and subslice optim. for steps.
- PR #6619: Hide "inner" {x,y}labels in label_outer too.
- PR #6639: Simplify get_legend_handler method
- PR #6694: Improve Line2D and MarkerStyle instantiation
- PR #6692: Remove explicit children invalidation in update_position method
- PR #6703: DOC: explain behavior of notches beyond quartiles
- PR #6707: Call `gc.collect` after each test only if the user asks for it
- PR #6711: Added support for `mgs` to Ghostscript dependecy checker
- PR #6700: Don't convert vmin, vmax to floats.
- PR #6714: fixed font_manager.is_opentype_cff_font()
- PR #6701: Colours like 'XeYYYY' don't get recognised properly if X, Y's are numbers
- PR #6512: Add computer modern font family
- PR #6383: Qt editor alpha
- PR #6381: Fix canonical name for "None" linestyle.
- PR #6689: Str Categorical Axis Support
- PR #6686: Merged _bool from axis into cbook._string_to_bool
- PR #6683: New entry in `.mailmap`

- PR #6520: Appveyor overhaul
- PR #6697: Fixed path caching bug in `Path.unit_regular_star`
- PR #6688: DOC: fix radial increase of size & OO style in polar_scatter_demo
- PR #6681: Fix #6680 (minor typo in IdentityTransform docstring)
- PR #6676: Fixed AppVeyor building script
- PR #6672: Fix example of streamplot `start_points` option
- PR #6601: BF: protect against locale in sphinext text
- PR #6662: [MRG+1] adding from_list to custom cmap tutorial
- PR #6666: Guard against too-large figures
- PR #6659: Fix image alpha
- PR #6642: Fix rectangle selector release bug
- PR #6652: Minor doc updates.
- PR #6653: DOC: Incorrect rendering of dashes
- PR #6648: adding a new color and editing an existing color in fivethirtyeight.mplstyle
- PR #6548: Fix typo.
- PR #6628: fix the swab bug to compile on solaris system
- PR #6622: colors: ensure masked array data is an ndarray
- PR #6625: DOC: Found a typo.
- PR #6614: Fix docstring for PickEvent.
- PR #6554: Update mpl_toolkits.gtktools.
- PR #6564: Cleanup for drawstyles.
- PR #6577: Fix mlab.rec_join.
- PR #6596: Added a new example to create error boxes using a PatchCollection
- PR #2370: Implement draw_markers in the cairo backend.
- PR #6599: Drop conditional import of figureoptions.
- PR #6573: Some general cleanups
- PR #6568: Add OSX to travis tests
- PR #6600: Typo: markeredgewith -> markeredgewidth
- PR #6526: ttconv: Also replace carriage return with spaces.
- PR #6530: Update make.py
- PR #6405: ToolManager/Tools adding methods to set figure after initialization
- PR #6553: Drop prettyplotlib from the list of toolkits.

- PR #6557: Merge 2.x to master
- PR #5626: New toolbar icons
- PR #6555: Fix docstrings for `warn_deprecated`.
- PR #6544: Fix typo in margins handling.
- PR #6014: Patch for issue #6009
- PR #6517: Fix conversion of string grays with alpha.
- PR #6522: DOC: made sure boxplot demos share y-axes
- PR #6529: TST Remove plt.show() from test_axes.test_dash_offset
- PR #6519: Fix FigureCanvasAgg.print_raw(...)
- PR #6481: Default boxplot style rebase
- PR #6504: Patch issue 6035 rebase
- PR #5593: ENH: errorbar color cycle clean up
- PR #6497: Line2D._path obeys drawstyle.
- PR #6487: Added docstring to scatter_with_legend.py [MEP12]
- PR #6485: Barchart demo example clean up [MEP 12]
- PR #6472: Install all dependencies from pypi
- PR #6482: Skip test broken with numpy 1.11
- PR #6475: Do not turn on interactive mode on in example script
- PR #6442: MRG: loading TCL / Tk symbols dynamically
- PR #6467: ENH: add unified seaborn style sheet
- PR #6465: updated boxplot figure
- PR #6462: CI: Use Miniconda already installed on AppVeyor.
- PR #6456: FIX: unbreak master after 2.x merge
- PR #6445: Offset text colored by labelcolor param
- PR #6417: Showraise gtk gtk3
- PR #6423: TST: splitlines in rec2txt test
- PR #6427: Output pdf dicts in deterministic order
- PR #6431: Merge from v2.x
- PR #6433: Make the frameworkpython script compatible with Python 3
- PR #6358: Stackplot weighted_wiggle zero-area fix
- PR #6382: New color conversion machinery.
- PR #6372: DOC: add whats_new for qt configuration editor.

- PR #6415: removing unused DialogLineprops from gtk3
- PR #6390: Use xkcd: prefix to avoid color name clashes.
- PR #6397: key events handler return value to True to stop propagation
- PR #6402: more explicit message for missing image
- PR #5785: Better choice of offset-text.
- PR #6302: FigureCanvasQT key auto repeat
- PR #6334: ENH: webagg: Handle ioloop shutdown correctly
- PR #5267: AutoMinorLocator and and logarithmic axis
- PR #6386: Minor improvements concerning #6353 and #6357
- PR #6388: Remove wrongly commited test.txt
- PR #6379: Install basemap from git trying to fix build issue with docs
- PR #6369: Update demo_floating_axes.py with comments
- PR #6377: Remove unused variable in GeoAxes class
- PR #6373: Remove misspelled and unused variable in GeoAxes class
- PR #6376: Update index.rst - add Windrose as third party tool
- PR #6371: Set size of static figure to match widget on hidp displays
- PR #6370: Restore webagg backend following the merge of widget nbagg backend
- PR #6366: Sort default labels numerically in Qt editor.
- PR #6367: Remove stray nonascii char from nbagg
- PR #5754: IPython Widget
- PR #6146: ticker.LinearLocator view_limits algorithm improvement closes #6142
- PR #6287: ENH: add axisbelow option 'line', make it the default
- PR #6339: Fix #6335: Queue boxes to update
- PR #6347: Allow setting image clims in Qt options editor.
- PR #6354: Update events handling documentation to work with Python 3.
- PR #6356: Merge 2.x to master
- PR #6304: Updating animation file writer to allow keywork arguments when using `with` construct
- PR #6328: Add default scatter marker option to rcParams
- PR #6342: Remove shebang lines from all examples. [MEP12]
- PR #6337: Add a 'useMathText' param to method 'ticklabel_format'
- PR #6346: Avoid duplicate cmap in image options.
- PR #6253: MAINT: Updates to formatters in `matplotlib.ticker`

- PR #6291: Color cycle handling
- PR #6340: BLD: make minimum cycler version 0.10.0
- PR #6322: Typo fixes and wording modifications (minor)
- PR #6319: Add PyUpSet as extension
- PR #6314: Only render markers on a line when markersize > 0
- PR #6303: DOC Clean up on about half the Mplot3d examples
- PR #6311: Seaborn sheets
- PR #6300: Remake of #6286
- PR #6297: removed duplicate word in Choosing Colormaps documentation
- PR #6200: Tick vertical alignment
- PR #6203: Fix #5998: Support fallback font correctly
- PR #6198: Make hatch linewidth an rcParam
- PR #6275: Fix cycler validation
- PR #6283: Use `figure.stale` instead of internal member in macosx
- PR #6247: DOC: Clarify fillbetween_x example.
- PR #6251: ENH: Added a `PercentFormatter` class to `matplotlib.ticker`
- PR #6267: MNT: trap inappropriate use of color kwarg in scatter; closes #6266
- PR #6249: Adjust test tolerance to pass for me on OSX
- PR #6263: TST: skip broken test
- PR #6260: Bug fix and general touch ups for hist3d_demo example (#1702)
- PR #6239: Clean warnings in examples
- PR #6170: getter for ticks for colorbar
- PR #6246: Merge v2.x into master
- PR #6238: Fix sphinx 1.4.0 issues
- PR #6241: Force Qt validator to use C locale.
- PR #6234: Limit Sphinx to 1.3.6 for the time being
- PR #6178: Use Agg for rendering in the Mac OSX backend
- PR #6232: MNT: use stdlib tools in allow_rasterization
- PR #6211: A method added to Colormap classes to reverse the colormap
- PR #6205: Use io.BytesIO instead of io.StringIO in examples
- PR #6229: Add a locator to AutoDateFormatters example code
- PR #6222: ENH: Added `file` keyword to `setp` to redirect output

- PR #6217: BUG: Made `setp` accept arbitrary iterables
- PR #6154: Some small cleanups based on Quantified code
- PR #4446: Label outer offset text
- PR #6218: DOC: fix typo
- PR #6202: Fix #6136: Don't hardcode default scatter size
- PR #6195: Documentation bug #6180
- PR #6194: Documentation bug fix: #5517
- PR #6011: Fix issue #6003
- PR #6179: Issue #6105: Adds targetfig parameter to the subplot2grid function
- PR #6185: Fix to csv2rec bug for review
- PR #6192: More precise choice of axes limits.
- PR #6176: DOC: Updated docs for rc_context
- PR #5617: Legend tuple handler improve
- PR #6188: Merge 2x into master
- PR #6158: Fix: pandas series of strings
- PR #6156: Bug: Fixed regression of `drawstyle=None`
- PR #5343: Boxplot stats w/ equal quartiles
- PR #6132: Don't check if in range if the caller passed norm
- PR #6091: Fix for issue 5575 along with testing
- PR #6123: docstring added
- PR #6145: BUG: Allowing unknown drawstyles
- PR #6148: Fix: Pandas indexing Error in collections
- PR #6140: clarified color argument in scatter
- PR #6137: Fixed outdated link to thirdpartypackages, and simplified the page
- PR #6095: Bring back the module level 'backend'
- PR #6124: Fix about dialog on Qt 5
- PR #6110: Fixes matplotlib/matplotlib#1235
- PR #6122: MNT: improve image array argument checking in to_rgba. Closes #2499.
- PR #6047: bug fix related #5479
- PR #6119: added comment on "usetex=False" to ainde debugging when latex not ava...
- PR #6073: fixed bug 6028
- PR #6116: CI: try explicitly including msvc_runtime

- PR #6100: Update INSTALL
- PR #6099: Fix #6069. Handle image masks correctly
- PR #6079: Fixed Issue 4346
- PR #6102: Update installing_faq.rst
- PR #6101: Update INSTALL
- PR #6074: Fixes an error in the documentation, linestyle is dash_dot and should be dashdot
- PR #6068: Text class: changed __str__ method and added __repr__ method
- PR #6018: Added get_status() function to the CheckButtons widget
- PR #6013: Mnt cleanup pylab setup
- PR #5984: Suggestion for Rasterization to docs pgf-backend
- PR #5911: Fix #5895: Properly clip MOVETO commands
- PR #6039: DOC: added missing import to navigation_toolbar.rst
- PR #6036: BUG: fix ListedColormap._resample, hence plt.get_cmap; closes #6025
- PR #6029: TST: Always use / in URLs for visual results.
- PR #6022: Make @cleanup *really* support generative tests.
- PR #6024: Add Issue template with some guidelines
- PR #5718: Rewrite of image infrastructure
- PR #3973: WIP: BUG: Convert qualitative colormaps to ListedColormap
- PR #6005: FIX: do not short-cut all white-space strings
- PR #5727: Refresh pgf baseline images.
- PR #5975: ENH: add kwarg normalization function to cbook
- PR #5931: use `locale.getpreferredencoding()` to prevent OS X locale issues
- PR #5972: add support for PySide2, #5971
- PR #5625: DOC: add FAQ about np.datetime64
- PR #5131: fix #4854: set default numpoints of legend entries to 1
- PR #5926: Fix #5917. New dash patterns. Scale dashes by lw
- PR #5976: Lock calls to latex in texmanager
- PR #5628: Reset the available animation movie writer on rcParam change
- PR #5951: tkagg: raise each new window; partially addresses #596
- PR #5958: TST: add a test for tilde in tempfile for the PS backend
- PR #5957: Win: add mgs as a name for ghostscript executable
- PR #5928: fix for latex call on PS backend (Issue #5895)

- PR #5954: Fix issues with getting tempdir when unknown uid
- PR #5922: Fixes for Windows test failures on appveyor
- PR #5953: Fix typos in Axes.boxplot and Axes.bxp docstrings
- PR #5947: Fix #5944: Fix PNG writing from notebook backend
- PR #5936: Merge 2x to master
- PR #5629: WIP: more windows build and CI changes
- PR #5914: Make barbs draw correctly (Fixes #5803)
- PR #5906: Merge v2x to master
- PR #5809: Support generative tests in @cleanup.
- PR #5910: Fix reading/writing from urllib.request objects
- PR #5882: mathtext: Fix comma behaviour at start of string
- PR #5880: mathtext: Fix bugs in conversion of apostrophes to primes
- PR #5872: Fix issue with Sphinx 1.3.4
- PR #5894: Boxplot concept figure update
- PR #5870: Docs / examples fixes.
- PR #5892: Fix gridspec.Gridspec: check ratios for consistency with rows and columns
- PR #5901: Fixes incorrect ipython sourcecode
- PR #5893: Show significant digits by default in QLineEdit.
- PR #5881: Allow build children to run
- PR #5886: Revert "Build the docs with python 3.4 which should fix the Traitlets…
- PR #5877: DOC: added blurb about external mpl-proscale package
- PR #5879: Build the docs with python 3.4 which should fix the Traitlets/IPython…
- PR #5871: Fix sized delimiters for regular-sized mathtext (#5863)
- PR #5852: FIX: create _dashSeq and _dashOfset before use
- PR #5832: Rewordings for normalizations docs.
- PR #5849: Update setupext.py to solve issue #5846
- PR #5853: Typo: fix some typos in patches.FancyArrowPatch
- PR #5842: Allow image comparison outside tests module
- PR #5845: V2.x merge to master
- PR #5813: mathtext: no space after comma in brackets
- PR #5828: FIX: overzealous clean up of imports
- PR #5826: Strip spaces in properties doc after newline.

- PR #5815: Properly minimize the rasterized layers
- PR #5752: Reorganise mpl_toolkits documentation
- PR #5788: Fix ImportError: No module named 'StringIO' on Python 3
- PR #5797: Build docs on python3.5 with linkcheck running on python 2.7
- PR #5778: Fix #5777. Don't warn when applying default style
- PR #4857: Toolbars keep history if axes change (navtoolbar2 + toolmanager)
- PR #5790: Fix ImportError: No module named 'Tkinter' on Python 3
- PR #5789: Index.html template. Only insert snippet if found
- PR #5783: MNT: remove reference to deleted example
- PR #5780: Choose offset text from ticks, not axes limits.
- PR #5776: Add .noseids to .gitignore.
- PR #5466: Fixed issue with `rasterized` not working for errorbar
- PR #5773: Fix eb rasterize
- PR #5440: Fix #4855: Blacklist rcParams that aren't style
- PR #5764: BUG: make clabel obey fontsize kwarg
- PR #5771: Remove no longer used Scikit image code
- PR #5766: Deterministic LaTeX text in SVG images
- PR #5762: Don't fallback to old ipython_console_highlighting
- PR #5728: Use custom RNG for sketch path
- PR #5454: ENH: Create an abstract base class for movie writers.
- PR #5600: Fix #5572: Allow passing empty range to broken_barh
- PR #4874: Document mpl_toolkits.axes_grid1.anchored_artists
- PR #5746: Clarify that easy_install may be used to install all dependencies
- PR #5739: Silence labeled data warning in tests
- PR #5732: RF: fix annoying parens bug
- PR #5735: Correct regex in filterwarnings
- PR #5640: Warning message prior to fc-list command
- PR #5686: Remove banner about updating styles in 2.0
- PR #5676: Fix #5646: bump the font manager version
- PR #5719: Fix #5693: Implemented is_sorted in C
- PR #5721: Remove unused broken doc example axes_zoom_effect
- PR #5664: Low-hanging performance improvements

- PR #5709: Addresses issue #5704. Makes usage of parameters clearer
- PR #5716: Fix #5715.
- PR #5690: Fix #5687: Don't pass unicode to QApplication()
- PR #5707: Fix string format substitution key missing error
- PR #5706: Fix SyntaxError on Python 3
- PR #5700: BUG: handle colorbar ticks with boundaries and NoNorm; closes #5673
- PR #5702: Add missing substitution value
- PR #5701: str.formatter invalid
- PR #5697: TST: add missing decorator
- PR #5683: Include outward ticks in bounding box
- PR #5688: Improved documentation for FuncFormatter formatter class
- PR #5469: Image options
- PR #5677: Fix #5573: Use SVG in docs
- PR #4864: Add documentation for mpl_toolkits.axes_grid1.inset_locator
- PR #5434: Remove setup.py tests and adapt docs to use tests.py
- PR #5586: Fix errorbar extension arrows
- PR #5653: Update banner logo on main website
- PR #5667: Nicer axes names in selector for figure options.
- PR #5672: Fix #5670. No double endpoints in Path.to_polygon
- PR #5553: qt: raise each new window
- PR #5594: FIX: formatting in LogFormatterExponent
- PR #5588: Adjust number of ticks based on length of axis
- PR #5671: Deterministic svg
- PR #5659: Change `savefig.dpi` and `figure.dpi` defaults
- PR #5662: Bugfix for test_triage tool on Python 2
- PR #5661: Fix #5660. No FileNotFoundError on Py2
- PR #4921: Add a quit_all key to the default keymap
- PR #5651: Shorter svg files
- PR #5656: Fix #5495. Combine two tests to prevent race cond
- PR #5383: Handle HiDPI displays in WebAgg/NbAgg backends
- PR #5307: Lower test tolerance
- PR #5631: WX/WXagg backend add code that zooms properly on a Mac with a Retina display

- PR #5644: Fix typo in pyplot_scales.py
- PR #5639: Test if a frame is not already being deleted before trying to Destroy.
- PR #5583: Use data limits plus a little padding by default
- PR #4702: sphinxext/plot_directive does not accept a caption
- PR #5612: mathtext: Use DejaVu display symbols when available
- PR #5374: MNT: Mailmap fixes and simplification
- PR #5516: OSX virtualenv fixing by creating a simple alias
- PR #5546: Fix #5524: Use large, but finite, values for contour extensions
- PR #5621: Tst up coverage
- PR #5620: FIX: quiver key pivot location
- PR #5607: Clarify error when plot() args have bad shapes.
- PR #5604: WIP: testing on windows and conda packages/ wheels for master
- PR #5611: Update colormap user page
- PR #5587: No explicit mathdefault in log formatter
- PR #5591: fixed ordering of lightness plots and changed from getting lightness …
- PR #5605: Fix DeprecationWarning in stackplot.py
- PR #5603: Draw markers around center of pixels
- PR #5596: No edges on filled things by default
- PR #5249: Keep references to modules required in pgf LatexManager destructor
- PR #5589: return extension metadata
- PR #5566: DOC: Fix typo in Axes.bxp.__doc__
- PR #5570: use base64.encodestring on python2.7
- PR #5578: Fix #5576: Handle CPLUS_INCLUDE_PATH
- PR #5555: Use shorter float repr in figure options dialog.
- PR #5552: Dep contourset vminmax
- PR #5433: ENH: pass dash_offset through to gc for Line2D
- PR #5342: Sort and uniquify style entries in figure options.
- PR #5484: fix small typo in documentation about CheckButtons.
- PR #5547: Fix #5545: Fix collection scale in data space
- PR #5500: Fix #5475: Support tolerance when picking patches
- PR #5501: Use facecolor instead of axisbg/axis_bgcolor
- PR #5544: Revert "Fix #5524. Use finfo.max instead of np.inf"

- PR #5146: Move impl. of plt.subplots to Figure.add_subplots.
- PR #5534: Fix #5524. Use finfo.max instead of np.inf
- PR #5521: Add test triage tool
- PR #5537: Fix for broken maplotlib.test function
- PR #5539: Fix docstring of violin{,plot} for return value.
- PR #5515: Fix some theoretical problems with png reading
- PR #5526: Add boxplot params to rctemplate
- PR #5533: Fixes #5522, bug in custom scale example
- PR #5514: adding str to force string in format
- PR #5512: V2.0.x
- PR #5465: Better test for isarray in figaspect(). Closes #5464.
- PR #5503: Fix #4487: Take hist bins from rcParam
- PR #5485: Contour levels must be increasing
- PR #4678: TST: Enable coveralls/codecov code coverage
- PR #5437: Make "classic" style have effect
- PR #5458: Removed normalization of arrows in 3D quiver
- PR #5480: make sure an autoreleasepool is in place
- PR #5451: [Bug] masking of NaN Z values in pcolormesh
- PR #5453: Force frame rate of FFMpegFileWriter input
- PR #5452: Fix axes.set_prop_cycle to handle any generic iterable sequence.
- PR #5448: Fix #5444: do not access subsuper nucleus _metrics if not available
- PR #5439: Use DejaVu Sans as default fallback font
- PR #5204: Minor cleanup work on navigation, text, and customization files.
- PR #5432: Don't draw text when it's completely clipped away
- PR #5426: MNT: examples: Set the aspect ratio to "equal" in the double pendulum animation.
- PR #5214: Use DejaVu fonts as default for text and mathtext
- PR #5306: Use a specific version of Freetype for testing
- PR #5410: Remove uses of font.get_charmap
- PR #5407: DOC: correct indentation
- PR #4863: [mpl_toolkits] Allow "figure" kwarg for host functions in parasite_axes
- PR #5166: [BUG] Don't allow 1d-arrays in plot_surface.
- PR #5360: Add a new memleak script that does everything

- PR #5361: Fix #347: Faster text rendering in Agg
- PR #5373: Remove various Python 2.6 related workarounds
- PR #5398: Updating 2.0 schedule
- PR #5389: Faster image generation in WebAgg/NbAgg backends
- PR #4970: Fixed ZoomPanBase to work with log plots
- PR #5387: Fix #3314 assert mods.pop(0) fails
- PR #5385: Faster event delegation in WebAgg/NbAgg backends
- PR #5384: BUG: Make webagg work without IPython installed
- PR #5358: Fix #5337. Turn off –no-capture (-s) on nose
- PR #5379: DOC: Fix typo, broken link in references
- PR #5371: DOC: Add what's new entry for TransformedPatchPath.
- PR #5299: Faster character mapping
- PR #5356: Replace numpy funcs for scalars.
- PR #5359: Fix memory leaks found by memleak_hawaii3.py
- PR #5357: Fixed typo
- PR #4920: ENH: Add TransformedPatchPath for clipping.
- PR #5301: BUG: Dot should not be spaced when used as a decimal separator
- PR #5103: Add option to package DLL files
- PR #5348: windows dlls packaging
- PR #5346: Make sure that pyparsing 2.0.4 is not installed.
- PR #5340: Improve compatibility for h264 ffmpeg-encoded videos.
- PR #5295: Reduce number of font file handles opened
- PR #5330: Reduce dupe between tests.py and matplotlib.test
- PR #5324: Fix #5302: Proper alpha-blending for jpeg
- PR #5339: PEP8 on Python 3.5
- PR #5215: TST: drop py2.6 & py3.3 testing
- PR #5313: Fix the minortick-fix
- PR #5333: Patch 2
- PR #5276: Use lock directory to prevent race conditions
- PR #5322: Fix #5316: Remove hardcoded parameter from barh doc
- PR #5300: Fixed compiler warnings in _macosx.m
- PR #5304: Prelimiary fix for Mac OSX backend threading issues

- PR #5297: BUG: recent numpy fails on non-int shape
- PR #5283: Make new colormaps full-fledged citizens
- PR #5296: Fix STIX virtual font entry for M script character
- PR #5285: Fix some compiler warnings
- PR #5288: Doc build fixes
- PR #5289: Fix IndexError in cursor_demo.py.
- PR #5290: implemeted get_ticks_direction()
- PR #4965: WIP: Add new Colormaps to docs
- PR #5284: New Colormaps to docs
- PR #4329: Write status message in single line in Qt toolbar.
- PR #3838: Fix units examples under python3
- PR #5279: On Windows, use absolute paths to figures in Sphinx documents if necessary
- PR #5274: Check dimensions of arrays passed to C++, handle 0 dimensions
- PR #5273: Provide message if test data is not installed
- PR #5268: Document and generalise $MATPLOTLIBRC
- PR #4898: HostAxesBase now adds appropriate _remove_method to its parasite axes.
- PR #5244: Matlab Style Label Warns In Test
- PR #5236: DOC: tweak README formatting
- PR #5228: Remove mentions of SourceForge
- PR #5231: include links to the mailing list in the README
- PR #5235: Add link to "mastering matplotlib" book
- PR #5233: Skip over broken TTF font when creating cache
- PR #5230: Fix casting bug in streamplot
- PR #5177: MAINT: dviread refactoring
- PR #5223: Update dateutil URL.
- PR #5186: DOC: Fix docstrings for multiple parameters
- PR #5217: Fix PathEffect rendering on some backends
- PR #5216: Enable testing without internet access.
- PR #5183: TST: fix `AttributeError: 'module' object has no attribute 'nl_langinfo'` on Windows
- PR #5203: Fix mathtext_wx example not redrawing plots
- PR #5039: sphinxext pot_directive: more robust backend switching

- PR #4915: TransformWrapper pickling fixes
- PR #5170: [MAINT] Add symlog locator to __all__ and to the docs
- PR #5207: V1.5.x
- PR #5021: Use json for the font cache instead of pickle
- PR #5184: TST: fix test_mlab.test_griddata_nn failures on Windows
- PR #5182: Fix `ValueError: invalid PNG header` on Windows
- PR #5189: DOC: Fix encoding for LaTeX
- PR #5178: DOC: Fix description of draw_markers in api_changes.rst
- PR #5147: Cleaned up text in pyplot_tutorial.rst
- PR #5171: Fix exception with Pillow 3
- PR #5153: MNT: more minor tweaks to qt_compat.py
- PR #5167: [BUG] symlog support for ax.minorticks_on()
- PR #5168: Fix a bounds check
- PR #5108: added None option to _get_view, also fixed a typo
- PR #5106: FIX: array_view construction for empty arrays
- PR #5157: Update MEP12.rst
- PR #5127: mep12 on cursor_demo.py
- PR #5154: TST: use patched nose for py3.6 compat
- PR #5150: FIX: set internal flags first in FigureCanvasBase
- PR #5134: qt imports fix
- PR #5080: Try to make backend_gdk compatible with numpy 1.6
- PR #5148: FIX: scatter accepts 2-D x, y, c; closes #5141
- PR #5138: MAINT: use travis wheel repository for 3.5 build
- PR #5129: FIX: be more careful about import gobject
- PR #5130: DOC: add API notes for jquery upgrade
- PR #5133: DOC: Update polar examples to use projection kwarg
- PR #5091: Upgrade jquery and jquery-ui
- PR #5110: Travis: Update Python to 3.5 final
- PR #5126: mep12 on customize_rc.py
- PR #5124: mep12 on ellipse_rotated.py
- PR #5125: mep12 on ellipse_demo.py
- PR #5123: mep12 on errorbar_limits.py

- PR #5117: mep12 on fill_spiral.py
- PR #5118: mep12 on figure_title.py
- PR #5116: Mep12 fonts table ttf.py
- PR #5115: mep12 on fonts_demo.py
- PR #5114: BLD: setup.py magic to get versioneer to work
- PR #5109: Fix for bug in set_cmap in NonUniformImage
- PR #5100: The Visual C++ Redistributable for Visual Studio 2015 is required for Python 3.5
- PR #5099: Fix corrupted stix_fonts_demo example
- PR #5084: Fix segfault in ft2font
- PR #5092: Generate reversed ListedColormaps
- PR #5085: corrected doc string
- PR #5081: Add WinPython and Cycler to installation instructions for Windows
- PR #5079: Improve whats new
- PR #5063: added tick labels from values demo
- PR #5075: mep12 on fonts_demo_kw.py
- PR #5073: DOC: updated documented dependencies
- PR #5014: Add Travis job with 3.6 nightly
- PR #5071: Fix URLError: <urlopen error unknown url type: c> on Windows
- PR #5070: Bugfix for TriAnalyzer mismatched indices, part 2
- PR #5072: Fix backend_driver.py fails on non-existent files
- PR #5069: Typos in api_changes and whats_new
- PR #5068: Fix format string for Python 2.6
- PR #5066: Doc merge whatsnew apichanges
- PR #5062: Fix for issue4977 mac osx
- PR #5064: Use versioneer for version
- PR #5065: Bugfix for TriAnalyzer mismatched indexes
- PR #5060: FIX: add check if the renderer exists
- PR #4803: Fix unit support with `plot` and `pint`
- PR #4909: figure option dialog does not properly handle units
- PR #5053: Unpack labeled data alternative
- PR #4829: ENH: plotting methods can unpack labeled data
- PR #5044: Added PDF version of navigation icons

- PR #5048: Test with 3.5rc4
- PR #5043: resize_event not working with MacOSX backend
- PR #5041: mep12 on ganged_plots.py
- PR #5040: mep12 on ginput_demo.py
- PR #5038: PRF: only try IPython if it is already imported
- PR #5020: mathtext: Add – to spaced symbols, and do not space symbols at start of string
- PR #5036: Update what's new for RectangeSelector
- PR #3937: Rectangle Selector Upgrade
- PR #5031: support subslicing when x is masked or has nans; closes #5016
- PR #5025: [MRG] ENH Better error message when providing wrong fontsizes
- PR #5032: ENH: More useful warning about locale errors
- PR #5019: locale.getdefaultlocale() fails on OS X
- PR #5030: mep12 on geo_demo.py
- PR #5024: FIX
- PR #5023: Fix Agg clipping
- PR #5017: MEP22 warnings
- PR #4887: FIX: mathtext accents
- PR #4995: animation fixes
- PR #4972: Qt5: Move agg draw to main thread and fix rubberband
- PR #5015: Fix the fontdict parameter in set_xticklabels/set_yticklabels
- PR #5009: TST: bump python 3.5 version to rc2
- PR #5008: fix #5007
- PR #4807: setupext.py: let the user set a different pkg-config
- PR #5010: DOC: Add information on new views for custom Axes.
- PR #4994: Fix syntax error
- PR #4686: [WIP] Property Cycling
- PR #5006: fix bug
- PR #4795: ENH: Add API to manage view state in custom Axes.
- PR #4924: MNT: changed close button color and text
- PR #4992: showpage at the end of .eps files
- PR #4991: FIX: double z-axis draw in mplot3D
- PR #4988: BUG: in ScalarFormatter, handle two identical locations; closes #4761

- PR #4873: mathtext: Finetuning sup/super block to match TeX reference
- PR #4985: Fix for #4984
- PR #4982: Mep12 hist2d log demo.py
- PR #4981: Mep12 image demo2.py
- PR #4980: Mep12 image interp.py
- PR #4983: MEP12 on hist2d_demo.py
- PR #4942: text update properties does not handle bbox properly
- PR #4904: position of text annotations looses unit information
- PR #4979: PY2K : in python2 lists don't have copy method
- PR #4689: Update to score_family in font_manager.py
- PR #4944: qt backend draw_idle doesn't work
- PR #4943: qt backend has more draws than necessary
- PR #4969: FIX: account for None in Line2D.axes setter
- PR #4964: Clarify what "axes" means
- PR #4961: Bounds checking for get_cursor_data(). Closes #4957
- PR #4963: Grammar fix for pyplot tutorial
- PR #4958: BUG: allow facecolors to be overridden in LineCollection
- PR #4959: Fix link in documentation. Closes #4391.
- PR #4956: MEP12 on image masked.py
- PR #4950: Mep12 image origin.py
- PR #4953: Make sure that data is a number before formatting. Fix for #4806
- PR #4948: Mep12 layer images.py
- PR #4949: Mep12 invert axes.py
- PR #4951: FIX: argument order in RendereAgg.restore_region
- PR #4945: qt backend default bbox not set when blitting
- PR #4456: FIX : first pass at fixing nbagg close issue
- PR #4939: NBAgg: fix Jupyter shim warning
- PR #4932: MEP12 on load_converter.py
- PR #4935: Add api change note about lena removal
- PR #4878: PRF: only check some artists on mousemove
- PR #4934: Colormep12rebase
- PR #4933: MEP12 on line_collection2.py

- PR #4931: MEP12 on loadrec.py
- PR #4929: Correct numpy doc format in cbook api docs
- PR #4928: remove lena images
- PR #4926: Mep12 log test.py
- PR #4925: Make sure _edgecolors is a string before comparison to string.
- PR #4923: modifying sourceforge links
- PR #4738: MNT: overhaul stale handling
- PR #4922: DOC: update qt related prose
- PR #4669: Creation of the 'classic' matplotlib style
- PR #4913: Agg restore_region is broken
- PR #4911: Super short lines with arrows do not act well
- PR #4919: Issue08
- PR #4906: broken_barh does not properly support units
- PR #4895: Add latex preamble to texmanager _fontconfig
- PR #4816: FIX: violinplot crashed if input variance was zero
- PR #4890: Reduce redudant code in axes_grid{,1}.colorbar
- PR #4892: Fix single-shot timers in nbagg backend
- PR #4875: FIX: add explict draw_if_interactive in figure()
- PR #4885: changed a pylab reference
- PR #4884: mep12 on manual_axis.py
- PR #4899: Replace kwdocd in docs with docstring.interpd/dedent_interpd
- PR #4894: Qt5: Eliminate slow path when showing messages
- PR #4824: Two bugs in colors.BoundaryNorm
- PR #4876: Create a temporary bitmap context if needed
- PR #4881: mep12 on matplotlib_icon.py
- PR #4882: mep12 on masked_demo.py
- PR #4844: Avoid possible exception when toggling full-screen
- PR #4843: Rev coord wrapping
- PR #4542: Fix cairo graphics context
- PR #4743: BUG: Fix alternate toolbar import on Python 3.
- PR #4870: mep12 on matshow.py
- PR #4871: mep12 on mri_demo.py

- PR #4846: mep12 on plotfile_demo.py
- PR #4868: mep12 on multiline.py
- PR #4861: mep12 on multiple_figs_demo.py
- PR #4845: mep12 on print_stdout.py
- PR #4860: Document get_cachedir() in troubleshooting
- PR #4833: mep12 on quiver_demo.py
- PR #4848: Mep12 newscalarformatter demo.py
- PR #4852: Null strides wireframe
- PR #4588: FIX: re-order symbol and acent in mathtext
- PR #4800: Fixes to funcanimation
- PR #4838: scale descent back
- PR #4840: Improve error when trying to edit empty figure.
- PR #4836: mep12 on psd_demo.py
- PR #4835: Calculate text size and descent correctly
- PR #4831: mep12 changes to axes_props.py
- PR #4834: Test on Python 3.5 beta4
- PR #4832: mep12: changed pylab to pyplot
- PR #4813: Prf mouse move hitlist
- PR #4830: mep12 on axes_demo.py
- PR #4819: mep12 on pstest.py
- PR #4817: mep12 on log_bar.py
- PR #4820: mep12 on arctest.py
- PR #4826: mep12 on image_demo2.py
- PR #4825: Remove trailing zeroes in path string output
- PR #4818: Mep12 logo.py
- PR #4804: BUG: Fix ordering in radar chart example.
- PR #4801: Travis switch from nightly to 3.5 beta
- PR #4811: nan_test.py mep12
- PR #4771: NF - New legend example with line collection
- PR #4798: Fix msvc14 compile errors
- PR #4805: Axes3d doc typo
- PR #4797: remove empty constuctor

- PR #4785: Animation conversion to HTML5 video
- PR #4793: Added code information to Poly3DCollection
- PR #4790: Test Cleanup Closes #4772
- PR #4778: FIX: remove equality check in line2D.set_color
- PR #4777: mep12 on pythonic_matplotlib.py
- PR #4776: mep12 on scatter_masked.py
- PR #4707: ENH: Add newly proposed colormaps
- PR #4768: ENH: add remove call back to axes
- PR #4766: FIX: fix python2 unicode compatibility
- PR #4763: Return from draw_idle as soon as possible
- PR #4718: Expose interpolation short names at module level.
- PR #4757: Use BytesIO from io.
- PR #4752: FIX: cast input to Rectangle to float
- PR #4605: ENH: Use png predictors when compressing images in pdf files
- PR #4178: Annotation: always use FancyBboxPatch instead of bbox_artist
- PR #3947: Date fixes
- PR #4433: ENH : stepfill between
- PR #4733: Backport #4335 to master
- PR #4612: Only use asynchronous redraw methods when handling GUI events in Qt5Agg (fix #4604)
- PR #4719: ENH: add inverse function to _deprecated_map
- PR #4727: FIX: fix afm + py3k + logscale
- PR #4747: Added mplstereonet blurb to mpl_toolkits listing
- PR #4646: MEP12 on tex_unicode_demo.py
- PR #4631: Standardized imports
- PR #4734: mep12 on scatter_profile.py
- PR #4664: MEP12 on axis_equal_demo.py
- PR #4660: MEP12-on-arrow_demo.py
- PR #4657: MEP12-on-anscombe.py
- PR #4663: MEP12 on axes_props.py
- PR #4654: MEP12 on annotation_demo.py
- PR #4726: DOC: whats_new for axes.labelpad
- PR #4739: MNT: Remove unused code in pdf backend

- PR #4724: DOC: slightly update demo
- PR #4731: Implement draw_idle
- PR #3648: dates.YearLocator doesn't handle inverted axes
- PR #4722: STY: pep8 that slipped by the tests
- PR #4723: Travis: Revert to using tests.py. Temp fix for #4720
- PR #4721: CLN: remove unused code path
- PR #4717: BUG: when autoscaling, handle tiny but non-zero values; closes #4318
- PR #4506: Enh python repl rd2
- PR #4714: Add an option to streamplot to manually specify the seed points.
- PR #4709: FIX: update scale on shared axes
- PR #4713: API/CLN: remove threading classes from cbook
- PR #4473: ENH: property if DrawingArea clips children
- PR #4710: FIX: gracefully deal with empty size lists
- PR #4593: FIX: Correct output of mlab._spectral_helper when scale_by_freq=False
- PR #4708: Travis: Set exit to true in nose.main
- PR #4701: minor typo in docstring
- PR #4677: Set figure width and height with set_size_inches
- PR #4684: MEP12 on set_and_get.py
- PR #4683: MEP12 on stix_fonts_demo.py
- PR #4668: Remove test dependencies from install_requires
- PR #4687: Travis: Upgrade pip and setuptools
- PR #4685: MEP12-on-barchart_demo2.py
- PR #4682: Mods to documentation.
- PR #4218: Addition of RC parameters
- PR #4659: Mep12 shared to spectrum
- PR #4670: Mep12 usetex
- PR #4647: Be more correct when validating bbox rc params
- PR #4639: MEP12 on transoffset.py
- PR #4648: MEP12 on system_monitor.py
- PR #4655: Mep12 step demo.py
- PR #4656: Mep12 spine to stem
- PR #4653: MEP12 on alignment_test.py

- PR #4652: Mep12 stock demo.py
- PR #4651: Mep12 subplot toolbar.py
- PR #4649: MEP12 changes on symlog_demo.py
- PR #4645: MEP12 on text_handles.py
- PR #4611: Add % bachelors degrees plot example
- PR #4667: Install latest version of mock on python 2.7
- PR #4644: MEP12 on text_rotation.py
- PR #4650: MEP12 on subplots_adjust.py
- PR #4640: MEP12 on toggle_images.py
- PR #4643: MEP12 on text_rotation_relative_to_line.py
- PR #4641: MEP12 on to_numeric.py
- PR #4630: MEP12 pylab changes on zorder_demo.py
- PR #4635: MEP12 on tricontour_vs_griddata.py
- PR #4665: PEP8 fix usetex_fonteffects
- PR #4662: usetex_fonteffects.py: Import matplotlib here as needed
- PR #4637: MEP12 on tricontour_smooth_user.py
- PR #4583: Mnt mailmap
- PR #4642: Fixed and classified equal_aspect_ratio.py
- PR #4632: Changed pylab to plt.
- PR #4629: translated pylab import to plts
- PR #4634: MEP12 changes to use_tex_baseline_test.py
- PR #4627: Reclassify contourf log.py
- PR #4626: In coutourf_log.py, changed `P.` to `plt.`
- PR #4623: Provide std::isfinite for msvc
- PR #4624: Fix segfault on Windows
- PR #4617: Fix for issue 4609
- PR #4608: Axes.hist: use bottom for minimum if log and histtype='step...'
- PR #4618: swap standard deviations so that men's means are shown with men's std…
- PR #4616: Explicitly install Mock at version 1.0.1
- PR #4610: MNT: Replace outdated comment with self-explaining code (hatching in pdf backend)
- PR #4603: MNT: Minor cleanups in the pdf backend and related files
- PR #4601: FIX: handle empty legend in qt figureoption

- PR #4589: Add separate drawstyles options to Qt figureoptions dialog
- PR #4547: FIX: accept non-ascii in dvipng –version output
- PR #4595: Fix alpha channels in PDF images
- PR #4591: _create_tmp_config_dir() "mkdirs" the returned dir
- PR #4596: Add remaining seaborn style sheets
- PR #4594: Revert "WX Monkey patch ClientDC for name changes"
- PR #4586: BUG: respect alpha in RGBA markeredgecolor; closes #4580
- PR #4570: Add Seaborn style sheets; addresses #4566
- PR #4587: DOC: clairify auto-level behavior
- PR #4544: MNT: Deprecate idle_event and remove it from all but wx backends
- PR #4522: type1font.py fixes and test case
- PR #4578: Fixed typo in docstring #4562
- PR #4564: DOC/MNT: Throwing some docstrings at axes_rgb.py
- PR #4565: DOC: clean up rst in whats_new folder
- PR #4572: FIX: remove unicode in wx_compat
- PR #4571: Don't ignore the `fig` arg in demo code
- PR #4569: FIX: sign is not defined
- PR #4503: Fix draw on show
- PR #4551: %s -> %r else if invalid char unable to print error
- PR #4554: A few WX phoenix related changes
- PR #4555: Avoid making nose a dependency for matplotlib.testing.compare
- PR #4553: BUG fix: prevent 2D axis from showing up after calling Axes3D.cla()
- PR #3602: Add rcParams support for markers' fillstyle prop
- PR #4499: Jklymak colormap norm examp
- PR #3518: Left ventricle bull eye
- PR #4550: Doc AHA bullseye
- PR #4527: Use C++ stdlib for isfinite etc.
- PR #2783: Use metric identifiers to parse an AFM character metric line
- PR #4548: qt_compat: supply more helpful message when no pyqt or pyside is found
- PR #4541: Directly link matplotlib.org and not sourceforge.net
- PR #4530: Get rid of annoying border for Tk Canvases
- PR #3242: DateFormatter shows microseconds instead of %f for years <= 1900

- PR #4153: bytes2pdatenum
- PR #4535: FIX: move non-finite position check in text.draw
- PR #4208: Fix compression of grayscale rasterized images when using (e)ps distilled with xpdf.
- PR #4533: Revert "made idle_event() in backend_bases.py return True"
- PR #4163: Fix #4154: Return a writable buffer from conv_color
- PR #4310: Square plots
- PR #4449: capsize with default in matplotlibrc
- PR #4474: Possible fix for hatching problems inside legends (PDF backend)
- PR #4524: CLN: explicitly cast (void *) -> (char *)
- PR #4519: Removing intel preprocessors from qhull_a.h
- PR #4521: Raise more useful error when tfm file is missing
- PR #4477: OffsetBoxes now considered by tight_layout
- PR #4426: FIX : hide ref counting violence unless needed
- PR #4408: Fix path length limit
- PR #4510: Try expanding user for _open_file_or_url.
- PR #4256: Allow URL strings to be passed to imread
- PR #4508: DOC: "Customizing matplotlib" should mention style sheets
- PR #4481: Rasterize colorbar when it has many colors; closes #4480
- PR #4505: Added reference to the Matplotlib-Venn package
- PR #4497: Add link to new book
- PR #4494: Returning the Poly3DCollection when calling bar3d
- PR #4452: Fix for issue4372
- PR #4483: BUG: Do not correct orientation of triangles returned by Qhull (master)
- PR #4479: Problems with mpl.pyplot
- PR #4466: Clipping for OffsetBoxes
- PR #4091: ENH : add function to add displayhook
- PR #4471: Minor improvements to the docstring of `step`.
- PR #4393: Fix Line2D function set_markersize so it doesn't fail if given a string ...
- PR #3989: Allow Artists to show pixel data in cursor display
- PR #4459: Downscale iterm2 backend example image in matplotlib toolkit docs.
- PR #4458: Raise missing ValueError in transform_angles
- PR #3421: make wx backends compatible with wxPython-Phoenix

- PR #4455: Fix csv2rec for passing in both names and comments.
- PR #4342: Implementation of Issue #3418 - Auto-wrapping text
- PR #4435: MRG: use travis wheels for dependencies
- PR #4441: Mentioned iTerm2 external backend in mpl_toolkit docs.
- PR #4439: Import cbook.restrict_dict into backend_gdk
- PR #4436: Travis, remove quite and verbose from nosetest flags
- PR #3834: Remove lod
- PR #4014: Fix Axes `get_children` order to match `draw` order
- PR #4427: DOC : revert some documentation changes from #3772
- PR #3772: Allow both linestyle definition "accents" and dash-patterns as linestyle
- PR #4411: improvements to qt edit widget
- PR #4422: FIX : turn path snapping off on 'o' marker path
- PR #4423: TST : suppress all of the success messages
- PR #4401: Fix #4333: Whitespace after sub/super cluster
- PR #4350: Sets additional default values for axes and grid.
- PR #4377: Memory leak for Cursor useblit=True on PySide/Python3
- PR #4399: Enable travis tests on nightly python version (3.5 alpha)
- PR #4398: Remove unnecessary pyplot import from axes_grid1
- PR #4395: Travis docs fixes
- PR #4355: TST : first pass updating to use travis containers
- PR #4358: cbook.is_sequence_of_strings knows string objects
- PR #4388: BUG : fix svg corner case
- PR #4381: Legend rcparams doc tests
- PR #4370: DOC: cp missing `manage_xticks` from `bxp` to `boxplot` docstring [backport]
- PR #4356: STY: update example with preferred plt.subplots()
- PR #4361: STY: update with use of plt.subplots(), other readability edits
- PR #4362: fix rcParams legend.facecolor and edgecolor never being used
- PR #4357: Change documentation of legend to reflect default upper-right
- PR #4193: BUG/API : fix color validation
- PR #4345: DOC : document exact freetype versions for tests
- PR #4259: Implementation of Issue #4044. Added ScientificTable and ScientificCell subclasses.
- PR #4228: BUG : fix non-uniform grids in pcolorfast

- PR #4352: API/FIX : don't accept None for x or y in plot
- PR #4311: BUG : bbox with any nan points can not overlap
- PR #4265: DOC/API : StrMethodFormatter
- PR #4343: decode the execution path string based file system encoding
- PR #4351: STY: update example with preferred plt.subplots
- PR #4348: Reorder the code in the draw() method of Line2D to fix issue 4338
- PR #4347: DOC: delete the repetitive word 'the' in docstrings and comments
- PR #4298: Prevent 'color' argument to eventplot from overriding 'colors' kwarg (fixes #4297)
- PR #4330: Add tick_values method to the date Locators
- PR #4327: Fix lw float cast
- PR #4266: Add functionality to plot bar and barh with string labels (Implement #2516)
- PR #4225: Provide way to disable Multi Cursor (Implement #2663)
- PR #4274: Fix Angstrom issues
- PR #4286: Added native dpi option for print_figure
- PR #4312: Some fixes to qt 4 and 5 examples
- PR #4315: added resize parameter to plot 2d-arrays using figimage
- PR #4317: DOC: Note about pixel placement in imshow
- PR #3652: MEP22: Navigation by events
- PR #4196: DOC/TST : document and test negative width to bar
- PR #4291: Add note about nbagg middle click button
- PR #4304: Labels do not becomes color anymore in figure options panel for qt toolb...
- PR #4308: fixes #2885, #3935, #3693, for hatched fill
- PR #4305: Improve error message when freetype headers are not found using python3
- PR #4300: Fix #4299: Add support for leftVert etc.
- PR #4293: Massive MEP move
- PR #4119: Fix ValueError being raised when plotting hist and hexbin on empty dataset (Fix #3886)
- PR #4249: DOC : start to move MEP to docs
- PR #4278: Replace use of str() with six.text_type() for Py2&3 compatibility [backport to color_overhaul]
- PR #4264: Fix for unpickling polar plot issue #4068
- PR #4267: correct rst syntax for code blocks
- PR #4263: Py26 format

- PR #3060: converted assert into exception
- PR #4261: STY: update example with preferred plt.subplots
- PR #4250: BUG: Quiver must copy U, V, C args so they can't change before draw()
- PR #4254: Minor typo fix.
- PR #4248: backend_pgf: don't clip filled paths (fixes #2885, #3935, #3693)
- PR #4236: multiple canvas support for Windows
- PR #4244: Fix #4239: Don't include scientific notation in path strings
- PR #4234: Added mock, coverage and pep8 dep. Added pep8 options
- PR #4233: Fix small option for docs build with sphinx 1.3
- PR #4221: Suggest non-existing default filename (Implement #3608)
- PR #4231: Fix #4230: Don't overflow buffer with sketch path.
- PR #4224: DOC : update testing docs
- PR #4229: Bug in ParseTuple for PyQuadContourGenerator_init
- PR #4226: Refactoring: fewer variables, slightly faster code
- PR #4220: Add rcParams to enable/disable minor ticks on axes separately issue #3024
- PR #4219: Implemented new feature for Issue #2880
- PR #4197: Generate path strings in C++ for PDF and PS
- PR #4113: forcing weight to int
- PR #3985: Widget and animation improvements
- PR #4203: DOC: Colormap synonyms in examples, fix errors caused by removing duplicates
- PR #4118: CallbackRegistry fix
- PR #4134: Axis Labels with offset Spines
- PR #4173: Fix for issue #3930:ConnectionPatch with fancy arrow of length zero produces no plot
- PR #4182: colorbar: edit tick locations based on vmin and vmax; closes #4181
- PR #4213: Fix test docs build on Travis with Sphinx 1.3.0 Edit (Lock travis on 1.2.3 for now)
- PR #4075: backend_cairo: Clip drawn paths to context.clip_extents()
- PR #4209: More updates on dead URLs
- PR #4206: Fix C++ warnings from latest clang-analyzer
- PR #4204: Updated links in INSTALL
- PR #4201: Bug in text draw method when path_effects are set
- PR #4191: Adding 'api_changes' and 'whats_new' docs for PR #4172
- PR #4198: Plot: convert 'c' to 'color' immediately; closes #4162, #4157 [backport to color_overhaul]

- PR #4061: Allow users to decide whether a vector graphics backend combines multiple images into a single image
- PR #4186: Close clipped paths
- PR #4172: axes.locator_params fails with LogLocator (and most Locator subclasses) #3658
- PR #3753: Logit scale
- PR #4171: set `fig.waiting = false` when image data is received [backport to color_overhaul]
- PR #4165: Make _is_writable_dir more flexible to obscure failure modes
- PR #4177: MNT : fix typo in no-lint flag
- PR #4149: Clean up matplotlib.colors
- PR #4155: Various pep8 fixes - specifically targeting files which are failing travis pep8 tests
- PR #4159: ENH better error message for wrong fontsize
- PR #4176: Fix Travis building of docs with IPython 3
- PR #3787: Refactors axis3d.py to address issue #3610
- PR #4174: ENH: speed-up mlab.contiguous_regions using numpy
- PR #4166: Ensure the gc module is available during interpreter exit
- PR #4170: Travis: Commit docs on top of first_commit
- PR #4164: Fix Gtk3 Backend Source ID was not found
- PR #4158: Ensure that MPL_REPO_DIR is set on Travis
- PR #4150: Travis syntax
- PR #4151: BUG: fix bad edits to travis.yml file
- PR #4148: Fix mathtext image bounding box
- PR #4138: TST: trigger travis OSX tests if Linux tests pass
- PR #3874: New C++ contour code with corner_mask kwarg
- PR #4144: Fix for issue 4142: Let show() exit the run loop after all windows are closed in a non-interactive session
- PR #4141: Modify set_ticklabels() to fix counterintuitive behavior of set_ticklabels(get_ticklabels)#2246
- PR #3949: PEP8: adjust some long lines
- PR #4130: Qt event fix
- PR #3957: Corrected cax attributes of ImageGrid axes
- PR #4129: MNT : fix text-based text with new advance-width
- PR #4084: Updated some broken and outdated links in testing docs [backport 1.4.2-doc]
- PR #4093: Gtk.main_iteration takes no arguments

- PR #4031: Font advance width
- PR #4079: scatter: fix marker kwarg bug. Closes #4073, #3895.
- PR #4123: Link fix in external ressources + 1 addition
- PR #4121: added guiEvent to PickEvent
- PR #4116: DOC: Correct docstring typo in subplot2grid
- PR #4100: Add guiEvent handling for web backends
- PR #4104: Pep8 fixes
- PR #4097: Fix scale factor label issue #4043
- PR #4101: Add guiEvent data to Qt backend
- PR #4096: Fix minor typo in artist tutorial
- PR #4089: Fix #4074: Bug introduced in 91725d8
- PR #4087: Fix #4076. Change how result is stored in point_in_path/point_on_path.
- PR #4006: Allow interrupts to be delivered once Python is fixed.
- PR #3994: Add per-page pdf notes in PdfFile and PdfPages.
- PR #4080: test_axes: remove extraneous "show()"
- PR #4081: Pep8 version fixes
- PR #3992: Code removal
- PR #4039: added some fixes in order to use the result obtained from `mpl._get_configdir()` [backport to 1.4.2-doc]
- PR #4050: Fix masked array handling
- PR #4051: Correct FA 4 name of Download icon
- PR #4041: Prevent Windows from opening command prompt (#4021) [backport to 1.4.x]
- PR #4032: Disable context menu in webagg
- PR #4029: Fix key modifier handling in Web backends [backport 1.4.x]
- PR #4035: FIX: resizing a figure in webagg
- PR #4034: quiver: always recalculate in draw(); improve docstring; closes #3709, #3817 [backport to 1.4.x]
- PR #4022: More helpful error message for pgf backend
- PR #3997: Change documented "Optional" ScaleBase method to "Required"
- PR #4009: Fix name of variable in doc string
- PR #4005: Try to fix mencoder tests. [backport to 1.4.x]
- PR #4004: Provide arguments to mencoder in a more proper way
- PR #4002: fix find_output_cell for IPython >= 3.0 [backport to 1.4.x]

- PR #3995: Fix wx._core.PyAssertionError ... wxGetStockLabel(): invalid stock item ID
- PR #3974: Add Save Tool to NbAgg Figure [backport to 1.4.x]
- PR #3676: Fix #3647 [backport to 1.4.x]
- PR #3968: Add Support for `scroll_event` in WebAgg and NbAgg [backport to 1.4.x]
- PR #3965: Js fixes for key events + ipython notebooks
- PR #3993: Fix stupid typo
- PR #3939: Deploy development documentation from Travis [not ready to merge]
- PR #3988: MNT : deprecate FigureCanvasBase.onHilite
- PR #3982: pgf can not write to `BytesIO` [back port to 1.4.x]
- PR #3971: Added "val" attribute to widgets.RadioButtons
- PR #3981: Fixes for File Saving in Webagg
- PR #3978: Fix clipping/zooming of inverted images
- PR #3970: Add Figure Enter/Leave Events to Webagg
- PR #3969: Connect the Resize Event for WebAgg
- PR #3967: FIX: Webagg `save_figure` - Raise a Warning Instead of an Error
- PR #3916: RF: always close old figure windows
- PR #3958: Suppress some warnings in examples
- PR #3831: Fix python3 issues in some examples
- PR #3612: Minor tick fix [backport to 1.4.x]
- PR #3943: Legend deprecate removal + cleanup
- PR #3955: API : tighten validation on pivot in Quiver
- PR #3950: Ensure that fonts are present on travis when building docs.
- PR #3883: BUG/API : relax validation in hist
- PR #3954: Simplify set_boxstyle Accepts section of FancyBboxPatch
- PR #3942: MNT : slight refactor of Axis.set_ticklabels
- PR #3924: Fix PEP8 coding style violations
- PR #3941: Change name of dev version
- PR #3925: Text.{get,set}_usetex: manually enable/disable TeX
- PR #3933: Fix minor typo in docs: s/right/left/
- PR #3923: Fixed PEP8 coding style violations
- PR #3835: Single axes artist
- PR #3868: Ensure that font family is unicode

- PR #3893: Don't close GzipFile before it is used
- PR #3850: FIX str.decode in python2.6 does not take keyword arguments [backport to 1.4.x]
- PR #3863: Fix log transforms (fixes #3809) [back port to 1.4.x]
- PR #3888: Update collections.py
- PR #3885: Fix indentation
- PR #3866: Regression in transforms: raises exception when applied to single point
- PR #3196: Issue with iterability of axes arguments [backport to 1.4.x]
- PR #3853: typeFace as bytestring in Py3
- PR #3861: Added missing implementation of get_window_extent for AxisImage and test (fixes #2980).
- PR #3845: BUG: non integer overlap might lead to corrupt memory access in as_strided [backport 1.4.x]
- PR #3846: wrong method name
- PR #3795: RcParams instances for matplotlib.style.use
- PR #3839: backend_wx: delete remaining lines for removal of printer support
- PR #3832: Remove deprecated nonorm and normalize
- PR #3402: Image tutorial notebook edit
- PR #3830: Merge of #3402
- PR #3824: Path.contains_points() returns a uint8 array instead of a bool array
- PR #2743: Updated the macosx backed figure manager show function to bring the
- PR #3812: insert deprecation warning for set_graylevel
- PR #3813: Make array_view::operator= non-const
- PR #3814: [examples] use np.radians/np.degrees where appropriate
- PR #3710: allow selecting the backend by setting the environment variable MPLBACKEND
- PR #3811: copy all array_view members in copy constructor
- PR #3806: OSX backend. 2D histograms are flipped vertically
- PR #3810: extend #if to include both CLONGDOUBLE related definitions
- PR #3808: BUG : fix #3805
- PR #3807: A couple of simple to fix warnings in the examples
- PR #3801: Fonts demos improvments
- PR #3774: [examples] final pep8 fixes
- PR #3799: Update to doc/conf.py to allow for building docs without qt installed
- PR #3797: Fix for #3789, segfault in _tri

- PR #3698: fixed axvline description of ymin/ymax args. Little edit in axhline doc
- PR #3083: New rcParams to set pyplot.suptitle() defaults
- PR #3788: Fix Sphinx warning in widgets
- PR #3683: remove _orig_color which is duplicate of _rgb
- PR #3502: Improved selection widget
- PR #3786: Fix 'version version not identified' message.
- PR #3784: Fix warning in docs causing Travis error
- PR #3736: Boxplot examples
- PR #3762: WebAgg: flush stdout after printing, redirect "stopped" message to stder... [backport to 1.4.x]
- PR #3770: Treat Sphinx warnings as errors when building docs on Travis
- PR #3777: Upgrade agg to SVN version
- PR #3781: Fix compiler warning
- PR #3780: backend_pgf: pgftext now requires color inside argument (fix #3779) [backport to 1.4.x]
- PR #3778: Reduce coupling between _tkagg and _backend_agg modules
- PR #3737: Rgb2lab minimal
- PR #3771: [examples] fix pep8 error classes e225, e227 and e228
- PR #3769: made idle_event() in backend_bases.py return True
- PR #3768: Mock backens when building doc
- PR #3714: [examples] fix pep8 error classes e231 and e241
- PR #3764: MNT : removed *args from CallbackRegistry init
- PR #3767: RST fixes for the docs
- PR #3765: MNT : delete unused Image
- PR #3763: WebAgg: _png.write_png raises TypeError
- PR #3760: ENH: use fewer points for 3d quiver plot
- PR #3499: Legend marker label placement
- PR #3735: ENH: add pivot kwarg to 3d quiver plot
- PR #3755: Reenable shading tests for numpy 1.9.1 and later
- PR #3744: Final decxx corrections to PR #3723
- PR #3752: Make sure that initial state gets reset if anything goes wrong in ``rc_context`` [backport to 1.4.x]
- PR #3743: remove mention to %pylab [backport to 1.4.2-doc]
- PR #3691: Minor C++ improvements

- PR #3729: handling of color=None by eventplot(), fixes #3728
- PR #3546: Example of embedding a figure into an existing Tk canvas
- PR #3717: Github status upgrade
- PR #3687: Errorbar markers not drawn in png output
- PR #3724: Remove duplicate import_array() call
- PR #3725: Fix invalid symbol if numpy 1.6
- PR #3723: Complete removal of PyCXX
- PR #3721: Subplots deprecation
- PR #3719: Turn rcparams warning into error and remove knowfail
- PR #3718: Use is to compare with None in backend_pdf
- PR #3716: Ignore doc generated files
- PR #3702: Remove the check on path length over 18980 in Cairo backend
- PR #3684: Build failure on Launchpad
- PR #3668: [examples] pep8 fix E26*
- PR #3303: Adding legend handler to PolyCollection and labels to stackplot
- PR #3675: Additional Warnings in docs build on travis after merge of decxx
- PR #3630: refactor ftface_props example
- PR #3671: fix for #3669 Font issue without PyCXX
- PR #3681: use _fast_from_codes_and_verts in transform code
- PR #3678: DOC/PEP8 : details related to PR #3433
- PR #3677: Rotation angle between 0 and 360.
- PR #3674: Silince UnicodeWarnings in tests
- PR #3298: Wedge not honouring specified angular range
- PR #3351: Update demo_floating_axes.py
- PR #3448: Fix scaling of custom markers [backport to 1.4.x]
- PR #3485: Reduce the use of XObjects in pdf backend [backport to 1.4.x]
- PR #3672: Python3 pep8 fixes
- PR #3558: Adds multiple histograms side-by-side example
- PR #3665: Remove usage of raw strides member in _backend_gdk.c
- PR #3309: Explicitly close read and write of Popen process (latex)
- PR #3662: Make all classes new-style.
- PR #3646: Remove PyCXX dependency for core extension modules

- PR #3664: [examples] pep8 fix e251 e27*
- PR #3294: fix typo in figlegend_demo.py
- PR #3666: remove print from test
- PR #3638: MNT : slight refactoring of Gcf
- PR #3387: include PySide in qt4agg backend check
- PR #3597: BUG/TST : skip example pep8 if don't know source path
- PR #3661: Numpy 1.6 fixes
- PR #3635: fix pep8 error classes e20[12] and e22[12] in examples
- PR #3547: Don't use deprecated numpy APIs
- PR #3628: Document auto-init behavior of colors.Normalize and cm.ScalarMappable.
- PR #3640: figure.max_num_figures was renamed to figure.max_open_warning.
- PR #3650: Typo fixes. [backport to doc branch]
- PR #3642: TST : know-fail shadding tests
- PR #3619: PatchCollection: pass other kwargs for match_original=True
- PR #3629: examples: fix pep8 error class E211
- PR #3515: examples: fix pep8 error classes E111 and E113
- PR #3625: animate_decay.py example code is less complicated
- PR #3613: Fix problem with legend if data has NaN's [backport to 1.4.x]
- PR #3611: Fix spelling error
- PR #3600: BUG: now only set 'marker' and 'color' attribute of fliers in boxplots
- PR #3594: Unicode decode error [backport to 1.4.x]
- PR #3595: Some small doc fixes only relevant on the master branch
- PR #3291: Lightsource enhancements
- PR #3578: Fixes test to assert instead of print
- PR #3575: Supports locale-specified encoding for rcfile.
- PR #3556: copy/paste corrections in test_backend_qt5
- PR #3545: Provide an informative error message if something goes wrong in setfont [backport to 1.4.x]
- PR #3369: Added legend.framealpha to rcParams, as mentioned in axes.legend docstring
- PR #3510: Fix setupext [backport to 1.4.x]
- PR #3513: examples: fully automated fixing of E30 pep8 errors
- PR #3507: general pep8 fixes

- PR #3506: Named colors example, figure size correction [backport to 1.4.0-doc]
- PR #3501: Bugfix for text.xytext property
- PR #3376: Move widget.{get,set}_active to AxisWidget.
- PR #3419: Better repr for Bboxes.
- PR #3474: call set cursor on zoom/pan toggle [backpont to 1.4.x]
- PR #3425: Pep8ify examples
- PR #3477: Better check for required dependency libpng
- PR #2900: Remove no-longer-necessary KnownFail for python 3.2.
- PR #3467: Bugfix in mlab for strided views of np.arrays [backport to 1.4.x]
- PR #3469: Fix handling of getSaveFileName to be consistent [backport to 1.4.x]
- PR #3384: Test marker styles
- PR #3457: Add Qt5Agg to backends in matplotlibrc.template.
- PR #3438: Get rid of unused pre python 2.6 code in doc make.py
- PR #3432: Update whats_new.rst
- PR #3282: Catch warning thrown in Mollweide projection.
- PR #2635: Crash on saving figure if text.usetex is True
- PR #3241: Cast to integer to get rid of numpy warning
- PR #3244: Filter warnings in rcparams test (and others)
- PR #3378: BUG: Fixes custom path marker sizing for issue #1980

Issues (1189):

- #7009: No good way to disable SpanSelector
- #7040: It is getting increasingly difficult to build the matplotlib documentation
- #6965: ArtistAnimation cannot animate Figure-only artists
- #6964: Docstring for ArtistAnimation is incorrect
- #7062: remove the contour on a Basemap object
- #7061: remove the contour on Basemap
- #7054: Whether the new version 2.0 will support high-definition screen?
- #7053: When will release 2.0 official version?
- #6797: Undefined Symbol Error On Ubuntu
- #6523: matplotlib-2.0.0b1 test errors on Windows
- #4753: rubber band in qt5agg slow
- #6959: extra box on histogram plot with a single value

- #6816: Segmentation fault on Qt5Agg when using the wrong linestyle
- #4212: Hist showing wrong first bin
- #4602: bar / hist : gap between first bar and other bars with lw=0.0
- #6641: Edge ticks sometimes disappear
- #7041: Python 3.5.2 crashes when launching matplotlib 1.5.1
- #7028: Latex Greek fonts not working in legend
- #6998: dash pattern scaling with linewidth should get it's own rcParam
- #7021: How to prevent matplotlib from importing qt4 libraries when only
- #7020: Using tick_right() removes any styling applied to tick labels.
- #7018: Website Down
- #6785: Callbacks of draggable artists should check that they have not been removed
- #6783: Draggable annotations specified in offset coordinates switch to figure coordinates after dragging
- #7015: pcolor() not using "data" keyword argument
- #7014: matplotlib works well in ipython note book but can't display in a terminal running
- #6999: cycler 0.10 is required due to change_key() usage
- #6794: Incorrect text clipping in presence of multiple subplots
- #7004: Zooming with a large range in y-values while using the linestyle "–" is very slow
- #6828: Spikes in small wedges of a pie chart
- #6940: large memory leak in new contour routine
- #6894: bar(..., linewidth=None) doesn't display bar edges with mpl2.0b3
- #6989: bar3d no longer allows default colors
- #6980: problem accessing canvas on MacOS 10.11.6 with matplotlib 2.0.0b3
- #6804: Histogram of xarray.DataArray can be extremely slow
- #6859: Update URL for links to ggplot
- #6852: Switching to log scale when there is no positive data crashes the Qt5 backend, causes inconsistent internal state in others
- #6740: PGF Backend: Support interpolation='none'?
- #6665: regression: builtin latex rendering doesn't find the right mathematical fonts
- #6984: plt.annotate(): segmentation fault when coordinates are too high
- #6979: plot won't show with plt.show(block=False)
- #6981: link to ggplot is broken...
- #6975: [Feature request] Simple ticks generator for given range

- #6905: pcolorfast results in invalid cursor data
- #6970: quiver problems when angles is an array of values rather than 'uv' or 'xy'
- #6966: No Windows wheel available on PyPI for new version of matplotlib (1.5.2)
- #6721: Font cache building of matplotlib blocks requests made to HTTPd
- #6844: scatter edgecolor is broken in Matplotlib 2.0.0b3
- #6849: BUG: endless loop with MaxNLocator integer kwarg and short axis
- #6935: matplotlib.dates.DayLocator cannot handle invalid input
- #6951: Ring over A in AA is too high in Matplotlib 1.5.1
- #6960: axvline is sometimes not shown
- #6473: Matplotlib manylinux wheel - ready to ship?
- #5013: Add Hershey Fonts a la IDL
- #6953: ax.vlines adds unwanted padding, changes ticks
- #6946: No Coveralls reports on GitHub
- #6933: Misleading error message for matplotlib.pyplot.errorbar()
- #6945: Matplotlib 2.0.0b3 wheel can't load libpng in OS X 10.6
- #3865: Improvement suggestions for matplotlib.Animation.save('video.mp4')
- #6932: Investigate issue with pyparsing 2.1.6
- #6941: Interfering with yahoo_finance
- #6913: Cant get currency from yahoo finance with matplotlib
- #6901: Add API function for removing legend label from graph
- #6510: 2.0 beta: Boxplot patches zorder differs from lines
- #6911: freetype build won't become local
- #6866: examples/misc/longshort.py is outdated
- #6912: Matplotlib fail to compile matplotlib._png
- #1711: Autoscale to automatically include a tiny margin with `Axes.errorbar()`
- #6903: RuntimeError('Invalid DISPLAY variable') - With docker and django
- #6888: Can not maintain zoom level when left key is pressed
- #6855: imsave-generated PNG files missing edges for certain resolutions
- #6479: Hexbin with log scale takes extent range as logarithm of the data along the log axis
- #6795: suggestion: set_xticklabels and set_yticklabels default to current labels
- #6825: I broke imshow(<signed integer array>) :-(
- #6858: PyQt5 pyplot error

- #6853: PyQt5 (v5.7) backend - TypeError upon calling figure()
- #6835: Which image formats to build in docs.
- #6856: Incorrect plotting for versions > 1.3.1 and GTK.
- #6838: Figures not showing in interactive mode with macosx backend
- #6846: GTK Warning
- #6839: Test `test_pickle.test_complete` is broken
- #6691: rcParam missing tick side parameters
- #6833: plot contour with levels from discrete data
- #6636: DOC: gallery supplies 2 pngs, neither of which is default
- #3896: dates.date2num bug with daylight switching hour
- #6685: 2.0 dev legend breaks on scatterplot
- #3655: ensure removal of font cache on version upgrade
- #6818: Failure to build docs: unknown property
- #6798: clean and regenerate travis cache
- #6782: 2.x: Contour level count is not respected
- #6796: plot/lines not working for datetime objects that span old dates
- #6660: cell focus/cursor issue when plotting to nbagg
- #6775: Last figure in http://matplotlib.org/users/pyplot_tutorial.html is not displayed correctly
- #5981: Increased tick width in 3D plots looks odd
- #6771: ImportError: No module named artist
- #6289: Grids are not rendered in backend implementation
- #6621: Change in the result of test_markevery_linear_scales_zoomed
- #6515: Dotted grid lines in v2.0.0b1
- #6511: Dependencies in installation of 2.0.0b1
- #6668: “Bachelor’s degrees…” picture in the gallery is cropped
- #6751: Tableau style
- #6742: import matplotlib.pyplot as plt throws an erro
- #6097: anaconda package missing nose dependency
- #6299: savefig() to eps/pdf does not work
- #6393: Pair of floats breaks plotting renderer (weirdest bug I’ve ever seen)
- #6387: import matplotlib causes UnicodeDecodeError
- #6471: Colorbar label position different when executing a block of code

- #6732: Adding `pairplot` functionality?
- #6749: Step diagram does not support xlim() and ylim()
- #6748: Step diagram does not suppot
- #6615: Bad event index for step plots
- #6588: Different line styles between PNG and PDF exports.
- #6693: linestyle="None" argument for fill_between() doesn't work
- #6592: Linestyle pattern depends on current style, not style set at creation
- #5430: Linestyle: dash tuple with offset
- #6728: Can't install matplotlib with specific python version
- #6546: Recommendation to install packages for various OS
- #6536: get_sample_data() in cbook.py duplicates code from _get_data_path() __init__.py
- #3631: Better document meaning of notches in boxplots
- #6705: The test suite spends 20% of it's time in `gc.collect()`
- #6698: Axes3D scatter crashes without alpha keyword
- #5860: Computer Modern Roman should be the default serif when using TeX backend
- #6702: Bad fonts crashes matplotlib on startup
- #6671: Issue plotting big endian images
- #6196: Qt properties editor discards color alpha
- #6509: pylab image_masked is broken
- #6657: appveyor is failing on pre-install
- #6610: Icons for Tk are not antialiased.
- #6687: Small issues with the example `polar_scatter_demo.py`
- #6541: Time to deprecate the GTK backend
- #6680: Minor typo in the docstring of `IdentityTransform`?
- #6670: plt.text object updating incorrectly with blit=False
- #6646: Incorrect fill_between chart when use set_xscale('log')
- #6540: imshow(..., alpha=0.5) produces different results in 2.x
- #6650: fill_between() not working properly
- #6566: Regression: Path.contains_points now returns uint instead of bool
- #6624: bus error: fc-list
- #6655: Malware found on matplotlib components
- #6623: RectangleSelector disappears after resizing

- #6629: matplotlib version error
- #6638: get_ticklabels returns ‘’ in ipython/python interpreter
- #6631: can’t build matplotlib on smartos system(open solaris)
- #6562: 2.x: Cairo backends cannot render images
- #6507: custom scatter marker demo broken
- #6591: DOC: update static image for interpolation_none_vs_nearest.py example
- #6607: BUG: saving image to png changes colors
- #6587: please copy http://matplotlib.org/devdocs/users/colormaps.html to http://matplotlib.org/users
- #6594: Documentation Typo
- #5784: dynamic ticking (#5588) should avoid (if possible) single ticks
- #6492: mpl_toolkits.mplot3d has a null byte somewhere
- #5862: Some Microsoft fonts produce unreadable EPS
- #6537: bundled six 1.9.0 causes ImportError: No module named ‘winreg’ in Pympler
- #6563: pyplot.errorbar attempts to plot 0 on a log axis in SVGs
- #6571: Unexpected behavior with ttk.Notebook - graph not loaded unless tab preselected
- #6570: Unexpected behavior with ttk.Notebook - graph not loaded unless tab preselected
- #6539: network tests are not skipped when running tests.py with –no-network
- #6567: qt_compat fails to identify PyQt5
- #6559: mpl 1.5.1 requires pyqt even with a wx backend
- #6009: No space before unit symbol when there is no SI prefix in ticker.EngFormatter
- #6528: Fail to install matplotlib by “pip install” on SmartOS(like open solaris system)
- #6531: Segmentation fault with any backend (matplotlib 1.4.3 and 1.5.1) when calling pyplot.show()
- #6513: Using gray shade from string ignores alpha parameters
- #6477: Savefig() to pdf renders markers differently than show()
- #6525: PS export issue with custom font
- #6514: LaTeX axis labels can no longer have custom fonts
- #2663: Multi Cursor disable broken
- #6083: Figure linewidth default in rcparams
- #1069: Add a donation information page
- #6035: Issue(?): head size of FancyArrowPatch changes between interactive figure and picture export
- #6495: new figsize is bad for subplots with fontsize 12
- #6493: Stepfilled color cycle for background and edge different

- #6380: Implicit addition of "color" to property_cycle breaks semantics
- #6447: Line2D.contains does not take drawstyle into account.
- #6257: option for default space between title and axes
- #5868: tight_layout doesn't leave enough space between outwards ticks and axes title
- #5987: Outward ticks cause labels to be clipped by default
- #5269: Default changes: legend
- #6489: Test errors with numpy 1.11.1rc1
- #5960: Misplaced shadows when using FilteredArtistList
- #6452: Please add a generic "seaborn" style
- #6469: Test failures testing matplotlib 1.5.1 manylinux wheels
- #5854: New cycler does not work with bar plots
- #5977: legend needs logic to deal with new linestyle scaling by linewidth
- #6365: Default format time series xtick labels changed
- #6104: docs: latex required for PDF plotting?
- #6451: Inequality error on web page http://matplotlib.org/faq/howto_faq.html
- #6459: use conda already installed on appveyor
- #6043: Advanced hillshading example looks strange with new defaults.
- #6440: BUG: set_tick_params labelcolor should apply to offset
- #6458: Wrong package name in INSTALL file
- #2842: matplotlib.tests.test_basic.test_override_builtins() fails with Python >=3.4
- #2375: matplotlib 1.3.0 doesn't compile with Solaris Studio 12.1 CC
- #2667: matplotlib.tests.test_mathtext.test_mathtext_{cm,stix,stixsans}_{37,53}.test are failing
- #2243: axes limits with aspect='equal'
- #1758: y limit with dashed or dotted lines hangs with somewhat big data
- #5994: Points annotation coords not working in 2.x
- #6444: matplotlib.path.contains_points is a LOT slower in 1.51
- #5461: Feature request: allow a default line alpha to be set in mpl.rcParams
- #5132: ENH: Set the alpha value for plots in rcParams
- #6449: axhline and axvline linestyle as on-off seq doesn't work if set directly in function call
- #6416: animation with 'ffmpeg' backend and 'savefig.bbox = tight' garbles video
- #6437: Improperly spaced time axis
- #5974: scatter is not changing color in Axes3D

- #6436: clabels plotting outside of projection limb
- #6438: Cant get emoji working in Pie chart legend with google app engine. Need help.
- #6362: greyscale scatter points appearing blue
- #6301: tricky bug in ticker due to special behaviour of numpy
- #6276: Ticklabel format not preserved after editing plot limits
- #6173: `linestyle` parameter does not support default cycler through `None`, crashes instead.
- #6109: colorbar _ticker +_locate bug
- #6231: Segfault when figures are deleted in random order
- #6432: micro sign doesn't show in EngFormatter
- #6057: Infinite Loop: LogLocator Colorbar & update_ticks
- #6270: pyplot.contour() not working with matplotlib.ticker.LinearLocator()
- #6058: "Configure subplots" tool is initialized very inefficiently in the Qt backends
- #6363: Change `legend` to accept `alpha` instead of (only) `framealpha`.
- #6394: Severe bug in ``‘‘imshow‘‘`` when plotting images with small values
- #6368: Bug: matplotlib.pyplot.spy: does not work correctly for sparse matrices with many entries (>= 2**32)
- #6419: Imshow does not copy data array but determines colormap values upon call
- #3615: mouse scroll event in Gtk3 backend
- #3373: add link to gtk embedding cookbook to website
- #6121: opening the configure subplots menu moves the axes by a tiny amount
- #2511: NavigationToolbar breaks if axes are added during use.
- #6349: Down arrow on GTK3 backends selects toolbar, which eats furthur keypress events
- #6408: minor ticks don't respect rcParam xtick.top / ytick.right
- #6398: sudden install error with pip (pyparsing 2.1.2 related)
- #5819: 1.5.1rc1: dont use absolute links in the "new updates" on the homepage
- #5969: urgent bug after 1.5.0: offset of LineCollection when apply agg_filter
- #5767: axes limits (in old "round_numbers" mode) affected by floating point issues
- #5755: Better choice of axes offset value
- #5938: possible bug with ax.set_yscale('log') when all values in array are zero
- #5836: Repeated warning about fc-list
- #6399: pyparsing version 2.1.2 not supported (2.1.1 works though)
- #5884: `numpy` as no Attribute `string0`

- #6395: Deprecation warning for axes.color_cycle
- #6385: Possible division by zero in new `get_tick_space()` methods; is rotation ignored?
- #6344: Installation issue
- #6315: Qt properties editor could sort lines labels using natsort
- #5219: Notebook backend: possible to remove javascript/html when figure is closed?
- #5111: nbagg backend captures exceptions raised by callbacks
- #4940: NBAgg figure management issues
- #4582: Matplotlib IPython Widget
- #6142: matplotlib.ticker.LinearLocator view_limits algorithm improvement?
- #6326: Unicode invisible after image saved
- #5980: Gridlines on top of plot by default in 2.0?
- #6272: Ability to set default scatter marker in matplotlibrc
- #6335: subplots animation example is broken on OS X with qt4agg
- #6357: pyplot.hist: normalization fails
- #6352: clim doesn't update after draw
- #6353: hist won't norm for small numbers
- #6343: prop_cycle breaks keyword aliases
- #6226: Issue saving figure as eps when using gouraud shaded triangulation
- #6330: ticklabel_format reset to default by ScalarFormatter
- #4975: Non-default `color_cycle` not working in Pie plot
- #5990: Scatter markers do not follow new colour cycle
- #5577: Handling of "next color in cycle" should be handled differently
- #5489: Special color names to pull colors from the currently active color cycle
- #6325: Master requires cycler 0.10.0
- #6278: imshow with pgf backend does not render transparency
- #5945: Figures in the notebook backend are too large following DPI changes
- #6332: Animation with blit broken
- #6331: matplotlib pcolormesh seems to slide some data around on the plot
- #6307: Seaborn style sheets don't edit `patch.facecolor`
- #6294: Zero size ticks show up as single pixels in rendered pdf
- #6318: Cannot import mpl_toolkits in Python3
- #6316: Viridis exists but not in plt.cm.datad.keys()

- #6082: Cannot interactively edit axes limits using Qt5 backend
- #6309: Make CheckButtons based on subplots automatically
- #6306: Can't show images when plt.show() was executed
- #2527: Vertical alignment of text is too high
- #4827: Pickled Figure Loses sharedx Properties
- #5998: math??{} font styles are ignored in 2.x
- #6293: matplotlib notebook magic cells with output plots - skips next cell for computation
- #235: hatch linewidth patch
- #5875: Manual linestyle specification ignored if 'prop_cycle' contains 'ls'
- #5959: imshow rendering issue
- #6237: MacOSX agg version: doesn't redraw after keymap.grid keypress
- #6266: Better fallback when color is a float
- #6002: Potential bug with 'start_points' argument of 'pyplot.streamplot'
- #6265: Document how to set viridis as default colormap in mpl 1.x
- #6258: Rendering vector graphics: parsing polygons?
- #1702: Bug in 3D histogram documentation
- #5937: xticks/yticks default behaviour
- #4706: Documentation - Basemap
- #6255: Can't build matplotlib.ft2font in cygwin
- #5792: Not easy to get colorbar tick mark locations
- #6233: ImportError from Sphinx plot_directive from Cython
- #6235: Issue with building docs with Sphinx 1.4.0
- #4383: xkcd color names
- #6219: Example embedding_in_tk.py freezes in Python3.5.1
- #5067: improve whats_new entry for prop cycler
- #4614: Followup items from the matplotlib 2.0 BoF
- #5986: mac osx backend does not scale dashes by linewidth
- #4680: Set forward=True by default when setting the figure size
- #4597: use mkdtemp in _create_tmp_config_dir
- #3437: Interactive save should respect 'savefig.facecolor' rcParam.
- #2467: Improve default colors and layouts
- #4194: matplotlib crashes on OS X when saving to JPEG and then displaying the plot

- #4320: Pyplot.imshow() "None" interpolation is not supported on Mac OSX
- #1266: Draggable legend results RuntimeError and AttributeError on Mac OS 10.8.1
- #5442: xkcd plots rendered as regular plots on Mac OS X
- #2697: Path snapping does not respect quantization scale appropriate for Retina displays
- #6049: Incorrect TextPath display under interactive mode
- #1319: macosx backend lacks support for cursor-type widgets
- #531: macosx backend does not work with blitting
- #5964: slow rendering with backend_macosx on El Capitan
- #5847: macosx backend color rendering
- #6224: References to non-existing class FancyBoxPatch
- #781: macosx backend doesn't find fonts the same way as other backends
- #4271: general colormap reverser
- #6201: examples svg_histogram.html failes with UnicodeEncodeError
- #6212: ENH? BUG? `pyplot.setp`/`Artist.setp` does not accept non-indexable iterables of handles.
- #4445: Two issues with the axes offset indicator
- #6209: Qt4 backend uses Qt5 backend
- #6136: Feature request: configure default scatter plot marker size
- #6180: Minor typos in the style sheets users' guide
- #5517: "interactive example" not working with PySide
- #4607: bug in font_manager.FontManager.score_family()
- #4400: Setting annotation background covers arrow
- #596: Add "bring window to front" functionality
- #4674: Default marker edge width in plot vs. scatter
- #5988: rainbow_text example is missing some text
- #6165: MacOSX backend hangs drawing lines with many dashes/dots
- #6155: Deprecation warnings with Dateutil 2.5
- #6003: In 'pyplot.streamplot', starting points near the same streamline raise 'InvalidIndexError'
- #6105: Accepting figure argument in subplot2grid
- #6184: csv2rec handles dates differently to datetimes when datefirst is specified.
- #6164: Unable to use PySide with gui=qt
- #6166: legends do not refresh

- #3897: bug: inconsistent types accepted in DateLocator subclasses
- #6160: EPS issues with rc parameters used in seaborn library on Win 8.1
- #6163: Can´t make matplotlib run in my computer
- #5331: Boxplot with zero IQR sets whiskers to max and min and leaves no outliers
- #5575: plot_date() ignores timezone
- #6143: drawstyle accepts anything as default rather than raising
- #6151: Matplotlib 1.5.1 ignores annotation_clip parameter
- #6147: colormaps issue
- #5916: Headless get_window_extent or equivalent
- #6141: Matplotlib subplots and datetime x-axis functionality not working as intended?
- #6138: No figure shows, no error
- #6134: Cannot plot a line of width=1 without antialiased
- #6120: v2.x failures on travis
- #6092: %matplotlib notebook broken with current matplotlib master
- #1235: Legend placement bug
- #2499: Showing np.uint16 images of the form (h,w,3) is broken
- #5479: Table: auto_set_column_width not working
- #6028: Appearance of non-math hyphen changes with math in text
- #6113: ValueError after moving legend and rcParams.update
- #6111: patches fails when data are array, not list
- #6108: Plot update issue within event callback for multiple updates
- #6069: imshow no longer correctly handles 'bad' (`nan`) values
- #6103: ticklabels empty when not interactive
- #6084: Despined figure is cropped
- #6067: pyplot.savefig doesn't expand ~ (tilde) in path
- #4754: Change default color cycle
- #6063: Axes.relim() seems not to work when copying Line2D objects
- #6065: Proposal to change color – 'indianred'
- #6056: quiver plot in polar projection - how to make the quiver density latitude-dependent ?
- #6051: Matplotlib v1.5.1 apparently not compatible with python-dateutil 2.4.2
- #5513: Call get_backend in pylab_setup
- #5983: Option to Compress Graphs for pgf-backend

- #5895: Polar Projection PDF Issue
- #5948: tilted line visible in generated pdf file
- #5737: matplotlib 1.5 compatibility with wxPython
- #5645: Missing line in a self-sufficient example in navigation_toolbar.rst :: a minor bug in docs
- #6037: Matplotlib xtick appends .%f after %H:%M%:%S on chart
- #6025: Exception in Tkinter/to_rgb with new colormaps
- #6034: colormap name is broken for ListedColormap?
- #5982: Styles need update after default style changes
- #6017: Include tests.py in archive of release
- #5520: 'nearest' interpolation not working with low dpi
- #4280: imsave reduces 1row from the image
- #3057: DPI-connected bug of imshow when using multiple masked arrays
- #5490: Don't interpolate images in RGB space
- #5996: 2.x: Figure.add_axes(..., facecolor='color') does not set axis background colour
- #4760: Default linewidth thicker than axes linewidth
- #2698: ax.text() fails to draw a box if the text content is full of blank spaces and linefeeds.
- #3948: a weird thing in the source code comments
- #5921: test_backend.pgf.check_for(texsystem) does not do what it says...
- #4295: Draggable annotation position wrong with negative x/y
- #1986: Importing pyplot messes with command line argument parsing
- #5885: matplotlib stepfilled histogram breaks at the value 10^-1 on xubuntu
- #5050: pandas v0.17.0rc1
- #3658: axes.locator_params fails with LogLocator (and most Locator subclasses)
- #3742: Square plots
- #3900: debugging Segmentation fault with Qt5 backend
- #4192: Error when color value is None
- #4210: segfault: fill_between with Python3
- #4325: FancyBboxPatch wrong size
- #4340: Histogram gap artifacts
- #5096: Add xtick.top.visible, xtick.bottom.visible, ytick.left.visible, ytick.right.visible to rcParams
- #5120: custom axis scale doesn't work in 1.4.3
- #5212: shifted(?) bin positions when plotting multiple histograms at the same time

- #5293: Qt4Agg: RuntimeError: call __init__ twice
- #5971: Add support for PySide2 (Qt5)
- #5993: Basemap readshapefile should read shapefile for the long/lat specified in the Basemap instance.
- #5991: basemap crashes with no error message when passed numpy nan's
- #5883: New colormaps : Inferno, Viridis, ...
- #5841: extra label for non-existent tick
- #4502: Default style proposal: outward tick marks
- #875: Replace "jet" as the default colormap
- #5047: Don't draw end caps on error bars by default
- #4700: Overlay blend mode
- #4671: Change default legend location to 'best'.
- #5419: Default setting of figure transparency in NbAgg is a performance problem
- #4815: Set default axis limits in 2D-plots to the limits of the data
- #4854: set numpoints to 1
- #5917: improved dash styles
- #5900: Incorrect Image Tutorial Inline Sample Code
- #5965: xkcd example in gallery
- #5616: Better error message if no animation writer is available
- #5920: How to rotate secondary y axis label so it doesn't overlap with y-ticks, matplotlib
- #5966: SEGFAULT if `pyplot` is imported
- #5967: savefig SVG and PDF output for scatter plots is excessively complex, crashses Inkscape
- #1943: legend doesn't work with stackplot
- #5923: Windows usetex=True error in long usernames
- #5940: KeyError: 'getpwuid(): uid not found: 5001'
- #5748: Windows test failures on appveyor
- #5944: Notebook backend broken on Master
- #5946: Calling subplots_adjust breaks savefig output
- #5929: Fallback font doesn't work on windows?
- #5925: Data points beyond axes range plotted when saved to SVG
- #5918: Pyplot.savefig is very slow with some combinations of data/ylim scales
- #5919: Error when trying to import matplotlib into IPython notebook

- #5803: Barbs broken
- #5846: setupext.py: problems parsing setup.cfg (not updated to changes in configparser)
- #5309: Differences between function and keywords for savefig.bbox and axes.facecolor
- #5889: Factual errors in HowTo FAQ Box Plot Image
- #5618: New rcParams requests
- #5810: Regression in test_remove_shared_axes
- #5281: plt.tight_layout(pad=0) cuts away outer ticks
- #5909: The documentation for LinearLocator's presets keyword is unclear
- #5864: mathtext mishandling of certain exponents
- #5869: doc build fails with mpl-1.5.1 and sphinx-1.3.4 (sphinx-1.3.3 is fine)
- #5835: gridspec.Gridspec doesn't check for consistency in arguments
- #5867: No transparency in *.pgf file when using pgf Backend.
- #5863: left(... right) are too small
- #5850: prop_cycler for custom dashes – linestyle such as (<offset>, (<on>, <off>)) throws error
- #5861: Marker style request
- #5851: Bar and box plots use the 'default' matplotlib colormap, even if the style is changed
- #5857: FAIL: matplotlib.tests.test_coding_standards.test_pep8_conformance_examples
- #5831: tests.py is missing from pypi tarball
- #5829: test_rasterize_dpi fails with 1.5.1
- #5843: what is the source code of ax.pcolormesh(T, R, Z,vmin=0,vmax=255,cmap='jet') ?
- #5799: mathtext kerning around comma
- #2841: There is no set_linestyle_cycle in the matplotlib axes API
- #5821: Consider using an offline copy of Raleway font
- #5822: FuncAnimation.save() only saving 1 frame
- #5449: Incomplete dependency list for installation from source
- #5793: GTK backends
- #5814: Adding colorbars to row subplots doesn't render the main plots when saving to .eps in 1.5.0
- #5816: matplotlib.pyplot.boxplot ignored showmeans keyword
- #5086: Default date format for axis formatting
- #4808: AutoDateFormatter shows too much precision
- #5812: Widget event issue
- #5794: –no-network not recognized as valid option for tests.py

- #5801: No such file or directory: '/usr/share/matplotlib/stylelib'
- #5777: Using default style raises warnings about non style parameters
- #5738: Offset text should be computed based on lowest and highest ticks, not actual axes limits
- #5403: Document minimal MovieWriter sub-class
- #5558: The link to the John Hunter Memorial fund is a 404
- #5757: Several axes_grid1 and axisartist examples broken on master
- #5557: plt.hist throws KeyError when passed a pandas.Series without 0 in index
- #5550: Plotting datetime values from Pandas dataframe
- #4855: Limit what `style.use` can affect?
- #5765: import matplotlib._png as _png ImportError: libpng16.so.16: cannot open shared object
- #5753: Handling of zero in log shared axes depends on whether axes are shared
- #5756: 3D rendering, scatterpoints disapear near edges of surfaces
- #5747: Figure.suptitle does not respect `size` argument
- #5641: plt.errorbar error with empty list
- #5476: annotate doesn't trigger redraw
- #5572: Matplotlib 1.5 broken_barh fails on empty data.
- #5089: axes.properties calls get_axes internally
- #5745: Using internal qhull despite the presence of pyqhull installed in the system
- #5744: cycler is required, is missing, yet build succeeds.
- #5592: Problem with _init_func in ArtistAnimation
- #5729: Test matplotlib.tests.test_backend_svg.test_determinism fails on OSX in virtual envs.
- #4756: font_manager.py takes multiple seconds to import
- #5435: Unable to upgrade matplotlib 1.5.0 through pip
- #5636: Generating legend from figure options panel of qt backend raise exception for large number of plots
- #5365: Warning in test_lines.test_nan_is_sorted
- #5646: Version the font cache
- #5692: Can't remove StemContainer
- #5635: RectangleSelector creates not wanted lines in axes
- #5427: BUG? Normalize modifies pandas Series inplace
- #5693: Invalid caching of long lines with nans
- #5705: doc/users/plotting/examples/axes_zoom_effect.py is not a Python file

- #4359: savefig crashes with malloc error on os x
- #5715: Minor error in set up fork
- #5687: Segfault on plotting with PySide as backend.qt4
- #5708: Segfault with Qt4Agg backend in 1.5.0
- #5704: Issue with xy and xytext
- #5673: colorbar labelling bug (1.5 regression)
- #4491: Document how to get a framework build in a virtual env
- #5468: axes selection in axes editor
- #5684: AxesGrid demo exception with LogNorm: 'XAxis' object has no attribute 'set_scale'
- #5663: AttributeError: 'NoneType' object has no attribute 'canvas'
- #5573: Support HiDPI (retina) displays in docs
- #5680: SpanSelector span_stays fails with use_blit=True
- #5679: Y-axis switches to log scale when an X-axis is shared multiple times.
- #5655: Problems installing basemap behind a proxy
- #5670: Doubling of coordinates in polygon clipping
- #4725: change set_adjustable for share axes with aspect ratio of 1
- #5488: The default number of ticks should be based on the length of the axis
- #5543: num2date ignoring tz in v1.5.0
- #305: Change canvas.print_figure default resolution
- #5660: Cannot raise FileNotFoundError in python2
- #5658: A way to remove the image of plt.figimage()?
- #5495: Something fishy in png reading
- #5549: test_streamplot:test_colormap test broke unintentionally
- #5381: HiDPI support in Notebook backend
- #5531: test_mplot3d:test_quiver3d broke unintentionally
- #5530: test_axes:test_polar_unit broke unintentionally
- #5525: Comparison failure in text_axes:test_phase_spectrum_freqs
- #5650: Wrong backend selection with PyQt4
- #5649: Documentation metadata (release version) does not correspond with some of the 'younger' documentation content
- #5648: Some tests require non-zero tolerance
- #3980: zoom in wx with retnia behaves badly

- #5642: Mistype in pyplot_scales.py of pyplot_tutorial.rst :: a minor bug in docs
- #3316: wx crashes on exit if figure not shown and not explicitly closed
- #5624: Cannot manually close matplotlib plot window in Mac OS X Yosemite
- #4891: Better auto-selection of axis limits
- #5633: No module named externals
- #5634: No module named 'matplotlib.tests'
- #5473: Strange OS warning when import pyplot after upgrading to 1.5.0
- #5524: Change in colorbar extensions
- #5627: Followup for Windows CI stuff
- #5613: Quiverkey() positions arrow incorrectly with labelpos 'N' or 'S'
- #5615: tornado now a requirement?
- #5582: FuncAnimation crashes the interpreter (win7, 64bit)
- #5610: Testfailures on windows
- #5595: automatically build windows conda packages and wheels in master
- #5535: test_axes:test_rc_grid image comparison test has always been broken
- #4396: Qt5 is not mentioned in backends list in doc
- #5205: pcolor does not handle non-array C data
- #4839: float repr in axes parameter editing window (aka the green tick button)
- #5542: Bad superscript positioning for some fonts
- #3791: Update colormap examples.
- #4679: Relationship between line-art markers and the markeredgewidth parameter
- #5601: Scipy/matplotlib recipe with plt.connect() has trouble in python 3 (AnnoteFinder)
- #4211: Axes3D quiver: variable length arrows
- #773: mplot3d enhancement
- #395: need 3D examples for tricontour and tricontourf
- #186: Axes3D with PolyCollection broken
- #178: Incorrect mplot3d contourf rendering
- #5508: Animation.to_html5_video requires python3 base64 module
- #5576: Improper reliance upon pkg-config when C_INCLUDE_PATH is set
- #5369: Change in zorder of streamplot between 1.3.1 and 1.4.0
- #5569: Stackplot does not handle NaNs
- #5565: label keyword is not interpreted proporly in errorbar() for pandas.DataFrame-like objects

- #5561: interactive mode doesn't display images with standard python interpreter
- #5559: Setting window titles when in interactive mode
- #5554: Cropping text to axes
- #5545: EllipseCollection renders incorrectly when passed a sequence of widths
- #5475: artist picker tolerance has no effect
- #5529: Wrong image/code for legend_demo (pylab)
- #5139: plt.subplots for already existing Figure
- #5497: violin{,plot} return value
- #5441: boxplot rcParams are not in matplotlibrc.template
- #5522: axhline fails on custom scale example
- #5528: rho in text for plots erroring
- #4799: Probability axes scales
- #5487: Trouble importing image_comparison decorator in v1.5
- #5464: figaspect not working with numpy floats
- #4487: Should default hist() bins be changed in 2.0?
- #5499: UnicodeDecodeError in IPython Notebook caused by negative numbers in plt.legend()
- #5498: Labels' collisions while plotting named DataFrame iterrows
- #5491: clippedline.py example should be removed
- #5482: RuntimeError: could not open display
- #5481: value error : unknown locale: UTF-8
- #4780: Non-interactive backend calls draw more than 100 times
- #5470: colorbar values could take advantage of offsetting and/or scientific notation
- #5471: FuncAnimation video saving results in one frame file
- #5457: Example of new colormaps is misleading
- #3920: Please fix pip install, so that plt.show() etc works correctly
- #5418: install backend gtk in Cygwin
- #5368: New axes.set_prop_cycle method cannot handle any generic iterable
- #5446: Tests fail to run (killed manually after 7000 sec)
- #5225: Rare race condition in makedirs with parallel processes
- #5444: overline and subscripts/superscripts in mathtext
- #4859: Call `tight_layout()` by default
- #5429: Segfault in matplotlib.tests.test_image:test_get_window_extent_for_AxisImage on python3.5

- #5431: Matplotlib 1.4.3 broken on Windows
- #5409: Match zdata cursor display scalling with colorbar ?
- #5128: ENH: Better default font
- #5420: [Mac OS X 10.10.5] Macports install error :unknown locale: UTF-8
- #3867: OSX compile broken since CXX removal (conda only?)
- #5411: XKCD style fails except for inline mode
- #5406: Hangs on OS X 10.11.1: No such file or directory: '~/.matplotlib/fontList.json'
- #3116: mplot3d: argument checking in plot_surface should be improved.
- #347: Faster Text drawing needed
- #5399: FuncAnimation w/o init_func breaks when saving
- #5395: Style changes doc has optimistic release date
- #5393: wrong legend in errorbar plot for pandas series
- #5396: fill_between() with gradient
- #5221: infinite range for hist(histtype="step")
- #4901: Error running double pendulum animation example
- #3314: assert mods.pop(0) == 'tests' errors for multiprocess tests on OSX
- #5337: Remove –nocapture from nosetests on .travis.yml?
- #5378: errorbar fails with pandas data frame
- #5367: histogram and digitize do not agree on the definition of a bin
- #5314: ValueError: insecure string pickle
- #5347: Problem with importing matplotlib.animation
- #4788: Modified axes patch will not re-clip artists
- #4968: Lasso-ing in WxAgg causes flickering of the entire figure
- #5093: wx event loop broken (1.5.0rc1)
- #5259: 1.5.0~rc2: unittest failures/errors on (debian) i386
- #3315: "Too many open files" in test runs on Python 3.3
- #5328: Reduce duplication between `tests.py` and `matplotlib.__init__:test()`
- #5302: Pixelated fonts when plot saved as jpeg
- #5226: Font cache thread safety
- #5310: Regression in axes.color_cycle assignment on 1.5rc2
- #5316: Axes.bar: wrong default parameter in documentation
- #5317: Make nbagg recognise the requested facecolor of a figure

- #5312: error in set_linestyle
- #5277: implement `get_ticks_direction()`
- #5303: strange issues trying to play wit Matplotlib1.5rc3 (win32, cgohlke)
- #5280: Separate test data from matplotlib package
- #5202: New colormaps are not included in the plt.cm.datad dictionary.
- #4783: Adapt http://matplotlib.org/devdocs/users/colormaps.html to include new colormaps
- #5291: ERROR: matplotlib.tests.test_patheffects.test_PathEffect_get_proxy
- #5286: unit_scatter.py example crashes on Python 3.4
- #5185: Random test failures in Legent tests (1.5.0rc2)
- #5270: Issues zooming in and out with shared axis
- #5265: Document $MATPLOTLIBRC
- #5260: 1.5.0~rc2: unittest failures/errors on (debian) mipsel
- #5237: Error with bar plot and no data
- #5254: no ax_get_lines.style_cycle
- #4896: [mpl_toolkits.axes_grid1] Can't remove host axes' twin axes
- #5242: Legend color race condition
- #5227: Link to PyPI, rather than SourceForge, for direct download links
- #5232: RuntimeError: No SFNT name table
- #5229: Documentation build failures with Numpy 1.10
- #5136: move tarballs from SF to pypi
- #2046: Miscelaneous travis test failiures
- #5173: Error in boxplot, 1.5.0.rc2
- #5209: pyplot fill_between warning since upgrade of numpy to 1.10.10
- #5220: Any chance of retiring Lena from Sampledoc background
- #5218: Figure should be a contextmanager?
- #5049: xkcd plots stopped working on Mac OS X.
- #4024: Path effects applied to annotation text containing n
- #5198: use nose attributes to mark network tests
- #5211: ValueError when passing numpy array as edgecolors argument to scatter
- #5155: .whl package for Python 3.5
- #5149: basemap warpimage does not handle transparency in non-cylindral projections
- #4908: TransformWrapper is not reliably pickleable

- #5196: pyside is broken
- #5194: Refactoring of qt import logic has broken qt tests on 1.5.x
- #5191: self._renderer = _RendererAgg(int(width), int(height), dpi, debug=False) ValueError: width and height must each be below 32768
- #5190: savefig output blank eps files.
- #5175: "prop_cycle" or "prop_cycler" in ver 1.5.0.rc2?
- #5200: plot label will not show up on legend if _ is used as first charcter in label
- #5199: plot color yields inconsistent results
- #5197: pyplot.plot() doesn't respect the selected color
- #5176: git repo contains Mac OS metafile
- #5174: curious issue on Windows trying to run test
- #3588: ax.minorticks_on won't play nicely with symlog-scale.
- #5105: 1.5rc1: Calling figure transform with wrong argument crashs python
- #5107: 1.5.0rc1: Crash with multiline text.
- #5165: _
- #5162: mathtext fails to render left right (for sympy matrices)
- #5156: mpl.rcParams["lines.markeredgewidth"] = 0 removes markers in lines?
- #5094: 'FigureCanvasAgg' object has no attribute '_is_idle_drawing' (1.5.0rc1)
- #5135: BUG: Issue with blitting of PyQt autoscaled figure
- #5078: numpy incompatibility
- #5141: Scatter "c" kwarg hsa changed
- #5140: installing matplotlib fails using pip on python3
- #5059: Precompiled wheel for Python 3.5
- #5083: Please include un-minified version of JQuery
- #5122: docs: backend list don't mention qt5agg in interactive backend list
- #5119: RegularPolygon does not close path if fill=False
- #5077: make versioneer play nice with how we build docs on travis
- #5113: __version__ not filled for master
- #5101: wx: Recapturing the mouse in the same window? especially on Linux
- #5088: 1.5 and 2.0 release schedule
- #5082: Can't build 1.5RC1 on Mac OSX
- #4999: TriAnalyzer.scale_factors() has mismatched boolean indexes

- #4543: Aggregate whats_new and api_changes rst files for docs
- #5058: PyQt4 canvas crashing on MacOS X 10.8
- #4574: Removing figureoptions from subclassed NavigationToolbar2QT
- #5055: Contourf Colorbar
- #5042: Feature request: pre_draw_event
- #5037: IPython is always imported by pyplot, if present
- #750: axes.cla() in mplot3d - grid lines don't fall back.
- #5018: axes3d.py: Several docstrings concatenate with NoneType
- #5016: plot line not shown in some cases involving masked arrays
- #4967: Animations are (mostly) broken
- #5007: wrong zooming behaviour in constrain mode ('x' or 'y' is pressed)
- #4837: fill_between in matplotlib does not handle "rasterized=True" option correctly
- #4691: Numpy version >= 1.10 dependency issue.
- #4736: Docs build hangs at pylab_examples/system_monitor example
- #4744: Release Schedule 1.5
- #5005: bug in matplotlib/examples/pylab_examples/table_demo.py
- #4971: axis('equal') in mplot3d plots causes duplicate z axis in the nbagg backend
- #4989: Install from source on linux sets incorrect permissions for mpl-data files
- #4761: ScalarFormatter throws math domain errors with polar curvilinear grid examples
- #4973: Darkjet
- #4767: Regression Qt5 Backend
- #4966: Removing Line2D is broken
- #4957: IndexError when moused over panned image
- #4960: Bug on the pyplot.table colColours property (version 1.3.0)
- #4391: Switched links in AXISARTIST documentation
- #4955: Switch default Qt library
- #4806: axes.format_cursor_data is raising a (2.7) ValueError and (3.4) TypeError in Jupyter notebook with nbagg backend
- #4947: animation blitting is no longer working
- #4457: nbagg backend: multiple execution of same cell with plots
- #4841: nbagg close figure when removed from dom
- #4281: nbagg: "Close figure" action does not 'close' the figure object

- #4918: Notebook backend zoom rect VEEERY slow in 1.5dev
- #4927: Remove lena.jpg/png, it is undistributable and violating copyright
- #4732: Interactivity is fragile
- #3649: Matplotlib Installing Test Dependencies
- #4914: alpha value of markeredgecolor is overwritten by markerfacecolor
- #4774: Inline backend not working on master
- #4883: Incorrect default backend set for matplotlib after installing pyside
- #4888: Pyplot ability to read 16 bit image (tiff)
- #2654: ``CGContextRef is NULL`` of `tight_layout` with MacOSX backend
- #1368: spanselector does not respect widgetlock
- #4540: add scroll-to zoom to main codebase
- #2694: Provide public access to the toolbar state for widget interaction
- #2699: key_press_handler captures number keys and 'a'?
- #4879: "%matplotlib notebook" required before *every* call to plot?
- #4012: Arrow annotations behave differently between 1.3.1 and 1.4.2
- #4292: Annotation with negative axes fraction coordinate placed incorrectly with v1.4.3
- #4865: Wrong coordinate transform on jupyter notebook (inline backend)
- #4866: plt.plot(..., c="...") doesn't always set the color properly
- #4858: Can't use mpl_toolkit.axes_grid1 together with GTK3 libraries/backend
- #4735: The mailing list is down
- #4462: following mathematic symbols failed in matplotlib.mathtext, version 1.3.1
- #4791: Consider not having osx backend be default for people using macs
- #4796: Ticklabel alignment issue with Arial
- #4823: Can't render multiple contour plots in Django
- #4794: The following required packages can not be built: png
- #4784: Incorrect polygon filling of mplot3d filled contouring
- #4772: Tests not independent of each other
- #4782: Plots missing from http://matplotlib.org/devdocs/users/colormaps.html
- #4775: Update backend_wx and backend_wxagg to remove deprecated wxPython Phoenix items
- #4765: Multiple legend labels in `axes.hist` do not support unicode
- #4764: Conflict of local module Collections
- #4758: matplotlib %notebook steals focus in jupyter notebooks

- #4751: Bar plot seems not to understand numpy.uint8 dtype
- #4139: Annotation text bbox calculated incorrectly in matplotlib>=1.4.3?
- #4140: Annotation bbox clipping incorrect for `boxstyle='round'` Matplotlib >= 1.4.0
- #643: add drawstyle option to fill_between function
- #4335: Whitespace in mathtext is too large and inconsistent with LaTeX
- #4604: Slow/blocking panning in Qt5Agg backend
- #4720: pep8 tests are not really running but reporting success
- #4715: RcParams class does not have proper "get" ability of deprecated keywords
- #3473: afm.py error in python3
- #4742: Error when adding matrix in the X-axis
- #4745: bug in dark_background and transparent = True ?
- #2907: Expose ax.yaxis.labelpad and ax.xaxis.labelpad to the rc file
- #1318: qt4backend zoom-to-rect box is drawn incorrectly on OSX
- #4740: Difference between backend_wx and backend_wxag?
- #4260: Saving a pdf, with a single image, grayscale colormap, and transparent background produces a black background
- #3330: macosx backend doesn't support numeric fontweight
- #2517: PDF output does not respect zorder when using imshow
- #1883: Cairo backend reports file closed on writing to SVGZ file format.
- #1325: Auto-scaling of extent not working for subplots when using sharex and sharey
- #796: im caching in draw_tex
- #699: Error in AxesGrid docs
- #4318: pyplot.savefig fails with ValueError: cannot convert float NaN to integer
- #4504: issue with display hook in base python repl
- #4705: Matplotlib differences MacOSX and other Aggs
- #4365: ValueError exception when plotting a legend on an empty scatter graph with marker size = data length
- #4712: mpl_toolkits does not have __init__.py (Linux)
- #4339: inconsistent plotting behavier between x coordinate in number and dates
- #4207: eps export of rasterized grayscale images leads to compression artifacts
- #4328: Incorrect and Inconsistent output of function PSD when scale_by_freq=False
- #4364: Add test for legend.*color
- #4437: tricontour and tricontourf hang

- #3146: Display z value in matshow in addition of x, y.
- #4673: unify setting figure size
- #4498: Wrong default value “loc” for pyplot.legend() in documentation or implementation
- #4613: min mock version on travis
- #4609: background color of text is forground color on MacOSX
- #4620: Default bottom for step and stepfilled histograms creates offset on log plots
- #4606: Axes.hist with log=True, histtype='step...' ignores bottom kwarg
- #4599: Qt figure options legend crash with no labels
- #4600: Qt figure options colors in text fields
- #4447: Qt figure options widget can't undo step linestyle
- #4598: Error shows when I import matplotlib after installation
- #4545: Latex not work because dvipng_hack_alpha does not work with dvipng that prints non ascii characters
- #4514: GTK3Cairo backend: “TypeError: Can't convert 'bytes' object to str implicitly” with python 3, when using log scale
- #4331: pdf backend not outputting masks for grayscale images
- #4120: _create_tmp_config_dir() in __init__.py does not create a directory
- #4580: alpha value lost on marker edges
- #4566: Add stylesheets to mimic Seaborn and Bokeh
- #4549: Strange behavior with data from grib file
- #4537: Confusion about the number of contour levels
- #4409: cannot convert float nan to integer int(divmod
- #4469: Incorrect hatching in legend for PDF backend
- #4515: Allow rotations and flips of 'x axis' in Geo projections
- #4534: The future of Idle event - (GTK warning on window close)
- #4470: Issue encoding type1 fonts with python 3
- #4579: ImportError: subprocess
- #4556: update errorbar artists
- #4558: Last legend text as path outline with usetex
- #4557: Problem with datetime
- #4577: plot failed.
- #4567: mpl.image.AxesImage array is stored as a MaskedArray
- #4576: python 3.4 import matplotlib error for tkagg

- #4562: fix typo in figure docs
- #4560: cmr12 file not found when using pdf backend.
- #4552: Unable to run ScrollingPlot example
- #4525: masked arrays broken in py3k + gcc 5.1 on arch linux
- #4546: How to get the mpl_toolkits to install
- #4492: Emoji missing when use plt.savefig()
- #4511: he third twinx axis can't be converted to log scale
- #4518: Switching Compilers (Intel Related)
- #3041: pyplot.scatter() does not cycle colors
- #4538: problem matplotlib in eclipse con python 2.7
- #4539: Qt backend should have default way to zoom out
- #3179: Bug : (minor) time axis labels show "%f" instead of microseconds for years up to 1900
- #4126: Load_converter: TypeError: strptime() argument 0 must be str, not <class 'bytes'>
- #4066: Nan issue in text.py
- #4092: CPU 100% usage while idling on some backends.
- #4154: imshow bug on GTKCairo and GTK3Cairo
- #4529: Alpha Channel does not work with custom colormaps
- #4531: matplotlib.pyplot.plot does not use the kwarg 'c'
- #4475: Matplotlib gives useless error message when latex not installed
- #4523: linewidth antialiased=False image is not the same as matlab
- #3898: axes3d.py error when using lines3d and surface3d demos
- #4517: Why is _gci a private method?
- #4283: Memory hole when using Cursor widget with Qt (PySide) and useblit=True on Windows
- #4512: Color defaults
- #4407: Regression in maximum path size
- #4507: "Customizing matplotlib" should mention style sheets
- #4480: Colorbar consist of many segments in SVG output
- #4484: Error converting to_rgba while using `color` syntax versus `c` syntax
- #4500: Save single transparent .gif file from plot?
- #4495: Saving figure as EPS file fails
- #4493: TypeError: boxplot() got an unexpected keyword argument 'labels'
- #4372: Keyboard shortcuts to close the figure are not active on OS X with the backend MacOSX

- #4461: Seqfault in WX backend following the merge of Phoenix
- #4460: WX Phoenix AttributeError: 'NavigationToolbar2WxAgg' issue
- #4478: please re-schedule "Color Overhaul" milestone
- #4454: savefig.format config has no effect under Qt4Agg
- #3984: Support for Scalar Image Cursor Display?
- #3418: auto-wrapping text
- #4453: %matplotlib notebook and procedural interface
- #4195: "Unexpected null receiver" when using python -c
- #4443: Enable Sourcegraph
- #4440: MatplotlibDeprecationWarning: Use `xyann` instead
- #4442: Cannot use umlauts in plot title
- #1709: Feature Requestion: filled step plot
- #4432: Can't print scroll mouse events
- #4424: mpl.tri.Triangulation does not work but plt.tricontourf does
- #2136: Inconsistent linestyle specifications between Line2D and Patch artists
- #4425: Make limits options for vlines and hlines?
- #4410: Curves, line & axis parameter editor has poor support for colors
- #4323: seaborn.tsplot and matplotlib's "Curve lines and axes parameters" editor
- #4421: IPython Notebook: irregular marker edge for 'o'
- #4420: extra kwargs raises exemption
- #4417: incorrect color assigned by scatter plot when plotting points of a single c value
- #4333: Rogue mathtext rendered spaces
- #4412: clabel support for manual singleton levels
- #4415: `matplotlib.cm.gist_rainbow` appears to be always red
- #4405: FancyArrowPatch misinterprets keyword for horizontal alignment
- #4349: SVG backend is assigning same id to clipPath elements
- #4392: What is the real correct function name?
- #4341: Matplotlib: savefig produces incorrect SVG image for bar chart with log-scaled Y-axis
- #2277: Easy fix for clipping misrendering of matplotlib's SVG in other viewers
- #4179: SVG node order issue
- #4386: matplotlib.pyplot.plot() named parameter c sometimes ignored (but color seems to always work)

- #4296: libfreetype not found if installed at uncommon path
- #4344: Feature: “default data point popups”
- #4360: matplotlib styles - setting a legend’s background color?
- #4044: Delete vertical lines (or vertical rules) in plt.table? How to remove cell boundaries and shows table as scientific tables?
- #4227: pcolorfast fails in master when image is not uniform
- #4309: Bbox overlaps method returns True for NaN vertices
- #4338: pylab.plot markers aren’t independent from lines (pylab: 1.9.2)
- #4297: ‘color’ LineCollection prop overrides ‘colors’ kwarg in ax.eventplot()
- #4336: Easiest way to limit clabel to current axis
- #4306: Inconsistent behaviour of float vs. str valued linewidth kwarg when saving plots as ps or eps.
- #4324: Inconsistency in function PSD when the NFFT parameter is an odd number
- #2516: bar() (and possibly other plots) should take an array of string labels for x axis
- #3656: FigureCanvasQT backend_qt5agg bug (backend_qt4agg) works
- #4316: Automatic number of bins for matplotlib histograms
- #4289: nbagg scroll wheel / middle click button event
- #4303: Labels become colors in figure options in qt backends
- #4307: Percent signs don’t show up in labels with ps backend and usetex=True
- #4301: nbagg backend API incomplete?
- #4299: Mathtext left and right with other delimiters (Vert)
- #3886: behavior when plotting no data
- #4252: Simplify handling of remote JPGs
- #4275: UnicodeEncodeError when trying to save a figure
- #4276: BrokenBarHCollection not working with log scale axis
- #4272: inset_axes does not work on top of imshow
- #4068: Cannot unpickle polar plot
- #4262: Bug in pyplot.plot() with zorder/solid_capstyle kwarg combinations
- #2885: PGF backend messes up fill
- #3935: Clipping errors in pgf export when using fill_between and set_ylim()/ylim()
- #3693: pgf files created by PGF backend are incorrect for paths which extend beyond axes limits
- #4239: Errorbar caps missing in master when saving to PDF
- #4242: Histograms in log-scale miss the “unity” bins

- #4215: Get docs building with Sphinx 1.3.0
- #3608: Suggest unexisting filename when saving displayed figure
- #4230: Buffer overflow in xkcd example with pdf backend.
- #4222: Bus error in contour on OSX in master
- #3024: Option to turn on minor ticks in matplotlibrc
- #2880: fignum_exists() could handle string numbers?
- #3715: Axis labels only move with spines if there are ticks
- #3930: ConnectionPath with fancy arrow of length zero produces no plot
- #4181: Extraneous invalid ticks with colorbar extend keyword
- #4216: Cant install Matplolib
- #4214: add legend to error fill port / mlab.offset_line
- #4054: Drawn lines on plot get incorrect over a certain width
- #4199: PDF backend + TeX renders Unicode BOM as visible junk characters on Python 3
- #3903: Dealing (gracefully?) with problematic fonts
- #3285: legend: reverse horizontal order of symbols and labels
- #4162: c=color not changing line color
- #4157: BUG: different colorcycle behavior when using c or color as keyword when plotting.
- #4185: Colorbar outline has broken path in vector backends
- #4110: Move testing support into setup.py
- #4180: UniformTriRefiner gives uncorrect results for transposed arrays
- #4168: nbagg backend stops working
- #3483: Cannot properly reload figure that was previously pickled
- #3610: axes3d label padding does not work
- #4169: Error with nbagg backend on Windows 8.1 and Anaconda
- #4156: AttributeError: 'FigureCanvasAgg' object has no attribute 'invalidate'
- #4160: Bug when clearing axes created with twinx
- #4161: How to test for optional build dependencies?
- #4147: Problems with text that is just a LaTeX minus sign
- #4027: Text going outside subplot with multiple axes
- #4142: MacOSX backend not closing properly
- #2246: Counterintuitive behavior using get/set _yticklabels (or _xticklabels)
- #4136: SVG backend ignores interpolation='nearest' option in imshow()

- #2387: Clean up imports
- #4083: Marker edges are randomly missing
- #3895: MarkerStyle Instance not Accepted for scatter
- #253: Align text using advance width, not glyph width
- #4073: Can't set marker fillstyle
- #4102: Unsharp text in the Inline-backend.
- #4117: Colormap set_under and set_over have no effect on color bar for PolyCollection
- #4098: best way to access mouse event in webagg after going through backend?
- #4106: Hanging in Django
- #4043: tick_right() doesn't move scale factor
- #1341: Add label support to fill_between
- #4095: Plotting with marker style ',' (pixel marker) shows nothing
- #4074: Sliders show as (truncated) triangles when using Cairo backends, fine with Agg.
- #4076: contains() is broken with scatter plots with master, works with v1.4.3rc1
- #3064: BUG signed/unsigned sloppiness in _image.cpp
- #4078: backend_pdf has a FutureWarning
- #3654: Mac OS X backend: <Error>: CGContextClosePath: no current point
- #2577: _backend_agg.so: undefined symbol: MPL_matplotlib__path_ARRAY_API
- #2359: cxxsupport.cxx not found
- #2568: Matplotlib Animation.save() hangs from stalled pipe
- #2536: fill_between and a y log axis
- #2721: default matplotlib build process crashes python during build process on OS X 10.9
- #3019: Can not find fonts when export pdf using matplotlib Agg backend with usetex=True
- #2778: Change font to Helvetica (Neue) in Linux
- #2954: sort out why tests did not catch #2925
- #3584: Trouble reducing figure size with tkagg backend
- #4040: Baseline alignment of xtick labels when using usetex=True
- #4047: line color not picked up in rc arguments?
- #4071: boxplot raises KeyError when notch == True and one of the boxes has no data.
- #4065: Histogram with histtype='step' dows not respect lines.linewidth
- #3913: After successfully saved a png picture, you can not use it
- #3934: Axis limits default to (0.0, 1.0, 0.0, 1.0) with a Circle patch

- #3986: unable to pickle.load an AxesSubplot object
- #3908: Bug in example user_interfaces/printing_in_wx.html
- #4064: 3D figures cannot be created in 1.4.2: 'module' object has no attribute '_string_to_bool'
- #3750: ImageGrid crashes when specifying axes class
- #4053: PowerNorm().__call__ doesn't work with scalar input value
- #3816: savefig with bbox_inches='tight' ignores annotation without text
- #3095: Memory issue when plotting large arrays with pcolormesh
- #3998: Semilog plots with zero values
- #4049: Issue with plotting zeros in log space
- #4021: using animation.save with ffmpeg writer after compiling with py2exe opens command prompt
- #4046: Can't import pylab after1.4.x update: TypeError: cannot return std::string from Unicode object
- #4030: Disable context menu for WebAgg backend?
- #3633: twinx clear problem
- #3673: drop test files from source distribution?
- #3817: quiver is not plotting only 1 vector
- #3709: quiver does not consider direction of an axis
- #4037: Hist Plot Should allow Normalized Values other than 1
- #4036: WebAgg fails to start on Windows 7 with v1.4.2
- #3272: figure fractions are only accurate with qt backend
- #3352: devel doc independent from release
- #4010: Crash when saving figure with "special" text
- #4018: Text issues when "redirecting" stdout/stderr
- #4013: multiple savefig() –> Error: alloc: invalid block
- #2678: mencoder tests fail
- #4003: Saving animation with mencoder fails if bitrate is specified
- #3734: Image redraw problem on OSX with wx>2.9 & mpl>1.1
- #3820: matplotlib.org/dev points to 1.3.x
- #3962: No image appears with nbagg
- #3996: Choose rcfile to use at runtime
- #3557: Bug fix for plotting minor ticks
- #3647: text backgroundcolor not clipped in figures
- #3963: Missing Events in NbAgg Backend

- #2764: animation save gets error writing to file with ffmpeg and ogv (or ogg)
- #3990: Plot markers are not drawn when no edge is requested
- #3987: unable to pickle.load an AxesSubplot object
- #3979: WebAgg Saving JPEG Raises Error
- #3977: Zooming on images doesn't work properly (anymore)
- #3972: WebAgg Backend Canvas is Not an Image
- #3966: Error in WebAgg When "S" Key is Pressed
- #3960: Matplotlib requires sudo on Ubuntu 14.04 for me
- #3823: latex support fails when $MPLCONFIGDIR is specified on command line
- #3822: test_axes_grid1.py is in the wrong location
- #3766: Transparency not respected in legendPatch
- #3946: Should imshow display a grid when axes.grid is set in rcParams?
- #3869: Numeric labels do not work with plt.hist
- #3956: plot function with pixel marker (,) not displaying data points
- #3951: validation of `pivot` in `quiver`
- #3882: hist / hist2d inconsistent for plotting single data point
- #3929: fill_between() no longer works with border linewidth of 0
- #3907: 'pivot' property not recognised in quiver
- #3626: Manual xlim log stackplots not showing, linear stackplots segfault
- #2845: Why are we doing second -> date conversion our selves?
- #3785: Set legend's background using rcParams/style
- #3697: Using arrowstyle causes "Unknown property shrink"
- #3940: Qt Version Validation Failure
- #3776: Bug in circle patch plotting when using the same patch for two different figures
- #3932: ConnectionPatch with log scale and non-positive starting coordinates fails to produce a figure
- #3927: matplotlib power spectral density (PSD) value discrepancy
- #3928: Unable to install matplotlib when freetype is installed somewhere other than in /usr/lib and /usr/include
- #3915: Numpy.array elementwise comparison to None very slow in backend_pdf.py
- #3910: basemap drawcountries() old country file
- #3911: iptest3 utils with 'is not a writable location'
- #3906: No italics with latex typesetting and pdf backend

- #3901: PolyCollection broken on all Agg backends with linewidth=0
- #3809: Workaround needed to make example in `Transformations Tutorial` work with log axis
- #3894: Incorrect drawing order when plotting polygons in separate Poly3DCollections
- #3887: _affine = transforms.Affine2D removed but still used in collections.py
- #3013: matplotlib does not compile with libpng 1.6.10 installed
- #3892: Matplotlib simple case memory leak
- #3889: Matplotlib 1.4.2 fails to build in Python 3 (TypeError: unorderable types: str() < int())
- #3890: Mercurial fails with Python 3
- #3862: ffmpeg_file vframes arg broken
- #3870: Regression: import fails (MacOS X 10.10.1, Python 3.4.2, matplotlib ca10a34e06645c35a98e9ed3db35f6a8efee70f2)
- #3849: Issue with Seaborn and log plots in Python3
- #3843: xkcd plots with background color have white outlines for text and axes spines
- #2980: get_window_extent method for an image object gives all zeros
- #3864: Mpld3 and networkx
- #2971: Feature Request: Zoom Functions
- #3263: debugging matplotlib
- #3819: Saving pickles from the default UI
- #3705: the style module is not in the generated documentation.
- #3775: matplotlib.cbook.boxplot_stats - typo in documentation (list of returned dictionary keys)
- #1184: AttributeError: 'FigureCanvasQTAgg' object has no attribute 'callbacks'
- #3852: mec='none' removes marker (face) from PNG output
- #1481: Fix variable naming in axes.quiver
- #2413: ERROR: matplotlib.tests.test_bbox_tight.test_bbox_inches_tight_suptile_legend.test
- #2509: Locators: First visible tick does not have `pos=0`
- #2580: phantom axis in eps images despite having set invisible axis in matplotlib
- #3137: DeprecationWarning from calc_label_rot_and_inline in contour.py - array index is a float
- #3206: Unable to use font "CMU Serif"
- #3290: hatch.py could use some integer vs. float division clarification
- #3356: Why does `set_fontproperties` copy?
- #3375: LassoSelector could implement {set,get}_active
- #3441: example code misleading: examples/api/scatter_piecharts.py scaled incorrectly

- #3508: memory leak when using pcolormesh with pylab inline
- #3696: markeredgecolor none visibility bug in 1.5.x (re-opening #598)
- #3711: Rasterized points not rendering when markeredgecolor is none
- #3844: switching styles keeps some of the previous style settings
- #3847: animation and subplots
- #3694: mac yosemite seems to have broken things
- #3758: Missing axes and background when using Gtk3Agg backend and animations in a GtkNotebook
- #3842: Issue with step plot
- #3840: spine location not reported correctly
- #3833: same code, different outputs between executions (transData weird behaviour)
- #3804: UnicodeDecodeError seems still exist
- #3553: fill_between disappears on axis limit change
- #3498: What to do about PyCXX?
- #3805: Cannot initialize ColorbarBase without a norm.
- #3800: scale created by drawmapscale seems incorrect
- #3803: Re-evaluate Cairo's path size limit
- #3789: Segfault on Travis
- #3790: Colorbar's edge drawn in really thick line for saved figures.
- #3796: Boxplot outliers in 1.4.2
- #3794: Slower Imports?
- #3148: On Windows, matplotlib fails to load fonts when installed to a folder with non-ascii path
- #3011: Allow to customize default font size for suptitle() in matplotlibrc
- #3782: segfaults in test suite
- #3779: Text color not respected in output from pgf backend
- #3689: boxplot() whiskerprops argument not recognized (ver 1.4.1)
- #3444: Assume backend/optional package checks fail after defined timeout
- #3685: Docs require skimage
- #3757: Matplotlib can't read its configuration file on non-ascii Windows accounts
- #3738: possible race condition stops build with python3.4
- #3598: test_light_source_shading_default and test_light_source_masked_shading fails with numpy 1.9
- #3732: ensure freetype is really optional

- #3739: Downloads don't specify which specific Python version is used
- #3442: Figures in http://matplotlib.org/users/colormaps.html require scikit-image to build
- #3733: Boxplot and violinplot gallery
- #3731: Aligning `axvline` and `axhline` with increasing linewidth
- #3728: eventplot breaks for default `colors=None` kwarg
- #3712: Invalid symbol in _backend_gdk.c if numpy version < 1.7
- #3692: /usr/include/libpng12/pngconf.h:371:12: error: '__pngconf' does not name a type
- #3704: UnicodeDecodeError and failed test_multiline.test
- #3703: UnicodeDecodeError and failed test_multiline.test
- #3669: Test faliures after merging the decxx branch (#3646)
- #3680: Problem with histograms and normed=True
- #2247: plot_surface: hidden lines re-appearing in PDF and SVG backends
- #3345: too large file size created by the errorbar of matplotlib
- #2910: Cannot set stackplot linewidth=0 when writing to pdf
- #497: keymap defaults aren't always lists
- #3667: A bug in mpl_toolkits.mplot3d.axes3d
- #3596: Pep8 tests fails when running python tests.py from base mpl dir.
- #3660: shading tests + numpy 1.6
- #2092: Move to new Numpy API
- #3601: matplotlib.style.available not updated upon adding/deleting .mplstyle files
- #3616: matplotlib.pyplot.imread silently fails on uint16 images.
- #3651: Error when saving rasterized figure to PDF
- #3470: MacOSX backend breaks for matplotlib 1.4 after importing seaborn
- #3641: Annotations with Latex code cause errors in 1.5 master
- #3623: Qt5 backend doesn't work with Qt 5.3
- #3636: mp4 is a container format, not a codec
- #3639: Shading tests failing on master
- #3617: PatchCollection.__init__ ignores all kwargs if match_original=True
- #2873: Add violin plots
- #3213: add whats_new entry for nbagg
- #3392: Cannot pickle `figure` or `axes` (TypeError: instancemethod)
- #3614: Pickling imshow fails (?due to _imcache)

- #3606: nbagg issues with ipython 3.0
- #3494: corrupt eps output on python3
- #3505: Interactive mode not working in 1.4
- #3311: Ship conda package metadata with matplotlib?
- #3248: Divide by zero error in matplotlib.tests.test_colors.test_light_source_shading_color_range
- #3618: UnicodeDecodeError when I try to import matplotlib from directory with non-ascii name
- #3605: matplotlib.pylab.specgram generate bad image in 1.4.0
- #3604: regression in pandas test suite with mpl 1.4.0
- #3603: Error saving file (Qt5 backend)
- #3544: flier objects missing from structure return by boxplot
- #3516: import error when non-ascii characters are present in cwd or user name (windows)
- #3459: boxplot in version 1.4.0 does not respect property settings for fliers (flierprops)
- #3590: Won't use a font although it can be found by the FontManager
- #3412: Matplotlib 1.4 doesn't install from source on CentOS 6
- #3423: Pytz should be specified and documented as a required dependency
- #3569: boxplot stats regression on empty data
- #3563: boxplot() and xticklabels
- #1713: Can't store Unicode values in .matplotlibrc
- #233: Make hist with 'step' histtype draw Line2D instead of Patch
- #3522: Inverting a datetime / plot_date y-axis
- #3570: matplotlib save dynamic user changes to plot
- #3568: Daily build fails at "import matplotlib.pyplot as plt"
- #3565: clabel randomly inconsistend when placed manually
- #3551: Window isn't drawn
- #3538: Importing matplotlib failing when pacakge "six" is 1.3.0
- #3542: fix boxplot docs
- #3455: Documentation bug: boxplot docs have contradicting information
- #3468: boxplot() draws (min, max) whiskers after a zero-IQR input regardless of whis value
- #3436: matplotlib.use('nbagg ') does not work in Python 3
- #3529: Symlog norm still gives wrong result with integer lintresh.
- #3537: 3D figures cannot be created in 1.4.0: 'module' object has no attribute '_string_to_bool'
- #3527: Drawing an arrow using axis.annotate raises DeprecationWarning

- #3523: invalid EPS figure in Mac OS X
- #3504: postscript axes corner is not perfect
- #3520: a question about subplot in spyder
- #3512: What else apart from `useOffset` is controlling tick label offsets?
- #3493: Incorrect use of super() in mplot3d?
- #3439: Registering backends broken by backwards incompatible change
- #3511: Error in plot-gui while saving image
- #3509: Add Build Instructions for Windows 7 Using Visual Studio?
- #3500: Annotation xytext property does not return xyann value
- #3497: Ortho basemap projection with limits crashes
- #3447: cursor doesn't change on keypress (GTKAgg backend)
- #3472: Memory leak displaying PIL image.
- #3484: TclError for draw_event handler calling close()
- #3480: Duplicate labels produced when using custom Locators/Formatters
- #3475: need for rubberband in zoom tool
- #3465: psd() draw a wrong line with sliced array(Matplotlib 1.4.0)
- #3454: backend_qt5 (1.4.0): Not saving the figure with NavigationToolbar (solved)
- #3416: Specify difficulties installing mpl on OSX.
- #2970: add test of all the standard marker symbols
- #3318: Running `setup.py egg_info` starts to compile everything
- #3466: Invalid DISPLAY variable
- #3463: when executing a small script nothing happens!!
- #2934: Line labels don't update in the legend after changing them through the Qt4Agg dialog box
- #3431: Qt5 toolbar support not working in release 1.4.0
- #3407: Update dns/IP adress
- #3460: zoomed_inset_axes shows a incorrect result.
- #3417: update citation page
- #3450: Wrong permissions when installing from source on Linux
- #3449: matplotlib/colors.py: modifying dict while iterating
- #3445: can't bring plot to front eclipse after running the script on mac ox 10.9
- #3443: Pip install matplotlib does not work on Python 3.2 anymore
- #3411: fix rst mark up

- #3413: update freetype version in docs
- #3396: Sort out OSX dmg files
- #3410: Latex rendering fails in ipython
- #3404: Wrong plot on basemap with `latlon=True`
- #3406: A layer stacking problem of exported svg image compatible with inkscape
- #3327: FontProperties are shared by all three titles of an Axes object
- #1980: Custom marker created from vertex list scales wrong
- #3395: Update Downloads page

CHAPTER SEVEN

LICENSE

Matplotlib only uses BSD compatible code, and its license is based on the PSF license. See the Open Source Initiative licenses page for details on individual licenses. Non-BSD compatible licenses (e.g., LGPL) are acceptable in matplotlib toolkits. For a discussion of the motivations behind the licencing choice, see *Licenses*.

7.1 Copyright Policy

John Hunter began matplotlib around 2003. Since shortly before his passing in 2012, Michael Droettboom has been the lead maintainer of matplotlib, but, as has always been the case, matplotlib is the work of many.

Prior to July of 2013, and the 1.3.0 release, the copyright of the source code was held by John Hunter. As of July 2013, and the 1.3.0 release, matplotlib has moved to a shared copyright model.

matplotlib uses a shared copyright model. Each contributor maintains copyright over their contributions to matplotlib. But, it is important to note that these contributions are typically only changes to the repositories. Thus, the matplotlib source code, in its entirety, is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire matplotlib Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change, when they commit the change to one of the matplotlib repositories.

The Matplotlib Development Team is the set of all contributors to the matplotlib project. A full list can be obtained from the git version control logs.

7.2 License agreement for matplotlib 1.5.3

1. This LICENSE AGREEMENT is between the Matplotlib Development Team ("MDT"), and the Individual or Organization ("Licensee") accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, MDT hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 1.5.3 alone or in any derivative version, provided, however, that MDT's License Agreement and MDT's notice of copyright, i.e., "Copyright (c) 2012-2013

Matplotlib Development Team; All Rights Reserved" are retained in matplotlib 1.5.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib 1.5.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib 1.5.3.

4. MDT is making matplotlib 1.5.3 available to Licensee on an "AS IS" basis. MDT MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, MDT MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB 1.5.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. MDT SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 1.5.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 1.5.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between MDT and Licensee. This License Agreement does not grant permission to use MDT trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib 1.5.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

7.3 License agreement for matplotlib versions prior to 1.3.0

1. This LICENSE AGREEMENT is between John D. Hunter ("JDH"), and the Individual or Organization ("Licensee") accessing and otherwise using matplotlib software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, JDH hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use matplotlib 1.5.3 alone or in any derivative version, provided, however, that JDH's License Agreement and JDH's notice of copyright, i.e., "Copyright (c) 2002-2009 John D. Hunter; All Rights Reserved" are retained in matplotlib 1.5.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates matplotlib 1.5.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to matplotlib 1.5.3.

4. JDH is making matplotlib 1.5.3 available to Licensee on an "AS IS" basis. JDH MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, JDH MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF MATPLOTLIB 1.5.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. JDH SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF MATPLOTLIB 1.5.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING MATPLOTLIB 1.5.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between JDH and Licensee. This License Agreement does not grant permission to use JDH trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using matplotlib 1.5.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CHAPTER

EIGHT

CREDITS

matplotlib was written by John Hunter and is now developed and maintained by a number of active developers. The current co-lead developers of matplotlib are Michael Droettboom and Thomas A. Caswell.

Special thanks to those who have made valuable contributions (roughly in order of first contribution by date). Any list like this is bound to be incomplete and can't capture the thousands and thousands of contributions over the years from these and others:

Jeremy O'Donoghue wrote the wx backend

Andrew Straw Provided much of the log scaling architecture, the fill command, PIL support for imshow, and provided many examples. He also wrote the support for dropped axis spines and the original buildbot unit testing infrastructure which triggered the JPL/James Evans platform specific builds and regression test image comparisons from svn matplotlib across platforms on svn commits.

Charles Twardy provided the impetus code for the legend class and has made countless bug reports and suggestions for improvement.

Gary Ruben made many enhancements to errorbar to support x and y errorbar plots, and added a number of new marker types to plot.

John Gill wrote the table class and examples, helped with support for auto-legend placement, and added support for legending scatter plots.

David Moore wrote the paint backend (no longer used)

Todd Miller supported by STSCI contributed the TkAgg backend and the numerix module, which allows matplotlib to work with either numeric or numarray. He also ported image support to the postscript backend, with much pain and suffering.

Paul Barrett supported by STSCI overhauled font management to provide an improved, free-standing, platform independent font manager with a WC3 compliant font finder and cache mechanism and ported truetype and mathtext to PS.

Perry Greenfield supported by STSCI overhauled and modernized the goals and priorities page, implemented an improved colormap framework, and has provided many suggestions and a lot of insight to the overall design and organization of matplotlib.

Jared Wahlstrand wrote the initial SVG backend.

Steve Chaplin served as the GTK maintainer and wrote the Cairo and GTKCairo backends.

Jim Benson provided the patch to handle vertical mathttext.

Gregory Lielens provided the FltkAgg backend and several patches for the frontend, including contributions to toolbar2, and support for log ticking with alternate bases and major and minor log ticking.

Darren Dale

did the work to do mathtext exponential labeling for log plots, added improved support for scalar formatting, and did the lions share of the psfrag LaTeX support for postscript. He has made substantial contributions to extending and maintaining the PS and Qt backends, and wrote the site.cfg and matplotlib.conf build and runtime configuration support. He setup the infrastructure for the sphinx documentation that powers the mpl docs.

Paul Mcguire provided the pyparsing module on which mathtext relies, and made a number of optimizations to the matplotlib mathtext grammar.

Fernando Perez has provided numerous bug reports and patches for cleaning up backend imports and expanding pylab functionality, and provided matplotlib support in the pylab mode for ipython. He also provided the `matshow()` command, and wrote TConfig, which is the basis for the experimental traited mpl configuration.

Andrew Dalke of Dalke Scientific Software contributed the strftime formatting code to handle years earlier than 1900.

Jochen Voss served as PS backend maintainer and has contributed several bugfixes.

Nadia Dencheva

supported by STSCI provided the contouring and contour labeling code.

Baptiste Carvello provided the key ideas in a patch for proper shared axes support that underlies ganged plots and multiscale plots.

Jeffrey Whitaker at NOAA wrote the *Basemap* toolkit

Sigve Tjoraand, Ted Drain, James Evans and colleagues at the JPL collaborated on the QtAgg backend and sponsored development of a number of features including custom unit types, datetime support, scale free ellipses, broken bar plots and more. The JPL team wrote the unit testing image comparison infrastructure for regression test image comparisons.

James Amundson did the initial work porting the qt backend to qt4

Eric Firing has contributed significantly to contouring, masked array, pcolor, image and quiver support, in addition to ongoing support and enhancements in performance, design and code quality in most aspects of matplotlib.

Daishi Harada added support for “Dashed Text”. See dashpointlabel.py and `TextWithDash`.

Nicolas Young added support for byte images to imshow, which are more efficient in CPU and memory, and added support for irregularly sampled images.

The brainvisa Orsay team and Fernando Perez added Qt support to ipython in pylab mode.

Charlie Moad contributed work to matplotlib’s Cocoa support and has done a lot of work on the OSX and win32 binary releases.

Jouni K. Seppänen wrote the PDF backend and contributed numerous fixes to the code, to tex support and to the get_sample_data handler

Paul Kienzle improved the picking infrastructure for interactive plots, and with Alex Mont contributed fast rendering code for quadrilateral meshes.

Michael Droettboom supported by STSCI wrote the enhanced mathtext support, implementing Knuth's box layout algorithms, saving to file-like objects across backends, and is responsible for numerous bug-fixes, much better font and unicode support, and feature and performance enhancements across the matplotlib code base. He also rewrote the transformation infrastructure to support custom projections and scales.

John Porter, Jonathon Taylor and Reinier Heeres John Porter wrote the mplot3d module for basic 3D plotting in matplotlib, and Jonathon Taylor and Reinier Heeres ported it to the refactored transform trunk.

Jae-Joon Lee Implemented fancy arrows and boxes, rewrote the legend support to handle multiple columns and fancy text boxes, wrote the axes grid toolkit, and has made numerous contributions to the code and documentation

Paul Ivanov Has worked on getting matplotlib integrated better with other tools, such as Sage and IPython, and getting the test infrastructure faster, lighter and meaner. Listen to his podcast.

Tony Yu Has been involved in matplotlib since the early days, and recently has contributed stream plotting among many other improvements. He is the author of mpltools.

Michiel de Hoon Wrote and maintains the macosx backend.

Ian Thomas Contributed, among other things, the triangulation (tricolor and tripcontour) methods.

Benjamin Root Has significantly improved the capabilities of the 3D plotting. He has improved matplotlib's documentation and code quality throughout, and does invaluable triaging of pull requests and bugs.

Phil Elson Fixed some deep-seated bugs in the transforms framework, and has been laser-focused on improving polish throughout matplotlib, tackling things that have been considered to large and daunting for a long time.

Damon McDougall Added triangulated 3D surfaces and stack plots to matplotlib.

Part III

The Matplotlib FAQ

CHAPTER

NINE

INSTALLATION

Contents

9.1 Report a compilation problem

See *Getting help*.

9.2 matplotlib compiled fine, but nothing shows up when I use it

The first thing to try is a *clean install* and see if that helps. If not, the best way to test your install is by running a script, rather than working interactively from a python shell or an integrated development environment such as IDLE which add additional complexities. Open up a UNIX shell or a DOS command prompt and cd into a directory containing a minimal example in a file. Something like `simple_plot.py` for example:

```
from pylab import *
plot([1,2,3])
show()
```

and run it with:

```
python simple_plot.py --verbose-helpful
```

This will give you additional information about which backends matplotlib is loading, version information, and more. At this point you might want to make sure you understand matplotlib's *configuration* process, governed by the `matplotlibrc` configuration file which contains instructions within and the concept of the matplotlib backend.

If you are still having trouble, see *Getting help*.

9.3 How to completely remove matplotlib

Occasionally, problems with matplotlib can be solved with a clean installation of the package.

The process for removing an installation of matplotlib depends on how matplotlib was originally installed on your system. Follow the steps below that goes with your original installation method to cleanly remove matplotlib from your system.

9.3.1 Source install

Unfortunately:

```
python setup.py clean
```

does not properly clean the build directory, and does nothing to the install directory. To cleanly rebuild:

1. Delete the caches from your *.matplotlib configuration directory*.
2. Delete the `build` directory in the source tree.
3. Delete any matplotlib directories or eggs from your *installation directory*.

9.4 How to Install

9.4.1 Source install from git

Clone the main source using one of:

```
git clone git@github.com:matplotlib/matplotlib.git
```

or:

```
git clone git://github.com/matplotlib/matplotlib.git
```

and build and install as usual with:

```
> cd matplotlib
> python setup.py install
```

Note: If you are on debian/ubuntu, you can get all the dependencies required to build matplotlib with:

```
sudo apt-get build-dep python-matplotlib
```

If you are on Fedora/RedHat, you can get all the dependencies required to build matplotlib by first installing `yum-builddep` and then running:

```
su -c "yum-builddep python-matplotlib"
```

This does not build matplotlib, but it does get all of the build dependencies, which will make building from source easier.

If you want to be able to follow the development branch as it changes just replace the last step with (make sure you have **setuptools** installed):

```
> python setup.py develop
```

This creates links in the right places and installs the command line script to the appropriate places.

Note: Mac OSX users please see the *Building on OSX* guide.

Windows users please see the *Building on Windows* guide.

Then, if you want to update your matplotlib at any time, just do:

```
> git pull
```

When you run `git pull`, if the output shows that only Python files have been updated, you are all set. If C files have changed, you need to run the `python setup.py develop` command again to compile them.

There is more information on *using git* in the developer docs.

9.5 Linux Notes

Because most Linux distributions use some sort of package manager, we do not provide a pre-built binary for the Linux platform. Instead, we recommend that you use the "Add Software" method for your system to install matplotlib. This will guarantee that everything that is needed for matplotlib will be installed as well.

If, for some reason, you can not use the package manager, Linux usually comes with at least a basic build system. Follow the *instructions* found above for how to build and install matplotlib.

9.6 OS-X Notes

9.6.1 Which python for OS X?

Apple ships OS X with its own Python, in `/usr/bin/python`, and its own copy of matplotlib. Unfortunately, the way Apple currently installs its own copies of numpy, scipy and matplotlib means that these packages are difficult to upgrade (see system python packages). For that reason we strongly suggest that you install a fresh version of Python and use that as the basis for installing libraries such as numpy and matplotlib. One convenient way to install matplotlib with other useful Python software is to use one of the excellent Python scientific software collections that are now available:

- Anaconda from Continuum Analytics
- Canopy from Enthought

These collections include Python itself and a wide range of libraries; if you need a library that is not available from the collection, you can install it yourself using standard methods such as *pip*. Continuum and Enthought offer their own installation support for these collections; see the Ananconda and Canopy web pages for more information.

Other options for a fresh Python install are the standard installer from python.org, or installing Python using a general OSX package management system such as homebrew or macports. Power users on OSX will likely want one of homebrew or macports on their system to install open source software packages, but it is perfectly possible to use these systems with another source for your Python binary, such as Anaconda, Canopy or Python.org Python.

9.6.2 Installing OSX binary wheels

If you are using recent Python from http://www.python.org, Macports or Homebrew, then you can use the standard pip installer to install matplotlib binaries in the form of wheels.

Python.org Python

Install pip following the standard pip install instructions. For the impatient, open a new Terminal.app window and:

```
curl -O https://bootstrap.pypa.io/get-pip.py
```

Then (Python 2.7):

```
python get-pip.py
```

or (Python 3):

```
python3 get-pip.py
```

You can now install matplotlib and all its dependencies with:

```
pip install matplotlib
```

Macports

For Python 2.7:

```
sudo port install py27-pip
sudo pip-2.7 install matplotlib
```

For Python 3.4:

```
sudo port install py34-pip
sudo pip-3.4 install matplotlib
```

Homebrew

For Python 2.7:

```
pip2 install matplotlib
```

For Python 3.4:

```
pip3 install matplotlib
```

You might also want to install IPython; we recommend you install IPython with the IPython notebook option, like this:

- Python.org Python: `pip install ipython[notebook]`
- Macports `sudo pip-2.7 install ipython[notebook]` or `sudo pip-3.4 install ipython[notebook]`
- Homebrew `pip2 install ipython[notebook]` or `pip3 install ipython[notebook]`

Pip problems

If you get errors with pip trying to run a compiler like `gcc` or `clang`, then the first thing to try is to install xcode and retry the install. If that does not work, then check *Getting help*.

9.6.3 Installing via OSX mpkg installer package

matplotlib also has a disk image (`.dmg`) installer, which contains a typical Installer.app package to install matplotlib. You should use binary wheels instead of the disk image installer if you can, because:

- wheels work with Python.org Python, homebrew and macports, the disk image installer only works with Python.org Python.
- The disk image installer doesn't check for recent versions of packages that matplotlib depends on, and unconditionally installs the versions of dependencies contained in the disk image installer. This can overwrite packages that you have already installed, which might cause problems for other packages, if you have a pre-existing Python.org setup on your computer.

If you still want to use the disk image installer, read on.

Note: Before installing via the disk image installer, be sure that all of the packages were compiled for the same version of python. Often, the download site for NumPy and matplotlib will display a supposed 'current' version of the package, but you may need to choose a different package from the full list that was built for your combination of python and OSX.

The disk image installer will have a `.dmg` extension, and will have a name like `matplotlib-1.4.0-py2.7-macosx10.6.dmg`. The name of the installer depends on the versions of python and matplotlib it was built for, and the version of OSX that the matching Python.org installer was built for. For example, if the mathing Python.org Python installer was built for OSX 10.6 or greater, the dmg file will end in `-macosx10.6.dmg`. You need to download this disk image file, open the disk image file by double clicking, and find the new matplotlib disk image icon on your desktop. Double click on that icon to show the contents of the image. Then double-click on the `.mpkg` icon, which will have a name like `matplotlib-1.4.0-py2.7-macosx10.6.mpkg`, it will run the Installer.app, prompt you for a password if you need system-wide installation privileges, and install to a directory like `/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages` (exact path depends on your Python version).

9.6.4 Checking your installation

The new version of matplotlib should now be on your Python "path". Check this with one of these commands at the Terminal.app command line:

```
python2.7 -c 'import matplotlib; print matplotlib.__version__, matplotlib.__file__'
```

(Python 2.7) or:

```
python3.4 -c 'import matplotlib; print(matplotlib.__version__, matplotlib.__file__)'
```

(Python 3.4). You should see something like this:

```
1.4.0 /Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/
↪matplotlib/__init__.pyc
```

where `1.4.0` is the matplotlib version you just installed, and the path following depends on whether you are using Python.org Python, Homebrew or Macports. If you see another version, or you get an error like this:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named matplotlib
```

then check that the Python binary is the one you expected by doing one of these commands in Terminal.app:

```
which python2.7
```

or:

```
which python3.4
```

If you get the result `/usr/bin/python2.7`, then you are getting the Python installed with OSX, which is probably not what you want. Try closing and restarting Terminal.app before running the check again. If that doesn't fix the problem, depending on which Python you wanted to use, consider reinstalling Python.org Python, or check your homebrew or macports setup. Remember that the disk image installer only works for Python.org Python, and will not get picked up by other Pythons. If all these fail, please let us know: see *Getting help*.

9.7 Windows Notes

See *Windows*.

CHAPTER

TEN

USAGE

Contents

10.1 General Concepts

`matplotlib` has an extensive codebase that can be daunting to many new users. However, most of matplotlib can be understood with a fairly simple conceptual framework and knowledge of a few important points.

Plotting requires action on a range of levels, from the most general (e.g., 'contour this 2-D array') to the most specific (e.g., 'color this screen pixel red'). The purpose of a plotting package is to assist you in visualizing your data as easily as possible, with all the necessary control – that is, by using relatively high-level commands most of the time, and still have the ability to use the low-level commands when needed.

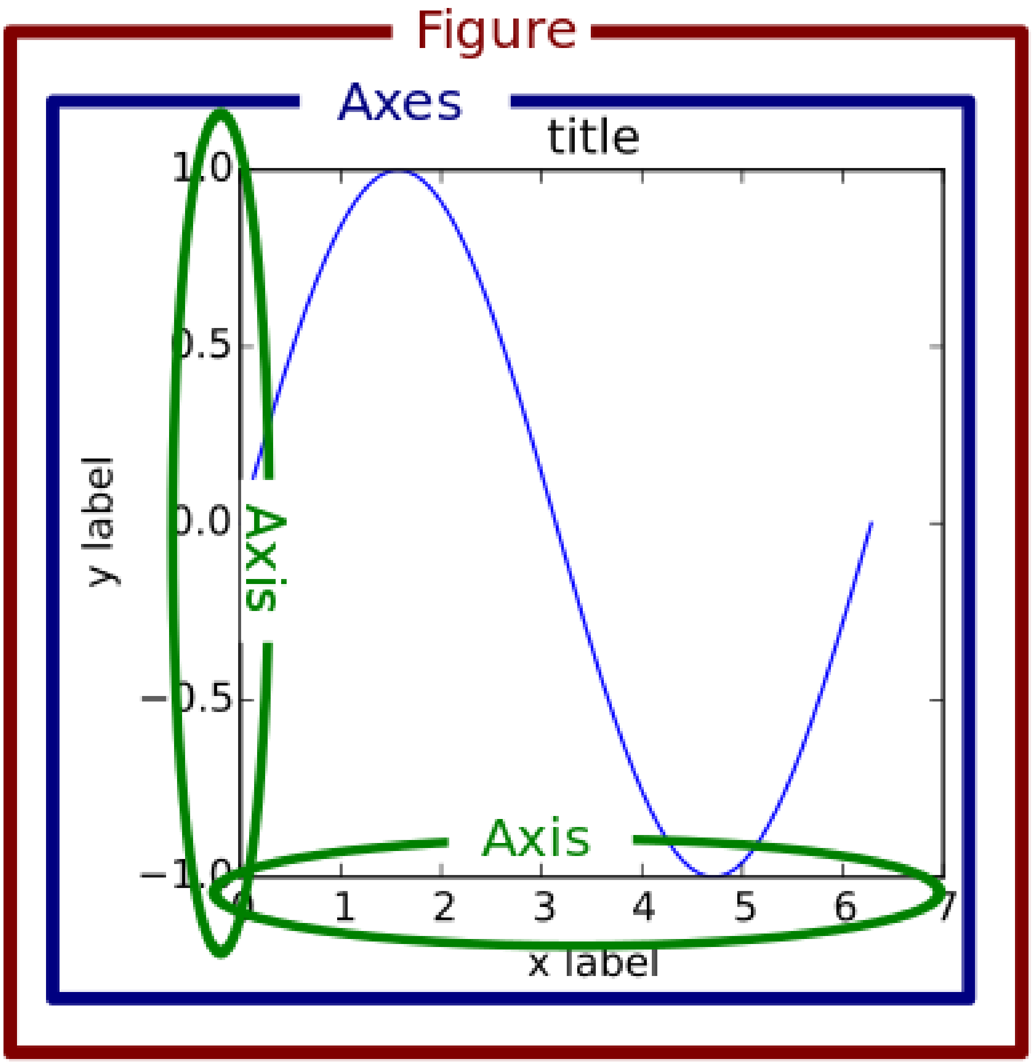
Therefore, everything in matplotlib is organized in a hierarchy. At the top of the hierarchy is the matplotlib "state-machine environment" which is provided by the `matplotlib.pyplot` module. At this level, simple functions are used to add plot elements (lines, images, text, etc.) to the current axes in the current figure.

Note: Pyplot's state-machine environment behaves similarly to MATLAB and should be most familiar to users with MATLAB experience.

The next level down in the hierarchy is the first level of the object-oriented interface, in which pyplot is used only for a few functions such as figure creation, and the user explicitly creates and keeps track of the figure and axes objects. At this level, the user uses pyplot to create figures, and through those figures, one or more axes objects can be created. These axes objects are then used for most plotting actions.

For even more control – which is essential for things like embedding matplotlib plots in GUI applications – the pyplot level may be dropped completely, leaving a purely object-oriented approach.

10.2 Parts of a Figure



10.2.1 Figure

The **whole** figure (marked as the outer red box). The figure keeps track of all the child *Axes*, a smattering of 'special' artists (titles, figure legends, etc), and the **canvas**. (Don't worry too much about the canvas, it is crucial as it is the object that actually does the drawing to get you your plot, but as the user it is more-or-less invisible to you). A figure can have any number of *Axes*, but to be useful should have at least one.

The easiest way to create a new figure is with pyplot:

```
fig = plt.figure()  # an empty figure with no axes
fig, ax_lst = plt.subplots(2, 2)  # a figure with a 2x2 grid of Axes
```

10.2.2 Axes

This is what you think of as 'a plot', it is the region of the image with the data space (marked as the inner blue box). A given figure can contain many Axes, but a given `Axes` object can only be in one `Figure`. The Axes contains two (or three in the case of 3D) `Axis` objects (be aware of the difference between **Axes** and **Axis**) which take care of the data limits (the data limits can also be controlled via set via the `set_xlim()` and `set_ylim()` `Axes` methods). Each `Axes` has a title (set via `set_title()`), an x-label (set via `set_xlabel()`), and a y-label set via `set_ylabel()`).

The `Axes` class and it's member functions are the primary entry point to working with the OO interface.

10.2.3 Axis

These are the number-line-like objects (circled in green). They take care of setting the graph limits and generating the ticks (the marks on the axis) and ticklabels (strings labeling the ticks). The location of the ticks is determined by a `Locator` object and the ticklabel strings are formatted by a `Formatter`. The combination of the correct `Locator` and `Formatter` gives very fine control over the tick locations and labels.

10.2.4 Artist

Basically everything you can see on the figure is an artist (even the `Figure`, `Axes`, and `Axis` objects). This includes `Text` objects, `Line2D` objects, `collection` objects, `Patch` objects ... (you get the idea). When the figure is rendered, all of the artists are drawn to the **canvas**. Most Artists are tied to an Axes; such an Artist cannot be shared by multiple Axes, or moved from one to another.

10.3 Types of inputs to plotting functions

All of plotting functions expect `np.array` or `np.ma.masked_array` as input. Classes that are 'array-like' such as `pandas` data objects and `np.matrix` may or may not work as intended. It is best to convert these to `np.array` objects prior to plotting.

For example, to covert a `pandas.DataFrame`

```
a = pandas.DataFrame(np.random.rand(4,5), columns = list('abcde'))
a_asndarray = a.values
```

and to covert a `np.matrix`

```
b = np.matrix([[1,2],[3,4]])
b_asarray = np.asarray(b)
```

10.4 Matplotlib, pyplot and pylab: how are they related?

Matplotlib is the whole package; *matplotlib.pyplot* is a module in matplotlib; and `pylab` is a module that gets installed alongside `matplotlib`.

Pyplot provides the state-machine interface to the underlying object-oriented plotting library. The state-machine implicitly and automatically creates figures and axes to achieve the desired plot. For example:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')

plt.title("Simple Plot")

plt.legend()

plt.show()
```

The first call to `plt.plot` will automatically create the necessary figure and axes to achieve the desired plot. Subsequent calls to `plt.plot` re-use the current axes and each add another line. Setting the title, legend, and axis labels also automatically use the current axes and set the title, create the legend, and label the axis respectively.

`pylab` is a convenience module that bulk imports *matplotlib.pyplot* (for plotting) and `numpy` (for mathematics and working with arrays) in a single name space. Although many examples use `pylab`, it is no longer recommended.

For non-interactive plotting it is suggested to use pyplot to create the figures and then the OO interface for plotting.

10.5 Coding Styles

When viewing this documentation and examples, you will find different coding styles and usage patterns. These styles are perfectly valid and have their pros and cons. Just about all of the examples can be converted into another style and achieve the same results. The only caveat is to avoid mixing the coding styles for your own code.

Note: Developers for matplotlib have to follow a specific style and guidelines. See *The Matplotlib Developers' Guide*.

Of the different styles, there are two that are officially supported. Therefore, these are the preferred ways to use matplotlib.

For the pyplot style, the imports at the top of your scripts will typically be:

```
import matplotlib.pyplot as plt
import numpy as np
```

Then one calls, for example, np.arange, np.zeros, np.pi, plt.figure, plt.plot, plt.show, etc. Use the pyplot interface for creating figures, and then use the object methods for the rest:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.2)
y = np.sin(x)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x, y)
plt.show()
```

So, why all the extra typing instead of the MATLAB-style (which relies on global state and a flat namespace)? For very simple things like this example, the only advantage is academic: the wordier styles are more explicit, more clear as to where things come from and what is going on. For more complicated applications, this explicitness and clarity becomes increasingly valuable, and the richer and more complete object-oriented interface will likely make the program easier to write and maintain.

Typically one finds oneself making the same plots over and over again, but with different data sets, which leads to needing to write specialized functions to do the plotting. The recommended function signature is something like:

```
def my_plotter(ax, data1, data2, param_dict):
    """
    A helper function to make a graph

    Parameters
    ----------
    ax : Axes
        The axes to draw to

    data1 : array
       The x data

    data2 : array
       The y data

    param_dict : dict
       Dictionary of kwargs to pass to ax.plot

    Returns
    -------
    out : list
        list of artists added
    """
```

```
    out = ax.plot(data1, data2, **param_dict)
    return out
```

which you would then use as:

```
fig, ax = plt.subplots(1, 1)
my_plotter(ax, data1, data2, {'marker':'x'})
```

or if you wanted to have 2 sub-plots:

```
fig, (ax1, ax2) = plt.subplots(1, 2)
my_plotter(ax1, data1, data2, {'marker':'x'})
my_plotter(ax2, data3, data4, {'marker':'o'})
```

Again, for these simple examples this style seems like overkill, however once the graphs get slightly more complex it pays off.

10.6 What is a backend?

A lot of documentation on the website and in the mailing lists refers to the "backend" and many new users are confused by this term. matplotlib targets many different use cases and output formats. Some people use matplotlib interactively from the python shell and have plotting windows pop up when they type commands. Some people embed matplotlib into graphical user interfaces like wxpython or pygtk to build rich applications. Others use matplotlib in batch scripts to generate postscript images from some numerical simulations, and still others in web application servers to dynamically serve up graphs.

To support all of these use cases, matplotlib can target different outputs, and each of these capabilities is called a backend; the "frontend" is the user facing code, i.e., the plotting code, whereas the "backend" does all the hard work behind-the-scenes to make the figure. There are two types of backends: user interface backends (for use in pygtk, wxpython, tkinter, qt4, or macosx; also referred to as "interactive backends") and hardcopy backends to make image files (PNG, SVG, PDF, PS; also referred to as "non-interactive backends").

There are four ways to configure your backend. If they conflict each other, the method mentioned last in the following list will be used, e.g. calling `use()` will override the setting in your `matplotlibrc`.

1. The `backend` parameter in your `matplotlibrc` file (see *Customizing matplotlib*):

   ```
   backend : WXAgg   # use wxpython with antigrain (agg) rendering
   ```

2. Setting the `MPLBACKEND` environment variable, either for your current shell or for a single script:

   ```
   > export MPLBACKEND="module://my_backend"
   > python simple_plot.py

   > MPLBACKEND="module://my_backend" python simple_plot.py
   ```

   Setting this environment variable will override the `backend` parameter in *any* `matplotlibrc`, even if there is a `matplotlibrc` in your current working directory. Therefore setting `MPLBACKEND` globally, e.g. in your `.bashrc` or `.profile`, is discouraged as it might lead to counter-intuitive behavior.

3. To set the backend for a single script, you can alternatively use the `-d` command line argument:

```
> python script.py -dbackend
```

   This method is **deprecated** as the `-d` argument might conflict with scripts which parse command line arguments (see issue #1986). You should use *MPLBACKEND* instead.

4. If your script depends on a specific backend you can use the `use()` function:

```
import matplotlib
matplotlib.use('PS')   # generate postscript output by default
```

   If you use the `use()` function, this must be done before importing `matplotlib.pyplot`. Calling `use()` after pyplot has been imported will have no effect. Using `use()` will require changes in your code if users want to use a different backend. Therefore, you should avoid explicitly calling `use()` unless absolutely necessary.

Note: Backend name specifications are not case-sensitive; e.g., 'GTKAgg' and 'gtkagg' are equivalent.

With a typical installation of matplotlib, such as from a binary installer or a linux distribution package, a good default backend will already be set, allowing both interactive work and plotting from scripts, with output to the screen and/or to a file, so at least initially you will not need to use any of the methods given above.

If, however, you want to write graphical user interfaces, or a web application server (*Matplotlib in a web application server*), or need a better understanding of what is going on, read on. To make things a little more customizable for graphical user interfaces, matplotlib separates the concept of the renderer (the thing that actually does the drawing) from the canvas (the place where the drawing goes). The canonical renderer for user interfaces is `Agg` which uses the Anti-Grain Geometry C++ library to make a raster (pixel) image of the figure. All of the user interfaces except `macosx` can be used with agg rendering, e.g., `WXAgg`, `GTKAgg`, `QT4Agg`, `TkAgg`. In addition, some of the user interfaces support other rendering engines. For example, with GTK, you can also select GDK rendering (backend `GTK`) or Cairo rendering (backend `GTKCairo`).

For the rendering engines, one can also distinguish between vector or raster renderers. Vector graphics languages issue drawing commands like "draw a line from this point to this point" and hence are scale free, and raster backends generate a pixel representation of the line whose accuracy depends on a DPI setting.

Here is a summary of the matplotlib renderers (there is an eponymous backed for each; these are *non-interactive backends*, capable of writing to a file):

Renderer	Filetypes	Description
AGG	*png*	*raster graphics* – high quality images using the Anti-Grain Geometry engine
PS	*ps eps*	*vector graphics* – Postscript output
PDF	*pdf*	*vector graphics* – Portable Document Format
SVG	*svg*	*vector graphics* – Scalable Vector Graphics
Cairo	*png ps pdf svg* ...	*vector graphics* – Cairo graphics
GDK	*png jpg tiff* ...	*raster graphics* – the Gimp Drawing Kit

And here are the user interfaces and renderer combinations supported; these are *interactive backends*, capable of displaying to the screen and of using appropriate renderers from the table above to write to a file:

Backend	Description
GTK-Agg	Agg rendering to a *GTK* 2.x canvas (requires PyGTK and pycairo or cairocffi; Python2 only)
GTK3Agg	Agg rendering to a *GTK* 3.x canvas (requires PyGObject and pycairo or cairocffi)
GTK	GDK rendering to a *GTK* 2.x canvas (not recommended) (requires PyGTK and pycairo or cairocffi; Python2 only)
GTK-Cairo	Cairo rendering to a *GTK* 2.x canvas (requires PyGTK and pycairo or cairocffi; Python2 only)
GTK3Cairo	Cairo rendering to a *GTK* 3.x canvas (requires PyGObject and pycairo or cairocffi)
WXAgg	Agg rendering to to a *wxWidgets* canvas (requires wxPython)
WX	Native *wxWidgets* drawing to a *wxWidgets* Canvas (not recommended) (requires wxPython)
TkAgg	Agg rendering to a *Tk* canvas (requires TkInter)
Qt4Agg	Agg rendering to a *Qt4* canvas (requires PyQt4 or `pyside`)
Qt5Agg	Agg rendering in a *Qt5* canvas (requires PyQt5)
macosx	Cocoa rendering in OSX windows (presently lacks blocking show() behavior when matplotlib is in non-interactive mode)

10.7 WX backends

At present the release version of `wxPython` (also known as wxPython classic) does not support python3. A work in progress redesigned version known as wxPython-Phoenix does support python3. Matplotlib should work with both versions.

10.8 GTK and Cairo

Both `GTK2` and `GTK3` have implicit dependencies on PyCairo regardless of the specific Matplotlib backend used. Unfortunatly the latest release of PyCairo for Python3 does not implement the Python wrappers needed for the `GTK3Agg` backend. `Cairocffi` can be used as a replacement which implements the correct wrapper.

10.9 How do I select PyQt4 or PySide?

You can choose either PyQt4 or PySide when using the `qt4` backend by setting the appropriate value for `backend.qt4` in your `matplotlibrc` file. The default value is `PyQt4`.

The setting in your `matplotlibrc` file can be overridden by setting the `QT_API` environment variable to either `pyqt` or `pyside` to use `PyQt4` or `PySide`, respectively.

Since the default value for the bindings to be used is `PyQt4`, `matplotlib` first tries to import it, if the import fails, it tries to import `PySide`.

10.10 What is interactive mode?

Use of an interactive backend (see *What is a backend?*) permits–but does not by itself require or ensure–plotting to the screen. Whether and when plotting to the screen occurs, and whether a script or shell session continues after a plot is drawn on the screen, depends on the functions and methods that are called, and on a state variable that determines whether matplotlib is in "interactive mode". The default Boolean value is set by the `matplotlibrc` file, and may be customized like any other configuration parameter (see *Customizing matplotlib*). It may also be set via `matplotlib.interactive()`, and its value may be queried via `matplotlib.is_interactive()`. Turning interactive mode on and off in the middle of a stream of plotting commands, whether in a script or in a shell, is rarely needed and potentially confusing, so in the following we will assume all plotting is done with interactive mode either on or off.

Note: Major changes related to interactivity, and in particular the role and behavior of `show()`, were made in the transition to matplotlib version 1.0, and bugs were fixed in 1.0.1. Here we describe the version 1.0.1 behavior for the primary interactive backends, with the partial exception of *macosx*.

Interactive mode may also be turned on via `matplotlib.pyplot.ion()`, and turned off via `matplotlib.pyplot.ioff()`.

Note: Interactive mode works with suitable backends in ipython and in the ordinary python shell, but it does *not* work in the IDLE IDE. If the default backend does not support interactivity, an interactive backend can be explicitly activated using any of the methods discussed in *What is a backend?*.

10.10.1 Interactive example

From an ordinary python prompt, or after invoking ipython with no options, try this:

```
import matplotlib.pyplot as plt
plt.ion()
plt.plot([1.6, 2.7])
```

Assuming you are running version 1.0.1 or higher, and you have an interactive backend installed and selected by default, you should see a plot, and your terminal prompt should also be active; you can type additional commands such as:

```
plt.title("interactive test")
plt.xlabel("index")
```

and you will see the plot being updated after each line. This is because you are in interactive mode *and* you are using pyplot functions. Now try an alternative method of modifying the plot. Get a reference to the `Axes` instance, and call a method of that instance:

```
ax = plt.gca()
ax.plot([3.1, 2.2])
```

Nothing changed, because the Axes methods do not include an automatic call to `draw_if_interactive()`; that call is added by the pyplot functions. If you are using methods, then when you want to update the plot on the screen, you need to call *draw()*:

```
plt.draw()
```

Now you should see the new line added to the plot.

10.10.2 Non-interactive example

Start a fresh session as in the previous example, but now turn interactive mode off:

```
import matplotlib.pyplot as plt
plt.ioff()
plt.plot([1.6, 2.7])
```

Nothing happened–or at least nothing has shown up on the screen (unless you are using *macosx* backend, which is anomalous). To make the plot appear, you need to do this:

```
plt.show()
```

Now you see the plot, but your terminal command line is unresponsive; the `show()` command *blocks* the input of additional commands until you manually kill the plot window.

What good is this–being forced to use a blocking function? Suppose you need a script that plots the contents of a file to the screen. You want to look at that plot, and then end the script. Without some blocking command such as show(), the script would flash up the plot and then end immediately, leaving nothing on the screen.

In addition, non-interactive mode delays all drawing until show() is called; this is more efficient than redrawing the plot each time a line in the script adds a new feature.

Prior to version 1.0, show() generally could not be called more than once in a single script (although sometimes one could get away with it); for version 1.0.1 and above, this restriction is lifted, so one can write a script like this:

```
import numpy as np
import matplotlib.pyplot as plt
plt.ioff()
for i in range(3):
    plt.plot(np.random.rand(10))
    plt.show()
```

which makes three plots, one at a time.

10.10.3 Summary

In interactive mode, pyplot functions automatically draw to the screen.

When plotting interactively, if using object method calls in addition to pyplot functions, then call *draw()* whenever you want to refresh the plot.

Use non-interactive mode in scripts in which you want to generate one or more figures and display them before ending or generating a new set of figures. In that case, use `show()` to display the figure(s) and to block execution until you have manually destroyed them.

CHAPTER

ELEVEN

HOW-TO

Contents

11.1 Plotting: howto

11.1.1 Find all objects in a figure of a certain type

Every matplotlib artist (see *Artist tutorial*) has a method called *findobj()* that can be used to recursively search the artist for any artists it may contain that meet some criteria (e.g., match all *Line2D* instances or match some arbitrary filter function). For example, the following snippet finds every object in the figure which has a `set_color` property and makes the object blue:

```
def myfunc(x):
    return hasattr(x, 'set_color')

for o in fig.findobj(myfunc):
    o.set_color('blue')
```

You can also filter on class instances:

```
import matplotlib.text as text
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')
```

11.1.2 How to prevent ticklabels from having an offset

The default formatter will use an offset to reduce the length of the ticklabels. To turn this feature off on a per-axis basis:

```
ax.get_xaxis().get_major_formatter().set_useOffset(False)
```

set the rcParam `axes.formatter.useoffset`, or use a different formatter. See *ticker* for details.

11.1.3 Save transparent figures

The *savefig()* command has a keyword argument *transparent* which, if 'True', will make the figure and axes backgrounds transparent when saving, but will not affect the displayed image on the screen.

If you need finer grained control, e.g., you do not want full transparency or you want to affect the screen displayed version as well, you can set the alpha properties directly. The figure has a `Rectangle` instance called *patch* and the axes has a Rectangle instance called *patch*. You can set any property on them directly (*facecolor*, *edgecolor*, *linewidth*, *linestyle*, *alpha*). e.g.:

```
fig = plt.figure()
fig.patch.set_alpha(0.5)
ax = fig.add_subplot(111)
ax.patch.set_alpha(0.5)
```

If you need *all* the figure elements to be transparent, there is currently no global alpha setting, but you can set the alpha channel on individual elements, e.g.:

```
ax.plot(x, y, alpha=0.5)
ax.set_xlabel('volts', alpha=0.5)
```

11.1.4 Save multiple plots to one pdf file

Many image file formats can only have one image per file, but some formats support multi-page files. Currently only the pdf backend has support for this. To make a multi-page pdf file, first initialize the file:

```
from matplotlib.backends.backend_pdf import PdfPages
pp = PdfPages('multipage.pdf')
```

You can give the `PdfPages` object to `savefig()`, but you have to specify the format:

```
plt.savefig(pp, format='pdf')
```

An easier way is to call `PdfPages.savefig`:

```
pp.savefig()
```

Finally, the multipage pdf object has to be closed:

```
pp.close()
```

11.1.5 Move the edge of an axes to make room for tick labels

For subplots, you can control the default spacing on the left, right, bottom, and top as well as the horizontal and vertical spacing between multiple rows and columns using the `matplotlib.figure.Figure.subplots_adjust()` method (in pyplot it is `subplots_adjust()`). For example, to move the bottom of the subplots up to make room for some rotated x tick labels:

```
fig = plt.figure()
fig.subplots_adjust(bottom=0.2)
ax = fig.add_subplot(111)
```

You can control the defaults for these parameters in your `matplotlibrc` file; see *Customizing matplotlib*. For example, to make the above setting permanent, you would set:

```
figure.subplot.bottom : 0.2   # the bottom of the subplots of the figure
```

The other parameters you can configure are, with their defaults

***left* = 0.125** the left side of the subplots of the figure

***right* = 0.9** the right side of the subplots of the figure

***bottom* = 0.1** the bottom of the subplots of the figure

***top* = 0.9** the top of the subplots of the figure

***wspace* = 0.2** the amount of width reserved for blank space between subplots

***hspace* = 0.2** the amount of height reserved for white space between subplots

If you want additional control, you can create an `Axes` using the `axes()` command (or equivalently the figure `add_axes()` method), which allows you to specify the location explicitly:

```
ax = fig.add_axes([left, bottom, width, height])
```

where all values are in fractional (0 to 1) coordinates. See *pylab_examples example code: axes_demo.py* for an example of placing axes manually.

11.1.6 Automatically make room for tick labels

Note: This is now easier to handle than ever before. Calling `tight_layout()` can fix many common layout issues. See the *Tight Layout guide*.

The information below is kept here in case it is useful for other purposes.

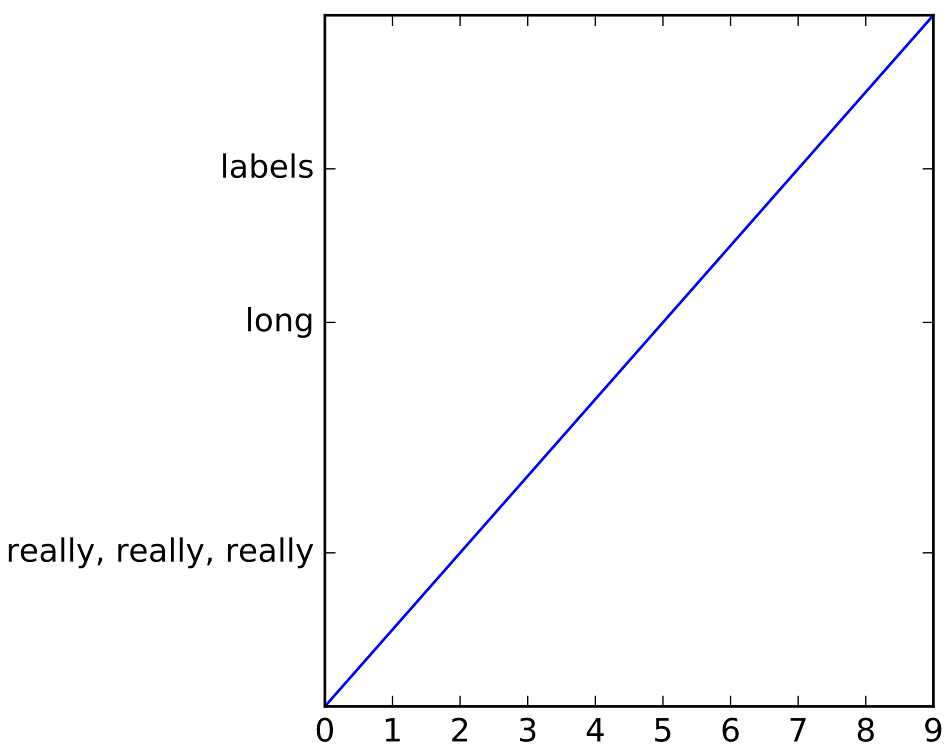
In most use cases, it is enough to simply change the subplots adjust parameters as described in *Move the edge of an axes to make room for tick labels*. But in some cases, you don't know ahead of time what your tick labels will be, or how large they will be (data and labels outside your control may be being fed into your graphing application), and you may need to automatically adjust your subplot parameters based on the size of the tick labels. Any `Text` instance can report its extent in window coordinates (a negative x coordinate is outside the window), but there is a rub.

The `RendererBase` instance, which is used to calculate the text size, is not known until the figure is drawn (`draw()`). After the window is drawn and the text instance knows its renderer, you can call `get_window_extent()`. One way to solve this chicken and egg problem is to wait until the figure is draw by connecting (`mpl_connect()`) to the "on_draw" signal (`DrawEvent`) and get the window extent there, and then do something with it, e.g., move the left of the canvas over; see *Event handling and picking*.

Here is an example that gets a bounding box in relative figure coordinates (0..1) of each of the labels and uses it to move the left of the subplots over so that the tick labels fit in the figure

```
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
fig = plt.figure()
ax = fig.add_subplot(111)
```

```
ax.plot(range(10))
ax.set_yticks((2,5,7))
labels = ax.set_yticklabels(('really, really, really', 'long', 'labels'))

def on_draw(event):
   bboxes = []
   for label in labels:
       bbox = label.get_window_extent()
       # the figure transform goes from relative coords->pixels and we
       # want the inverse of that
       bboxi = bbox.inverse_transformed(fig.transFigure)
       bboxes.append(bboxi)

   # this is the bbox that bounds all the bboxes, again in relative
   # figure coords
   bbox = mtransforms.Bbox.union(bboxes)
   if fig.subplotpars.left < bbox.width:
       # we need to move it over
       fig.subplots_adjust(left=1.1*bbox.width) # pad a little
       fig.canvas.draw()
   return False

fig.canvas.mpl_connect('draw_event', on_draw)

plt.show()
```



11.1.7 Configure the tick linewidths

In matplotlib, the ticks are *markers*. All `Line2D` objects support a line (solid, dashed, etc) and a marker (circle, square, tick). The tick linewidth is controlled by the "markeredgewidth" property:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(range(10))

for line in ax.get_xticklines() + ax.get_yticklines():
    line.set_markersize(10)

plt.show()
```

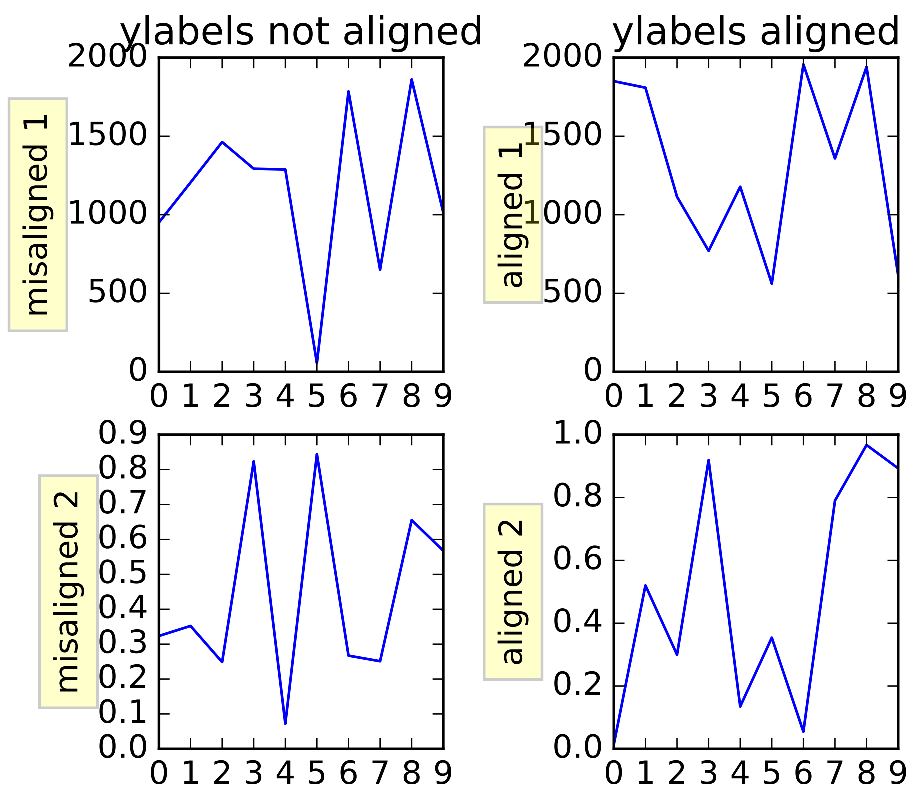
The other properties that control the tick marker, and all markers, are `markerfacecolor`, `markeredgecolor`, `markeredgewidth`, `markersize`. For more information on configuring ticks, see *Axis containers* and *Tick containers*.

11.1.8 Align my ylabels across multiple subplots

If you have multiple subplots over one another, and the y data have different scales, you can often get ylabels that do not align vertically across the multiple subplots, which can be unattractive. By default, matplotlib positions the x location of the ylabel so that it does not overlap any of the y ticks. You can override this default behavior by specifying the coordinates of the label. The example below shows the default behavior in the left subplots, and the manual setting in the right subplots.

```
import numpy as np
import matplotlib.pyplot as plt

box = dict(facecolor='yellow', pad=5, alpha=0.2)

fig = plt.figure()
fig.subplots_adjust(left=0.2, wspace=0.6)


ax1 = fig.add_subplot(221)
ax1.plot(2000*np.random.rand(10))
ax1.set_title('ylabels not aligned')
ax1.set_ylabel('misaligned 1', bbox=box)
ax1.set_ylim(0, 2000)
ax3 = fig.add_subplot(223)
ax3.set_ylabel('misaligned 2',bbox=box)
ax3.plot(np.random.rand(10))


labelx = -0.3  # axes coords

ax2 = fig.add_subplot(222)
ax2.set_title('ylabels aligned')
ax2.plot(2000*np.random.rand(10))
```

```
ax2.set_ylabel('aligned 1', bbox=box)
ax2.yaxis.set_label_coords(labelx, 0.5)
ax2.set_ylim(0, 2000)

ax4 = fig.add_subplot(224)
ax4.plot(np.random.rand(10))
ax4.set_ylabel('aligned 2', bbox=box)
ax4.yaxis.set_label_coords(labelx, 0.5)


plt.show()
```



11.1.9 Skip dates where there is no data

When plotting time series, e.g., financial time series, one often wants to leave out days on which there is no data, e.g., weekends. By passing in dates on the x-xaxis, you get large horizontal gaps on periods when there is not data. The solution is to pass in some proxy x-data, e.g., evenly sampled indices, and then use a custom formatter to format these as dates. The example below shows how to use an ‘index formatter’ to achieve the desired plot:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
```

```
import matplotlib.ticker as ticker

r = mlab.csv2rec('../data/aapl.csv')
r.sort()
r = r[-30:]  # get the last 30 days

N = len(r)
ind = np.arange(N)  # the evenly spaced plot indices

def format_date(x, pos=None):
    thisind = np.clip(int(x+0.5), 0, N-1)
    return r.date[thisind].strftime('%Y-%m-%d')

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(ind, r.adj_close, 'o-')
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
fig.autofmt_xdate()

plt.show()
```

11.1.10 Control the depth of plot elements

Within an axes, the order that the various lines, markers, text, collections, etc appear is determined by the *set_zorder()* property. The default order is patches, lines, text, with collections of lines and collections of patches appearing at the same level as regular lines and patches, respectively:

```
line, = ax.plot(x, y, zorder=10)
```

You can also use the Axes property *set_axisbelow()* to control whether the grid lines are placed above or below your other plot elements.

11.1.11 Make the aspect ratio for plots equal

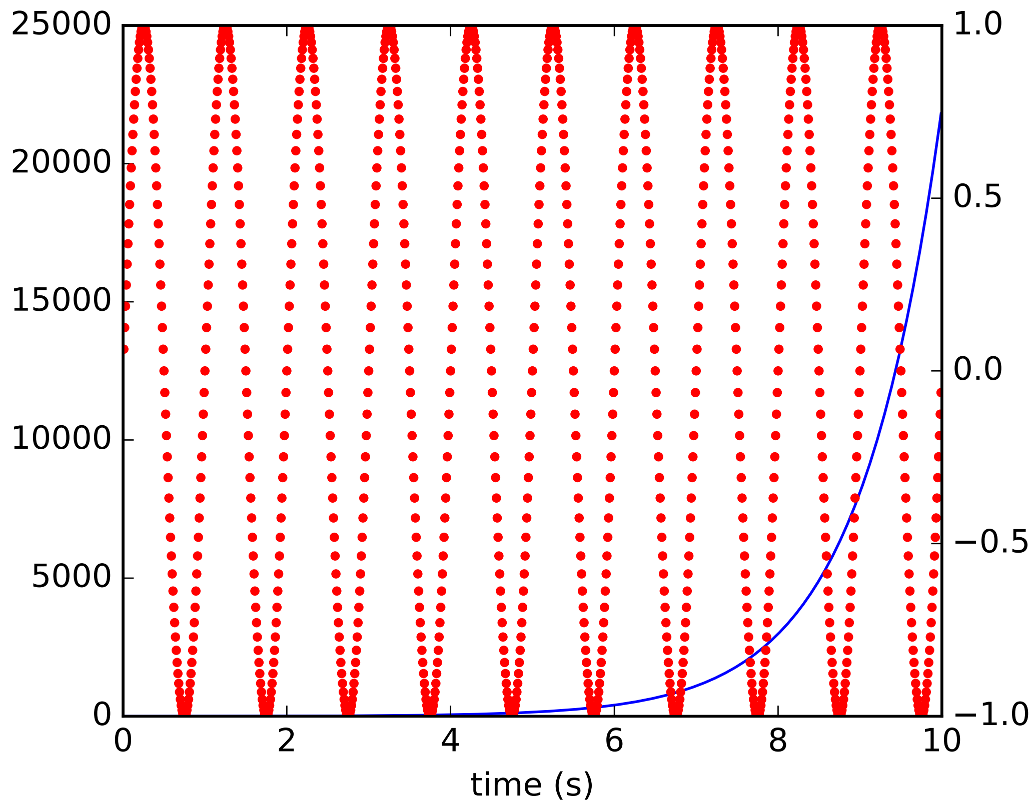
The Axes property *set_aspect()* controls the aspect ratio of the axes. You can set it to be 'auto', 'equal', or some ratio which controls the ratio:

```
ax = fig.add_subplot(111, aspect='equal')
```

11.1.12 Multiple y-axis scales

A frequent request is to have two scales for the left and right y-axis, which is possible using *twinx()* (more than two scales are not currently supported, though it is on the wish list). This works pretty well, though there are some quirks when you are trying to interactively pan and zoom, because both scales do not get the signals.

The approach uses *twinx()* (and its sister *twiny()*) to use *2 different axes*, turning the axes rectangular frame off on the 2nd axes to keep it from obscuring the first, and manually setting the tick locs and labels as desired. You can use separate matplotlib.ticker formatters and locators as desired because the two axes are independent.



11.1.13 Generate images without having a window appear

The easiest way to do this is use a non-interactive backend (see *What is a backend?*) such as Agg (for PNGs), PDF, SVG or PS. In your figure-generating script, just call the `matplotlib.use()` directive before importing pylab or pyplot:

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
plt.plot([1,2,3])
plt.savefig('myfig')
```

See also:

Matplotlib in a web application server for information about running matplotlib inside of a web application.

11.1.14 Use show()

When you want to view your plots on your display, the user interface backend will need to start the GUI mainloop. This is what `show()` does. It tells matplotlib to raise all of the figure windows created so far and start the mainloop. Because this mainloop is blocking by default (i.e., script execution is paused), you should only call this once per script, at the end. Script execution is resumed after the last window is closed. Therefore, if you are using matplotlib to generate only images and do not want a user interface window, you do not need to call `show` (see *Generate images without having a window appear* and *What is a backend?*).

Note: Because closing a figure window invokes the destruction of its plotting elements, you should call `savefig()` *before* calling `show` if you wish to save the figure as well as view it.

New in version v1.0.0: `show` now starts the GUI mainloop only if it isn't already running. Therefore, multiple calls to `show` are now allowed.

Having `show` block further execution of the script or the python interpreter depends on whether matplotlib is set for interactive mode or not. In non-interactive mode (the default setting), execution is paused until the last figure window is closed. In interactive mode, the execution is not paused, which allows you to create additional figures (but the script won't finish until the last figure window is closed).

Note: Support for interactive/non-interactive mode depends upon the backend. Until version 1.0.0 (and subsequent fixes for 1.0.1), the behavior of the interactive mode was not consistent across backends. As of v1.0.1, only the macosx backend differs from other backends because it does not support non-interactive mode.

Because it is expensive to draw, you typically will not want matplotlib to redraw a figure many times in a script such as the following:

```
plot([1,2,3])            # draw here ?
xlabel('time')           # and here ?
ylabel('volts')          # and here ?
title('a simple plot')   # and here ?
show()
```

However, it is *possible* to force matplotlib to draw after every command, which might be what you want when working interactively at the python console (see *Using matplotlib in a python shell*), but in a script you want to defer all drawing until the call to `show`. This is especially important for complex figures that take some time to draw. `show()` is designed to tell matplotlib that you're all done issuing commands and you want to draw the figure now.

Note: `show()` should typically only be called at most once per script and it should be the last line of your script. At that point, the GUI takes control of the interpreter. If you want to force a figure draw, use `draw()` instead.

Many users are frustrated by `show` because they want it to be a blocking call that raises the figure, pauses the script until they close the figure, and then allow the script to continue running until the next figure is created and the next show is made. Something like this:

```
# WARNING : illustrating how NOT to use show
for i in range(10):
```

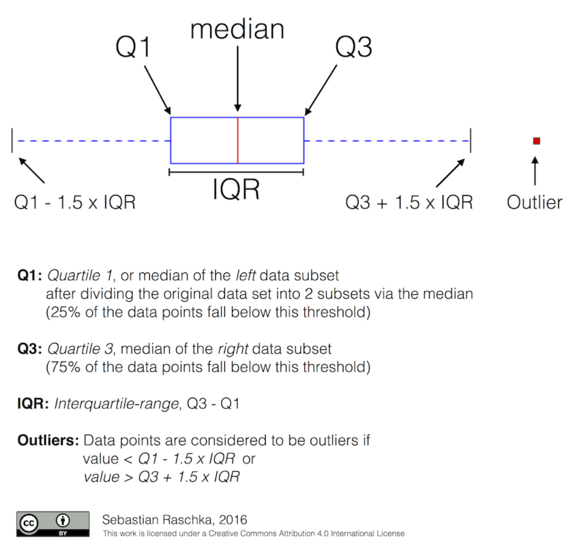
```
    # make figure i
    show()
```

This is not what show does and unfortunately, because doing blocking calls across user interfaces can be tricky, is currently unsupported, though we have made significant progress towards supporting blocking events.

New in version v1.0.0: As noted earlier, this restriction has been relaxed to allow multiple calls to `show`. In *most* backends, you can now expect to be able to create new figures and raise them in a subsequent call to `show` after closing the figures from a previous call to `show`.

11.1.15 Interpreting box plots and violin plots

Tukey's box plots (Robert McGill, John W. Tukey and Wayne A. Larsen: "The American Statistician" Vol. 32, No. 1, Feb., 1978, pp. 12-16) are statistical plots that provide useful information about the data distribution such as skewness. However, bar plots with error bars are still the common standard in most scientific literature, and thus, the interpretation of box plots can be challenging for the unfamiliar reader. The figure below illustrates the different visual features of a box plot.



Violin plots are closely related to box plots but add useful information such as the distribution of the sample data (density trace). Violin plots were added in matplotlib 1.4.

11.2 Contributing: howto

11.2.1 Request a new feature

Is there a feature you wish matplotlib had? Then ask! The best way to get started is to email the developer mailing list for discussion. This is an open source project developed primarily in the contributors free time, so there is no guarantee that your feature will be added. The *best* way to get the feature you need added is to contribute it your self.

11.2.2 Reporting a bug or submitting a patch

The development of matplotlib is organized through github. If you would like to report a bug or submit a patch please use that interface.

To report a bug create an issue on github (this requires having a github account). Please include a Short, Self Contained, Correct (Compilable), Example demonstrating what the bug is. Including a clear, easy to test example makes it easy for the developers to evaluate the bug. Expect that the bug reports will be a conversation. If you do not want to register with github, please email bug reports to the mailing list.

The easiest way to submit patches to matplotlib is through pull requests on github. Please see the *The Matplotlib Developers' Guide* for the details.

11.2.3 Contribute to matplotlib documentation

matplotlib is a big library, which is used in many ways, and the documentation has only scratched the surface of everything it can do. So far, the place most people have learned all these features are through studying the examples (*Search examples*), which is a recommended and great way to learn, but it would be nice to have more official narrative documentation guiding people through all the dark corners. This is where you come in.

There is a good chance you know more about matplotlib usage in some areas, the stuff you do every day, than many of the core developers who wrote most of the documentation. Just pulled your hair out compiling matplotlib for windows? Write a FAQ or a section for the *Installation* page. Are you a digital signal processing wizard? Write a tutorial on the signal analysis plotting functions like `xcorr()`, `psd()` and `specgram()`. Do you use matplotlib with django or other popular web application servers? Write a FAQ or tutorial and we'll find a place for it in the *User's Guide*. Bundle matplotlib in a py2exe app? ... I think you get the idea.

matplotlib is documented using the sphinx extensions to restructured text (ReST). sphinx is an extensible python framework for documentation projects which generates HTML and PDF, and is pretty easy to write; you can see the source for this document or any page on this site by clicking on the *Show Source* link at the end of the page in the sidebar (or here for this document).

The sphinx website is a good resource for learning sphinx, but we have put together a cheat-sheet at *Documenting matplotlib* which shows you how to get started, and outlines the matplotlib conventions and extensions, e.g., for including plots directly from external code in your documents.

Once your documentation contributions are working (and hopefully tested by actually *building* the docs) you can submit them as a patch against git. See *Install git* and *Reporting a bug or submitting a patch*. Looking for something to do? Search for TODO or look at the open issues on github.

11.3 Matplotlib in a web application server

Many users report initial problems trying to use maptlotlib in web application servers, because by default matplotlib ships configured to work with a graphical user interface which may require an X11 connection. Since many barebones application servers do not have X11 enabled, you may get errors if you don't configure matplotlib for use in these environments. Most importantly, you need to decide what kinds of images you want to generate (PNG, PDF, SVG) and configure the appropriate default backend. For 99% of users, this will be the Agg backend, which uses the C++ antigrain rendering engine to make nice PNGs. The Agg backend is also configured to recognize requests to generate other output formats (PDF, PS, EPS, SVG). The easiest way to configure matplotlib to use Agg is to call:

```
# do this before importing pylab or pyplot
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

For more on configuring your backend, see *What is a backend?*.

Alternatively, you can avoid pylab/pyplot altogether, which will give you a little more control, by calling the API directly as shown in *api example code: agg_oo.py*.

You can either generate hardcopy on the filesystem by calling savefig:

```
# do this before importing pylab or pyplot
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot([1,2,3])
fig.savefig('test.png')
```

or by saving to a file handle:

```
import sys
fig.savefig(sys.stdout)
```

Here is an example using Pillow. First, the figure is saved to a BytesIO object which is then fed to Pillow for further processing:

```
from io import BytesIO
from PIL import Image
imgdata = BytesIO()
fig.savefig(imgdata, format='png')
imgdata.seek(0)  # rewind the data
im = Image.open(imgdata)
```

11.3.1 matplotlib with apache

TODO; see *Contribute to matplotlib documentation*.

11.3.2 matplotlib with django

TODO; see *Contribute to matplotlib documentation*.

11.3.3 matplotlib with zope

TODO; see *Contribute to matplotlib documentation*.

11.3.4 Clickable images for HTML

Andrew Dalke of Dalke Scientific has written a nice article on how to make html click maps with matplotlib agg PNGs. We would also like to add this functionality to SVG. If you are interested in contributing to these efforts that would be great.

11.4 Search examples

The nearly 300 code *Matplotlib Examples* included with the matplotlib source distribution are full-text searchable from the search page, but sometimes when you search, you get a lot of results from the *The Matplotlib API* or other documentation that you may not be interested in if you just want to find a complete, free-standing, working piece of example code. To facilitate example searches, we have tagged every code example page with the keyword `codex` for *code example* which shouldn't appear anywhere else on this site except in the FAQ. So if you want to search for an example that uses an ellipse, search for `codex ellipse`.

11.5 Cite Matplotlib

If you want to refer to matplotlib in a publication, you can use "Matplotlib: A 2D Graphics Environment" by J. D. Hunter In Computing in Science & Engineering, Vol. 9, No. 3. (2007), pp. 90-95 (see this reference page):

```
@article{Hunter:2007,
        Address = {10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1314␣
 ↪USA},
        Author = {Hunter, John D.},
        Date-Added = {2010-09-23 12:22:10 -0700},
        Date-Modified = {2010-09-23 12:22:10 -0700},
        Isi = {000245668100019},
        Isi-Recid = {155389429},
        Journal = {Computing In Science \& Engineering},
        Month = {May-Jun},
        Number = {3},
        Pages = {90--95},
        Publisher = {IEEE COMPUTER SOC},
        Times-Cited = {21},
        Title = {Matplotlib: A 2D graphics environment},
        Type = {Editorial Material},
        Volume = {9},
```

```
        Year = {2007},
        Abstract = {Matplotlib is a 2D graphics package used for Python for application
                    development, interactive scripting, and publication-quality image
                    generation across user interfaces and operating systems.},
        Bdsk-Url-1 = {http://gateway.isiknowledge.com/gateway/Gateway.cgi?GWVersion=2&
↪SrcAuth=Alerting&SrcApp=Alerting&DestApp=WOS&DestLinkType=FullRecord;
↪KeyUT=000245668100019}}
```

CHAPTER

TWELVE

TROUBLESHOOTING

Contents

12.1 Obtaining matplotlib version

To find out your matplotlib version number, import it and print the `__version__` attribute:

```
>>> import matplotlib
>>> matplotlib.__version__
'0.98.0'
```

12.2 `matplotlib` install location

You can find what directory matplotlib is installed in by importing it and printing the `__file__` attribute:

```
>>> import matplotlib
>>> matplotlib.__file__
'/home/jdhunter/dev/lib64/python2.5/site-packages/matplotlib/__init__.pyc'
```

12.3 `matplotlib` configuration and cache directory locations

Each user has a matplotlib configuration directory which may contain a *matplotlibrc* file. To locate your `matplotlib/` configuration directory, use `matplotlib.get_configdir()`:

```
>>> import matplotlib as mpl
>>> mpl.get_configdir()
'/home/darren/.config/matplotlib'
```

On unix-like systems, this directory is generally located in your *HOME* directory under the `.config/` directory.

In addition, users have a cache directory. On unix-like systems, this is separate from the configuration directory by default. To locate your `.cache/` directory, use `matplotlib.get_cachedir()`:

```
>>> import matplotlib as mpl
>>> mpl.get_cachedir()
'/home/darren/.cache/matplotlib'
```

On windows, both the config directory and the cache directory are the same and are in your `Documents and Settings` or `Users` directory by default:

```
>>> import matplotlib
>>> mpl.get_configdir()
'C:\\Documents and Settings\\jdhunter\\.matplotlib'
>>> mpl.get_cachedir()
'C:\\Documents and Settings\\jdhunter\\.matplotlib'
```

If you would like to use a different configuration directory, you can do so by specifying the location in your *MPLCONFIGDIR* environment variable – see *Setting environment variables in Linux and OS-X*. Note that *MPLCONFIGDIR* sets the location of both the configuration directory and the cache directory.

12.4 Getting help

There are a number of good resources for getting help with matplotlib. There is a good chance your question has already been asked:

- The mailing list archive.
- Github issues.
- Stackoverflow questions tagged matplotlib.

If you are unable to find an answer to your question through search, please provide the following information in your e-mail to the mailing list:

- your operating system; (Linux/UNIX users: post the output of `uname -a`)
- matplotlib version:

```
python -c `import matplotlib; print matplotlib.__version__`
```

- where you obtained matplotlib (e.g., your Linux distribution's packages, github, PyPi, or Anaconda or Enthought Canopy).
- any customizations to your `matplotlibrc` file (see *Customizing matplotlib*).
- if the problem is reproducible, please try to provide a *minimal*, standalone Python script that demonstrates the problem. This is *the* critical step. If you can't post a piece of code that we can run and reproduce your error, the chances of getting help are significantly diminished. Very often, the mere act of trying to minimize your code to the smallest bit that produces the error will help you find a bug in *your* code that is causing the problem.
- you can get very helpful debugging output from matlotlib by running your script with a `verbose-helpful` or `--verbose-debug` flags and posting the verbose output the lists:

```
> python simple_plot.py --verbose-helpful > output.txt
```

If you compiled matplotlib yourself, please also provide

- any changes you have made to `setup.py` or `setupext.py`
- the output of:

```
rm -rf build
python setup.py build
```

  The beginning of the build output contains lots of details about your platform that are useful for the matplotlib developers to diagnose your problem.

- your compiler version – e.g., `gcc --version`

Including this information in your first e-mail to the mailing list will save a lot of time.

You will likely get a faster response writing to the mailing list than filing a bug in the bug tracker. Most developers check the bug tracker only periodically. If your problem has been determined to be a bug and can not be quickly solved, you may be asked to file a bug in the tracker so the issue doesn't get lost.

12.5 Problems with recent git versions

First make sure you have a clean build and install (see *How to completely remove matplotlib*), get the latest git update, install it and run a simple test script in debug mode:

```
rm -rf build
rm -rf /path/to/site-packages/matplotlib*
git pull
python setup.py install > build.out
python examples/pylab_examples/simple_plot.py --verbose-debug > run.out
```

and post `build.out` and `run.out` to the matplotlib-devel mailing list (please do not post git problems to the users list).

Of course, you will want to clearly describe your problem, what you are expecting and what you are getting, but often a clean build and install will help. See also *Getting help*.

CHAPTER

THIRTEEN

ENVIRONMENT VARIABLES

Contents

HOME
The user's home directory. On linux, ~ is shorthand for *HOME*.

PATH
The list of directories searched to find executable programs

PYTHONPATH
The list of directories that is added to Python's standard search list when importing packages and modules

MPLCONFIGDIR
This is the directory used to store user customizations to matplotlib, as well as some caches to improve performance. If *MPLCONFIGDIR* is not defined, *HOME*/.matplotlib is used if it is writable. Otherwise, the python standard library `tempfile.gettmpdir()` is used to find a base directory in which the `matplotlib` subdirectory is created.

MPLBACKEND
This optional variable can be set to choose the matplotlib backend. Using the `-d` command line parameter or the `use()` function will override this value. See *What is a backend?*.

13.1 Setting environment variables in Linux and OS-X

To list the current value of *PYTHONPATH*, which may be empty, try:

```
echo $PYTHONPATH
```

The procedure for setting environment variables in depends on what your default shell is. **BASH** seems to be the most common, but **CSH** is also common. You should be able to determine which by running at the command prompt:

```
echo $SHELL
```

13.1.1 BASH/KSH

To create a new environment variable:

```
export PYTHONPATH=~/Python
```

To prepend to an existing environment variable:

```
export PATH=~/bin:${PATH}
```

The search order may be important to you, do you want `~/bin` to be searched first or last? To append to an existing environment variable:

```
export PATH=${PATH}:~/bin
```

To make your changes available in the future, add the commands to your `~/.bashrc` file.

13.1.2 CSH/TCSH

To create a new environment variable:

```
setenv PYTHONPATH ~/Python
```

To prepend to an existing environment variable:

```
setenv PATH ~/bin:${PATH}
```

The search order may be important to you, do you want `~/bin` to be searched first or last? To append to an existing environment variable:

```
setenv PATH ${PATH}:~/bin
```

To make your changes available in the future, add the commands to your `~/.cshrc` file.

13.2 Setting environment variables in windows

Open the **Control Panel** (*Start* → *Control Panel*), start the **System** program. Click the *Advanced* tab and select the *Environment Variables* button. You can edit or add to the *User Variables*.

CHAPTER

FOURTEEN

WORKING WITH MATPLOTLIB IN VIRTUAL ENVIRONMENTS

Contents

14.1 Introduction

When running `matplotlib` in a virtual environment you may discover a few issues. `matplotlib` itself has no issue with virtual environments. However, the GUI frameworks that `matplotlib` uses for interactive figures have some issues with virtual environments. Everything below assumes some familiarity with the Matplotlib backends as found in *What is a backend?*.

If you only use the `IPython/Jupyter Notebook`'s `inline` and `notebook` backends and non interactive backends you should not have any issues and can ignore everything below.

14.2 GUI Frameworks

Interactive Matplotlib relies heavily on the interaction with external GUI frameworks.

Most GUI frameworks are not pip installable. This makes it tricky to install them within a virtual environment. This problem does not exist if you use Conda environments where you can install all Conda supported

GUI frameworks directly into the environment. In regular virtualenv environment various workarounds exist. Some of these are given here:

- The `TKAgg` backend doesn't require any external dependencies and is normally always available.
- The `QT4` framework `PySide` is pip installable.
- The upcoming WX Phoenix toolkit is `pip` installable.

Other frameworks are harder to install into a virtual environment. There are at least two possible ways to get access to these in a virtual environment.

One often suggested solution is to use the `--system-site-packages` option to virtualenv when creating an environment. This adds all system wide packages to the virtual environment. However, this breaks the isolation between the virtual environment and the system install. Among other issues it results in hard to debug problems with system packages shadowing the environment packages. If you use virtualenvwrapper this can be toggled with the `toggleglobalsitepackages` command.

Alternatively, you can manually symlink the GUI frameworks into the environment. I.e. to use PyQt5, you should symlink `PyQt5` and `sip` from your system site packages directory into the environment taking care that the environment and the systemwide install use the same python version.

14.3 OSX

14.3.1 Short version

If you are on Python 3, use `venv` instead of `virtualenv`:

```
python -m venv my-virtualenv
source my-virtualenv/bin/activate
```

Otherwise you will need one of the workarounds below.

14.3.2 Long version

On OSX, two different types of Python Builds exist: a regular build and a framework build. In order to interact correctly with OSX through some GUI frameworks you need a framework build of Python. At the time of writing the `macosx`, `WX` and `WXAgg` backends require a framework build to function correctly. Unfortunately virtualenv creates a non framework build even if created from a framework build of Python. Conda environments are framework builds. From Matplotlib 1.5 onwards the `macosx` backend checks that a framework build is available and fails if a non framework build is found. WX has a similar check build in.

The issue has been reported on the virtualenv bug tracker here and here

Until this is fixed, one of the following workarounds must be used:

PYTHONHOME Script

The best known workaround, borrowed from the WX wiki, is to use the non virtualenv python along with the PYTHONHOME environment variable. This can be implemented in a script as below. To use this modify `PYVER` and `PATHTOPYTHON` and put the script in the virtualenv bin directory i.e. `PATHTOVENV/bin/frameworkpython`

```
#!/bin/bash

# what real Python executable to use
PYVER=2.7
PATHTOPYTHON=/usr/local/bin/
PYTHON=${PATHTOPYTHON}python${PYVER}

# find the root of the virtualenv, it should be the parent of the dir this script is in
ENV=`$PYTHON -c "import os; print(os.path.abspath(os.path.join(os.path.dirname(\"$0\"),
↪'..')))"`

# now run Python with the virtualenv set as Python's HOME
export PYTHONHOME=$ENV
exec $PYTHON "$@"
```

With this in place you can run `frameworkpython` to get an interactive framework build within the virtualenv. To run a script you can do `frameworkpython test.py` where `test.py` is a script that requires a framework build. To run an interactive `IPython` session with the framework build within the virtual environment you can do `frameworkpython -m IPython`

PYTHONHOME Function

Alternatively you can define a function in your `.bashrc` using

```
function frameworkpython {
    if [[ ! -z "$VIRTUAL_ENV" ]]; then
        PYTHONHOME=$VIRTUAL_ENV /usr/local/bin/python "$@"
    else
        /usr/local/bin/python "$@"
    fi
}
```

This function can then be used in all of your virtualenvs without having to fix every single one of them.

PythonW Compiler

In addition virtualenv-pythonw-osx provides an alternative workaround which may be used to solve the issue.

Part IV

External Resources

CHAPTER

FIFTEEN

BOOKS, CHAPTERS AND ARTICLES

- Mastering matplotlib by Duncan M. McGreggor
- Interactive Applications Using Matplotlib by Benjamin Root
- Matplotlib for Python Developers by Sandro Tosi
- Matplotlib chapter by John Hunter and Michael Droettboom in The Architecture of Open Source Applications
- Graphics with Matplotlib by David J. Raymond
- Ten Simple Rules for Better Figures by Nicolas P. Rougier, Michael Droettboom and Philip E. Bourne

CHAPTER

SIXTEEN

VIDEOS

- Getting started with Matplotlib by unpingco
- Plotting with matplotlib by Mike Müller
- Introduction to NumPy and Matplotlib by Eric Jones
- Anatomy of Matplotlib by Benjamin Root
- Data Visualization Basics with Python (O'Reilly) by Randal S. Olson

CHAPTER

SEVENTEEN

TUTORIALS

- Matplotlib tutorial by Nicolas P. Rougier
- Anatomy of Matplotlib - IPython Notebooks by Benjamin Root

Part V

The Matplotlib Developers' Guide

CHAPTER

EIGHTEEN

CODING GUIDE

18.1 Pull request checklist

This checklist should be consulted when creating pull requests to make sure they are complete before merging. These are not intended to be rigidly followed—it's just an attempt to list in one place all of the items that are necessary for a good pull request. Of course, some items will not always apply.

18.1.1 Branch selection

- In general, simple bugfixes that are unlikely to introduce new bugs of their own should be merged onto the maintenance branch. New features, or anything that changes the API, should be made against master. The rules are fuzzy here – when in doubt, try to get some consensus.
  - Once changes are merged into the maintenance branch, they should be merged into master.

18.1.2 Style

- Formatting should follow PEP8. Exceptions to these rules are acceptable if it makes the code objectively more readable.
  - You should consider installing/enabling automatic PEP8 checking in your editor. Part of the test suite is checking PEP8 compliance, things go smoother if the code is mostly PEP8 compliant to begin with.
- No tabs (only spaces). No trailing whitespace.
  - Configuring your editor to remove these things upon saving will save a lot of trouble.
- Import the following modules using the standard scipy conventions:

```
import numpy as np
import numpy.ma as ma
import matplotlib as mpl
from matplotlib import pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.collections as mcol
import matplotlib.patches as mpatches
```

- See below for additional points about *Keyword argument processing*, if code in your pull request does that.
- Adding a new pyplot function involves generating code. See *Writing a new pyplot function* for more information.

18.1.3 Documentation

- Every new feature should be documented. If it's a new module, don't forget to add a new rst file to the API docs.
- Docstrings should be in numpydoc format. Don't be thrown off by the fact that many of the existing docstrings are not in that format; we are working to standardize on `numpydoc`.

  Docstrings should look like (at a minimum):

```
def foo(bar, baz=None):
    """
    This is a prose description of foo and all the great
    things it does.

    Parameters
    ----------
    bar : (type of bar)
        A description of bar

    baz : (type of baz), optional
        A description of baz

    Returns
    -------
    foobar : (type of foobar)
        A description of foobar
    foobaz : (type of foobaz)
        A description of foobaz
    """
    # some very clever code
    return foobar, foobaz
```

- Each high-level plotting function should have a simple example in the `Example` section of the docstring. This should be as simple as possible to demonstrate the method. More complex examples should go in the `examples` tree.
- Build the docs and make sure all formatting warnings are addressed.
- See *Documenting matplotlib* for our documentation style guide.
- If your changes are non-trivial, please make an entry in the `CHANGELOG`.
- If your change is a major new feature, add an entry to `doc/users/whats_new.rst`.
- If you change the API in a backward-incompatible way, please document it in `doc/api/api_changes.rst`.

18.1.4 Testing

Using the test framework is discussed in detail in the section *Testing*.

- If the PR is a bugfix, add a test that fails prior to the change and passes with the change. Include any relevant issue numbers in the docstring of the test.
- If this is a new feature, add a test that exercises as much of the new feature as possible. (The `--with-coverage` option may be useful here).
- Make sure the Travis tests are passing before merging.
  - The Travis tests automatically test on all of the Python versions matplotlib supports whenever a pull request is created or updated. The `tox` support in matplotlib may be useful for testing locally.

18.1.5 Installation

- If you have added new files or directories, or reorganized existing ones, make sure the new files included in the match patterns in `MANIFEST.in`, and/or in `package_data` in `setup.py`.

18.1.6 C/C++ extensions

- Extensions may be written in C or C++.
- Code style should conform to PEP7 (understanding that PEP7 doesn't address C++, but most of its admonitions still apply).
- Interfacing with Python may be done either with the raw Python/C API or Cython.
- Python/C interface code should be kept separate from the core C/C++ code. The interface code should be named `FOO_wrap.cpp` or `FOO_wrapper.cpp`.
- Header file documentation (aka docstrings) should be in Numpydoc format. We don't plan on using automated tools for these docstrings, and the Numpydoc format is well understood in the scientific Python community.

18.2 Style guide

18.2.1 Keyword argument processing

Matplotlib makes extensive use of `**kwargs` for pass-through customizations from one function to another. A typical example is in `matplotlib.pylab.text()`. The definition of the pylab text function is a simple pass-through to *`matplotlib.axes.Axes.text()`*:

```
# in pylab.py
def text(*args, **kwargs):
    ret =  gca().text(*args, **kwargs)
```

```
    draw_if_interactive()
    return ret
```

text() in simplified form looks like this, i.e., it just passes all `args` and `kwargs` on to `matplotlib.text.Text.__init__()`:

```
# in axes.py
def text(self, x, y, s, fontdict=None, withdash=False, **kwargs):
    t = Text(x=x, y=y, text=s, **kwargs)
```

and `__init__()` (again with liberties for illustration) just passes them on to the *matplotlib.artist.Artist.update()* method:

```
# in text.py
def __init__(self, x=0, y=0, text='', **kwargs):
    Artist.__init__(self)
    self.update(kwargs)
```

`update` does the work looking for methods named like `set_property` if `property` is a keyword argument. i.e., no one looks at the keywords, they just get passed through the API to the artist constructor which looks for suitably named methods and calls them with the value.

As a general rule, the use of `**kwargs` should be reserved for pass-through keyword arguments, as in the example above. If all the keyword args are to be used in the function, and not passed on, use the key/value keyword args in the function definition rather than the `**kwargs` idiom.

In some cases, you may want to consume some keys in the local function, and let others pass through. You can `pop` the ones to be used locally and pass on the rest. For example, in *plot()*, `scalex` and `scaley` are local arguments and the rest are passed on as *Line2D()* keyword arguments:

```
# in axes.py
def plot(self, *args, **kwargs):
    scalex = kwargs.pop('scalex', True)
    scaley = kwargs.pop('scaley', True)
    if not self._hold: self.cla()
    lines = []
    for line in self._get_lines(*args, **kwargs):
        self.add_line(line)
        lines.append(line)
```

Note: there is a use case when `kwargs` are meant to be used locally in the function (not passed on), but you still need the `**kwargs` idiom. That is when you want to use `*args` to allow variable numbers of non-keyword args. In this case, python will not allow you to use named keyword args after the `*args` usage, so you will be forced to use `**kwargs`. An example is `matplotlib.contour.ContourLabeler.clabel()`:

```
# in contour.py
def clabel(self, *args, **kwargs):
    fontsize = kwargs.get('fontsize', None)
    inline = kwargs.get('inline', 1)
    self.fmt = kwargs.get('fmt', '%1.3f')
    colors = kwargs.get('colors', None)
    if len(args) == 0:
```

```
        levels = self.levels
        indices = range(len(self.levels))
    elif len(args) == 1:
       ...etc...
```

18.3 Hints

This section describes how to add certain kinds of new features to matplotlib.

18.3.1 Developing a new backend

If you are working on a custom backend, the *backend* setting in `matplotlibrc` (*Customizing matplotlib*) supports an external backend via the `module` directive. if `my_backend.py` is a matplotlib backend in your *PYTHONPATH*, you can set use it on one of several ways

- in matplotlibrc:

  ```
  backend : module://my_backend
  ```

- with the *MPLBACKEND* environment variable:

  ```
  > export MPLBACKEND="module://my_backend"
  > python simple_plot.py
  ```

- from the command shell with the `-d` flag:

  ```
  > python simple_plot.py -dmodule://my_backend
  ```

- with the use directive in your script:

  ```
  import matplotlib
  matplotlib.use('module://my_backend')
  ```

18.3.2 Writing examples

We have hundreds of examples in subdirectories of `matplotlib/examples`, and these are automatically generated when the website is built to show up both in the examples and gallery sections of the website.

Any sample data that the example uses should be kept small and distributed with matplotlib in the `lib/matplotlib/mpl-data/sample_data/` directory. Then in your example code you can load it into a file handle with:

```
import matplotlib.cbook as cbook
fh = cbook.get_sample_data('mydata.dat')
```

18.3.3 Writing a new pyplot function

A large portion of the pyplot interface is automatically generated by the `boilerplate.py` script (in the root of the source tree). To add or remove a plotting method from pyplot, edit the appropriate list in `boilerplate.py` and then run the script which will update the content in `lib/matplotlib/pyplot.py`. Both the changes in `boilerplate.py` and `lib/matplotlib/pyplot.py` should be checked into the repository.

Note: boilerplate.py looks for changes in the installed version of matplotlib and not the source tree. If you expect the pyplot.py file to show your new changes, but they are missing, this might be the cause.

Install your new files by running `python setup.py build` and `python setup.py install` followed by `python boilerplate.py`. The new pyplot.py file should now have the latest changes.

CHAPTER

NINETEEN

WRITING CODE FOR PYTHON 2 AND 3

As of matplotlib 1.4, the six library is used to support Python 2 and 3 from a single code base. The 2to3 tool is no longer used.

This document describes some of the issues with that approach and some recommended solutions. It is not a complete guide to Python 2 and 3 compatibility.

19.1 Welcome to the __future__

The top of every py file should include the following:

```
from __future__ import (absolute_import, division,
                        print_function, unicode_literals)
import six
```

This will make the Python 2 interpreter behave as close to Python 3 as possible.

All matplotlib files should also import six, whether they are using it or not, just to make moving code between modules easier, as six gets used *a lot*.

19.2 Finding places to use six

The only way to make sure code works on both Python 2 and 3 is to make sure it is covered by unit tests.

However, the 2to3 commandline tool can also be used to locate places that require special handling with six.

(The modernize tool may also be handy, though I've never used it personally).

The six documentation serves as a good reference for the sorts of things that need to be updated.

19.3 The dreaded \u escapes

When from __future__ import unicode_literals is used, all string literals (not preceded with a b) will become unicode literals.

Normally, one would use "raw" string literals to encode strings that contain a lot of slashes that we don't want Python to interpret as special characters. A common example in matplotlib is when it deals with TeX and has to represent things like `r"\usepackage{foo}"`. Unfortunately, on Python 2there is no way to represent `u` in a raw unicode string literal, since it will always be interpreted as the start of a unicode character escape, such as `u20af`. The only solution is to use a regular (non-raw) string literal and repeat all slashes, e.g. `"\\usepackage{foo}"`.

The following shows the problem on Python 2:

```
>>> ur'\u'
  File "<stdin>", line 1
SyntaxError: (unicode error) 'rawunicodeescape' codec can't decode bytes in
position 0-1: truncated \uXXXX
>>> ur'\\u'
u'\\\\u'
>>> u'\u'
  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
position 0-1: truncated \uXXXX escape
>>> u'\\u'
u'\\u'
```

This bug has been fixed in Python 3, however, we can't take advantage of that and still support Python 2:

```
>>> r'\u'
'\\u'
>>> r'\\u'
'\\\\u'
>>> '\u'
  File "<stdin>", line 1
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
position 0-1: truncated \uXXXX escape
>>> '\\u'
'\\u'
```

19.4 Iteration

The behavior of the methods for iterating over the items, values and keys of a dictionary has changed in Python 3. Additionally, other built-in functions such as `zip`, `range` and `map` have changed to return iterators rather than temporary lists.

In many cases, the performance implications of iterating vs. creating a temporary list won't matter, so it's tempting to use the form that is simplest to read. However, that results in code that behaves differently on Python 2 and 3, leading to subtle bugs that may not be detected by the regression tests. Therefore, unless the loop in question is provably simple and doesn't call into other code, the `six` versions that ensure the same behavior on both Python 2 and 3 should be used. The following table shows the mapping of equivalent semantics between Python 2, 3 and six for `dict.items()`:

Python 2	Python 3	six
`d.items()`	`list(d.items())`	`list(six.iteritems(d))`
`d.iteritems()`	`d.items()`	`six.iteritems(d)`

19.5 Numpy-specific things

When specifying dtypes, all strings must be byte strings on Python 2 and unicode strings on Python 3. The best way to handle this is to force cast them using `str()`. The same is true of structure specifiers in the `struct` built-in module.

CHAPTER

TWENTY

LICENSES

Matplotlib only uses BSD compatible code. If you bring in code from another project make sure it has a PSF, BSD, MIT or compatible license (see the Open Source Initiative licenses page for details on individual licenses). If it doesn't, you may consider contacting the author and asking them to relicense it. GPL and LGPL code are not acceptable in the main code base, though we are considering an alternative way of distributing L/GPL code through an separate channel, possibly a toolkit. If you include code, make sure you include a copy of that code's license in the license directory if the code's license requires you to distribute the license with it. Non-BSD compatible licenses are acceptable in matplotlib toolkits (e.g., basemap), but make sure you clearly state the licenses you are using.

20.1 Why BSD compatible?

The two dominant license variants in the wild are GPL-style and BSD-style. There are countless other licenses that place specific restrictions on code reuse, but there is an important difference to be considered in the GPL and BSD variants. The best known and perhaps most widely used license is the GPL, which in addition to granting you full rights to the source code including redistribution, carries with it an extra obligation. If you use GPL code in your own code, or link with it, your product must be released under a GPL compatible license. i.e., you are required to give the source code to other people and give them the right to redistribute it as well. Many of the most famous and widely used open source projects are released under the GPL, including linux, gcc, emacs and sage.

The second major class are the BSD-style licenses (which includes MIT and the python PSF license). These basically allow you to do whatever you want with the code: ignore it, include it in your own open source project, include it in your proprietary product, sell it, whatever. python itself is released under a BSD compatible license, in the sense that, quoting from the PSF license page:

```
There is no GPL-like "copyleft" restriction. Distributing
binary-only versions of Python, modified or not, is allowed. There
is no requirement to release any of your source code. You can also
write extension modules for Python and provide them only in binary
form.
```

Famous projects released under a BSD-style license in the permissive sense of the last paragraph are the BSD operating system, python and TeX.

There are several reasons why early matplotlib developers selected a BSD compatible license. matplotlib is a python extension, and we choose a license that was based on the python license (BSD compatible).

Also, we wanted to attract as many users and developers as possible, and many software companies will not use GPL code in software they plan to distribute, even those that are highly committed to open source development, such as enthought, out of legitimate concern that use of the GPL will "infect" their code base by its viral nature. In effect, they want to retain the right to release some proprietary code. Companies and institutions who use matplotlib often make significant contributions, because they have the resources to get a job done, even a boring one. Two of the matplotlib backends (FLTK and WX) were contributed by private companies. The final reason behind the licensing choice is compatibility with the other python extensions for scientific computing: ipython, numpy, scipy, the enthought tool suite and python itself are all distributed under BSD compatible licenses.

CHAPTER

TWENTYONE

WORKING WITH *MATPLOTLIB* SOURCE CODE

Contents:

21.1 Introduction

These pages describe a git and github workflow for the matplotlib project.

There are several different workflows here, for different ways of working with *matplotlib*.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git see *git resources*.

21.2 Install git

21.2.1 Overview

Debian / Ubuntu	`sudo apt-get install git-core`
Fedora	`sudo yum install git-core`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

21.2.2 In detail

See the git page for the most recent information.

Have a look at the github install help pages available from github help

There are good instructions here: http://git-scm.com/book/en/v2/Getting-Started-Installing-Git

21.3 Following the latest source

These are the instructions if you just want to follow the latest *matplotlib* source, but you don't need to do any development for now.

The steps are:

- *Install git*
- get local copy of the git repository from github
- update local copy from time to time

21.3.1 Get the local copy of the code

From the command line:

```
git clone git://github.com/matplotlib/matplotlib.git
```

You now have a copy of the code tree in the new `matplotlib` directory.

21.3.2 Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd matplotlib
git pull
```

The tree in `matplotlib` will now have the latest changes from the initial repository.

21.4 Git for development

Contents:

21.4.1 Making your own copy (fork) of matplotlib

You need to do this only once. The instructions here are very similar to the instructions at http://help.github.com/forking/ — please see that page for more detail. We're repeating some of it here just to give the specifics for the matplotlib project, and to suggest some default names.

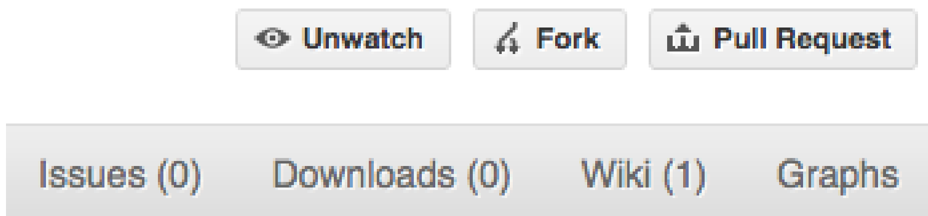
Set up and configure a github account

If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the `Generating SSH keys` help on github help.

Create your own forked copy of matplotlib

1. Log into your github account.
2. Go to the matplotlib github home at matplotlib github.
3. Click on the *fork* button:

   Now, after a short pause you should find yourself at the home page for your own forked copy of matplotlib.

21.4.2 Set up your fork

First you follow the instructions for *Making your own copy (fork) of matplotlib*.

Overview

```
git clone git@github.com:your-user-name/matplotlib.git
cd matplotlib
git remote add upstream git://github.com/matplotlib/matplotlib.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/matplotlib.git`
2. Investigate. Change directory to your new repo: `cd matplotlib`. Then `git branch -a` to show you all branches. You'll get something like:

   ```
   * master
   remotes/origin/master
   ```

   This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

   Now you want to connect to the upstream matplotlib github repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd matplotlib
git remote add upstream git://github.com/matplotlib/matplotlib.git
```

`upstream` here is just the arbitrary name we're using to refer to the main matplotlib repository at matplotlib github.

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Note this command needs to be run on every clone of the repository that you make. It is not tracked in your personal repository on github.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream     git://github.com/matplotlib/matplotlib.git (fetch)
upstream     git://github.com/matplotlib/matplotlib.git (push)
origin       git@github.com:your-user-name/matplotlib.git (fetch)
origin       git@github.com:your-user-name/matplotlib.git (push)
```

21.4.3 Configure git

Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory. Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true

[apply]
        whitespace = fix
```

```
[core]
        autocrlf = input
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

In detail

user.name and user.email

It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
```

```
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor

You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging

To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

21.4.4 Development workflow

You've discovered a bug or something else you want to change in matplotlib .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way to contribute to matplotlib is through github. If for some reason you don't want to use github, see *Making patches* for instructions on how to email patches to the mailing list.

You already have your own forked copy of the matplotlib repository, by following *Making your own copy (fork) of matplotlib*, *Set up your fork*, and you have configured git by following *Configure git*.

Workflow summary

- Keep your `master` branch clean of edits that have not been merged to the main matplotlib development repo. Your `master` then will follow the main matplotlib repository.
- Start a new *feature branch* for each set of edits that you do.
- Do not merge the `master` branch or maintenance tracking branches into your feature branch. If you need to include commits from upstream branches (either to pick up a bug fix or to resolve a conflict) please *rebase* your branch on the upstream branch.
- Ask for review!

This way of working really helps to keep work well organized, and in keeping history as clear as possible.

See — for example — linux git workflow.

Making a new feature branch

```
git checkout -b my-new-feature master
```

This will create and immediately check out a feature branch based on `master`. To create a feature branch based on a maintenance branch, use:

```
git fetch origin
git checkout -b my-new-feature origin/v1.0.x
```

Generally, you will want to keep this also on your public github fork of matplotlib. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

You will need to use this exact command, rather than simply `git push` every time you want to push changes on your feature branch to your github repo. However, in git >1.7 you can set up a link by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

and then next time you need to push changes to your branch a simple `git push` will suffice. Note that `git push` pushes out all branches that are linked to a remote branch.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag? — and the helpful use-case description in the tangled working copy problem. The git commit manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see `git push`).

Asking for code review — open a Pull Request (PR)

It's a good idea to consult the *Pull request checklist* to make sure your pull request is ready for merging.

1. Go to your repo URL — e.g., `http://github.com/your-user-name/matplotlib`.
2. Select your feature branch from the drop down menu:
3. Click on the green button:
4. Make sure that you are requesting a pull against the correct branch
5. Enter a PR heading and description (if there is only one commit in the PR github will automatically fill these fields for you). If this PR is addressing a specific issue, please reference it by number (ex #1325) which github will automatically make into links.

6. Click 'Create Pull Request' button!
7. Discussion of the change will take place in the pull request thread.

Staying up to date with changes in the central repository

This updates your working copy from the upstream matplotlib github repo.

Overview

```
# go to your master branch
git checkout master
# pull changes from github
git fetch upstream
# merge from upstream
git merge --ff-only upstream/master
```

In detail

We suggest that you do this only for your `master` branch, and leave your 'feature' branches unmerged, to keep their history as clean as possible. This makes code review easier:

```
git checkout master
```

Make sure you have done *Linking your repository to the upstream repo*.

Merge the upstream code into your current development by first pulling the upstream repo to a copy on your local machine:

```
git fetch upstream
```

then merging into your current branch:

```
git merge --ff-only upstream/master
```

The `--ff-only` option guarantees that if you have mistakenly committed code on your `master` branch, the merge fails at this point. If you were to merge `upstream/master` to your `master`, you would start to diverge from the upstream. If this command fails, see the section on *accidents*.

The letters 'ff' in `--ff-only` mean 'fast forward', which is a special case of merge where git can simply update your branch to point to the other branch and not do any actual merging of files. For `master` and other integration branches this is exactly what you want.

Other integration branches

Some people like to keep separate local branches corresponding to the maintenance branches on github. At the time of this writing, `v1.0.x` is the active maintenance branch. If you have such a local branch, treat is just as master: don't commit on it, and before starting new branches off of it, update it from upstream:

```
git checkout v1.0.x
git fetch upstream
git merge --ff-only upstream/v1.0.x
```

But you don't necessarily have to have such a branch. Instead, if you are preparing a bugfix that applies to the maintenance branch, fetch from upstream and base your bugfix on the remote branch:

```
git fetch upstream
git checkout -b my-bug-fix upstream/v1.0.x
```

Recovering from accidental commits on master

If you have accidentally committed changes on master and git merge --ff-only fails, don't panic! First find out how much you have diverged:

```
git diff upstream/master...master
```

If you find that you want simply to get rid of the changes, reset your master branch to the upstream version:

```
git reset --hard upstream/master
```

As you might surmise from the words 'reset' and 'hard', this command actually causes your changes to the current branch to be lost, so think twice.

If, on the other hand, you find that you want to preserve the changes, create a feature branch for them:

```
git checkout -b my-important-changes
```

Now my-important-changes points to the branch that has your changes, and you can safely reset master as above — but make sure to reset the correct branch:

```
git checkout master
git reset --hard upstream/master
```

Deleting a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

Note the colon : before my-unwanted-branch. See also: https://help.github.com/articles/pushing-to-a-remote/#deleting-a-remote-branch-or-tag

Exploring your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your github repo.

21.4.5 Two and three dots in difference specs

Thanks to Yarik Halchenko for this explanation.

Imagine a series of commits A, B, C, D... Imagine that there are two branches, *topic* and *master*. You branched *topic* off *master* when *master* was at commit 'E'. The graph of the commits looks like this:

```
     A---B---C topic
     /
D---E---F---G master
```

Then:

```
git diff master..topic
```

will output the difference from G to C (i.e. with effects of F and G), while:

```
git diff master...topic
```

would output just differences in the topic branch (i.e. only A, B, and C).

21.5 git resources

21.5.1 Tutorials and summaries

- github help has an excellent series of how-to guides.
- learn.github has an excellent series of tutorials
- The pro git book is a good in-depth book on git.
- A git cheat sheet is a page giving summaries of common commands.
- The git user manual
- The git tutorial
- The git community book

- git ready — a nice series of tutorials
- git casts — video snippets giving git how-tos.
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- Our own git foundation expands on the git parable.
- Fernando Perez' git page — Fernando's git page — many links and tips
- A good but technical page on git concepts
- git svn crash course: git for those of us used to subversion

21.5.2 Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

21.5.3 Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

21.6 Making a patch

21.6.1 Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/matplotlib/matplotlib.git
# make a branch for your patching
cd matplotlib
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the matplotlib mailing list — where we will thank you warmly.

In detail

1. Tell git who you are so it can label the commits you've made:

   ```
   git config --global user.email you@yourdomain.example.com
   git config --global user.name "Your Name Comes Here"
   ```

2. If you don't already have one, clone a copy of the matplotlib repository:

   ```
   git clone git://github.com/matplotlib/matplotlib.git
   cd matplotlib
   ```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

   ```
   git branch the-fix-im-thinking-of
   git checkout the-fix-im-thinking-of
   ```

4. Do some edits, and commit them as you go:

   ```
   # hack, hack, hack
   # Tell git about any new files you've made
   git add somewhere/tests/test_my_bug.py
   ```

```
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag?.

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
git format-patch -M -C master
```

You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the matplotlib mailing list.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

CHAPTER

TWENTYTWO

TESTING

Matplotlib has a testing infrastructure based on nose, making it easy to write new tests. The tests are in `matplotlib.tests`, and customizations to the nose testing infrastructure are in `matplotlib.testing`. (There is other old testing cruft around, please ignore it while we consolidate our testing to these locations.)

22.1 Requirements

The following software is required to run the tests:

- nose, version 1.0 or later
- mock, when running python versions < 3.3
- Ghostscript (to render PDF files)
- Inkscape (to render SVG files)

Optionally you can install:

- coverage to collect coverage information
- pep8 to test coding standards

22.2 Running the tests

Running the tests is simple. Make sure you have nose installed and run:

```
python tests.py
```

in the root directory of the distribution. The script takes a set of commands, such as:

`--pep8`	pep8 checks
`--no-pep8`	Do not perform pep8 checks
`--no-network`	Disable tests that require network access

Additional arguments are passed on to nosetests. See the nose documentation for supported arguments. Some of the more important ones are given here:

--verbose	Be more verbose
--processes=NUM	Run tests in parallel over NUM processes
--process-timeout=SECONDS	Set timeout for results from test runner process
--nocapture	Do not capture stdout

To run a single test from the command line, you can provide a dot-separated path to the module followed by the function separated by a colon, e.g., (this is assuming the test is installed):

```
python tests.py matplotlib.tests.test_simplification:test_clipping
```

If you want to run the full test suite, but want to save wall time try running the tests in parallel:

```
python tests.py --nocapture --nose-verbose --processes=5 --process-timeout=300
```

An alternative implementation that does not look at command line arguments works from within Python is to run the tests from the matplotlib library function `matplotlib.test()`:

```
import matplotlib
matplotlib.test()
```

Hint: To run the tests you need to install nose and mock if using python 2.7:

```
pip install nose
pip install mock
```

22.3 Writing a simple test

Many elements of Matplotlib can be tested using standard tests. For example, here is a test from `matplotlib.tests.test_basic`:

```
from nose.tools import assert_equal

def test_simple():
    """
    very simple example test
    """
    assert_equal(1+1,2)
```

Nose determines which functions are tests by searching for functions beginning with "test" in their name.

If the test has side effects that need to be cleaned up, such as creating figures using the pyplot interface, use the `@cleanup` decorator:

```
from matplotlib.testing.decorators import cleanup

@cleanup
def test_create_figure():
    """
```

```
    very simple example test that creates a figure using pyplot.
    """
    fig = figure()
    ...
```

22.4 Writing an image comparison test

Writing an image based test is only slightly more difficult than a simple test. The main consideration is that you must specify the "baseline", or expected, images in the `image_comparison()` decorator. For example, this test generates a single image and automatically tests it:

```
import numpy as np
import matplotlib
from matplotlib.testing.decorators import image_comparison
import matplotlib.pyplot as plt

@image_comparison(baseline_images=['spines_axes_positions'],
                  extensions=['png'])
def test_spines_axes_positions():
    # SF bug 2852168
    fig = plt.figure()
    x = np.linspace(0,2*np.pi,100)
    y = 2*np.sin(x)
    ax = fig.add_subplot(1,1,1)
    ax.set_title('centered spines')
    ax.plot(x,y)
    ax.spines['right'].set_position(('axes',0.1))
    ax.yaxis.set_ticks_position('right')
    ax.spines['top'].set_position(('axes',0.25))
    ax.xaxis.set_ticks_position('top')
    ax.spines['left'].set_color('none')
    ax.spines['bottom'].set_color('none')
```

The first time this test is run, there will be no baseline image to compare against, so the test will fail. Copy the output images (in this case `result_images/test_category/spines_axes_positions.png`) to the correct subdirectory of `baseline_images` tree in the source directory (in this case `lib/matplotlib/tests/baseline_images/test_category`). Put this new file under source code revision control (with `git add`). When rerunning the tests, they should now pass.

The `image_comparison()` decorator defaults to generating `png`, `pdf` and `svg` output, but in interest of keeping the size of the library from ballooning we should only include the `svg` or `pdf` outputs if the test is explicitly exercising a feature dependent on that backend.

There are two optional keyword arguments to the `image_comparison` decorator:

- `extensions`: If you only wish to test additional image formats (rather than just `png`), pass any additional file types in the list of the extensions to test. When copying the new baseline files be sure to only copy the output files, not their conversions to `png`. For example only copy the files ending in `pdf`, not in `_pdf.png`.

- tol: This is the image matching tolerance, the default 1e-3. If some variation is expected in the image between runs, this value may be adjusted.

22.5 Freetype version

Due to subtle differences in the font rendering under different version of freetype some care must be taken when generating the baseline images. Currently (early 2015), almost all of the images were generated using freetype 2.5.3-21 on Fedora 21 and only the fonts that ship with matplotlib (regenerated in PR #4031 / commit 005cfde02751d274f2ab8016eddd61c3b3828446) and travis is using freetype 2.4.8 on ubuntu.

22.6 Known failing tests

If you're writing a test, you may mark it as a known failing test with the knownfailureif() decorator. This allows the test to be added to the test suite and run on the buildbots without causing undue alarm. For example, although the following test will fail, it is an expected failure:

```
from nose.tools import assert_equal
from matplotlib.testing.decorators import knownfailureif

@knownfailureif(True)
def test_simple_fail():
    '''very simple example test that should fail'''
    assert_equal(1+1,3)
```

Note that the first argument to the knownfailureif() decorator is a fail condition, which can be a value such as True, False, or 'indeterminate', or may be a dynamically evaluated expression.

22.7 Creating a new module in matplotlib.tests

We try to keep the tests categorized by the primary module they are testing. For example, the tests related to the mathtext.py module are in test_mathtext.py.

Let's say you've added a new module named whizbang.py and you want to add tests for it in matplotlib.tests.test_whizbang. To add this module to the list of default tests, append its name to default_test_modules in lib/matplotlib/__init__.py.

22.8 Using Travis CI

Travis CI is a hosted CI system "in the cloud".

Travis is configured to receive notifications of new commits to GitHub repos (via GitHub "service hooks") and to run builds or tests when it sees these new commits. It looks for a YAML file called .travis.yml in the root of the repository to see how to test the project.

Travis CI is already enabled for the main matplotlib GitHub repository – for example, see its Travis page.

If you want to enable Travis CI for your personal matplotlib GitHub repo, simply enable the repo to use Travis CI in either the Travis CI UI or the GitHub UI (Admin | Service Hooks). For details, see the Travis CI Getting Started page. This generally isn't necessary, since any pull request submitted against the main matplotlib repository will be tested.

Once this is configured, you can see the Travis CI results at http://travis-ci.org/your_GitHub_user_name/matplotlib – here's an example.

22.9 Using tox

Tox is a tool for running tests against multiple Python environments, including multiple versions of Python (e.g., 2.7, 3.4, 3.5) and even different Python implementations altogether (e.g., CPython, PyPy, Jython, etc.)

Testing all versions of Python (2.6, 2.7, 3.*) requires having multiple versions of Python installed on your system and on the PATH. Depending on your operating system, you may want to use your package manager (such as apt-get, yum or MacPorts) to do this.

tox makes it easy to determine if your working copy introduced any regressions before submitting a pull request. Here's how to use it:

```
$ pip install tox
$ tox
```

You can also run tox on a subset of environments:

```
$ tox -e py26,py27
```

Tox processes everything serially so it can take a long time to test several environments. To speed it up, you might try using a new, parallelized version of tox called `detox`. Give this a try:

```
$ pip install -U -i http://pypi.testrun.org detox
$ detox
```

Tox is configured using a file called `tox.ini`. You may need to edit this file if you want to add new environments to test (e.g., `py33`) or if you want to tweak the dependencies or the way the tests are run. For more info on the `tox.ini` file, see the Tox Configuration Specification.

CHAPTER

TWENTYTHREE

DOCUMENTING MATPLOTLIB

23.1 Getting started

The documentation for matplotlib is generated from ReStructured Text using the Sphinx documentation generation tool. Sphinx-1.0 or later and numpydoc 0.4 or later is required.

The documentation sources are found in the `doc/` directory in the trunk. To build the users guide in html format, cd into `doc/` and do:

```
python make.py html
```

or:

```
./make.py html
```

you can also pass a `latex` flag to make.py to build a pdf, or pass no arguments to build everything.

The output produced by Sphinx can be configured by editing the `conf.py` file located in the `doc/`.

23.2 Organization of matplotlib's documentation

The actual ReStructured Text files are kept in `doc/users`, `doc/devel`, `doc/api` and `doc/faq`. The main entry point is `doc/index.rst`, which pulls in the `index.rst` file for the users guide, developers guide, api reference, and faqs. The documentation suite is built as a single document in order to make the most effective use of cross referencing, we want to make navigating the Matplotlib documentation as easy as possible.

Additional files can be added to the various guides by including their base file name (the .rst extension is not necessary) in the table of contents. It is also possible to include other documents through the use of an include statement, such as:

```
.. include:: ../../TODO
```

23.2.1 docstrings

In addition to the "narrative" documentation described above, matplotlib also defines its API reference documentation in docstrings. For the most part, these are standard Python docstrings, but matplotlib also includes some features to better support documenting getters and setters.

Matplotlib uses artist introspection of docstrings to support properties. All properties that you want to support through `setp` and `getp` should have a `set_property` and `get_property` method in the *Artist* class. Yes, this is not ideal given python properties or enthought traits, but it is a historical legacy for now. The setter methods use the docstring with the ACCEPTS token to indicate the type of argument the method accepts. e.g., in *matplotlib.lines.Line2D*:

```
# in lines.py
def set_linestyle(self, linestyle):
    """
    Set the linestyle of the line

    ACCEPTS: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' | ' ' | '' ]
    """
```

Since matplotlib uses a lot of pass-through `kwargs`, e.g., in every function that creates a line (*plot()*, *semilogx()*, *semilogy()*, etc...), it can be difficult for the new user to know which `kwargs` are supported. Matplotlib uses a docstring interpolation scheme to support documentation of every function that takes a `**kwargs`. The requirements are:

1. single point of configuration so changes to the properties don't require multiple docstring edits.
2. as automated as possible so that as properties change, the docs are updated automagically.

The function *matplotlib.artist.kwdoc()* and the decorator `matplotlib.docstring.dedent_interpd()` facilitate this. They combine python string interpolation in the docstring with the matplotlib artist introspection facility that underlies `setp` and `getp`. The `kwdoc` function gives the list of properties as a docstring. In order to use this in another docstring, first update the `matplotlib.docstring.interpd` object, as seen in this example from *matplotlib.lines*:

```
# in lines.py
docstring.interpd.update(Line2D=artist.kwdoc(Line2D))
```

Then in any function accepting *Line2D* pass-through `kwargs`, e.g., *matplotlib.axes.Axes.plot()*:

```
# in axes.py
@docstring.dedent_interpd
def plot(self, *args, **kwargs):
    """
    Some stuff omitted

    The kwargs are Line2D properties:
    %(Line2D)s

    kwargs scalex and scaley, if defined, are passed on
    to autoscale_view to determine whether the x and y axes are
    autoscaled; default True.  See Axes.autoscale_view for more
    information
```

```
    """
    pass
```

Note there is a problem for *Artist* `__init__` methods, e.g., `matplotlib.patches.Patch.__init__()`, which supports `Patch kwargs`, since the artist inspector cannot work until the class is fully defined and we can't modify the `Patch.__init__.__doc__` docstring outside the class definition. There are some some manual hacks in this case, violating the "single entry point" requirement above – see the `docstring.interpd.update` calls in *matplotlib.patches*.

23.3 Formatting

The Sphinx website contains plenty of documentation concerning ReST markup and working with Sphinx in general. Here are a few additional things to keep in mind:

- Please familiarize yourself with the Sphinx directives for inline markup. Matplotlib's documentation makes heavy use of cross-referencing and other semantic markup. For example, when referring to external files, use the `:file:` directive.
- Function arguments and keywords should be referred to using the *emphasis* role. This will keep matplotlib's documentation consistent with Python's documentation:

  ```
  Here is a description of *argument*
  ```

  Please do not use the `default role`:

  ```
  Please do not describe `argument` like this.
  ```

  nor the `literal` role:

  ```
  Please do not describe ``argument`` like this.
  ```

- Sphinx does not support tables with column- or row-spanning cells for latex output. Such tables can not be used when documenting matplotlib.
- Mathematical expressions can be rendered as png images in html, and in the usual way by latex. For example:

  `:math:‘\sin(x_n^2)‘` yields: $\sin(x_n^2)$, and:

  ```
  .. math::

    \int_{-\infty}^{\infty}\frac{e^{i\phi}}{1+x^2\frac{e^{i\phi}}{1+x^2}}
  ```

  yields:

  $$\int_{-\infty}^{\infty}\frac{e^{i\phi}}{1+x^2\frac{e^{i\phi}}{1+x^2}} \tag{23.1}$$

- Interactive IPython sessions can be illustrated in the documentation using the following directive:

```
.. sourcecode:: ipython

  In [69]: lines = plot([1,2,3])
```

which would yield:

```
In [69]: lines = plot([1,2,3])
```

- Footnotes [1] can be added using `[#]_`, followed later by:

```
.. rubric:: Footnotes

.. [#]
```

- Use the *note* and *warning* directives, sparingly, to draw attention to important comments:

```
.. note::
   Here is a note
```

yields:

Note: here is a note

also:

Warning: here is a warning

- Use the *deprecated* directive when appropriate:

```
.. deprecated:: 0.98
   This feature is obsolete, use something else.
```

yields:

Deprecated since version 0.98: This feature is obsolete, use something else.

- Use the *versionadded* and *versionchanged* directives, which have similar syntax to the *deprecated* role:

```
.. versionadded:: 0.98
   The transforms have been completely revamped.
```

New in version 0.98: The transforms have been completely revamped.

- Use the *seealso* directive, for example:

[1] For example.

```
.. seealso::

   Using ReST :ref:`emacs-helpers`:
      One example

   A bit about :ref:`referring-to-mpl-docs`:
      One more
```

yields:

See also:

Using ResT ***Emacs helpers*****:** One example

A bit about ***Referring to mpl documents*****:** One more

- Please keep the *Glossary* in mind when writing documentation. You can create a references to a term in the glossary with the `:term:` role.
- The autodoc extension will handle index entries for the API, but additional entries in the index need to be explicitly added.
- Please limit the text width of docstrings to 70 characters.
- Keyword arguments should be described using a definition list.

Note: matplotlib makes extensive use of keyword arguments as pass-through arguments, there are a many cases where a table is used in place of a definition list for autogenerated sections of docstrings.

23.4 Figures

23.4.1 Dynamically generated figures

Figures can be automatically generated from scripts and included in the docs. It is not necessary to explicitly save the figure in the script, this will be done automatically at build time to ensure that the code that is included runs and produces the advertised figure.

The path should be relative to the `doc` directory. Any plots specific to the documentation should be added to the `doc/pyplots` directory and committed to git. Plots from the `examples` directory may be referenced through the symlink `mpl_examples` in the `doc` directory. e.g.:

```
.. plot:: mpl_examples/pylab_examples/simple_plot.py
```

The `:scale:` directive rescales the image to some percentage of the original size, though we don't recommend using this in most cases since it is probably better to choose the correct figure size and dpi in mpl and let it handle the scaling.

Plot directive documentation

A directive for including a matplotlib plot in a Sphinx document.

By default, in HTML output, `plot` will include a .png file with a link to a high-res .png and .pdf. In LaTeX output, it will include a .pdf.

The source code for the plot may be included in one of three ways:

1. **A path to a source file** as the argument to the directive:

   ```
   .. plot:: path/to/plot.py
   ```

   When a path to a source file is given, the content of the directive may optionally contain a caption for the plot:

   ```
   .. plot:: path/to/plot.py

      This is the caption for the plot
   ```

   Additionally, one may specify the name of a function to call (with no arguments) immediately after importing the module:

   ```
   .. plot:: path/to/plot.py plot_function1
   ```

2. Included as **inline content** to the directive:

   ```
   .. plot::

      import matplotlib.pyplot as plt
      import matplotlib.image as mpimg
      import numpy as np
      img = mpimg.imread('_static/stinkbug.png')
      imgplot = plt.imshow(img)
   ```

3. Using **doctest** syntax:

   ```
   .. plot::
      A plotting example:
      >>> import matplotlib.pyplot as plt
      >>> plt.plot([1,2,3], [4,5,6])
   ```

Options

The `plot` directive supports the following options:

format [{'python', 'doctest'}] Specify the format of the input

include-source [bool] Whether to display the source code. The default can be changed using the `plot_include_source` variable in conf.py

encoding [str] If this source file is in a non-UTF8 or non-ASCII encoding, the encoding must be specified using the `:encoding:` option. The encoding will not be inferred using the `-*- coding -*-` metacomment.

context [bool or str] If provided, the code will be run in the context of all previous plot directives for which the `:context:` option was specified. This only applies to inline code plot directives, not those run from files. If the `:context: reset` option is specified, the context is reset for this and future plots, and previous figures are closed prior to running the code. `:context:close-figs` keeps the context but closes previous figures before running the code.

nofigs [bool] If specified, the code block will be run, but no figures will be inserted. This is usually useful with the `:context:` option.

Additionally, this directive supports all of the options of the `image` directive, except for `target` (since plot will add its own target). These include `alt`, `height`, `width`, `scale`, `align` and `class`.

Configuration options

The plot directive has the following configuration options:

plot_include_source Default value for the include-source option

plot_html_show_source_link Whether to show a link to the source in HTML.

plot_pre_code Code that should be executed before each plot.

plot_basedir Base directory, to which `plot::` file names are relative to. (If None or empty, file names are relative to the directory where the file containing the directive is.)

plot_formats File formats to generate. List of tuples or strings:

```
[(suffix, dpi), suffix, ...]
```

that determine the file format and the DPI. For entries whose DPI was omitted, sensible defaults are chosen. When passing from the command line through sphinx_build the list should be passed as suffix:dpi,suffix:dpi, ...

plot_html_show_formats Whether to show links to the files in HTML.

plot_rcparams A dictionary containing any non-standard rcParams that should be applied before each plot.

plot_apply_rcparams By default, rcParams are applied when `context` option is not used in a plot directive. This configuration option overrides this behavior and applies rcParams before each plot.

plot_working_directory By default, the working directory will be changed to the directory of the example, so the code can get at its data files, if any. Also its path will be added to `sys.path` so it can import any helper modules sitting beside it. This configuration option can be used to specify a central directory (also added to `sys.path`) where data files and helper modules for all code are located.

plot_template Provide a customized template for preparing restructured text.

23.4.2 Static figures

Any figures that rely on optional system configurations need to be handled a little differently. These figures are not to be generated during the documentation build, in order to keep the prerequisites to the documentation effort as low as possible. Please run the `doc/pyplots/make.py` script when adding such figures, and commit the script **and** the images to git. Please also add a line to the README in doc/pyplots for any additional requirements necessary to generate a new figure. Once these steps have been taken, these figures can be included in the usual way:

```
.. plot:: pyplots/tex_unicode_demo.py
   :include-source:
```

23.4.3 Examples

The source of the files in the `examples` directory are automatically included in the HTML docs. An image is generated and included for all examples in the `api` and `pylab_examples` directories. To exclude the example from having an image rendered, insert the following special comment anywhere in the script:

```
# -*- noplot -*-
```

23.4.4 Animations

We have a matplotlib google/gmail account with username `mplgithub` which we used to setup the github account but can be used for other purposes, like hosting google docs or youtube videos. You can embed a matplotlib animation in the docs by first saving the animation as a movie using *`matplotlib.animation.Animation.save()`*, and then uploading to matplotlib's youtube channel and inserting the embedding string youtube provides like:

```
.. raw:: html

   <iframe width="420" height="315"
     src="http://www.youtube.com/embed/32cjc6V0OZY"
     frameborder="0" allowfullscreen>
   </iframe>
```

An example save command to generate a movie looks like this

```
ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)),
    interval=25, blit=True, init_func=init)

ani.save('double_pendulum.mp4', fps=15)
```

Contact Michael Droettboom for the login password to upload youtube videos of google docs to the mplgithub account.

23.5 Referring to mpl documents

In the documentation, you may want to include to a document in the matplotlib src, e.g., a license file or an image file from `mpl-data`, refer to it via a relative path from the document where the rst file resides, e.g., in `users/navigation_toolbar.rst`, we refer to the image icons with:

```
.. image:: ../../lib/matplotlib/mpl-data/images/subplots.png
```

In the `users` subdirectory, if I want to refer to a file in the mpl-data directory, I use the symlink directory. For example, from `customizing.rst`:

```
.. literalinclude:: ../../lib/matplotlib/mpl-data/matplotlibrc
```

One exception to this is when referring to the examples dir. Relative paths are extremely confusing in the sphinx plot extensions, so without getting into the dirty details, it is easier to simply include a symlink to the files at the top doc level directory. This way, API documents like *matplotlib.pyplot.plot()* can refer to the examples in a known location.

In the top level doc directory we have symlinks pointing to the mpl `examples`:

```
home:~/mpl/doc> ls -l mpl_*
mpl_examples -> ../examples
```

So we can include plots from the examples dir using the symlink:

```
.. plot:: mpl_examples/pylab_examples/simple_plot.py
```

We used to use a symlink for `mpl-data` too, but the distro becomes very large on platforms that do not support links (e.g., the font files are duplicated and large)

23.6 Internal section references

To maximize internal consistency in section labeling and references, use hyphen separated, descriptive labels for section references, e.g.,:

```
.. _howto-webapp:
```

and refer to it using the standard reference syntax:

```
See :ref:`howto-webapp`
```

Keep in mind that we may want to reorganize the contents later, so let's avoid top level names in references like `user` or `devel` or `faq` unless necessary, because for example the FAQ "what is a backend?" could later become part of the users guide, so the label:

```
.. _what-is-a-backend
```

is better than:

```
.. _faq-backend
```

In addition, since underscores are widely used by Sphinx itself, let's prefer hyphens to separate words.

23.7 Section names, etc

For everything but top level chapters, please use `Upper lower` for section titles, e.g., `Possible hangups` rather than `Possible Hangups`

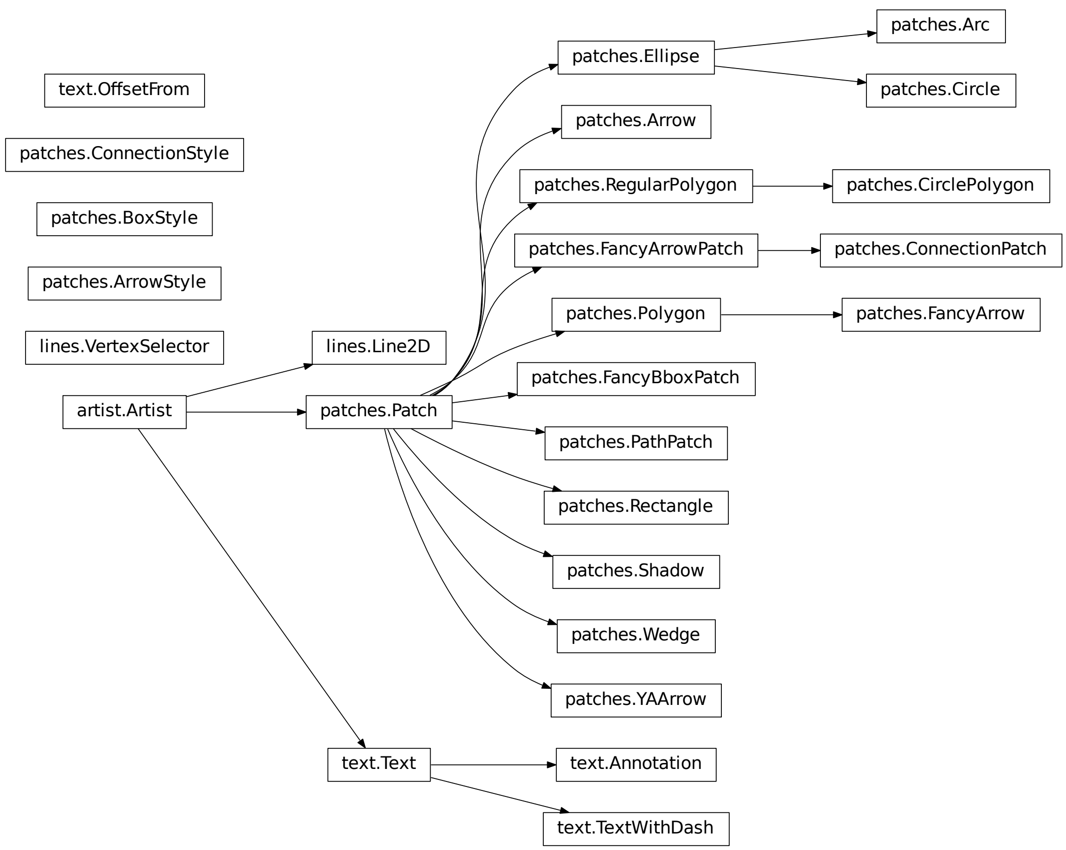
23.8 Inheritance diagrams

Class inheritance diagrams can be generated with the `inheritance-diagram` directive. To use it, you provide the directive with a number of class or module names (separated by whitespace). If a module name is provided, all classes in that module will be used. All of the ancestors of these classes will be included in the inheritance diagram.

A single option is available: *parts* controls how many of parts in the path to the class are shown. For example, if *parts* == 1, the class `matplotlib.patches.Patch` is shown as `Patch`. If *parts* == 2, it is shown as `patches.Patch`. If *parts* == 0, the full path is shown.

Example:

```
.. inheritance-diagram:: matplotlib.patches matplotlib.lines matplotlib.text
   :parts: 2
```



23.9 Emacs helpers

There is an emacs mode rst.el which automates many important ReST tasks like building and updating table-of-contents, and promoting or demoting section headings. Here is the basic `.emacs` configuration:

```
(require 'rst)
(setq auto-mode-alist
      (append '(("\\.txt$" . rst-mode)
                ("\\.rst$" . rst-mode)
                ("\\.rest$" . rst-mode)) auto-mode-alist))
```

Some helpful functions:

```
C-c TAB - rst-toc-insert

  Insert table of contents at point

C-c C-u - rst-toc-update
```

```
    Update the table of contents at point

C-c C-l rst-shift-region-left

    Shift region to the left

C-c C-r rst-shift-region-right

    Shift region to the right
```

CHAPTER

TWENTYFOUR

DOING A MATPLOTLIB RELEASE

A guide for developers who are doing a matplotlib release.

- Edit `__init__.py` and bump the version number

24.1 Testing

- Run all of the regression tests by running `python tests.py` at the root of the source tree.
- Run `unit/memleak_hawaii3.py` and make sure there are no memory leaks
- try some GUI examples, e.g., `simple_plot.py` with GTKAgg, TkAgg, etc...
- remove font cache and tex cache from `.matplotlib` and test with and without cache on some example script
- Optionally, make sure `examples/tests/backend_driver.py` runs without errors and check the output of the PNG, PDF, PS and SVG backends

24.2 Branching

Once all the tests are passing and you are ready to do a release, you need to create a release branch. These only need to be created when the second part of the version number changes:

```
git checkout -b v1.1.x
git push git@github.com:matplotlib/matplotlib.git v1.1.x
```

On the branch, do any additional testing you want to do, and then build binaries and source distributions for testing as release candidates.

For each release candidate as well as for the final release version, please `git tag` the commit you will use for packaging like so:

```
git tag -a v1.1.0rc1
```

The `-a` flag will allow you to write a message about the tag, and affiliate your name with it. A reasonable tag message would be something like `v1.1.0 Release Candidate 1 (September 24, 2011)`. To tag a release after the fact, just track down the commit hash, and:

```
git tag -a v1.0.1rc1 a9f3f3a50745
```

Tags allow developers to quickly checkout different releases by name, and also provides source download via zip and tarball on github.

Then push the tags to the main repository:

```
git push upstream v1.0.1rc1
```

24.3 Packaging

- Make sure the `MANIFEST.in` is up to date and remove `MANIFEST` so it will be rebuilt by MANIFEST.in
- run `git clean` in the mpl git directory before building the sdist
- unpack the sdist and make sure you can build from that directory
- Use `setup.cfg` to set the default backends. For windows and OSX, the default backend should be TkAgg. You should also turn on or off any platform specific build options you need. Importantly, you also need to make sure that you delete the `build` dir after any changes to `setup.cfg` before rebuilding since cruft in the `build` dir can get carried along.
- On windows, unix2dos the rc file.
- We have a Makefile for the OS X builds in the mpl source dir `release/osx`, so use this to prepare the OS X releases.
- We have a Makefile for the win32 mingw builds in the mpl source dir `release/win32` which you can use this to prepare the windows releases.

24.4 Update PyPI

This step tells PyPI about the release and uploads a source tarball. This should only be done with final (non-release-candidate) releases, since doing so will hide any available stable releases.

You may need to set up your `pypirc` file as described in the distutils register command documentation.

Then updating the record on PyPI is as simple as:

```
python setup.py register
```

This will hide any previous releases automatically.

Then, to upload the source tarball:

```
rm -rf dist
python setup.py sdist upload
```

24.5 Documentation updates

The built documentation exists in the matplotlib.github.com repository. Pushing changes to master automatically updates the website.

The documentation is organized by version. At the root of the tree is always the documentation for the latest stable release. Under that, there are directories containing the documentation for older versions as well as the bleeding edge release version called `dev` (usually based on what's on master in the github repository, but it may also temporarily be a staging area for proposed changes). There is also a symlink directory with the name of the most recently released version that points to the root. With each new release, these directories may need to be reorganized accordingly. Any time these version directories are added or removed, the `versions.html` file (which contains a list of the available documentation versions for the user) must also be updated.

To make sure everyone's hard work gets credited, regenerate the github stats. `cd` into the tools directory and run:

```
python github_stats.py $TAG > ../doc/users/github_stats.rst
```

where `$TAG` is the tag of the last major release. This will generate stats for all work done since that release.

In the matplotlib source repository, build the documentation:

```
cd doc
python make.py html
python make.py latex
```

Then copy the build products into your local checkout of the `matplotlib.github.com` repository (assuming here to be checked out in `com`:

```
cp -r build/html/* ~/matplotlib.github.com
cp build/latex/Matplotlib.pdf ~/matplotlib.github.com
```

Then, from the `matplotlib.github.com` directory, commit and push the changes upstream:

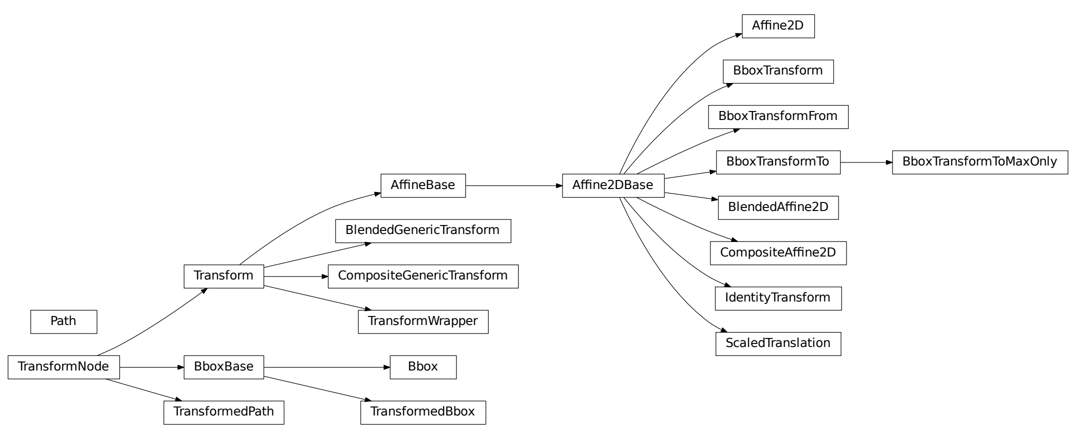
```
git commit -m "Updating for v1.0.1"
git push upstream master
```

24.6 Announcing

Announce the release on matplotlib-announce, matplotlib-users, and matplotlib-devel. Final (non-release-candidate) versions should also be announced on python-announce. Include a summary of highlights from the CHANGELOG and/or post the whole CHANGELOG since the last release.

CHAPTER TWENTYFIVE

WORKING WITH TRANSFORMATIONS

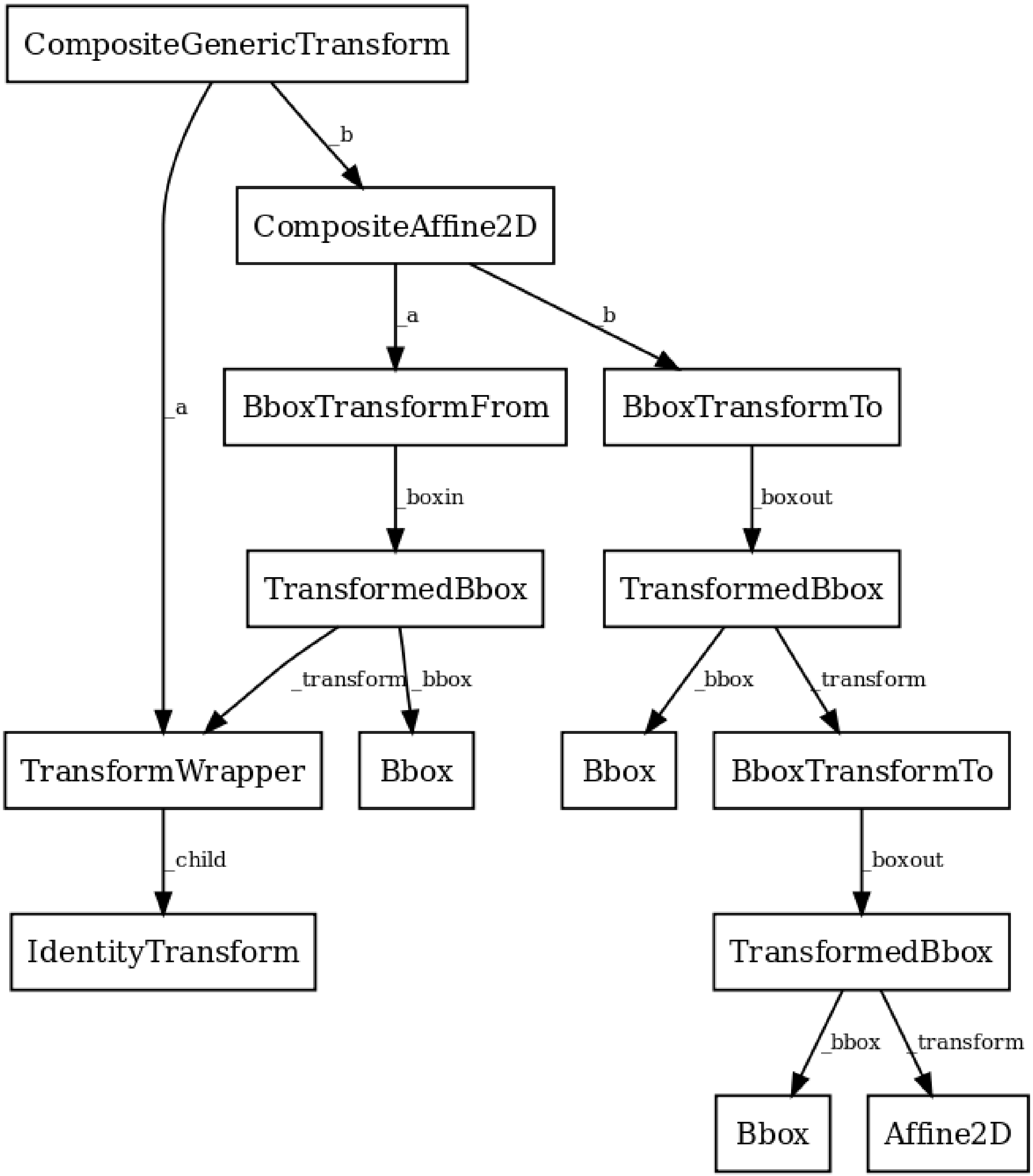


25.1 `matplotlib.transforms`

matplotlib includes a framework for arbitrary geometric transformations that is used determine the final position of all elements drawn on the canvas.

Transforms are composed into trees of *TransformNode* objects whose actual value depends on their children. When the contents of children change, their parents are automatically invalidated. The next time an invalidated transform is accessed, it is recomputed to reflect those changes. This invalidation/caching approach prevents unnecessary recomputations of transforms, and contributes to better interactive performance.

For example, here is a graph of the transform tree used to plot data to the graph:



The framework can be used for both affine and non-affine transformations. However, for speed, we want use the backend renderers to perform affine transformations whenever possible. Therefore, it is possible to perform just the affine or non-affine part of a transformation on a set of data. The affine is always assumed to occur after the non-affine. For any transform:

```
full transform == non-affine part + affine part
```

The backends are not expected to handle non-affine transformations themselves.

class `matplotlib.transforms.TransformNode`(*shorthand_name=None*)
Bases: `object`

TransformNode is the base class for anything that participates in the transform tree and needs to invalidate its parents or be invalidated. This includes classes that are not really transforms, such as bounding boxes, since some transforms depend on bounding boxes to compute their values.

Creates a new *TransformNode*.

shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`frozen()`
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

`invalidate()`
Invalidate this *TransformNode* and triggers an invalidation of its ancestors. Should be called any time the transform changes.

`pass_through` **= False**
If pass_through is True, all ancestors will always be invalidated, even if 'self' is already invalid.

`set_children`(**children*)
Set the children of the transform, to let the invalidation system know which transforms can invalidate this transform. Should be called from the constructor of any transforms that depend on other transforms.

class `matplotlib.transforms.BboxBase`(*shorthand_name=None*)
Bases: *matplotlib.transforms.TransformNode*

This is the base class of all bounding boxes, and provides read-only access to its data. A mutable bounding box is provided by the *Bbox* class.

The canonical representation is as two points, with no restrictions on their ordering. Convenience properties are provided to get the left, bottom, right and top edges and width and height, but these are not stored explicitly.

Creates a new *TransformNode*.

shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

`anchored`(*c*, *container=None*)
Return a copy of the *Bbox*, shifted to position *c* within a container.

c: may be either:

- a sequence (*cx*, *cy*) where *cx* and *cy* range from 0 to 1, where 0 is left or bottom and 1 is right or top
- a string: - 'C' for centered - 'S' for bottom-center - 'SE' for bottom-left - 'E' for left - etc.

Optional argument *container* is the box within which the *Bbox* is positioned; it defaults to the initial *Bbox*.

bounds
(property) Returns (*x0*, *y0*, *width*, *height*).

contains(*x*, *y*)
Returns *True* if (*x*, *y*) is a coordinate inside the bounding box or on its edge.

containsx(*x*)
Returns True if *x* is between or equal to *x0* and *x1*.

containsy(*y*)
Returns True if *y* is between or equal to *y0* and *y1*.

corners()
Return an array of points which are the four corners of this rectangle. For example, if this *Bbox* is defined by the points (*a*, *b*) and (*c*, *d*), *corners()* returns (*a*, *b*), (*a*, *d*), (*c*, *b*) and (*c*, *d*).

count_contains(*vertices*)
Count the number of vertices contained in the *Bbox*.

vertices is a Nx2 Numpy array.

count_overlaps(*bboxes*)
Count the number of bounding boxes that overlap this one.

bboxes is a sequence of *BboxBase* objects

expanded(*sw*, *sh*)
Return a new *Bbox* which is this *Bbox* expanded around its center by the given factors *sw* and *sh*.

extents
(property) Returns (*x0*, *y0*, *x1*, *y1*).

frozen()
TransformNode is the base class for anything that participates in the transform tree and needs to invalidate its parents or be invalidated. This includes classes that are not really transforms, such as bounding boxes, since some transforms depend on bounding boxes to compute their values.

fully_contains(*x*, *y*)
Returns True if (*x*, *y*) is a coordinate inside the bounding box, but not on its edge.

fully_containsx(*x*)
Returns True if *x* is between but not equal to *x0* and *x1*.

fully_containsy(*y*)
Returns True if *y* is between but not equal to *y0* and *y1*.

fully_overlaps(*other*)
Returns True if this bounding box overlaps with the given bounding box *other*, but not on its edge alone.

height
(property) The height of the bounding box. It may be negative if *y1* < *y0*.

static **intersection**(*bbox1*, *bbox2*)
Return the intersection of the two bboxes or None if they do not intersect.

Implements the algorithm described at:

http://www.tekpool.com/node/2687

intervalx
(property) *intervalx* is the pair of *x* coordinates that define the bounding box. It is not guaranteed to be sorted from left to right.

intervaly
(property) *intervaly* is the pair of *y* coordinates that define the bounding box. It is not guaranteed to be sorted from bottom to top.

inverse_transformed(*transform*)
Return a new *Bbox* object, statically transformed by the inverse of the given transform.

is_unit()
Returns True if the *Bbox* is the unit bounding box from (0, 0) to (1, 1).

max
(property) *max* is the top-right corner of the bounding box.

min
(property) *min* is the bottom-left corner of the bounding box.

overlaps(*other*)
Returns True if this bounding box overlaps with the given bounding box *other*.

p0
(property) *p0* is the first pair of (*x*, *y*) coordinates that define the bounding box. It is not guaranteed to be the bottom-left corner. For that, use *min*.

p1
(property) *p1* is the second pair of (*x*, *y*) coordinates that define the bounding box. It is not guaranteed to be the top-right corner. For that, use *max*.

padded(*p*)
Return a new *Bbox* that is padded on all four sides by the given value.

rotated(*radians*)
Return a new bounding box that bounds a rotated version of this bounding box by the given radians. The new bounding box is still aligned with the axes, of course.

shrunk(*mx*, *my*)
Return a copy of the *Bbox*, shrunk by the factor *mx* in the *x* direction and the factor *my* in the *y* direction. The lower left corner of the box remains unchanged. Normally *mx* and *my* will be less than 1, but this is not enforced.

shrunk_to_aspect(*box_aspect*, *container=None*, *fig_aspect=1.0*)
Return a copy of the *Bbox*, shrunk so that it is as large as it can be while having the desired aspect ratio, *box_aspect*. If the box coordinates are relative—that is, fractions of a larger box such as a figure—then the physical aspect ratio of that figure is specified with *fig_aspect*, so that *box_aspect* can also be given as a ratio of the absolute dimensions, not the relative dimensions.

size
(property) The width and height of the bounding box. May be negative, in the same way as *width* and *height*.

splitx(**args*)
e.g., `bbox.splitx(f1, f2, ...)`

Returns a list of new *Bbox* objects formed by splitting the original one with vertical lines at fractional positions *f1*, *f2*, ...

splity(**args*)
e.g., `bbox.splitx(f1, f2, ...)`

Returns a list of new *Bbox* objects formed by splitting the original one with horizontal lines at fractional positions *f1*, *f2*, ...

transformed(*transform*)
Return a new *Bbox* object, statically transformed by the given transform.

translated(*tx*, *ty*)
Return a copy of the *Bbox*, statically translated by *tx* and *ty*.

static **union**(*bboxes*)
Return a *Bbox* that contains all of the given bboxes.

width
(property) The width of the bounding box. It may be negative if *x1* < *x0*.

x0
(property) *x0* is the first of the pair of *x* coordinates that define the bounding box. *x0* is not guaranteed to be less than *x1*. If you require that, use *xmin*.

x1
(property) *x1* is the second of the pair of *x* coordinates that define the bounding box. *x1* is not guaranteed to be greater than *x0*. If you require that, use *xmax*.

xmax
(property) *xmax* is the right edge of the bounding box.

xmin
(property) *xmin* is the left edge of the bounding box.

y0
(property) *y0* is the first of the pair of *y* coordinates that define the bounding box. *y0* is not guaranteed to be less than *y1*. If you require that, use *ymin*.

y1
(property) *y1* is the second of the pair of *y* coordinates that define the bounding box. *y1* is not guaranteed to be greater than *y0*. If you require that, use *ymax*.

ymax
(property) *ymax* is the top edge of the bounding box.

ymin
(property) *ymin* is the bottom edge of the bounding box.

class matplotlib.transforms.**Bbox**(*points*, ***kwargs*)
Bases: `matplotlib.transforms.BboxBase`

A mutable bounding box.

points: a 2x2 numpy array of the form [[x0, y0], [x1, y1]]

If you need to create a `Bbox` object from another form of data, consider the static methods `unit()`, `from_bounds()` and `from_extents()`.

static **from_bounds**(*x0*, *y0*, *width*, *height*)
(staticmethod) Create a new `Bbox` from *x0*, *y0*, *width* and *height*.

width and *height* may be negative.

static **from_extents**(**args*)
(staticmethod) Create a new Bbox from *left*, *bottom*, *right* and *top*.

The *y*-axis increases upwards.

get_points()
Get the points of the bounding box directly as a numpy array of the form: [[x0, y0], [x1, y1]].

ignore(*value*)
Set whether the existing bounds of the box should be ignored by subsequent calls to `update_from_data()` or `update_from_data_xy()`.

value:

- When True, subsequent calls to `update_from_data()` will ignore the existing bounds of the `Bbox`.
- When False, subsequent calls to `update_from_data()` will include the existing bounds of the `Bbox`.

mutated()
return whether the bbox has changed since init

mutatedx()
return whether the x-limits have changed since init

mutatedy()
return whether the y-limits have changed since init

static **null**()
(staticmethod) Create a new null `Bbox` from (inf, inf) to (-inf, -inf).

set(*other*)
Set this bounding box from the "frozen" bounds of another `Bbox`.

set_points(*points*)
Set the points of the bounding box directly from a numpy array of the form: [[x0, y0], [x1, y1]]. No error checking is performed, as this method is mainly for internal use.

static **unit**()
(staticmethod) Create a new unit `Bbox` from (0, 0) to (1, 1).

update_from_data(*x*, *y*, *ignore=None*)

Update the bounds of the Bbox based on the passed in data. After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

x: a numpy array of *x*-values

y: a numpy array of *y*-values

ignore:

- when True, ignore the existing bounds of the Bbox.
- when False, include the existing bounds of the Bbox.
- when None, use the last value passed to ignore().

update_from_data_xy(*xy*, *ignore=None*, *updatex=True*, *updatey=True*)

Update the bounds of the Bbox based on the passed in data. After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

xy: a numpy array of 2D points

ignore:

- when True, ignore the existing bounds of the Bbox.
- when False, include the existing bounds of the Bbox.
- when None, use the last value passed to ignore().

updatex: when True, update the x values

updatey: when True, update the y values

update_from_path(*path*, *ignore=None*, *updatex=True*, *updatey=True*)

Update the bounds of the Bbox based on the passed in data. After updating, the bounds will have positive *width* and *height*; *x0* and *y0* will be the minimal values.

path: a Path instance

ignore:

- when True, ignore the existing bounds of the Bbox.
- when False, include the existing bounds of the Bbox.
- when None, use the last value passed to ignore().

updatex: when True, update the x values

updatey: when True, update the y values

class matplotlib.transforms.**TransformedBbox**(*bbox*, *transform*, ***kwargs*)

Bases: matplotlib.transforms.BboxBase

A Bbox that is automatically transformed by a given transform. When either the child bounding box or transform changes, the bounds of this bbox will update accordingly.

bbox: a child Bbox

transform: a 2D Transform

get_points()
Get the points of the bounding box directly as a numpy array of the form: [[x0, y0], [x1, y1]].

class matplotlib.transforms.**Transform**(*shorthand_name=None*)
Bases: *matplotlib.transforms.TransformNode*

The base class of all *TransformNode* instances that actually perform a transformation.

All non-affine transformations should be subclasses of this class. New affine transformations should be subclasses of *Affine2D*.

Subclasses of this class should override the following members (at minimum):

- *input_dims*
- *output_dims*
- *transform()*
- *is_separable*
- *has_inverse*
- *inverted()* (if *has_inverse* is True)

If the transform needs to do something non-standard with *matplotlib.path.Path* objects, such as adding curves where there were once line segments, it should override:

- *transform_path()*

Creates a new *TransformNode*.

shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

contains_branch(*other*)
Return whether the given transform is a sub-tree of this transform.

This routine uses transform equality to identify sub-trees, therefore in many situations it is object id which will be used.

For the case where the given transform represents the whole of this transform, returns True.

contains_branch_seperately(*other_transform*)
Returns whether the given branch is a sub-tree of this transform on each seperate dimension.

A common use for this method is to identify if a transform is a blended transform containing an axes' data transform. e.g.:

```
x_isdata, y_isdata = trans.contains_branch_seperately(ax.transData)
```

depth
Returns the number of transforms which have been chained together to form this Transform instance.

Note: For the special case of a Composite transform, the maximum depth of the two is returned.

get_affine()
Get the affine part of this transform.

get_matrix()
Get the Affine transformation array for the affine part of this transform.

has_inverse = False
True if this transform has a corresponding inverse transform.

input_dims = None
The number of input dimensions of this transform. Must be overridden (with integers) in the subclass.

inverted()
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`x === self.inverted().transform(self.transform(x))`

is_separable = False
True if this transform is separable in the x- and y- dimensions.

output_dims = None
The number of output dimensions of this transform. Must be overridden (with integers) in the subclass.

transform(*values*)
Performs the transformation on the given array of values.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_affine(*values*)
Performs only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_angles(*angles*, *pts*, *radians=False*, *pushoff=1e-05*)
Performs transformation on a set of angles anchored at specific locations.

The *angles* must be a column vector (i.e., numpy array).

The *pts* must be a two-column numpy array of x,y positions (angle transforms currently only work in 2D). This array must have the same number of rows as *angles*.

***radians* indicates whether or not input angles are given in** radians (True) or degrees (False; the default).

***pushoff* is the distance to move away from *pts* for** determining transformed angles (see discussion of method below).

The transformed angles are returned in an array with the same size as *angles*.

The generic version of this method uses a very generic algorithm that transforms *pts*, as well as locations very close to *pts*, to find the angle in the transformed system.

transform_bbox(*bbox*)

Transform the given bounding box.

Note, for smarter transforms including caching (a common requirement for matplotlib figures), see `TransformedBbox`.

transform_non_affine(*values*)

Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_path(*path*)

Returns a transformed path.

path: a `Path` instance.

In some cases, this transform may insert curves into the path that began as line segments.

transform_path_affine(*path*)

Returns a path, transformed only by the affine part of this transform.

path: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

transform_path_non_affine(*path*)

Returns a path, transformed only by the non-affine part of this transform.

path: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

transform_point(*point*)

A convenience function that returns the transformed copy of a single point.

The point is given as a sequence of length *input_dims*. The transformed point is returned as a sequence of length *output_dims*.

class matplotlib.transforms.**TransformWrapper**(*child*)
Bases: *matplotlib.transforms.Transform*

A helper class that holds a single child transform and acts equivalently to it.

This is useful if a node of the transform tree must be replaced at run time with a transform of a different type. This class allows that replacement to correctly trigger invalidation.

Note that *TransformWrapper* instances must have the same input and output dimensions during their entire lifetime, so the child transform may only be replaced with another child transform of the same dimensions.

child: A class:*Transform* instance. This child may later be replaced with *set()*.

frozen()
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

set(*child*)
Replace the current child of this transform with another one.

The new child must have the same number of input and output dimensions as the current child.

class matplotlib.transforms.**AffineBase**(**args*, ***kwargs*)
Bases: *matplotlib.transforms.Transform*

The base class of all affine transformations of any number of dimensions.

get_affine()
Get the affine part of this transform.

transform(*values*)
Performs the transformation on the given array of values.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_affine(*values*)
Performs only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_non_affine(*points*)

Performs only the non-affine part of the transformation.

transform(values) is always equivalent to transform_affine(transform_non_affine(values)).

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

transform_path(*path*)

Returns a transformed path.

path: a Path instance.

In some cases, this transform may insert curves into the path that began as line segments.

transform_path_affine(*path*)

Returns a path, transformed only by the affine part of this transform.

path: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values

transform_path_non_affine(*path*)

Returns a path, transformed only by the non-affine part of this transform.

path: a Path instance.

transform_path(path) is equivalent to transform_path_affine(transform_path_non_affine(values

class matplotlib.transforms.**Affine2DBase**(**args*, ***kwargs*)

Bases: *matplotlib.transforms.AffineBase*

The base class of all 2D affine transformations.

2D affine transformations are performed using a 3x3 numpy array:

```
a c e
b d f
0 0 1
```

This class provides the read-only interface. For a mutable 2D affine transformation, use *Affine2D*.

Subclasses of this class will generally only need to override a constructor and get_matrix() that generates a custom 3x3 matrix.

frozen()

Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where copy.deepcopy() might normally be used.

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`x === self.inverted().transform(self.transform(x))`

static matrix_from_values(*a*, *b*, *c*, *d*, *e*, *f*)

(staticmethod) Create a new transformation matrix as a 3x3 numpy array of the form:

```
a c e
b d f
0 0 1
```

to_values()

Return the values of the matrix as a sequence (a,b,c,d,e,f)

transform_affine(*points*)

Performs only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_point(*point*)

A convenience function that returns the transformed copy of a single point.

The point is given as a sequence of length `input_dims`. The transformed point is returned as a sequence of length `output_dims`.

class matplotlib.transforms.**Affine2D**(*matrix=None*, ***kwargs*)

Bases: `matplotlib.transforms.Affine2DBase`

A mutable 2D affine transformation.

Initialize an Affine transform from a 3x3 numpy float array:

```
a c e
b d f
0 0 1
```

If *matrix* is None, initialize with the identity transform.

clear()

Reset the underlying matrix to the identity transform.

static from_values(*a*, *b*, *c*, *d*, *e*, *f*)

(staticmethod) Create a new Affine2D instance from the given values:

```
a c e
b d f
0 0 1
```

.

get_matrix()
Get the underlying transformation matrix as a 3x3 numpy array:

```
a c e
b d f
0 0 1
```

.

static identity()
(staticmethod) Return a new `Affine2D` object that is the identity transform.

Unless this transform will be mutated later on, consider using the faster `IdentityTransform` class instead.

rotate(*theta*)
Add a rotation (in radians) to this transform in place.

Returns *self*, so this method can easily be chained with more calls to `rotate()`, `rotate_deg()`, `translate()` and `scale()`.

rotate_around(*x*, *y*, *theta*)
Add a rotation (in radians) around the point (x, y) in place.

Returns *self*, so this method can easily be chained with more calls to `rotate()`, `rotate_deg()`, `translate()` and `scale()`.

rotate_deg(*degrees*)
Add a rotation (in degrees) to this transform in place.

Returns *self*, so this method can easily be chained with more calls to `rotate()`, `rotate_deg()`, `translate()` and `scale()`.

rotate_deg_around(*x*, *y*, *degrees*)
Add a rotation (in degrees) around the point (x, y) in place.

Returns *self*, so this method can easily be chained with more calls to `rotate()`, `rotate_deg()`, `translate()` and `scale()`.

scale(*sx*, *sy=None*)
Adds a scale in place.

If *sy* is None, the same scale is applied in both the *x*- and *y*-directions.

Returns *self*, so this method can easily be chained with more calls to `rotate()`, `rotate_deg()`, `translate()` and `scale()`.

set(*other*)
Set this transformation from the frozen copy of another `Affine2DBase` object.

set_matrix(*mtx*)
Set the underlying transformation matrix from a 3x3 numpy array:

```
a c e
b d f
0 0 1
```

.

skew(*xShear*, *yShear*)
Adds a skew in place.

xShear and *yShear* are the shear angles along the *x*- and *y*-axes, respectively, in radians.

Returns *self*, so this method can easily be chained with more calls to `rotate()`, `rotate_deg()`, `translate()` and `scale()`.

skew_deg(*xShear*, *yShear*)
Adds a skew in place.

xShear and *yShear* are the shear angles along the *x*- and *y*-axes, respectively, in degrees.

Returns *self*, so this method can easily be chained with more calls to `rotate()`, `rotate_deg()`, `translate()` and `scale()`.

translate(*tx*, *ty*)
Adds a translation in place.

Returns *self*, so this method can easily be chained with more calls to `rotate()`, `rotate_deg()`, `translate()` and `scale()`.

class matplotlib.transforms.**IdentityTransform**(**args*, ***kwargs*)
Bases: `matplotlib.transforms.Affine2DBase`

A special class that does on thing, the identity transform, in a fast way.

frozen()
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

get_affine()
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`x === self.inverted().transform(self.transform(x))`

get_matrix()
Get the Affine transformation array for the affine part of this transform.

inverted()
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

```
x === self.inverted().transform(self.transform(x))
```

transform(*points*)

Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_affine(*points*)

Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_non_affine(*points*)

Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_path(*path*)

Returns a path, transformed only by the non-affine part of this transform.

path: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(value`

transform_path_affine(*path*)

Returns a path, transformed only by the non-affine part of this transform.

path: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(value`

transform_path_non_affine(*path*)

Returns a path, transformed only by the non-affine part of this transform.

path: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

class `matplotlib.transforms.`**BlendedGenericTransform**(*x_transform*, *y_transform*, ***kwargs*)

Bases: `matplotlib.transforms.Transform`

A "blended" transform uses one transform for the *x*-direction, and another transform for the *y*-direction.

This "generic" version can handle any given child transform in the *x*- and *y*-directions.

Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

You will generally not call this constructor directly but use the `blended_transform_factory()` function instead, which can determine automatically which kind of blended transform to create.

frozen()

Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

get_affine()

Get the affine part of this transform.

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`x === self.inverted().transform(self.transform(x))`

transform_non_affine(*points*)

Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

class `matplotlib.transforms.`**BlendedAffine2D**(*x_transform*, *y_transform*, ***kwargs*)

Bases: `matplotlib.transforms.Affine2DBase`

A "blended" transform uses one transform for the *x*-direction, and another transform for the *y*-direction.

This version is an optimization for the case where both child transforms are of type *Affine2DBase*.

Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

Both *x_transform* and *y_transform* must be 2D affine transforms.

You will generally not call this constructor directly but use the *blended_transform_factory()* function instead, which can determine automatically which kind of blended transform to create.

get_matrix()
Get the Affine transformation array for the affine part of this transform.

matplotlib.transforms.**blended_transform_factory**(*x_transform*, *y_transform*)
Create a new "blended" transform using *x_transform* to transform the *x*-axis and *y_transform* to transform the *y*-axis.

A faster version of the blended transform is returned for the case where both child transforms are affine.

class matplotlib.transforms.**CompositeGenericTransform**(*a*, *b*, ***kwargs*)
Bases: *matplotlib.transforms.Transform*

A composite transform formed by applying transform *a* then transform *b*.

This "generic" version can handle any two arbitrary transformations.

Create a new composite transform that is the result of applying transform *a* then transform *b*.

You will generally not call this constructor directly but use the *composite_transform_factory()* function instead, which can automatically choose the best kind of composite transform instance to create.

frozen()
Returns a frozen copy of this transform node. The frozen copy will not update when its children change. Useful for storing a previously known state of a transform where `copy.deepcopy()` might normally be used.

get_affine()
Get the affine part of this transform.

inverted()
Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`x === self.inverted().transform(self.transform(x))`

transform_affine(*points*)
Performs only the affine part of this transformation on the given array of values.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally a no-op. In affine transformations, this is equivalent to `transform(values)`.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

`transform_non_affine`(*points*)
Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

`transform_path_non_affine`(*path*)
Returns a path, transformed only by the non-affine part of this transform.

path: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(values`

class `matplotlib.transforms.`**`CompositeAffine2D`**(*a*, *b*, ***kwargs*)
Bases: *`matplotlib.transforms.Affine2DBase`*

A composite transform formed by applying transform *a* then transform *b*.

This version is an optimization that handles the case where both *a* and *b* are 2D affines.

Create a new composite transform that is the result of applying transform *a* then transform *b*.

Both *a* and *b* must be instances of *`Affine2DBase`*.

You will generally not call this constructor directly but use the *`composite_transform_factory()`* function instead, which can automatically choose the best kind of composite transform instance to create.

`get_matrix`()
Get the Affine transformation array for the affine part of this transform.

`matplotlib.transforms.`**`composite_transform_factory`**(*a*, *b*)
Create a new composite transform that is the result of applying transform a then transform b.

Shortcut versions of the blended transform are provided for the case where both child transforms are affine, or one or the other is the identity transform.

Composite transforms may also be created using the '+' operator, e.g.:

```
c = a + b
```

class `matplotlib.transforms.`**`BboxTransform`**(*boxin*, *boxout*, ***kwargs*)
Bases: *`matplotlib.transforms.Affine2DBase`*

BboxTransform linearly transforms points from one *Bbox* to another *Bbox*.

Create a new *BboxTransform* that linearly transforms points from *boxin* to *boxout*.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.**BboxTransformTo**(*boxout*, ***kwargs*)

Bases: *matplotlib.transforms.Affine2DBase*

BboxTransformTo is a transformation that linearly transforms points from the unit bounding box to a given *Bbox*.

Create a new *BboxTransformTo* that linearly transforms points from the unit bounding box to *boxout*.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.**BboxTransformFrom**(*boxin*, ***kwargs*)

Bases: *matplotlib.transforms.Affine2DBase*

BboxTransformFrom linearly transforms points from a given *Bbox* to the unit bounding box.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.**ScaledTranslation**(*xt*, *yt*, *scale_trans*, ***kwargs*)

Bases: *matplotlib.transforms.Affine2DBase*

A transformation that translates by *xt* and *yt*, after *xt* and *yt* have been transformad by the given transform *scale_trans*.

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.transforms.**TransformedPath**(*path*, *transform*)

Bases: *matplotlib.transforms.TransformNode*

A *TransformedPath* caches a non-affine transformed copy of the *Path*. This cached copy is automatically updated when the non-affine part of the transform changes.

Note: Paths are considered immutable by this class. Any update to the path's vertices/codes will not trigger a transform recomputation.

Create a new *TransformedPath* from the given *Path* and *Transform*.

get_fully_transformed_path()

Return a fully-transformed copy of the child path.

get_transformed_path_and_affine()

Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation.

`get_transformed_points_and_affine()`
Return a copy of the child path, with the non-affine part of the transform already applied, along with the affine part of the path necessary to complete the transformation. Unlike `get_transformed_path_and_affine()`, no interpolation will be performed.

`matplotlib.transforms.nonsingular`(*vmin*, *vmax*, *expander=0.001*, *tiny=1e-15*, *increasing=True*)
Modify the endpoints of a range as needed to avoid singularities.

vmin*, *vmax the initial endpoints.

tiny threshold for the ratio of the interval to the maximum absolute value of its endpoints. If the interval is smaller than this, it will be expanded. This value should be around 1e-15 or larger; otherwise the interval will be approaching the double precision resolution limit.

expander fractional amount by which *vmin* and *vmax* are expanded if the original interval is too small, based on *tiny*.

***increasing*: [True | False]** If True (default), swap *vmin*, *vmax* if *vmin* > *vmax*

Returns *vmin*, *vmax*, expanded and/or swapped if necessary.

If either input is inf or NaN, or if both inputs are 0 or very close to zero, it returns -*expander*, *expander*.

CHAPTER

TWENTYSIX

ADDING NEW SCALES AND PROJECTIONS TO MATPLOTLIB

Matplotlib supports the addition of custom procedures that transform the data before it is displayed.

There is an important distinction between two kinds of transformations. Separable transformations, working on a single dimension, are called "scales", and non-separable transformations, that handle data in two or more dimensions at a time, are called "projections".

From the user's perspective, the scale of a plot can be set with *set_xscale()* and *set_yscale()*. Projections can be chosen using the `projection` keyword argument to the `plot()` or `subplot()` functions, e.g.:

```
plot(x, y, projection="custom")
```

This document is intended for developers and advanced users who need to create new scales and projections for matplotlib. The necessary code for scales and projections can be included anywhere: directly within a plot script, in third-party code, or in the matplotlib source tree itself.

26.1 Creating a new scale

Adding a new scale consists of defining a subclass of *matplotlib.scale.ScaleBase*, that includes the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- A function to limit the range of the axis to acceptable values (`limit_range_for_scale()`). A log scale, for instance, would prevent the range from including values less than or equal to zero.
- Locators (major and minor) that determine where to place ticks in the plot, and optionally, how to adjust the limits of the plot to some "good" values. Unlike `limit_range_for_scale()`, which is always enforced, the range setting here is only used when automatically setting the range of the plot.
- Formatters (major and minor) that specify how the tick labels should be drawn.

Once the class is defined, it must be registered with matplotlib so that the user can select it.

A full-fledged and heavily annotated example is in `examples/api/custom_scale_example.py`. There are also some classes in *matplotlib.scale* that may be used as starting points.

26.2 Creating a new projection

Adding a new projection consists of defining a projection axes which subclasses `matplotlib.axes.Axes` and includes the following elements:

- A transformation from data coordinates into display coordinates.
- An inverse of that transformation. This is used, for example, to convert mouse positions from screen space back into data space.
- Transformations for the gridlines, ticks and ticklabels. Custom projections will often need to place these elements in special locations, and matplotlib has a facility to help with doing so.
- Setting up default values (overriding `cla()`), since the defaults for a rectilinear axes may not be appropriate.
- Defining the shape of the axes, for example, an elliptical axes, that will be used to draw the background of the plot and for clipping any data elements.
- Defining custom locators and formatters for the projection. For example, in a geographic projection, it may be more convenient to display the grid in degrees, even if the data is in radians.
- Set up interactive panning and zooming. This is left as an "advanced" feature left to the reader, but there is an example of this for polar plots in `matplotlib.projections.polar`.
- Any additional methods for additional convenience or features.

Once the projection axes is defined, it can be used in one of two ways:

- By defining the class attribute `name`, the projection axes can be registered with `matplotlib.projections.register_projection()` and subsequently simply invoked by name:

  ```
  plt.axes(projection='my_proj_name')
  ```

- For more complex, parameterisable projections, a generic "projection" object may be defined which includes the method `_as_mpl_axes`. `_as_mpl_axes` should take no arguments and return the projection's axes subclass and a dictionary of additional arguments to pass to the subclass' `__init__` method. Subsequently a parameterised projection can be initialised with:

  ```
  plt.axes(projection=MyProjection(param1=param1_value))
  ```

  where MyProjection is an object which implements a `_as_mpl_axes` method.

A full-fledged and heavily annotated example is in `examples/api/custom_projection_example.py`. The polar plot functionality in `matplotlib.projections.polar` may also be of interest.

26.3 API documentation

- `matplotlib.scale`
- `matplotlib.projections`

- *matplotlib.projections.polar*

CHAPTER

TWENTYSEVEN

DEFAULT COLOR CHANGES

As discussed at length elsewhere [insert links], `jet` is an empirically bad color map and should not be the default color map. Due to the position that changing the appearance of the plot breaks backward compatibility, this change has been put off for far longer than it should have been. In addition to changing the default color map we plan to take the chance to change the default color-cycle on plots and to adopt a different color map for filled plots (`imshow`, `pcolor`, `contourf`, etc) and for scatter like plots.

27.1 Default Heat Map Colormap

The choice of a new color map is fertile ground to bike-shedding ("No, it should be _this_ color") so we have a proposed set criteria (via Nathaniel Smith) to evaluate proposed color maps.

- it should be a sequential colormap, because diverging colormaps are really misleading unless you know where the "center" of the data is, and for a default colormap we generally won't.
- it should be perceptually uniform, i.e., human subjective judgments of how far apart nearby colors are should correspond as linearly as possible to the difference between the numerical values they represent, at least locally.
- it should have a perceptually uniform luminance ramp, i.e. if you convert to greyscale it should still be uniform. This is useful both in practical terms (greyscale printers are still a thing!) and because luminance is a very strong and natural cue to magnitude.
- it should also have some kind of variation in hue, because hue variation is a really helpful additional cue to perception, having two cues is better than one, and there's no reason not to do it.
- the hue variation should be chosen to produce reasonable results even for viewers with the more common types of colorblindness. (Which rules out things like red-to-green.)
- For bonus points, it would be nice to choose a hue ramp that still works if you throw away the luminance variation, because then we could use the version with varying luminance for 2d plots, and the version with just hue variation for 3d plots. (In 3d plots you really want to reserve the luminance channel for lighting/shading, because your brain is *really* good at extracting 3d shape from luminance variation. If the 3d surface itself has massively varying luminance then this screws up the ability to see shape.)
- Not infringe any existing IP

27.1.1 Example script

27.1.2 Proposed Colormaps

27.2 Default Scatter Colormap

For heat-map like applications it can be desirable to cover as much of the luminence scale as possible, however when color mapping markers, having markers too close to white can be a problem. For that reason we propose using a different (but maybe related) color map to the heat map for marker-based. The design parameters are the same as above, only with a more limited luminence variation.

27.2.1 Example script

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(1234)

fig, (ax1, ax2) = plt.subplots(1, 2)

N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radiuses

ax1.scatter(x, y, s=area, c=colors, alpha=0.5)


X,Y = np.meshgrid(np.arange(0, 2*np.pi, .2),
                  np.arange(0, 2*np.pi, .2))
U = np.cos(X)
V = np.sin(Y)
Q = ax2.quiver(X, Y, U, V, units='width')
qd = np.random.rand(np.prod(X.shape))
Q.set_array(qd)
```

27.2.2 Proposed Colormaps

27.3 Color Cycle / Qualitative color map

When plotting lines it is frequently desirable to plot multiple lines or artists which need to be distinguishable, but there is no inherent ordering.

27.3.1 Example script

```
import numpy as np
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)

x = np.linspace(0, 1, 10)

for j in range(10):
    ax1.plot(x, x * j)


th = np.linspace(0, 2*np.pi, 1024)
for j in np.linspace(0, np.pi, 10):
    ax2.plot(th, np.sin(th + j))

ax2.set_xlim(0, 2*np.pi)
```

27.3.2 Proposed Color cycle

CHAPTER

TWENTYEIGHT

MATPLOTLIB ENHANCEMENT PROPOSALS

28.1 MEP Template

This MEP template is a guideline of the sections that a MEP should contain. Extra sections may be added if appropriate, and unnecessary sections may be noted as such.

28.1.1 Status

MEPs go through a number of phases in their lifetime:

- **Discussion**: The MEP is being actively discussed on the mailing list and it is being improved by its author. The mailing list discussion of the MEP should include the MEP number (MEPxxx) in the subject line so they can be easily related to the MEP.
- **Progress**: Consensus was reached on the mailing list and implementation work has begun.
- **Completed**: The implementation has been merged into master.
- **Superseded**: This MEP has been abandoned in favor of another approach.

28.1.2 Branches and Pull requests

All development branches containing work on this MEP should be linked to from here.

All pull requests submitted relating to this MEP should be linked to from here. (A MEP does not need to be implemented in a single pull request if it makes sense to implement it in discrete phases).

28.1.3 Abstract

The abstract should be a short description of what the MEP will achieve.

28.1.4 Detailed description

This section describes the need for the MEP. It should describe the existing problem that it is trying to solve and why this MEP makes the situation better. It should include examples of how the new functionality would be used and perhaps some use cases.

28.1.5 Implementation

This section lists the major steps required to implement the MEP. Where possible, it should be noted where one step is dependent on another, and which steps may be optionally omitted. Where it makes sense, each step should include a link related pull requests as the implementation progresses.

28.1.6 Backward compatibility

This section describes the ways in which the MEP breaks backward incompatibility.

28.1.7 Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

28.2 MEP8: PEP8

28.2.1 Status

Discussion

28.2.2 Branches and Pull requests

None so far.

28.2.3 Abstract

The matplotlib codebase predates PEP8, and therefore is less than consistent style-wise in some areas. Bringing the codebase into compliance with PEP8 would go a long way to improving its legibility.

28.2.4 Detailed description

Some files use four space indentation, some use three. Some use different levels in the same file.

For the most part, class/function/variable naming follows PEP8, but it wouldn't hurt to fix where necessary.

28.2.5 Implementation

The implementation should be fairly mechanical: running the pep8 tool over the code and fixing where appropriate.

This should be merged in after the 2.0 release, since the changes will likely make merging any pending pull requests more difficult.

Additionally, and optionally, PEP8 compliance could be tracked by an automated build system.

28.2.6 Backward compatibility

Public names of classes and functions that require change (there shouldn't be many of these) should first be deprecated and then removed in the next release cycle.

28.2.7 Alternatives

PEP8 is a popular standard for Python code style, blessed by the Python core developers, making any alternatives less desirable.

28.3 MEP9: Global interaction manager

Add a global manager for all user interactivity with artists; make any artist resizeable, moveable, highlightable, and selectable as desired by the user.

28.3.1 Status

Discussion

28.3.2 Branches and Pull requests

https://github.com/dhyams/matplotlib/tree/MEP9

28.3.3 Abstract

The goal is to be able to interact with matplotlib artists in a very similar way as drawing programs do. When appropriate, the user should be able to move, resize, or select an artist that is already on the canvas. Of course, the script writer is ultimately in control of whether an artist is able to be interacted with, or whether it is static.

This code to do this has already been privately implemented and tested, and would need to be migrated from its current "mixin" implementation, to a bona-fide part of matplotlib.

The end result would be to have four new keywords available to matplotlib.artist.Artist: _moveable_, _resizeable_, _selectable_, and _highlightable_. Setting any one of these keywords to True would activate interactivity for that artist.

In effect, this MEP is a logical extension of event handling in matplotlib; matplotlib already supports "low level" interactions like left mouse presses, a key press, or similar. The MEP extends the support to the logical level, where callbacks are performed on the artists when certain interactive gestures from the user are detected.

28.3.4 Detailed description

This new functionality would be used to allow the end-user to better interact with the graph. Many times, a graph is almost what the user wants, but a small repositioning and/or resizing of components is necessary. Rather than force the user to go back to the script to trial-and-error the location, and simple drag and drop would be appropriate.

Also, this would better support applications that use matplotlib; here, the end-user has no reasonable access or desire to edit the underlying source in order to fine-tune a plot. Here, if matplotlib offered the capability, one could move or resize artists on the canvas to suit their needs. Also, the user should be able to highlight (with a mouse over) an artist, and select it with a double-click, if the application supports that sort of thing. In this MEP, we also want to support the highlighting and selection natively; it is up to application to handle what happens when the artist is selected. A typical handling would be to display a dialog to edit the properties of the artist.

In the future, as well (this is not part of this MEP), matplotlib could offer backend-specific property dialogs for each artist, which are raised on artist selection. This MEP would be a necessary stepping stone for that sort of capability.

There are currently a few interactive capabilities in matplotlib (e.g. legend.draggable()), but they tend to be scattered and are not available for all artists. This MEP seeks to unify the interactive interface and make it work for all artists.

The current MEP also includes grab handles for resizing artists, and appropriate boxes drawn when artists are moved or resized.

28.3.5 Implementation

- Add appropriate methods to the "tree" of artists so that the interactivity manager has a consistent interface for the interactivity manager to deal with. The proposed methods to add to the artists, if they are to support interactivity, are:
  - get_pixel_position_ll(self): get the pixel position of the lower left corner of the artist's bounding box
  - get_pixel_size(self): get the size of the artist's bounding box, in pixels
  - set_pixel_position_and_size(self,x,y,dx,dy): set the new size of the artist, such that it fits within the specified bounding box.
- add capability to the backends to 1) provide cursors, since these are needed for visual indication of moving/resizing, and 2) provide a function that gets the current mouse position
- Implement the manager. This has already been done privately (by dhyams) as a mixin, and has been tested quite a bit. The goal would be to move the functionality of the manager into the artists so that it is in matplotlib properly, and not as a "monkey patch" as I currently have it coded.

28.3.6 Current summary of the mixin

(Note that this mixin is for now just private code, but can be added to a branch obviously)

InteractiveArtistMixin:

Mixin class to make any generic object that is drawn on a matplotlib canvas moveable and possibly resizeable. The Powerpoint model is followed as closely as possible; not because I'm enamoured with Powerpoint, but because that's what most people understand. An artist can also be selectable, which means that the artist will receive the on_activated() callback when double clicked. Finally, an artist can be highlightable, which means that a highlight is drawn on the artist whenever the mouse passes over. Typically, highlightable artists will also be selectable, but that is left up to the user. So, basically there are four attributes that can be set by the user on a per-artist basis:

- highlightable
- selectable
- moveable
- resizeable

To be moveable (draggable) or resizeable, the object that is the target of the mixin must support the following protocols:

- get_pixel_position_ll(self)
- get_pixel_size(self)
- set_pixel_position_and_size(self,x,y,sx,sy)

Note that nonresizeable objects are free to ignore the sx and sy parameters. To be highlightable, the object that is the target of the mixin must also support the following protocol:

- get_highlight(self)

Which returns a list of artists that will be used to draw the highlight.

If the object that is the target of the mixin is not an matplotlib artist, the following protocols must also be implemented. Doing so is usually fairly trivial, as there has to be an artist *somewhere* that is being drawn. Typically your object would just route these calls to that artist.

- get_figure(self)
- get_axes(self)
- contains(self,event)
- set_animated(self,flag)
- draw(self,renderer)
- get_visible(self)

The following notifications are called on the artist, and the artist can optionally implement these.

- on_select_begin(self)
- on_select_end(self)
- on_drag_begin(self)
- on_drag_end(self)

- on_activated(self)
- on_highlight(self)
- on_right_click(self,event)
- on_left_click(self,event)
- on_middle_click(self,event)
- on_context_click(self,event)
- on_key_up(self,event)
- on_key_down(self,event)

The following notifications are called on the canvas, if no interactive artist handles the event:

- on_press(self,event)
- on_left_click(self,event)
- on_middle_click(self,event)
- on_right_click(self,event)
- on_context_click(self,event)
- on_key_up(self,event)
- on_key_down(self,event)

The following functions, if present, can be used to modify the behavior of the interactive object:

- press_filter(self,event) # determines if the object wants to have the press event routed to it
- handle_unpicked_cursor() # can be used by the object to set a cursor as the cursor passes over the object when it is unpicked.

Supports multiple canvases, maintaining a drag lock, motion notifier, and a global "enabled" flag per canvas. Supports fixed aspect ratio resizings by holding the shift key during the resize.

Known problems:

- Zorder is not obeyed during the selection/drag operations. Because of the blit technique used, I do not believe this can be fixed. The only way I can think of is to search for all artists that have a zorder greater then me, set them all to animated, and then redraw them all on top during each drag refresh. This might be very slow; need to try.
- the mixin only works for wx backends because of two things: 1) the cursors are hardcoded, and 2) there is a call to wx.GetMousePosition() Both of these shortcomings are reasonably fixed by having each backend supply these things.

28.3.7 Backward compatibility

No problems with backward compatibility, although once this is in place, it would be appropriate to obsolete some of the existing interactive functions (like legend.draggable())

28.3.8 Alternatives

None that I know of.

28.4 MEP10: Docstring consistency

28.4.1 Status

Progress

Targeted for 1.3

28.4.2 Branches and Pull requests

#1665 #1757 #1795

28.4.3 Abstract

matplotlib has a great deal of inconsistency between docstrings. This not only makes the docs harder to read, but it is harder on contributors, because they don't know which specifications to follow. There should be a clear docstring convention that is followed consistently.

The organization of the API documentation is difficult to follow. Some pages, such as pyplot and axes, are enormous and hard to browse. There should instead be short summary tables that link to detailed documentation. In addition, some of the docstrings themselves are quite long and contain redundant information.

Building the documentation takes a long time and uses a `make.py` script rather than a Makefile.

28.4.4 Detailed description

There are number of new tools and conventions available since matplotlib started using Sphinx that make life easier. The following is a list of proposed changes to docstrings, most of which involve these new features.

Numpy docstring format

Numpy docstring format: This format divides the docstring into clear sections, each having different parsing rules that make the docstring easy to read both as raw text and as HTML. We could consider alternatives, or invent our own, but this is a strong choice, as it's well used and understood in the Numpy/Scipy community.

Cross references

Most of the docstrings in matplotlib use explicit "roles" when linking to other items, for example: `:func:`myfunction``. As of Sphinx 0.4, there is a "default_role" that can be set to "obj", which will polymorphically link to a Python object of any type. This allows one to write ``myfunction`` instead. This makes docstrings much easier to read and edit as raw text. Additionally, Sphinx allows for setting a current module, so links like ``~matplotlib.axes.Axes.set_xlim`` could be written as ``~axes.Axes.set_xlim``.

Overriding signatures

Many methods in matplotlib use the `*args` and `**kwargs` syntax to dynamically handle the keyword arguments that are accepted by the function, or to delegate on to another function. This, however, is often not useful as a signature in the documentation. For this reason, many matplotlib methods include something like:

```
def annotate(self, *args, **kwargs):
    """
    Create an annotation: a piece of text referring to a data
    point.

    Call signature::

      annotate(s, xy, xytext=None, xycoords='data',
               textcoords='data', arrowprops=None, **kwargs)
    """
```

This can't be parsed by Sphinx, and is rather verbose in raw text. As of Sphinx 1.1, if the `autodoc_docstring_signature` config value is set to True, Sphinx will extract a replacement signature from the first line of the docstring, allowing this:

```
def annotate(self, *args, **kwargs):
    """
    annotate(s, xy, xytext=None, xycoords='data',
             textcoords='data', arrowprops=None, **kwargs)

    Create an annotation: a piece of text referring to a data
    point.
    """
```

The explicit signature will replace the actual Python one in the generated documentation.

Linking rather than duplicating

Many of the docstrings include long lists of accepted keywords by interpolating things into the docstring at load time. This makes the docstrings very long. Also, since these tables are the same across many docstrings, it inserts a lot of redundant information in the docs – particularly a problem in the printed version.

These tables should be moved to docstrings on functions whose only purpose is for help. The docstrings that refer to these tables should link to them, rather than including them verbatim.

autosummary extension

The Sphinx autosummary extension should be used to generate summary tables, that link to separate pages of documentation. Some classes that have many methods (e.g. `Axes.axes`) should be documented with one method per page, whereas smaller classes should have all of their methods together.

Examples linking to relevant documentation

The examples, while helpful at illustrating how to use a feature, do not link back to the relevant docstrings. This could be addressed by adding module-level docstrings to the examples, and then including that docstring in the parsed content on the example page. These docstrings could easily include references to any other part of the documentation.

Documentation using help() vs a browser

Using Sphinx markup in the source allows for good-looking docs in your browser, but the markup also makes the raw text returned using help() look terrible. One of the aims of improving the docstrings should be to make both methods of accessing the docs look good.

28.4.5 Implementation

1. The numpydoc extensions should be turned on for matplotlib. There is an important question as to whether these should be included in the matplotlib source tree, or used as a dependency. Installing Numpy is not sufficient to get the numpydoc extensions – it's a separate install procedure. In any

case, to the extent that they require customization for our needs, we should endeavor to submit those changes upstream and not fork them.

2. Manually go through all of the docstrings and update them to the new format and conventions. Updating the cross references (from `:func:`myfunc`` to ``func``) may be able to be semi-automated. This is a lot of busy work, and perhaps this labor should be divided on a per-module basis so no single developer is over-burdened by it.
3. Reorganize the API docs using autosummary and `sphinx-autogen`. This should hopefully have minimal impact on the narrative documentation.
4. Modify the example page generator (`gen_rst.py`) so that it extracts the module docstring from the example and includes it in a non-literal part of the example page.
5. Use `sphinx-quickstart` to generate a new-style Sphinx Makefile. The following features in the current `make.py` will have to be addressed in some other way:
   - Copying of some static content
   - Specifying a "small" build (only low-resolution PNG files for examples)

Steps 1, 2, and 3 are interdependent. 4 and 5 may be done independently, though 5 has some dependency on 3.

28.4.6 Backward compatibility

As this mainly involves docstrings, there should be minimal impact on backward compatibility.

28.4.7 Alternatives

None yet discussed.

28.5 MEP11: Third-party dependencies

- *Alternatives*

This MEP attempts to improve the way in which third-party dependencies in matplotlib are handled.

28.5.1 Status

Completed – needs to be merged

28.5.2 Branches and Pull requests

#1157: Use automatic dependency resolution

#1290: Debundle pyparsing

#1261: Update six to 1.2

28.5.3 Abstract

One of the goals of matplotlib has been to keep it as easy to install as possible. To that end, some third-party dependencies are included in the source tree and, under certain circumstances, installed alongside matplotlib. This MEP aims to resolve some problems with that approach, bring some consistency, while continuing to make installation convenient.

At the time that was initially done, `setuptools`, `easy_install` and `PyPI` were not mature enough to be relied on. However, at present, we should be able to safely leverage the "modern" versions of those tools, `distribute` and `pip`.

While matplotlib has dependencies on both Python libraries and C/C++ libraries, this MEP addresses only the Python libraries so as to not confuse the issue. C libraries represent a larger and mostly orthogonal set of problems.

28.5.4 Detailed description

matplotlib depends on the following third-party Python libraries:

- Numpy
- dateutil (pure Python)
- pytz (pure Python)
- six – required by dateutil (pure Python)
- pyparsing (pure Python)
- PIL (optional)
- GUI frameworks: pygtk, gobject, tkinter, PySide, PyQt4, wx (all optional, but one is required for an interactive GUI)

Current behavior

When installing from source, a `git` checkout or `pip`:

- `setup.py` attempts to `import numpy`. If this fails, the installation fails.
- For each of `dateutil`, `pytz` and `six`, `setup.py` attempts to import them (from the top-level namespace). If that fails, matplotlib installs its local copy of the library into the top-level namespace.
- `pyparsing` is always installed inside of the matplotlib namespace.

This behavior is most surprising when used with `pip`, because no `pip` dependency resolution is performed, even though it is likely to work for all of these packages.

The fact that `pyparsing` is installed in the matplotlib namespace has reportedly (#1290) confused some users into thinking it is a matplotlib-related module and import it from there rather than the top-level.

When installing using the Windows installer, `dateutil`, `pytz` and `six` are installed at the top-level *always*, potentially overwriting already installed copies of those libraries.

TODO: Describe behavior with the OS-X installer.

When installing using a package manager (Debian, RedHat, MacPorts etc.), this behavior actually does the right thing, and there are no special patches in the matplotlib packages to deal with the fact that we handle `dateutil`, `pytz` and `six` in this way. However, care should be taken that whatever approach we move to continues to work in that context.

Maintaining these packages in the matplotlib tree and making sure they are up-to-date is a maintenance burden. Advanced new features that may require a third-party pure Python library have a higher barrier to inclusion because of this burden.

Desired behavior

Third-party dependencies are downloaded and installed from their canonical locations by leveraging `pip`, `distribute` and `PyPI`.

`dateutil`, `pytz`, and `pyparsing` should be made into optional dependencies – though obviously some features would fail if they aren't installed. This will allow the user to decide whether they want to bother installing a particular feature.

28.5.5 Implementation

For installing from source, and assuming the user has all of the C-level compilers and dependencies, this can be accomplished fairly easily using `distribute` and following the instructions here. The only anticipated change to the matplotlib library code will be to import `pyparsing` from the top-level namespace rather than from within matplotlib. Note that `distribute` will also allow us to remove the direct dependency on `six`, since it is, strictly speaking, only a direct dependency of `dateutil`.

For binary installations, there are a number of alternatives (here ordered from best/hardest to worst/easiest):

1. The distutils wininst installer allows a post-install script to run. It might be possible to get this script to run `pip` to install the other dependencies. (See this thread for someone who has trod that ground before).
2. Continue to ship `dateutil`, `pytz`, `six` and `pyparsing` in our installer, but use the post-install-script to install them *only* if they can not already be found.
3. Move all of these packages inside a (new) `matplotlib.extern` namespace so it is clear for outside users that these are external packages. Add some conditional imports in the core matplotlib codebase so `dateutil` (at the top-level) is tried first, and failing that `matplotlib.extern.dateutil` is used.

2 and 3 are undesirable as they still require maintaining copies of these packages in our tree – and this is exacerbated by the fact that they are used less – only in the binary installers. None of these 3 approaches address Numpy, which will still have to be manually installed using an installer.

TODO: How does this relate to the Mac OS-X installer?

28.5.6 Backward compatibility

At present, matplotlib can be installed from source on a machine without the third party dependencies and without an internet connection. After this change, an internet connection (and a working PyPI) will be required to install matplotlib for the first time. (Subsequent matplotlib updates or development work will run without accessing the network).

28.5.7 Alternatives

Distributing binary `eggs` doesn't feel like a usable solution. That requires getting `easy_install` installed first, and Windows users generally prefer the well known `exe` or `msi` installer that works out of the box.

28.6 MEP12: Improve Gallery and Examples

28.6.1 Status

Progress

Initial changes added in 1.3. Conversion of the gallery is on-going. 29 September 2015 - The last `pylab_examples` where `pylab` is imported has been converted over to use `matplotlib pyplot` and `numpy`.

28.6.2 Branches and Pull requests

#1623, #1924, #2181

PR `#2474 <https://github.com/matplotlib/matplotlib/pull/2474>_` demonstrates a single example being cleaned up and moved to the appropriate section.

28.6.3 Abstract

Reorganizing the matplotlib plot gallery would greatly simplify navigation of the gallery. In addition, examples should be cleaned-up and simplified for clarity.

28.6.4 Detailed description

The matplotlib gallery was recently set up to split examples up into sections. As discussed in that PR [1], the current example sections (`api`, `pylab_examples`) aren't terribly useful to users: New sections in the gallery would help users find relevant examples.

These sections would also guide a cleanup of the examples: Initially, all the current examples would remain and be listed under their current directories. Over time, these examples could be cleaned up and moved into one of the new sections.

This process allows users to easily identify examples that need to be cleaned up; i.e. anything in the `api` and `pylab_examples` directories.

28.6.5 Implementation

1. Create new gallery sections. [Done]
2. Clean up examples and move them to the new gallery sections (over the course of many PRs and with the help of many users/developers). [In progress]

[1] http://github.com/matplotlib/matplotlib/pull/714

Gallery sections

The naming of sections is critical and will guide the clean-up effort. The current sections are:

- Lines, bars, and markers (more-or-less 1D data)
- Shapes and collections
- Statistical plots
- Images, contours, and fields
- Pie and polar charts: Round things
- Color
- Text, labels, and annotations
- Ticks and spines
- Subplots, axes, and figures
- Specialty plots (e.g., sankey, radar, tornado)
- Showcase (plots with tweaks to make them publication-quality)
- separate sections for toolboxes (already exists: 'mplot3d', 'axes_grid', 'units', 'widgets')

These names are certainly up for debate. As these sections grow, we should reevaluate them and split them up as necessary.

Clean up guidelines

The current examples in the `api` and `pylab_examples` sections of the gallery would remain in those directories until they are cleaned up. After clean-up, they would be moved to one of the new gallery sections described above. "Clean-up" should involve:

- PEP8 clean-ups (running flake8, or a similar checker, is highly recommended)
- Commented-out code should be removed.
- Add introductory sentence or paragraph in the main docstring. See 6d1b8a2.
- Replace uses of `pylab` interface with `pyplot` (+ `numpy`, etc.). See c25ef1e
- Remove shebang line, e.g.:

  #!/usr/bin/env python

- Use consistent imports. In particular:

  import numpy as np

  import matplotlib.pyplot as plt

  Avoid importing specific functions from these modules (e.g. `from numpy import sin`)

- Each example should focus on a specific feature (excluding `showcase` examples, which will show more "polished" plots). Tweaking unrelated to that feature should be removed. See f7b2217, e57b5fc, and 1458aa8

Use of `pylab` should be demonstrated/discussed on a dedicated help page instead of the gallery examples.

Note: When moving an existing example, you should search for references to that example. For example, the API documentation for `axes.py` and `pyplot.py` may use these examples to generate plots. Use your favorite search tool (e.g., grep, ack, grin, pss) to search the matplotlib package. See 2dc9a46 and aa6b410

Additional suggestions

- Provide links (both ways) between examples and API docs for the methods/objects used. (issue #2222)
- Use `plt.subplots` (note trailing "s") in preference over `plt.subplot`.
- Rename the example to clarify it's purpose. For example, the most basic demo of `imshow` might be `imshow_demo.py`, and one demonstrating different interpolation settings would be `imshow_demo_interpolation.py` (*not* `imshow_demo2.py`).
- Split up examples that try to do too much. See 5099675 and fc2ab07
- Delete examples that don't show anything new.
- Some examples exercise esoteric features for unit testing. These tweaks should be moved out of the gallery to an example in the `unit` directory located in the root directory of the package.
- Add plot titles to clarify intent of the example. See bd2b13c

28.6.6 Backward compatibility

The website for each Matplotlib version is readily accessible, so users who want to refer to old examples can still do so.

28.6.7 Alternatives

Tags

Tagging examples will also help users search the example gallery. Although tags would be a big win for users with specific goals, the plot gallery will remain the entry point to these examples, and sections could really help users navigate the gallery. Thus, tags are complementary to this reorganization.

28.7 MEP13: Use properties for Artists

28.7.1 Status

- **Discussion**

28.7.2 Branches and Pull requests

None

28.7.3 Abstract

Wrap all of the matplotlib getter and setter methods with python properties, allowing them to be read and written like class attributes.

28.7.4 Detailed description

Currently matplotlib uses getter and setter functions (usually prefixed with get_ and set_, respectively) for reading and writing data related to classes. However, since 2.6 python supports properties, which allow such setter and getter functions to be accessed as though they were attributes. This proposal would implement all existing setter and getter methods as properties.

28.7.5 Implementation

1. All existing getter and setter methods will need to have two aliases, one with the get_ or set_ prefix and one without. Getter methods that currently lack prefixes should be recording in a text file.

2. Classes should be reorganized so setter and getter methods are sequential in the code, with getter methods first.
3. Getter and setter methods the provide additional optional optional arguments should have those arguments accessible in another manner, either as additional getter or setter methods or attributes of other classes. If those classes are not accessible, getters for them should be added.
4. Property decorators will be added to the setter and getter methods without the prefix. Those with the prefix will be marked as deprecated.
5. Docstrings will need to be rewritten so the getter with the prefix has the current docstring and the getter without the prefix has a generic docstring appropriate for an attribute.
6. Automatic alias generation will need to be modified so it will also create aliases for the properties.
7. All instances of getter and setter method calls will need to be changed to attribute access.
8. All setter and getter aliases with prefixes will be removed

The following steps can be done simultaneously: 1, 2, and 3; 4 and 5; 6 and 7.

Only the following steps must be done in the same release: 4, 5, and 6. All other changes can be done in separate releases. 8 should be done several major releases after everything else.

28.7.6 Backward compatibility

All existing getter methods that do not have a prefix (such as get_) will need to be changed from function calls to attribute access. In most cases this will only require removing the parenthesis.

setter and getter methods that have additional optional arguments will need to have those arguments implemented in another way, either as a separate property in the same class or as attributes or properties of another class.

Cases where the setter returns a value will need to be changed to using the setter followed by the getter.

Cases where there are set_ATTR_on() and set_ATTR_off() methods will be changed to ATTR_on properties.

28.7.7 Examples

axes.Axes.set_axis_off/set_axis_on

Current implementation:

```
axes.Axes.set_axis_off()
axes.Axes.set_axis_on()
```

New implementation:

```
True = axes.Axes.axis_on
False = axes.Axes.axis_on
axes.Axes.axis_on = True
axes.Axes.axis_on = False
```

axes.Axes.get_xlim/set_xlim and get_autoscalex_on/set_autoscalex_on

Current implementation:

```
[left, right] = axes.Axes.get_xlim()
auto = axes.Axes.get_autoscalex_on()

[left, right] = axes.Axes.set_xlim(left=left, right=right, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(left=left, right=None, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(left=None, right=right, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(left=left, emit=emit, auto=auto)
[left, right] = axes.Axes.set_xlim(right=right, emit=emit, auto=auto)

axes.Axes.set_autoscalex_on(auto)
```

New implementation:

```
[left, right] = axes.Axes.axes_xlim
auto = axes.Axes.autoscalex_on

axes.Axes.axes_xlim = [left, right]
axes.Axes.axes_xlim = [left, None]
axes.Axes.axes_xlim = [None, right]
axes.Axes.axes_xlim[0] = left
axes.Axes.axes_xlim[1] = right

axes.Axes.autoscalex_on = auto

axes.Axes.emit_xlim = emit
```

axes.Axes.get_title/set_title

Current implementation:

```
string = axes.Axes.get_title()
axes.Axes.set_title(string, fontdict=fontdict, **kwargs)
```

New implementation:

```
string = axes.Axes.title
string = axes.Axes.title_text.text

text.Text = axes.Axes.title_text
text.Text.<attribute> = attribute
text.Text.fontdict = fontdict

axes.Axes.title = string
axes.Axes.title = text.Text
axes.Axes.title_text = string
axes.Axes.title_text = text.Text
```

axes.Axes.get_xticklabels/set_xticklabels

Current implementation:

```
[text.Text] = axes.Axes.get_xticklabels()
[text.Text] = axes.Axes.get_xticklabels(minor=False)
[text.Text] = axes.Axes.get_xticklabels(minor=True)
[text.Text] = axes.Axes.([string], fontdict=None, **kwargs)
[text.Text] = axes.Axes.([string], fontdict=None, minor=False, **kwargs)
[text.Text] = axes.Axes.([string], fontdict=None, minor=True, **kwargs)
```

New implementation:

```
[text.Text] = axes.Axes.xticklabels
[text.Text] = axes.Axes.xminorticklabels
axes.Axes.xticklabels = [string]
axes.Axes.xminorticklabels = [string]
axes.Axes.xticklabels = [text.Text]
axes.Axes.xminorticklabels = [text.Text]
```

28.7.8 Alternatives

Instead of using decorators, it is also possible to use the property function. This would change the procedure so that all getter methods that lack a prefix will need to be renamed or removed. This makes handling docstrings more difficult and harder to read.

It is not necessary to deprecate the setter and getter methods, but leaving them in will complicate the code.

This could also serve as an opportunity to rewrite or even remove automatic alias generation.

Another alternate proposal:

Convert `set_xlim`, `set_xlabel`, `set_title`, etc. to `xlim`, `xlabel`, `title`,... to make the transition from `plt` functions to `axes` methods significantly simpler. These would still be methods, not properties, but it's still a great usability enhancement while retaining the interface.

28.8 MEP14: Text handling

- *Alternatives*

28.8.1 Status

- **Discussion**

28.8.2 Branches and Pull requests

Issue #253 demonstrates a bug where using the bounding box rather than the advance width of text results in misaligned text. This is a minor point in the grand scheme of things, but it should be addressed as part of this MEP.

28.8.3 Abstract

By reorganizing how text is handled, this MEP aims to:

- improve support for Unicode and non-ltr languages
- improve text layout (especially multi-line text)
- allow support for more fonts, especially non-Apple-format TrueType fonts and OpenType fonts.
- make the font configuration easier and more transparent

28.8.4 Detailed description

Text layout

At present, matplotlib has two different ways to render text: "built-in" (based on FreeType and our own Python code), and "usetex" (based on calling out to a TeX installation). Adjunct to the "built-in" renderer there is also the Python-based "mathtext" system for rendering mathematical equations using a subset of the TeX language without having a TeX installation available. Support for these two engines in strewn about many source files, including every backend, where one finds clauses like

```
if rcParams['text.usetex']: # do one thing else: # do another
```

Adding a third text rendering approach (more on that later) would require editing all of these places as well, and therefore doesn't scale.

Instead, this MEP proposes adding a concept of "text engines", where the user could select one of many different approaches for rendering text. The implementations of each of these would be localized to their own set of modules, and not have little pieces around the whole source tree.

Why add more text rendering engines? The "built-in" text rendering has a number of shortcomings.

- It only handles right-to-left languages, and doesn't handle many special features of Unicode, such as combining diacriticals.

- The multiline support is imperfect and only supports manual line-breaking – it can not break up a paragraph into lines of a certain length.
- It also does not handle inline formatting changes in order to support something like Markdown, reStructuredText or HTML. (Though rich-text formatting is contemplated in this MEP, since we want to make sure this design allows it, the specifics of a rich-text formatting implementation is outside of the scope of this MEP.)

Supporting these things is difficult, and is the "full-time job" of a number of other projects:

- pango/harfbuzz
- QtTextLayout
- Microsoft DirectWrite
- Apple Core Text

Of the above options, it should be noted that `harfbuzz` is designed from the start as a cross platform option with minimal dependencies, so therefore is a good candidate for a single option to support.

Additionally, for supporting rich text, we could consider using WebKit, and possibly whether than represents a good single cross-platform option. Again, however, rich text formatting is outside of the scope of this project.

Rather than trying to reinvent the wheel and add these features to matplotlib's "built-in" text renderer, we should provide a way to leverage these projects to get more powerful text layout. The "built-in" renderer will still need to exist for reasons of ease of installation, but its feature set will be more limited compared to the others. [TODO: This MEP should clearly decide what those limited features are, and fix any bugs to bring the implementation into a state of working correctly in all cases that we want it to work. I know @leejjoon has some thoughts on this.]

Font selection

Going from an abstract description of a font to a file on disk is the task of the font selection algorithm – it turns out to be much more complicated than it seems at first.

The "built-in" and "usetex" renderers have very different ways of handling font selection, given their different technologies. TeX requires the installation of TeX-specific font packages, for example, and can not use TrueType fonts directly. Unfortunately, despite the different semantics for font selection, the same set of font properties are used for each. This is true of both the `FontProperties` class and the font-related `rcParams` (which basically share the same code underneath). Instead, we should define a core set of font selection parameters that will work across all text engines, and have engine-specific configuration to allow the user to do engine-specific things when required. For example, it is possible to directly select a font by name in the "built-in" using `font.family`, but the same is not possible with "usetex". It may be possible to make it easier to use TrueType fonts by using XeTeX, but users will still want to use the traditional metafonts through TeX font packages. So the issue still stands that different text engines will need engine-specific configuration, and it should be more obvious to the user which configuration will work across text engines and which are engine-specific.

Note that even excluding "usetex", there are different ways to find fonts. The default is to use the font list cache in `font_manager.py` which matches fonts using our own algorithm based on the CSS font matching algorithm. It doesn't always do the same thing as the native font selection algorithms on Linux (fontconfig), Mac and Windows, and it doesn't always find all of the fonts on the system that the OS would normally pick up. However, it is cross-platform, and always finds the fonts that ship with matplotlib. The Cairo and MacOSX backends (and presumably a future HTML5-based backend) currently bypass this mechanism and use the OS-native ones. The same is true when not embedding fonts in SVG, PS or PDF files and opening them in a third-party viewer. A downside there is that (at least with Cairo, need to confirm with MacOSX) they don't always find the fonts we ship with matplotlib. (It may be possible to add the fonts to their search path, though, or we may need to find a way to install our fonts to a location the OS expects to find them).

There are also special modes in the PS and PDF to only use the core fonts that are always available to those formats. There, the font lookup mechanism must only match against those fonts. It is unclear whether the OS-native font lookup systems can handle this case.

There is also experimental support for using fontconfig for font selection in matplotlib, turned off by default. fontconfig is the native font selection algorithm on Linux, but is also cross platform and works well on the other platforms (though obviously is an additional dependency there).

Many of the text layout libraries proposed above (pango, QtTextLayout, DirectWrite and CoreText etc.) insist on using the font selection library from their own ecosystem.

All of the above seems to suggest that we should move away from our self-written font selection algorithm and use the native APIs where possible. That's what Cairo and MacOSX backends already want to use, and it will be a requirement of any complex text layout library. On Linux, we already have the bones of a `fontconfig` implementation (which could also be accessed through pango). On Windows and Mac we may need to write custom wrappers. The nice thing is that the API for font lookup is relatively small, and essentially consist of "given a dictionary of font properties, give me a matching font file".

Font subsetting

Font subsetting is currently handled using ttconv. ttconv was a standalone commandline utility for converting TrueType fonts to subsetted Type 3 fonts (among other features) written in 1995, which matplotlib (well, I) forked in order to make it work as a library. It only handles Apple-style TrueType fonts, not ones with the Microsoft (or other vendor) encodings. It doesn't handle OpenType fonts at all. This means that even though the STIX fonts come as .otf files, we have to convert them to .ttf files to ship them with matplotlib. The Linux packagers hate this – they'd rather just depend on the upstream STIX fonts. ttconv has also been shown to have a few bugs that have been difficult to fix over time.

Instead, we should be able to use FreeType to get the font outlines and write our own code (probably in Python) to output subsetted fonts (Type 3 on PS and PDF and SVGFonts or paths on SVG). Freetype, as a popular and well-maintained project, handles a wide variety of fonts in the wild. This would remove a lot of custom C code, and remove some code duplication between backends.

Note that subsetting fonts this way, while the easiest route, does lose the hinting in the font, so we will need to continue, as we do now, provide a way to embed the entire font in the file where possible.

Alternative font subsetting options include using the subsetting built-in to Cairo (not clear if it can be used without the rest of Cairo), or using `fontforge` (which is a heavy and not terribly cross-platform dependency).

Freetype wrappers

Our FreeType wrapper could really use a reworking. It defines its own image buffer class (when a Numpy array would be easier). While FreeType can handle a huge diversity of font files, there are limitations to our wrapper that make it much harder to support non-Apple-vendor TrueType files, and certain features of

OpenType files. (See #2088 for a terrible result of this, just to support the fonts that ship with Windows 7 and 8). I think a fresh rewrite of this wrapper would go a long way.

Text anchoring and alignment and rotation

The handling of baselines was changed in 1.3.0 such that the backends are now given the location of the baseline of the text, not the bottom of the text. This is probably the correct behavior, and the MEP refactoring should also follow this convention.

In order to support alignment on multi-line text, it should be the responsibility of the (proposed) text engine to handle text alignment. For a given chunk of text, each engine calculates a bounding box for that text and the offset of the anchor point within that box. Therefore, if the va of a block was "top", the anchor point would be at the top of the box.

Rotating of text should always be around the anchor point. I'm not sure that lines up with current behavior in matplotlib, but it seems like the sanest/least surprising choice. [This could be revisited once we have something working]. Rotation of text should not be handled by the text engine – that should be handled by a layer between the text engine and the rendering backend so it can be handled in a uniform way. [I don't see any advantage to rotation being handled by the text engines individually...]

There are other problems with text alignment and anchoring that should be resolved as part of this work. [TODO: enumerate these].

Other minor problems to fix

The mathtext code has backend-specific code – it should instead provide its output as just another text engine. However, it's still desirable to have mathtext layout inserted as part of a larger layout performed by another text engine, so it should be possible to do this. It's an open question whether embedding the text layout of an arbitrary text engine in another should be possible.

The text mode is currently set by a global rcParam ("text.usetex") so it's either all on or all off. We should continue to have a global rcParam to choose the text engine ("text.layout_engine"), but it should under the hood be an overridable property on the `Text` object, so the same figure can combine the results of multiple text layout engines if necessary.

28.8.5 Implementation

A concept of a "text engine" will be introduced. Each text engine will implement a number of abstract classes. The `TextFont` interface will represent text for a given set of font properties. It isn't necessarily limited to a single font file – if the layout engine supports rich text, it may handle a number of font files in a family. Given a `TextFont` instance, the user can get a `TextLayout` instance, which represents the layout for a given string of text in a given font. From a `TextLayout`, an iterator over `TextSpans` is returned so the engine can output raw editable text using as few spans as possible. If the engine would rather get individual characters, they can be obtained from the `TextSpan` instance:

```
class TextFont(TextFontBase):
    def __init__(self, font_properties):
        """
        Create a new object for rendering text using the given font properties.
        """
        pass
```

```
    def get_layout(self, s, ha, va):
        """
        Get the TextLayout for the given string in the given font and
        the horizontal (left, center, right) and verticalalignment (top,
        center, baseline, bottom)
        """
        pass

class TextLayout(TextLayoutBase):
    def get_metrics(self):
        """
        Return the bounding box of the layout, anchored at (0, 0).
        """
        pass

    def get_spans(self):
        """
        Returns an iterator over the spans of different in the layout.
        This is useful for backends that want to editable raw text as
        individual lines.  For rich text where the font may change,
        each span of different font type will have its own span.
        """
        pass

    def get_image(self):
        """
        Returns a rasterized image of the text.  Useful for raster backends,
        like Agg.

        In all likelihood, this will be overridden in the backend, as it can
        be created from get_layout(), but certain backends may want to
        override it if their library provides it (as freetype does).
        """
        pass

    def get_rectangles(self):
        """
        Returns an iterator over the filled black rectangles in the layout.
        Used by TeX and mathtext for drawing, for example, fraction lines.
        """
        pass

    def get_path(self):
        """
        Returns a single Path object of the entire layed out text.

        [Not strictly necessary, but might be useful for textpath
        functionality]
        """
        pass

class TextSpan(TextSpanBase):
    x, y      # Position of the span -- relative to the text layout as a whole
```

```
            # where (0, 0) is the anchor.  y is the baseline of the span.
    fontfile  # The font file to use for the span
    text      # The text content of the span

    def get_path(self):
        pass  # See TextLayout.get_path

    def get_chars(self):
        """
        Returns an iterator over the characters in the span.
        """
        pass

class TextChar(TextCharBase):
    x, y      # Position of the character -- relative to the text layout as
              # a whole, where (0, 0) is the anchor.  y is in the baseline
              # of the character.
    codepoint # The unicode code point of the character -- only for informational
              # purposes, since the mapping of codepoint to glyph_id may have been
              # handled in a complex way by the layout engine.  This is an int
              # to avoid problems on narrow Unicode builds.
    glyph_id  # The index of the glyph within the font
    fontfile  # The font file to use for the char

    def get_path(self):
        """
        Get the path for the character.
        """
pass
```

Graphic backends that want to output subset of fonts would likely build up a file-global dictionary of characters where the keys are (fontname, glyph_id) and the values are the paths so that only one copy of the path for each character will be stored in the file.

Special casing: The “usetex” functionality currently is able to get Postscript directly from TeX to insert directly in a Postscript file, but for other backends, parses a DVI file and generates something more abstract. For a case like this, `TextLayout` would implement `get_spans` for most backends, but add `get_ps` for the Postscript backend, which would look for the presence of this method and use it if available, or fall back to `get_spans`. This kind of special casing may also be necessary, for example, when the graphics backend and text engine belong to the same ecosystem, e.g. Cairo and Pango, or MacOSX and CoreText.

There are three main pieces to the implementation:

1. Rewriting the freetype wrapper, and removing ttconv.

1. Once (1) is done, as a proof of concept, we can move to the upstream STIX .otf fonts

2. Add support for web fonts loaded from a remote URL. (Enabled by using freetype for font subsetting).

2. Refactoring the existing “builtin” and “usetex” code into separate text engines and to follow the API outlined above.

3. Implementing support for advanced text layout libraries.

(1) and (2) are fairly independent, though having (1) done first will allow (2) to be simpler. (3) is dependent on (1) and (2), but even if it doesn't get done (or is postponed), completing (1) and (2) will make it easier to move forward with improving the "builtin" text engine.

28.8.6 Backward compatibility

The layout of text with respect to its anchor and rotation will change in hopefully small, but improved, ways. The layout of multiline text will be much better, as it will respect horizontal alignment. The layout of bidirectional text or other advanced Unicode features will now work inherently, which may break some things if users are currently using their own workarounds.

Fonts will be selected differently. Hacks that used to sort of work between the "builtin" and "usetex" text rendering engines may no longer work. Fonts found by the OS that weren't previously found by matplotlib may be selected.

28.8.7 Alternatives

TBD

28.9 MEP15 - Fix axis autoscaling when limits are specified for one axis only

28.9.1 Status

Discussion

28.9.2 Branches and Pull requests

None so far.

28.9.3 Abstract

When one axis of a 2-dimensional plot if overridden via `xlim` or `ylim`, automatic scaling of the remaining axis should be based on the data that falls within the specified limits of the first axis.

28.9.4 Detailed description

When axis limits for a 2-D plot are specified for one axis only (via `xlim` or `ylim`), matplotlib currently does not currently rescale the other axis. The result is that the displayed curves or symbols may be compressed into a tiny portion of the available area, so that the final plot conveys much less information than it would with appropriate axis scaling.

The proposed change of behavior would make matplotlib choose the scale for the remaining axis using only the data that falls within the limits for the axis where limits were specified.

28.9.5 Implementation

I don't know enough about the internals of matplotlib to be able to suggest an implementation.

28.9.6 Backward compatibility

From the standpoint of software interfaces, there would be no break in backward compatibility. Some outputs would be different, but if the user truly desires the previous behavior, he/she can achieve this by overriding the axis scaling for both axes.

28.9.7 Alternatives

The only alternative that I can see is to maintain the status quo.

28.10 MEP19: Continuous Integration

28.10.1 Status

Discussion

28.10.2 Branches and Pull requests

28.10.3 Abstract

matplotlib could benefit from better and more reliable continuous integration, both for testing and building installers and documentation.

28.10.4 Detailed description

Current state-of-the-art

Testing

matplotlib currently uses Travis-CI for automated tests. While Travis-CI should be praised for how much it does as a free service, it has a number of shortcomings:

- It often fails due to network timeouts when installing dependencies.
- It often fails for inexplicable reasons.
- build or test products can only be saved from build off of branches on the main repo, not pull requsts, so it is often difficult to "post mortem" analyse what went wrong. This is particularly frustrating when the failure can not be subsequently reproduced locally.
- It is not extremely fast. matplotlib's cpu and memory requirements for testing are much higher than the average Python project.
- It only tests on Ubuntu Linux, and we have only minimal control over the specifics of the platform. It can be upgraded at any time outside of our control, causing unexpected delays at times that may not be convenient in our release schedule.

On the plus side, Travis-CI's integration with github – automatically testing all pending pull requests – is exceptional.

Builds

There is no centralized effort for automated binary builds for matplotlib. However, the following disparate things are being done [If the authors mentioned here could fill in detail, that would be great!]:

- @sandrotosi: builds Debian packages
- @takluyver: Has automated Ubuntu builds on Launchpad
- @cgohlke: Makes Windows builds (don't know how automated that is)
- @r-owen: Makes OS-X builds (don't know how automated that is)

Documentation

Documentation of master is now built by travis and uploaded to http://matplotlib.org/devdocs/index.html

@NelleV, I believe, generates the docs automatically and posts them on the web to chart MEP10 progress.

Peculiarities of matplotlib

matplotlib has complex requirements that make testing and building more taxing than many other Python projects.

- The CPU time to run the tests is quite high. It puts us beyond the free accounts of many CI services (e.g. ShiningPanda)
- It has a large number of dependencies, and testing the full matrix of all combinations is impractical. We need to be clever about what space we test and guarantee to support.

Requirements

This section outlines the requirements that we would like to have.

1. Testing all pull requests by hooking into the Github API, as Travis-CI does
2. Testing on all major platforms: Linux, Mac OS-X, MS Windows (in that order of priority, based on user survey)
3. Retain the last n days worth of build and test products, to aid in post-mortem debugging.
4. Automated nightly binary builds, so that users can test the bleeding edge without installing a complete compilation environment.
5. Automated benchmarking. It would be nice to have a standard benchmark suite (separate from the tests) whose performance could be tracked over time, in different backends and platforms. While this is separate from building and testing, ideally it would run on the same infrastructure.
6. Automated nightly building and publishing of documentation (or as part of testing, to ensure PRs don't introduce documentation bugs). (This would not replace the static documentation for stable releases as a default).
7. The test systems should be managable by multiple developers, so that no single person becomes a bottleneck. (Travis-CI's design does this well – storing build configuration in the git repository, rather than elsewhere, is a very good design.)
8. Make it easy to test a large but sparse matrix of different versions of matplotlib's dependencies. The matplotlib user survey provides some good data as to where to focus our efforts: https://docs.google.com/spreadsheet/ccc?key=0AjrPjlTMRTwTdHpQS25pcTZIRWdqX0pNckNSU01sMHc#gid=0
9. Nice to have: A decentralized design so that those with more obscure platforms can publish build results to a central dashboard.

28.10.5 Implementation

This part is yet-to-be-written.

However, ideally, the implementation would be a third-party service, to avoid adding system administration to our already stretched time. As we have some donated funds, this service may be a paid one if it offers significant time-saving advantages over free offerings.

28.10.6 Backward compatibility

Backward compatibility is not a major concern for this MEP. We will replace current tools and procedures with something better and throw out the old.

28.10.7 Alternatives

28.10.8 Hangout Notes

CI Infrastructure

- We like Travis and it will probably remain part of our arsenal in any event. The reliability issues are being looked into.
- Enable Amazon S3 uploads of testing products on Travis. This will help with post-mortem of failures (@mdboom is looking into this now).
- We want Mac coverage. The best bet is probably to push Travis to enable it for our project by paying them for a Pro account (since they don't otherwise allow testing on both Linux and Mac).
- We want Windows coverage. Shining Panda is an option there.
- Investigate finding or building a tool that would collect and synthesize test results from a number of sources and post it to Github using the Github API. This may be of general use to the Scipy community.
- For both Windows and Mac, we should document (or better yet, script) the process of setting up the machine for a build, and how to build binaries and installers. This may require getting information from Russel Owen and Christoph Gohlke. This is a necessary step for doing automated builds, but would also be valuable for a number of other reasons.

The test framework itself

- We should investigate ways to make it take less time
  - Eliminating redundant tests, if possible
  - General performance improvements to matplotlib will help
- We should be covering more things, particularly more backends
- We should have more unit tests, fewer integration tests, if possible

28.11 MEP21: color and cm refactor

28.11.1 Status

- **Discussion**: This MEP has not commenced yet, but here are some ongoing ideas which may become a part of this MEP:

28.11.2 Branches and Pull requests

28.11.3 Abstract

- color
  - tidy up the namespace
  - Define a "Color" class
  - make it easy to convert from one color type to another `'hex -> RGB'`, `'RGB -> hex'`, `'HSV -> RGB'` etc.
  - improve the construction of a colormap - the dictionary approach is archaic and overly complex (though incredibly powerful)
  - make it possible to interpolate between two or more color types in different modes, especially useful for construction of colormaps in HSV space for instance
- cm
  - rename the module to something more descriptive - mappables?

Overall, there are a lot of improvements that can be made with matplotlib color handling - managing backwards compatibility will be difficult as there are some badly named variables/modules which really shouldn't exist - but a clear path and message for migration should be available, with a large amount of focus on this in the API changes documentation.

28.11.4 Detailed description

28.11.5 Implementation

28.11.6 Backward compatibility

28.11.7 Alternatives

28.12 MEP22: Toolbar rewrite

28.12.1 Status

Progress

28.12.2 Branches and Pull requests

Previous work

- https://github.com/matplotlib/matplotlib/pull/1849
- https://github.com/matplotlib/matplotlib/pull/2557
- https://github.com/matplotlib/matplotlib/pull/2465

Pull Requests:

- Removing the NavigationToolbar classes https://github.com/matplotlib/matplotlib/pull/2740 **CLOSED**
- Keeping the NavigationToolbar classes https://github.com/matplotlib/matplotlib/pull/2759 **CLOSED**
- Navigation by events: https://github.com/matplotlib/matplotlib/pull/3652

28.12.3 Abstract

The main goal of this MEP is to make it easier to modify (add, change, remove) the way the user interacts with the figures.

The user interaction with the figure is deeply integrated within the Canvas and Toolbar. Making extremely difficult to do any modification.

This MEP proposes the separation of this interaction into Toolbar, Navigation and Tools to provide independent access and reconfiguration.

This approach will make easier to create and share tools among users. In the far future, we can even foresee a kind of Marketplace for `Tools` where the most popular can be added into the main distribution.

28.12.4 Detailed description

The reconfiguration of the Toolbar is complex, most of the time it requires a custom backend.

The creation of custom Tools sometimes interferes with the Toolbar, as example see https://github.com/matplotlib/matplotlib/issues/2694 also the shortcuts are hardcoded and again not easily modifiable https://github.com/matplotlib/matplotlib/issues/2699

The proposed solution is to take the actions out of the `Toolbar` and the shortcuts out of the `Canvas`. This actions and shortcuts will be in the form of `Tools`.

A new class `Navigation` will be the bridge between the events from the `Canvas` and `Toolbar` and redirect them to the appropiate `Tool`.

At the end the user interaction will be divided into three classes:

- NavigationBase: This class is instantiated for each FigureManager and connect the all user interactions with the Tools
- ToolbarBase: This existing class is relegated only as a GUI access to Tools.
- ToolBase: Is the basic definition of Tools.

28.12.5 Implementation

ToolBase(object)

Tools can have a graphical representation as the `SubplotTool` or not even be present in the Toolbar as `Quit`

The `ToolBase` has the following class attributes for configuration at definition time

- keymap = None: Key(s) to be used to trigger the tool
- description = ‘’: Small description of the tool
- image = None: Image that is used in the toolbar

The following instance attributes are set at instantiation:

- name
- navigation

Methods

- trigger(self, event): This is the main method of the Tool, it is called when the Tool is triggered by: * Toolbar button click * keypress associated with the Tool Keymap * Call to navigation.trigger_tool(name)
- set_figure(self, figure): Set the figure and navigation attributes
- `destroy(self, *args)`: Destroy the `Tool` graphical interface (if exists)

Available Tools

- ToolQuit

- ToolEnableAllNavigation
- ToolEnableNavigation
- ToolToggleGrid
- ToolToggleFullScreen
- ToolToggleYScale
- ToolToggleXScale
- ToolHome
- ToolBack
- ToolForward
- SaveFigureBase
- ConfigureSubplotsBase

ToolToggleBase(ToolBase)

The `ToolToggleBase` has the following class attributes for configuration at definition time

- radio_group = None: Attribute to group 'radio' like tools (mutually exclusive)
- cursor = None: Cursor to use when the tool is active

The **Toggleable** Tools, can capture keypress, mouse moves, and mouse button press

It defines the following methods

- enable(self, event): Called by `ToolToggleBase.trigger` method
- disable(self, event): Called when the tool is untoggled
- toggled : **Property** True or False

Available Tools

- ToolZoom
- ToolPan

NavigationBase

Defines the following attributes

- canvas:
- **keypresslock: Lock to know if the `canvas` key_press_event' is** available and process it
- messagelock: Lock to know if the message is available to write

Public methods for User use:

- nav_connect(self, s, func): Connect to to navigation for events

- nav_disconnect(self, cid): Disconnect from navigation event
- message_event(self, message, sender=None): Emit a tool_message_event event
- active_toggle(self): **Property** The currently toggled tools or None
- get_tool_keymap(self, name): Return a list of keys that are associated with the tool
- set_tool_keymap(self, name, `*keys`): Set the keys for the given tool
- remove_tool(self, name): Removes tool from the navigation control.
- add_tools(self, tools): Add multiple tools to `Navigation`
- add_tool(self, name, tool, group=None, position=None): Add a tool to the Navigation
- tool_trigger_event(self, name, sender=None, canvasevent=None, data=None): Trigger a tool and fire the event
- tools(self) **Property**: Return a dict with available tools with corresponding keymaps, descriptions and objects
- get_tool(self, name): Return the tool object

ToolbarBase

Methods for Backend implementation

- add_toolitem(self, name, group, position, image, description, toggle): Add a toolitem to the toolbar. This method is a callback from `tool_added_event` (emited by navigation)
- set_message(self, s): Display a message on toolbar or in status bar
- toggle_toolitem(self, name): Toggle the toolitem without firing event.
- remove_toolitem(self, name): Remove a toolitem from the `Toolbar`

28.12.6 Backward compatibility

For backward compatibility added a 'navigation' key to `rcsetup.validate_toolbar`, that is used for Navigation classes instantiation instead of the NavigationToolbar classes

With this parameter, it makes it transparent to anyone using the existing backends.

[@pelson comment: This also gives us an opportunity to avoid needing to implement all of this in the same PR - some backends can potentially exist without the new functionality for a short while (but it must be done at some point).]

28.13 MEP23: Multiple Figures per GUI window

28.13.1 Status

Discussion

28.13.2 Branches and Pull requests

Previous work - https://github.com/matplotlib/matplotlib/pull/2465 **To-delete**

28.13.3 Abstract

Add the possibility to have multiple figures grouped under the same `FigureManager`

28.13.4 Detailed description

Under the current structure, every canvas has its own window.

This is and may continue to be the desired method of operation for most use cases.

Sometimes when there are too many figures open at the same time, it is desirable to be able to group these under the same window [see](https://github.com/matplotlib/matplotlib/issues/2194).

The proposed solution modifies `FigureManagerBase` to contain and manage more than one `canvas`. The settings parameter `rcParams['backend.multifigure']` control when the **MultiFigure** behaviour is desired.

Note

It is important to note, that the proposed solution, assumes that the [MEP22](https://github.com/matplotlib/matplotlib/wiki/Mep22) is already in place. This is simply because the actual implementation of the `Toolbar` makes it pretty hard to switch between canvases.

28.13.5 Implementation

The first implementation will be done in `GTK3` using a Notebook as canvas container.

`FigureManagerBase`

will add the following new methods

- `add_canvas`: To add a canvas to an existing `FigureManager` object
- `remove_canvas`: To remove a canvas from a `FigureManager` object, if it is the last one, it will be destroyed
- `move_canvas`: To move a canvas from one `FigureManager` to another.
- `set_canvas_title`: To change the title associated with a specific canvas container
- `get_canvas_title`: To get the title associated with a specific canvas container
- `get_active_canvas`: To get the canvas that is in the foreground and is subject to the gui events. There is no `set_active_canvas` because the active canvas, is defined when `show` is called on a `Canvas` object.

`new_figure_manager`

To control which `FigureManager` will contain the new figures, an extra optional parameter `figuremanager` will be added, this parameter value will be passed to `new_figure_manager_given_figure`

`new_figure_manager_given_figure`

- If `figuremanager` parameter is give, this `FigureManager` object will be used instead of creating a new one.
- If `rcParams['backend.multifigure'] == True`: The last `FigureManager` object will be used instead of creating a new one.

`NavigationBase`

Modifies the `NavigationBase` to keep a list of canvases, directing the actions to the active one

28.13.6 Backward compatibility

For the **MultiFigure** properties to be visible, the user has to activate them directly setting `rcParams['backend.multifigure'] = True`

It should be backwards compatible for backends that adhere to the current `FigureManagerBase` structure even if they have not implemented the **MultiFigure** magic yet.

28.13.7 Alternatives

Insted of modifing the `FigureManagerBase` it could be possible to add a parallel class, that handles the cases where `rcParams['backend.multifigure'] = True`. This will warranty that there won't be any problems with custom made backends, but also makes bigger the code, and more things to mantain.

28.14 MEP24: negative radius in polar plots

28.14.1 Status

Discussion

28.14.2 Branches and Pull requests

None

28.14.3 Abstract

It is clear that polar plots need to be able to gracefully handle negative r values (not by clipping or reflection).

28.14.4 Detailed description

One obvious application that we should support is bB plots (see https://github.com/matplotlib/matplotlib/issues/1730#issuecomment-40815837), but this seems more generally useful (for example growth rate as a function of angle). The assumption in the current code (as I understand it) is that the center of the graph is `r==0`, however it would be good to be able to set the center to be at any `r` (with any value less than the off set clipped).

28.14.5 Implementation

28.14.6 Related Issues

#1730, #1603, #2203, #2133

28.14.7 Backward compatibility

28.14.8 Alternatives

28.15 MEP25: Serialization

28.15.1 Status

Discussion

28.15.2 Branches and Pull requests

- development branches:
- related pull requests:

28.15.3 Abstract

This MEP aims at adding a serializable `Controller` objects to act as an `Artist` managers. Users would then communicate changes to an `Artist` via a `Controller`. In this way, functionality of the `Controller` objects may be added incrementally since each `Artist` is still responsible for drawing everything. The goal is to create an API that is usable both by graphing libraries requiring high-level descriptions of figures and libraries requiring low-level interpretations.

28.15.4 Detailed description

Matplotlib is a core plotting engine with an API that many users already understand. It's difficult/impossible for other graphing libraries to (1) get a complete figure description, (2) output raw data from the figure object as the user has provided it, (3) understand the semantics of the figure objects without heuristics, and (4) give matplotlib a complete figure description to visualize. In addition, because an `Artist` has no conception of its own semantics within the figure, it's difficult to interact with them in a natural way.

In this sense, matplotlib will adopt a standard Model-View-Controller (MVC) framework. The *Model* will be the user defined data, style, and semantics. The *Views* are the ensemble of each individual `Artist`, which are responsible for producing the final image based on the *model*. The *Controller* will be the `Controller` object managing its set of `Artist` objects.

The `Controller` must be able to export the information that it's carrying about the figure on command, perhaps via a `to_json` method or similar. Because it would be extremely extraneous to duplicate all of the information in the model with the controller, only user-specified information (data + style) are explicitly kept. If a user wants more information (defaults) from the view/model, it should be able to query for it.

- This might be annoying to do, non-specified kwargs are pulled from the rcParams object which is in turn created from reading a user specified file and can be dynamically changed at run time. I suppose we could keep a dict of default defaults and compare against that. Not clear how this will interact with the style sheet [[MEP26]] - @tacaswell

Additional Notes:

- The `raw data` does not necessarily need to be a `list`, `ndarray`, etc. Rather, it can more abstractly just have a method to yield data when needed.
- Because the `Controller` will contain extra information that users may not want to keep around, it should *not* be created by default. You should be able to both (a) instantiate a `Controller` with a figure and (b) build a figure with a `Controller`.

Use Cases:

- Export all necessary informat
- Serializing a matplotlib figure, saving it, and being able to rerun later.
- Any other source sending an appropriately formatted representation to matplotlib to open

28.15.5 Examples

Here are some examples of what the controllers should be able to do.

1. Instantiate a matplotlib figure from a serialized representation (e.g., JSON):

```
import json
from matplotlib.controllers import Controller
with open('my_figure') as f:
    o = json.load(f)
c = Controller(o)
fig = c.figure
```

2. Manage artists from the controller (e.g., Line2D):

```
# not really sure how this should look
c.axes[0].lines[0].color = 'b'
# ?
```

3. Export serializable figure representation:

```
o = c.to_json()
# or... we should be able to throw a figure object in there too
o = Controller.to_json(mpl_fig)
```

28.15.6 Implementation

1. Create base `Controller` objects that are able to manage `Artist` objects (e.g., `Hist`)

   Comments:

   - initialization should happen via unpacking `**`, so we need a copy of call signature parameter for the `Artist` we're ultimately trying to control. Unfortunate hard-coded repetition...
   - should the additional `**kwargs` accepted by each `Artist` be tracked at the `Controller`
   - how does a `Controller` know which artist belongs where? E.g., do we need to pass `axes` references?

   Progress:

   - A simple NB demonstrating some functionality for `Line2DController` objects: http://nbviewer.ipython.org/gist/theengineear/f0aa8d79f64325e767c0

2. Write in protocols for the `Controller` to *update* the model.

   Comments:

   - how should containers be dealt with? E.g., what happens to old patches when we re-bin a histogram?
   - in the link from (1), the old line is completely destroyed and redrawn, what if something is referencing it?

3. Create method by which a json object can be assembled from the `Controllers`

4. Deal with serializing the unserializable aspects of a figure (e.g., non-affine transforms?)

5. Be able to instantiate from a serialized representation
6. Reimplement the existing pyplot and Axes method, e.g. `pyplot.hist` and `Axes.hist` in terms of the new controller class.

> @theengineer: in #2 above, what do you mean by *get updates* from each `Artist`?

^ Yup. The `Controller` *shouldn't* need to get updated. This just happens in #3. Delete comments when you see this.

28.15.7 Backward compatibility

- pickling will change
- non-affine transformations will require a defined pickling method

28.15.8 Alternatives

PR #3150 suggested adding semantics by parasitically attaching extra containers to axes objects. This is a more complete solution with what should be a more developed/flexible/powerful framework.

28.16 MEP26: Artist styling

28.16.1 Status

Proposed

28.16.2 Branches and Pull requests

28.16.3 Abstract

This MEP proposes a new stylesheet implementation to allow more comprehensive and dynamic styling of artists.

The current version of matplotlib (1.4.0) allows stylesheets based on the rcParams syntax to be applied before creation of a plot. The methodology below proposes a new syntax, based on CSS, which would allow styling of individual artists and properties, which can be applied dynamically to existing objects.

This is related to (and makes steps toward) the overall goal of moving to a DOM/tree-like architecture.

28.16.4 Detailed description

Currently, the look and appearance of existing artist objects (figure, axes, Line2D etc...) can only be updated via `set_` and `get_` methods on the artist object, which is quite laborious, especially if no reference to the artist(s) has been stored. The new style sheets introduced in 1.4 allow styling before a plot is created, but do not offer any means to dynamically update plots or distinguish between artists of the same type (i.e. to specifiy the `line color` and `line style` separately for differing `Line2D` objects).

The initial development should concentrate on allowing styling of artist primitives (those `artists` that do not contain other `artists`), and further development could expand the CSS syntax rules and parser to allow more complex styling. See the appendix for a list of primitives.

The new methodology would require development of a number of steps:

- A new stylesheet syntax (likely based on CSS) to allow selection of artists by type, class, id etc...
- A mechanism by which to parse a stylesheet into a tree
- A mechanism by which to translate the parse-tree into something which can be used to update the properties of relevant artists. Ideally this would implement a method by which to traverse the artists in a tree-like structure.
- A mechanism by which to generate a stylesheet from existing artist properties. This would be useful to allow a user to export a stylesheet from an existing figure (where the appearance may have been set using the matplotlib API)...

28.16.5 Implementation

It will be easiest to allow a '3rd party' to modify/set the style of an artist if the 'style' is created as a separate class and store against the artist as a property. The `GraphicsContext` class already provides a the basis of a `Style` class and an artists `draw` method can be refactored to use the `Style` class rather than setting up it's own `GraphicsContext` and transferring it's style-related properties to it. A minimal example of how this could be implemented is shown here: https://github.com/JamesRamm/mpl_experiment

IMO, this will also make the API and code base much neater as individual get/set methods for artist style properties are now redundant... Indirectly related would be a general drive to replace get/set methods with properties. Implementing the style class with properties would be a big stride toward this...

For initial development, I suggest developing a syntax based on a much (much much) simplified version of CSS. I am in favour of dubbing this Artist Style Sheets :+1: :

BNF Grammar

I propose a very simple syntax to implement initially (like a proof of concept), which can be expanded upon in the future. The BNF form of the syntax is given below and then explained

```
RuleSet ::= SelectorSequence "{"Declaration"}"

SelectorSequence :: = Selector {"," Selector}

Declaration ::= propName":" propValue";"

Selector ::= ArtistIdent{"#"Ident}

propName ::= Ident

propValue ::= Ident | Number | Colour | "None"
```

`ArtistIdent`, `Ident`, `Number` and `Colour` are tokens (the basic building blocks of the expression) which are defined by regular expressions.

Syntax

A CSS stylesheet consists of a series of **rule sets** in hierarchical order (rules are applied from top to bottom). Each rule follows the syntax

```
selector {attribute: value;}
```

Each rule can have any number of `attribute`: `value` pairs, and a stylesheet can have any number of rules.

The initial syntax is designed only for `artist` primitives. It does not address the question of how to set properties on `container` types (whose properties may themselves be `artists` with settable properties), however, a future solution to this could simply be nested `RuleSet` s

Selectors

Selectors define the object to which the attribute updates should be applied. As a starting point, I propose just 2 selectors to use in initial development:

Artist Type Selector

Select an `artist` by it's type. E.g `Line2D` or `Text`:

```
Line2D {attribute: value}
```

The regex for matching the artist type selector (`ArtistIdent` in the BNF grammar) would be:

```
ArtistIdent = r'(?P<ArtistIdent>
↪\bLine2D\b|\bText\b|\bAxesImage\b|\bFigureImage\b|\bPatch\b)'
```

GID selector

Select an `artist` by its `gid`:

```
Line2D#myGID {attribute: value}
```

A `gid` can be any string, so the regex could be as follows:

```
Ident = r'(?P<Ident>[a-zA-Z_][a-zA-Z_0-9]*)'
```

The above selectors roughly correspond to their CSS counterparts (http://www.w3.org/TR/CSS21/selector.html)

Attributes and values

- `Attributes` are any valid (settable) property for the `artist` in question.
- `Values` are any valid value for the property (Usually a string, or number).

Parsing

Parsing would consist of breaking the stylesheet into tokens (the python cookbook gives a nice tokenizing recipe on page 66), applying the syntax rules and constructing a `Tree`. This requires defining the grammar of the stylesheet (again, we can borrow from CSS) and writing a parser. Happily, there is a recipe for this in the python cookbook aswell.

Visitor pattern for matplotlib figure

In order to apply the stylesheet rules to the relevant artists, we need to 'visit' each artist in a figure and apply the relevant rule. Here is a visitor class (again, thanks to python cookbook), where each `node` would be an artist in the figure. A `visit_` method would need to be implemented for each mpl artist, to handle the different properties for each

```
class Visitor:
    def visit(self, node):
       name = 'visit_' + type(node).__name__
       meth = getattr(self, name, None)
       if meth is None:
          raise NotImplementedError
       return meth(node)
```

An `evaluator` class would then take the stylesheet rules and implement the visitor on each one of them.

28.16.6 Backward compatibility

Implementing a separate `Style` class would break backward compatibility as many get/set methods on an artist would become redundant. While it would be possible to alter these methods to hook into the `Style` class (stored as a property against the artist), I would be in favor of simply removing them to both neaten/simplify the codebase and to provide a simple, uncluttered API...

28.16.7 Alternatives

No alternatives, but some of the ground covered here overlaps with MEP25, which may assist in this development

28.16.8 Appendix

Matplotlib primitives

This will form the initial selectors which stylesheets can use.

- Line2D
- Text
- AxesImage
- FigureImage
- Patch

28.17 MEP27: decouple pyplot from backends

28.17.1 Status

Discussion

28.17.2 Branches and Pull requests

Main PR (including GTK3): + https://github.com/matplotlib/matplotlib/pull/4143

Backend specific branch diffs: + https://github.com/OceanWolf/matplotlib/compare/backend-refactor...OceanWolf:backend-refactor-tkagg + https://github.com/OceanWolf/matplotlib/compare/backend-refactor...OceanWolf:backend-refactor-qt + https://github.com/OceanWolf/matplotlib/compare/backend-refactor...backend-refactor-wx

28.17.3 Abstract

This MEP refactors the backends to give a more structured and consistent API, removing generic code and consolidate existing code. To do this we propose splitting:

1. `FigureManagerBase` and its derived classes into the core functionality class `FigureManager` and a backend specific class `WindowBase` and
2. `ShowBase` and its derived classes into `Gcf.show_all` and `MainLoopBase`.

28.17.4 Detailed description

This MEP aims to consolidate the backends API into one single uniform API, removing generic code out of the backend (which includes `_pylab_helpers` and `Gcf`), and push code to a more appropriate level in matplotlib. With this we automatically remove inconsistencies that appear in the backends, such as `FigureManagerBase.resize(w, h)` which sometimes sets the canvas, and other times set the entire window to the dimensions given, depending on the backend.

Two main places for generic code appear in the classes derived from `FigureManagerBase` and `ShowBase`.

1. `FigureManagerBase` has **three** jobs at the moment:

(a) The documentation describes it as a *‘‘Helper class for pyplot mode, wraps everything up into a neat bundle’’*

(b) But it doesn't just wrap the canvas and toolbar, it also does all of the windowing tasks itself. The conflation of these two tasks gets seen the best in the following line: `‘python self.set_window_title("Figure %d" % num) ‘` This combines backend specific code `self.set_window_title(title)` with matplotlib generic code `title = "Figure %d" % num`.

(c) Currently the backend specific subclass of `FigureManager` decides when to end the mainloop. This also seems very wrong as the figure should have no control over the other figures.

2. `ShowBase` has two jobs:

(a) It has the job of going through all figure managers registered in `_pylab_helpers.Gcf` and telling them to show themselves.

(b) And secondly it has the job of performing the backend specific `mainloop` to block the main programme and thus keep the figures from dying.

28.17.5 Implementation

The description of this MEP gives us most of the solution:

1. To remove the windowing aspect out of `FigureManagerBase` letting it simply wrap this new class along with the other backend classes. Create a new `WindowBase` class that can handle this functionality, with pass-through methods (:arrow_right:) to `WindowBase`. Classes that subclass `WindowBase` should also subclass the GUI specific window class to ensure backward compatibility (`manager.window == manager.window`).

2. Refactor the mainloop of `ShowBase` into `MainLoopBase`, which encapsulates the end of the loop as well. We give an instance of `MainLoop` to `FigureManager` as a key unlock the exit method (requiring all keys returned before the loop can die). Note this opens the possibility for multiple backends to run concurrently.

3. Now that `FigureManagerBase` has no backend specifics in it, to rename it to `FigureManager`, and move to a new file `backend_managers.py` noting that:

   (a) This allows us to break up the conversion of backends into separate PRs as we can keep the existing `FigureManagerBase` class and its dependencies intact.

   (b) and this also anticipates MEP22 where the new `NavigationBase` has morphed into a backend independent `ToolManager`.

FigureManagerBase(canvas, num)	FigureManager(figure, num)	`WindowBase(title)`	Notes
show		show	
destroy	calls destroy on all components	destroy	
full_screen_toggle	handles logic	set_fullscreen	
resize		resize	
key_press	key_press		
show_popup	show_poup		Not used anywhere in mpl, and does nothing.
get_window_title		get_window_title	
set_window_title		set_window_title	
	_get_toolbar		A common method to all subclasses of FigureManagerBase
		set_default_size	
		add_element_to_window	

ShowBase	MainLoopBase	Notes
mainloop	begin	
	end	Gets called automagically when no more instances of the subclass exist
__call__		Method moved to Gcf.show_all

28.17.6 Future compatibility

As eluded to above when discussing MEP 22, this refactor makes it easy to add in new generic features. At the moment, MEP 22 has to make ugly hacks to each class extending from `FigureManagerBase`. With this code, this only needs to get made in the single `FigureManager` class. This also makes the later deprecation of `NavigationToolbar2` very straightforward, only needing to touch the single `FigureManager` class

MEP 23 makes for another use case where this refactored code will come in very handy.

28.17.7 Backward compatibility

As we leave all backend code intact, only adding missing methods to existing classes, this should work seamlessly for all use cases. The only difference will lie for backends that used `FigureManager.resize` to resize the canvas and not the window, due to the standardisation of the API.

I would envision that the classes made obsolete by this refactor get deprecated and removed on the same timetable as `NavigationToolbar2`, also note that the change in call signature to the `FigureCanvasWx` constructor, while backward compatible, I think the old (imho ugly style) signature should get deprecated and removed in the same manner as everything else.

back-end	manager.resize(w,h)	Extra
gtk3	window	
Tk	canvas	
Qt	window	
Wx	canvas	FigureManagerWx had `frame` as an alias to window, so this also breaks BC.

28.17.8 Alternatives

If there were any alternative solutions to solving the same problem, they should be discussed here, along with a justification for the chosen approach.

28.17.9 Questions

Mdehoon: Can you elaborate on how to run multiple backends concurrently?

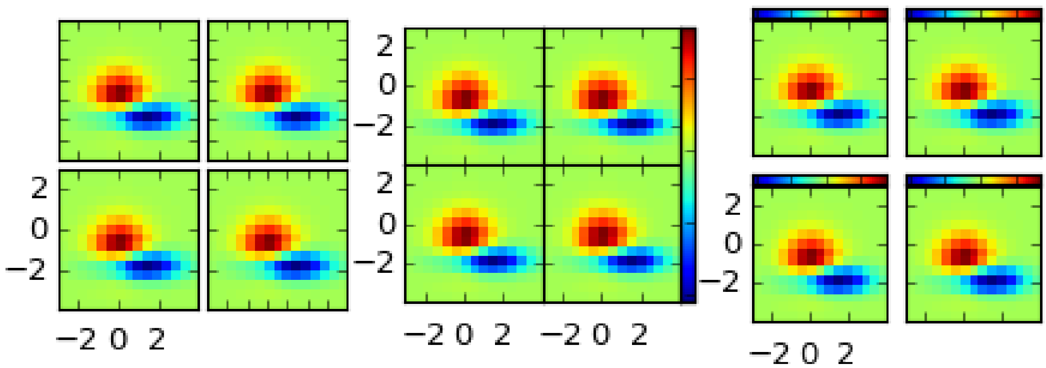
OceanWolf: @mdehoon, as I say, not for this MEP, but I see this MEP opens it up as a future possibility. Basically the `MainLoopBase` class acts a per backend Gcf, in this MEP it tracks the number of figures open per backend, and manages the mainloops for those backends. It closes the backend specific mainloop when it detects that no figures remain open for that backend. Because of this I imagine that with only a small amount of tweaking that we can do full-multi-backend matplotlib. No idea yet why one would want to, but I leave the possibility there in MainLoopBase. With all the backend-code specifics refactored out of `FigureManager` also aids in this, one manager to rule them (the backends) all.

Mdehoon: @OceanWolf, OK, thanks for the explanation. Having a uniform API for the backends is very important for the maintainability of matplotlib. I think this MEP is a step in the right direction.

Part VI

Matplotlib AxesGrid Toolkit

The matplotlib AxesGrid toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. While the aspect parameter in matplotlib adjust the position of the single axes, AxesGrid toolkit provides a framework to adjust the position of multiple axes according to their aspects.



Note: AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of *axes_grid*. In more recent version (since svn r8226), the toolkit has divided into two separate namespace (*axes_grid1* and *axisartist*). While *axes_grid* namespace is maintained for the backward compatibility, use of *axes_grid1* and *axisartist* is recommended.

Warning: *axes_grid* and *axisartist* (but not *axes_grid1*) uses a custom Axes class (derived from the mpl's original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use *axes_grid1* to avoid this, or see how things are different in *axes_grid* and *axisartist* (LINK needed)

CHAPTER

TWENTYNINE

OVERVIEW OF AXESGRID TOOLKIT

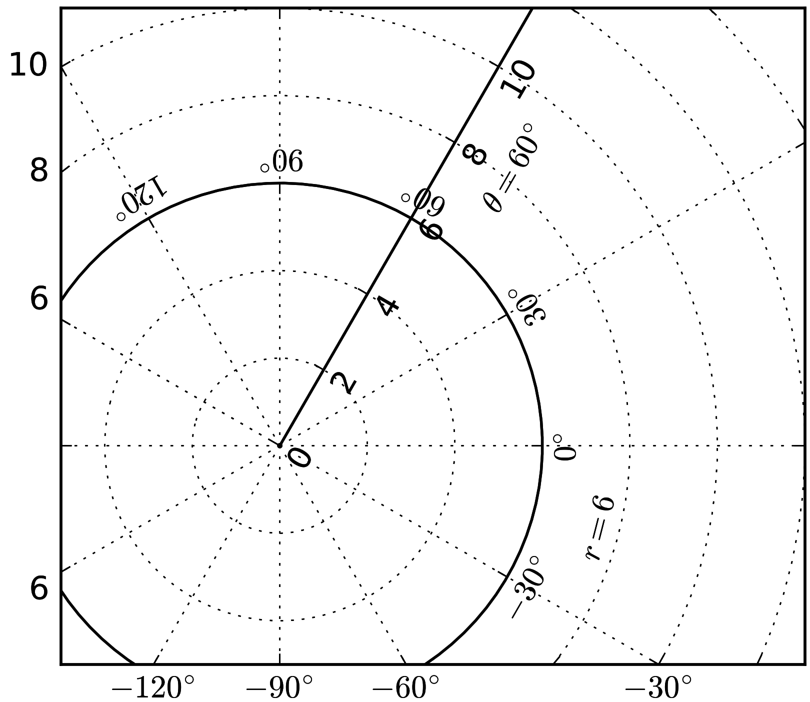
29.1 What is AxesGrid toolkit?

The matplotlib AxesGrid toolkit is a collection of helper classes, mainly to ease displaying (multiple) images in matplotlib.

Note: AxesGrid toolkit has been a part of matplotlib since v 0.99. Originally, the toolkit had a single namespace of *axes_grid*. In more recent version (since svn r8226), the toolkit has divided into two separate namespace (*axes_grid1* and *axisartist*). While *axes_grid* namespace is maintained for the backward compatibility, use of *axes_grid1* and *axisartist* is recommended.

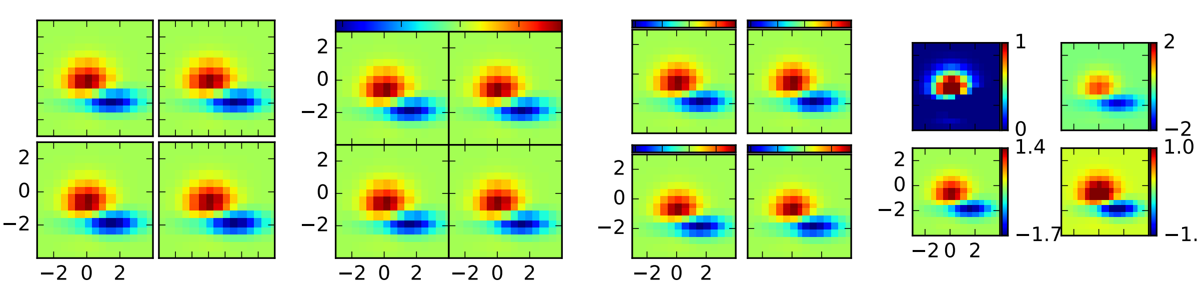
Warning: *axes_grid* and *axisartist* (but not *axes_grid1*) uses a custom Axes class (derived from the mpl's original Axes class). As a side effect, some commands (mostly tick-related) do not work. Use *axes_grid1* to avoid this, or see how things are different in *axes_grid* and *axisartist* (LINK needed)

AxesGrid toolkit has two namespaces (*axes_grid1* and *axisartist*). *axisartist* contains custom Axes class that is meant to support for curvilinear grids (e.g., the world coordinate system in astronomy). Unlike mpl's original Axes class which uses Axes.xaxis and Axes.yaxis to draw ticks, ticklines and etc., Axes in axisartist uses special artist (AxisArtist) which can handle tick, ticklines and etc. for curved coordinate systems.



Since it uses a special artists, some mpl commands that work on Axes.xaxis and Axes.yaxis may not work. See LINK for more detail.

axes_grid1 is a collection of helper classes to ease displaying (multiple) images with matplotlib. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates, which may not be ideal for displaying images that needs to have a given aspect ratio. For example, it helps you to have a colorbar whose height always matches that of the image. *ImageGrid*, *RGB Axes* and *AxesDivider* are helper classes that deals with adjusting the location of (multiple) Axes. They provides a framework to adjust the position of multiple axes at the drawing time. *ParasiteAxes* provides twinx(or twiny)-like features so that you can plot different data (e.g., different y-scale) in a same Axes. *AnchoredArtists* includes custom artists which are placed at some anchored position, like the legend.

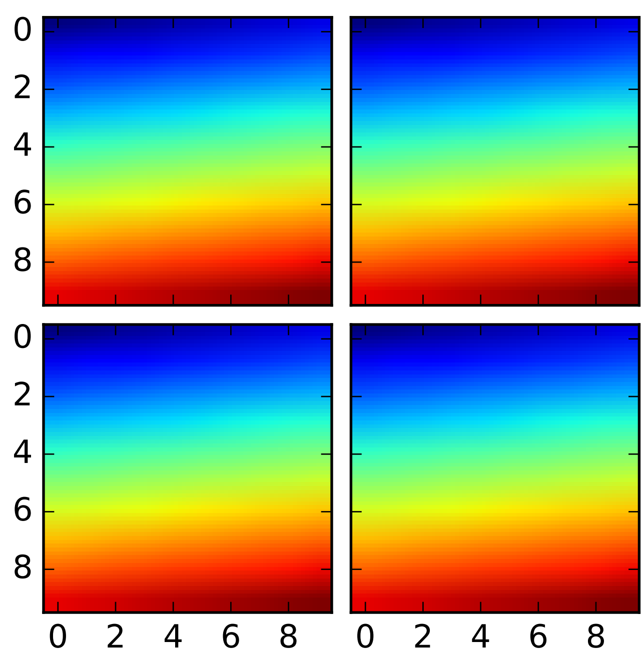


29.2 AXES_GRID1

29.2.1 ImageGrid

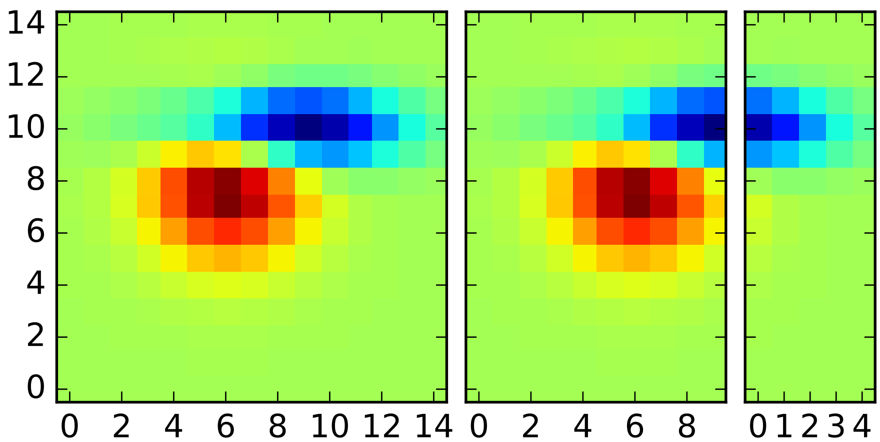
A class that creates a grid of Axes. In matplotlib, the axes location (and size) is specified in the normalized figure coordinates. This may not be ideal for images that needs to be displayed with a given aspect ratio. For example, displaying images of a same size with some fixed padding between them cannot be easily done in matplotlib. ImageGrid is used in such case.

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

im = np.arange(100)
im.shape = 10, 10

fig = plt.figure(1, (4., 4.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
                 nrows_ncols=(2, 2),  # creates 2x2 grid of axes
                 axes_pad=0.1,  # pad between axes in inch.
                 )

for i in range(4):
    grid[i].imshow(im)  # The AxesGrid object work as a list of axes.

plt.show()
```



- The position of each axes is determined at the drawing time (see *AxesDivider*), so that the size of the entire grid fits in the given rectangle (like the aspect of axes). Note that in this example, the paddings between axes are fixed even if you changes the figure size.
- axes in the same column has a same axes width (in figure coordinate), and similarly, axes in the same row has a same height. The widths (height) of the axes in the same row (column) are scaled according to their view limits (xlim or ylim).

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid


def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

F = plt.figure(1, (5.5, 3.5))
grid = ImageGrid(F, 111,  # similar to subplot(111)
                 nrows_ncols=(1, 3),
                 axes_pad=0.1,
                 add_all=True,
                 label_mode="L",
                 )

Z, extent = get_demo_image()  # demo image

im1 = Z
im2 = Z[:, :10]
im3 = Z[:, 10:]
vmin, vmax = Z.min(), Z.max()
for i, im in enumerate([im1, im2, im3]):
    ax = grid[i]
    ax.imshow(im, origin="lower", vmin=vmin,
              vmax=vmax, interpolation="nearest")

plt.draw()
plt.show()
```



- xaxis are shared among axes in a same column. Similarly, yaxis are shared among axes in a same row. Therefore, changing axis properties (view limits, tick location, etc. either by plot commands or using your mouse in interactive backends) of one axes will affect all other shared axes.

When initialized, ImageGrid creates given number (*ngrids* or *ncols* * *nrows* if *ngrids* is None) of Axes instances. A sequence-like interface is provided to access the individual Axes instances (e.g., grid[0] is the first Axes in the grid. See below for the order of axes).

AxesGrid takes following arguments,

Name	Default	Description
fig		
rect		
nrows_ncols		number of rows and cols. e.g., (2,2)
ngrids	None	number of grids. nrows x ncols if None
direction	"row"	increasing direction of axes number. [row|column]
axes_pad	0.02	pad between axes in inches
add_all	True	Add axes to figures if True
share_all	False	xaxis & yaxis of all axes are shared if True
aspect	True	aspect of axes
label_mode	"L"	location of tick labels thaw will be displayed. "1" (only the lower left axes), "L" (left most and bottom most axes), or "all".
cbar_mode	None	[None|single|each]
cbar_location	"right"	[right|top]
cbar_pad	None	pad between image axes and colorbar axes
cbar_size	"5%"	size of the colorbar
axes_class	None	

rect specifies the location of the grid. You can either specify coordinates of the rectangle to be used (e.g., (0.1, 0.1, 0.8, 0.8) as in the Axes), or the subplot-like position (e.g., “121”).

direction means the increasing direction of the axes number.

aspect By default (False), widths and heights of axes in the grid are scaled independently. If True, they are scaled according to their data limits (similar to aspect parameter in mpl).

share_all if True, xaxis and yaxis of all axes are shared.

direction direction of increasing axes number. For “row”,

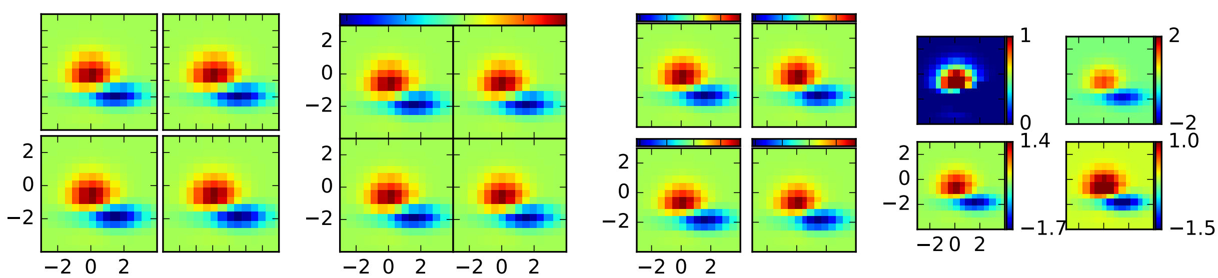
grid[0]	grid[1]
grid[2]	grid[3]

For “column”,

grid[0]	grid[2]
grid[1]	grid[3]

You can also create a colorbar (or colorbars). You can have colorbar for each axes (cbar_mode=”each”), or you can have a single colorbar for the grid (cbar_mode=”single”). The colorbar can be placed on your right, or top. The axes for each colorbar is stored as a *cbar_axes* attribute.

The examples below show what you can do with AxesGrid.



29.2.2 AxesDivider

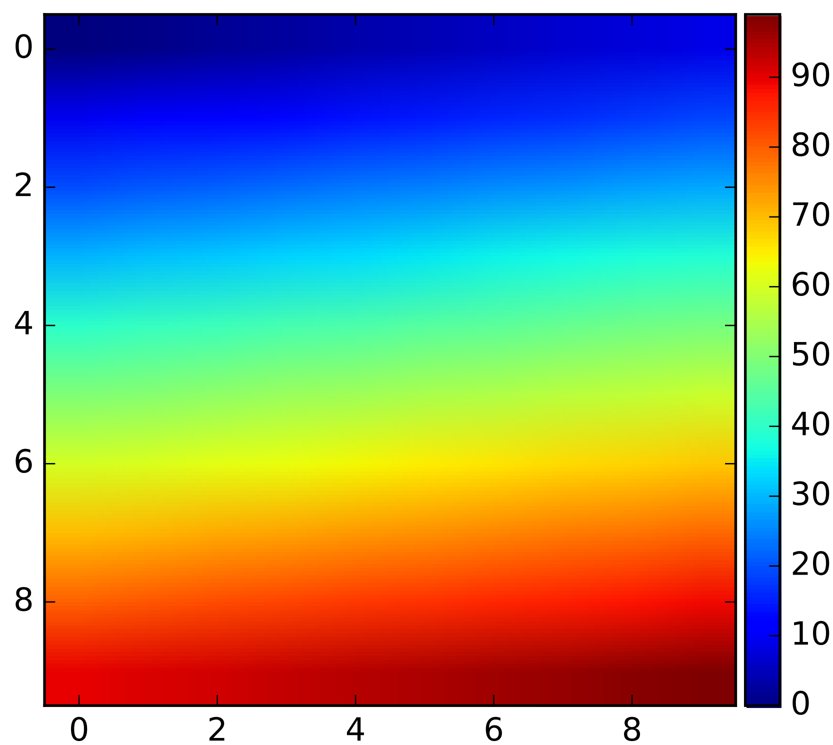
Behind the scene, the ImageGrid class and the RGBAxes class utilize the AxesDivider class, whose role is to calculate the location of the axes at drawing time. While a more about the AxesDivider is (will be) explained in (yet to be written) AxesDividerGuide, direct use of the AxesDivider class will not be necessary for most users. The axes_divider module provides a helper function make_axes_locatable, which can be useful. It takes a existing axes instance and create a divider for it.

```
ax = subplot(1,1,1)
divider = make_axes_locatable(ax)
```

make_axes_locatable returns an instance of the AxesLocator class, derived from the Locator. It provides *append_axes* method that creates a new axes on the given side of (“top”, “right”, “bottom” and “left”) of the original axes.

29.2.3 colorbar whose height (or width) in sync with the master axes

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import numpy as np

ax = plt.subplot(111)
im = ax.imshow(np.arange(100).reshape((10,10)))

# create an axes on the right side of ax. The width of cax will be 5%
# of ax and the padding between cax and ax will be fixed at 0.05 inch.
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)

plt.colorbar(im, cax=cax)
```



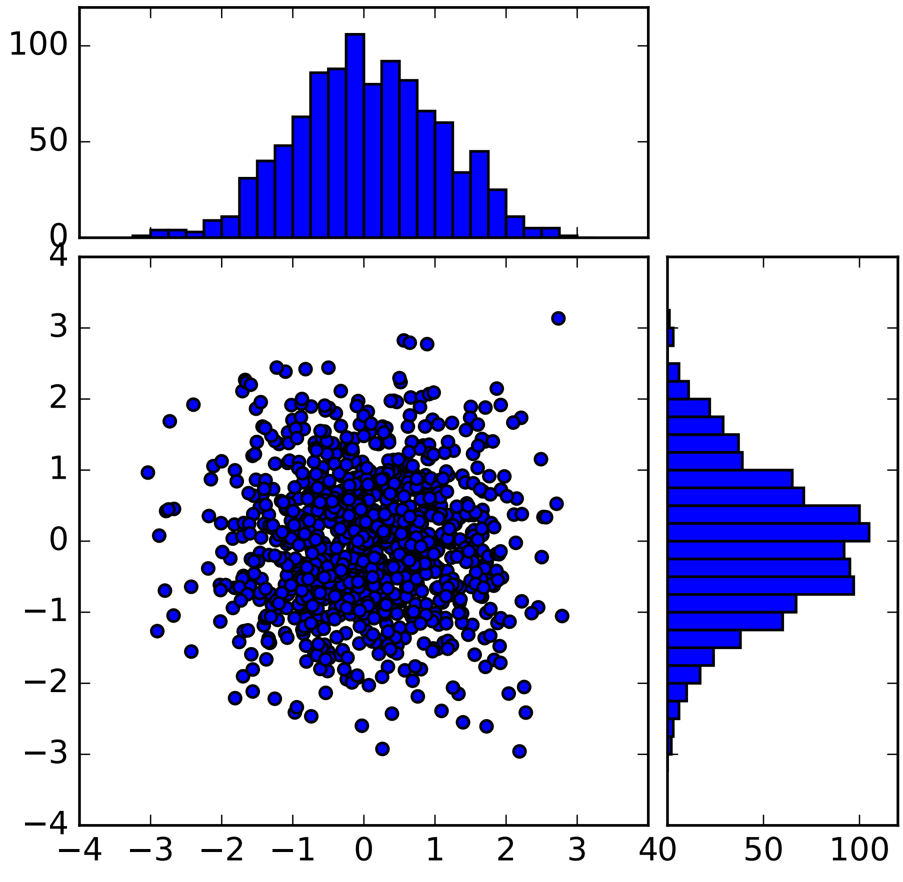
scatter_hist.py with AxesDivider

The “scatter_hist.py” example in mpl can be rewritten using *make_axes_locatable*.

```
axScatter = subplot(111)
axScatter.scatter(x, y)
axScatter.set_aspect(1.)
```

```
# create new axes on the right and on the top of the current axes.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", size=1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", size=1.2, pad=0.1, sharey=axScatter)

# the scatter plot:
# histograms
bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')
```

See the full source code below.



The scatter_hist using the AxesDivider has some advantage over the original scatter_hist.py in mpl. For example, you can set the aspect ratio of the scatter plot, even with the x-axis or y-axis is shared accordingly.

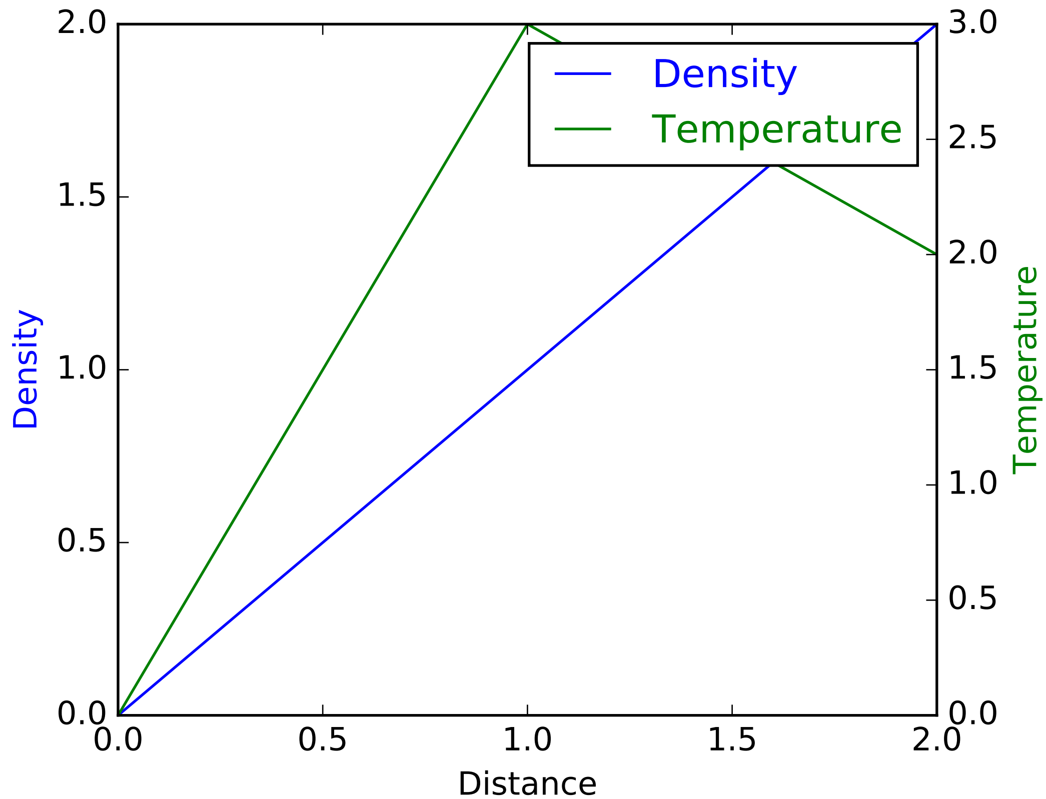
29.2.4 ParasiteAxes

The ParasiteAxes is an axes whose location is identical to its host axes. The location is adjusted in the drawing time, thus it works even if the host change its location (e.g., images).

In most cases, you first create a host axes, which provides a few method that can be used to create parasite axes. They are *twinx*, *twiny* (which are similar to twinx and twiny in the matplotlib) and *twin*. *twin* takes an arbitrary transformation that maps between the data coordinates of the host axes and the parasite axes. *draw* method of the parasite axes are never called. Instead, host axes collects artists in parasite axes and draw them as if they belong to the host axes, i.e., artists in parasite axes are merged to those of the host axes and then drawn according to their zorder. The host and parasite axes modifies some of the axes behavior. For example, color cycle for plot lines are shared between host and parasites. Also, the legend command in host, creates a legend that includes lines in the parasite axes. To create a host axes, you may use *host_suplot* or *host_axes* command.

Example 1. twinx

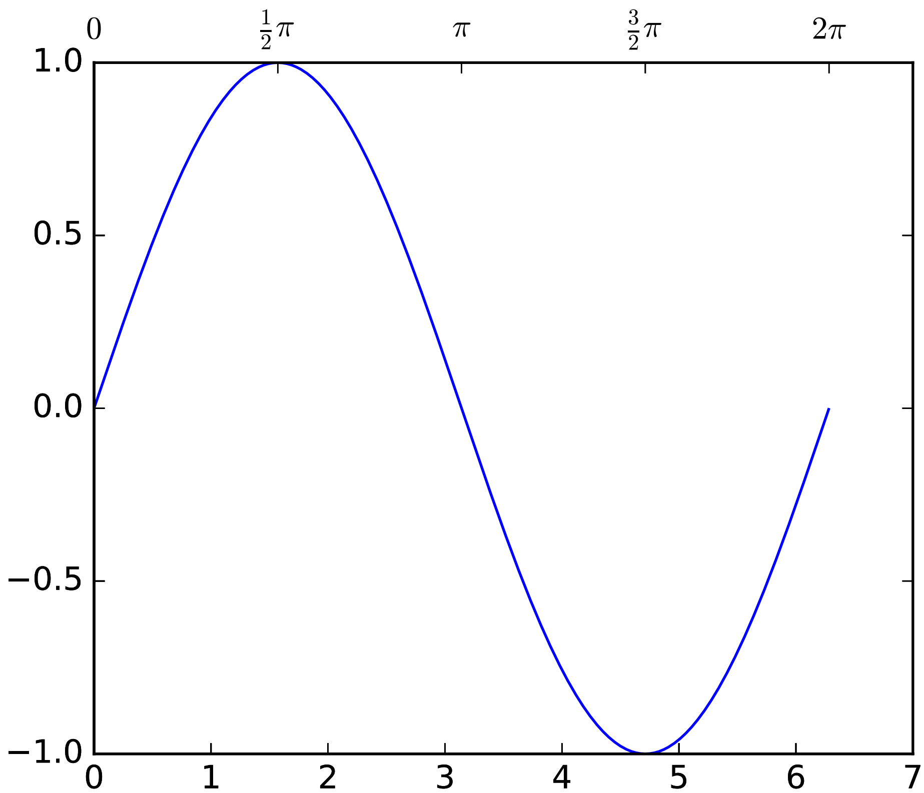
```
from mpl_toolkits.axes_grid1 import host_subplot
import matplotlib.pyplot as plt

host = host_subplot(111)

par = host.twinx()

host.set_xlabel("Distance")
host.set_ylabel("Density")
par.set_ylabel("Temperature")

p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
p2, = par.plot([0, 1, 2], [0, 3, 2], label="Temperature")

leg = plt.legend()

host.yaxis.get_label().set_color(p1.get_color())
leg.texts[0].set_color(p1.get_color())

par.yaxis.get_label().set_color(p2.get_color())
leg.texts[1].set_color(p2.get_color())

plt.show()
```



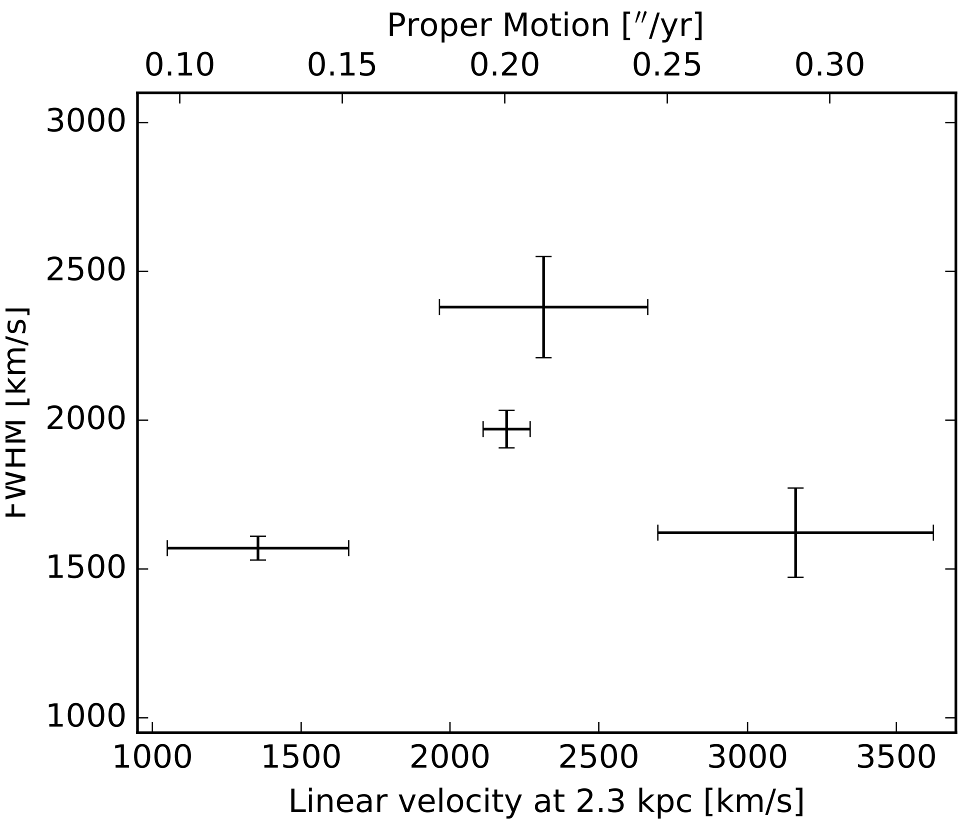
Example 2. twin

twin without a transform argument treat the parasite axes to have a same data transform as the host. This can be useful when you want the top(or right)-axis to have different tick-locations, tick-labels, or tick-formatter for bottom(or left)-axis.

```
ax2 = ax.twin() # now, ax2 is responsible for "top" axis and "right" axis
ax2.set_xticks([0., .5*np.pi, np.pi, 1.5*np.pi, 2*np.pi])
ax2.set_xticklabels(["0", r"$\frac{1}{2}\pi$",
                     r"$\pi$", r"$\frac{3}{2}\pi$", r"$2\pi$"])
```



A more sophisticated example using twin. Note that if you change the x-limit in the host axes, the x-limit of the parasite axes will change accordingly.

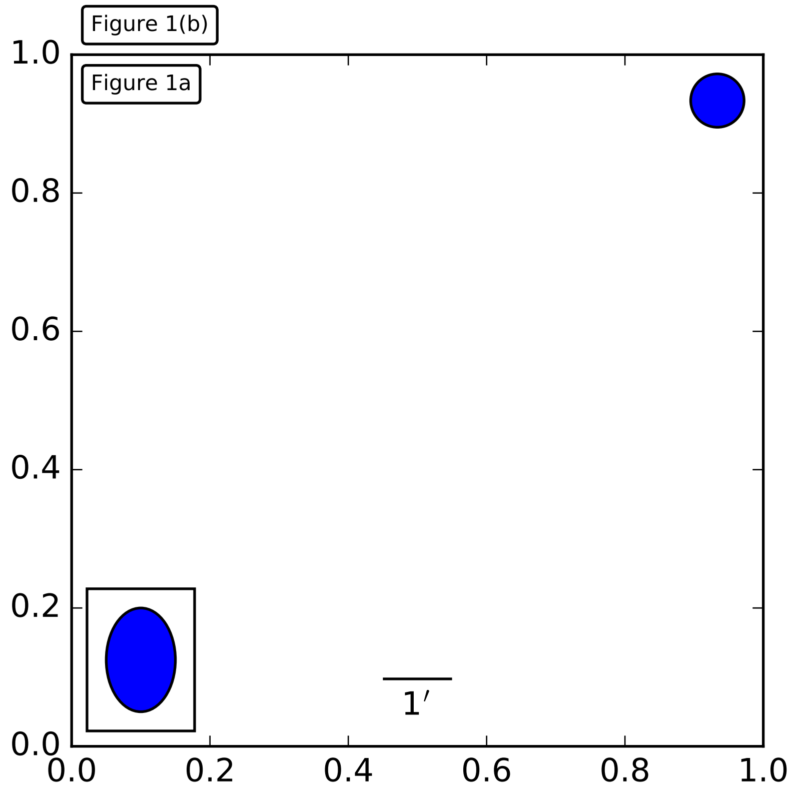


29.2.5 AnchoredArtists

It's a collection of artists whose location is anchored to the (axes) bbox, like the legend. It is derived from *OffsetBox* in mpl, and artist need to be drawn in the canvas coordinate. But, there is a limited support for an arbitrary transform. For example, the ellipse in the example below will have width and height in the data coordinate.

```
import matplotlib.pyplot as plt


def draw_text(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredText
    at = AnchoredText("Figure 1a",
                      loc=2, prop=dict(size=8), frameon=True,
                      )
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)",
                       loc=3, prop=dict(size=8), frameon=True,
                       bbox_to_anchor=(0., 1.),
                       bbox_transform=ax.transAxes
                       )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
```

```
    ax.add_artist(at2)


def draw_circle(ax):  # circle in the canvas coordinate
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea
    from matplotlib.patches import Circle
    ada = AnchoredDrawingArea(20, 20, 0, 0,
                              loc=1, pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)


def draw_ellipse(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredEllipse
    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
                         loc=3, pad=0.5, borderpad=0.4, frameon=True)

    ax.add_artist(ae)


def draw_sizebar(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    # draw a horizontal bar with length of 0.1 in Data coordinate
    # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData,
                          0.1,
                          r"1$^{\prime}$",
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)
    ax.add_artist(asb)


if 1:
    ax = plt.gca()
    ax.set_aspect(1.)

    draw_text(ax)
    draw_circle(ax)
    draw_ellipse(ax)
    draw_sizebar(ax)

    plt.show()
```



29.2.6 InsetLocator

`mpl_toolkits.axes_grid.inset_locator` provides helper classes and functions to place your (inset) axes at the anchored position of the parent axes, similarly to AnchoredArtist.

Using `mpl_toolkits.axes_grid.inset_locator.inset_axes()`, you can have inset axes whose size is either fixed, or a fixed proportion of the parent axes. For example,:

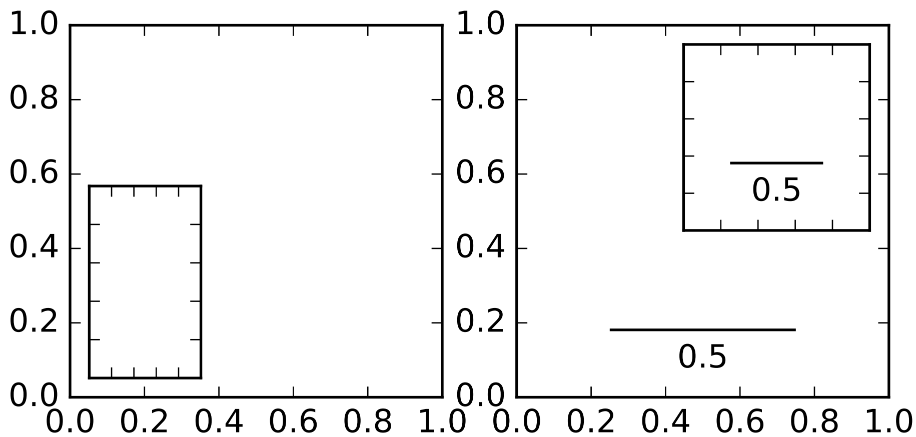
```
inset_axes = inset_axes(parent_axes,
                        width="30%", # width = 30% of parent_bbox
                        height=1., # height : 1 inch
                        loc=3)
```

creates an inset axes whose width is 30% of the parent axes and whose height is fixed at 1 inch.

You may creates your inset whose size is determined so that the data scale of the inset axes to be that of the parent axes multiplied by some factor. For example,

```
inset_axes = zoomed_inset_axes(ax,
                               0.5, # zoom = 0.5
                               loc=1)
```

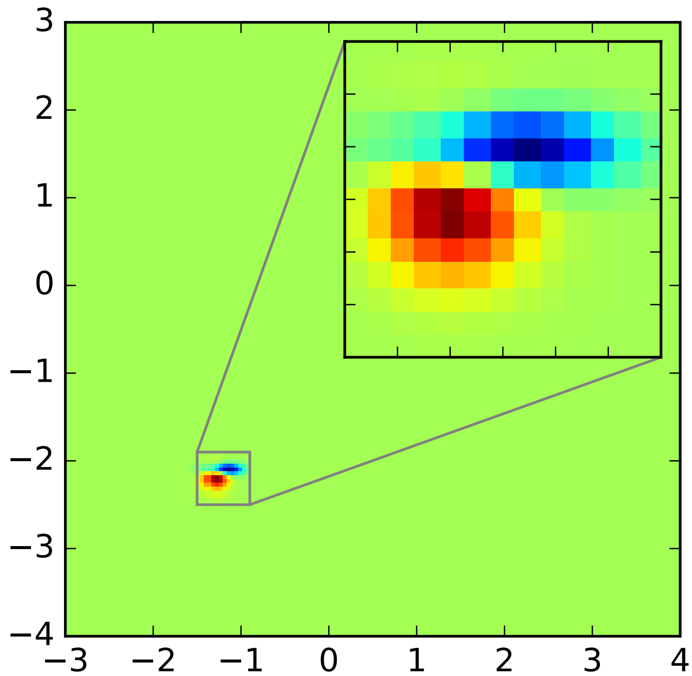
creates an inset axes whose data scale is half of the parent axes. Here is complete examples.



For example, `zoomed_inset_axes()` can be used when you want the inset represents the zoom-up of the small portion in the parent axes. And `mpl_toolkits/axes_grid/inset_locator` provides a helper function `mark_inset()` to mark the location of the area represented by the inset axes.

```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset

import numpy as np


def get_demo_image():
    from matplotlib.cbook import get_sample_data
    import numpy as np
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

fig, ax = plt.subplots(figsize=[5, 4])

# prepare the demo image
Z, extent = get_demo_image()
Z2 = np.zeros([150, 150], dtype="d")
ny, nx = Z.shape
Z2[30:30 + ny, 30:30 + nx] = Z

# extent = [-3, 4, -4, 3]
ax.imshow(Z2, extent=extent, interpolation="nearest",
          origin="lower")

axins = zoomed_inset_axes(ax, 6, loc=1)  # zoom = 6
axins.imshow(Z2, extent=extent, interpolation="nearest",
             origin="lower")
```

```
# sub region of the original image
x1, x2, y1, y2 = -1.5, -0.9, -2.5, -1.9
axins.set_xlim(x1, x2)
axins.set_ylim(y1, y2)

plt.xticks(visible=False)
plt.yticks(visible=False)

# draw a bbox of the region of the inset axes in the parent axes and
# connecting lines between the bbox and the inset axes area
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")

plt.draw()
plt.show()
```

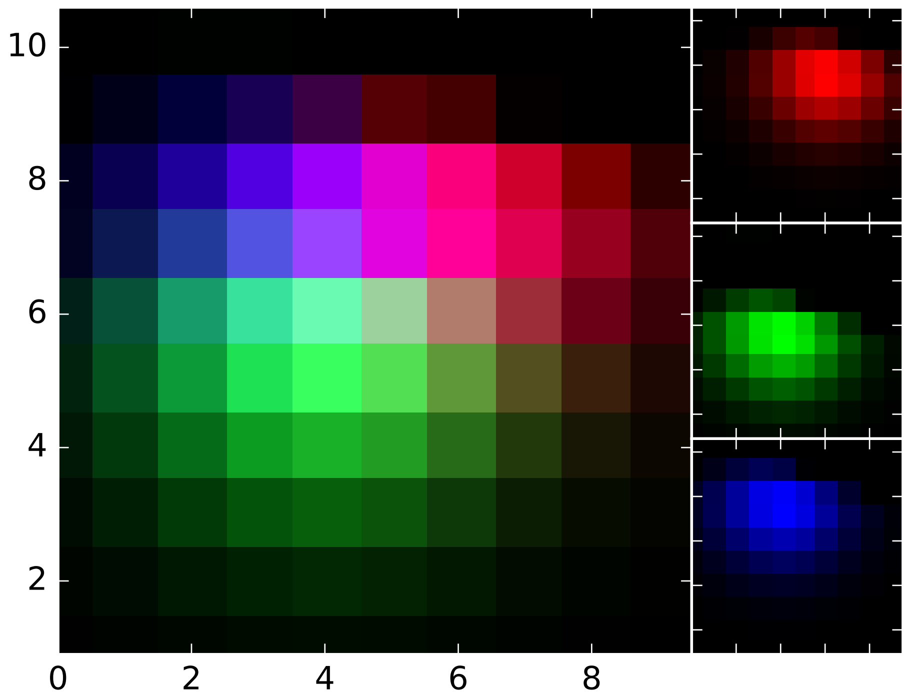


RGB Axes

RGBAxes is a helper class to conveniently show RGB composite images. Like ImageGrid, the location of axes are adjusted so that the area occupied by them fits in a given rectangle. Also, the xaxis and yaxis of each axes are shared.

```
from mpl_toolkits.axes_grid1.axes_rgb import RGBAxes

fig = plt.figure(1)
ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8])
```

```
r, g, b = get_rgb() # r,g,b are 2-d images
ax.imshow_rgb(r, g, b,
              origin="lower", interpolation="nearest")
```



29.3 AXISARTIST

29.3.1 AxisArtist

AxisArtist module provides a custom (and very experimental) Axes class, where each axis (left, right, top and bottom) have a separate artist associated which is responsible to draw axis-line, ticks, ticklabels, label. Also, you can create your own axis, which can pass through a fixed position in the axes coordinate, or a fixed position in the data coordinate (i.e., the axis floats around when viewlimit changes).

The axes class, by default, have its xaxis and yaxis invisible, and has 4 additional artists which are responsible to draw axis in “left”,”right”,”bottom” and “top”. They are accessed as ax.axis[”left”], ax.axis[”right”], and so on, i.e., ax.axis is a dictionary that contains artists (note that ax.axis is still a callable methods and it behaves as an original Axes.axis method in mpl).

To create an axes,

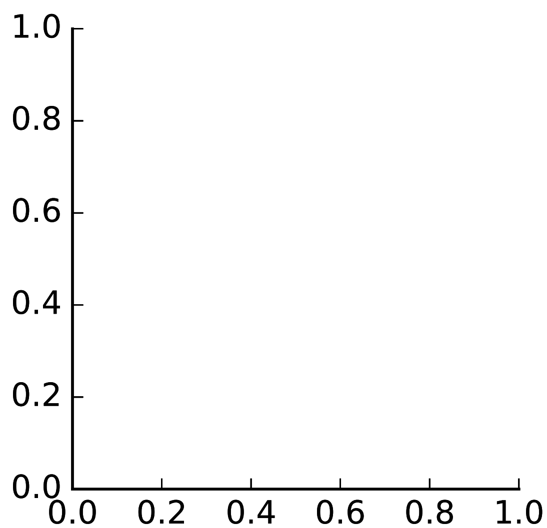
```
import mpl_toolkits.axisartist as AA
fig = plt.figure(1)
ax = AA.Axes(fig, [0.1, 0.1, 0.8, 0.8])
fig.add_axes(ax)
```

or to create a subplot

```
ax = AA.Subplot(fig, 111)
fig.add_subplot(ax)
```

For example, you can hide the right, and top axis by

```
ax.axis["right"].set_visible(False)
ax.axis["top"].set_visible(False)
```

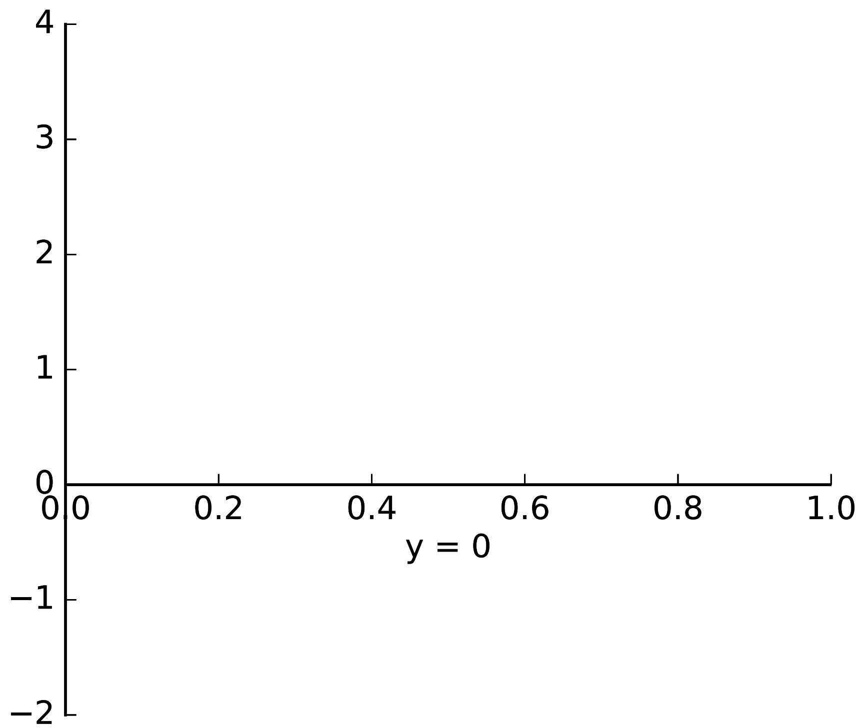


It is also possible to add an extra axis. For example, you may have an horizontal axis at y=0 (in data coordinate).

```
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0)
```

```
import matplotlib.pyplot as plt
import mpl_toolkits.axisartist as AA

fig = plt.figure(1)
fig.subplots_adjust(right=0.85)
ax = AA.Subplot(fig, 1, 1, 1)
fig.add_subplot(ax)

# make some axis invisible
ax.axis["bottom", "top", "right"].set_visible(False)

# make an new axis along the first axis axis (x-axis) which pass
# throught y=0.
```

```
ax.axis["y=0"] = ax.new_floating_axis(nth_coord=0, value=0,
                                      axis_direction="bottom")
ax.axis["y=0"].toggle(all=True)
ax.axis["y=0"].label.set_text("y = 0")

ax.set_ylim(-2, 4)

plt.show()
```



Or a fixed axis with some offset

```
# make new (right-side) yaxis, but wth some offset
ax.axis["right2"] = ax.new_fixed_axis(loc="right",
                                      offset=(20, 0))
```

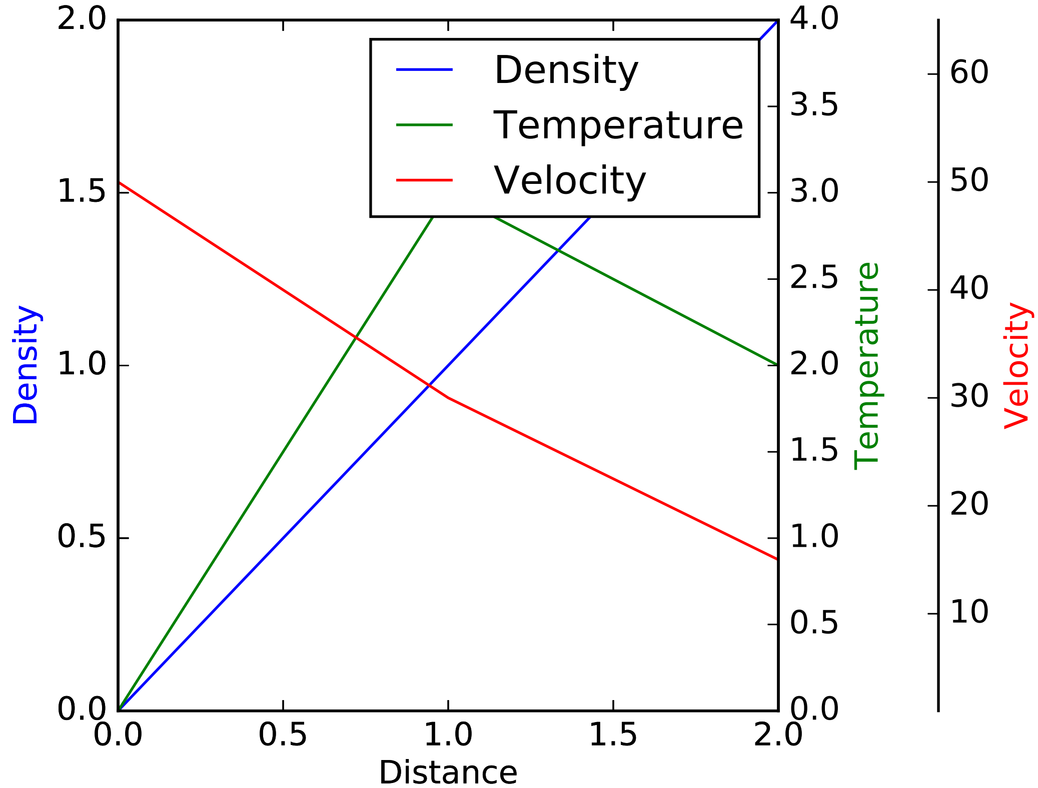
AxisArtist with ParasiteAxes

Most commands in the axes_grid1 toolkit can take a axes_class keyword argument, and the commands creates an axes of the given class. For example, to create a host subplot with axisartist.Axes,

```
import mpl_tookits.axisartist as AA
from mpl_toolkits.axes_grid1 import host_subplot
```

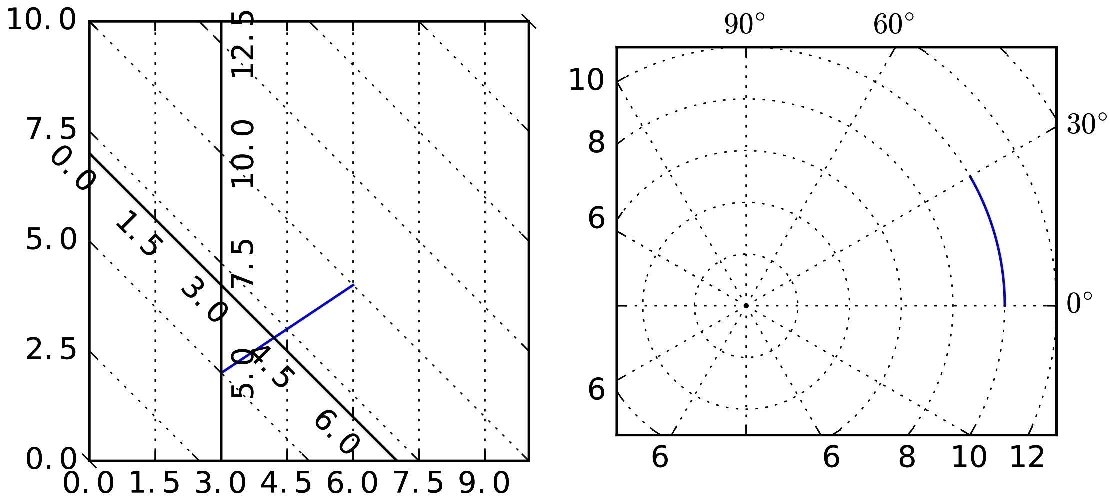
```
host = host_subplot(111, axes_class=AA.Axes)
```

Here is an example that uses parasiteAxes.



29.3.2 Curvilinear Grid

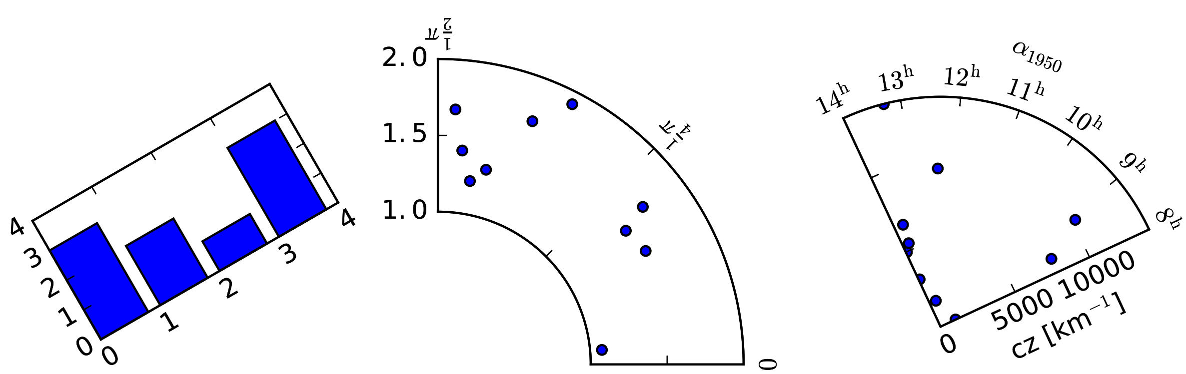
The motivation behind the AxisArtist module is to support curvilinear grid and ticks.



See *AXISARTIST namespace* for more details.

29.3.3 Floating Axes

This also support a Floating Axes whose outer axis are defined as floating axis.



CHAPTER

THIRTY

THE MATPLOTLIB AXESGRID TOOLKIT USER'S GUIDE

Release 1.5.3

Date December 05, 2016

30.1 AxesDivider

The axes_divider module provide helper classes to adjust the axes positions of set of images in the drawing time.

- *axes_size* provides a classes of units that the size of each axes will be determined. For example, you can specify a fixed size
- `Divider` this is the class that is used calculates the axes position. It divides the given rectangular area into several areas. You initialize the divider by setting the horizontal and vertical list of sizes that the division will be based on. You then use the new_locator method, whose return value is a callable object that can be used to set the axes_locator of the axes.

You first initialize the divider by specifying its grids, i.e., horizontal and vertical.

for example,:

```
rect = [0.2, 0.2, 0.6, 0.6]
horiz=[h0, h1, h2, h3]
vert=[v0, v1, v2]
divider = Divider(fig, rect, horiz, vert)
```

where, rect is a bounds of the box that will be divided and h0,..h3, v0,..v2 need to be an instance of classes in the *axes_size*. They have *get_size* method that returns a tuple of two floats. The first float is the relative size, and the second float is the absolute size. Consider a following grid.

v0			
v1			
h0,v2	h1	h2	h3

- v0 => 0, 2
- v1 => 2, 0
- v2 => 3, 0

The height of the bottom row is always 2 (axes_divider internally assumes that the unit is inch). The first and the second rows with height ratio of 2:3. For example, if the total height of the grid 6, then the first and second row will each occupy 2/(2+3) and 3/(2+3) of (6-1) inches. The widths of columns (horiz) will be similarly determined. When aspect ratio is set, the total height (or width) will be adjusted accordingly.

The *mpl_toolkits.axes_grid.axes_size* contains several classes that can be used to set the horizontal and vertical configurations. For example, for the vertical configuration above will be:

```
from mpl_toolkits.axes_grid.axes_size import Fixed, Scaled
vert = [Fixed(2), Scaled(2), Scaled(3)]
```

After you set up the divider object, then you create a locator instance which will be given to the axes.:

```
locator = divider.new_locator(nx=0, ny=1)
ax.set_axes_locator(locator)
```

The return value of the new_locator method is a instance of the AxesLocator class. It is a callable object that returns the location and size of the cell at the first column and the second row. You may create a locator that spans over multiple cells.:

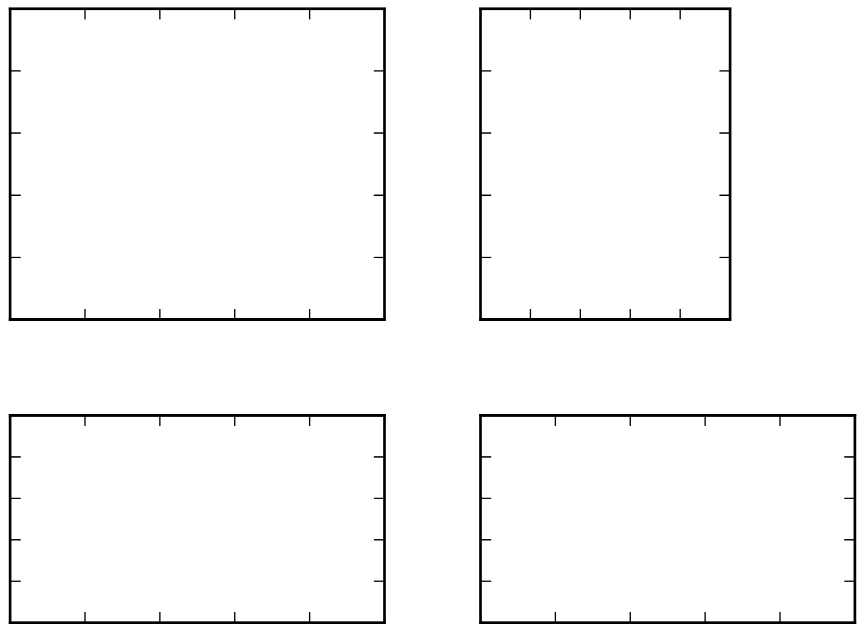
```
locator = divider.new_locator(nx=0, nx=2, ny=1)
```

The above locator, when called, will return the position and size of the cells spanning the first and second column and the first row. You may consider it as [0:2, 1].

See the example,

```
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt

fig1 = plt.figure(1, (5.5, 4.))

# the rect parameter will be ignore as we will set axes_locator
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d"%i) for i in range(4)]

horiz = [Size.Scaled(1.5), Size.Fixed(.5), Size.Scaled(1.),
         Size.Scaled(.5)]

vert = [Size.Scaled(1.), Size.Fixed(.5), Size.Scaled(1.5)]

# divide the axes rectangle into grid whose size is specified by horiz * vert
divider = Divider(fig1, rect, horiz, vert, aspect=False)

ax[0].set_axes_locator(divider.new_locator(nx=0, ny=0))
ax[1].set_axes_locator(divider.new_locator(nx=0, ny=2))
ax[2].set_axes_locator(divider.new_locator(nx=2, ny=2))
ax[3].set_axes_locator(divider.new_locator(nx=2, nx1=4, ny=0))

for ax1 in ax:
    plt.setp(ax1.get_xticklabels()+ax1.get_yticklabels(),
             visible=False)
```

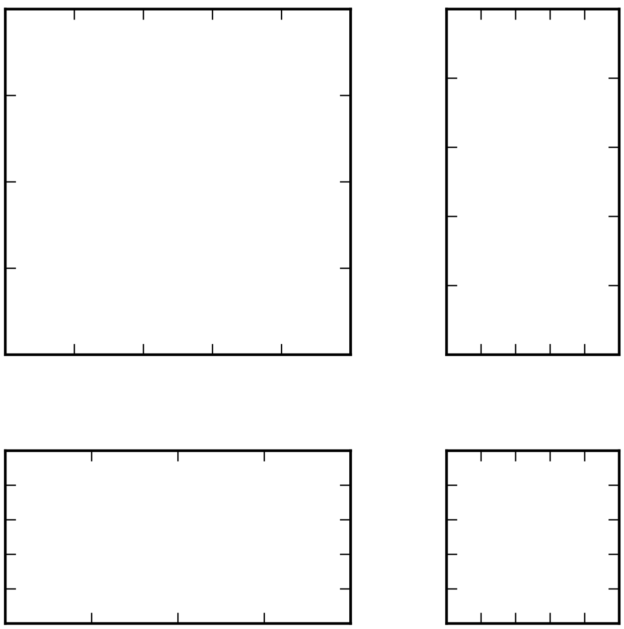
```
plt.draw()
plt.show()
```



You can adjust the size of the each axes according to their x or y data limits (AxesX and AxesY), similar to the axes aspect parameter.

```
import mpl_toolkits.axes_grid.axes_size as Size
from mpl_toolkits.axes_grid import Divider
import matplotlib.pyplot as plt


fig1 = plt.figure(1, (5.5, 4))

# the rect parameter will be ignore as we will set axes_locator
rect = (0.1, 0.1, 0.8, 0.8)
ax = [fig1.add_axes(rect, label="%d"%i) for i in range(4)]


horiz = [Size.AxesX(ax[0]), Size.Fixed(.5), Size.AxesX(ax[1])]
vert = [Size.AxesY(ax[0]), Size.Fixed(.5), Size.AxesY(ax[2])]

# divide the axes rectangle into grid whose size is specified by horiz * vert
divider = Divider(fig1, rect, horiz, vert, aspect=False)


ax[0].set_axes_locator(divider.new_locator(nx=0, ny=0))
ax[1].set_axes_locator(divider.new_locator(nx=2, ny=0))
```

```
ax[2].set_axes_locator(divider.new_locator(nx=0, ny=2))
ax[3].set_axes_locator(divider.new_locator(nx=2, ny=2))

ax[0].set_xlim(0, 2)
ax[1].set_xlim(0, 1)

ax[0].set_ylim(0, 1)
ax[2].set_ylim(0, 2)

divider.set_aspect(1.)

for ax1 in ax:
    plt.setp(ax1.get_xticklabels()+ax1.get_yticklabels(),
             visible=False)

plt.draw()
plt.show()
```



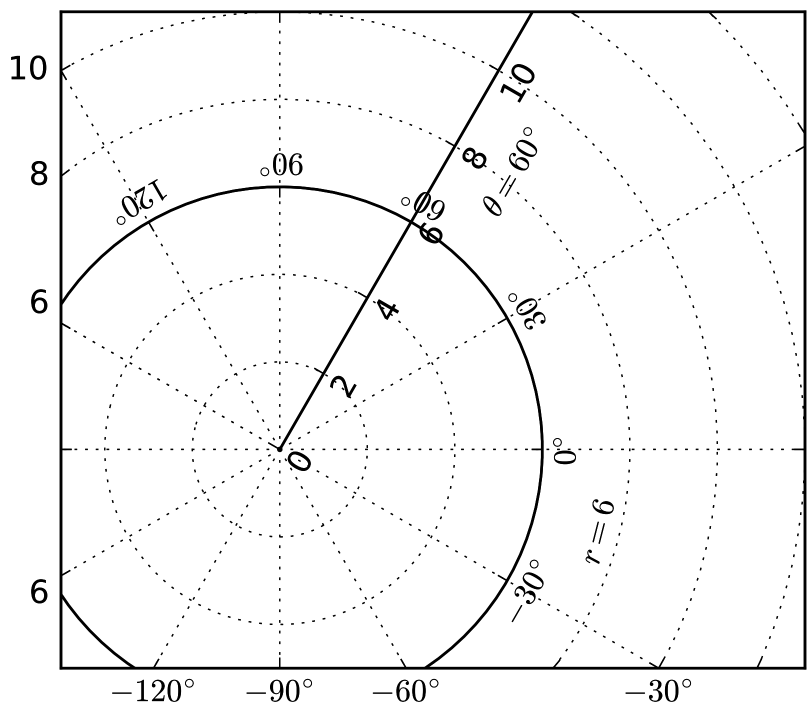
30.2 AXISARTIST namespace

The AxisArtist namespace includes a derived Axes implementation. The biggest difference is that the artists responsible to draw axis line, ticks, ticklabel and axis labels are separated out from the mpl's Axis class, which are much more than artists in the original mpl. This change was strongly motivated to support curvilinear grid. Here are a few things that mpl_tootlkits.axisartist.Axes is different from original Axes from mpl.

- Axis elements (axis line(spine), ticks, ticklabel and axis labels) are drawn by a AxisArtist instance. Unlike Axis, left, right, top and bottom axis are drawn by separate artists. And each of them may have different tick location and different tick labels.
- gridlines are drawn by a Gridlines instance. The change was motivated that in curvilinear coordinate, a gridline may not cross axis-lines (i.e., no associated ticks). In the original Axes class, gridlines are tied to ticks.
- ticklines can be rotated if necessary (i.e, along the gridlines)

In summary, all these changes was to support

- a curvilinear grid.
- a floating axis



mpl_toolkits.axisartist.Axes class defines a *axis* attribute, which is a dictionary of AxisArtist instances. By default, the dictionary has 4 AxisArtist instances, responsible for drawing of left, right, bottom and top axis.

xaxis and yaxis attributes are still available, however they are set to not visible. As separate artists are used for rendering axis, some axis-related method in mpl may have no effect. In addition to AxisArtist instances, the mpl_toolkits.axisartist.Axes will have *gridlines* attribute (Gridlines), which obviously draws grid lines.

In both AxisArtist and Gridlines, the calculation of tick and grid location is delegated to an instance of

GridHelper class. mpl_toolkits.axisartist.Axes class uses GridHelperRectlinear as a grid helper. The GridHelperRectlinear class is a wrapper around the *xaxis* and *yaxis* of mpl's original Axes, and it was meant to work as the way how mpl's original axes works. For example, tick location changes using set_ticks method and etc. should work as expected. But change in artist properties (e.g., color) will not work in general, although some effort has been made so that some often-change attributes (color, etc.) are respected.

30.2.1 AxisArtist

AxisArtist can be considered as a container artist with following attributes which will draw ticks, labels, etc.

- line
- major_ticks, major_ticklabels
- minor_ticks, minor_ticklabels
- offsetText
- label

line

Derived from Line2d class. Responsible for drawing a spinal(?) line.

major_ticks, minor_ticks

Derived from Line2d class. Note that ticks are markers.

major_ticklabels, minor_ticklabels

Derived from Text. Note that it is not a list of Text artist, but a single artist (similar to a collection).

axislabel

Derived from Text.

Default AxisArtists

By default, following for axis artists are defined.:

```
ax.axis["left"], ax.axis["bottom"], ax.axis["right"], ax.axis["top"]
```

The ticklabels and axislabel of the top and the right axis are set to not visible.

For example, if you want to change the color attributes of major_ticklabels of the bottom x-axis

```
ax.axis["bottom"].major_ticklabels.set_color("b")
```

Similarly, to make ticklabels invisible

```
ax.axis["bottom"].major_ticklabels.set_visible(False)
```

AxisAritst provides a helper method to control the visibility of ticks, ticklabels, and label. To make ticklabel invisible,

```
ax.axis["bottom"].toggle(ticklabels=False)
```

To make all of ticks, ticklabels, and (axis) label invisible

```
ax.axis["bottom"].toggle(all=False)
```

To turn all off but ticks on

```
ax.axis["bottom"].toggle(all=False, ticks=True)
```

To turn all on but (axis) label off

```
ax.axis["bottom"].toggle(all=True, label=False))
```

ax.axis's __getitem__ method can take multiple axis names. For example, to turn ticklabels of "top" and "right" axis on,

```
ax.axis["top","right"].toggle(ticklabels=True))
```

Note that 'ax.axis["top","right"]' returns a simple proxy object that translate above code to something like below.

```
for n in ["top","right"]:
  ax.axis[n].toggle(ticklabels=True))
```

So, any return values in the for loop are ignored. And you should not use it anything more than a simple method.

Like the list indexing ":" means all items, i.e.,

```
ax.axis[:].major_ticks.set_color("r")
```

changes tick color in all axis.

30.2.2 HowTo

1. Changing tick locations and label.

   Same as the original mpl's axes.:

```
ax.set_xticks([1,2,3])
```

2. Changing axis properties like color, etc.

   Change the properties of appropriate artists. For example, to change the color of the ticklabels:

```
ax.axis["left"].major_ticklabels.set_color("r")
```

3. To change the attributes of multiple axis:

```
ax.axis["left","bottom"].major_ticklabels.set_color("r")
```

   or to change the attributes of all axis:

```
ax.axis[:].major_ticklabels.set_color("r")
```

4. **To change the tick size (length), you need to use** axis.major_ticks.set_ticksize method. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use axis.major_ticks.set_tick_out method.

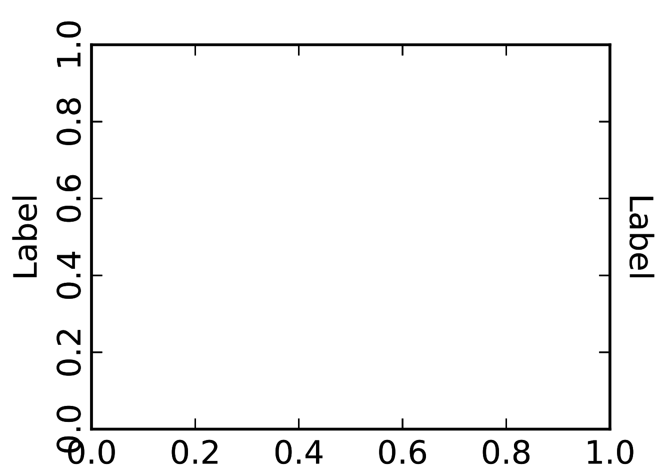
   To change the pad between ticks and ticklabels, use axis.major_ticklabels.set_pad method.

   To change the pad between ticklabels and axis label, axis.label.set_pad method.

30.2.3 Rotation and Alignment of TickLabels

This is also quite different from the original mpl and can be confusing. When you want to rotate the ticklabels, first consider using "set_axis_direction" method.

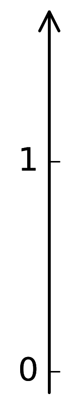
```
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
ax1.axis["right"].label.set_axis_direction("left")
```



The parameter for set_axis_direction is one of ["left", "right", "bottom", "top"].

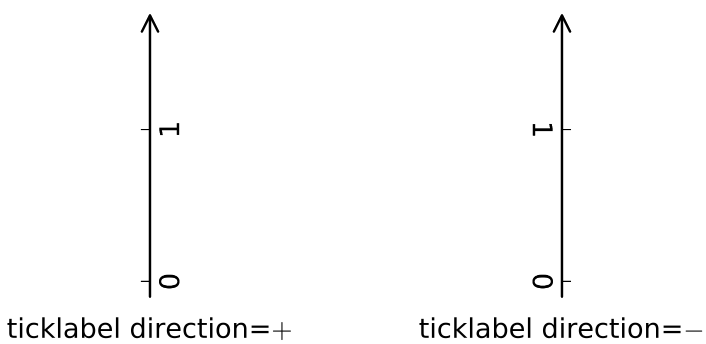
You must understand some underlying concept of directions.

1. There is a reference direction which is defined as the direction of the axis line with increasing coordinate. For example, the reference direction of the left x-axis is from bottom to top.



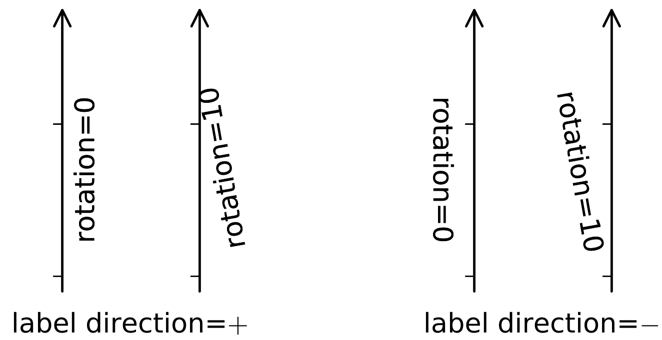
The direction, text angle, and alignments of the ticks, ticklabels and axis-label is determined with respect to the reference direction

2. *ticklabel_direction* is either the right-hand side (+) of the reference direction or the left-hand side (-).



3. same for the *label_direction*

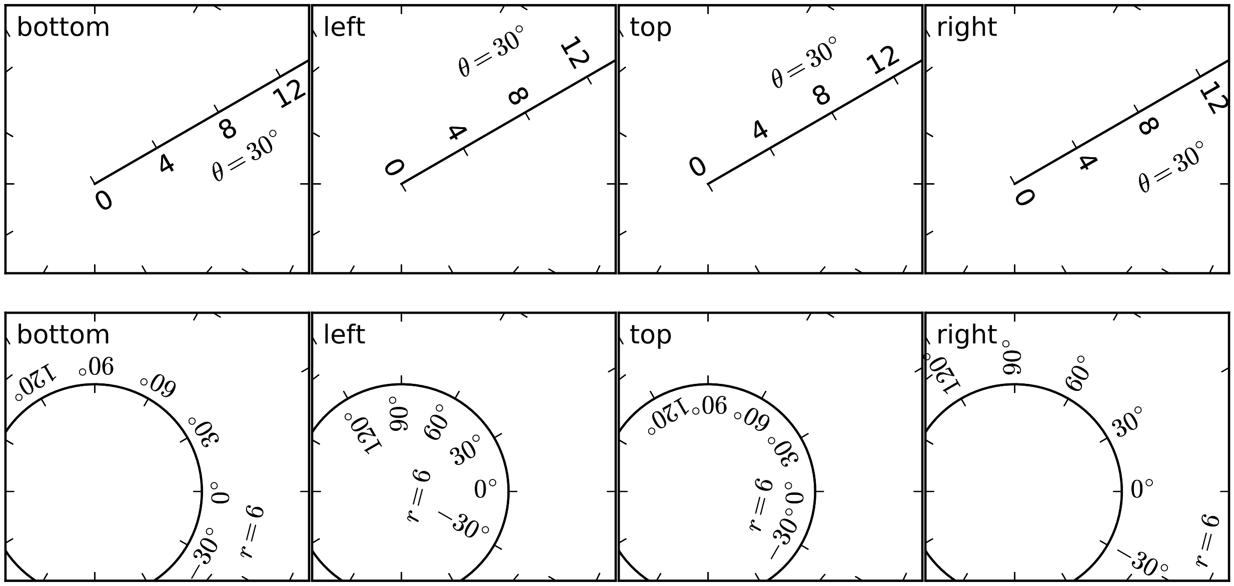
4. ticks are by default drawn toward the opposite direction of the ticklabels.
5. text rotation of ticklabels and label is determined in reference to the *ticklabel_direction* or *label_direction*, respectively. The rotation of ticklabels and label is anchored.



On the other hand, there is a concept of "axis_direction". This is a default setting of above properties for each, "bottom", "left", "top", and "right" axis.

?	?	left	bottom	right	top
axislabel	direction	'-'	'+'	'+'	'-'
axislabel	rotation	180	0	0	180
axislabel	va	center	top	center	bottom
axislabel	ha	right	center	right	center
ticklabel	direction	'-'	'+'	'+'	'-'
ticklabels	rotation	90	0	-90	180
ticklabel	ha	right	center	right	center
ticklabel	va	center	baseline	center	baseline

And, 'set_axis_direction("top")' means to adjust the text rotation etc, for settings suitable for "top" axis. The concept of axis direction can be more clear with curved axis.



The axis_direction can be adjusted in the AxisArtist level, or in the level of its child arists, i.e., ticks, ticklabels, and axis-label.

```
ax1.axis["left"].set_axis_direction("top")
```

changes axis_direction of all the associated artist with the “left” axis, while

```
ax1.axis["left"].major_ticklabels.set_axis_direction("top")
```

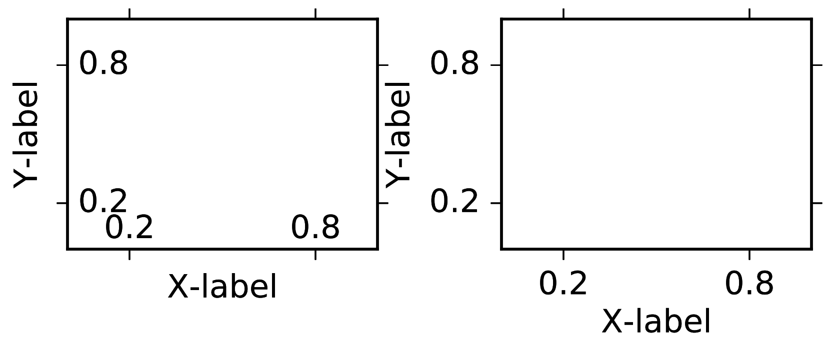
changes the axis_direction of only the major_ticklabels. Note that set_axis_direction in the AxisArtist level changes the ticklabel_direction and label_direction, while changing the axis_direction of ticks, ticklabels, and axis-label does not affect them.

If you want to make ticks outward and ticklabels inside the axes, use invert_ticklabel_direction method.

```
ax.axis[:].invert_ticklabel_direction()
```

A related method is “set_tick_out”. It makes ticks outward (as a matter of fact, it makes ticks toward the opposite direction of the default direction).

```
ax.axis[:].major_ticks.set_tick_out(True)
```

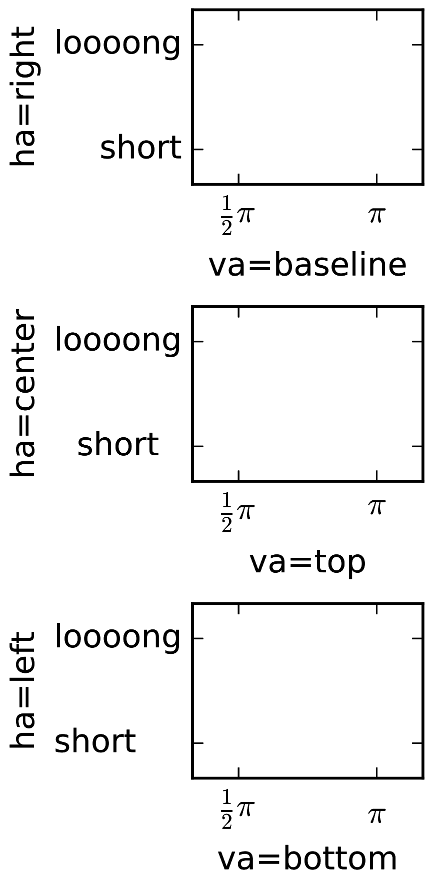


So, in summary,

- **AxisArtist's methods**
    - set_axis_direction : "left", "right", "bottom", or "top"
    - set_ticklabel_direction : "+" or "-"
    - set_axislabel_direction : "+" or "-"
    - invert_ticklabel_direction
- **Ticks' methods (major_ticks and minor_ticks)**
    - set_tick_out : True or False
    - set_ticksize : size in points
- **TickLabels' methods (major_ticklabels and minor_ticklabels)**
    - set_axis_direction : "left", "right", "bottom", or "top"
    - set_rotation : angle with respect to the reference direction
    - set_ha and set_va : see below
- **AxisLabels' methods (label)**
    - set_axis_direction : "left", "right", "bottom", or "top"
    - set_rotation : angle with respect to the reference direction
    - set_ha and set_va

Adjusting ticklabels alignment

Alignment of TickLabels are treated specially. See below



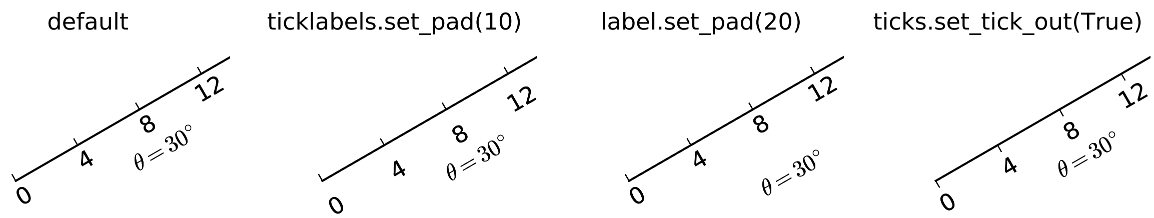
Adjusting pad

To change the pad between ticks and ticklabels

```
ax.axis["left"].major_ticklabels.set_pad(10)
```

Or ticklabels and axis-label

```
ax.axis["left"].label.set_pad(10)
```



30.2.4 GridHelper

To actually define a curvilinear coordinate, you have to use your own grid helper. A generalised version of grid helper class is supplied and this class should suffice in most of cases. A user may provide two functions which defines a transformation (and its inverse pair) from the curved coordinate to (rectilinear) image coordinate. Note that while ticks and grids are drawn for curved coordinate, the data transform of the axes itself (ax.transData) is still rectilinear (image) coordinate.

```
from  mpl_toolkits.axisartist.grid_helper_curvelinear \
     import GridHelperCurveLinear
from mpl_toolkits.axisartist import Subplot

# from curved coordinate to rectlinear coordinate.
def tr(x, y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y-x

# from rectlinear coordinate to curved coordinate.
def inv_tr(x,y):
    x, y = np.asarray(x), np.asarray(y)
    return x, y+x


grid_helper = GridHelperCurveLinear((tr, inv_tr))

ax1 = Subplot(fig, 1, 1, 1, grid_helper=grid_helper)

fig.add_subplot(ax1)
```

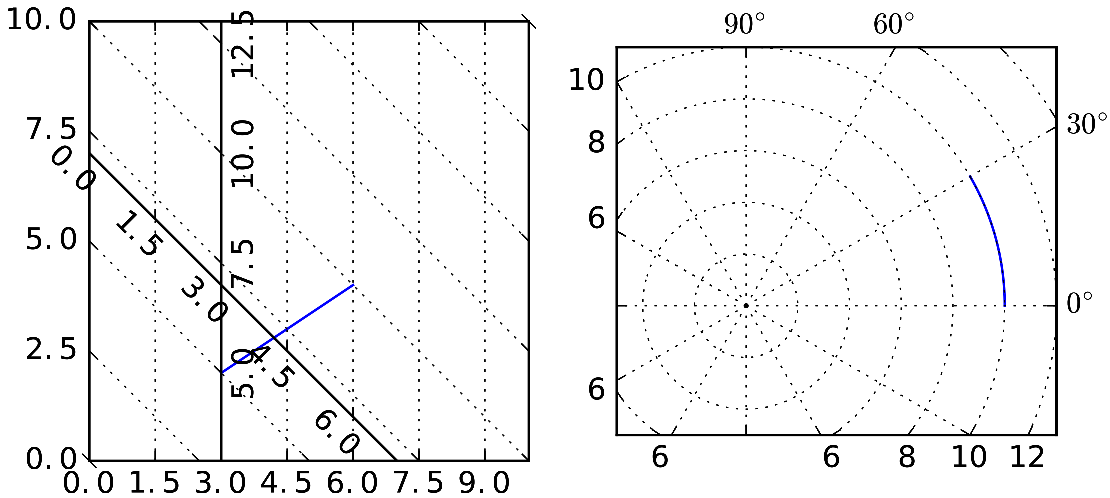
You may use matplotlib's Transform instance instead (but a inverse transformation must be defined). Often, coordinate range in a curved coordinate system may have a limited range, or may have cycles. In those cases, a more customized version of grid helper is required.

```
import  mpl_toolkits.axisartist.angle_helper as angle_helper

# PolarAxes.PolarTransform takes radian. However, we want our coordinate
# system in degree
```

```
tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()


# extreme finder :  find a range of coordinate.
# 20, 20 : number of sampling points along x, y direction
# The first coordinate (longitude, but theta in polar)
#   has a cycle of 360 degree.
# The second coordinate (latitude, but radius in polar)  has a minimum of 0
extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
                                                 lon_cycle = 360,
                                                 lat_cycle = None,
                                                 lon_minmax = None,
                                                 lat_minmax = (0, np.inf),
                                                 )

# Find a grid values appropriate for the coordinate (degree,
# minute, second). The argument is a approximate number of grids.
grid_locator1 = angle_helper.LocatorDMS(12)

# And also uses an appropriate formatter.  Note that,the
# acceptable Locator and Formatter class is a bit different than
# that of mpl's, and you cannot directly use mpl's Locator and
# Formatter here (but may be possible in the future).
tick_formatter1 = angle_helper.FormatterDMS()

grid_helper = GridHelperCurveLinear(tr,
                                    extreme_finder=extreme_finder,
                                    grid_locator1=grid_locator1,
                                    tick_formatter1=tick_formatter1
                                    )
```

Again, the *transData* of the axes is still a rectilinear coordinate (image coordinate). You may manually do conversion between two coordinates, or you may use Parasite Axes for convenience.:

```
ax1 = SubplotHost(fig, 1, 2, 2, grid_helper=grid_helper)

# A parasite axes with given transform
ax2 = ParasiteAxesAuxTrans(ax1, tr, "equal")
# note that ax2.transData == tr + ax1.transData
# Anthing you draw in ax2 will match the ticks and grids of ax1.
ax1.parasites.append(ax2)
```



30.2.5 FloatingAxis

A floating axis is an axis one of whose data coordinate is fixed, i.e, its location is not fixed in Axes coordinate but changes as axes data limits changes. A floating axis can be created using *new_floating_axis* method. However, it is your responsibility that the resulting AxisArtist is properly added to the axes. A recommended way is to add it as an item of Axes's axis attribute.:

```
# floating axis whose first (index starts from 0) coordinate
# (theta) is fixed at 60

ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
axis.label.set_text(r"$\theta = 60^{\circ}$")
axis.label.set_visible(True)
```

See the first example of this page.

30.2.6 Current Limitations and TODO's

The code need more refinement. Here is a incomplete list of issues and TODO's

- No easy way to support a user customized tick location (for curvilinear grid). A new Locator class needs to be created.
- FloatingAxis may have coordinate limits, e.g., a floating axis of x = 0, but y only spans from 0 to 1.
- The location of axislabel of FloatingAxis needs to be optionally given as a coordinate value. ex, a floating axis of x=0 with label at y=1

CHAPTER

THIRTYONE

THE MATPLOTLIB AXESGRID TOOLKIT API

Release 1.5.3

Date December 05, 2016

31.1 mpl_toolkits.axes_grid.axes_size

class mpl_toolkits.axes_grid.axes_size.**Fixed**(*fixed_size*)
Simple fixed size with absolute part = *fixed_size* and relative part = 0

class mpl_toolkits.axes_grid.axes_size.**Scaled**(*scalable_size*)
Simple scaled(?) size with absolute part = 0 and relative part = *scalable_size*

class mpl_toolkits.axes_grid.axes_size.**AxesX**(*axes*, *aspect=1.0*, *ref_ax=None*)
Scaled size whose relative part corresponds to the data width of the *axes* multiplied by the *aspect*.

class mpl_toolkits.axes_grid.axes_size.**AxesY**(*axes*, *aspect=1.0*, *ref_ax=None*)
Scaled size whose relative part corresponds to the data height of the *axes* multiplied by the *aspect*.

class mpl_toolkits.axes_grid.axes_size.**MaxWidth**(*artist_list*)
Size whose absolute part is the largest width of the given *artist_list*.

class mpl_toolkits.axes_grid.axes_size.**MaxHeight**(*artist_list*)
Size whose absolute part is the largest height of the given *artist_list*.

class mpl_toolkits.axes_grid.axes_size.**Fraction**(*fraction*, *ref_size*)
An instance whose size is a *fraction* of the *ref_size*.

```
>>> s = Fraction(0.3, AxesX(ax))
```

class mpl_toolkits.axes_grid.axes_size.**Padded**(*size*, *pad*)
Return a instance where the absolute part of *size* is increase by the amount of *pad*.

mpl_toolkits.axes_grid.axes_size.**from_any**(*size*, *fraction_ref=None*)
Creates Fixed unit when the first argument is a float, or a Fraction unit if that is a string that ends with %. The second argument is only meaningful when Fraction unit is created.:

```
>>> a = Size.from_any(1.2) # => Size.Fixed(1.2)
>>> Size.from_any("50%", a) # => Size.Fraction(0.5, a)
```

31.2 mpl_toolkits.axes_grid.axes_divider

class mpl_toolkits.axes_grid.axes_divider.**Divider**(*fig*, *pos*, *horizontal*, *vertical*, *aspect=None*, *anchor='C'*)

This is the class that is used calculates the axes position. It divides the given rectangular area into several sub-rectangles. You initialize the divider by setting the horizontal and vertical lists of sizes (*mpl_toolkits.axes_grid.axes_size*) that the division will be based on. You then use the new_locator method to create a callable object that can be used as the axes_locator of the axes.

Parameters fig : Figure

pos : tuple of 4 floats

position of the rectangle that will be divided

horizontal : list of *axes_size*

sizes for horizontal division

vertical : list of *axes_size*

sizes for vertical division

aspect : bool

if True, the overall rectangular area is reduced so that the relative part of the horizontal and vertical scales have the same scale.

anchor : {'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}

placement of the reduced rectangle when *aspect* is True

add_auto_adjustable_area(*use_axes*, *pad=0.1*, *adjust_dirs=None*)

append_size(*position*, *size*)

get_anchor()

return the anchor

get_aspect()

return aspect

get_horizontal()

return horizontal sizes

get_horizontal_sizes(*renderer*)

get_locator()

get_position()

return the position of the rectangle.

`get_position_runtime`(*ax*, *renderer*)

`get_vertical`()
return vertical sizes

`get_vertical_sizes`(*renderer*)

`get_vsize_hsize`()

`locate`(*nx*, *ny*, *nx1=None*, *ny1=None*, *axes=None*, *renderer=None*)

Parameters nx, nx1 : int

Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1 : int

Same as *nx* and *nx1*, but for row positions.

axes

renderer

`new_locator`(*nx*, *ny*, *nx1=None*, *ny1=None*)
Returns a new locator (`mpl_toolkits.axes_grid.axes_divider.AxesLocator`) for specified cell.

Parameters nx, nx1 : int

Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1 : int

Same as *nx* and *nx1*, but for row positions.

`set_anchor`(*anchor*)

Parameters anchor : {'C', 'SW', 'S', 'SE', 'E', 'NE', 'N', 'NW', 'W'}

anchor position

value	description
'C'	Center
'SW'	bottom left
'S'	bottom
'SE'	bottom right
'E'	right
'NE'	top right
'N'	top
'NW'	top left
'W'	left

set_aspect(*aspect=False*)

Parameters aspect : bool

set_horizontal(*h*)

Parameters h : list of `axes_size`

sizes for horizontal division

set_locator(*_locator*)

set_position(*pos*)

set the position of the rectangle.

Parameters pos : tuple of 4 floats

position of the rectangle that will be divided

set_vertical(*v*)

Parameters v : list of `axes_size`

sizes for vertical division

class mpl_toolkits.axes_grid.axes_divider.**AxesLocator**(*axes_divider*, *nx*, *ny*, *nx1=None*, *ny1=None*)

A simple callable object, initialized with AxesDivider class, returns the position and size of the given cell.

Parameters axes_divider : AxesDivider

nx, nx1 : int

Integers specifying the column-position of the cell. When *nx1* is None, a single *nx*-th column is specified. Otherwise location of columns spanning between *nx* to *nx1* (but excluding *nx1*-th column) is specified.

ny, ny1 : int

Same as *nx* and *nx1*, but for row positions.

get_subplotspec()

class mpl_toolkits.axes_grid.axes_divider.**SubplotDivider**(*fig*, **args*, ***kwargs*)
The Divider class whose rectangle area is specified as a subplot geometry.

Parameters fig : `matplotlib.figure.Figure`

args : tuple (*numRows*, *numCols*, *plotNum*)

The array of subplots in the figure has dimensions *numRows*, *numCols*, and *plotNum* is the number of the subplot being created. *plotNum* starts at 1 in the upper left corner and increases to the right.

If *numRows* <= *numCols* <= *plotNum* < 10, *args* can be the decimal integer *numRows* * 100 + *numCols* * 10 + *plotNum*.

change_geometry(*numrows*, *numcols*, *num*)
change subplot geometry, e.g., from 1,1,1 to 2,2,3

get_geometry()
get the subplot geometry, e.g., 2,2,3

get_position()
return the bounds of the subplot box

get_subplotspec()
get the SubplotSpec instance

set_subplotspec(*subplotspec*)
set the SubplotSpec instance

update_params()
update the subplot position from fig.subplotpars

class mpl_toolkits.axes_grid.axes_divider.**AxesDivider**(*axes*, *xref=None*, *yref=None*)
Divider based on the pre-existing axes.

Parameters axes : `Axes`

xref

yref

append_axes(*position*, *size*, *pad=None*, *add_to_figure=True*, ***kwargs*)
create an axes at the given *position* with the same height (or width) of the main axes.

position [”left”|”right”|”bottom”|”top”]

size and *pad* should be axes_grid.axes_size compatible.

new_horizontal(*size*, *pad=None*, *pack_start=False*, ***kwargs*)
Add a new axes on the right (or left) side of the main axes.

Parameters size : `axes_size` or float or string

A width of the axes. If float or string is given, *from_any* function is used to create the size, with *ref_size* set to AxesX instance of the current axes.

pad : `axes_size` or float or string

Pad between the axes. It takes same argument as *size*.

pack_start : bool

If False, the new axes is appended at the end of the list, i.e., it became the right-most axes. If True, it is inserted at the start of the list, and becomes the left-most axes.

kwargs

All extra keywords arguments are passed to the created axes. If *axes_class* is given, the new axes will be created as an instance of the given class. Otherwise, the same class of the main axes will be used.

new_vertical(*size*, *pad=None*, *pack_start=False*, ***kwargs*)

Add a new axes on the top (or bottom) side of the main axes.

Parameters size : `axes_size` or float or string

A height of the axes. If float or string is given, *from_any* function is used to create the size, with *ref_size* set to AxesX instance of the current axes.

pad : `axes_size` or float or string

Pad between the axes. It takes same argument as *size*.

pack_start : bool

If False, the new axes is appended at the end of the list, i.e., it became the right-most axes. If True, it is inserted at the start of the list, and becomes the left-most axes.

kwargs

All extra keywords arguments are passed to the created axes. If *axes_class* is given, the new axes will be created as an instance of the given class. Otherwise, the same class of the main axes will be used.

31.3 mpl_toolkits.axes_grid.axes_grid

class mpl_toolkits.axes_grid.axes_grid.**Grid**(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *add_all=True*, *share_all=False*, *share_x=True*, *share_y=True*, *label_mode='L'*, *axes_class=None*)

Build an `Grid` instance with a grid nrows*ncols `Axes` in `Figure` *fig* with *rect=[left, bottom, width, height]* (in `Figure` coordinates) or the subplot position code (e.g., "121").

Optional keyword arguments:

Keyword	Default	Description
direction	"row"	["row" | "column"]
axes_pad	0.02	float| pad between axes given in inches or tuple-like of floats, (horizontal padding, vertical padding)
add_all	True	[True | False]
share_all	False	[True | False]
share_x	True	[True | False]
share_y	True	[True | False]
label_mode	"L"	["L" | "1" | "all"]
axes_class	None	a type object which must be a subclass of `Axes`

class `mpl_toolkits.axes_grid.axes_grid.`**`ImageGrid`**(*fig*, *rect*, *nrows_ncols*, *ngrids=None*, *direction='row'*, *axes_pad=0.02*, *add_all=True*, *share_all=False*, *aspect=True*, *label_mode='L'*, *cbar_mode=None*, *cbar_location='right'*, *cbar_pad=None*, *cbar_size='5%'*, *cbar_set_cax=True*, *axes_class=None*)

Build an `ImageGrid` instance with a grid nrows*ncols `Axes` in `Figure` *fig* with *rect=[left, bottom, width, height]* (in `Figure` coordinates) or the subplot position code (e.g., "121").

Optional keyword arguments:

Keyword	Default	Description
direction	"row"	["row" | "column"]
axes_pad	0.02	float| pad between axes given in inches or tuple-like of floats, (horizontal padding, vertical padding)
add_all	True	[True | False]
share_all	False	[True | False]
aspect	True	[True | False]
label_mode	"L"	["L" | "1" | "all"]
cbar_mode	None	["each" | "single" | "edge"]
cbar_location	"right"	["left" | "right" | "bottom" | "top"]
cbar_pad	None	
cbar_size	"5%"	
cbar_set_cax	True	[True | False]
axes_class	None	a type object which must be a subclass of axes_grid's subclass of `Axes`

cbar_set_cax [if True, each axes in the grid has a cax] attribute that is bind to associated cbar_axes.

31.4 mpl_toolkits.axes_grid.axis_artist

class `mpl_toolkits.axes_grid.axis_artist.`**`AxisArtist`**(*axes*, *helper*, *offset=None*, *axis_direction='bottom'*, ***kw*)

An artist which draws axis (a line along which the n-th axes coord is constant) line, ticks, ticklabels, and axis label.

axes : axes *helper* : an AxisArtistHelper instance.

`LABELPAD`

`ZORDER` = 2.5

`draw`(*artist*, *renderer*, **args*, ***kwargs*)
Draw the axis lines, tick lines and labels

`get_axisline_style`()
return the current axisline style.

`get_helper`()
Return axis artist helper instance.

`get_tightbbox`(*renderer*)

`get_transform`()

`invert_ticklabel_direction`()

`set_axis_direction`(*axis_direction*)
Adjust the direction, text angle, text alignment of ticklabels, labels following the matplotlib convention for the rectangle axes.

The *axis_direction* must be one of [left, right, bottom, top].

property	left	bottom	right	top
ticklabels location	"-"	"+"	"+"	"-"
axislabel location	"-"	"+"	"+"	"-"
ticklabels angle	90	0	-90	180
ticklabel va	center	baseline	center	baseline
ticklabel ha	right	center	right	center
axislabel angle	180	0	0	180
axislabel va	center	top	center	bottom
axislabel ha	right	center	right	center

Note that the direction "+" and "-" are relative to the direction of the increasing coordinate. Also, the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

set_axislabel_direction(*label_direction*)
Adjust the direction of the axislabel.

ACCEPTS: ["+" | "-"]

Note that the label_direction '+' and '-' are relative to the direction of the increasing coordinate.

set_axisline_style(*axisline_style=None*, ***kw*)
Set the axisline style.

***axisline_style* can be a string with axisline style name with optional** comma-separated attributes. Alternatively, the attrs can be provided as keywords.

set_arrowstyle("->,size=1.5") set_arrowstyle("->", size=1.5)

Old attrs simply are forgotten.

Without argument (or with arrowstyle=None), return available styles as a list of strings.

set_label(*s*)

set_ticklabel_direction(*tick_direction*)
Adjust the direction of the ticklabel.

ACCEPTS: ["+" | "-"]

Note that the label_direction '+' and '-' are relative to the direction of the increasing coordinate.

toggle(*all=None*, *ticks=None*, *ticklabels=None*, *label=None*)
Toggle visibility of ticks, ticklabels, and (axis) label. To turn all off,

```
axis.toggle(all=False)
```

To turn all off but ticks on

```
axis.toggle(all=False, ticks=True)
```

To turn all on but (axis) label off

```
axis.toggle(all=True, label=False))
```

class mpl_toolkits.axes_grid.axis_artist.**Ticks**(*ticksize*, *tick_out=False*, ***kwargs*)
Ticks are derived from Line2D, and note that ticks themselves are markers. Thus, you should use set_mec, set_mew, etc.

To change the tick size (length), you need to use set_ticksize. To change the direction of the ticks (ticks are in opposite direction of ticklabels by default), use set_tick_out(False).

get_tick_out()
Return True if the tick will be rotated by 180 degree.

get_ticksize()
Return length of the ticks in points.

set_tick_out(*b*)
set True if tick need to be rotated by 180 degree.

set_ticksize(*ticksize*)
set length of the ticks in points.

class mpl_toolkits.axes_grid.axis_artist.**AxisLabel**(**kl*, ***kwargs*)
Axis Label. Derived from Text. The position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text.

To change the pad between ticklabels and axis label, use set_pad.

get_pad()
return pad in points. See set_pad for more details.

set_axis_direction(*d*)
Adjust the text angle and text alignment of axis label according to the matplotlib convention.

property	left	bottom	right	top
axislabel angle	180	0	0	180
axislabel va	center	top	center	bottom
axislabel ha	right	center	right	center

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

set_pad(*pad*)
Set the pad in points. Note that the actual pad will be the sum of the internal pad and the external pad (that are set automatically by the AxisArtist), and it only set the internal pad

class mpl_toolkits.axes_grid.axis_artist.**TickLabels**(***kwargs*)
Tick Labels. While derived from Text, this single artist draws all ticklabels. As in AxisLabel, the position of the text is updated in the fly, so changing text position has no effect. Otherwise, the properties can be changed as a normal Text. Unlike the ticklabels of the mainline matplotlib, properties of single ticklabel alone cannot modified.

To change the pad between ticks and ticklabels, use set_pad.

get_texts_widths_heights_descents(*renderer*)
return a list of width, height, descent for ticklabels.

set_axis_direction(*label_direction*)
Adjust the text angle and text alignment of ticklabels according to the matplotlib convention.

The *label_direction* must be one of [left, right, bottom, top].

property	left	bottom	right	top
ticklabels angle	90	0	-90	180
ticklabel va	center	baseline	center	baseline
ticklabel ha	right	center	right	center

Note that the text angles are actually relative to (90 + angle of the direction to the ticklabel), which gives 0 for bottom axis.

CHAPTER

THIRTYTWO

THE MATPLOTLIB AXES_GRID1 TOOLKIT API

Release 1.5.3

Date December 05, 2016

32.1 mpl_toolkits.axes_grid1.inset_locator

A collection of functions and objects for creating or placing inset axes.

class mpl_toolkits.axes_grid1.inset_locator.**AnchoredLocatorBase**(*bbox_to_anchor*, *offsetbox*, *loc*, *borderpad=0.5*, *bbox_transform=None*)

Bases: *matplotlib.offsetbox.AnchoredOffsetbox*

draw(*renderer*)

class mpl_toolkits.axes_grid1.inset_locator.**AnchoredSizeLocator**(*bbox_to_anchor*, *x_size*, *y_size*, *loc*, *borderpad=0.5*, *bbox_transform=None*)

Bases: *mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase*

get_extent(*renderer*)

class mpl_toolkits.axes_grid1.inset_locator.**AnchoredZoomLocator**(*parent_axes*, *zoom*, *loc*, *borderpad=0.5*, *bbox_to_anchor=None*, *bbox_transform=None*)

Bases: *mpl_toolkits.axes_grid1.inset_locator.AnchoredLocatorBase*

get_extent(*renderer*)

class mpl_toolkits.axes_grid1.inset_locator.**BboxConnector**(*bbox1*, *bbox2*, *loc1*, *loc2=None*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

Connect two bboxes with a straight line.

Parameters bbox1, bbox2 : *matplotlib.transforms.Bbox*

Bounding boxes to connect.

loc1 : {1, 2, 3, 4}

Corner of *bbox1* to draw the line. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

loc2 : {1, 2, 3, 4}, optional

Corner of *bbox2* to draw the line. If None, defaults to *loc1*. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

****kwargs**

Patch properties for the line drawn. Valid arguments include:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'
linewidth or lw	float or None for default

Continued on

Table 32.1 – continued from previous page

Property	Description
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

static connect_bbox(*bbox1*, *bbox2*, *loc1*, *loc2=None*)

Helper function to obtain a Path from one bbox to another.

Parameters bbox1, bbox2 : *matplotlib.transforms.Bbox*

Bounding boxes to connect.

loc1 : {1, 2, 3, 4}

Corner of *bbox1* to use. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

loc2 : {1, 2, 3, 4}, optional

Corner of *bbox2* to use. If None, defaults to *loc1*. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

Returns path : *matplotlib.path.Path*

A line segment from the *loc1* corner of *bbox1* to the *loc2* corner of *bbox2*.

static get_bbox_edge_pos(*bbox*, *loc*)

Helper function to obtain the location of a corner of a bbox

Parameters bbox : *matplotlib.transforms.Bbox*

loc : {1, 2, 3, 4}

Corner of *bbox*. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

Returns x, y : float
Coordinates of the corner specified by *loc*.

get_path()
Return the path of this patch

class mpl_toolkits.axes_grid1.inset_locator.**BboxConnectorPatch**(*bbox1*, *bbox2*, *loc1a*, *loc2a*, *loc1b*, *loc2b*, ***kwargs*)

Bases: *mpl_toolkits.axes_grid1.inset_locator.BboxConnector*

Connect two bboxes with a quadrilateral.

The quadrilateral is specified by two lines that start and end at corners of the bboxes. The four sides of the quadrilateral are defined by the two lines given, the line between the two corners specified in *bbox1* and the line between the two corners specified in *bbox2*.

Parameters bbox1, bbox2 : *matplotlib.transforms.Bbox*
Bounding boxes to connect.

loc1a, loc2a : {1, 2, 3, 4}
Corners of *bbox1* and *bbox2* to draw the first line. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

loc1b, loc2b : {1, 2, 3, 4}
Corners of *bbox1* and *bbox2* to draw the second line. Valid values are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4
```

****kwargs**
Patch properties for the line drawn:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color

Continued on

Table 32.2 – continued from previous page

Property	Description
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

get_path()

Return the path of this patch

class mpl_toolkits.axes_grid1.inset_locator.**BboxPatch**(*bbox*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

Patch showing the shape bounded by a Bbox.

Parameters bbox : *matplotlib.transforms.Bbox*

Bbox to use for the extents of this patch.

****kwargs**

Patch properties. Valid arguments include:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a *matplotlib.figure.Figure* instance

Continued on

Table 32.3 – continued from previous page

Property	Description
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

get_path()

Return the path of this patch

class `mpl_toolkits.axes_grid1.inset_locator.`**InsetPosition**(*parent*, *lbwh*)

Bases: `object`

An object for positioning an inset axes.

This is created by specifying the normalized coordinates in the axes, instead of the figure.

Parameters parent : *matplotlib.axes.Axes*

Axes to use for normalizing coordinates.

lbwh : iterable of four floats

The left edge, bottom edge, width, and height of the inset axes, in units of the normalized coordinate of the *parent* axes.

See also:

matplotlib.axes.Axes.set_axes_locator()

Examples

The following bounds the inset axes to a box with 20% of the parent axes's height and 40% of the width. The size of the axes specified ([0, 0, 1, 1]) ensures that the axes completely fills the bounding box:

```
>>> parent_axes = plt.gca()
>>> ax_ins = plt.axes([0, 0, 1, 1])
>>> ip = InsetPosition(ax, [0.5, 0.1, 0.4, 0.2])
>>> ax_ins.set_axes_locator(ip)
```

`mpl_toolkits.axes_grid1.inset_locator.`**`inset_axes`**(*parent_axes*, *width*, *height*, *loc=1*, *bbox_to_anchor=None*, *bbox_transform=None*, *axes_class=None*, *axes_kwargs=None*, *borderpad=0.5*)

Create an inset axes with a given width and height.

Both sizes used can be specified either in inches or percentage of the parent axes.

Parameters parent_axes : `matplotlib.axes.Axes`

Axes to place the inset axes.

width, height : float or str

Size of the inset axes to create.

loc : int or string, optional, default to 1

Location to place the inset axes. The valid locations are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

bbox_to_anchor : tuple or `matplotlib.transforms.BboxBase`, optional

Bbox that the inset axes will be anchored. Can be a tuple of [left, bottom, width, height], or a tuple of [left, bottom].

bbox_transform : `matplotlib.transforms.Transform`, optional

Transformation for the bbox. if None, `parent_axes.transAxes` is used.

axes_class : `matplotlib.axes.Axes` type, optional

If specified, the inset axes created with be created with this class's constructor.

axes_kwargs : dict, optional

Keyworded arguments to pass to the constructor of the inset axes. Valid arguments include:

Property	Description
`adjustable`	['box' | 'datalim' | 'box-forced']
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`anchor`	unknown
`animated`	[True | False]
`aspect`	unknown
`autoscale_on`	unknown
`autoscalex_on`	unknown
`autoscaley_on`	unknown
`axes`	an `Axes` instance
`axes_locator`	unknown

Continued on next page

Table 32.4 – continued from previous page

Property	Description
axis_bgcolor	any matplotlib color - see colors()
axisbelow	[*True* | *False*]
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color_cycle	unknown
contains	a callable function
figure	unknown
frame_on	[*True* | *False*]
gid	an id string
label	string or anything printable with '%s' conversion.
navigate	[*True* | *False*]
navigate_mode	unknown
path_effects	unknown
picker	[None|float|boolean|callable]
position	unknown
rasterization_zorder	unknown
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
title	unknown
transform	Transform instance
url	a url string
visible	[True | False]
xbound	unknown
xlabel	unknown
xlim	length 2 sequence of floats
xmargin	unknown
xscale	['linear' | 'log' | 'logit' | 'symlog']
xticklabels	sequence of strings
xticks	sequence of floats
ybound	unknown
ylabel	unknown
ylim	length 2 sequence of floats
ymargin	unknown
yscale	['linear' | 'log' | 'logit' | 'symlog']
yticklabels	sequence of strings
yticks	sequence of floats
zorder	any number

borderpad : float, optional
Padding between inset axes and the bbox_to_anchor. Defaults to 0.5.

Returns inset_axes : `axes_class`
Inset axes object created.

`mpl_toolkits.axes_grid1.inset_locator.`**`mark_inset`**(*parent_axes*, *inset_axes*, *loc1*, *loc2*, ***kwargs*)

Draw a box to mark the location of an area represented by an inset axes.

This function draws a box in *parent_axes* at the bounding box of *inset_axes*, and shows a connection with the inset axes by drawing lines at the corners, giving a "zoomed in" effect.

Parameters parent_axes : `matplotlib.axes.Axes`

Axes which contains the area of the inset axes.

inset_axes : `matplotlib.axes.Axes`

The inset axes.

loc1, loc2 : {1, 2, 3, 4}

Corners to use for connecting the inset axes and the area in the parent axes.

****kwargs**

Patch properties for the lines and box drawn:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]

Continued on

Table 32.5 – continued from previous page

Property	Description
zorder	any number

Returns pp : *matplotlib.patches.Patch*
The patch drawn to represent the area of the inset axes.

p1, p2 : *matplotlib.patches.Patch*
The patches connecting two corners of the inset axes and its area.

`mpl_toolkits.axes_grid1.inset_locator.`**`zoomed_inset_axes`**(*parent_axes*, *zoom*, *loc=1*, *bbox_to_anchor=None*, *bbox_transform=None*, *axes_class=None*, *axes_kwargs=None*, *borderpad=0.5*)

Create an anchored inset axes by scaling a parent axes.

Parameters parent_axes : *matplotlib.axes.Axes*
Axes to place the inset axes.

zoom : float
Scaling factor of the data axes. *zoom* > 1 will enlargen the coordinates (i.e., "zoomed in"), while *zoom* < 1 will shrink the coordinates (i.e., "zoomed out").

loc : int or string, optional, default to 1
Location to place the inset axes. The valid locations are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10
```

bbox_to_anchor : tuple or *matplotlib.transforms.BboxBase*, optional
Bbox that the inset axes will be anchored. Can be a tuple of [left, bottom, width, height], or a tuple of [left, bottom].

bbox_transform : *matplotlib.transforms.Transform*, optional
Transformation for the bbox. if None, `parent_axes.transAxes` is used.

axes_class : *matplotlib.axes.Axes* type, optional
If specified, the inset axes created with be created with this class's constructor.

axes_kwargs : dict, optional
Keyworded arguments to pass to the constructor of the inset axes. Valid arguments include:

Property	Description
adjustable	['box' | 'datalim' | 'box-forced']
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
anchor	unknown
animated	[True | False]
aspect	unknown
autoscale_on	unknown
autoscalex_on	unknown
autoscaley_on	unknown
axes	an *Axes* instance
axes_locator	unknown
axis_bgcolor	any matplotlib color - see *colors()*
axisbelow	[*True* | *False*]
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color_cycle	unknown
contains	a callable function
figure	unknown
frame_on	[*True* | *False*]
gid	an id string
label	string or anything printable with '%s' conversion.
navigate	[*True* | *False*]
navigate_mode	unknown
path_effects	unknown
picker	[None|float|boolean|callable]
position	unknown
rasterization_zorder	unknown
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
title	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
xbound	unknown
xlabel	unknown
xlim	length 2 sequence of floats
xmargin	unknown
xscale	['linear' | 'log' | 'logit' | 'symlog']
xticklabels	sequence of strings
xticks	sequence of floats
ybound	unknown
ylabel	unknown

Continued on next page

Table 32.6 – continued from previous page

Property	Description
`ylim`	length 2 sequence of floats
`ymargin`	unknown
`yscale`	['linear' | 'log' | 'logit' | 'symlog']
`yticklabels`	sequence of strings
`yticks`	sequence of floats
`zorder`	any number

borderpad : float, optional
    Padding between inset axes and the bbox_to_anchor. Defaults to 0.5.

Returns inset_axes : `axes_class`
    Inset axes object created.

Part VII

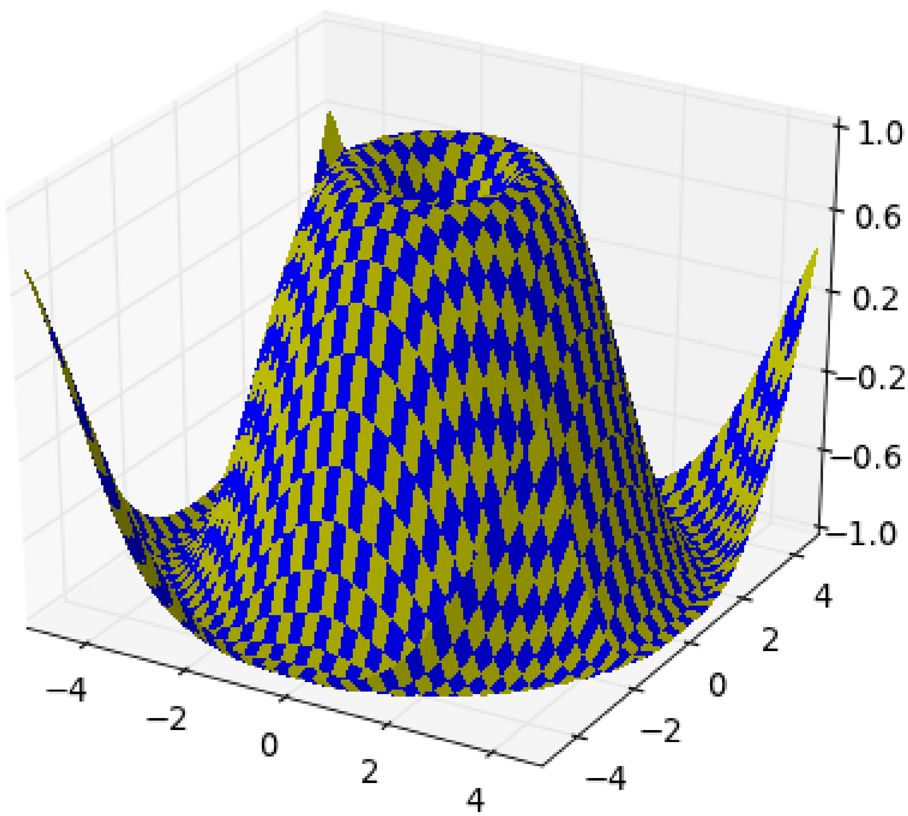
mplot3d

CHAPTER

THIRTYTHREE

MATPLOTLIB MPLOT3D TOOLKIT

The mplot3d toolkit adds simple 3D plotting capabilities to matplotlib by supplying an axes object that can create a 2D projection of a 3D scene. The resulting graph will have the same look and feel as regular 2D plots.



The interactive backends also provide the ability to rotate and zoom the 3D scene. One can rotate the 3D scene by simply clicking-and-dragging the scene. Zooming is done by right-clicking the scene and dragging the mouse up and down. Note that one does not use the zoom button like one would use for regular 2D plots.

33.1 mplot3d tutorial

Contents

33.1.1 Getting started

An Axes3D object is created just like any other axes using the projection='3d' keyword. Create a new `matplotlib.figure.Figure` and add a new axes to it of type `Axes3D`:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

New in version 1.0.0: This approach is the preferred method of creating a 3D axes.

Note: Prior to version 1.0.0, the method of creating a 3D axes was different. For those using older versions of matplotlib, change `ax = fig.add_subplot(111, projection='3d')` to `ax = Axes3D(fig)`.

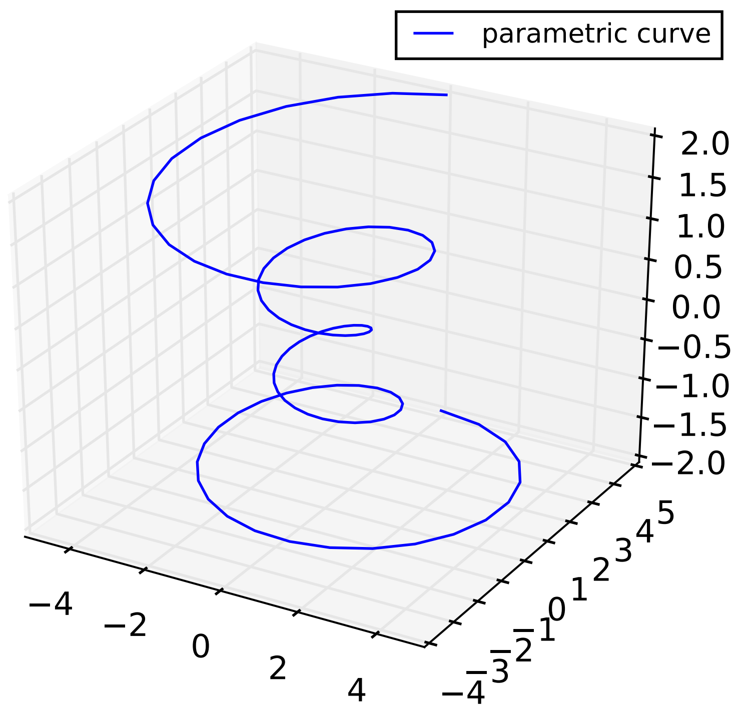
33.1.2 Line plots

`Axes3D.plot`(*xs*, *ys*, **args*, ***kwargs*)

Plot 2D or 3D data.

Argument	Description
xs, *ys*	x, y coordinates of vertices
zs	z value(s), either one for all points or one for each point.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Other arguments are passed on to `plot()`



33.1.3 Scatter plots

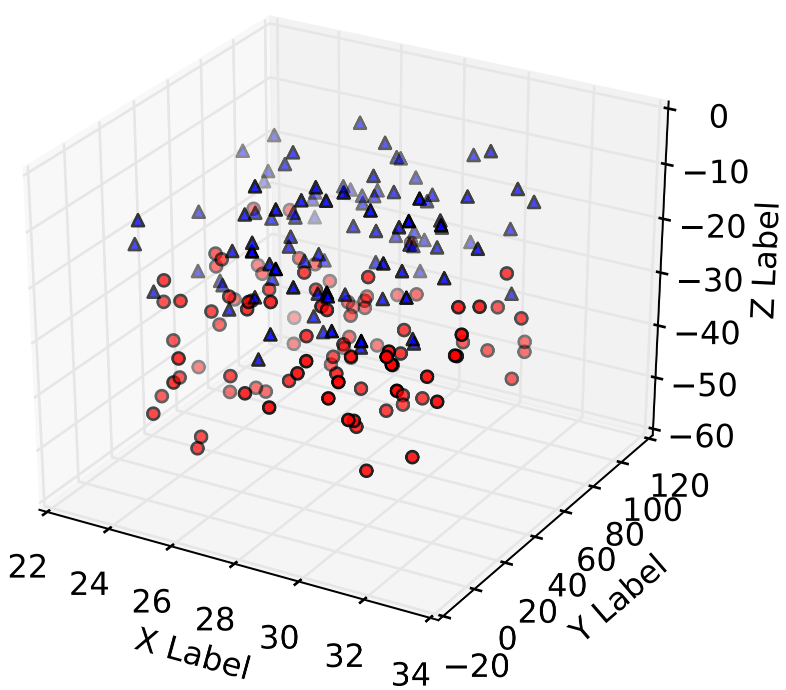
`Axes3D.scatter`(*xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c=None*, *depthshade=True*, **args*, ***kwargs*)

Create a scatter plot.

Argument	Description
xs, *ys*	Positions of data points.
zs	Either an array of the same length as *xs* and *ys* or a single value to place all points in the same plane. Default is 0.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.
s	Size in points^2. It is a scalar or an array of the same length as *x* and *y*.
c	A color. *c* can be a single color format string, or a sequence of color specifications of length *N*, or a sequence of *N* numbers to be mapped to colors using the *cmap* and *norm* specified via kwargs (see below). Note that *c* should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. *c* can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.
depthshade	Whether or not to shade the scatter markers to give the appearance of depth. Default is *True*.

Keyword arguments are passed on to `scatter()`.

Returns a `Patch3DCollection`



33.1.4 Wireframe plots

`Axes3D.plot_wireframe(X, Y, Z, *args, **kwargs)`

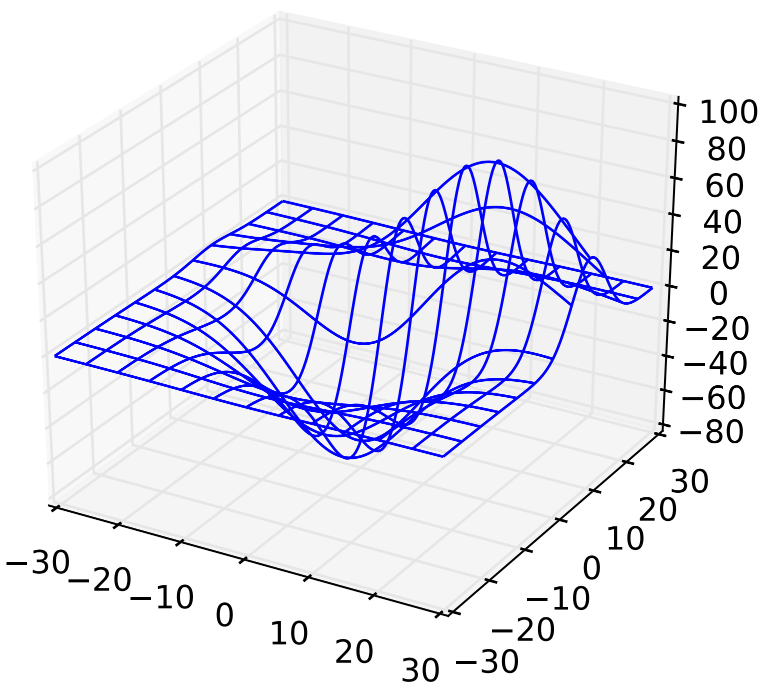
Plot a 3D wireframe.

The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If either is 0 the input data in not sampled along this direction producing a 3D line plot rather than a wireframe plot.

Argument	Description
X, Y,	Data values as 2D arrays
Z	
rstride	Array row stride (step size), defaults to 1
cstride	Array column stride (step size), defaults to 1

Keyword arguments are passed on to *LineCollection*.

Returns a *Line3DCollection*



33.1.5 Surface plots

`Axes3D.plot_surface(X, Y, Z, *args, **kwargs)`

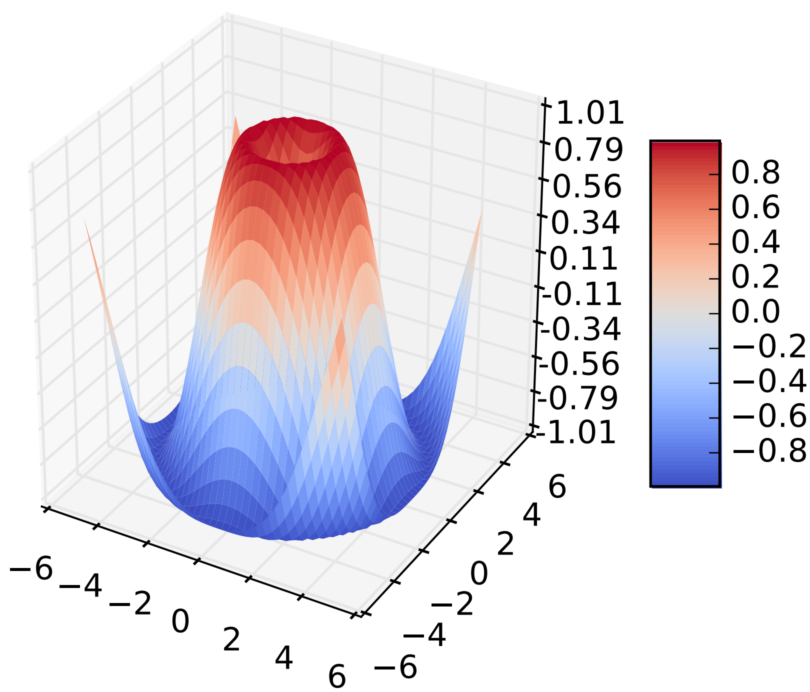
Create a surface plot.

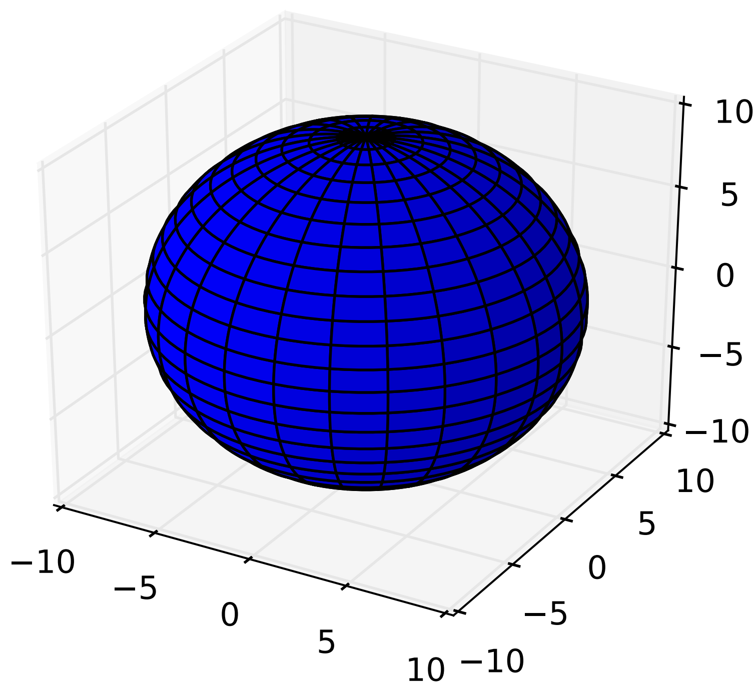
By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the *cmap* argument.

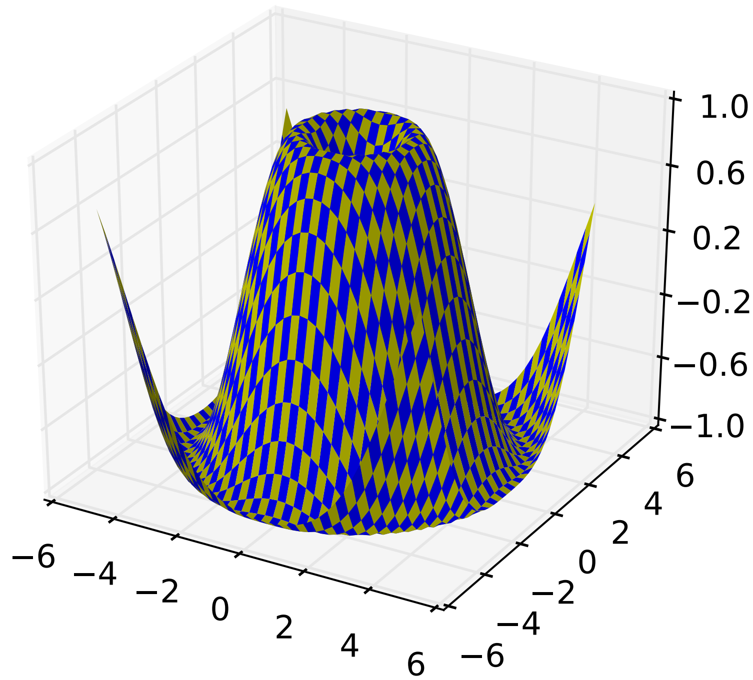
The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in the default values for the strides will result in a 100x100 grid being plotted.

Argument	Description
X, *Y*, *Z*	Data values as 2D arrays
rstride	Array row stride (step size), defaults to 10
cstride	Array column stride (step size), defaults to 10
color	Color of the surface patches
cmap	A colormap for the surface patches.
facecolors	Face colors for the individual patches
norm	An instance of Normalize to map values to colors
vmin	Minimum value to map
vmax	Maximum value to map
shade	Whether to shade the facecolors

Other arguments are passed on to `Poly3DCollection`




10
5
0
−5
−10
10
5
0
−5
−10
−10
−5
0
5
10



33.1.6 Tri-Surface plots

Axes3D.**plot_trisurf**(**args*, ***kwargs*)

Argument	Description
X, *Y*, *Z*	Data values as 1D arrays
color	Color of the surface patches
cmap	A colormap for the surface patches.
norm	An instance of Normalize to map values to colors
vmin	Minimum value to map
vmax	Maximum value to map
shade	Whether to shade the facecolors

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a `Triangulation` object, or:

```
plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

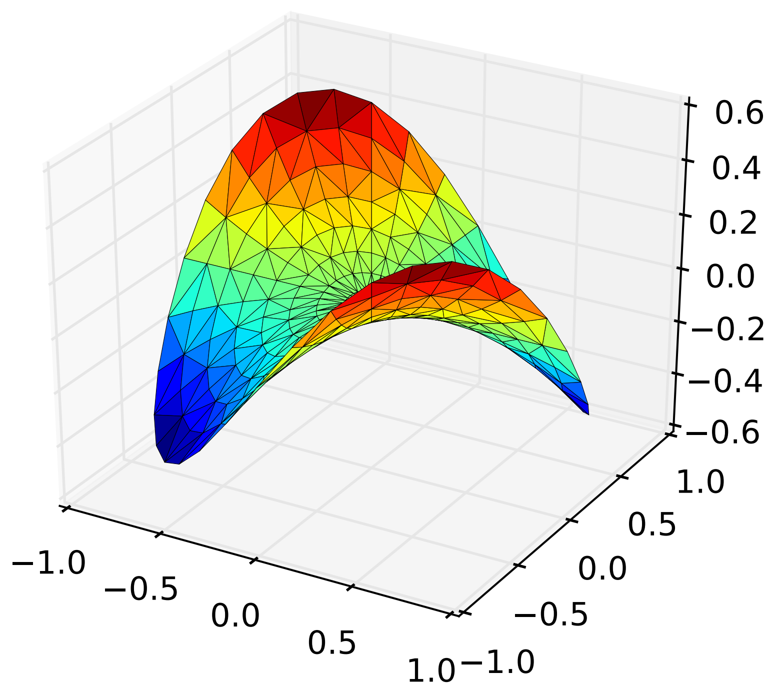
The remaining arguments are:

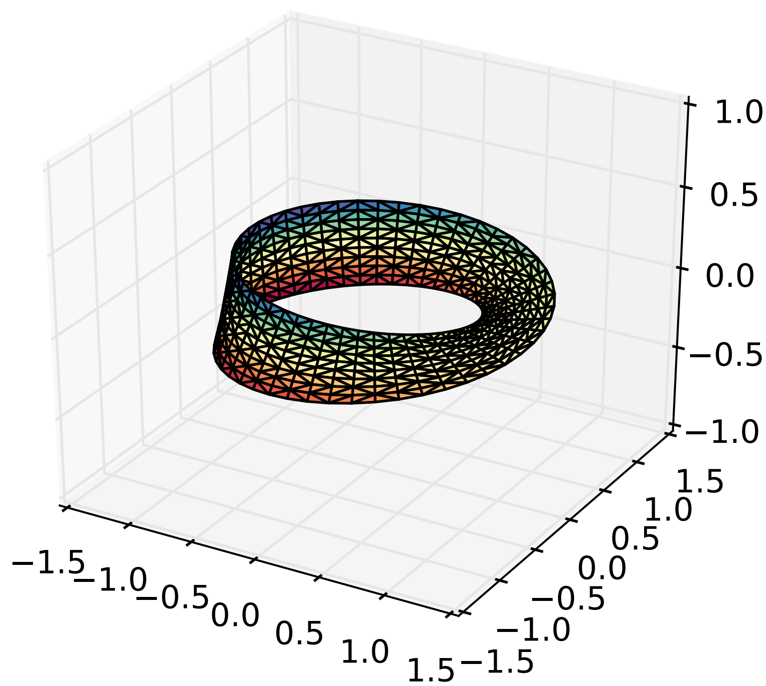
```
plot_trisurf(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation.

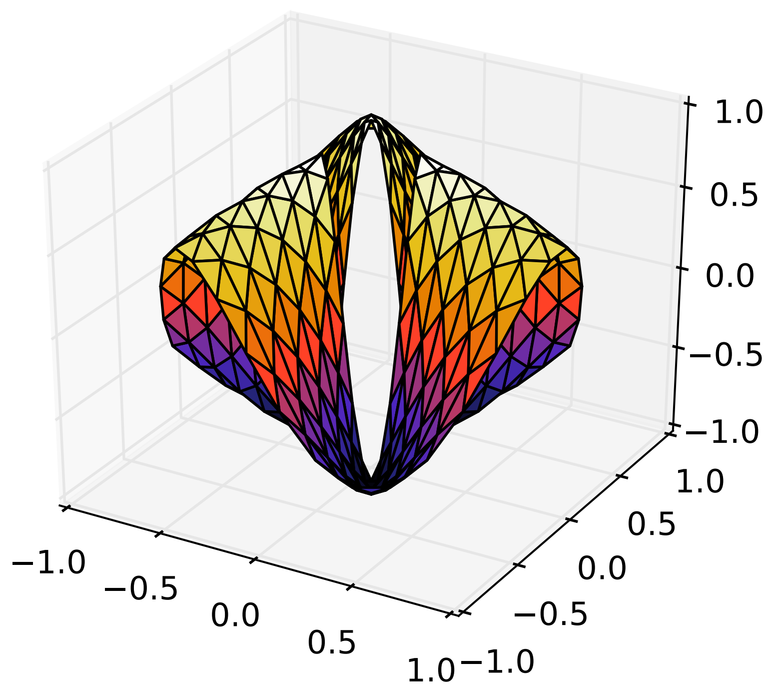
Other arguments are passed on to *Poly3DCollection*

Examples:

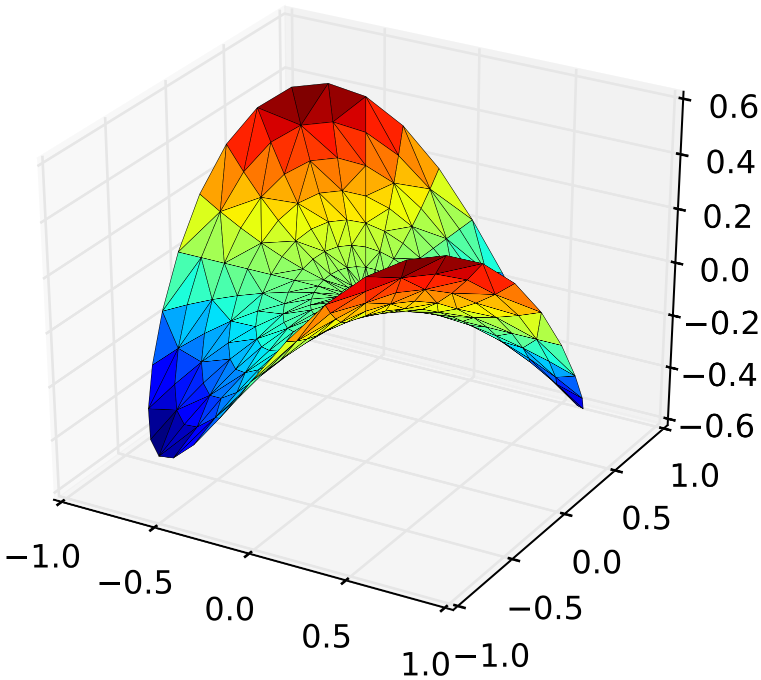




1.0
0.5
0.0
−0.5
−1.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5



New in version 1.2.0: This plotting function was added for the v1.2.0 release.



33.1.7 Contour plots

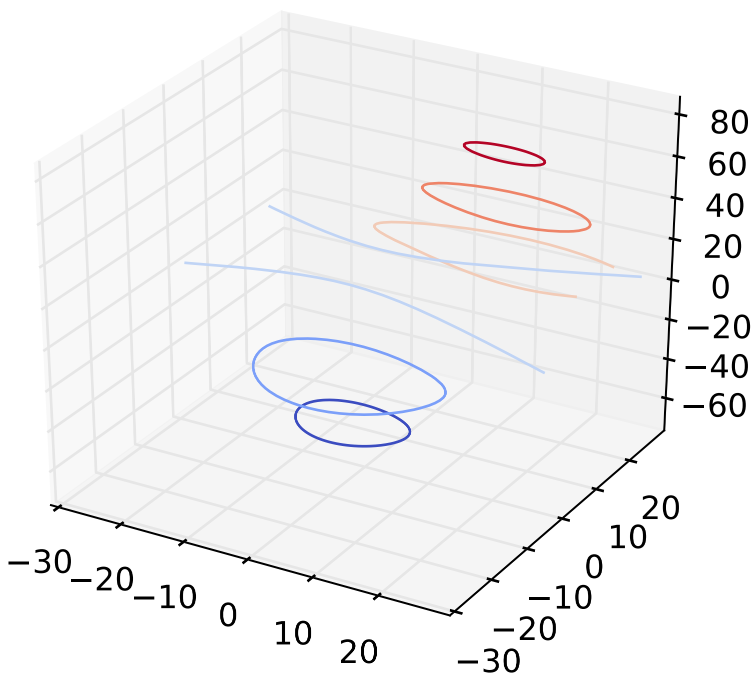
`Axes3D.contour`(*X*, *Y*, *Z*, **args*, ***kwargs*)

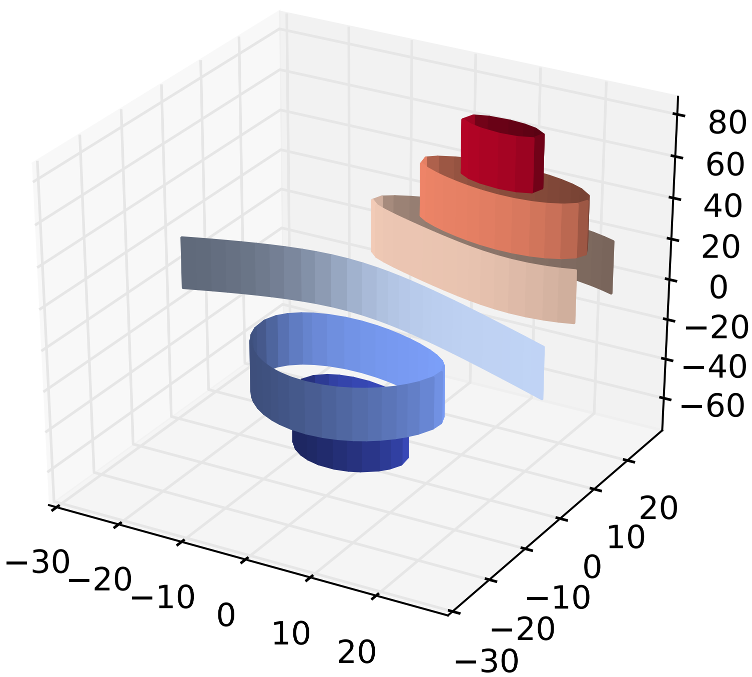
Create a 3D contour plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

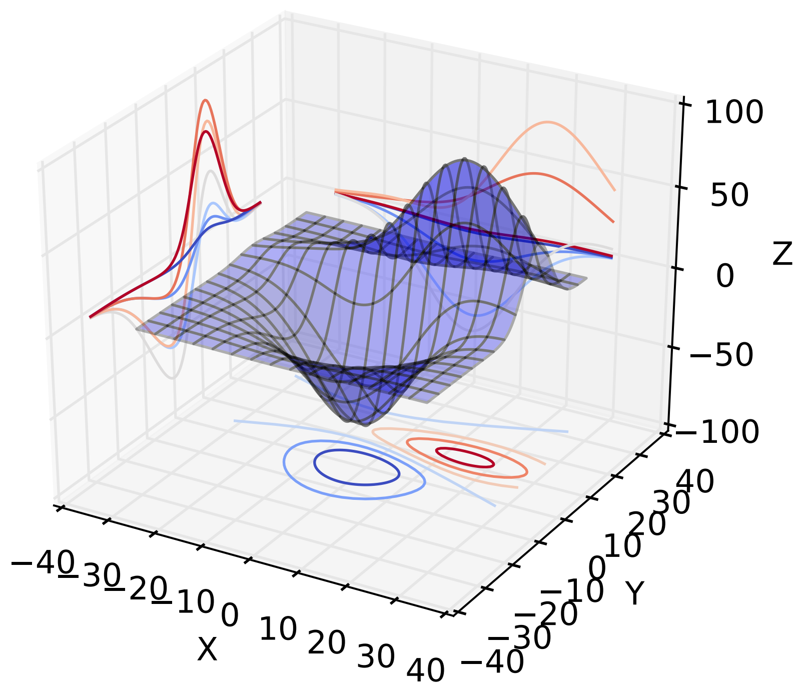
The positional and other keyword arguments are passed on to `contour()`

Returns a `contour`


80
60
40
20
0
−20
−40
−60
20
10
0
−10
−20
−30
−30
−20
−10
0
10
20



80
60
40
20
0
−20
−40
−60
−30
−20
−10
0
10
20
20
10
0
−10
−20
−30



33.1.8 Filled contour plots

`Axes3D.contourf`(*X*, *Y*, *Z*, **args*, ***kwargs*)

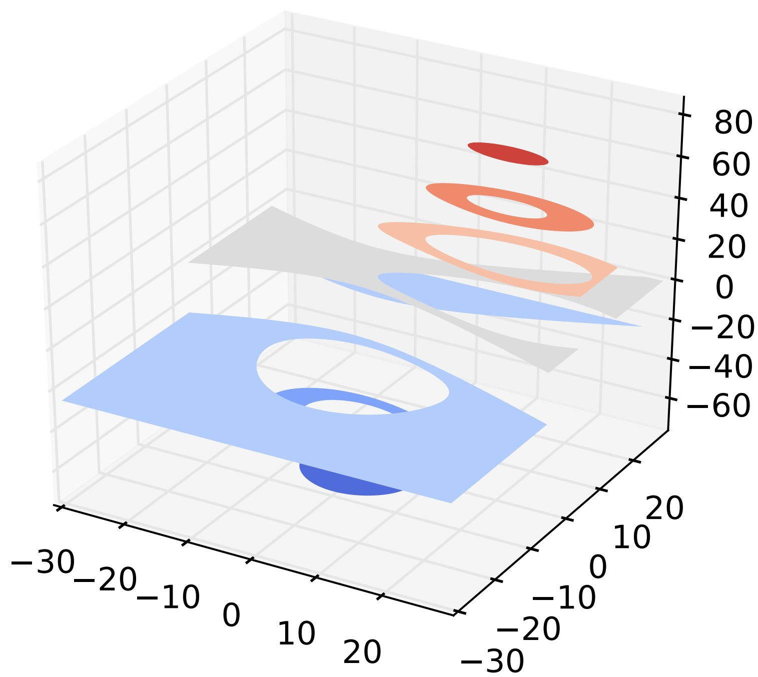
Create a 3D contourf plot.

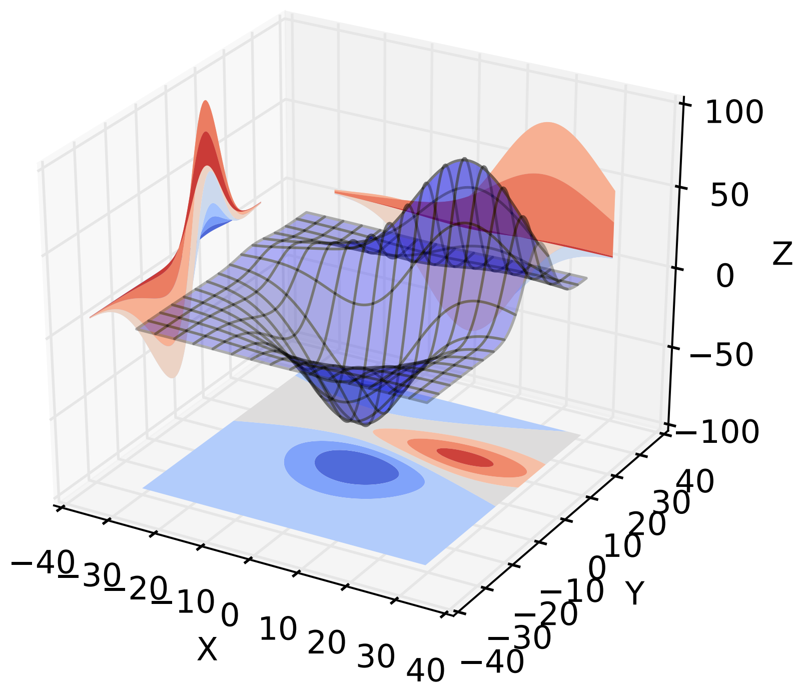
Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the filled contour on this position in plane normal to zdir

The positional and keyword arguments are passed on to `contourf()`

Returns a `contourf`

Changed in version 1.1.0: The *zdir* and *offset* kwargs were added.


80
60
40
20
0
−20
−40
−60
−30
−20
−10
0
10
20
20
10
0
−10
−20
−30



New in version 1.1.0: The feature demoed in the second contourf3d example was enabled as a result of a bugfix for version 1.1.0.

33.1.9 Polygon plots

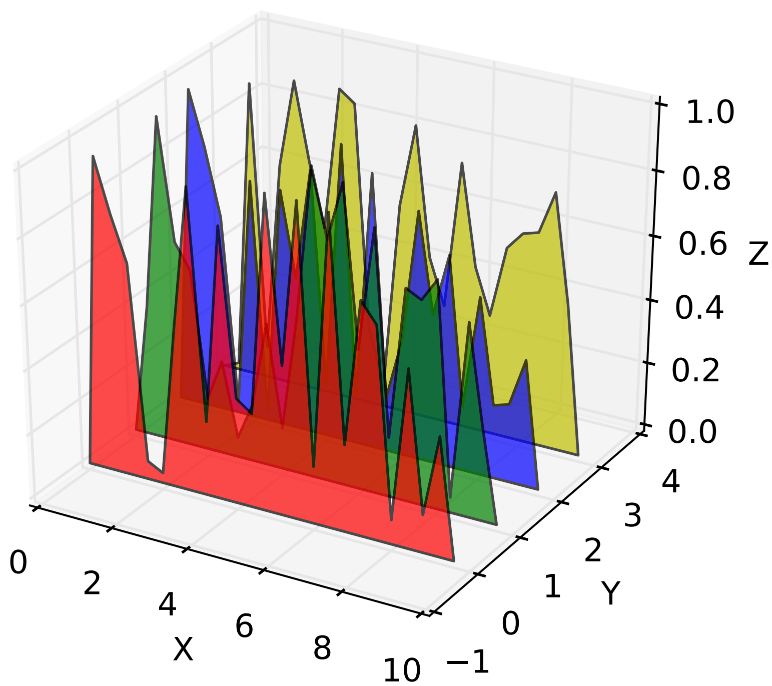
`Axes3D.add_collection3d`(*col*, *zs=0*, *zdir='z'*)

Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

Supported are:

- PolyCollection
- LineColleciton
- PatchCollection



33.1.10 Bar plots

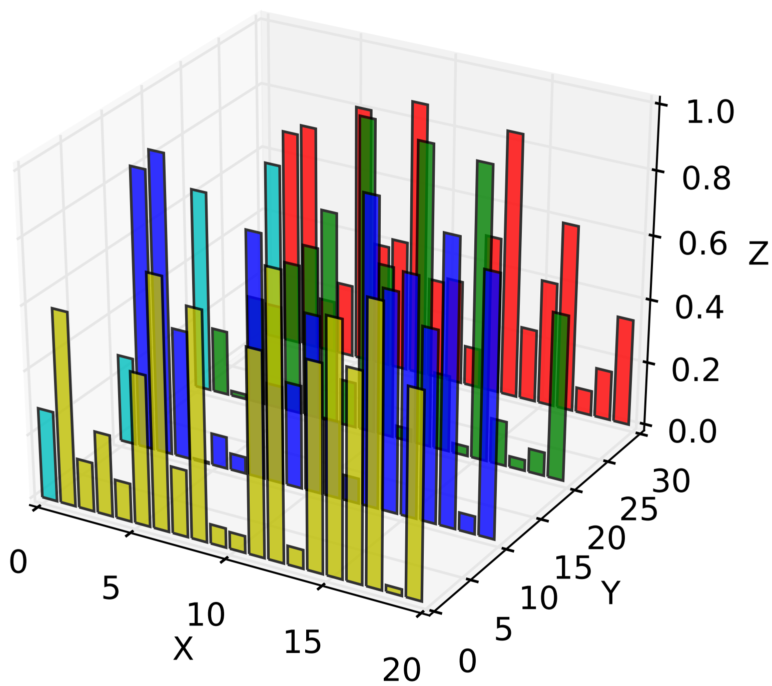
`Axes3D.bar`(*left*, *height*, *zs=0*, *zdir='z'*, **args*, ***kwargs*)

Add 2D bar(s).

Argument	Description
left	The x coordinates of the left sides of the bars.
height	The height of the bars.
zs	Z coordinate of bars, if one value is specified they will all be placed at the same z.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`



33.1.11 Quiver

`Axes3D.quiver(*args, **kwargs)`

Plot a 3D field of arrows.

call signatures:

```
quiver(X, Y, Z, U, V, W, **kwargs)
```

Arguments:

***X*, *Y*, *Z*:** The x, y and z coordinates of the arrow locations (default is tip of arrow; see *pivot* kwarg)

***U*, *V*, *W*:** The x, y and z components of the arrow vectors

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

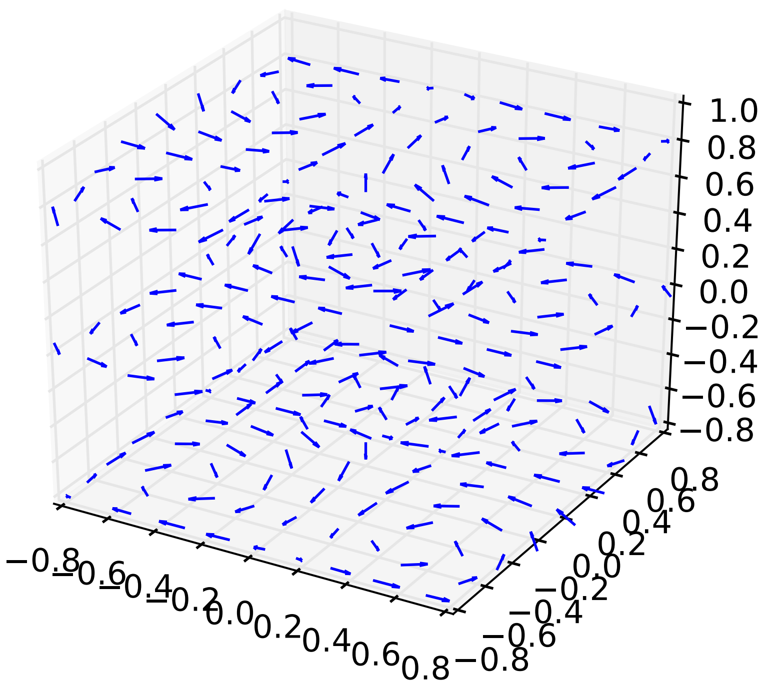
Keyword arguments:

***length*: [1.0 | float]** The length of each quiver, default to 1.0, the unit is the same with the axes

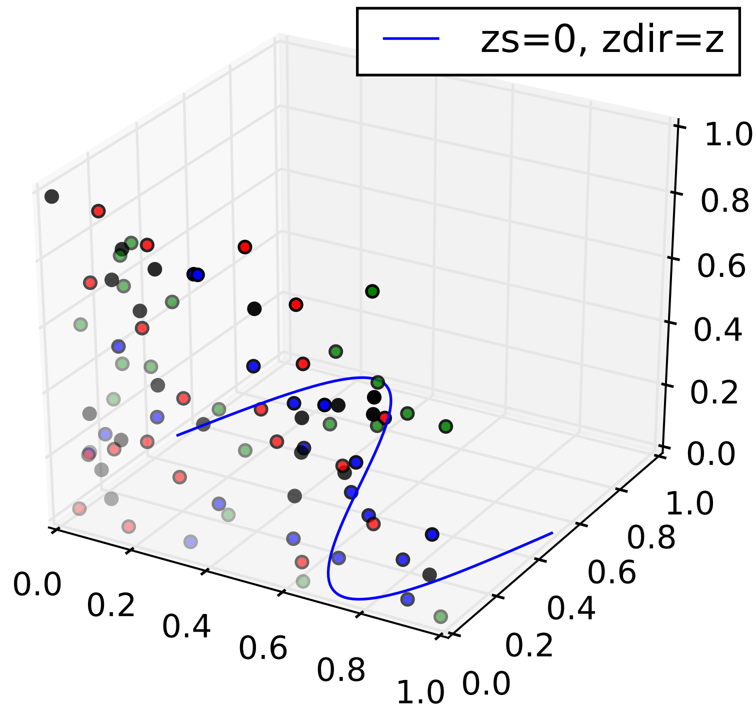
***arrow_length_ratio*: [0.3 | float]** The ratio of the arrow head with respect to the quiver, default to 0.3

***pivot*: ['tail' | 'middle' | 'tip']** The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

Any additional keyword arguments are delegated to `LineCollection`



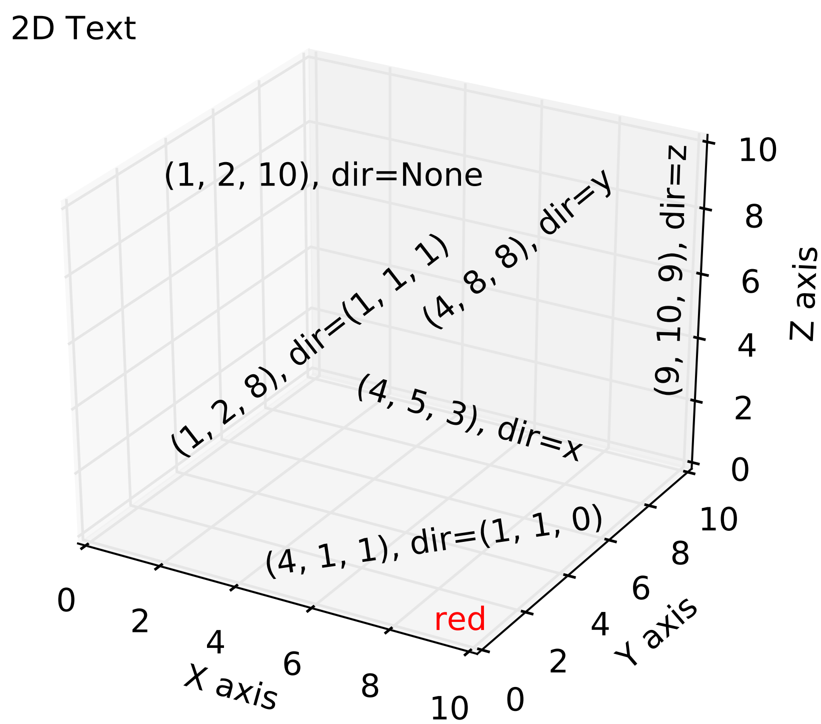
33.1.12 2D plots in 3D



33.1.13 Text

`Axes3D.text`(*x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)

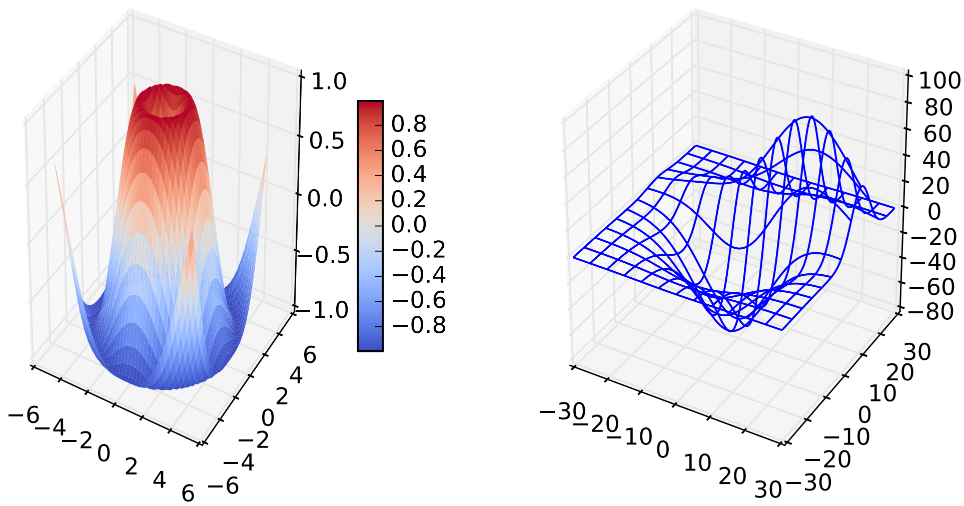
Add text to the plot. kwargs will be passed on to Axes.text, except for the `zdir` keyword, which sets the direction to be used as the z direction.

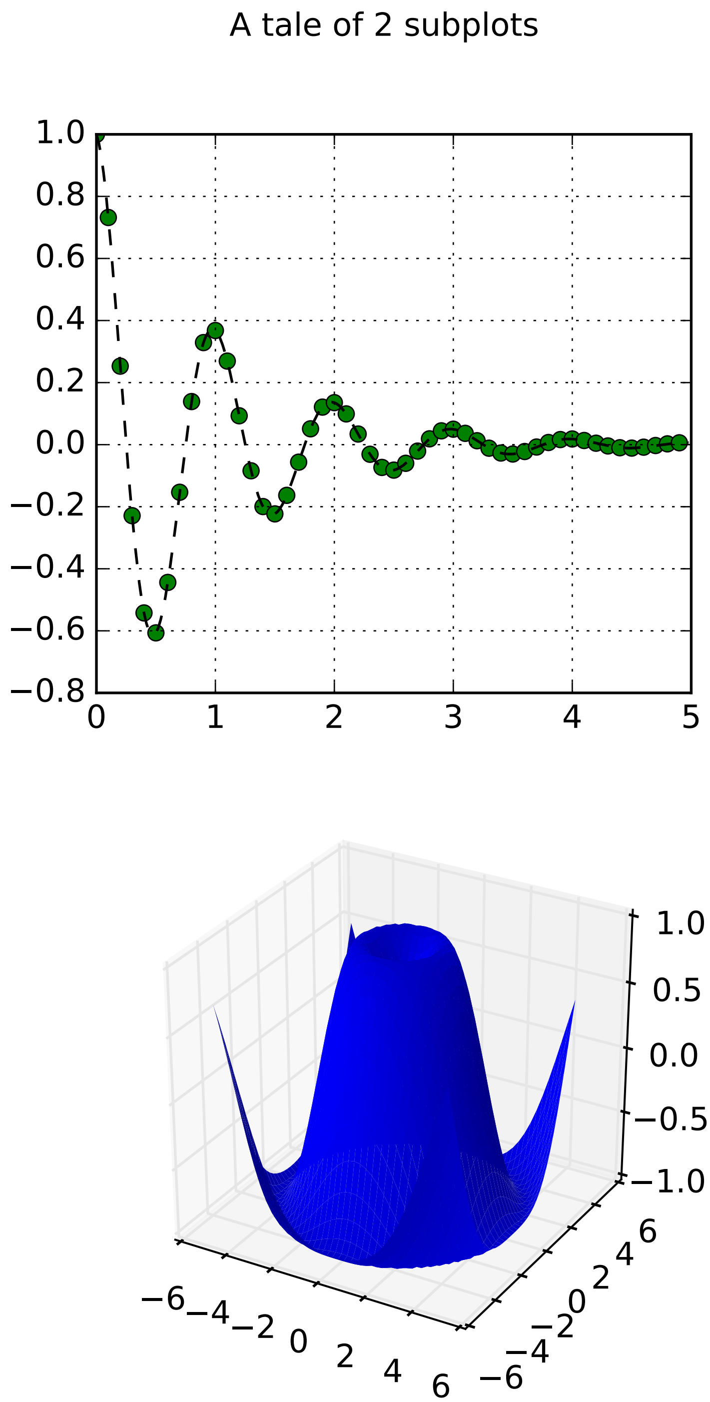


33.1.14 Subplotting

Having multiple 3D plots in a single figure is the same as it is for 2D plots. Also, you can have both 2D and 3D plots in the same figure.

New in version 1.0.0: Subplotting 3D plots was added in v1.0.0. Earlier version can not do this.


1.0
0.5
0.0
−0.5
−1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−6 −4 −2 0 2 4 6
−6 −4 −2 0 2 4 6
100
80
60
40
20
0
−20
−40
−60
−80
−30 −20 −10 0 10 20 30
−30 −20 −10 0 10 20 30



33.2 mplot3d API

Contents

33.2.1 axes3d

Note: Significant effort went into bringing axes3d to feature-parity with regular axes objects for version 1.1.0. However, more work remains. Please report any functions that do not behave as expected as a bug. In addition, help and patches would be greatly appreciated!

Module containing Axes3D, an object which can plot 3D objects on a 2D matplotlib figure.

class `mpl_toolkits.mplot3d.axes3d.`**`Axes3D`**(*fig*, *rect=None*, **args*, ***kwargs*)

Bases: `matplotlib.axes._axes.Axes`

3D axes object.

`add_collection3d`(*col*, *zs=0*, *zdir='z'*)

Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

Supported are:
- PolyCollection
- LineColleciton
- PatchCollection

`add_contour_set`(*cset*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*)

`add_contourf_set`(*cset*, *zdir='z'*, *offset=None*)

`auto_scale_xyz`(*X*, *Y*, *Z=None*, *had_data=None*)

`autoscale`(*enable=True*, *axis='both'*, *tight=None*)

Convenience method for simple axis view autoscaling. See `matplotlib.axes.Axes.autoscale()` for full explanation. Note that this function behaves the same, but for all three axes. Therfore, 'z' can be passed for *axis*, and 'both' applies to all three axes.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

autoscale_view(*tight=None*, *scalex=True*, *scaley=True*, *scalez=True*)

Autoscale the view limits using the data limits. See `matplotlib.axes.Axes.autoscale_view()` for documentation. Note that this function applies to the 3D axes, and as such adds the *scalez* to the function arguments.

Changed in version 1.1.0: Function signature was changed to better match the 2D version. *tight* is now explicitly a kwarg and placed first.

Changed in version 1.2.1: This is now fully functional.

bar(*left*, *height*, *zs=0*, *zdir='z'*, **args*, ***kwargs*)

Add 2D bar(s).

Argument	Description
left	The x coordinates of the left sides of the bars.
height	The height of the bars.
zs	Z coordinate of bars, if one value is specified they will all be placed at the same z.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`

bar3d(*x*, *y*, *z*, *dx*, *dy*, *dz*, *color='b'*, *zsort='average'*, **args*, ***kwargs*)

Generate a 3D bar, or multiple bars.

When generating multiple bars, x, y, z have to be arrays. dx, dy, dz can be arrays or scalars.

color can be:

- A single color value, to color all bars the same color.
- An array of colors of length N bars, to color each bar independently.
- An array of colors of length 6, to color the faces of the bars similarly.
- An array of colors of length 6 * N bars, to color each face independently.

When coloring the faces of the boxes specifically, this is the order of the coloring:

1. -Z (bottom of box)
2. +Z (top of box)
3. -Y
4. +Y
5. -X
6. +X

Keyword arguments are passed onto `Poly3DCollection()`

can_pan()

Return *True* if this axes supports the pan/zoom button functionality.

3D axes objects do not use the pan/zoom button.

can_zoom()

Return *True* if this axes supports the zoom box button functionality.

3D axes objects do not use the zoom box button.

cla()
Clear axes

clabel(**args*, ***kwargs*)
This function is currently not implemented for 3D axes. Returns *None*.

contour(*X*, *Y*, *Z*, **args*, ***kwargs*)
Create a 3D contour plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

The positional and other keyword arguments are passed on to `contour()`

Returns a `contour`

contour3D(*X*, *Y*, *Z*, **args*, ***kwargs*)
Create a 3D contour plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

The positional and other keyword arguments are passed on to `contour()`

Returns a `contour`

contourf(*X*, *Y*, *Z*, **args*, ***kwargs*)
Create a 3D contourf plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the filled contour on this position in plane normal to zdir

The positional and keyword arguments are passed on to `contourf()`

Returns a `contourf`

Changed in version 1.1.0: The *zdir* and *offset* kwargs were added.

contourf3D(*X*, *Y*, *Z*, **args*, ***kwargs*)
Create a 3D contourf plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the filled contour on this position in plane normal to zdir

The positional and keyword arguments are passed on to `contourf()`

Returns a `contourf`

Changed in version 1.1.0: The *zdir* and *offset* kwargs were added.

convert_zunits(*z*)
For artists in an axes, if the zaxis has units support, convert *z* using zaxis unit type

New in version 1.2.1.

disable_mouse_rotation()
Disable mouse button callbacks.

draw(*renderer*)

format_coord(*xd*, *yd*)
Given the 2D view coordinates attempt to guess a 3D coordinate. Looks for the nearest edge to the point and then assumes that the point is at the same z location as the nearest point on the edge.

format_zdata(*z*)
Return *z* string formatted. This function will use the `fmt_zdata` attribute if it is callable, else will fall back on the zaxis major formatter

get_autoscale_on()
Get whether autoscaling is applied for all axes on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_autoscalez_on()
Get whether autoscaling for the z-axis is applied on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_axis_position()

get_axisbelow()
Get whether axis below is true or not.

For axes3d objects, this will always be *True*

New in version 1.1.0: This function was added for completeness.

get_children()

get_frame_on()
Get whether the 3D axes panels are drawn

New in version 1.1.0.

get_proj()
Create the projection matrix from the current viewing position.

elev stores the elevation angle in the z plane azim stores the azimuth angle in the x,y plane

dist is the distance of the eye viewing point from the object point.

get_w_lims()
Get 3D world limits.

get_xlim()
Get the x-axis range [*left*, *right*]

Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_xlim3d()
Get the x-axis range [*left*, *right*]

Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_ylim()
Get the y-axis range [*bottom*, *top*]

Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_ylim3d()
Get the y-axis range [*bottom*, *top*]

Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_zbound()
Returns the z-axis numerical bounds where:

```
lowerBound < upperBound
```

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_zlabel()
Get the z-label text string.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_zlim()
Get 3D z limits.

get_zlim3d()
Get 3D z limits.

get_zmajorticklabels()
Get the ztick labels as a list of Text instances

New in version 1.1.0.

get_zminorticklabels()
Get the ztick labels as a list of Text instances

Note: Minor ticks are not supported. This function was added only for completeness.

New in version 1.1.0.

get_zscale()

get_zticklabels(*minor=False*)
Get ztick labels as a list of Text instances. See `matplotlib.axes.Axes.get_yticklabels()` for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

get_zticklines()
Get ztick lines as a list of Line2D instances. Note that this function is provided merely for completeness. These lines are re-calculated as the display changes.

New in version 1.1.0.

get_zticks(*minor=False*)
Return the z ticks as a list of locations See `matplotlib.axes.Axes.get_yticks()` for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

grid(*b=True*, ***kwargs*)
Set / unset 3D grid.

Note: Currently, this function does not behave the same as `matplotlib.axes.Axes.grid()`, but it is intended to eventually support that behavior.

Changed in version 1.1.0: This function was changed, but not tested. Please report any bugs.

have_units()
Return *True* if units are set on the *x*, *y*, or *z* axes

invert_zaxis()

Invert the z-axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

locator_params(*axis='both'*, *tight=None*, ***kwargs*)

Convenience method for controlling tick locators.

See `matplotlib.axes.Axes.locator_params()` for full documentation Note that this is for Axes3D objects, therefore, setting *axis* to 'both' will result in the parameters being set for all three axes. Also, *axis* can also take a value of 'z' to apply parameters to the z axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

margins(**args*, ***kw*)

Convenience method to set or retrieve autoscaling margins.
signatures:: margins()
returns xmargin, ymargin, zmargin

```
margins(margin)

margins(xmargin, ymargin, zmargin)

margins(x=xmargin, y=ymargin, z=zmargin)

margins(..., tight=False)
```

All forms above set the xmargin, ymargin and zmargin parameters. All keyword parameters are optional. A single argument specifies xmargin, ymargin and zmargin. The *tight* parameter is passed to `autoscale_view()`, which is executed after a margin is changed; the default here is *True*, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting *tight* to *None* will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if *xmargin* is not None, then *xmargin* times the X data interval will be added to each end of that interval before it is used in autoscaling.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

mouse_init(*rotate_btn=1*, *zoom_btn=3*)

Initializes mouse button callbacks to enable 3D rotation of the axes. Also optionally sets the mouse buttons for 3D rotation and zooming.

Argument	Description
rotate_btn	The integer or list of integers specifying which mouse button or buttons to use for 3D rotation of the axes. Default = 1.
zoom_btn	The integer or list of integers specifying which mouse button or buttons to use to zoom the 3D axes. Default = 3.

name = '3d'

plot(*xs*, *ys*, **args*, ***kwargs*)
Plot 2D or 3D data.

Argument	Description
xs, *ys*	x, y coordinates of vertices
zs	z value(s), either one for all points or one for each point.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Other arguments are passed on to `plot()`

plot3D(*xs*, *ys*, **args*, ***kwargs*)
Plot 2D or 3D data.

Argument	Description
xs, *ys*	x, y coordinates of vertices
zs	z value(s), either one for all points or one for each point.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Other arguments are passed on to `plot()`

plot_surface(*X*, *Y*, *Z*, **args*, ***kwargs*)
Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the *cmap* argument.

The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If 1k by 1k arrays are passed in the default values for the strides will result in a 100x100 grid being plotted.

Argument	Description
X, *Y*, *Z*	Data values as 2D arrays
rstride	Array row stride (step size), defaults to 10
cstride	Array column stride (step size), defaults to 10
color	Color of the surface patches
cmap	A colormap for the surface patches.
facecolors	Face colors for the individual patches
norm	An instance of Normalize to map values to colors
vmin	Minimum value to map
vmax	Maximum value to map
shade	Whether to shade the facecolors

Other arguments are passed on to `Poly3DCollection`

plot_trisurf(**args*, ***kwargs*)

Argument	Description
X, *Y*, *Z*	Data values as 1D arrays
color	Color of the surface patches
cmap	A colormap for the surface patches.
norm	An instance of Normalize to map values to colors
vmin	Minimum value to map
vmax	Maximum value to map
shade	Whether to shade the facecolors

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a *Triangulation* object, or:

```
plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

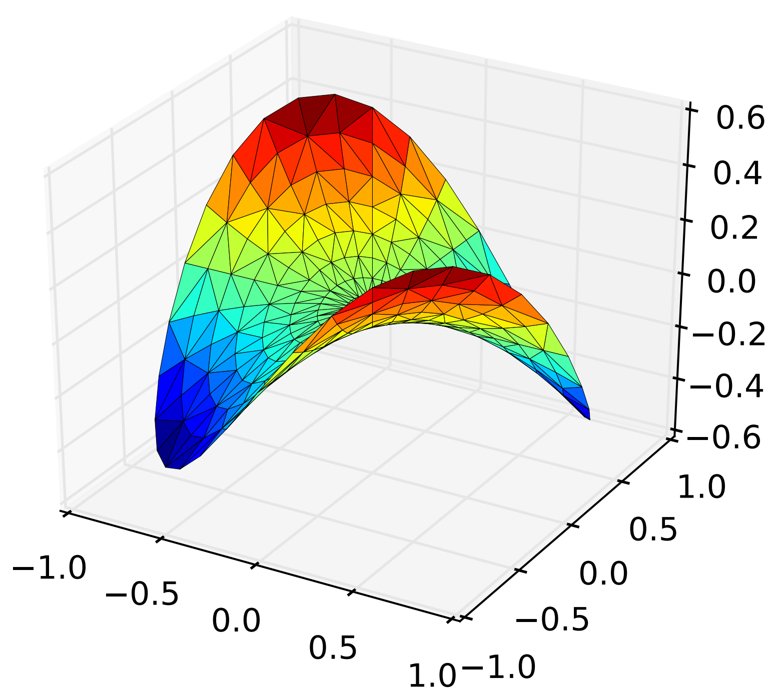
The remaining arguments are:

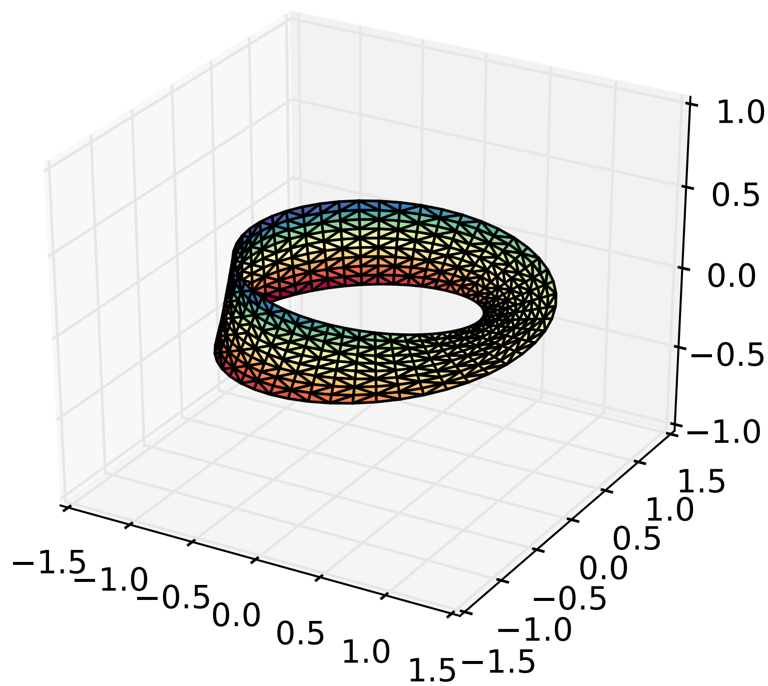
```
plot_trisurf(..., Z)
```

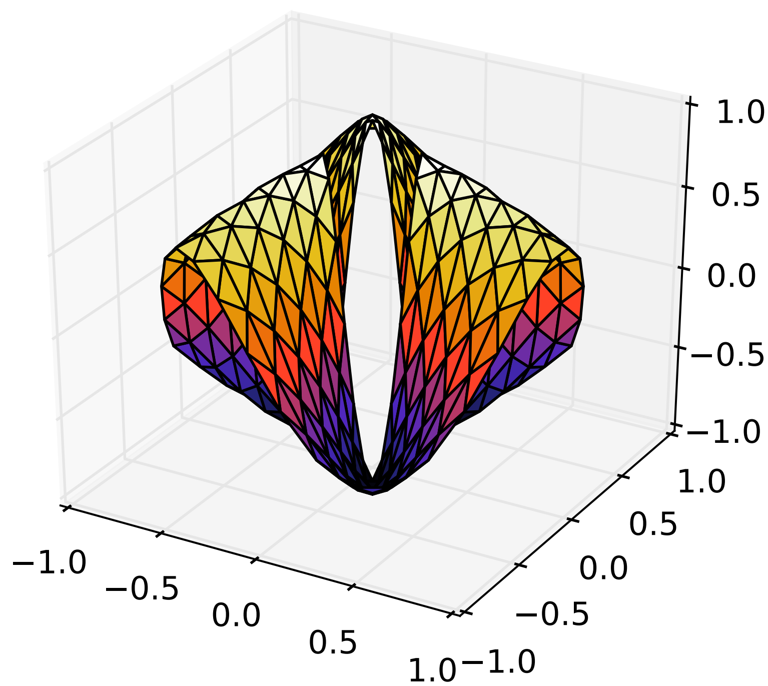
where *Z* is the array of values to contour, one per point in the triangulation.

Other arguments are passed on to *Poly3DCollection*

Examples:




1.0
0.5
0.0
−0.5
−1.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5



New in version 1.2.0: This plotting function was added for the v1.2.0 release.

plot_wireframe(*X*, *Y*, *Z*, **args*, ***kwargs*)
Plot a 3D wireframe.

The `rstride` and `cstride` kwargs set the stride used to sample the input data to generate the graph. If either is 0 the input data in not sampled along this direction producing a 3D line plot rather than a wireframe plot.

Argument	Description
X, *Y*,	Data values as 2D arrays
Z	
rstride	Array row stride (step size), defaults to 1
cstride	Array column stride (step size), defaults to 1

Keyword arguments are passed on to *LineCollection*.

Returns a *Line3DCollection*

quiver(**args*, ***kwargs*)
Plot a 3D field of arrows.

call signatures:

```
quiver(X, Y, Z, U, V, W, **kwargs)
```

Arguments:

***X*, *Y*, *Z*:** The x, y and z coordinates of the arrow locations (default is tip of arrow; see *pivot* kwarg)

***U*, *V*, *W*:** The x, y and z components of the arrow vectors

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

***length*: [1.0 | float]** The length of each quiver, default to 1.0, the unit is the same with the axes

***arrow_length_ratio*: [0.3 | float]** The ratio of the arrow head with respect to the quiver, default to 0.3

***pivot*: ['tail' | 'middle' | 'tip']** The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

Any additional keyword arguments are delegated to `LineCollection`

quiver3D(**args*, ***kwargs*)

Plot a 3D field of arrows.

call signatures:

```
quiver(X, Y, Z, U, V, W, **kwargs)
```

Arguments:

***X*, *Y*, *Z*:** The x, y and z coordinates of the arrow locations (default is tip of arrow; see *pivot* kwarg)

***U*, *V*, *W*:** The x, y and z components of the arrow vectors

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

***length*: [1.0 | float]** The length of each quiver, default to 1.0, the unit is the same with the axes

***arrow_length_ratio*: [0.3 | float]** The ratio of the arrow head with respect to the quiver, default to 0.3

***pivot*: ['tail' | 'middle' | 'tip']** The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

Any additional keyword arguments are delegated to `LineCollection`

scatter(*xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c=None*, *depthshade=True*, **args*, ***kwargs*)

Create a scatter plot.

Argument	Description
xs, *ys*	Positions of data points.
zs	Either an array of the same length as *xs* and *ys* or a single value to place all points in the same plane. Default is 0.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.
s	Size in points^2. It is a scalar or an array of the same length as *x* and *y*.
c	A color. *c* can be a single color format string, or a sequence of color specifications of length *N*, or a sequence of *N* numbers to be mapped to colors using the *cmap* and *norm* specified via kwargs (see below). Note that *c* should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. *c* can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.
depthshade	Whether or not to shade the scatter markers to give the appearance of depth. Default is *True*.

Keyword arguments are passed on to `scatter()`.

Returns a `Patch3DCollection`

scatter3D(*xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c=None*, *depthshade=True*, **args*, ***kwargs*)
Create a scatter plot.

Argument	Description
xs, *ys*	Positions of data points.
zs	Either an array of the same length as *xs* and *ys* or a single value to place all points in the same plane. Default is 0.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.
s	Size in points^2. It is a scalar or an array of the same length as *x* and *y*.
c	A color. *c* can be a single color format string, or a sequence of color specifications of length *N*, or a sequence of *N* numbers to be mapped to colors using the *cmap* and *norm* specified via kwargs (see below). Note that *c* should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. *c* can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.
depthshade	Whether or not to shade the scatter markers to give the appearance of depth. Default is *True*.

Keyword arguments are passed on to `scatter()`.

Returns a `Patch3DCollection`

set_autoscale_on(*b*)
Set whether autoscaling is applied on plot commands

accepts: [*True* | *False*]

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_autoscalez_on(*b*)
Set whether autoscaling for the z-axis is applied on plot commands

accepts: [*True* | *False*]

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_axis_off()

set_axis_on()

set_axisbelow(*b*)

Set whether the axis ticks and gridlines are above or below most artists

For axes3d objects, this will ignore any settings and just use *True*

ACCEPTS: [*True* | *False*]

New in version 1.1.0: This function was added for completeness.

set_frame_on(*b*)

Set whether the 3D axes panels are drawn

ACCEPTS: [*True* | *False*]

New in version 1.1.0.

set_title(*label*, *fontdict=None*, *loc='center'*, ***kwargs*)

Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

Parameters label : str

Text to use for the title

fontdict : dict

A dictionary controlling the appearance of the title text, the default `fontdict` is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight' : rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

loc : {'center', 'left', 'right'}, str, optional

Which title to set, defaults to 'center'

Returns text : `Text`

The matplotlib text instance representing the title

Other Parameters kwargs : text properties

Other keyword arguments are text properties, see `Text` for a list of valid text properties.

set_top_view()

set_xlim(*left=None*, *right=None*, *emit=True*, *auto=False*, ***kw*)

Set 3D x limits.

See `matplotlib.axes.Axes.set_xlim()` for full documentation.

set_xlim3d(*left=None*, *right=None*, *emit=True*, *auto=False*, ***kw*)
Set 3D x limits.

See `matplotlib.axes.Axes.set_xlim()` for full documentation.

set_xscale(*value*, ***kwargs*)
Call signature:

```
set_xscale(value)
```

Set the scaling of the x-axis: 'linear' | 'log' | 'logit' | 'symlog'

ACCEPTS: ['linear' | 'log' | 'logit' | 'symlog']
Different kwargs are accepted, depending on the scale: 'linear'

'log'

***basex/basey*:** The base of the logarithm

***nonposx/nonposy*: ['mask' | 'clip']** non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

'logit'

***nonpos*: ['mask' | 'clip']** values beyond]0, 1[can be masked as invalid, or clipped to a number very close to 0 or 1

'symlog'

***basex/basey*:** The base of the logarithm

***linthreshx/linthreshy*:** The range (-*x*, *x*) within which the plot is linear (to avoid having the plot go to infinity around zero).

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

***linscalex/linscaley*:** This allows the linear range (-*linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_ylim(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***kw*)
Set 3D y limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation.

set_ylim3d(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***kw*)
Set 3D y limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation.

set_yscale(*value*, ***kwargs*)
Call signature:

```
set_yscale(value)
```

Set the scaling of the y-axis: 'linear' | 'log' | 'logit' | 'symlog'

ACCEPTS: ['linear' | 'log' | 'logit' | 'symlog']
Different kwargs are accepted, depending on the scale: 'linear'

'log'

***basex/basey*:** The base of the logarithm

***nonposx/nonposy*: ['mask' | 'clip']** non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

'logit'

***nonpos*: ['mask' | 'clip']** values beyond]0, 1[can be masked as invalid, or clipped to a number very close to 0 or 1

'symlog'

***basex/basey*:** The base of the logarithm

***linthreshx/linthreshy*:** The range (-*x*, *x*) within which the plot is linear (to avoid having the plot go to infinity around zero).

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

***linscalex/linscaley*:** This allows the linear range (-*linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zbound(*lower=None*, *upper=None*)
Set the lower and upper numerical bounds of the z-axis. This method will honor axes inversion

regardless of parameter order. It will not change the `_autoscaleZon` attribute.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zlabel(*zlabel*, *fontdict=None*, *labelpad=None*, ***kwargs*)
Set zlabel. See doc for `set_ylabel()` for description.

set_zlim(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***kw*)
Set 3D z limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation

set_zlim3d(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***kw*)
Set 3D z limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation

set_zmargin(*m*)
Set padding of Z data limits prior to autoscaling.

m times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zscale(*value*, ***kwargs*)
call signature:

```
set_zscale(value)
```

Set the scaling of the z-axis: 'linear' | 'log' | 'logit' | 'symlog'

ACCEPTS: ['linear' | 'log' | 'logit' | 'symlog']
Different kwargs are accepted, depending on the scale: 'linear'

'log'

***basex/basey*: ** The base of the logarithm
***nonposx/nonposy*: ['mask' | 'clip']** non-positive values in x or y can be masked as invalid, or clipped to a very small positive number
***subsx/subsy*: ** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

'logit'

***nonpos*: ['mask' | 'clip']** values beyond]0, 1[can be masked as invalid, or clipped to a number very close to 0 or 1

'symlog'

***basex/basey*: ** The base of the logarithm
***linthreshx/linthreshy*: ** The range (-*x*, *x*) within which the plot is linear (to avoid having the plot go to infinity around zero).

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

***linscalex/linscaley*:** This allows the linear range (*-linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

Note: Currently, Axes3D objects only supports linear scales. Other scales may or may not work, and support for these is improving with each release.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zticklabels(**args*, ***kwargs*)
Set z-axis tick labels. See `matplotlib.axes.Axes.set_yticklabels()` for more details.

Note: Minor ticks are not supported by Axes3D objects.

New in version 1.1.0.

set_zticks(**args*, ***kwargs*)
Set z-axis tick locations. See `matplotlib.axes.Axes.set_yticks()` for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

text(*x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)
Add text to the plot. kwargs will be passed on to Axes.text, except for the `zdir` keyword, which sets the direction to be used as the z direction.

text2D(*x*, *y*, *s*, *fontdict=None*, *withdash=False*, ***kwargs*)
Add text to the axes.

Add text in string `s` to axis at location `x`, `y`, data coordinates.

Parameters x, y : scalars
data coordinates

s : string
text

fontdict : dictionary, optional, default: None
A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.

withdash : boolean, optional, default: False

Creates a *TextWithDash* instance instead of a *Text* instance.

Other Parameters kwargs : *Text* properties.

Other miscellaneous text parameters.

Examples

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the axes:

```
>>> text(0.5, 0.5,'matplotlib', horizontalalignment='center',
...      verticalalignment='center',
...      transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword `bbox`. `bbox` is a dictionary of *Rectangle* properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

text3D(*x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)

Add text to the plot. kwargs will be passed on to Axes.text, except for the `zdir` keyword, which sets the direction to be used as the z direction.

tick_params(*axis='both'*, ***kwargs*)

Convenience method for changing the appearance of ticks and tick labels.

See *matplotlib.axes.Axes.tick_params()* for more complete documentation.

The only difference is that setting *axis* to 'both' will mean that the settings are applied to all three axes. Also, the *axis* parameter also accepts a value of 'z', which would mean to apply to only the z-axis.

Also, because of how Axes3D objects are drawn very differently from regular 2D axes, some of these settings may have ambiguous meaning. For simplicity, the 'z' axis will accept settings as if it was like the 'y' axis.

Note: While this function is currently implemented, the core part of the Axes3D object may ignore some of these settings. Future releases will fix this. Priority will be given to those who file bugs.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

ticklabel_format(***kwargs*)

Convenience method for manipulating the ScalarFormatter used by default for linear axes in Axed3D objects.

See `matplotlib.axes.Axes.ticklabel_format()` for full documentation. Note that this version applies to all three axes of the Axes3D object. Therefore, the *axis* argument will also accept a value of 'z' and the value of 'both' will apply to all three axes.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

tricontour(**args*, ***kwargs*)

Create a 3D contour plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

Other keyword arguments are passed on to `tricontour()`

Returns a `contour`

Changed in version 1.3.0: Added support for custom triangulations

EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

tricontourf(**args*, ***kwargs*)

Create a 3D contourf plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

Other keyword arguments are passed on to `tricontour()`

Returns a `contour`

Changed in version 1.3.0: Added support for custom triangulations

EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

tunit_cube(*vals=None*, *M=None*)

tunit_edges(*vals=None*, *M=None*)

unit_cube(*vals=None*)

update_datalim(*xys*, ***kwargs*)

view_init(*elev=None*, *azim=None*)
Set the elevation and azimuth of the axes.

This can be used to rotate the axes programatically.

'elev' stores the elevation angle in the z plane. 'azim' stores the azimuth angle in the x,y plane.

if elev or azim are None (default), then the initial value is used which was specified in the *Axes3D* constructor.

zaxis_date(*tz=None*)
Sets up z-axis ticks and labels that treat the z data as dates.

tz is a timezone string or `tzinfo` instance. Defaults to rc value.

Note: This function is merely provided for completeness. Axes3D objects do not officially support dates for ticks, and so this may or may not work as expected.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

zaxis_inverted()
Returns True if the z-axis is inverted.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

mpl_toolkits.mplot3d.axes3d.**get_test_data**(*delta=0.05*)
Return a tuple X, Y, Z with a test data set.

mpl_toolkits.mplot3d.axes3d.**unit_bbox**()

33.2.2 axis3d

Note: Historically, axis3d has suffered from having hard-coded constants controlling the look and feel of the 3D plot. This precluded user level adjustments such as label spacing, font colors and panel colors. For version 1.1.0, these constants have been consolidated into a single private member dictionary, `self._axinfo`, for the axis object. This is intended only as a stop-gap measure to allow user-level customization, but it is not intended to be permanent.

class mpl_toolkits.mplot3d.axis3d.**Axis**(*adir*, *v_intervalx*, *d_intervalx*, *axes*, **args*, ***kwargs*)
Bases: *matplotlib.axis.XAxis*

draw(*renderer*)

draw_pane(*renderer*)

get_major_ticks(*numticks=None*)

get_rotate_label(*text*)

get_tick_positions()

get_tightbbox(*renderer*)

get_view_interval()
return the Interval instance for this 3d axis view limits

init3d()

set_pane_color(*color*)
Set pane color to a RGBA tuple

set_pane_pos(*xys*)

set_rotate_label(*val*)
Whether to rotate the axis label: True, False or None. If set to None the label will be rotated if longer than 4 chars.

set_view_interval(*vmin*, *vmax*, *ignore=False*)

class mpl_toolkits.mplot3d.axis3d.**XAxis**(*adir*, *v_intervalx*, *d_intervalx*, *axes*, **args*, ***kwargs*)
Bases: *mpl_toolkits.mplot3d.axis3d.Axis*

get_data_interval()
return the Interval instance for this axis data limits

class mpl_toolkits.mplot3d.axis3d.**YAxis**(*adir*, *v_intervalx*, *d_intervalx*, *axes*, **args*, ***kwargs*)
Bases: *mpl_toolkits.mplot3d.axis3d.Axis*

get_data_interval()
return the Interval instance for this axis data limits

class mpl_toolkits.mplot3d.axis3d.**ZAxis**(*adir*, *v_intervalx*, *d_intervalx*, *axes*, **args*, ***kwargs*)
Bases: *mpl_toolkits.mplot3d.axis3d.Axis*

get_data_interval()
return the Interval instance for this axis data limits

mpl_toolkits.mplot3d.axis3d.**get_flip_min_max**(*coord*, *index*, *mins*, *maxs*)

mpl_toolkits.mplot3d.axis3d.**move_from_center**(*coord*, *centers*, *deltas*, *axmask=(True, True, True)*)
Return a coordinate that is moved by "deltas" away from the center.

mpl_toolkits.mplot3d.axis3d.**tick_update_position**(*tick*, *tickxs*, *tickys*, *labelpos*)
Update tick line and label position and style.

33.2.3 art3d

Module containing 3D artist code and functions to convert 2D artists into 3D versions which can be added to an Axes3D.

class mpl_toolkits.mplot3d.art3d.**Line3D**(*xs*, *ys*, *zs*, **args*, ***kwargs*)
Bases: `matplotlib.lines.Line2D`

3D line object.

Keyword arguments are passed onto `Line2D()`.

draw(*renderer*)

set_3d_properties(*zs=0*, *zdir='z'*)

class mpl_toolkits.mplot3d.art3d.**Line3DCollection**(*segments*, **args*, ***kwargs*)
Bases: `matplotlib.collections.LineCollection`

A collection of 3D lines.

Keyword arguments are passed onto `LineCollection()`.

do_3d_projection(*renderer*)
Project the points according to renderer matrix.

draw(*renderer*, *project=False*)

set_segments(*segments*)
Set 3D segments

set_sort_zpos(*val*)
Set the position to use for z-sorting.

class mpl_toolkits.mplot3d.art3d.**Patch3D**(**args*, ***kwargs*)
Bases: `matplotlib.patches.Patch`

3D patch object.

do_3d_projection(*renderer*)

draw(*renderer*)

get_facecolor()

get_path()

set_3d_properties(*verts*, *zs=0*, *zdir='z'*)

class mpl_toolkits.mplot3d.art3d.**Patch3DCollection**(**args*, ***kwargs*)
Bases: `matplotlib.collections.PatchCollection`

A collection of 3D patches.

Create a collection of flat 3D patches with its normal vector pointed in *zdir* direction, and located at *zs* on the *zdir* axis. 'zs' can be a scalar or an array-like of the same length as the number of patches in the collection.

Constructor arguments are the same as for `PatchCollection`. In addition, keywords *zs=0* and *zdir='z'* are available.

Also, the keyword argument "depthshade" is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is *True*). This is typically desired in scatter plots.

do_3d_projection(*renderer*)

set_3d_properties(*zs*, *zdir*)

set_sort_zpos(*val*)
Set the position to use for z-sorting.

class mpl_toolkits.mplot3d.art3d.**Path3DCollection**(**args*, ***kwargs*)
Bases: `matplotlib.collections.PathCollection`

A collection of 3D paths.

Create a collection of flat 3D paths with its normal vector pointed in *zdir* direction, and located at *zs* on the *zdir* axis. 'zs' can be a scalar or an array-like of the same length as the number of paths in the collection.

Constructor arguments are the same as for `PathCollection`. In addition, keywords *zs=0* and *zdir='z'* are available.

Also, the keyword argument "depthshade" is available to indicate whether or not to shade the patches in order to give the appearance of depth (default is *True*). This is typically desired in scatter plots.

do_3d_projection(*renderer*)

set_3d_properties(*zs*, *zdir*)

set_sort_zpos(*val*)
Set the position to use for z-sorting.

class mpl_toolkits.mplot3d.art3d.**PathPatch3D**(*path*, ***kwargs*)

Bases: `mpl_toolkits.mplot3d.art3d.Patch3D`

3D PathPatch object.

do_3d_projection(*renderer*)

set_3d_properties(*path*, *zs=0*, *zdir='z'*)

class mpl_toolkits.mplot3d.art3d.**Poly3DCollection**(*verts*, **args*, ***kwargs*)

Bases: `matplotlib.collections.PolyCollection`

A collection of 3D polygons.

Create a Poly3DCollection.

verts should contain 3D coordinates.

Keyword arguments: zsort, see set_zsort for options.

Note that this class does a bit of magic with the _facecolors and _edgecolors properties.

do_3d_projection(*renderer*)

Perform the 3D projection for this object.

draw(*renderer*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_vector(*segments3d*)

Optimize points for projection

set_3d_properties()

set_alpha(*alpha*)

Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_edgecolor(*colors*)

set_edgecolors(*colors*)

set_facecolor(*colors*)

set_facecolors(*colors*)

set_sort_zpos(*val*)
Set the position to use for z-sorting.

set_verts(*verts*, *closed=True*)
Set 3D vertices.

set_verts_and_codes(*verts*, *codes*)
Sets 3D vertices with path codes

set_zsort(*zsort*)

Set z-sorting behaviour: boolean: if True use default 'average' string: 'average', 'min' or 'max'

class `mpl_toolkits.mplot3d.art3d.`**Text3D**(*x=0*, *y=0*, *z=0*, *text=''*, *zdir='z'*, ***kwargs*)
Bases: `matplotlib.text.Text`

Text object with 3D position and (in the future) direction.

x, *y*, *z* Position of text *text* Text string to display *zdir* Direction of text

Keyword arguments are passed onto `Text()`.

draw(*renderer*)

set_3d_properties(*z=0*, *zdir='z'*)

`mpl_toolkits.mplot3d.art3d.`**get_colors**(*c*, *num*)
Stretch the color argument to provide the required number num

`mpl_toolkits.mplot3d.art3d.`**get_dir_vector**(*zdir*)

`mpl_toolkits.mplot3d.art3d.`**get_patch_verts**(*patch*)
Return a list of vertices for the path of a patch.

`mpl_toolkits.mplot3d.art3d.`**iscolor**(*c*)

`mpl_toolkits.mplot3d.art3d.`**juggle_axes**(*xs*, *ys*, *zs*, *zdir*)
Reorder coordinates so that 2D xs, ys can be plotted in the plane orthogonal to zdir. zdir is normally x, y or z. However, if zdir starts with a '-' it is interpreted as a compensation for rotate_axes.

`mpl_toolkits.mplot3d.art3d.`**line_2d_to_3d**(*line*, *zs=0*, *zdir='z'*)
Convert a 2D line to 3D.

`mpl_toolkits.mplot3d.art3d.`**line_collection_2d_to_3d**(*col*, *zs=0*, *zdir='z'*)
Convert a LineCollection to a Line3DCollection object.

`mpl_toolkits.mplot3d.art3d.`**`norm_angle`**(*a*)
Return angle between -180 and +180

`mpl_toolkits.mplot3d.art3d.`**`norm_text_angle`**(*a*)
Return angle between -90 and +90

`mpl_toolkits.mplot3d.art3d.`**`patch_2d_to_3d`**(*patch*, *z=0*, *zdir='z'*)
Convert a Patch to a Patch3D object.

`mpl_toolkits.mplot3d.art3d.`**`patch_collection_2d_to_3d`**(*col*, *zs=0*, *zdir='z'*, *depthshade=True*)
Convert a `PatchCollection` into a `Patch3DCollection` object (or a `PathCollection` into a `Path3DCollection` object).

Keywords:

za **The location or locations to place the patches in the** collection along the *zdir* axis. Defaults to 0.

zdir The axis in which to place the patches. Default is "z".

depthshade **Whether to shade the patches to give a sense of depth.** Defaults to *True*.

`mpl_toolkits.mplot3d.art3d.`**`path_to_3d_segment`**(*path*, *zs=0*, *zdir='z'*)
Convert a path to a 3D segment.

`mpl_toolkits.mplot3d.art3d.`**`path_to_3d_segment_with_codes`**(*path*, *zs=0*, *zdir='z'*)
Convert a path to a 3D segment with path codes.

`mpl_toolkits.mplot3d.art3d.`**`pathpatch_2d_to_3d`**(*pathpatch*, *z=0*, *zdir='z'*)
Convert a PathPatch to a PathPatch3D object.

`mpl_toolkits.mplot3d.art3d.`**`paths_to_3d_segments`**(*paths*, *zs=0*, *zdir='z'*)
Convert paths from a collection object to 3D segments.

`mpl_toolkits.mplot3d.art3d.`**`paths_to_3d_segments_with_codes`**(*paths*, *zs=0*, *zdir='z'*)
Convert paths from a collection object to 3D segments with path codes.

`mpl_toolkits.mplot3d.art3d.`**`poly_collection_2d_to_3d`**(*col*, *zs=0*, *zdir='z'*)
Convert a PolyCollection to a Poly3DCollection object.

`mpl_toolkits.mplot3d.art3d.`**`rotate_axes`**(*xs*, *ys*, *zs*, *zdir*)
Reorder coordinates so that the axes are rotated with zdir along the original z axis. Prepending the axis with a '-' does the inverse transform, so zdir can be x, -x, y, -y, z or -z

`mpl_toolkits.mplot3d.art3d.`**`text_2d_to_3d`**(*obj*, *z=0*, *zdir='z'*)
Convert a Text to a Text3D object.

`mpl_toolkits.mplot3d.art3d.`**`zalpha`**(*colors*, *zs*)
Modify the alphas of the color list according to depth

33.2.4 proj3d

Various transforms used for by the 3D code

`mpl_toolkits.mplot3d.proj3d.`**`inv_transform`**(*xs*, *ys*, *zs*, *M*)

mpl_toolkits.mplot3d.proj3d.**line2d**(*p0*, *p1*)
Return 2D equation of line in the form ax+by+c = 0

mpl_toolkits.mplot3d.proj3d.**line2d_dist**(*l*, *p*)
Distance from line to point line is a tuple of coefficients a,b,c

mpl_toolkits.mplot3d.proj3d.**line2d_seg_dist**(*p1*, *p2*, *p0*)
distance(s) from line defined by p1 - p2 to point(s) p0

p0[0] = x(s) p0[1] = y(s)

intersection point p = p1 + u*(p2-p1) and intersection point lies within segment if u is between 0 and 1

mpl_toolkits.mplot3d.proj3d.**mod**(*v*)
3d vector length

mpl_toolkits.mplot3d.proj3d.**persp_transformation**(*zfront*, *zback*)

mpl_toolkits.mplot3d.proj3d.**proj_points**(*points*, *M*)

mpl_toolkits.mplot3d.proj3d.**proj_trans_clip_points**(*points*, *M*)

mpl_toolkits.mplot3d.proj3d.**proj_trans_points**(*points*, *M*)

mpl_toolkits.mplot3d.proj3d.**proj_transform**(*xs*, *ys*, *zs*, *M*)
Transform the points by the projection matrix

mpl_toolkits.mplot3d.proj3d.**proj_transform_clip**(*xs*, *ys*, *zs*, *M*)
Transform the points by the projection matrix and return the clipping result returns txs,tys,tzs,tis

mpl_toolkits.mplot3d.proj3d.**proj_transform_vec**(*vec*, *M*)

mpl_toolkits.mplot3d.proj3d.**proj_transform_vec_clip**(*vec*, *M*)

mpl_toolkits.mplot3d.proj3d.**rot_x**(*V*, *alpha*)

mpl_toolkits.mplot3d.proj3d.**test_lines_dists**()

mpl_toolkits.mplot3d.proj3d.**test_proj**()

mpl_toolkits.mplot3d.proj3d.**test_proj_draw_axes**(*M*, *s=1*)

mpl_toolkits.mplot3d.proj3d.**test_proj_make_M**(*E=None*) `mpl_toolkits.mplot3d.proj3d.`**`test_rot`**`()`

`mpl_toolkits.mplot3d.proj3d.`**`test_world`**`()`

`mpl_toolkits.mplot3d.proj3d.`**`transform`**`(`*xs*, *ys*, *zs*, *M*`)`
Transform the points by the projection matrix

`mpl_toolkits.mplot3d.proj3d.`**`vec_pad_ones`**`(`*xs*, *ys*, *zs*`)`

`mpl_toolkits.mplot3d.proj3d.`**`view_transformation`**`(`*E*, *R*, *V*`)`

`mpl_toolkits.mplot3d.proj3d.`**`world_transformation`**`(`*xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*`)`

33.3 mplot3d FAQ

33.3.1 How is mplot3d different from MayaVi?

MayaVi2 is a very powerful and featureful 3D graphing library. For advanced 3D scenes and excellent rendering capabilities, it is highly recomended to use MayaVi2.

mplot3d was intended to allow users to create simple 3D graphs with the same "look-and-feel" as matplotlib's 2D plots. Furthermore, users can use the same toolkit that they are already familiar with to generate both their 2D and 3D plots.

33.3.2 My 3D plot doesn't look right at certain viewing angles

This is probably the most commonly reported issue with mplot3d. The problem is that – from some viewing angles – a 3D object would appear in front of another object, even though it is physically behind it. This can result in plots that do not look "physically correct."

Unfortunately, while some work is being done to reduce the occurance of this artifact, it is currently an intractable problem, and can not be fully solved until matplotlib supports 3D graphics rendering at its core.

The problem occurs due to the reduction of 3D data down to 2D + z-order scalar. A single value represents the 3rd dimension for all parts of 3D objects in a collection. Therefore, when the bounding boxes of two collections intersect, it becomes possible for this artifact to occur. Furthermore, the intersection of two 3D objects (such as polygons or patches) can not be rendered properly in matplotlib's 2D rendering engine.

This problem will likely not be solved until OpenGL support is added to all of the backends (patches are greatly welcomed). Until then, if you need complex 3D scenes, we recommend using MayaVi.

33.3.3 I don't like how the 3D plot is laid out, how do I change that?

Historically, mplot3d has suffered from a hard-coding of parameters used to control visuals such as label spacing, tick length, and grid line width. Work is being done to eliminate this issue. For matplotlib v1.1.0, there is a semi-official manner to modify these parameters. See the note in the *axis3d* section of the mplot3d API documentation for more information.

Part VIII

Toolkits

Toolkits are collections of application-specific functions that extend matplotlib.

CHAPTER

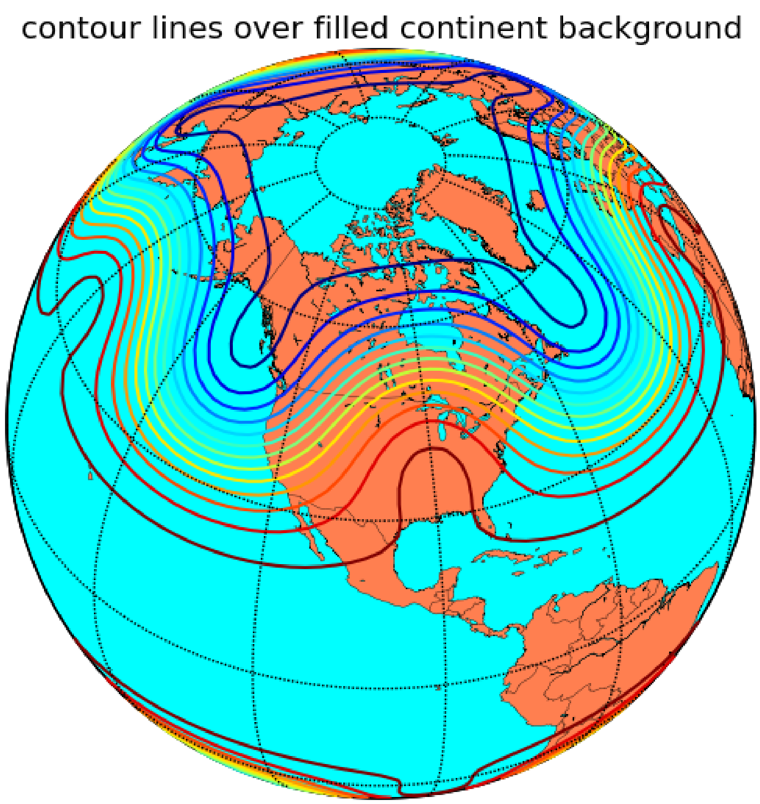
THIRTYFOUR

MAPPING TOOLKITS

34.1 Basemap

(*Not distributed with matplotlib*)

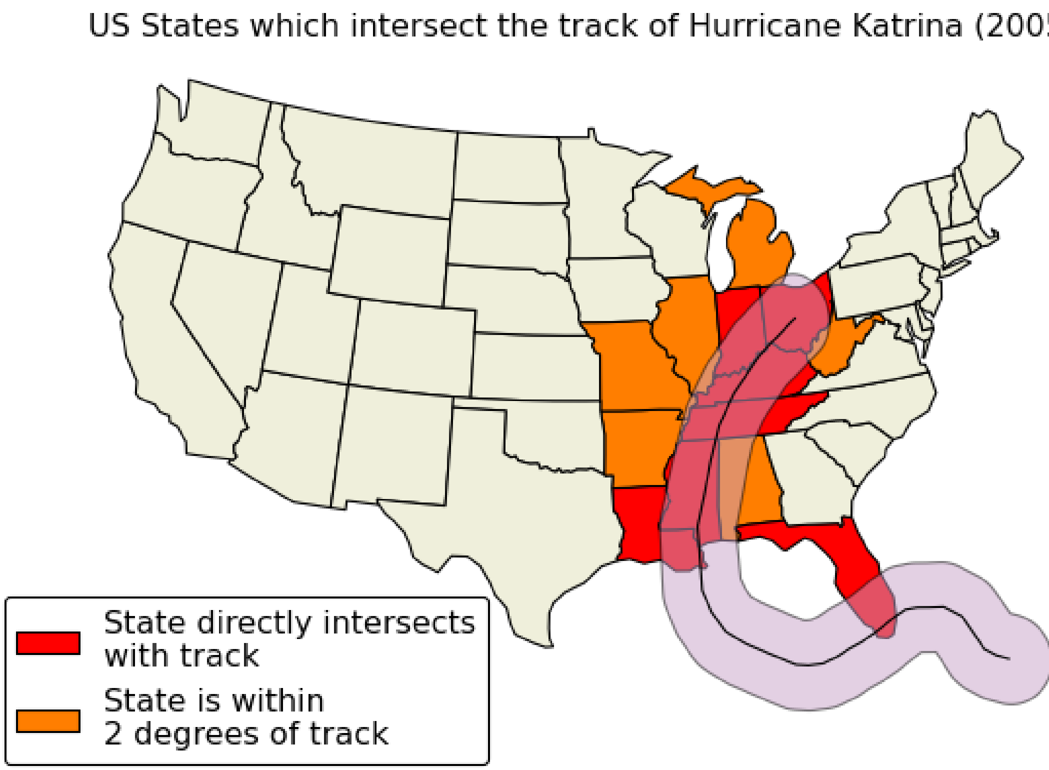
Plots data on map projections, with continental and political boundaries, see basemap docs.



34.2 Cartopy

(*Not distributed with matplotlib*)

An alternative mapping library written for matplotlib `v1.2` and beyond. Cartopy builds on top of matplotlib to provide object oriented map projection definitions and close integration with Shapely for powerful yet easy-to-use vector data processing tools. An example plot from the Cartopy gallery:



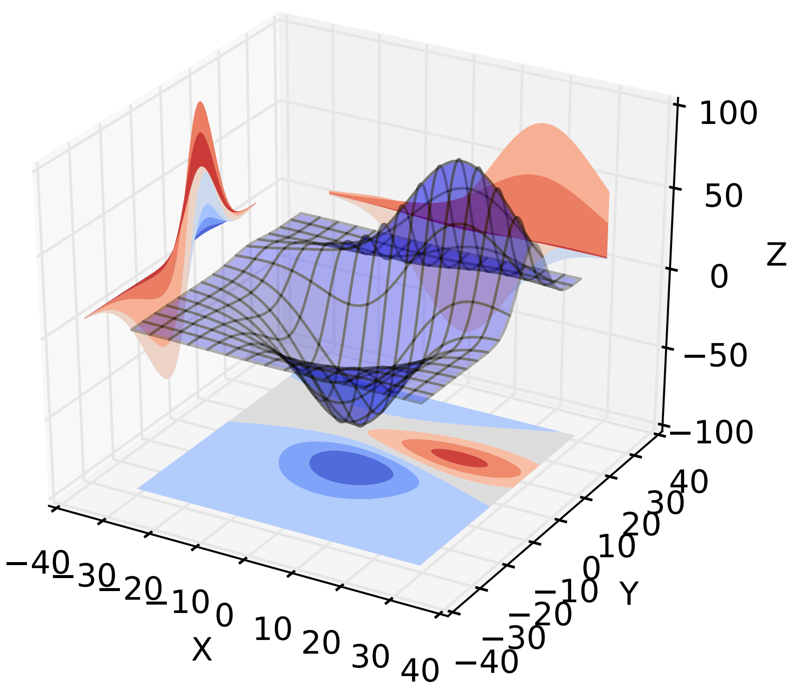
CHAPTER

THIRTYFIVE

GENERAL TOOLKITS

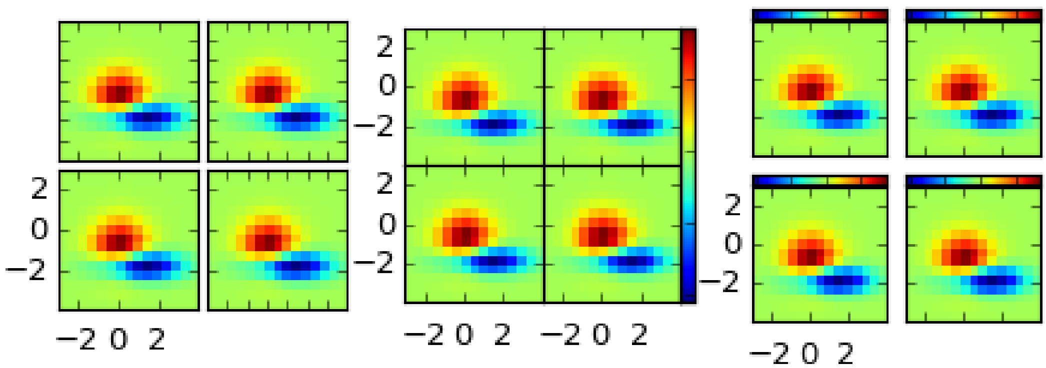
35.1 mplot3d

mpl_toolkits.mplot3d provides some basic 3D plotting (scatter, surf, line, mesh) tools. Not the fastest or feature complete 3D library out there, but ships with matplotlib and thus may be a lighter weight solution for some use cases.



35.2 AxesGrid

The matplotlib *AxesGrid* toolkit is a collection of helper classes to ease displaying multiple images in matplotlib. The AxesGrid toolkit is distributed with matplotlib source.



35.3 mplcursors

mplcursors provides interactive data cursors for matplotlib.

35.4 MplDataCursor

(*Not distributed with matplotlib*)

MplDataCursor is a toolkit written by Joe Kington to provide interactive "data cursors" (clickable annotation boxes) for matplotlib.

35.5 GTK Tools

mpl_toolkits.gtktools provides some utilities for working with GTK. This toolkit ships with matplotlib, but requires pygtk.

35.6 Excel Tools

mpl_toolkits.exceltools provides some utilities for working with Excel. This toolkit ships with matplotlib, but requires xlwt

35.7 Natgrid

(*Not distributed with matplotlib*)

mpl_toolkits.natgrid is an interface to natgrid C library for gridding irregularly spaced data. This requires a separate installation of the natgrid toolkit.

35.8 Matplotlib-Venn

(*Not distributed with matplotlib*)

Matplotlib-Venn provides a set of functions for plotting 2- and 3-set area-weighted (or unweighted) Venn diagrams.

35.9 mplstereonet

(*Not distributed with matplotlib*)

mplstereonet provides stereonets for plotting and analyzing orientation data in Matplotlib.

CHAPTER

THIRTYSIX

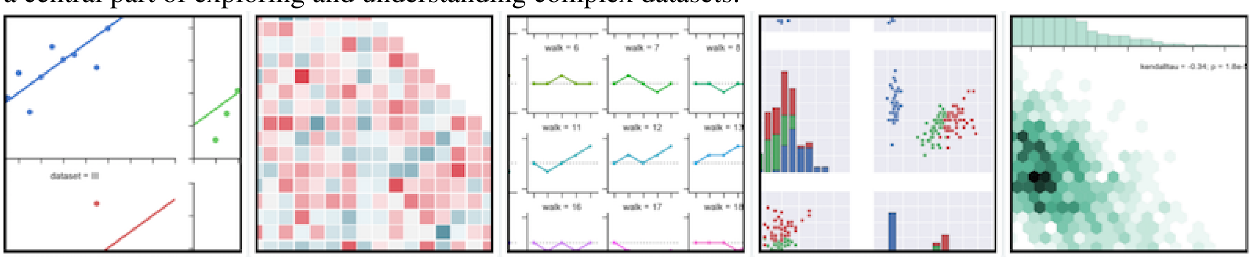
HIGH-LEVEL PLOTTING

Several projects have started to provide a higher-level interface to matplotlib. These are independent projects.

36.1 seaborn

(*Not distributed with matplotlib*)

seaborn is a high level interface for drawing statistical graphics with matplotlib. It aims to make visualization a central part of exploring and understanding complex datasets.

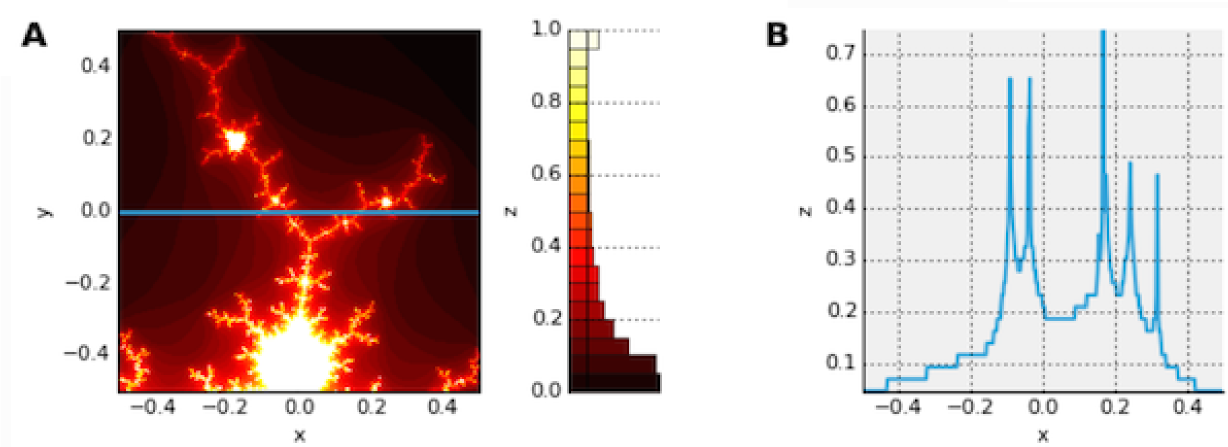


36.2 holoviews

(*Not distributed with matplotlib*)

holoviews makes it easier to visualize data interactively, especially in a Jupyter notebook, by providing a set of declarative plotting objects that store your data and associated metadata. Your data is then immediately visualizable alongside or overlaid with other data, either statically or with automatically provided widgets for parameter exploration.

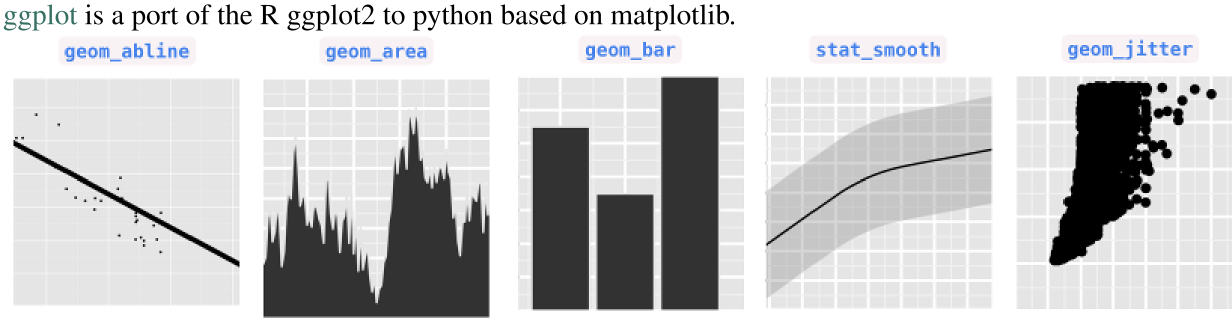
```
import numpy as np
import holoviews as hv
hv.notebook_extension('matplotlib')
fractal = hv.Image(np.load('mandelbrot.npy'))

((fractal * hv.HLine(y=0)).hist() + fractal.sample(y=0))
```



36.3 ggplot

(*Not distributed with matplotlib*)

ggplot is a port of the R ggplot2 to python based on matplotlib.



36.4 prettyplotlib

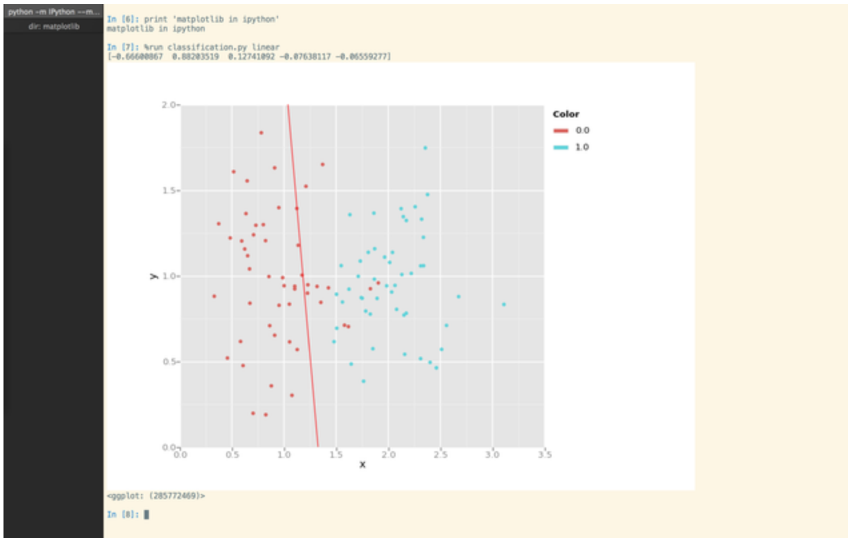
(*Not distributed with matplotlib*)

prettyplotlib is an extension to matplotlib which changes many of the defaults to make plots some consider more attractive.

36.5 iTerm2 terminal backend

(*Not distributed with matplotlib*)

matplotlib_iterm2 is an external matplotlib backend uses iTerm2 nightly build inline image display feature.



Part IX

The Matplotlib API

CHAPTER THIRTYSEVEN

PLOTTING COMMANDS SUMMARY

matplotlib.pyplot.**plotting**()

Function	Description
acorr	Plot the autocorrelation of x.
angle_spectrum	Plot the angle spectrum.
annotate	Annotate the point xy with text s.
arrow	Add an arrow to the axes.
autoscale	Autoscale the axis view to the data (toggle).
axes	Add an axes to the figure.
axhline	Add a horizontal line across the axis.
axhspan	Add a horizontal span (rectangle) across the axis.
axis	Convenience method to get or set axis properties.
axvline	Add a vertical line across the axes.
axvspan	Add a vertical span (rectangle) across the axes.
bar	Make a bar plot.
barbs	Plot a 2-D field of barbs.
barh	Make a horizontal bar plot.
box	Turn the axes box on or off.
boxplot	Make a box and whisker plot.
broken_barh	Plot horizontal bars.
cla	Clear the current axes.
clabel	Label a contour plot.
clf	Clear the current figure.
clim	Set the color limits of the current image.
close	Close a figure window.
cohere	Plot the coherence between *x* and *y*.
colorbar	Add a colorbar to a plot.
contour	Plot contours.
contourf	Plot contours.
csd	Plot the cross-spectral density.
delaxes	Remove an axes from the current figure.
draw	Redraw the current figure.
errorbar	Plot an errorbar graph.

Table 37.1 – continued from previous page

Function	Description
eventplot	Plot identical parallel lines at specific positions.
figimage	Adds a non-resampled image to the figure.
figlegend	Place a legend in the figure.
fignum_exists	
figtext	Add text to figure.
figure	Creates a new figure.
fill	Plot filled polygons.
fill_between	Make filled polygons between two curves.
fill_betweenx	Make filled polygons between two horizontal curves.
findobj	Find artist objects.
gca	Get the current Axes instance on the current figure matching the given keyword args,
gcf	Get a reference to the current figure.
gci	Get the current colorable artist.
get_figlabels	Return a list of existing figure labels.
get_fignums	Return a list of existing figure numbers.
grid	Turn the axes grids on or off.
hexbin	Make a hexagonal binning plot.
hist	Plot a histogram.
hist2d	Make a 2D histogram plot.
hlines	Plot horizontal lines at each `y` from `xmin` to `xmax`.
hold	Set the hold state.
imread	Read an image from a file into an array.
imsave	Save an array as in image file.
imshow	Display an image on the axes.
install_repl_displayhook	Install a repl display hook so that any stale figure are automatically redrawn when co
ioff	Turn interactive mode off.
ion	Turn interactive mode on.
ishold	Return the hold status of the current axes.
isinteractive	Return status of interactive mode.
legend	Places a legend on the axes.
locator_params	Control behavior of tick locators.
loglog	Make a plot with log scaling on both the *x* and *y* axis.
magnitude_spectrum	Plot the magnitude spectrum.
margins	Set or retrieve autoscaling margins.
matshow	Display an array as a matrix in a new figure window.
minorticks_off	Remove minor ticks from the current plot.
minorticks_on	Display minor ticks on the current plot.
over	Call a function with hold(True).
pause	Pause for *interval* seconds.
pcolor	Create a pseudocolor plot of a 2-D array.
pcolormesh	Plot a quadrilateral mesh.
phase_spectrum	Plot the phase spectrum.
pie	Plot a pie chart.

Table 37.1 – continued from previous page

Function	Description
plot	Plot lines and/or markers to the *Axes*.
plot_date	Plot with data with dates.
plotfile	Plot the data in in a file.
polar	Make a polar plot.
psd	Plot the power spectral density.
quiver	Plot a 2-D field of arrows.
quiverkey	Add a key to a quiver plot.
rc	Set the current rc params.
rc_context	Return a context manager for managing rc settings.
rcdefaults	Restore the default rc params.
rgrids	Get or set the radial gridlines on a polar plot.
savefig	Save the current figure.
sca	Set the current Axes instance to *ax*.
scatter	Make a scatter plot of x vs y, where x and y are sequence like objects of the same len
sci	Set the current image.
semilogx	Make a plot with log scaling on the *x* axis.
semilogy	Make a plot with log scaling on the *y* axis.
set_cmap	Set the default colormap.
setp	Set a property on an artist object.
show	Display a figure.
specgram	Plot a spectrogram.
spy	Plot the sparsity pattern on a 2-D array.
stackplot	Draws a stacked area plot.
stem	Create a stem plot.
step	Make a step plot.
streamplot	Draws streamlines of a vector flow.
subplot	Return a subplot axes positioned by the given grid definition.
subplot2grid	Create a subplot in a grid.
subplot_tool	Launch a subplot tool window for a figure.
subplots	Create a figure with a set of subplots already made.
subplots_adjust	Tune the subplot layout.
suptitle	Add a centered title to the figure.
switch_backend	Switch the default backend.
table	Add a table to the current axes.
text	Add text to the axes.
thetagrids	Get or set the theta locations of the gridlines in a polar plot.
tick_params	Change the appearance of ticks and tick labels.
ticklabel_format	Change the *ScalarFormatter* used by default for linear axes.
tight_layout	Automatically adjust subplot parameters to give specified padding.
title	Set a title of the current axes.
tricontour	Draw contours on an unstructured triangular grid.
tricontourf	Draw contours on an unstructured triangular grid.
tripcolor	Create a pseudocolor plot of an unstructured triangular grid.

Table 37.1 – continued from previous page

Function	Description
triplot	Draw a unstructured triangular grid as lines and/or markers.
twinx	Make a second axes that shares the *x*-axis.
twiny	Make a second axes that shares the *y*-axis.
uninstall_repl_displayhook	Uninstalls the matplotlib display hook.
violinplot	Make a violin plot.
vlines	Plot vertical lines.
xcorr	Plot the cross correlation between *x* and *y*.
xkcd	Turns on xkcd sketch-style drawing mode.
xlabel	Set the *x* axis label of the current axis.
xlim	Get or set the *x* limits of the current axes.
xscale	Set the scaling of the *x*-axis.
xticks	Get or set the *x*-limits of the current tick locations and labels.
ylabel	Set the *y* axis label of the current axis.
ylim	Get or set the *y*-limits of the current axes.
yscale	Set the scaling of the *y*-axis.
yticks	Get or set the *y*-limits of the current tick locations and labels.

matplotlib.pyplot.**colormaps**()

Matplotlib provides a number of colormaps, and others can be added using *register_cmap()*. This function documents the built-in colormaps, and will also return a list of all registered colormaps if called.

You can set the colormap for an image, pcolor, scatter, etc, using a keyword argument:

```
imshow(X, cmap=cm.hot)
```

or using the *set_cmap()* function:

```
imshow(X)
pyplot.set_cmap('hot')
pyplot.set_cmap('jet')
```

In interactive mode, *set_cmap()* will update the colormap post-hoc, allowing you to see which one works best for your data.

All built-in colormaps can be reversed by appending `_r`: For instance, `gray_r` is the reverse of `gray`.

There are several common color schemes used in visualization:

Sequential schemes for unipolar data that progresses from low to high

Diverging schemes for bipolar data that emphasizes positive or negative deviations from a central value

Cyclic schemes meant for plotting values that wrap around at the endpoints, such as phase angle, wind direction, or time of day

Qualitative schemes for nominal data that has no inherent ordering, where color is used only to distinguish categories

The base colormaps are derived from those of the same name provided with Matlab:

Colormap	Description
autumn	sequential linearly-increasing shades of red-orange-yellow
bone	sequential increasing black-white color map with a tinge of blue, to emulate X-ray film
cool	linearly-decreasing shades of cyan-magenta
copper	sequential increasing shades of black-copper
flag	repetitive red-white-blue-black pattern (not cyclic at endpoints)
gray	sequential linearly-increasing black-to-white grayscale
hot	sequential black-red-yellow-white, to emulate blackbody radiation from an object at increasing temperatures
hsv	cyclic red-yellow-green-cyan-blue-magenta-red, formed by changing the hue component in the HSV color space
inferno	perceptually uniform shades of black-red-yellow
jet	a spectral map with dark endpoints, blue-cyan-yellow-red; based on a fluid-jet simulation by NCSA [1]
magma	perceptually uniform shades of black-red-white
pink	sequential increasing pastel black-pink-white, meant for sepia tone colorization of photographs
plasma	perceptually uniform shades of blue-red-yellow
prism	repetitive red-yellow-green-blue-purple-...-green pattern (not cyclic at endpoints)
spring	linearly-increasing shades of magenta-yellow
summer	sequential linearly-increasing shades of green-yellow
viridis	perceptually uniform shades of blue-green-yellow
winter	linearly-increasing shades of blue-green

For the above list only, you can also set the colormap using the corresponding pylab shortcut interface function, similar to Matlab:

```
imshow(X)
hot()
jet()
```

The next set of palettes are from the Yorick scientific visualisation package, an evolution of the GIST package, both by David H. Munro:

[1] Rainbow colormaps, `jet` in particular, are considered a poor choice for scientific visualization by many researchers: Rainbow Color Map (Still) Considered Harmful

Colormap	Description
gist_earth	mapmaker's colors from dark blue deep ocean to green lowlands to brown highlands to white mountains
gist_heat	sequential increasing black-red-orange-white, to emulate blackbody radiation from an iron bar as it grows hotter
gist_ncar	pseudo-spectral black-blue-green-yellow-red-purple-white colormap from National Center for Atmospheric Research [2]
gist_rainbow	runs through the colors in spectral order from red to violet at full saturation (like *hsv* but not cyclic)
gist_stern	"Stern special" color table from Interactive Data Language software

The following colormaps are based on the ColorBrewer color specifications and designs developed by Cynthia Brewer:

ColorBrewer Diverging (luminance is highest at the midpoint, and decreases towards differently-colored endpoints):

Colormap	Description
BrBG	brown, white, blue-green
PiYG	pink, white, yellow-green
PRGn	purple, white, green
PuOr	orange, white, purple
RdBu	red, white, blue
RdGy	red, white, gray
RdYlBu	red, yellow, blue
RdYlGn	red, yellow, green
Spectral	red, orange, yellow, green, blue

ColorBrewer Sequential (luminance decreases monotonically):

Colormap	Description
Blues	white to dark blue
BuGn	white, light blue, dark green
BuPu	white, light blue, dark purple
GnBu	white, light green, dark blue
Greens	white to dark green
Greys	white to black (not linear)
Oranges	white, orange, dark brown
OrRd	white, orange, dark red
PuBu	white, light purple, dark blue
PuBuGn	white, light purple, dark green
PuRd	white, light purple, dark red
Purples	white to dark purple
RdPu	white, pink, dark purple
Reds	white to dark red
YlGn	light yellow, dark green
YlGnBu	light yellow, light green, dark blue
YlOrBr	light yellow, orange, dark brown
YlOrRd	light yellow, orange, dark red

[2] Resembles "BkBlAqGrYeOrReViWh200" from NCAR Command Language. See Color Table Gallery

ColorBrewer Qualitative:

(For plotting nominal data, `ListedColormap` should be used, not `LinearSegmentedColormap`. Different sets of colors are recommended for different numbers of categories. These continuous versions of the qualitative schemes may be removed or converted in the future.)

- Accent
- Dark2
- Paired
- Pastel1
- Pastel2
- Set1
- Set2
- Set3

Other miscellaneous schemes:

Colormap	Description
afmhot	sequential black-orange-yellow-white blackbody spectrum, commonly used in atomic force microscopy
brg	blue-red-green
bwr	diverging blue-white-red
coolwarm	diverging blue-gray-red, meant to avoid issues with 3D shading, color blindness, and ordering of colors [3]
CMRmap	"Default colormaps on color images often reproduce to confusing grayscale images. The proposed colormap maintains an aesthetically pleasing color image that automatically reproduces to a monotonic grayscale with discrete, quantifiable saturation levels." [4]
cubehelix	Unlike most other color schemes cubehelix was designed by D.A. Green to be monotonically increasing in terms of perceived brightness. Also, when printed on a black and white postscript printer, the scheme results in a greyscale with monotonically increasing brightness. This color scheme is named cubehelix because the r,g,b values produced can be visualised as a squashed helix around the diagonal in the r,g,b color cube.
gnuplot	gnuplot's traditional pm3d scheme (black-blue-red-yellow)
gnuplot2	sequential color printable as gray (black-blue-violet-yellow-white)
ocean	green-blue-white
rainbow	spectral purple-blue-green-yellow-orange-red colormap with diverging luminance
seismic	diverging blue-white-red
nipy_spectral	black-purple-blue-green-yellow-red-white spectrum, originally from the Neuroimaging in Python project
terrain	mapmaker's colors, blue-green-yellow-brown-white, originally from IGOR Pro

The following colormaps are redundant and may be removed in future versions. It's recommended to

[3] See Diverging Color Maps for Scientific Visualization by Kenneth Moreland.

[4] See A Color Map for Effective Black-and-White Rendering of Color-Scale Images by Carey Rappaport

use the names in the descriptions instead, which produce identical output:

Colormap	Description
gist_gray	identical to *gray*
gist_yarg	identical to *gray_r*
binary	identical to *gray_r*
spectral	identical to *nipy_spectral* [5]

[5] Changed to distinguish from ColorBrewer's *Spectral* map. `spectral()` still works, but `set_cmap('nipy_spectral')` is recommended for clarity.

CHAPTER

THIRTYEIGHT

API CHANGES

Log of changes to matplotlib that affect the outward-facing API. If updating matplotlib breaks your scripts, this list may help describe what changes may be necessary in your code or help figure out possible sources of the changes you are experiencing.

For new features that were added to matplotlib, please see *What's new in matplotlib*.

38.1 Changes in 1.5.3

38.1.1 `ax.plot(..., marker=None)` gives default marker

Prior to 1.5.3 kwargs passed to `plot` were handled in two parts – default kwargs generated internal to `plot` (such as the cycled styles) and user supplied kwargs. The internally generated kwargs were passed to the `matplotlib.lines.Line2D.__init__` and the user kwargs were passed to `ln.set(**kwargs)` to update the artist after it was created. Now both sets of kwargs are merged and passed to `__init__`. This change was made to allow `None` to be passed in via the user kwargs to mean 'do the default thing' as is the convention through out mpl rather than raising an exception.

Unlike most *`Line2D`* setter methods *`set_marker`* did accept `None` as a valid input which was mapped to 'no marker'. Thus, by routing this `marker=None` through `__init__` rather than `set(...)` the meaning of `ax.plot(..., marker=None)` changed from 'no markers' to 'default markers from rcparams'.

This is change is only evident if `mpl.rcParams['lines.marker']` has a value other than `'None'` (which is string `'None'` which means 'no marker').

38.2 Changes in 1.5.2

38.2.1 Default Behavior Changes

Changed default `autorange` behavior in boxplots

Prior to v1.5.2, the whiskers of boxplots would extend to the mininum and maximum values if the quartiles were all equal (i.e., Q1 = median = Q3). This behavior has been disabled by default to restore consistency with other plotting packages.

To restore the old behavior, simply set `autorange=True` when calling `plt.boxplot`.

38.3 Changes in 1.5.0

38.3.1 Code Changes

Split `matplotlib.cbook.ls_mapper` in two

The `matplotlib.cbook.ls_mapper` dictionary is split into two now to distinguish between qualified linestyle used by backends and unqualified ones. `ls_mapper` now maps from the short symbols (e.g. `"--"`) to qualified names (`"solid"`). The new ls_mapper_r is the reversed mapping.

Prevent moving artists between Axes, Property-ify Artist.axes, deprecate Artist.{get,set}_axes

The reason this was done was to prevent adding an Artist that is already associated with an Axes to be moved/added to a different Axes. This was never supported as it causes havoc with the transform stack. The apparent support for this (as it did not raise an exception) was the source of multiple bug reports and questions on SO.

For almost all use-cases, the assignment of the axes to an artist should be taken care of by the axes as part of the `Axes.add_*` method, hence the deprecation {get,set}_axes.

Removing the `set_axes` method will also remove the 'axes' line from the ACCEPTS kwarg tables (assuming that the removal date gets here before that gets overhauled).

Tighted input validation on 'pivot' kwarg to quiver

Tightened validation so that only {'tip', 'tail', 'mid', and 'middle'} (but any capitalization) are valid values for the 'pivot' kwarg in the `Quiver.__init__` (and hence `Axes.quiver` and `plt.quiver` which both fully delegate to `Quiver`). Previously any input matching 'mid.*' would be interpreted as 'middle', 'tip.*' as 'tip' and any string not matching one of those patterns as 'tail'.

The value of `Quiver.pivot` is normalized to be in the set {'tip', 'tail', 'middle'} in `Quiver.__init__`.

Reordered `Axes.get_children`

The artist order returned by `Axes.get_children` did not match the one used by `Axes.draw`. They now use the same order, as `Axes.draw` now calls `Axes.get_children`.

Changed behaviour of contour plots

The default behaviour of *`contour()`* and *`contourf()`* when using a masked array is now determined by the new keyword argument `corner_mask`, or if this is not specified then the new rcParam `contour.corner_mask` instead. The new default behaviour is equivalent to using `corner_mask=True`; the previous behaviour can be obtained using `corner_mask=False` or by changing the rcParam. The example http://matplotlib.org/examples/pylab_examples/contour_corner_mask.html demonstrates the difference. Use of the old contouring algorithm, which is obtained with `corner_mask='legacy'`, is now deprecated.

Contour labels may now appear in different places than in earlier versions of matplotlib.

In addition, the keyword argument `nchunk` now applies to *contour()* as well as *contourf()*, and it subdivides the domain into subdomains of exactly `nchunk` by `nchunk` quads, whereas previously it was only roughly `nchunk` by `nchunk` quads.

The C/C++ object that performs contour calculations used to be stored in the public attribute QuadContourSet.Cntr, but is now stored in a private attribute and should not be accessed by end users.

Added set_params function to all Locator types

This was a bug fix targeted at making the api for Locators more consistent.

In the old behavior, only locators of type MaxNLocator have set_params() defined, causing its use on any other Locator to throw an AttributeError *(aside: set_params(args) is a function that sets the parameters of a Locator instance to be as specified within args)*. The fix involves moving set_params() to the Locator class such that all subtypes will have this function defined.

Since each of the Locator subtype have their own modifiable parameters, a universal set_params() in Locator isn't ideal. Instead, a default no-operation function that raises a warning is implemented in Locator. Subtypes extending Locator will then override with their own implementations. Subtypes that do not have a need for set_params() will fall back onto their parent's implementation, which raises a warning as intended.

In the new behavior, all Locator instances will not throw an AttributeError when set_param() is called. For Locators that do not implement set_params(), the default implementation in Locator is used.

Disallow `None` as x or y value in ax.plot

Do not allow `None` as a valid input for the `x` or `y` args in `ax.plot`. This may break some user code, but this was never officially supported (ex documented) and allowing `None` objects through can lead to confusing exceptions downstream.

To create an empty line use

```
ln1, = ax.plot([], [], ...)
ln2, = ax.plot([], ...)
```

In either case to update the data in the `Line2D` object you must update both the `x` and `y` data.

Removed `args` and `kwargs` from `MicrosecondLocator.__call__`

The call signature of `__call__()` has changed from `__call__(self, *args, **kwargs)` to `__call__(self)`. This is consistent with the super class *Locator* and also all the other Locators derived from this super class.

No `ValueError` for the MicrosecondLocator and YearLocator

The *MicrosecondLocator* and *YearLocator* objects when called will return an empty list if the axes have no data or the view has no interval. Previously, they raised a `ValueError`. This is consistent with all the

Date Locators.

'OffsetBox.DrawingArea' respects the 'clip' keyword argument

The call signature was `OffsetBox.DrawingArea(..., clip=True)` but nothing was done with the `clip` argument. The object did not do any clipping regardless of that parameter. Now the object can and does clip the child `Artists` if they are set to be clipped.

You can turn off the clipping on a per-child basis using `child.set_clip_on(False)`.

Add salt to clipPath id

Add salt to the hash used to determine the id of the `clipPath` nodes. This is to avoid conflicts in two svg documents with the same clip path are included in the same document (see https://github.com/ipython/ipython/issues/8133 and https://github.com/matplotlib/matplotlib/issues/4349), however this means that the svg output is no longer deterministic if the same figure is saved twice. It is not expected that this will affect any users as the current ids are generated from an md5 hash of properties of the clip path and any user would have a very difficult time anticipating the value of the id.

Changed snap threshold for circle markers to inf

When drawing circle markers above some marker size (previously 6.0) the path used to generate the marker was snapped to pixel centers. However, this ends up distorting the marker away from a circle. By setting the snap threshold to inf snapping is never done on circles.

This change broke several tests, but is an improvement.

Preserve units with Text position

Previously the 'get_position' method on Text would strip away unit information even though the units were still present. There was no inherent need to do this, so it has been changed so that unit data (if present) will be preserved. Essentially a call to 'get_position' will return the exact value from a call to 'set_position'.

If you wish to get the old behaviour, then you can use the new method called 'get_unitless_position'.

New API for custom Axes view changes

Interactive pan and zoom were previously implemented using a Cartesian-specific algorithm that was not necessarily applicable to custom Axes. Three new private methods, `_get_view()`, `_set_view()`, and `_set_view_from_bbox()`, allow for custom `Axes` classes to override the pan and zoom algorithms. Implementors of custom `Axes` who override these methods may provide suitable behaviour for both pan and zoom as well as the view navigation buttons on the interactive toolbars.

38.3.2 MathTex visual changes

The spacing commands in mathtext have been changed to more closely match vanilla TeX.

Improved spacing in mathtext

The extra space that appeared after subscripts and superscripts has been removed.

No annotation coordinates wrap

In #2351 for 1.4.0 the behavior of ['axes points', 'axes pixel', 'figure points', 'figure pixel'] as coordinates was change to no longer wrap for negative values. In 1.4.3 this change was reverted for 'axes points' and 'axes pixel' and in addition caused 'axes fraction' to wrap. For 1.5 the behavior has been reverted to as it was in 1.4.0-1.4.2, no wrapping for any type of coordinate.

38.3.3 Deprecation

Deprecated `GraphicsContextBase.set_graylevel`

The `GraphicsContextBase.set_graylevel` function has been deprecated in 1.5 and will be removed in 1.6. It has been unused. The `GraphicsContextBase.set_foreground` could be used instead.

deprecated idle_event

The `idle_event` was broken or missing in most backends and causes spurious warnings in some cases, and its use in creating animations is now obsolete due to the animations module. Therefore code involving it has been removed from all but the wx backend (where it partially works), and its use is deprecated. The animations module may be used instead to create animations.

`color_cycle` deprecated

In light of the new property cycling feature, the Axes method *set_color_cycle* is now deprecated. Calling this method will replace the current property cycle with one that cycles just the given colors.

Similarly, the rc parameter *axes.color_cycle* is also deprecated in lieu of the new *axes.prop_cycle* parameter. Having both parameters in the same rc file is not recommended as the result cannot be predicted. For compatibility, setting *axes.color_cycle* will replace the cycler in *axes.prop_cycle* with a color cycle. Accessing *axes.color_cycle* will return just the color portion of the property cycle, if it exists.

Timeline for removal has not been set.

38.3.4 Bundled jquery

The version of jquery bundled with the webagg backend has been upgraded from 1.7.1 to 1.11.3. If you are using the version of jquery bundled with webagg you will need to update you html files as such

```
-    <script src="_static/jquery/js/jquery-1.7.1.min.js"></script>
+    <script src="_static/jquery/js/jquery-1.11.3.min.js"></script>
```

38.3.5 Code Removed

Removed `Image` from main namespace

`Image` was imported from PIL/pillow to test if PIL is available, but there is no reason to keep `Image` in the namespace once the availability has been determined.

Removed `lod` from Artist

Removed the method *set_lod* and all references to the attribute *_lod* as the are not used anywhere else in the code base. It appears to be a feature stub that was never built out.

Removed threading related classes from cbook

The classes `Scheduler`, `Timeout`, and `Idle` were in cbook, but are not used internally. They appear to be a prototype for the idle event system which was not working and has recently been pulled out.

Removed `Lena` images from sample_data

The `lena.png` and `lena.jpg` images have been removed from matplotlib's sample_data directory. The images are also no longer available from *matplotlib.cbook.get_sample_data*. We suggest using `matplotlib.cbook.get_sample_data('grace_hopper.png')` or `matplotlib.cbook.get_sample_data('grace_hopper.jpg')` instead.

Legend

Removed handling of `loc` as a positional argument to `Legend`

Legend handlers

Remove code to allow legend handlers to be callable. They must now implement a method `legend_artist`.

Axis

Removed method `set_scale`. This is now handled via a private method which should not be used directly by users. It is called via `Axes.set_{x,y}scale` which takes care of ensuring the coupled changes are also made to the Axes object.

finance.py

Removed functions with ambiguous argument order from finance.py

Annotation

Removed `textcoords` and `xytext` proprieties from Annotation objects.

spinxext.ipython_*.py

Both `ipython_console_highlighting` and `ipython_directive` have been moved to `IPython`.

Change your import from 'matplotlib.sphinxext.ipython_directive' to 'IPython.sphinxext.ipython_directive' and from 'matplotlib.sphinxext.ipython_directive' to 'IPython.sphinxext.ipython_directive'

LineCollection.color

Deprecated in 2005, use `set_color`

remove 'faceted' as a valid value for shading in tri.tripcolor

Use `edgecolor` instead. Added validation on `shading` to only be valid values.

Remove faceted kwarg from scatter

Remove support for the `faceted` kwarg. This was deprecated in d48b34288e9651ff95c3b8a071ef5ac5cf50bae7 (2008-04-18!) and replaced by `edgecolor`.

Remove set_colorbar method from ScalarMappable

Remove `set_colorbar` method, use `colorbar` attribute directly.

patheffects.svg

- remove `get_proxy_renderer` method from `AbstarctPathEffect` class
- remove `patch_alpha` and `offset_xy` from `SimplePatchShadow`

Remove testing.image_util.py

Contained only a no-longer used port of functionality from PIL

Remove mlab.FIFOBuffer

Not used internally and not part of core mission of mpl.

Remove `mlab.prepca`

Deprecated in 2009.

Remove `NavigationToolbar2QTAgg`

Added no functionality over the base `NavigationToolbar2Qt`

mpl.py

Remove the module `matplotlib.mpl`. Deprecated in 1.3 by PR #1670 and commit 78ce67d161625833cacff23cfe5d74920248c5b2

38.4 Changes in 1.4.x

38.4.1 Code changes

- A major refactoring of the axes module was made. The axes module has been split into smaller modules:
  - the `_base` module, which contains a new private _AxesBase class. This class contains all methods except plotting and labelling methods.
  - the `axes` module, which contains the Axes class. This class inherits from _AxesBase, and contains all plotting and labelling methods.
  - the `_subplot` module, with all the classes concerning subplotting.

There are a couple of things that do not exists in the `axes` module's namespace anymore. If you use them, you need to import them from their original location:

- math -> `import math`
- ma -> `from numpy import ma`
- cbook -> `from matplotlib import cbook`
- docstring -> `from matplotlib import docstring`
- is_sequence_of_strings -> `from matplotlib.cbook import is_sequence_of_strings`
- is_string_like -> `from matplotlib.cbook import is_string_like`
- iterable -> `from matplotlib.cbook import iterable`
- itertools -> `import itertools`
- martist -> `from matplotlib import artist as martist`
- matplotlib -> `import matplotlib`
- mcoll -> `from matplotlib import collections as mcoll`

- mcolors -> `from matplotlib import colors as mcolors`
- mcontour -> `from matplotlib import contour as mcontour`
- mpatches -> `from matplotlib import patches as mpatches`
- mpath -> `from matplotlib import path as mpath`
- mquiver -> `from matplotlib import quiver as mquiver`
- mstack -> `from matplotlib import stack as mstack`
- mstream -> `from matplotlib import stream as mstream`
- mtable -> `from matplotlib import table as mtable`
- As part of the refactoring to enable Qt5 support, the module `matplotlib.backends.qt4_compat` was renamed to `matplotlib.qt_compat`. `qt4_compat` is deprecated in 1.4 and will be removed in 1.5.
- The `errorbar()` method has been changed such that the upper and lower limits (*lolims*, *uplims*, *xlolims*, *xuplims*) now point in the correct direction.
- The *fmt* kwarg for `plot()` defaults.
- A bug has been fixed in the path effects rendering of fonts, which now means that the font size is consistent with non-path effect fonts. See https://github.com/matplotlib/matplotlib/issues/2889 for more detail.
- The Sphinx extensions `ipython_directive` and `ipython_console_highlighting` have been moved to the IPython project itself. While they remain in matplotlib for this release, they have been deprecated. Update your extensions in `conf.py` to point to `IPython.sphinxext.ipython_directive` instead of `matplotlib.sphinxext.ipython_directive`.
- In `finance`, almost all functions have been deprecated and replaced with a pair of functions name `*_ochl` and `*_ohlc`. The former is the 'open-close-high-low' order of quotes used previously in this module, and the latter is the 'open-high-low-close' order that is standard in finance.
- For consistency the `face_alpha` keyword to `matplotlib.patheffects.SimplePatchShadow` has been deprecated in favour of the `alpha` keyword. Similarly, the keyword `offset_xy` is now named `offset` across all `_Base`' has been renamed to `matplotlib.patheffects.AbstractPathEffect`. `matplotlib.patheffect.ProxyRenderer` has been renamed to `matplotlib.patheffects.PathEffectRenderer` and is now a full RendererBase subclass.
- The artist used to draw the outline of a `colorbar` has been changed from a `matplotlib.lines.Line2D` to `matplotlib.patches.Polygon`, thus `colorbar.ColorbarBase.outline` is now a `matplotlib.patches.Polygon` object.
- The legend handler interface has changed from a callable, to any object which implements the `legend_artists` method (a deprecation phase will see this interface be maintained for v1.4). See *Legend guide* for further details. Further legend changes include:
  - `matplotlib.axes.Axes._get_legend_handles()` now returns a generator of handles, rather than a list.

  - The *legend()* function's "loc" positional argument has been deprecated. Use the "loc" keyword instead.

- The rcParams `savefig.transparent` has been added to control default transparency when saving figures.
- Slightly refactored the `Annotation` family. The text location in `Annotation` is now handled entirely handled by the underlying `Text` object so `set_position` works as expected. The attributes `xytext` and `textcoords` have been deprecated in favor of `xyann` and `anncoords` so that `Annotation` and `AnnotaionBbox` can share a common sensibly named api for getting/setting the location of the text or box.
  - `xyann` -> set the location of the annotation
  - `xy` -> set where the arrow points to
  - `anncoords` -> set the units of the annotation location
  - `xycoords` -> set the units of the point location
  - `set_position()` -> `Annotation` only set location of annotation
- *matplotlib.mlab.specgram*, *matplotlib.mlab.psd*, *matplotlib.mlab.csd*, *matplotlib.mlab.cohere*, *matplotlib.mlab.cohere_pairs*, *matplotlib.pyplot.specgram*, *matplotlib.pyplot.psd*, *matplotlib.pyplot.csd*, and *matplotlib.pyplot.cohere* now raise ValueError where they previously raised AssertionError.
- For *matplotlib.mlab.psd*, *matplotlib.mlab.csd*, *matplotlib.mlab.cohere*, *matplotlib.mlab.cohere_pairs*, *matplotlib.pyplot.specgram*, *matplotlib.pyplot.psd*, *matplotlib.pyplot.csd*, and *matplotlib.pyplot.cohere*, in cases where a shape (n, 1) array is returned, this is now converted to a (n,) array. Previously, (n, m) arrays were averaged to an (n,) array, but (n, 1) arrays were returend unchanged. This change makes the dimensions consistent in both cases.
- Added the rcParam `axes.fromatter.useoffset` to control the default value of `useOffset` in `ticker.ScalarFormatter`
- Added `Formatter` sub-class `StrMethodFormatter` which does the exact same thing as `FormatStrFormatter`, but for new-style formatting strings.
- Deprecated `matplotlib.testing.image_util` and the only function within, `matplotlib.testing.image_util.autocontrast`. These will be removed completely in v1.5.0.
- The `fmt` argument of *plot_date()* has been changed from `bo` to just `o`, so color cycling can happen by default.
- Removed the class `FigureManagerQTAgg` and deprecated `NavigationToolbar2QTAgg` which will be removed in 1.5.
- Removed formerly public (non-prefixed) attributes `rect` and `drawRect` from `FigureCanvasQTAgg`; they were always an implementation detail of the (preserved) `drawRectangle()` function.
- The function signatures of `tight_bbox.adjust_bbox` and `tight_bbox.process_figure_for_rasterizing` have been changed. A new `fixed_dpi` parameter allows for overriding the `figure.dpi` setting instead of trying to deduce the intended behaviour from the file format.

- Added support for horizontal/vertical axes padding to `mpl_toolkits.axes_grid1.ImageGrid` — argument `axes_pad` can now be tuple-like if separate axis padding is required. The original behavior is preserved.
- Added support for skewed transforms to *matplotlib.transforms.Affine2D*, which can be created using the `skew` and `skew_deg` methods.
- Added clockwise parameter to control sectors direction in `axes.pie`
- In *matplotlib.lines.Line2D* the `markevery` functionality has been extended. Previously an integer start-index and stride-length could be specified using either a two-element-list or a two-element-tuple. Now this can only be done using a two-element-tuple. If a two-element-list is used then it will be treated as numpy fancy indexing and only the two markers corresponding to the given indexes will be shown.
- removed prop kwarg from `mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar` call. It was passed through to the base-class `__init__` and is only used for setting padding. Now `fontproperties` (which is what is really used to set the font properties of `AnchoredSizeBar`) is passed through in place of `prop`. If `fontpropreties` is not passed in, but `prop` is, then `prop` is used inplace of `fontpropreties`. If both are passed in, `prop` is silently ignored.
- The use of the index 0 in `plt.subplot` and related commands is deprecated. Due to a lack of validation calling `plt.subplots(2, 2, 0)` does not raise an exception, but puts an axes in the _last_ position. This is due to the indexing in subplot being 1-based (to mirror MATLAB) so before indexing into the `GridSpec` object used to determine where the axes should go, 1 is subtracted off. Passing in 0 results in passing -1 to `GridSpec` which results in getting the last position back. Even though this behavior is clearly wrong and not intended, we are going through a deprecation cycle in an abundance of caution that any users are exploiting this 'feature'. The use of 0 as an index will raise a warning in 1.4 and an exception in 1.5.
- Clipping is now off by default on offset boxes.
- matplotlib now uses a less-aggressive call to `gc.collect(1)` when closing figures to avoid major delays with large numbers of user objects in memory.
- The default clip value of *all* pie artists now defaults to `False`.

38.4.2 Code removal

- Removed `mlab.levypdf`. The code raised a numpy error (and has for a long time) and was not the standard form of the Levy distribution. `scipy.stats.levy` should be used instead

38.5 Changes in 1.3.x

38.5.1 Changes in 1.3.1

It is rare that we make an API change in a bugfix release, however, for 1.3.1 since 1.3.0 the following change was made:

- `text.Text.cached` (used to cache font objects) has been made into a private variable. Among the obvious encapsulation benefit, this removes this confusing-looking member from the documentation.
- The method *hist()* now always returns bin occupancies as an array of type `float`. Previously, it was sometimes an array of type `int`, depending on the call.

38.5.2 Code removal

- The following items that were deprecated in version 1.2 or earlier have now been removed completely.
  - The Qt 3.x backends (`qt` and `qtagg`) have been removed in favor of the Qt 4.x backends (`qt4` and `qt4agg`).
  - The FltkAgg and Emf backends have been removed.
  - The `matplotlib.nxutils` module has been removed. Use the functionality on *matplotlib.path.Path.contains_point* and friends instead.
  - Instead of `axes.Axes.get_frame`, use `axes.Axes.patch`.
  - The following `kwargs` to the `legend` function have been renamed:
    * `pad` -> `borderpad`
    * `labelsep` -> `labelspacing`
    * `handlelen` -> `handlelength`
    * `handletextsep` -> `handletextpad`
    * `axespad` -> `borderaxespad`

    Related to this, the following rcParams have been removed:
    * `legend.pad`, `legend.labelsep`, `legend.handlelen`, `legend.handletextsep` and `legend.axespad`
  - For the `hist` function, instead of `width`, use `rwidth` (relative width).
  - On `patches.Circle`, the `resolution` kwarg has been removed. For a circle made up of line segments, use `patches.CirclePolygon`.
  - The printing functions in the Wx backend have been removed due to the burden of keeping them up-to-date.
  - `mlab.liaupunov` has been removed.
  - `mlab.save`, `mlab.load`, `pylab.save` and `pylab.load` have been removed. We recommend using `numpy.savetxt` and `numpy.loadtxt` instead.
  - `widgets.HorizontalSpanSelector` has been removed. Use `widgets.SpanSelector` instead.

38.5.3 Code deprecation

- The CocoaAgg backend has been deprecated, with the possibility for deletion or resurrection in a future release.
- The top-level functions in *matplotlib.path* that are implemented in C++ were never meant to be public. Instead, users should use the Pythonic wrappers for them in the `path.Path` and `collections.Collection` classes. Use the following mapping to update your code:
  - `point_in_path` -> `path.Path.contains_point`
  - `get_path_extents` -> `path.Path.get_extents`
  - `point_in_path_collection` -> `collection.Collection.contains`
  - `path_in_path` -> `path.Path.contains_path`
  - `path_intersects_path` -> `path.Path.intersects_path`
  - `convert_path_to_polygons` -> `path.Path.to_polygons`
  - `cleanup_path` -> `path.Path.cleaned`
  - `points_in_path` -> `path.Path.contains_points`
  - `clip_path_to_rect` -> `path.Path.clip_to_bbox`
- `matplotlib.colors.normalize` and `matplotlib.colors.no_norm` have been deprecated in favour of *matplotlib.colors.Normalize* and *matplotlib.colors.NoNorm* respectively.
- The `ScalarMappable` class' `set_colorbar` is now deprecated. Instead, the *matplotlib.cm.ScalarMappable.colorbar* attribute should be used. In previous matplotlib versions this attribute was an undocumented tuple of `(colorbar_instance, colorbar_axes)` but is now just `colorbar_instance`. To get the colorbar axes it is possible to just use the *ax* attribute on a colorbar instance.
- The `mpl` module is now deprecated. Those who relied on this module should transition to simply using `import matplotlib as mpl`.

38.5.4 Code changes

- *Patch* now fully supports using RGBA values for its `facecolor` and `edgecolor` attributes, which enables faces and edges to have different alpha values. If the *Patch* object's `alpha` attribute is set to anything other than `None`, that value will override any alpha-channel value in both the face and edge colors. Previously, if *Patch* had `alpha=None`, the alpha component of `edgecolor` would be applied to both the edge and face.
- The optional `isRGB` argument to *set_foreground()* (and the other GraphicsContext classes that descend from it) has been renamed to `isRGBA`, and should now only be set to `True` if the `fg` color argument is known to be an RGBA tuple.
- For *Patch*, the `capstyle` used is now `butt`, to be consistent with the default for most other objects, and to avoid problems with non-solid `linestyle` appearing solid when using a large `linewidth`. Previously, *Patch* used `capstyle='projecting'`.

- Path objects can now be marked as readonly by passing readonly=True to its constructor. The built-in path singletons, obtained through Path.unit* class methods return readonly paths. If you have code that modified these, you will need to make a deepcopy first, using either:

```
import copy
path = copy.deepcopy(Path.unit_circle())

# or

path = Path.unit_circle().deepcopy()
```

  Deep copying a Path always creates an editable (i.e. non-readonly) Path.

- The list at Path.NUM_VERTICES was replaced by a dictionary mapping Path codes to the number of expected vertices at *NUM_VERTICES_FOR_CODE*.
- To support XKCD style plots, the matplotlib.path.cleanup_path() method's signature was updated to require a sketch argument. Users of matplotlib.path.cleanup_path() are encouraged to use the new cleaned() Path method.
- Data limits on a plot now start from a state of having "null" limits, rather than limits in the range (0, 1). This has an effect on artists that only control limits in one direction, such as axvline and axhline, since their limits will not longer also include the range (0, 1). This fixes some problems where the computed limits would be dependent on the order in which artists were added to the axes.
- Fixed a bug in setting the position for the right/top spine with data position type. Previously, it would draw the right or top spine at +1 data offset.
- In *FancyArrow*, the default arrow head width, head_width, has been made larger to produce a visible arrow head. The new value of this kwarg is head_width = 20 * width.
- It is now possible to provide number of levels + 1 colors in the case of extend='both' for contourf (or just number of levels colors for an extend value min or max) such that the resulting colormap's set_under and set_over are defined appropriately. Any other number of colors will continue to behave as before (if more colors are provided than levels, the colors will be unused). A similar change has been applied to contour, where extend='both' would expect number of levels + 2 colors.
- A new keyword *extendrect* in colorbar() and ColorbarBase allows one to control the shape of colorbar extensions.
- The extension of *MultiCursor* to both vertical (default) and/or horizontal cursor implied that self.line is replaced by self.vline for vertical cursors lines and self.hline is added for the horizontal cursors lines.
- On POSIX platforms, the *report_memory()* function raises NotImplementedError instead of OSError if the **ps** command cannot be run.
- The matplotlib.cbook.check_output() function has been moved to matplotlib.compat.subprocess().

38.5.5 Configuration and rcParams

- On Linux, the user-specific `matplotlibrc` configuration file is now located in `config/matplotlib/matplotlibrc` to conform to the XDG Base Directory Specification.
- The `font.*` rcParams now affect only text objects created after the rcParam has been set, and will not retroactively affect already existing text objects. This brings their behavior in line with most other rcParams.
- Removed call of `grid()` in `plotfile()`. To draw the axes grid, set the `axes.grid` rcParam to *True*, or explicitly call `grid()`.

38.6 Changes in 1.2.x

- The `classic` option of the rc parameter `toolbar` is deprecated and will be removed in the next release.
- The `isvector()` method has been removed since it is no longer functional.
- The `rasterization_zorder` property on `Axes` a zorder below which artists are rasterized. This has defaulted to -30000.0, but it now defaults to `None`, meaning no artists will be rasterized. In order to rasterize artists below a given zorder value, `set_rasterization_zorder` must be explicitly called.
- In `scatter()`, and `scatter`, when specifying a marker using a tuple, the angle is now specified in degrees, not radians.
- Using `twinx()` or `twiny()` no longer overrides the current locaters and formatters on the axes.
- In `contourf()`, the handling of the *extend* kwarg has changed. Formerly, the extended ranges were mapped after to 0, 1 after being normed, so that they always corresponded to the extreme values of the colormap. Now they are mapped outside this range so that they correspond to the special colormap values determined by the `set_under()` and `set_over()` methods, which default to the colormap end points.
- The new rc parameter `savefig.format` replaces `cairo.format` and `savefig.extension`, and sets the default file format used by `matplotlib.figure.Figure.savefig()`.
- In `pie()` and `pie()`, one can now set the radius of the pie; setting the *radius* to 'None' (the default value), will result in a pie with a radius of 1 as before.
- Use of `projection_factory()` is now deprecated in favour of axes class identification using `process_projection_requirements()` followed by direct axes class invocation (at the time of writing, functions which do this are: `add_axes()`, `add_subplot()` and `gca()`). Therefore:

```
key = figure._make_key(*args, **kwargs)
ispolar = kwargs.pop('polar', False)
projection = kwargs.pop('projection', None)
if ispolar:
    if projection is not None and projection != 'polar':
        raise ValueError('polar and projection args are inconsistent')
    projection = 'polar'
ax = projection_factory(projection, self, rect, **kwargs)
```

```
key = self._make_key(*args, **kwargs)

# is now

projection_class, kwargs, key = \
                    process_projection_requirements(self, *args, **kwargs)
ax = projection_class(self, rect, **kwargs)
```

This change means that third party objects can expose themselves as matplotlib axes by providing a `_as_mpl_axes` method. See *Adding new scales and projections to matplotlib* for more detail.

- A new keyword *extendfrac* in `colorbar()` and `ColorbarBase` allows one to control the size of the triangular minimum and maximum extensions on colorbars.
- A new keyword *capthick* in `errorbar()` has been added as an intuitive alias to the *markeredgewidth* and *mew* keyword arguments, which indirectly controlled the thickness of the caps on the errorbars. For backwards compatibility, specifying either of the original keyword arguments will override any value provided by *capthick*.
- Transform subclassing behaviour is now subtly changed. If your transform implements a non-affine transformation, then it should override the `transform_non_affine` method, rather than the generic `transform` method. Previously transforms would define `transform` and then copy the method into `transform_non_affine`:

```
class MyTransform(mtrans.Transform):
    def transform(self, xy):
        ...
    transform_non_affine = transform
```

This approach will no longer function correctly and should be changed to:

```
class MyTransform(mtrans.Transform):
    def transform_non_affine(self, xy):
        ...
```

- Artists no longer have `x_isdata` or `y_isdata` attributes; instead any artist's transform can be interrogated with `artist_instance.get_transform().contains_branch(ax.transData)`
- Lines added to an axes now take into account their transform when updating the data and view limits. This means transforms can now be used as a pre-transform. For instance:

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib.transforms as mtrans
>>> ax = plt.axes()
>>> ax.plot(range(10), transform=mtrans.Affine2D().scale(10) + ax.transData)
>>> print(ax.viewLim)
Bbox('array([[  0.,    0.],\n       [ 90.,   90.]])')
```

- One can now easily get a transform which goes from one transform's coordinate system to another, in an optimized way, using the new subtract method on a transform. For instance, to go from data coordinates to axes coordinates:

```
>>> import matplotlib.pyplot as plt
>>> ax = plt.axes()
>>> data2ax = ax.transData - ax.transAxes
>>> print(ax.transData.depth, ax.transAxes.depth)
3, 1
>>> print(data2ax.depth)
2
```

for versions before 1.2 this could only be achieved in a sub-optimal way, using `ax.transData + ax.transAxes.inverted()` (depth is a new concept, but had it existed it would return 4 for this example).

- `twinx` and `twiny` now returns an instance of SubplotBase if parent axes is an instance of SubplotBase.
- All Qt3-based backends are now deprecated due to the lack of py3k bindings. Qt and QtAgg backends will continue to work in v1.2.x for py2.6 and py2.7. It is anticipated that the Qt3 support will be completely removed for the next release.
- *`ColorConverter`*, *`Colormap`* and *`Normalize`* now subclasses `object`
- ContourSet instances no longer have a `transform` attribute. Instead, access the transform with the `get_transform` method.

38.7 Changes in 1.1.x

- Added new *`matplotlib.sankey.Sankey`* for generating Sankey diagrams.
- In *`imshow()`*, setting *interpolation* to 'nearest' will now always mean that the nearest-neighbor interpolation is performed. If you want the no-op interpolation to be performed, choose 'none'.
- There were errors in how the tri-functions were handling input parameters that had to be fixed. If your tri-plots are not working correctly anymore, or you were working around apparent mistakes, please see issue #203 in the github tracker. When in doubt, use kwargs.
- The 'symlog' scale had some bad behavior in previous versions. This has now been fixed and users should now be able to use it without frustrations. The fixes did result in some minor changes in appearance for some users who may have been depending on the bad behavior.
- There is now a common set of markers for all plotting functions. Previously, some markers existed only for *`scatter()`* or just for *`plot()`*. This is now no longer the case. This merge did result in a conflict. The string 'd' now means "thin diamond" while 'D' will mean "regular diamond".

38.8 Changes beyond 0.99.x

- The default behavior of *`matplotlib.axes.Axes.set_xlim()`*, *`matplotlib.axes.Axes.set_ylim()`*, and *`matplotlib.axes.Axes.axis()`*, and their corresponding pyplot functions, has been changed: when view limits are set explicitly with one of these methods, autoscaling is turned off for the matching axis. A new *auto* kwarg is available to control this behavior. The limit kwargs have been renamed to *left* and *right* instead of *xmin* and *xmax*, and *bottom* and *top* instead of *ymin* and *ymax*. The old names may still be used, however.

- There are five new Axes methods with corresponding pyplot functions to facilitate autoscaling, tick location, and tick label formatting, and the general appearance of ticks and tick labels:
  - `matplotlib.axes.Axes.autoscale()` turns autoscaling on or off, and applies it.
  - `matplotlib.axes.Axes.margins()` sets margins used to autoscale the `matplotlib.axes.Axes.viewLim` based on the `matplotlib.axes.Axes.dataLim`.
  - `matplotlib.axes.Axes.locator_params()` allows one to adjust axes locator parameters such as *nbins*.
  - `matplotlib.axes.Axes.ticklabel_format()` is a convenience method for controlling the `matplotlib.ticker.ScalarFormatter` that is used by default with linear axes.
  - `matplotlib.axes.Axes.tick_params()` controls direction, size, visibility, and color of ticks and their labels.
- The `matplotlib.axes.Axes.bar()` method accepts a *error_kw* kwarg; it is a dictionary of kwargs to be passed to the errorbar function.
- The `matplotlib.axes.Axes.hist()` *color* kwarg now accepts a sequence of color specs to match a sequence of datasets.
- The `EllipseCollection` has been changed in two ways:
  - There is a new *units* option, 'xy', that scales the ellipse with the data units. This matches the :class:'~matplotlib.patches.Ellipse' scaling.
  - The *height* and *width* kwargs have been changed to specify the height and width, again for consistency with `Ellipse`, and to better match their names; previously they specified the half-height and half-width.
- There is a new rc parameter `axes.color_cycle`, and the color cycle is now independent of the rc parameter `lines.color`. `matplotlib.Axes.set_default_color_cycle()` is deprecated.
- You can now print several figures to one pdf file and modify the document information dictionary of a pdf file. See the docstrings of the class `matplotlib.backends.backend_pdf.PdfPages` for more information.
- Removed configobj and enthought.traits packages, which are only required by the experimental traited config and are somewhat out of date. If needed, install them independently.
- The new rc parameter `savefig.extension` sets the filename extension that is used by `matplotlib.figure.Figure.savefig()` if its *fname* argument lacks an extension.
- In an effort to simplify the backend API, all clipping rectangles and paths are now passed in using GraphicsContext objects, even on collections and images. Therefore:

```
draw_path_collection(self, master_transform, cliprect, clippath,
                     clippath_trans, paths, all_transforms, offsets,
                     offsetTrans, facecolors, edgecolors, linewidths,
                     linestyles, antialiaseds, urls)
```

```
# is now

draw_path_collection(self, gc, master_transform, paths, all_transforms,
                     offsets, offsetTrans, facecolors, edgecolors,
                     linewidths, linestyles, antialiaseds, urls)


draw_quad_mesh(self, master_transform, cliprect, clippath,
               clippath_trans, meshWidth, meshHeight, coordinates,
               offsets, offsetTrans, facecolors, antialiased,
               showedges)

# is now

draw_quad_mesh(self, gc, master_transform, meshWidth, meshHeight,
               coordinates, offsets, offsetTrans, facecolors,
               antialiased, showedges)


draw_image(self, x, y, im, bbox, clippath=None, clippath_trans=None)

# is now

draw_image(self, gc, x, y, im)
```

- There are four new Axes methods with corresponding pyplot functions that deal with unstructured triangular grids:
  - `matplotlib.axes.Axes.tricontour()` draws contour lines on a triangular grid.
  - `matplotlib.axes.Axes.tricontourf()` draws filled contours on a triangular grid.
  - `matplotlib.axes.Axes.tripcolor()` draws a pseudocolor plot on a triangular grid.
  - `matplotlib.axes.Axes.triplot()` draws a triangular grid as lines and/or markers.

38.9 Changes in 0.99

- pylab no longer provides a load and save function. These are available in matplotlib.mlab, or you can use numpy.loadtxt and numpy.savetxt for text files, or np.save and np.load for binary numpy arrays.
- User-generated colormaps can now be added to the set recognized by `matplotlib.cm.get_cmap()`. Colormaps can be made the default and applied to the current image using `matplotlib.pyplot.set_cmap()`.
- changed use_mrecords default to False in mlab.csv2rec since this is partially broken
- Axes instances no longer have a "frame" attribute. Instead, use the new "spines" attribute. Spines is a dictionary where the keys are the names of the spines (e.g., 'left','right' and so on) and the values are the artists that draw the spines. For normal (rectilinear) axes, these artists are Line2D instances. For other axes (such as polar axes), these artists may be Patch instances.

- Polar plots no longer accept a resolution kwarg. Instead, each Path must specify its own number of interpolation steps. This is unlikely to be a user-visible change – if interpolation of data is required, that should be done before passing it to matplotlib.

38.10 Changes for 0.98.x

- psd(), csd(), and cohere() will now automatically wrap negative frequency components to the beginning of the returned arrays. This is much more sensible behavior and makes them consistent with specgram(). The previous behavior was more of an oversight than a design decision.
- Added new keyword parameters *nonposx*, *nonposy* to `matplotlib.axes.Axes` methods that set log scale parameters. The default is still to mask out non-positive values, but the kwargs accept 'clip', which causes non-positive values to be replaced with a very small positive value.
- Added new `matplotlib.pyplot.fignum_exists()` and `matplotlib.pyplot.get_fignums()`; they merely expose information that had been hidden in `matplotlib._pylab_helpers`.
- Deprecated numerix package.
- Added new `matplotlib.image.imsave()` and exposed it to the `matplotlib.pyplot` interface.
- Remove support for pyExcelerator in exceltools – use xlwt instead
- Changed the defaults of acorr and xcorr to use usevlines=True, maxlags=10 and normed=True since these are the best defaults
- Following keyword parameters for `matplotlib.label.Label` are now deprecated and new set of parameters are introduced. The new parameters are given as a fraction of the font-size. Also, *scatteryoffsets*, *fancybox* and *columnspacing* are added as keyword parameters.

Deprecated	New
pad	borderpad
labelsep	labelspacing
handlelen	handlelength
handlestextsep	handletextpad
axespad	borderaxespad

- Removed the configobj and experimental traits rc support
- Modified `matplotlib.mlab.psd()`, `matplotlib.mlab.csd()`, `matplotlib.mlab.cohere()`, and `matplotlib.mlab.specgram()` to scale one-sided densities by a factor of 2. Also, optionally scale the densities by the sampling frequency, which gives true values of densities that can be integrated by the returned frequency values. This also gives better MATLAB compatibility. The corresponding `matplotlib.axes.Axes` methods and `matplotlib.pyplot` functions were updated as well.
- Font lookup now uses a nearest-neighbor approach rather than an exact match. Some fonts may be different in plots, but should be closer to what was requested.
- `matplotlib.axes.Axes.set_xlim()`, `matplotlib.axes.Axes.set_ylim()` now return a copy of the `viewlim` array to avoid modify-in-place surprises.

- *matplotlib.afm.AFM.get_fullname()* and *matplotlib.afm.AFM.get_familyname()* no longer raise an exception if the AFM file does not specify these optional attributes, but returns a guess based on the required FontName attribute.
- Changed precision kwarg in *matplotlib.pyplot.spy()*; default is 0, and the string value 'present' is used for sparse arrays only to show filled locations.
- *matplotlib.collections.EllipseCollection* added.
- Added `angles` kwarg to *matplotlib.pyplot.quiver()* for more flexible specification of the arrow angles.
- Deprecated (raise NotImplementedError) all the mlab2 functions from *matplotlib.mlab* out of concern that some of them were not clean room implementations.
- Methods *matplotlib.collections.Collection.get_offsets()* and *matplotlib.collections.Collection.set_offsets()* added to *Collection* base class.
- `matplotlib.figure.Figure.figurePatch` renamed `matplotlib.figure.Figure.patch`; `matplotlib.axes.Axes.axesPatch` renamed `matplotlib.axes.Axes.patch`; `matplotlib.axes.Axes.axesFrame` renamed `matplotlib.axes.Axes.frame`. `matplotlib.axes.Axes.get_frame()`, which returns `matplotlib.axes.Axes.patch`, is deprecated.
- Changes in the `matplotlib.contour.ContourLabeler` attributes (*matplotlib.pyplot.clabel()* function) so that they all have a form like `.labelAttribute`. The three attributes that are most likely to be used by end users, `.cl`, `.cl_xy` and `.cl_cvalues` have been maintained for the moment (in addition to their renamed versions), but they are deprecated and will eventually be removed.
- Moved several functions in *matplotlib.mlab* and *matplotlib.cbook* into a separate module `matplotlib.numerical_methods` because they were unrelated to the initial purpose of mlab or cbook and appeared more coherent elsewhere.

38.11 Changes for 0.98.1

- Removed broken `matplotlib.axes3d` support and replaced it with a non-implemented error pointing to 0.91.x

38.12 Changes for 0.98.0

- *matplotlib.image.imread()* now no longer always returns RGBA data—if the image is luminance or RGB, it will return a MxN or MxNx3 array if possible. Also uint8 is no longer always forced to float.
- Rewrote the *matplotlib.cm.ScalarMappable* callback infrastructure to use *matplotlib.cbook.CallbackRegistry* rather than custom callback handling. Any users of `matplotlib.cm.ScalarMappable.add_observer()` of the *ScalarMappable* should use the `matplotlib.cm.ScalarMappable.callbacks` *CallbackRegistry* instead.

- New axes function and Axes method provide control over the plot color cycle: `matplotlib.axes.set_default_color_cycle()` and `matplotlib.axes.Axes.set_color_cycle()`.
- matplotlib now requires Python 2.4, so `matplotlib.cbook` will no longer provide `set`, `enumerate()`, `reversed()` or `izip()` compatibility functions.
- In Numpy 1.0, bins are specified by the left edges only. The axes method `matplotlib.axes.Axes.hist()` now uses future Numpy 1.3 semantics for histograms. Providing `binedges`, the last value gives the upper-right edge now, which was implicitly set to +infinity in Numpy 1.0. This also means that the last bin doesn't contain upper outliers any more by default.
- New axes method and pyplot function, `hexbin()`, is an alternative to `scatter()` for large datasets. It makes something like a `pcolor()` of a 2-D histogram, but uses hexagonal bins.
- New kwarg, `symmetric`, in `matplotlib.ticker.MaxNLocator` allows one require an axis to be centered around zero.
- Toolkits must now be imported from `mpl_toolkits` (not `matplotlib.toolkits`)

38.12.1 Notes about the transforms refactoring

A major new feature of the 0.98 series is a more flexible and extensible transformation infrastructure, written in Python/Numpy rather than a custom C extension.

The primary goal of this refactoring was to make it easier to extend matplotlib to support new kinds of projections. This is mostly an internal improvement, and the possible user-visible changes it allows are yet to come.

See `matplotlib.transforms` for a description of the design of the new transformation framework.

For efficiency, many of these functions return views into Numpy arrays. This means that if you hold on to a reference to them, their contents may change. If you want to store a snapshot of their current values, use the Numpy array method copy().

The view intervals are now stored only in one place – in the `matplotlib.axes.Axes` instance, not in the locator instances as well. This means locators must get their limits from their `matplotlib.axis.Axis`, which in turn looks up its limits from the `Axes`. If a locator is used temporarily and not assigned to an Axis or Axes, (e.g., in `matplotlib.contour`), a dummy axis must be created to store its bounds. Call `matplotlib.ticker.Locator.create_dummy_axis()` to do so.

The functionality of `Pbox` has been merged with `Bbox`. Its methods now all return copies rather than modifying in place.

The following lists many of the simple changes necessary to update code from the old transformation framework to the new one. In particular, methods that return a copy are named with a verb in the past tense, whereas methods that alter an object in place are named with a verb in the present tense.

matplotlib.transforms

Old method	New method
Bbox.get_bounds()	transforms.Bbox.bounds
Bbox.width()	transforms.Bbox.width
Bbox.height()	transforms.Bbox.height
Bbox.intervalx().get_bounds()	transforms.Bbox.intervalx
Bbox.intervalx().set_bounds()	[Bbox.intervalx is now a property.]
Bbox.intervaly().get_bounds()	transforms.Bbox.intervaly
Bbox.intervaly().set_bounds()	[Bbox.intervaly is now a property.]
Bbox.xmin()	transforms.Bbox.x0 or transforms.Bbox.xmin [1]
Bbox.ymin()	transforms.Bbox.y0 or transforms.Bbox.ymin [1]
Bbox.xmax()	transforms.Bbox.x1 or transforms.Bbox.xmax [1]
Bbox.ymax()	transforms.Bbox.y1 or transforms.Bbox.ymax [1]
Bbox.overlaps(bboxes)	Bbox.count_overlaps(bboxes)
bbox_all(bboxes)	Bbox.union(bboxes) [transforms.Bbox.union() is a staticmethod.]
lbwh_to_bbox(l, b, w, h)	Bbox.from_bounds(x0, y0, w, h) [transforms.Bbox.from_bounds() is a staticmethod.]
inverse_transform_bbox(trans, bbox)	Bbox.inverse_transformed(trans)
Interval.contains_open(v)	interval_contains_open(tuple, v)
Interval.contains(v)	interval_contains(tuple, v)
identity_transform()	*matplotlib.transforms.IdentityTransform*
blend_xy_sep_transform(xtrans, ytrans)	blended_transform_factory(xtrans, ytrans)
scale_transform(xs, ys)	Affine2D().scale(xs[, ys])
get_bbox_transform(boxin, boxout)	BboxTransform(boxin, boxout) or BboxTransformFrom(boxin) or BboxTransformTo(boxout)
Transform.seq_xy_tup(points)	Transform.transform(points)
Transform.inverse_xy_tup(points)	Transform.inverted().transform(points)

matplotlib.axes

Old method	New method
Axes.get_position()	*matplotlib.axes.Axes.get_position()* [2]
Axes.set_position()	*matplotlib.axes.Axes.set_position()* [3]
Axes.toggle_log_lineary()	*matplotlib.axes.Axes.set_yscale()* [4]
Subplot class	removed.

[1] The *Bbox* is bound by the points (x0, y0) to (x1, y1) and there is no defined order to these points, that is, x0 is not necessarily the left edge of the box. To get the left edge of the Bbox, use the read-only property xmin.

[2] *matplotlib.axes.Axes.get_position()* used to return a list of points, now it returns a *matplotlib.transforms.Bbox* instance.

[3] *matplotlib.axes.Axes.set_position()* now accepts either four scalars or a *matplotlib.transforms.Bbox* instance.

[4] Since the recfactoring allows for more than two scale types ('log' or 'linear'), it no longer makes sense to have a toggle. Axes.toggle_log_lineary() has been removed.

The `Polar` class has moved to *matplotlib.projections.polar*.

matplotlib.artist

Old method	New method
`Artist.set_clip_path(path)`	`Artist.set_clip_path(path, transform)` [5]

matplotlib.collections

Old method	New method
`linestyle`	`linestyles` [6]

matplotlib.colors

Old method	New method
`ColorConvertor.to_rgba_list(c)`	`ColorConvertor.to_rgba_array(c)` [`matplotlib.colors.ColorConvertor.to_rgba_array()` returns an Nx4 Numpy array of RGBA color quadruples.]

matplotlib.contour

Old method	New method
`Contour._segments`	`matplotlib.contour.Contour.get_paths`'`()` [Returns a list of *matplotlib.path.Path* instances.]

matplotlib.figure

Old method	New method
`Figure.dpi.get()` / `Figure.dpi.set()`	*matplotlib.figure.Figure.dpi (a property)*

matplotlib.patches

Old method	New method
`Patch.get_verts()`	*matplotlib.patches.Patch.get_path()* [Returns a *matplotlib.path.Path* instance]

[5] *matplotlib.artist.Artist.set_clip_path()* now accepts a *matplotlib.path.Path* instance and a *matplotlib.transforms.Transform* that will be applied to the path immediately before clipping.

[6] Linestyles are now treated like all other collection attributes, i.e. a single value or multiple values may be provided.

matplotlib.backend_bases

Old method	New method
GraphicsContext.set_clip_rectangle(tuple)	GraphicsContext.set_clip_rectangle(bbox)
GraphicsContext.get_clip_path()	GraphicsContext.get_clip_path() [7]
GraphicsContext.set_clip_path()	GraphicsContext.set_clip_path() [8]

RendererBase

New methods:

- *draw_path(self, gc, path, transform, rgbFace)*
- *draw_markers(self, gc, marker_path, marker_trans, path, trans, rgbFace)*
- *draw_path_collection(self, master_transform, cliprect, clippath, clippath_trans, paths, all_transforms, offsets, offsetTrans, facecolors, edgecolors, linewidths, linestyles, antialiaseds)* *[optional]*

Changed methods:

- draw_image(self, x, y, im, bbox) is now *draw_image(self, x, y, im, bbox, clippath, clippath_trans)*

Removed methods:

- draw_arc
- draw_line_collection
- draw_line
- draw_lines
- draw_point
- draw_quad_mesh
- draw_poly_collection
- draw_polygon
- draw_rectangle
- draw_regpoly_collection

38.13 Changes for 0.91.2

- For csv2rec(), checkrows=0 is the new default indicating all rows will be checked for type inference

[7] matplotlib.backend_bases.GraphicsContext.get_clip_path() returns a tuple of the form (*path*, *affine_transform*), where *path* is a *matplotlib.path.Path* instance and *affine_transform* is a *matplotlib.transforms.Affine2D* instance.

[8] matplotlib.backend_bases.GraphicsContext.set_clip_path() now only accepts a *matplotlib.transforms.TransformedPath* instance.

- A warning is issued when an image is drawn on log-scaled axes, since it will not log-scale the image data.
- Moved `rec2gtk()` to `matplotlib.toolkits.gtktools`
- Moved `rec2excel()` to `matplotlib.toolkits.exceltools`
- Removed, dead/experimental ExampleInfo, Namespace and Importer code from `matplotlib.__init__`

38.14 Changes for 0.91.1

38.15 Changes for 0.91.0

- Changed `cbook.is_file_like()` to `cbook.is_writable_file_like()` and corrected behavior.
- Added ax kwarg to `pyplot.colorbar()` and `Figure.colorbar()` so that one can specify the axes object from which space for the colorbar is to be taken, if one does not want to make the colorbar axes manually.
- Changed `cbook.reversed()` so it yields a tuple rather than a (index, tuple). This agrees with the python reversed builtin, and cbook only defines reversed if python doesn't provide the builtin.
- Made skiprows=1 the default on `csv2rec()`
- The gd and paint backends have been deleted.
- The errorbar method and function now accept additional kwargs so that upper and lower limits can be indicated by capping the bar with a caret instead of a straight line segment.
- The *`matplotlib.dviread`* file now has a parser for files like psfonts.map and pdftex.map, to map TeX font names to external files.
- The file *`matplotlib.type1font`* contains a new class for Type 1 fonts. Currently it simply reads pfa and pfb format files and stores the data in a way that is suitable for embedding in pdf files. In the future the class might actually parse the font to allow e.g., subsetting.
- `matplotlib.FT2Font` now supports `FT_Attach_File()`. In practice this can be used to read an afm file in addition to a pfa/pfb file, to get metrics and kerning information for a Type 1 font.
- The `AFM` class now supports querying CapHeight and stem widths. The get_name_char method now has an isord kwarg like get_width_char.
- Changed `pcolor()` default to shading='flat'; but as noted now in the docstring, it is preferable to simply use the edgecolor kwarg.
- The mathtext font commands (`\cal`, `\rm`, `\it`, `\tt`) now behave as TeX does: they are in effect until the next font change command or the end of the grouping. Therefore uses of `$\cal{R}$` should be changed to `${\cal R}$`. Alternatively, you may use the new LaTeX-style font commands (`\mathcal`, `\mathrm`, `\mathit`, `\mathtt`) which do affect the following group, e.g., `$\mathcal{R}$`.

- Text creation commands have a new default linespacing and a new `linespacing` kwarg, which is a multiple of the maximum vertical extent of a line of ordinary text. The default is 1.2; `linespacing=2` would be like ordinary double spacing, for example.
- Changed default kwarg in `matplotlib.colors.Normalize.__init__`‘()` to `clip=False`; clipping silently defeats the purpose of the special over, under, and bad values in the colormap, thereby leading to unexpected behavior. The new default should reduce such surprises.
- Made the emit property of `set_xlim()` and `set_ylim()` `True` by default; removed the Axes custom callback handling into a ‘callbacks’ attribute which is a `CallbackRegistry` instance. This now supports the ‘xlim_changed’ and ‘ylim_changed’ Axes events.

38.16 Changes for 0.90.1

```
The file dviread.py has a (very limited and fragile) dvi reader
for usetex support. The API might change in the future so don't
depend on it yet.

Removed deprecated support for a float value as a gray-scale;
now it must be a string, like '0.5'.  Added alpha kwarg to
ColorConverter.to_rgba_list.

New method set_bounds(vmin, vmax) for formatters, locators sets
the viewInterval and dataInterval from floats.

Removed deprecated colorbar_classic.

Line2D.get_xdata and get_ydata valid_only=False kwarg is replaced
by orig=True.  When True, it returns the original data, otherwise
the processed data (masked, converted)

Some modifications to the units interface.
units.ConversionInterface.tickers renamed to
units.ConversionInterface.axisinfo and it now returns a
units.AxisInfo object rather than a tuple.  This will make it
easier to add axis info functionality (e.g., I added a default label
on this iteration) w/o having to change the tuple length and hence
the API of the client code every time new functionality is added.
Also, units.ConversionInterface.convert_to_value is now simply
named units.ConversionInterface.convert.

Axes.errorbar uses Axes.vlines and Axes.hlines to draw its error
limits int he vertical and horizontal direction.  As you'll see
in the changes below, these functions now return a LineCollection
rather than a list of lines.  The new return signature for
errorbar is  ylins, caplines, errorcollections where
errorcollections is a xerrcollection, yerrcollection

Axes.vlines and Axes.hlines now create and returns a LineCollection, not a list
of lines.  This is much faster.  The kwarg signature has changed,
so consult the docs
```

```
MaxNLocator accepts a new Boolean kwarg ('integer') to force
ticks to integer locations.

Commands that pass an argument to the Text constructor or to
Text.set_text() now accept any object that can be converted
with '%s'.  This affects xlabel(), title(), etc.

Barh now takes a **kwargs dict instead of most of the old
arguments. This helps ensure that bar and barh are kept in sync,
but as a side effect you can no longer pass e.g., color as a
positional argument.

ft2font.get_charmap() now returns a dict that maps character codes
to glyph indices (until now it was reversed)

Moved data files into lib/matplotlib so that setuptools' develop
mode works. Re-organized the mpl-data layout so that this source
structure is maintained in the installation. (i.e., the 'fonts' and
'images' sub-directories are maintained in site-packages.).
Suggest removing site-packages/matplotlib/mpl-data and
~/.matplotlib/ttffont.cache before installing
```

38.17 Changes for 0.90.0

```
All artists now implement a "pick" method which users should not
call.  Rather, set the "picker" property of any artist you want to
pick on (the epsilon distance in points for a hit test) and
register with the "pick_event" callback.  See
examples/pick_event_demo.py for details

Bar, barh, and hist have "log" binary kwarg: log=True
sets the ordinate to a log scale.

Boxplot can handle a list of vectors instead of just
an array, so vectors can have different lengths.

Plot can handle 2-D x and/or y; it plots the columns.

Added linewidth kwarg to bar and barh.

Made the default Artist._transform None (rather than invoking
identity_transform for each artist only to have it overridden
later).  Use artist.get_transform() rather than artist._transform,
even in derived classes, so that the default transform will be
created lazily as needed

New LogNorm subclass of Normalize added to colors.py.
All Normalize subclasses have new inverse() method, and
the __call__() method has a new clip kwarg.
```

```
Changed class names in colors.py to match convention:
normalize -> Normalize, no_norm -> NoNorm.  Old names
are still available for now.

Removed obsolete pcolor_classic command and method.

Removed lineprops and markerprops from the Annotation code and
replaced them with an arrow configurable with kwarg arrowprops.
See examples/annotation_demo.py - JDH
```

38.18 Changes for 0.87.7

```
Completely reworked the annotations API because I found the old
API cumbersome.  The new design is much more legible and easy to
read.  See matplotlib.text.Annotation and
examples/annotation_demo.py

markeredgecolor and markerfacecolor cannot be configured in
matplotlibrc any more. Instead, markers are generally colored
automatically based on the color of the line, unless marker colors
are explicitly set as kwargs - NN

Changed default comment character for load to '#' - JDH

math_parse_s_ft2font_svg from mathtext.py & mathtext2.py now returns
width, height, svg_elements. svg_elements is an instance of Bunch (
cmbook.py) and has the attributes svg_glyphs and svg_lines, which are both
lists.

Renderer.draw_arc now takes an additional parameter, rotation.
It specifies to draw the artist rotated in degrees anti-
clockwise.  It was added for rotated ellipses.

Renamed Figure.set_figsize_inches to Figure.set_size_inches to
better match the get method, Figure.get_size_inches.

Removed the copy_bbox_transform from transforms.py; added
shallowcopy methods to all transforms.  All transforms already
had deepcopy methods.

FigureManager.resize(width, height): resize the window
specified in pixels

barh: x and y args have been renamed to width and bottom
respectively, and their order has been swapped to maintain
a (position, value) order.

bar and barh: now accept kwarg 'edgecolor'.
```

```
bar and barh: The left, height, width and bottom args can
now all be scalars or sequences; see docstring.

barh: now defaults to edge aligned instead of center
aligned bars

bar, barh and hist: Added a keyword arg 'align' that
controls between edge or center bar alignment.

Collections: PolyCollection and LineCollection now accept
vertices or segments either in the original form [(x,y),
(x,y), ...] or as a 2D numerix array, with X as the first column
and Y as the second. Contour and quiver output the numerix
form.  The transforms methods Bbox.update() and
Transformation.seq_xy_tups() now accept either form.

Collections: LineCollection is now a ScalarMappable like
PolyCollection, etc.

Specifying a grayscale color as a float is deprecated; use
a string instead, e.g., 0.75 -> '0.75'.

Collections: initializers now accept any mpl color arg, or
sequence of such args; previously only a sequence of rgba
tuples was accepted.

Colorbar: completely new version and api; see docstring.  The
original version is still accessible as colorbar_classic, but
is deprecated.

Contourf: "extend" kwarg replaces "clip_ends"; see docstring.
Masked array support added to pcolormesh.

Modified aspect-ratio handling:
    Removed aspect kwarg from imshow
    Axes methods:
        set_aspect(self, aspect, adjustable=None, anchor=None)
        set_adjustable(self, adjustable)
        set_anchor(self, anchor)
    Pylab interface:
        axis('image')

 Backend developers: ft2font's load_char now takes a flags
 argument, which you can OR together from the LOAD_XXX
 constants.
```

38.19 Changes for 0.86

```
Matplotlib data is installed into the matplotlib module.
This is similar to package_data.  This should get rid of
```

```
having to check for many possibilities in _get_data_path().
The MATPLOTLIBDATA env key is still checked first to allow
for flexibility.

1) Separated the color table data from cm.py out into
a new file, _cm.py, to make it easier to find the actual
code in cm.py and to add new colormaps. Everything
from _cm.py is imported by cm.py, so the split should be
transparent.
2) Enabled automatic generation of a colormap from
a list of colors in contour; see modified
examples/contour_demo.py.
3) Support for imshow of a masked array, with the
ability to specify colors (or no color at all) for
masked regions, and for regions that are above or
below the normally mapped region.  See
examples/image_masked.py.
4) In support of the above, added two new classes,
ListedColormap, and no_norm, to colors.py, and modified
the Colormap class to include common functionality. Added
a clip kwarg to the normalize class.
```

38.20 Changes for 0.85

```
Made xtick and ytick separate props in rc

made pos=None the default for tick formatters rather than 0 to
indicate "not supplied"

Removed "feature" of minor ticks which prevents them from
overlapping major ticks.  Often you want major and minor ticks at
the same place, and can offset the major ticks with the pad.  This
could be made configurable

Changed the internal structure of contour.py to a more OO style.
Calls to contour or contourf in axes.py or pylab.py now return
a ContourSet object which contains references to the
LineCollections or PolyCollections created by the call,
as well as the configuration variables that were used.
The ContourSet object is a "mappable" if a colormap was used.

Added a clip_ends kwarg to contourf. From the docstring:
         * clip_ends = True
           If False, the limits for color scaling are set to the
           minimum and maximum contour levels.
           True (default) clips the scaling limits.  Example:
           if the contour boundaries are V = [-100, 2, 1, 0, 1, 2, 100],
           then the scaling limits will be [-100, 100] if clip_ends
           is False, and [-3, 3] if clip_ends is True.
Added kwargs linewidths, antialiased, and nchunk to contourf.  These
```

```
are experimental; see the docstring.

Changed Figure.colorbar():
    kw argument order changed;
    if mappable arg is a non-filled ContourSet, colorbar() shows
            lines instead hof polygons.
    if mappable arg is a filled ContourSet with clip_ends=True,
            the endpoints are not labelled, so as to give the
            correct impression of open-endedness.

Changed LineCollection.get_linewidths to get_linewidth, for
consistency.
```

38.21 Changes for 0.84

```
Unified argument handling between hlines and vlines.  Both now
take optionally a fmt argument (as in plot) and a keyword args
that can be passed onto Line2D.

Removed all references to "data clipping" in rc and lines.py since
these were not used and not optimized.  I'm sure they'll be
resurrected later with a better implementation when needed.

'set' removed - no more deprecation warnings.  Use 'setp' instead.

Backend developers: Added flipud method to image and removed it
from to_str.  Removed origin kwarg from backend.draw_image.
origin is handled entirely by the frontend now.
```

38.22 Changes for 0.83

```
- Made HOME/.matplotlib the new config dir where the matplotlibrc
  file, the ttf.cache, and the tex.cache live.  The new default
  filenames in .matplotlib have no leading dot and are not hidden.
  e.g., the new names are matplotlibrc, tex.cache, and ttffont.cache.
  This is how ipython does it so it must be right.

  If old files are found, a warning is issued and they are moved to
  the new location.

- backends/__init__.py no longer imports new_figure_manager,
  draw_if_interactive and show from the default backend, but puts
  these imports into a call to pylab_setup.  Also, the Toolbar is no
  longer imported from WX/WXAgg.  New usage:

    from backends import pylab_setup
    new_figure_manager, draw_if_interactive, show = pylab_setup()
```

```
- Moved Figure.get_width_height() to FigureCanvasBase. It now
  returns int instead of float.
```

38.23 Changes for 0.82

```
- toolbar import change in GTKAgg, GTKCairo and WXAgg

- Added subplot config tool to GTK* backends -- note you must now
  import the NavigationToolbar2 from your backend of choice rather
  than from backend_gtk because it needs to know about the backend
  specific canvas -- see examples/embedding_in_gtk2.py.  Ditto for
  wx backend -- see examples/embedding_in_wxagg.py


- hist bin change

    Sean Richards notes there was a problem in the way we created
    the binning for histogram, which made the last bin
    underrepresented.  From his post:

      I see that hist uses the linspace function to create the bins
      and then uses searchsorted to put the values in their correct
      bin. That's all good but I am confused over the use of linspace
      for the bin creation. I wouldn't have thought that it does
      what is needed, to quote the docstring it creates a "Linear
      spaced array from min to max". For it to work correctly
      shouldn't the values in the bins array be the same bound for
      each bin? (i.e. each value should be the lower bound of a
      bin). To provide the correct bins for hist would it not be
      something like

      def bins(xmin, xmax, N):
        if N==1: return xmax
        dx = (xmax-xmin)/N # instead of N-1
        return xmin + dx*arange(N)


     This suggestion is implemented in 0.81.  My test script with these
     changes does not reveal any bias in the binning

      from matplotlib.numerix.mlab import randn, rand, zeros, Float
      from matplotlib.mlab import hist, mean

      Nbins = 50
      Ntests = 200
      results = zeros((Ntests,Nbins), typecode=Float)
      for i in range(Ntests):
          print 'computing', i
          x = rand(10000)
          n, bins = hist(x, Nbins)
```

```
        results[i] = n
    print mean(results)
```

38.24 Changes for 0.81

```
- pylab and artist "set" functions renamed to setp to avoid clash
  with python2.4 built-in set.  Current version will issue a
  deprecation warning which will be removed in future versions

- imshow interpolation arguments changes for advanced interpolation
  schemes.  See help imshow, particularly the interpolation,
  filternorm and filterrad kwargs

- Support for masked arrays has been added to the plot command and
  to the Line2D object.  Only the valid points are plotted.  A
  "valid_only" kwarg was added to the get_xdata() and get_ydata()
  methods of Line2D; by default it is False, so that the original
  data arrays are returned. Setting it to True returns the plottable
  points.

- contour changes:

  Masked arrays: contour and contourf now accept masked arrays as
    the variable to be contoured.  Masking works correctly for
    contour, but a bug remains to be fixed before it will work for
    contourf.  The "badmask" kwarg has been removed from both
    functions.

   Level argument changes:

     Old version: a list of levels as one of the positional
     arguments specified the lower bound of each filled region; the
     upper bound of the last region was taken as a very large
     number.  Hence, it was not possible to specify that z values
     between 0 and 1, for example, be filled, and that values
     outside that range remain unfilled.

     New version: a list of N levels is taken as specifying the
     boundaries of N-1 z ranges.  Now the user has more control over
     what is colored and what is not.  Repeated calls to contourf
     (with different colormaps or color specifications, for example)
     can be used to color different ranges of z.  Values of z
     outside an expected range are left uncolored.

     Example:
       Old: contourf(z, [0, 1, 2]) would yield 3 regions: 0-1, 1-2, and >2.
       New: it would yield 2 regions: 0-1, 1-2.  If the same 3 regions were
       desired, the equivalent list of levels would be [0, 1, 2,
       1e38].
```

38.25 Changes for 0.80

```
- xlim/ylim/axis always return the new limits regardless of
  arguments.  They now take kwargs which allow you to selectively
  change the upper or lower limits while leaving unnamed limits
  unchanged.  See help(xlim) for example
```

38.26 Changes for 0.73

```
- Removed deprecated ColormapJet and friends

- Removed all error handling from the verbose object

- figure num of zero is now allowed
```

38.27 Changes for 0.72

```
- Line2D, Text, and Patch copy_properties renamed update_from and
  moved into artist base class

- LineCollecitons.color renamed to LineCollections.set_color for
  consistency with set/get introspection mechanism,

- pylab figure now defaults to num=None, which creates a new figure
  with a guaranteed unique number

- contour method syntax changed - now it is MATLAB compatible

    unchanged: contour(Z)
    old: contour(Z, x=Y, y=Y)
    new: contour(X, Y, Z)

  see http://matplotlib.sf.net/matplotlib.pylab.html#-contour


 - Increased the default resolution for save command.

 - Renamed the base attribute of the ticker classes to _base to avoid conflict
   with the base method.  Sitt for subs

 - subs=none now does autosubbing in the tick locator.

 - New subplots that overlap old will delete the old axes.  If you
   do not want this behavior, use fig.add_subplot or the axes
   command
```

38.28 Changes for 0.71

```
Significant numerix namespace changes, introduced to resolve
namespace clashes between python built-ins and mlab names.
Refactored numerix to maintain separate modules, rather than
folding all these names into a single namespace.  See the following
mailing list threads for more information and background

  http://sourceforge.net/mailarchive/forum.php?thread_id=6398890&forum_id=36187
  http://sourceforge.net/mailarchive/forum.php?thread_id=6323208&forum_id=36187


OLD usage

  from matplotlib.numerix import array, mean, fft

NEW usage

  from matplotlib.numerix import array
  from matplotlib.numerix.mlab import mean
  from matplotlib.numerix.fft import fft

numerix dir structure mirrors numarray (though it is an incomplete
implementation)

  numerix
  numerix/mlab
  numerix/linear_algebra
  numerix/fft
  numerix/random_array

but of course you can use 'numerix : Numeric' and still get the
symbols.

pylab still imports most of the symbols from Numerix, MLab, fft,
etc, but is more cautious.  For names that clash with python names
(min, max, sum), pylab keeps the builtins and provides the numeric
versions with an a* prefix, e.g., (amin, amax, asum)
```

38.29 Changes for 0.70

```
MplEvent factored into a base class Event and derived classes
MouseEvent and KeyEvent

Removed definct set_measurement in wx toolbar
```

38.30 Changes for 0.65.1

```
removed add_axes and add_subplot from backend_bases.  Use
figure.add_axes and add_subplot instead.  The figure now manages the
current axes with gca and sca for get and set current axes.  If you
have code you are porting which called, e.g., figmanager.add_axes, you
can now simply do figmanager.canvas.figure.add_axes.
```

38.31 Changes for 0.65

```
mpl_connect and mpl_disconnect in the MATLAB interface renamed to
connect and disconnect

Did away with the text methods for angle since they were ambiguous.
fontangle could mean fontstyle (oblique, etc) or the rotation of the
text.  Use style and rotation instead.
```

38.32 Changes for 0.63

```
Dates are now represented internally as float days since 0001-01-01,
UTC.

All date tickers and formatters are now in matplotlib.dates, rather
than matplotlib.tickers

converters have been abolished from all functions and classes.
num2date and date2num are now the converter functions for all date
plots

Most of the date tick locators have a different meaning in their
constructors.  In the prior implementation, the first argument was a
base and multiples of the base were ticked.  e.g.,

  HourLocator(5)  # old: tick every 5 minutes

In the new implementation, the explicit points you want to tick are
provided as a number or sequence

   HourLocator(range(0,5,61))  # new: tick every 5 minutes

This gives much greater flexibility.  I have tried to make the
default constructors (no args) behave similarly, where possible.

Note that YearLocator still works under the base/multiple scheme.
The difference between the YearLocator and the other locators is
that years are not recurrent.
```

```
Financial functions:

  matplotlib.finance.quotes_historical_yahoo(ticker, date1, date2)

   date1, date2 are now datetime instances.  Return value is a list
   of quotes where the quote time is a float - days since gregorian
   start, as returned by date2num

   See examples/finance_demo.py for example usage of new API
```

38.33 Changes for 0.61

```
canvas.connect is now deprecated for event handling.  use
mpl_connect and mpl_disconnect instead.  The callback signature is
func(event) rather than func(widget, event)
```

38.34 Changes for 0.60

```
ColormapJet and Grayscale are deprecated.  For backwards
compatibility, they can be obtained either by doing

  from matplotlib.cm import ColormapJet

or

  from matplotlib.matlab import *

They are replaced by cm.jet and cm.grey
```

38.35 Changes for 0.54.3

```
removed the set_default_font / get_default_font scheme from the
font_manager to unify customization of font defaults with the rest of
the rc scheme.  See examples/font_properties_demo.py and help(rc) in
matplotlib.matlab.
```

38.36 Changes for 0.54

38.36.1 MATLAB interface

dpi

Several of the backends used a PIXELS_PER_INCH hack that I added to try and make images render consistently across backends. This just complicated matters. So you may find that some font sizes and line widths appear different than before. Apologies for the inconvenience. You should set the dpi to an accurate value for your screen to get true sizes.

pcolor and scatter

There are two changes to the MATLAB interface API, both involving the patch drawing commands. For efficiency, pcolor and scatter have been rewritten to use polygon collections, which are a new set of objects from matplotlib.collections designed to enable efficient handling of large collections of objects. These new collections make it possible to build large scatter plots or pcolor plots with no loops at the python level, and are significantly faster than their predecessors. The original pcolor and scatter functions are retained as pcolor_classic and scatter_classic.

The return value from pcolor is a PolyCollection. Most of the propertes that are available on rectangles or other patches are also available on PolyCollections, e.g., you can say:

```
c = scatter(blah, blah)
c.set_linewidth(1.0)
c.set_facecolor('r')
c.set_alpha(0.5)
```

or:

```
c = scatter(blah, blah)
set(c, 'linewidth', 1.0, 'facecolor', 'r', 'alpha', 0.5)
```

Because the collection is a single object, you no longer need to loop over the return value of scatter or pcolor to set properties for the entire list.

If you want the different elements of a collection to vary on a property, e.g., to have different line widths, see matplotlib.collections for a discussion on how to set the properties as a sequence.

For scatter, the size argument is now in points^2 (the area of the symbol in points) as in MATLAB and is not in data coords as before. Using sizes in data coords caused several problems. So you will need to adjust your size arguments accordingly or use scatter_classic.

mathtext spacing

For reasons not clear to me (and which I'll eventually fix) spacing no longer works in font groups. However, I added three new spacing commands which compensate for this '' (regular space), '/' (small space) and 'hspace{frac}' where frac is a fraction of fontsize in points. You will need to quote spaces in font strings, is:

```
title(r'$\rm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
```

38.36.2 Object interface - Application programmers

Autoscaling

The x and y axis instances no longer have autoscale view. These are handled by axes.autoscale_view

Axes creation

You should not instantiate your own Axes any more using the OO API. Rather, create a Figure as before and in place of:

```
f = Figure(figsize=(5,4), dpi=100)
a = Subplot(f, 111)
f.add_axis(a)
```

use:

```
f = Figure(figsize=(5,4), dpi=100)
a = f.add_subplot(111)
```

That is, add_axis no longer exists and is replaced by:

```
add_axes(rect, axisbg=defaultcolor, frameon=True)
add_subplot(num, axisbg=defaultcolor, frameon=True)
```

Artist methods

If you define your own Artists, you need to rename the _draw method to draw

Bounding boxes

matplotlib.transforms.Bound2D is replaced by matplotlib.transforms.Bbox. If you want to construct a bbox from left, bottom, width, height (the signature for Bound2D), use matplotlib.transforms.lbwh_to_bbox, as in

> bbox = clickBBox = lbwh_to_bbox(left, bottom, width, height)

The Bbox has a different API than the Bound2D. e.g., if you want to get the width and height of the bbox

> **OLD::** width = fig.bbox.x.interval() height = fig.bbox.y.interval()
>
> **New::** width = fig.bbox.width() height = fig.bbox.height()

Object constructors

You no longer pass the bbox, dpi, or transforms to the various Artist constructors. The old way or creating lines and rectangles was cumbersome because you had to pass so many attributes to the Line2D and Rectangle classes not related directly to the geometry and properties of the object. Now default values are added to the object when you call axes.add_line or axes.add_patch, so they are hidden from the user.

If you want to define a custom transformation on these objects, call o.set_transform(trans) where trans is a Transformation instance.

In prior versions of you wanted to add a custom line in data coords, you would have to do

> **l = Line2D(dpi, bbox, x, y,** color = color, transx = transx, transy = transy,)

now all you need is

> l = Line2D(x, y, color=color)

and the axes will set the transformation for you (unless you have set your own already, in which case it will eave it unchanged)

Transformations

The entire transformation architecture has been rewritten. Previously the x and y transformations where stored in the xaxis and yaxis instances. The problem with this approach is it only allows for separable transforms (where the x and y transformations don't depend on one another). But for cases like polar, they do. Now transformations operate on x,y together. There is a new base class matplotlib.transforms.Transformation and two concrete implementations, matplotlib.transforms.SeparableTransformation and matplotlib.transforms.Affine. The SeparableTransformation is constructed with the bounding box of the input (this determines the rectangular coordinate system of the input, i.e., the x and y view limits), the bounding box of the display, and possibly nonlinear transformations of x and y. The 2 most frequently used transformations, data coordinates -> display and axes coordinates -> display are available as ax.transData and ax.transAxes. See alignment_demo.py which uses axes coords.

Also, the transformations should be much faster now, for two reasons

- they are written entirely in extension code
- because they operate on x and y together, they can do the entire transformation in one loop. Earlier I did something along the lines of:

  ```
  xt = sx*func(x) + tx
  yt = sy*func(y) + ty
  ```

  Although this was done in numerix, it still involves 6 length(x) for-loops (the multiply, add, and function evaluation each for x and y). Now all of that is done in a single pass.

If you are using transformations and bounding boxes to get the cursor position in data coordinates, the method calls are a little different now. See the updated examples/coords_demo.py which shows you how to do this.

Likewise, if you are using the artist bounding boxes to pick items on the canvas with the GUI, the bbox methods are somewhat different. You will need to see the updated examples/object_picker.py.

See unit/transforms_unit.py for many examples using the new transformations.

38.37 Changes for 0.50

```
* refactored Figure class so it is no longer backend dependent.
  FigureCanvasBackend takes over the backend specific duties of the
  Figure.  matplotlib.backend_bases.FigureBase moved to
  matplotlib.figure.Figure.

* backends must implement FigureCanvasBackend (the thing that
  controls the figure and handles the events if any) and
  FigureManagerBackend (wraps the canvas and the window for MATLAB
  interface).  FigureCanvasBase implements a backend switching
  mechanism

* Figure is now an Artist (like everything else in the figure) and
  is totally backend independent

* GDFONTPATH renamed to TTFPATH

* backend faceColor argument changed to rgbFace

* colormap stuff moved to colors.py

* arg_to_rgb in backend_bases moved to class ColorConverter in
  colors.py

* GD users must upgrade to gd-2.0.22 and gdmodule-0.52 since new gd
  features (clipping, antialiased lines) are now used.

* Renderer must implement points_to_pixels

Migrating code:

MATLAB interface:

  The only API change for those using the MATLAB interface is in how
  you call figure redraws for dynamically updating figures.  In the
  old API, you did

    fig.draw()

  In the new API, you do

    manager = get_current_fig_manager()
    manager.canvas.draw()
```

```
  See the examples system_monitor.py, dynamic_demo.py, and anim.py

API

  There is one important API change for application developers.
  Figure instances used subclass GUI widgets that enabled them to be
  placed directly into figures.  e.g., FigureGTK subclassed
  gtk.DrawingArea.  Now the Figure class is independent of the
  backend, and FigureCanvas takes over the functionality formerly
  handled by Figure.  In order to include figures into your apps,
  you now need to do, for example

    # gtk example
    fig = Figure(figsize=(5,4), dpi=100)
    canvas = FigureCanvasGTK(fig)  # a gtk.DrawingArea
    canvas.show()
    vbox.pack_start(canvas)

  If you use the NavigationToolbar, this in now intialized with a
  FigureCanvas, not a Figure.  The examples embedding_in_gtk.py,
  embedding_in_gtk2.py, and mpl_with_glade.py all reflect the new
  API so use these as a guide.

  All prior calls to

   figure.draw()  and
   figure.print_figure(args)

  should now be

   canvas.draw()  and
   canvas.print_figure(args)

  Apologies for the inconvenience.  This refactorization brings
  significant more freedom in developing matplotlib and should bring
  better plotting capabilities, so I hope the inconvenience is worth
  it.
```

38.38 Changes for 0.42

```
* Refactoring AxisText to be backend independent.  Text drawing and
  get_window_extent functionality will be moved to the Renderer.

* backend_bases.AxisTextBase is now text.Text module

* All the erase and reset functionality removed from AxisText - not
  needed with double buffered drawing.  Ditto with state change.
  Text instances have a get_prop_tup method that returns a hashable
  tuple of text properties which you can use to see if text props
  have changed, e.g., by caching a font or layout instance in a dict
```

```
  with the prop tup as a key -- see RendererGTK.get_pango_layout in
  backend_gtk for an example.

* Text._get_xy_display renamed Text.get_xy_display

* Artist set_renderer and wash_brushes methods removed

* Moved Legend class from matplotlib.axes into matplotlib.legend

* Moved Tick, XTick, YTick, Axis, XAxis, YAxis from matplotlib.axes
  to matplotlib.axis

* moved process_text_args to matplotlib.text

* After getting Text handled in a backend independent fashion, the
  import process is much cleaner since there are no longer cyclic
  dependencies

* matplotlib.matlab._get_current_fig_manager renamed to
  matplotlib.matlab.get_current_fig_manager to allow user access to
  the GUI window attribute, e.g., figManager.window for GTK and
  figManager.frame for wx
```

38.39 Changes for 0.40

```
- Artist
    * __init__ takes a DPI instance and a Bound2D instance which is
      the bounding box of the artist in display coords
    * get_window_extent returns a Bound2D instance
    * set_size is removed; replaced by bbox and dpi
    * the clip_gc method is removed.  Artists now clip themselves with
      their box
    * added _clipOn boolean attribute.  If True, gc clip to bbox.

- AxisTextBase
    * Initialized with a transx, transy which are Transform instances
    * set_drawing_area removed
    * get_left_right and get_top_bottom are replaced by get_window_extent

- Line2D Patches now take transx, transy
    * Initialized with a transx, transy which are Transform instances

- Patches
   * Initialized with a transx, transy which are Transform instances

- FigureBase attributes dpi is a DPI intance rather than scalar and
  new attribute bbox is a Bound2D in display coords, and I got rid
  of the left, width, height, etc... attributes.  These are now
  accessible as, for example, bbox.x.min is left, bbox.x.interval()
  is width, bbox.y.max is top, etc...
```

```
- GcfBase attribute pagesize renamed to figsize

- Axes
    * removed figbg attribute
    * added fig instance to __init__
    * resizing is handled by figure call to resize.

- Subplot
    * added fig instance to __init__

- Renderer methods for patches now take gcEdge and gcFace instances.
  gcFace=None takes the place of filled=False

- True and False symbols provided by cbook in a python2.3 compatible
  way

- new module transforms supplies Bound1D, Bound2D and Transform
  instances and more

- Changes to the MATLAB helpers API

  * _matlab_helpers.GcfBase is renamed by Gcf.  Backends no longer
    need to derive from this class.  Instead, they provide a factory
    function new_figure_manager(num, figsize, dpi).  The destroy
    method of the GcfDerived from the backends is moved to the derived
    FigureManager.

  * FigureManagerBase moved to backend_bases

  * Gcf.get_all_figwins renamed to Gcf.get_all_fig_managers

Jeremy:

  Make sure to self._reset = False in AxisTextWX._set_font.  This was
  something missing in my backend code.
```

CHAPTER

THIRTYNINE

THE TOP LEVEL MATPLOTLIB MODULE

matplotlib.**use**(*arg*, *warn=True*, *force=False*)

Set the matplotlib backend to one of the known backends.

The argument is case-insensitive. *warn* specifies whether a warning should be issued if a backend has already been set up. *force* is an **experimental** flag that tells matplotlib to attempt to initialize a new backend by reloading the backend module.

Note: This function must be called *before* importing pyplot for the first time; or, if you are not using pyplot, it must be called before importing matplotlib.backends. If warn is True, a warning is issued if you try and call this after pylab or pyplot have been loaded. In certain black magic use cases, e.g. `pyplot.switch_backend()`, we are doing the reloading necessary to make the backend switch work (in some cases, e.g., pure image backends) so one can set warn=False to suppress the warnings.

To find out which backend is currently set, see `matplotlib.get_backend()`.

matplotlib.**get_backend**()

Return the name of the current backend.

matplotlib.**rcParams**

An instance of `RcParams` for handling default matplotlib values.

matplotlib.**rc**(*group*, ***kwargs*)

Set the current rc params. Group is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, e.g.,:

```
rc('lines', linewidth=2, color='r')
```

sets the current rc params and is equivalent to:

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

Alias	Property
'lw'	'linewidth'
'ls'	'linestyle'
'c'	'color'
'fc'	'facecolor'
'ec'	'edgecolor'
'mew'	'markeredgewidth'
'aa'	'antialiased'

Thus you could abbreviate the above rc command as:

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```
font = {'family' : 'monospace',
        'weight' : 'bold',
        'size'   : 'larger'}

rc('font', **font)  # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use *rcdefaults()* to restore the default rc params after changes.

matplotlib.**matplotlib_fname**()

Get the location of the config file.

The file location is determined in the following order

- `$PWD/matplotlibrc`
- `$MATPLOTLIBRC/matplotlibrc`
- `$MPLCONFIGDIR/matplotlibrc`
- On Linux,
  - `$HOME/.matplotlib/matplotlibrc`, if it exists
  - or `$XDG_CONFIG_HOME/matplotlib/matplotlibrc` (if $XDG_CONFIG_HOME is defined)
  - or `$HOME/.config/matplotlib/matplotlibrc` (if $XDG_CONFIG_HOME is not defined)
- On other platforms,
  - `$HOME/.matplotlib/matplotlibrc` if `$HOME` is defined.
- Lastly, it looks in `$MATPLOTLIBDATA/matplotlibrc` for a system-defined copy.

class matplotlib.**RcParams**(**args*, ***kwargs*)

A dictionary object including validation

validating functions are defined and associated with rc parameters in `matplotlib.rcsetup`

matplotlib.**rc_params**(*fail_on_error=False*)

Return a *matplotlib.RcParams* instance from the default matplotlib rc file.

matplotlib.**rc_params_from_file**(*fname*, *fail_on_error=False*, *use_default_template=True*)

Return *matplotlib.RcParams* from the contents of the given file.

Parameters fname : str
Name of file parsed for matplotlib settings.

fail_on_error : bool
If True, raise an error when the parser fails to convert a parameter.

use_default_template : bool
If True, initialize with default parameters before updating with those in the given file. If False, the configuration class only contains the parameters specified in the file. (Useful for updating dicts.)

class `matplotlib.rc_context`(*rc=None*, *fname=None*)
Return a context manager for managing rc settings.

This allows one to do:

```
with mpl.rc_context(fname='screen.rc'):
    plt.plot(x, a)
    with mpl.rc_context(fname='print.rc'):
        plt.plot(x, b)
    plt.plot(x, c)
```

The 'a' vs 'x' and 'c' vs 'x' plots would have settings from 'screen.rc', while the 'b' vs 'x' plot would have settings from 'print.rc'.

A dictionary can also be passed to the context manager:

```
with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
    plt.plot(x, a)
```

The 'rc' dictionary takes precedence over the settings loaded from 'fname'. Passing a dictionary only is also valid.

CHAPTER

FORTY

AFM (ADOBE FONT METRICS INTERFACE)

40.1 matplotlib.afm

This is a python interface to Adobe Font Metrics Files. Although a number of other python implementations exist, and may be more complete than this, it was decided not to go with them because they were either:

1. copyrighted or used a non-BSD compatible license
2. had too many dependencies and a free standing lib was needed
3. Did more than needed and it was easier to write afresh rather than figure out how to get just what was needed.

It is pretty easy to use, and requires only built-in python libs:

```
>>> from matplotlib import rcParams
>>> import os.path
>>> afm_fname = os.path.join(rcParams['datapath'],
...                          'fonts', 'afm', 'ptmr8a.afm')
>>>
>>> from matplotlib.afm import AFM
>>> afm = AFM(open(afm_fname))
>>> afm.string_width_height('What the heck?')
(6220.0, 694)
>>> afm.get_fontname()
'Times-Roman'
>>> afm.get_kern_dist('A', 'f')
0
>>> afm.get_kern_dist('A', 'y')
-92.0
>>> afm.get_bbox_char('!')
[130, -9, 238, 676]
```

class `matplotlib.afm.`**`AFM`**(*fh*)

Bases: `object`

Parse the AFM file in file object *fh*

`get_angle`()

Return the fontangle as float

get_bbox_char(*c*, *isord=False*)

get_capheight()
Return the cap height as float

get_familyname()
Return the font family name, e.g., 'Times'

get_fontname()
Return the font name, e.g., 'Times-Roman'

get_fullname()
Return the font full name, e.g., 'Times-Roman'

get_height_char(*c*, *isord=False*)
Get the height of character *c* from the bounding box. This is the ink height (space is 0)

get_horizontal_stem_width()
Return the standard horizontal stem width as float, or *None* if not specified in AFM file.

get_kern_dist(*c1*, *c2*)
Return the kerning pair distance (possibly 0) for chars *c1* and *c2*

get_kern_dist_from_name(*name1*, *name2*)
Return the kerning pair distance (possibly 0) for chars *name1* and *name2*

get_name_char(*c*, *isord=False*)
Get the name of the character, i.e., ';' is 'semicolon'

get_str_bbox(*s*)
Return the string bounding box

get_str_bbox_and_descent(*s*)
Return the string bounding box

get_underline_thickness()
Return the underline thickness as float

get_vertical_stem_width()
Return the standard vertical stem width as float, or *None* if not specified in AFM file.

get_weight()
Return the font weight, e.g., 'Bold' or 'Roman'

get_width_char(*c*, *isord=False*)
Get the width of the character from the character metric WX field

get_width_from_char_name(*name*)
Get the width of the character from a type1 character name

get_xheight()
Return the xheight as float

string_width_height(*s*)
Return the string width (including kerning) and string height as a (*w*, *h*) tuple.

`matplotlib.afm.parse_afm(fh)`

Parse the Adobe Font Metics file in file handle *fh*. Return value is a (*dhead*, *dcmetrics*, *dkernpairs*, *dcomposite*) tuple where *dhead* is a `_parse_header()` dict, *dcmetrics* is a `_parse_composites()` dict, *dkernpairs* is a `_parse_kern_pairs()` dict (possibly {}), and *dcomposite* is a `_parse_composites()` dict (possibly {})

CHAPTER

FORTYONE

ANIMATION

41.1 matplotlib.animation

class `matplotlib.animation.`**`AVConvBase`**

Bases: *`matplotlib.animation.FFMpegBase`*

args_key = 'animation.avconv_args'

exec_key = 'animation.avconv_path'

class `matplotlib.animation.`**`AVConvFileWriter`**(**args*, ***kwargs*)

Bases: *`matplotlib.animation.AVConvBase`*, *`matplotlib.animation.FFMpegFileWriter`*

class `matplotlib.animation.`**`AVConvWriter`**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Bases: *`matplotlib.animation.AVConvBase`*, *`matplotlib.animation.FFMpegWriter`*

Construct a new MovieWriter object.

fps: int Framerate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam `animation.codec` is used.

bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam `animation.bitrate`. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the 'animation.extra_args' rcParam.

metadata: dict of string:string or None A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

class `matplotlib.animation.`**`Animation`**(*fig*, *event_source=None*, *blit=False*)

Bases: `object`

This class wraps the creation of an animation using matplotlib. It is only a base class which should be subclassed to provide needed behavior.

fig is the figure object that is used to get draw, resize, and any other needed events.

event_source is a class that can run a callback when desired events are generated, as well as be stopped and started. Examples include timers (see `TimedAnimation`) and file system notifications.

blit is a boolean that controls whether blitting is used to optimize drawing.

new_frame_seq()
Creates a new sequence of frame information.

new_saved_frame_seq()
Creates a new sequence of saved/cached frame information.

save(*filename*, *writer=None*, *fps=None*, *dpi=None*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*, *extra_anim=None*, *savefig_kwargs=None*)
Saves a movie file by drawing every frame.

filename is the output filename, e.g., `mymovie.mp4`

writer is either an instance of `MovieWriter` or a string key that identifies a class to use, such as 'ffmpeg' or 'mencoder'. If nothing is passed, the value of the rcparam `animation.writer` is used.

dpi controls the dots per inch for the movie frames. This combined with the figure's size in inches controls the size of the movie.

savefig_kwargs is a dictionary containing keyword arguments to be passed on to the 'savefig' command which is called repeatedly to save the individual frames. This can be used to set tight bounding boxes, for example.

extra_anim is a list of additional `Animation` objects that should be included in the saved movie file. These need to be from the same `matplotlib.Figure` instance. Also, animation frames will just be simply combined, so there should be a 1:1 correspondence between the frames from the different animations.

These remaining arguments are used to construct a `MovieWriter` instance when necessary and are only considered valid if *writer* is not a `MovieWriter` instance.

fps is the frames per second in the movie. Defaults to None, which will use the animation's specified interval to set the frames per second.

codec is the video codec to be used. Not all codecs are supported by a given `MovieWriter`. If none is given, this defaults to the value specified by the rcparam `animation.codec`.

bitrate specifies the amount of bits used per second in the compressed movie, in kilobits per second. A higher number means a higher quality movie, but at the cost of increased file size. If no value is given, this defaults to the value given by the rcparam `animation.bitrate`.

extra_args is a list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the 'animation.extra_args' rcParam.

metadata is a dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

to_html5_video()
Returns animation as an HTML5 video tag.

This saves the animation as an h264 video, encoded in base64 directly into the HTML5 video tag. This respects the rc parameters for the writer as well as the bitrate. This also makes use of the `interval` to control the speed, and uses the `repeat` parameter to decide whether to loop.

class `matplotlib.animation.`**`ArtistAnimation`**(*fig*, *artists*, **args*, ***kwargs*)

Bases: `matplotlib.animation.TimedAnimation`

Before calling this function, all plotting should have taken place and the relevant artists saved.

frame_info is a list, with each list entry a collection of artists that represent what needs to be enabled on each frame. These will be disabled for other frames.

class `matplotlib.animation.`**`FFMpegBase`**

Bases: `object`

`args_key` **= 'animation.ffmpeg_args'**

`exec_key` **= 'animation.ffmpeg_path'**

`output_args`

class `matplotlib.animation.`**`FFMpegFileWriter`**(**args*, ***kwargs*)

Bases: `matplotlib.animation.FileMovieWriter`, `matplotlib.animation.FFMpegBase`

`supported_formats` **= ['png', 'jpeg', 'ppm', 'tiff', 'sgi', 'bmp', 'pbm', 'raw', 'rgba']**

class `matplotlib.animation.`**`FFMpegWriter`**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Bases: `matplotlib.animation.MovieWriter`, `matplotlib.animation.FFMpegBase`

Construct a new MovieWriter object.

fps: int Framerate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam `animation.codec` is used.

bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam `animation.bitrate`. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the 'animation.extra_args' rcParam.

metadata: dict of string:string or None A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

class `matplotlib.animation.`**`FileMovieWriter`**(**args*, ***kwargs*)

Bases: `matplotlib.animation.MovieWriter`

`MovieWriter` subclass that handles writing to a file.

cleanup()

finish()

frame_format

Format (png, jpeg, etc.) to use for saving the frames, which can be decided by the individual subclasses.

frame_size_can_vary = True

grab_frame(***savefig_kwargs*)

Grab the image information from the figure and save as a movie frame. All keyword arguments in savefig_kwargs are passed on to the 'savefig' command that saves the figure.

setup(*fig*, *outfile*, *dpi*, *frame_prefix='_tmp'*, *clear_temp=True*)

Perform setup for writing the movie file.

fig: `matplotlib.Figure` instance The figure object that contains the information for frames

outfile: string The filename of the resulting movie file

dpi: int The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.

frame_prefix: string, optional The filename prefix to use for the temporary files. Defaults to '_tmp'

clear_temp: bool Specifies whether the temporary files should be deleted after the movie is written. (Useful for debugging.) Defaults to True.

class matplotlib.animation.**FuncAnimation**(*fig*, *func*, *frames=None*, *init_func=None*, *fargs=None*, *save_count=None*, ***kwargs*)

Bases: `matplotlib.animation.TimedAnimation`

Makes an animation by repeatedly calling a function *func*, passing in (optional) arguments in *fargs*.

frames can be a generator, an iterable, or a number of frames.

init_func is a function used to draw a clear frame. If not given, the results of drawing from the first item in the frames sequence will be used. This function will be called once before the first frame.

If blit=True, *func* and *init_func* must return an iterable of artists to be re-drawn.

kwargs include *repeat*, *repeat_delay*, and *interval*: *interval* draws a new frame every *interval* milliseconds. *repeat* controls whether the animation should repeat when the sequence of frames is completed. *repeat_delay* optionally adds a delay in milliseconds before repeating the animation.

new_frame_seq()

new_saved_frame_seq()

class matplotlib.animation.**ImageMagickBase**

Bases: `object`

args_key = 'animation.convert_args'

delay

exec_key = 'animation.convert_path'

output_args

class matplotlib.animation.**ImageMagickFileWriter**(**args*, ***kwargs*)
Bases: *matplotlib.animation.FileMovieWriter*, *matplotlib.animation.ImageMagickBase*

supported_formats = ['png', 'jpeg', 'ppm', 'tiff', 'sgi', 'bmp', 'pbm', 'raw', 'rgba']

class matplotlib.animation.**ImageMagickWriter**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)
Bases: *matplotlib.animation.MovieWriter*, *matplotlib.animation.ImageMagickBase*

Construct a new MovieWriter object.

fps: int Framerate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam `animation.codec` is used.

bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam `animation.bitrate`. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the 'animation.extra_args' rcParam.

metadata: dict of string:string or None A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

class matplotlib.animation.**MencoderBase**
Bases: `object`

allowed_metadata = ['name', 'artist', 'genre', 'subject', 'copyright', 'srcform', 'comment']

args_key = 'animation.mencoder_args'

exec_key = 'animation.mencoder_path'

output_args

class `matplotlib.animation.`**`MencoderFileWriter`**(**args*, ***kwargs*)

Bases: `matplotlib.animation.FileMovieWriter`, `matplotlib.animation.MencoderBase`

`supported_formats` **= ['png', 'jpeg', 'tga', 'sgi']**

class `matplotlib.animation.`**`MencoderWriter`**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Bases: `matplotlib.animation.MovieWriter`, `matplotlib.animation.MencoderBase`

Construct a new MovieWriter object.

fps: int Framerate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam `animation.codec` is used.

bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam `animation.bitrate`. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the 'animation.extra_args' rcParam.

metadata: dict of string:string or None A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

class `matplotlib.animation.`**`MovieWriter`**(*fps=5*, *codec=None*, *bitrate=None*, *extra_args=None*, *metadata=None*)

Bases: `object`

Base class for writing movies. Fundamentally, what a MovieWriter does is provide is a way to grab frames by calling grab_frame(). setup() is called to start the process and finish() is called afterwards. This class is set up to provide for writing movie frame data to a pipe. saving() is provided as a context manager to facilitate this process as:

```
with moviewriter.saving('myfile.mp4'):
    # Iterate over frames
    moviewriter.grab_frame()
```

The use of the context manager ensures that setup and cleanup are performed as necessary.

frame_format: string The format used in writing frame data, defaults to 'rgba'

Construct a new MovieWriter object.

fps: int Framerate for movie.

codec: string or None, optional The codec to use. If None (the default) the setting in the rcParam `animation.codec` is used.

bitrate: int or None, optional The bitrate for the saved movie file, which is one way to control the output file size and quality. The default value is None, which uses the value stored in the rcParam `animation.bitrate`. A value of -1 implies that the bitrate should be determined automatically by the underlying utility.

extra_args: list of strings or None A list of extra string arguments to be passed to the underlying movie utility. The default is None, which passes the additional arguments in the 'animation.extra_args' rcParam.

metadata: dict of string:string or None A dictionary of keys and values for metadata to include in the output file. Some keys that may be of use include: title, artist, genre, subject, copyright, srcform, comment.

classmethod bin_path()
Returns the binary path to the commandline tool used by a specific subclass. This is a class method so that the tool can be looked for before making a particular MovieWriter subclass available.

cleanup()
Clean-up and collect the process used to write the movie file.

finish()
Finish any processing for writing the movie.

frame_size
A tuple (width,height) in pixels of a movie frame.

frame_size_can_vary = False

grab_frame(***savefig_kwargs*)
Grab the image information from the figure and save as a movie frame. All keyword arguments in savefig_kwargs are passed on to the 'savefig' command that saves the figure.

classmethod isAvailable()
Check to see if a MovieWriter subclass is actually available by running the commandline tool.

saving(**args*)
Context manager to facilitate writing the movie file.

`*args` are any parameters that should be passed to `setup`.

setup(*fig*, *outfile*, *dpi*, **args*)
Perform setup for writing the movie file.
fig: matplotlib.Figure instance The figure object that contains the information for frames
outfile: string The filename of the resulting movie file
dpi: int The DPI (or resolution) for the file. This controls the size in pixels of the resulting movie file.

class matplotlib.animation.**MovieWriterRegistry**
Bases: `object`

is_available(*name*)

list()
Get a list of available MovieWriters.

register(*name*)

class matplotlib.animation.**TimedAnimation**(*fig*, *interval=200*, *repeat_delay=None*, *repeat=True*, *event_source=None*, **args*, ***kwargs*)
Bases: `matplotlib.animation.Animation`

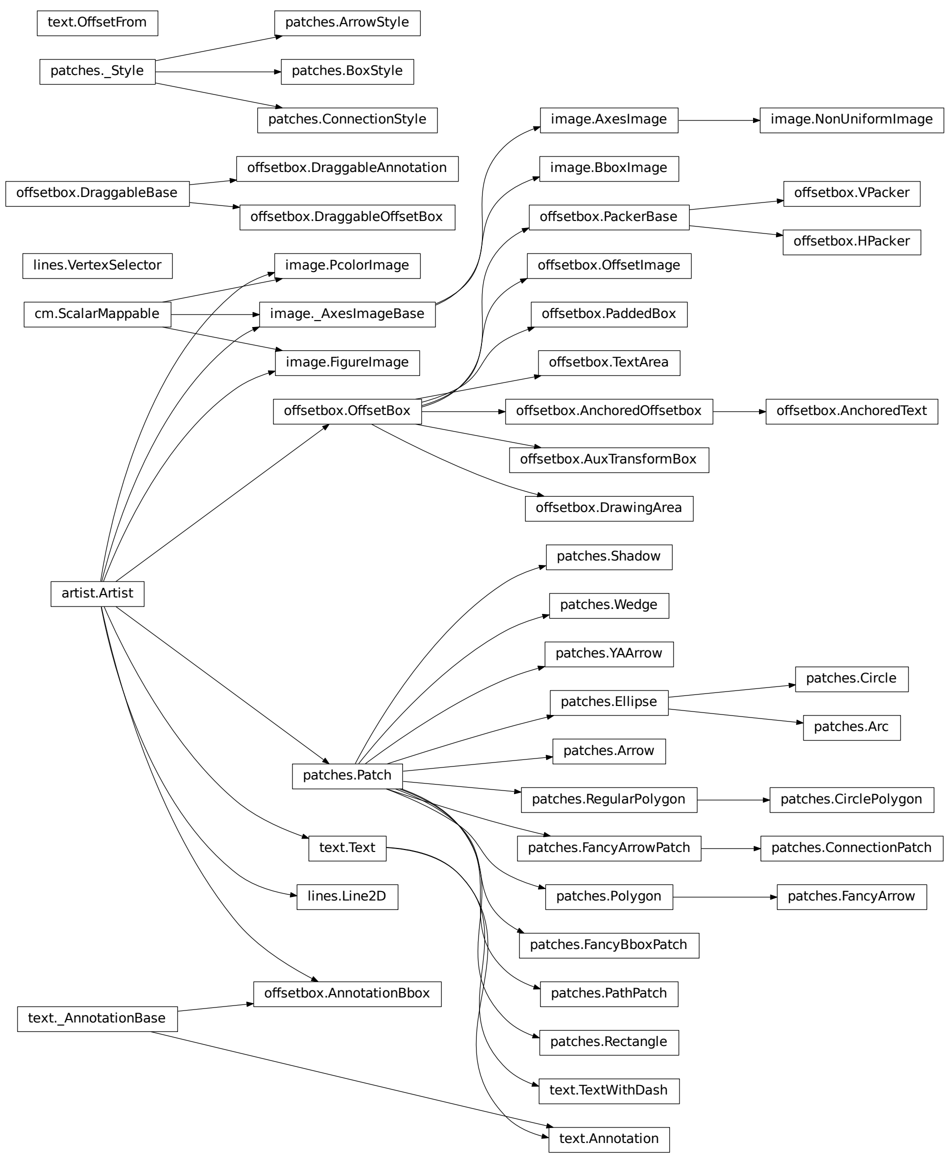
`Animation` subclass that supports time-based animation, drawing a new frame every *interval* milliseconds.

repeat controls whether the animation should repeat when the sequence of frames is completed.

repeat_delay optionally adds a delay in milliseconds before repeating the animation.

CHAPTER FORTYTWO

ARTISTS



42.1 matplotlib.artist

class matplotlib.artist.**Artist**

Bases: object

Abstract base class for someone who renders into a FigureCanvas.

add_callback(*func*)

Adds a callback function that will be called whenever one of the *Artist*'s properties changes.

Returns an *id* that is useful for removing the callback with *remove_callback()* later.

aname = 'Artist'

axes

The *Axes* instance the artist resides in, or *None*.

contains(*mouseevent*)

Test whether the artist contains the mouse event.

Returns the truth value and a dictionary of artist specific details of selection, such as which points are contained in the pick radius. See individual artists for details.

convert_xunits(*x*)

For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)

For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*renderer*, **args*, ***kwargs*)

Derived classes drawing method

findobj(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all *Artist* instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature boolean = match(artist) used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)

Return *cursor data* string formatted.

get_agg_filter()

return filter function to be used for agg filter

get_alpha()

Return the alpha value used for blending - not supported on all backends

get_animated()

Return the artist's animated state

get_axes()
Return the *Axes* instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_figure()
Return the *Figure* instance the artist belongs to.

get_gid()
Returns the group id

get_label()
Get the label used for this artist in the legend.

get_path_effects()

get_picker()
Return the picker object used by this artist

get_rasterized()
return True if the artist is to be rasterized

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center

- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_url()
Returns the url

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)
Get the axes bounding box in display space. Subclasses should override for inclusion in the bounding box "tight" calculation. Default is to return an empty bounding box at 0, 0.

Be careful when using this function, the results will not update if the artist window extent of the artist changes. The extent can change due to any changes in the transform stack, such as changing the axes limits, the figure size, or the canvas used (as is done when saving a figure). This can lead to unexpected behavior where interactive figures will look fine on the screen, but will save incorrectly.

get_zorder()
Return the *Artist*'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if *Artist* has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha value used for blending - not supported on all backends.

ACCEPTS: float (0.0 transparent through 1.0 opaque)

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)

Set the artist's clip path, which may be:

- a Patch (or subclass) instance
- **a Path instance, in which case** an optional Transform instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(Path, Transform) | Patch | None]

set_contains(*picker*)

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_figure(*fig*)

Set the Figure instance the artist belongs to.

ACCEPTS: a matplotlib.figure.Figure instance

set_gid(*gid*)

Sets the (group) id for the artist

ACCEPTS: an id string

set_label(*s*)

Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_path_effects(*path_effects*)

set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_picker(*picker*)

Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event

- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_rasterized(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)

Sets the sketch parameters.

Parameters scale : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)

Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)

Sets the url for the artist

ACCEPTS: a url string

set_visible(*b*)

Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
Copy properties from *other* to *self*.

zorder = 0

class `matplotlib.artist.`**ArtistInspector**(*o*)
Bases: `object`

A helper class to inspect an `Artist` and return information about it's settable properties and their current values.

Initialize the artist inspector with an `Artist` or sequence of `Artists`. If a sequence is used, we assume it is a homogeneous sequence (all `Artists` are of the same type) and it is your responsibility to make sure this is so.

aliased_name(*s*)
return 'PROPNAME or alias' if *s* has an alias, else return PROPNAME.

e.g., for the line markerfacecolor property, which has an alias, return 'markerfacecolor or mfc' and for the transform property, which does not, return 'transform'

aliased_name_rest(*s*, *target*)
return 'PROPNAME or alias' if *s* has an alias, else return PROPNAME formatted for ReST

e.g., for the line markerfacecolor property, which has an alias, return 'markerfacecolor or mfc' and for the transform property, which does not, return 'transform'

findobj(*match=None*)
Recursively find all `matplotlib.artist.Artist` instances contained in *self*.

If *match* is not None, it can be

- function with signature `boolean = match(artist)`
- class instance: e.g., `Line2D`

used to filter matches.

get_aliases()
Get a dict mapping *fullname* -> *alias* for each *alias* in the `ArtistInspector`.

e.g., for lines:

```
{'markerfacecolor': 'mfc',
 'linewidth'      : 'lw',
}
```

get_setters()
Get the attribute strings with setters for object. e.g., for a line, return `['markerfacecolor', 'linewidth', ....]`.

get_valid_values(*attr*)
Get the legal arguments for the setter associated with *attr*.

This is done by querying the docstring of the function *set_attr* for a line that begins with ACCEPTS:

e.g., for a line linestyle, return "[`'-'` | `'--'` | `'-.'` | `':'` | `'steps'` | `'None'`]"

is_alias(*o*)
Return *True* if method object *o* is an alias for another function.

pprint_getters()
Return the getters and actual values as list of strings.

pprint_setters(*prop=None*, *leadingspace=2*)
If *prop* is *None*, return a list of strings of all settable properies and their valid values.

If *prop* is not *None*, it is a valid property name and that property will be returned as a string of property : valid values.

pprint_setters_rest(*prop=None*, *leadingspace=2*)
If *prop* is *None*, return a list of strings of all settable properies and their valid values. Format the output for ReST

If *prop* is not *None*, it is a valid property name and that property will be returned as a string of property : valid values.

properties()
return a dictionary mapping property name -> value

matplotlib.artist.**allow_rasterization**(*draw*)
Decorator for Artist.draw method. Provides routines that run before and after the draw call. The before and after functions are useful for changing artist-dependant renderer attributes or making other setup function calls, such as starting and flushing a mixed-mode renderer.

matplotlib.artist.**get**(*obj*, *property=None*)
Return the value of object's property. *property* is an optional string for the property you want to return

Example usage:

```
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

obj is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the *property* is 'somename', this function returns

obj.get_somename()

`getp()` can be used to query all the gettable properties with `getp(obj)`. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:
linewidth or lw = 2

matplotlib.artist.**getp**(*obj*, *property=None*)

Return the value of object's property. *property* is an optional string for the property you want to return

Example usage:

```
getp(obj)  # get all the object properties
getp(obj, 'linestyle')  # get the linestyle property
```

obj is a `Artist` instance, e.g., `Line2D` or an instance of a `Axes` or `matplotlib.text.Text`. If the *property* is 'somename', this function returns

obj.get_somename()

`getp()` can be used to query all the gettable properties with `getp(obj)`. Many properties have aliases for shorter typing, e.g. 'lw' is an alias for 'linewidth'. In the output, aliases and full property names will be listed as:

property or alias = value

e.g.:

linewidth or lw = 2

matplotlib.artist.**kwdoc**(*a*)

matplotlib.artist.**setp**(*obj*, **args*, ***kwargs*)

Set a property on an artist object.

matplotlib supports the use of `setp()` ("set property") and `getp()` to set and get object properties, as well as to do introspection on the object. For example, to set the linestyle of a line to be dashed, you can do:

```
>>> line, = plot([1,2,3])
>>> setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value:

```
>>> setp(line, 'linestyle')
    linestyle: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]
```

If you want to see all the properties that can be set, and their possible values, you can do:

```
>>> setp(line)
    ... long output listing omitted
```

`setp()` operates on a single instance or a list of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. e.g., suppose you have a list of two lines, the following will make both lines thicker and red:

```
>>> x = arange(0,1.0,0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

`setp()` works with the MATLAB style string/value pairs or with python kwargs. For example, the following are equivalent:

```
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')        # python style
```

CHAPTER

FORTYTHREE

AXES

43.1 matplotlib.axes

class `matplotlib.axes.`**`Axes`**(*fig*, *rect*, *axisbg=None*, *frameon=True*, *sharex=None*, *sharey=None*, *label=''*, *xscale=None*, *yscale=None*, ***kwargs*)

The `Axes` contains most of the figure elements: *`Axis`*, *`Tick`*, *`Line2D`*, *`Text`*, *`Polygon`*, etc., and sets the coordinate system.

The `Axes` instance supports callbacks through a callbacks attribute which is a *`CallbackRegistry`* instance. The events you can connect to are 'xlim_changed' and 'ylim_changed' and the callback will be called with func(*ax*) where *ax* is the `Axes` instance.

`acorr`(*x*, ***kwargs*)

Plot the autocorrelation of `x`.

Parameters x : sequence of scalar

hold : boolean, optional, default: True

detrend : callable, optional, default: `mlab.detrend_none`
x is detrended by the `detrend` callable. Default is no normalization.

normed : boolean, optional, default: True
if True, normalize the data by the autocorrelation at the 0-th lag.

usevlines : boolean, optional, default: True
if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.

maxlags : integer, optional, default: 10
number of lags to show. If None, will return all 2 * len(x) - 1 lags.

Returns (lags, c, line, b) : where:

- `lags` are a length 2'maxlags+1 lag vector.
- `c` is the 2'maxlags+1 auto correlation vectorI
- `line` is a *`Line2D`* instance returned by `plot`.
- `b` is the x-axis.

Other Parameters linestyle : *`Line2D`* prop, optional, default: None
Only used if usevlines is False.

marker : string, optional, default: 'o'

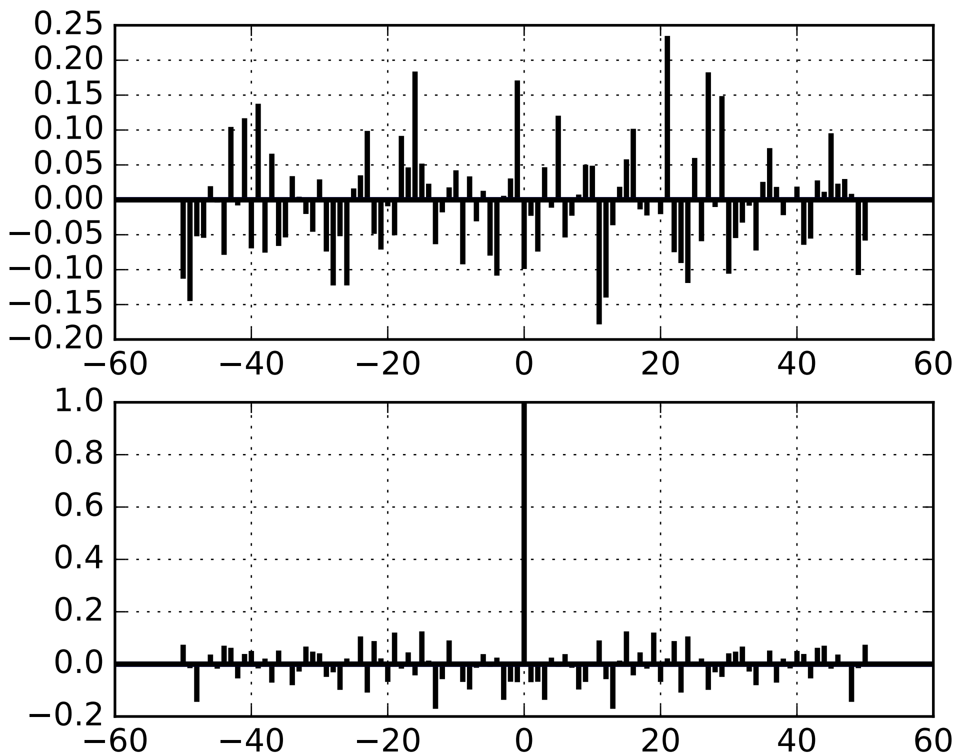
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

Examples

xcorr is top graph, and *acorr* is bottom graph.



add_artist(*a*)

Add any *Artist* to the axes.

Use `add_artist` only for artists for which there is no dedicated "add" method; and if necessary, use a method such as `update_datalim` or `update_datalim_numerix` to manually update the dataLim if the artist is to be included in autoscaling.

Returns the artist.

add_callback(*func*)

Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_collection(*collection*, *autolim=True*)

Add a *Collection* instance to the axes.

Returns the collection.

add_container(*container*)

Add a `Container` instance to the axes.

Returns the collection.

add_image(*image*)

Add a *AxesImage* to the axes.

Returns the image.

add_line(*line*)

Add a *Line2D* to the list of plot lines

Returns the line.

add_patch(*p*)

Add a *Patch* *p* to the list of axes patches; the clipbox will be set to the Axes clipping box. If the transform is not set, it will be set to `transData`.

Returns the patch.

add_table(*tab*)

Add a `Table` instance to the list of axes tables

Returns the table.

aname = 'Artist'

angle_spectrum(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, ***kwargs*)

Plot the angle spectrum.

Call signature:

```
angle_spectrum(x, Fs=2, Fc=0,  window=mlab.window_hanning,
               pad_to=None, sides='default', **kwargs)
```

Compute the angle spectrum (wrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer**

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns the tuple (*spectrum*, *freqs*, *line*):

***spectrum*: 1-D array** The values for the angle spectrum in radians (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

***line*: a `Line2D` instance** The line created by this function

kwargs control the `Line2D` properties:

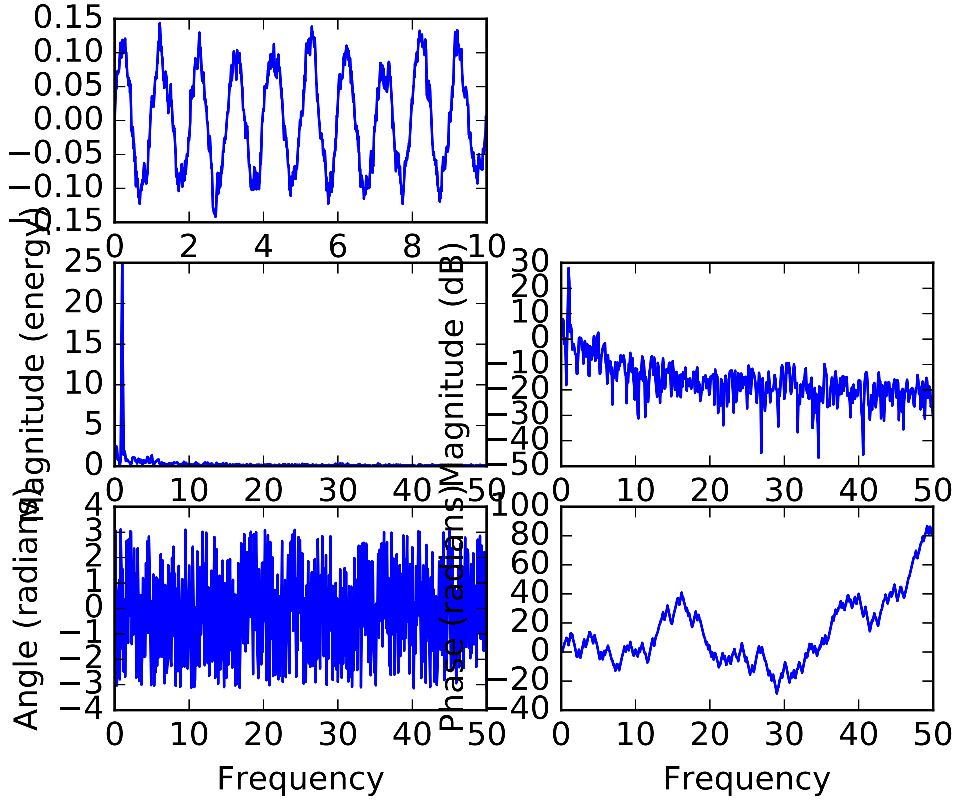
Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color

C

Table 43.1 – continued from previous page

Property	Description
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	[’butt’ | ‘round’ | ‘projecting’]
solid_joinstyle	[’miter’ | ‘round’ | ‘bevel’]
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:



See also:

magnitude_spectrum() `angle_spectrum()` plots the magnitudes of the corresponding frequencies.

phase_spectrum() `phase_spectrum()` plots the unwrapped version of this function.

specgram() `specgram()` can plot the angle spectrum of segments within the signal in a colormap.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

annotate(**args*, ***kwargs*)

Annotate the point `xy` with text `s`.

Additional kwargs are passed to `Text`.

Parameters s : str

The text of the annotation

xy : iterable

Length 2 sequence specifying the *(x,y)* point to annotate

xytext : iterable, optional

Length 2 sequence specifying the *(x,y)* to place the text at. If None, defaults to `xy`.

xycoords : str, Artist, Transform, callable or tuple, optional

The coordinate system that `xy` is given in.

For a `str` the allowed values are:

Property	Description
'figure points'	points from the lower left of the figure
'figure pixels'	pixels from the lower left of the figure
'figure fraction'	fraction of figure from lower left
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	fraction of axes from lower left
'data'	use the coordinate system of the object being annotated (default)
'polar'	*(theta,r)* if not native 'data' coordinates

If a `Artist` object is passed in the units are fraction if it's bounding box.

If a `Transform` object is passed in use that to transform `xy` to screen coordinates

If a callable it must take a `RendererBase` object as input and return a `Transform` or `Bbox` object

If a `tuple` must be length 2 tuple of str, `Artist`, `Transform` or callable objects. The first transform is used for the *x* coordinate and the second for *y*.

See *Annotating Axes* for more details.

Defaults to `'data'`

textcoords : str, `Artist`, `Transform`, callable or tuple, optional

The coordinate system that `xytext` is given in, which may be different than the coordinate system used for `xy`.

All `xycoords` values are valid as well as the following strings:

Property	Description
'offset points'	offset (in points) from the *xy* value
'offset pixels'	offset (in pixels) from the *xy* value

defaults to the input of `xycoords`

arrowprops : dict, optional

If not None, properties used to draw a *FancyArrowPatch* arrow between `xy` and `xytext`.

If `arrowprops` does not contain the key `'arrowstyle'` the allowed keys are:

Key	Description
width	the width of the arrow in points
headwidth	the width of the base of the arrow head in points
headlength	the length of the arrow head in points
shrink	fraction of total length to 'shrink' from both ends
?	any key to *matplotlib.patches.FancyArrowPatch*

If the `arrowprops` contains the key `'arrowstyle'` the above keys are forbidden. The allowed values of `'arrowstyle'` are:

Name	Attrs
`'-'`	None
`'->'`	head_length=0.4,head_width=0.2
`'-['`	widthB=1.0,lengthB=0.2,angleB=None
`'|-|'`	widthA=1.0,widthB=1.0
`'-|>'`	head_length=0.4,head_width=0.2
`'<-'`	head_length=0.4,head_width=0.2
`'<->'`	head_length=0.4,head_width=0.2
`'<|-'`	head_length=0.4,head_width=0.2
`'<|-|>'`	head_length=0.4,head_width=0.2
`'fancy'`	head_length=0.4,head_width=0.4,tail_width=0.4
`'simple'`	head_length=0.5,head_width=0.5,tail_width=0.2
`'wedge'`	tail_width=0.3,shrink_factor=0.5

Valid keys for *FancyArrowPatch* are:

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for *matplotlib.patches.PathPatch*

Defaults to None

annotation_clip : bool, optional

Controls the visibility of the annotation when it goes outside the axes area.

If `True`, the annotation will only be drawn when the `xy` is inside the axes. If `False`, the annotation will always be drawn regardless of its position.

The default is `None`, which behave as `True` only if *xycoords* is "data".

Returns Annotation

apply_aspect(*position=None*)

Use `_aspect()` and `_adjustable()` to modify the axes box or the view limits.

arrow(*x*, *y*, *dx*, *dy*, ***kwargs*)

Add an arrow to the axes.

Call signature:

```
arrow(x, y, dx, dy, **kwargs)
```

Draws arrow on specified axis from (*x*, *y*) to (*x* + *dx*, *y* + *dy*). Uses FancyArrow patch to construct the arrow.

The resulting arrow is affected by the axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem, use `annotate()` for example:

```
ax.annotate("", xy=(0.5, 0.5), xytext=(0, 0),
    arrowprops=dict(arrowstyle="->"))
```

Optional kwargs control the arrow construction and properties:

Constructor arguments

***width*: float (default: 0.001)** width of full arrow tail

***length_includes_head*: [True | False] (default: False)** True if head is to be counted in calculating the length.

***head_width*: float or None (default: 3*width)** total width of the full arrow head

***head_length*: float or None (default: 1.5 * head_width)** length of arrow head

***shape*: ['full', 'left', 'right'] (default: 'full')** draw the left-half, right-half, or full arrow

***overhang*: float (default: 0)** fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

***head_starts_at_zero*: [True | False] (default: False)** if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from `Patch`) are:

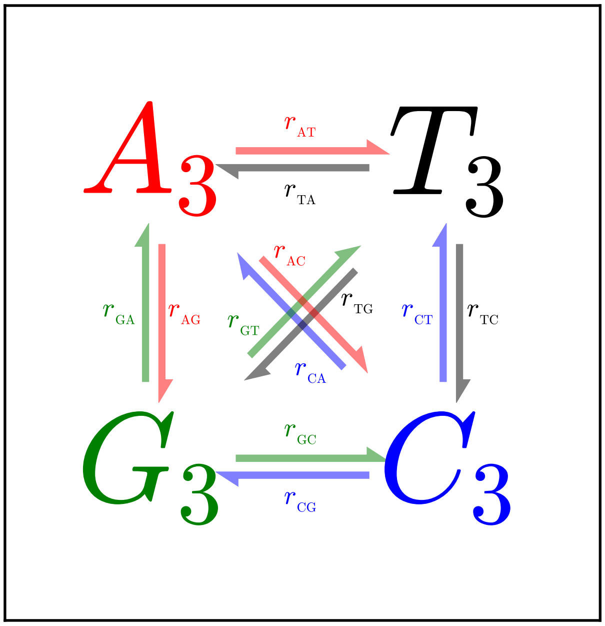
Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]

Continued on

Table 43.2 – continued from previous page

Property	Description
antialiased or aa	[True | False] or None for default
axes	an Axes instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example:



autoscale(*enable=True*, *axis='both'*, *tight=None*)

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes.

***enable*: [True | False | None]** True (default) turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

***axis*: ['x' | 'y' | 'both']** which axis to operate on; default is 'both'

***tight*: [True | False | None]** If True, set view limits to data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only artist is an image, otherwise treat *tight* as False. The *tight* setting is retained for future autoscaling until it is explicitly changed.

Returns None.

autoscale_view(*tight=None*, *scalex=True*, *scaley=True*)

Autoscale the view limits using the data limits. You can selectively autoscale only a single axis, e.g., the xaxis by setting *scaley* to *False*. The autoscaling preserves any axis direction reversal that has already been done.

The data limits are not updated automatically when artist data are changed after the artist has been added to an Axes instance. In that case, use `matplotlib.axes.Axes.relim()` prior to calling autoscale_view.

axes

The `Axes` instance the artist resides in, or *None*.

axhline(*y=0*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal line across the axis.

Parameters y : scalar, optional, default: 0
y position in data coordinates of the horizontal line.

xmin : scalar, optional, default: 0
Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

xmax : scalar, optional, default: 1
Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

Returns *Line2D*

See also:

axhspan for example plot and source code

Notes

kwargs are passed to *Line2D* and can be used to control the line properties.

Examples

- draw a thick red hline at 'y' = 0 that spans the xrange:

```
>>> axhline(linewidth=4, color='r')
```

- draw a default hline at 'y' = 1 that spans the xrange:

```
>>> axhline(y=1)
```

- draw a default hline at 'y' = .5 that spans the middle half of the xrange:

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

Valid kwargs are *Line2D* properties, with the exception of 'transform':

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an *Axes* instance
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
	C

Table 43.3 – continued from previous page

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>drawstyle</td><td>['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '</td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>marker</td><td>A valid marker style</td></tr>
<tr><td>markeredgecolor or mec</td><td>any matplotlib color</td></tr>
<tr><td>markeredgewidth or mew</td><td>float value in points</td></tr>
<tr><td>markerfacecolor or mfc</td><td>any matplotlib color</td></tr>
<tr><td>markerfacecoloralt or mfcalt</td><td>any matplotlib color</td></tr>
<tr><td>markersize or ms</td><td>float</td></tr>
<tr><td>markevery</td><td>[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>float distance in points or callable pick function <code>fn(artist, event)</code></td></tr>
<tr><td>pickradius</td><td>float distance in points</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>solid_capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>solid_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>transform</td><td>a matplotlib.transforms.Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>xdata</td><td>1D array</td></tr>
<tr><td>ydata</td><td>1D array</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

axhspan(*ymin*, *ymax*, *xmin=0*, *xmax=1*, ***kwargs*)

Add a horizontal span (rectangle) across the axis.

Call signature:

```
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
```

y coords are in data units and *x* coords are in axes (relative 0-1) units.

Draw a horizontal span (rectangle) from *ymin* to *ymax*. With the default values of *xmin* = 0 and *xmax* = 1, this always spans the xrange, regardless of the xlim settings, even if you change them, e.g., with the `set_xlim()` command. That is, the horizontal extent is in axes coords: 0=left, 0.5=middle, 1.0=right but the *y* location is in data coordinates.

Return value is a `matplotlib.patches.Polygon` instance.

Examples:

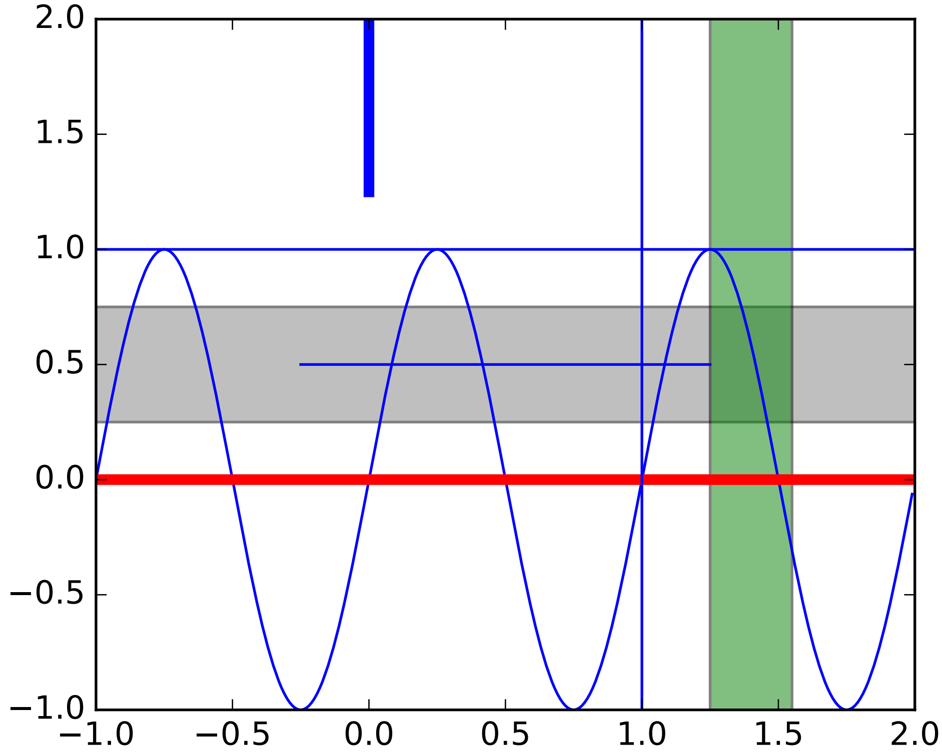
- draw a gray rectangle from $y = 0.25$-0.75 that spans the horizontal extent of the axes:

```
>>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
```

Valid kwargs are `Polygon` properties:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>agg_filter</td><td>unknown</td></tr>
<tr><td>alpha</td><td>float or None</td></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or aa</td><td>[True | False] or None for default</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>capstyle</td><td>[’butt’ | ‘round’ | ‘projecting’]</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color</td><td>matplotlib color spec</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or ec</td><td>mpl color spec, or None for default, or ‘none’ for no color</td></tr>
<tr><td>facecolor or fc</td><td>mpl color spec, or None for default, or ‘none’ for no color</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fill</td><td>[True | False]</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>hatch</td><td>[’/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]</td></tr>
<tr><td>joinstyle</td><td>[’miter’ | ‘round’ | ‘bevel’]</td></tr>
<tr><td>label</td><td>string or anything printable with ‘%s’ conversion.</td></tr>
<tr><td>linestyle or ls</td><td>[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'</td></tr>
<tr><td>linewidth or lw</td><td>float or None for default</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>transform</td><td>Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

Example:



axis(**v*, ***kwargs*)

Set axis properties.

Valid signatures:

```
xmin, xmax, ymin, ymax = axis()
xmin, xmax, ymin, ymax = axis(list_arg)
xmin, xmax, ymin, ymax = axis(string_arg)
xmin, xmax, ymin, ymax = axis(**kwargs)
```

Parameters v : list of float or {'on', 'off', 'equal', 'tight', 'scaled', 'normal', 'auto', 'image', 'square'}

Optional positional argument

Axis data limits set from a list; or a command relating to axes:

Value	Description
'on'	Toggle axis lines and labels on
'off'	Toggle axis lines and labels off
'equal'	Equal scaling by changing limits
'scaled'	Equal scaling by changing box dimensions
'tight'	Limits set such that all data is shown
'auto'	Automatic scaling, fill rectangle with data
'normal'	Same as 'auto'; deprecated
'image'	'scaled' with axis limits equal to data limits
'square'	Square plot; similar to 'scaled', but initially forcing xmax-xmin = ymax-ymin

emit : bool, optional
Passed to set_{x,y}lim functions, if observers are notified of axis limit change

xmin, ymin, xmax, ymax : float, optional
The axis limits to be set

Returns xmin, xmax, ymin, ymax : float
The axis limits

axvline(*x=0*, *ymin=0*, *ymax=1*, ***kwargs*)
Add a vertical line across the axes.

Parameters x : scalar, optional, default: 0
x position in data coordinates of the vertical line.

ymin : scalar, optional, default: 0
Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

ymax : scalar, optional, default: 1
Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

Returns `Line2D`

See also:

axhspan for example plot and source code

Examples

- draw a thick red vline at $x = 0$ that spans the yrange:

```
>>> axvline(linewidth=4, color='r')
```

- draw a default vline at $x = 1$ that spans the yrange:

```
>>> axvline(x=1)
```

- draw a default vline at $x = .5$ that spans the middle half of the yrange:

```
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

Valid kwargs are `Line2D` properties, with the exception of 'transform':

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
`path_effects`	unknown
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
	C

Table 43.5 – continued from previous page

Property	Description
ydata	1D array
zorder	any number

axvspan(*xmin*, *xmax*, *ymin=0*, *ymax=1*, ***kwargs*)

Add a vertical span (rectangle) across the axes.

Call signature:

```
axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
```

x coords are in data units and *y* coords are in axes (relative 0-1) units.

Draw a vertical span (rectangle) from *xmin* to *xmax*. With the default values of *ymin* = 0 and *ymax* = 1, this always spans the yrange, regardless of the ylim settings, even if you change them, e.g., with the `set_ylim()` command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the *y* location is in data coordinates.

Return value is the *matplotlib.patches.Polygon* instance.

Examples:

- draw a vertical green translucent rectangle from x=1.25 to 1.55 that spans the yrange of the axes:

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

Valid kwargs are *Polygon* properties:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']

Continued on

Table 43.6 – continued from previous page

Property	Description
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

See also:

`axhspan()` for example plot and source code

bar(*left*, *height*, *width=0.8*, *bottom=None*, ***kwargs*)

Make a bar plot.

Make a bar plot with rectangles bounded by:

left, left + width, bottom, bottom + height (left, right, bottom and top edges)

Parameters left : sequence of scalars
the x coordinates of the left sides of the bars

height : sequence of scalars
the heights of the bars

width : scalar or array-like, optional
the width(s) of the bars default: 0.8

bottom : scalar or array-like, optional
the y coordinate(s) of the bars default: None

color : scalar or array-like, optional
the colors of the bar faces

edgecolor : scalar or array-like, optional
the colors of the bar edges

linewidth : scalar or array-like, optional
width of bar edge(s). If None, use default linewidth; If 0, don't draw edges. default: None

tick_label : string or array-like, optional
the tick labels of the bars default: None

xerr : scalar or array-like, optional
if not None, will be used to generate errorbar(s) on the bar chart default: None

yerr : scalar or array-like, optional if not None, will be used to generate errorbar(s) on the bar chart default: None

ecolor : scalar or array-like, optional
specifies the color of errorbar(s) default: None

capsize : scalar, optional
determines the length in points of the error bar caps default: None, which will take the value from the `errorbar.capsize` *rcParam*.

error_kw : dict, optional
dictionary of kwargs to be passed to errorbar method. *ecolor* and *capsize* may be specified here rather than as independent kwargs.

align : {'edge', 'center'}, optional
If 'edge', aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If 'center', interpret the `left` argument as the coordinates of the centers of the bars. To align on the align bars on the right edge pass a negative `width`.

orientation : {'vertical', 'horizontal'}, optional
The orientation of the bars.

log : boolean, optional
If true, sets the axis to be log scale. default: False

Returns bars : matplotlib.container.BarContainer
Container with all of the bars + errorbars

See also:

barh Plot a horizontal bar plot.

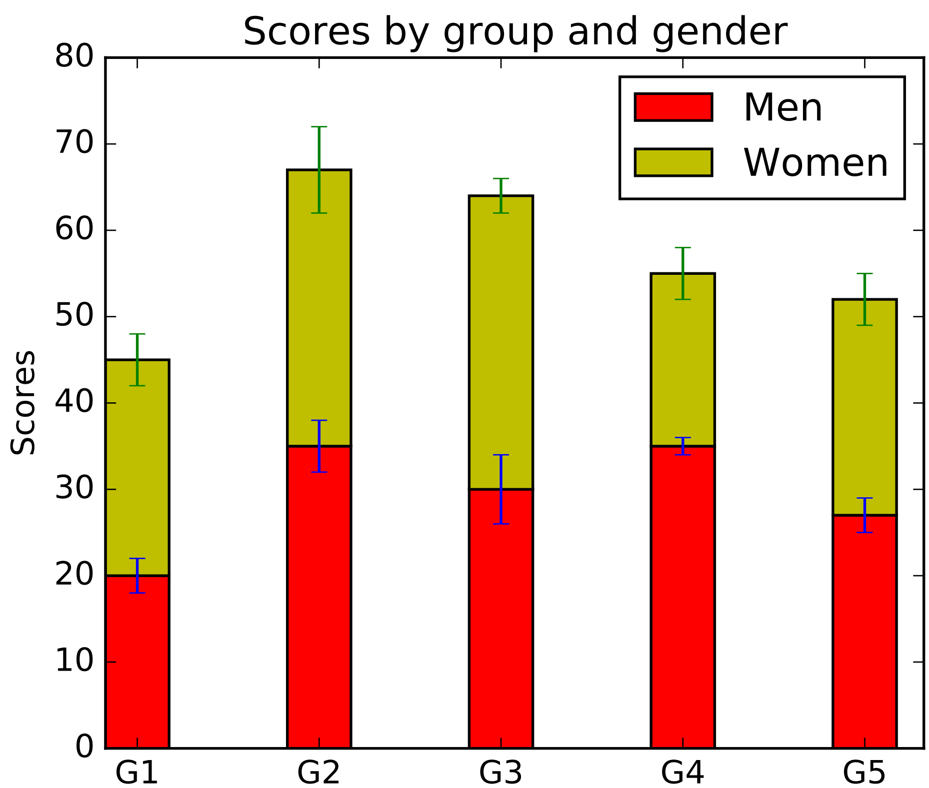
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'height', 'yerr', 'tick_label', 'xerr', 'width', 'color', 'ecolor', 'linewidth', 'left', 'bottom', 'edgecolor'.

Examples

Example: A stacked bar chart.



barbs(**args*, ***kw*)

Plot a 2-D field of barbs.

Call signatures:

```
barb(U, V, **kw)
barb(U, V, C, **kw)
barb(X, Y, U, V, **kw)
barb(X, Y, U, V, C, **kw)
```

Arguments:

***X, Y*:** The x and y coordinates of the barb locations (default is head of barb; see *pivot* kwarg)

***U, V*:** Give the x and y components of the barb shaft

***C*:** An optional array used to map colors to the barbs

All arguments may be 1-D or 2-D arrays or sequences. If *X* and *Y* are absent, they will be generated as a uniform grid. If *U* and *V* are 2-D arrays but *X* and *Y* are 1-D, and if `len(X)` and `len(Y)` match the column and row dimensions of *U*, then *X* and *Y* will be expanded with `numpy.meshgrid()`.

U, *V*, *C* may be masked arrays, but masked *X*, *Y* are not supported at present.

Keyword arguments:

***length*:** Length of the barb in points; the other parts of the barb are scaled against this. Default is 9

***pivot*: ['tip' | 'middle']** The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*. Default is 'tip'

***barbcolor*: [color | color sequence]** Specifies the color all parts of the barb except any flags. This parameter is analagous to the *edgecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor.

***flagcolor*: [color | color sequence]** Specifies the color of any flags on the barb. This parameter is analagous to the *facecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

***sizes*:** A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

***fill_empty*:** A flag on whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, they will be drawn such that no color is applied to the center. Default is False

***rounding*:** A flag to indicate whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple. Default is True

***barb_increments*:** A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

***flip_barb*:** Either a single boolean flag or an array of booleans. Single boolean indicates whether the lines and flags should point opposite to normal for all barbs. An array (which should be the same size as the other data arrays) indicates whether to flip for each individual barb. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere.) Default is False

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:

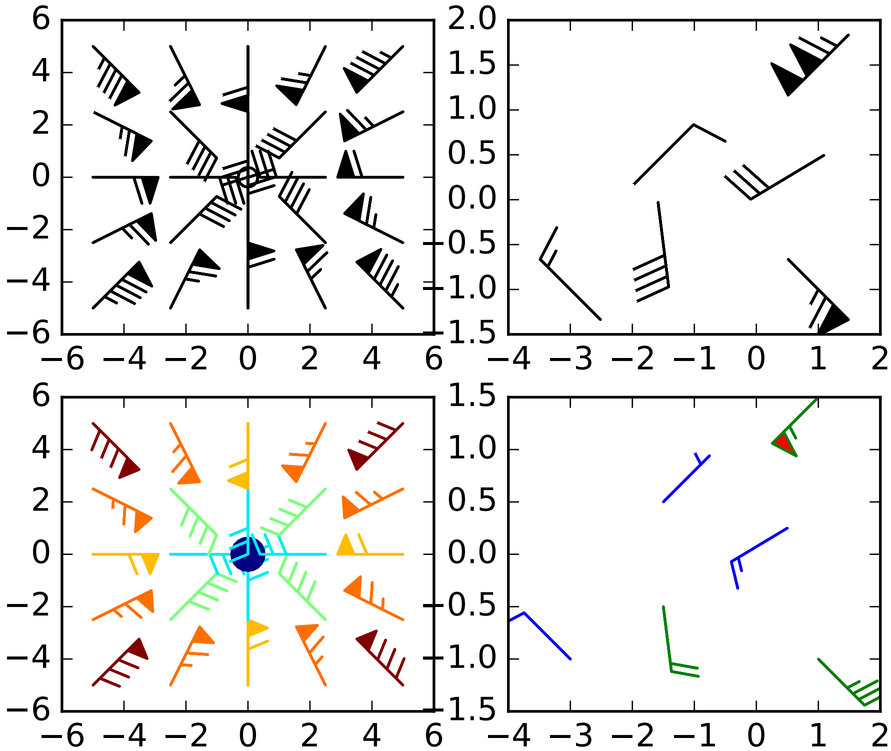
```
:      /\    \
:     /  \    \
:    /    \    \    \
:   /      \    \    \
:  ------------------------------
```

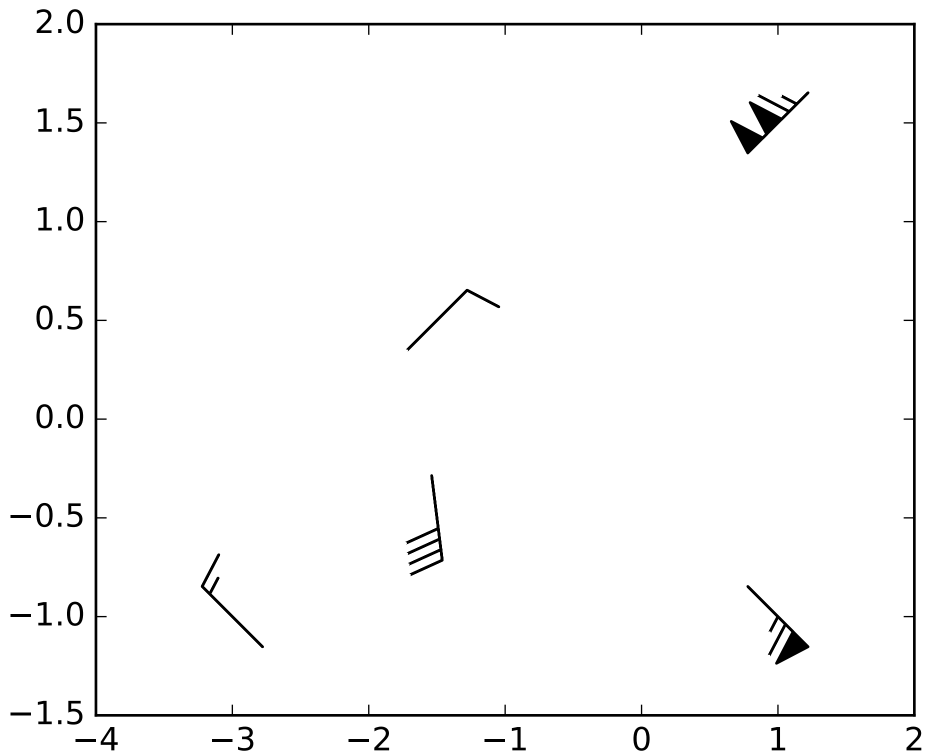
The largest increment is given by a triangle (or “flag”). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

linewidths and edgecolors can be used to customize the barb. Additional `PolyCollection` keyword arguments:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td><code>agg_filter</code></td><td>unknown</td></tr>
<tr><td><code>alpha</code></td><td>float or None</td></tr>
<tr><td><code>animated</code></td><td>[True | False]</td></tr>
<tr><td><code>antialiased</code> or antialiaseds</td><td>Boolean or sequence of booleans</td></tr>
<tr><td><code>array</code></td><td>unknown</td></tr>
<tr><td><code>axes</code></td><td>an <code>Axes</code> instance</td></tr>
<tr><td><code>clim</code></td><td>a length 2 sequence of floats</td></tr>
<tr><td><code>clip_box</code></td><td>a <code>matplotlib.transforms.Bbox</code> instance</td></tr>
<tr><td><code>clip_on</code></td><td>[True | False]</td></tr>
<tr><td><code>clip_path</code></td><td>[(<code>Path</code>, <code>Transform</code>) | <code>Patch</code> | None]</td></tr>
<tr><td><code>cmap</code></td><td>a colormap or registered colormap name</td></tr>
<tr><td><code>color</code></td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td><code>contains</code></td><td>a callable function</td></tr>
<tr><td><code>edgecolor</code> or edgecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>facecolor</code> or facecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>figure</code></td><td>a <code>matplotlib.figure.Figure</code> instance</td></tr>
<tr><td><code>gid</code></td><td>an id string</td></tr>
<tr><td><code>hatch</code></td><td>[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]</td></tr>
<tr><td><code>label</code></td><td>string or anything printable with ‘%s’ conversion.</td></tr>
<tr><td><code>linestyle</code> or linestyles or dashes</td><td>[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> |</td></tr>
<tr><td><code>linewidth</code> or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td><code>norm</code></td><td>unknown</td></tr>
<tr><td><code>offset_position</code></td><td>unknown</td></tr>
<tr><td><code>offsets</code></td><td>float or sequence of floats</td></tr>
<tr><td><code>path_effects</code></td><td>unknown</td></tr>
<tr><td><code>picker</code></td><td>[None|float|boolean|callable]</td></tr>
<tr><td><code>pickradius</code></td><td>unknown</td></tr>
<tr><td><code>rasterized</code></td><td>[True | False | None]</td></tr>
<tr><td><code>sketch_params</code></td><td>unknown</td></tr>
<tr><td><code>snap</code></td><td>unknown</td></tr>
<tr><td><code>transform</code></td><td><code>Transform</code> instance</td></tr>
<tr><td><code>url</code></td><td>a url string</td></tr>
<tr><td><code>urls</code></td><td>unknown</td></tr>
<tr><td><code>visible</code></td><td>[True | False]</td></tr>
<tr><td><code>zorder</code></td><td>any number</td></tr>
</table>

Example:


6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4 −3 −2 −1 0 1 2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

barh(*bottom*, *width*, *height=0.8*, *left=None*, ***kwargs*)

Make a horizontal bar plot.

Make a horizontal bar plot with rectangles bounded by:

left, left + width, bottom, bottom + height (left, right, bottom and top edges)

bottom, width, height, and left can be either scalars or sequences

Parameters bottom : scalar or array-like

the y coordinate(s) of the bars

width : scalar or array-like

the width(s) of the bars

height : sequence of scalars, optional, default: 0.8

the heights of the bars

left : sequence of scalars

the x coordinates of the left sides of the bars

Returns *matplotlib.patches.Rectangle* instances.

Other Parameters color : scalar or array-like, optional the colors of the bars

edgecolor : scalar or array-like, optional
the colors of the bar edges

linewidth : scalar or array-like, optional, default: None
width of bar edge(s). If None, use default linewidth; If 0, don't draw edges.

tick_label : string or array-like, optional, default: None
the tick labels of the bars

xerr : scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

yerr : scalar or array-like, optional, default: None
if not None, will be used to generate errorbar(s) on the bar chart

ecolor : scalar or array-like, optional, default: None
specifies the color of errorbar(s)

capsize : scalar, optional
determines the length in points of the error bar caps default: None, which will take the value from the `errorbar.capsize` *rcParam*.

error_kw :
dictionary of kwargs to be passed to errorbar method. `ecolor` and `capsize` may be specified here rather than as independent kwargs.

align : ['edge' | 'center'], optional, default: 'edge'
If `edge`, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If `center`, interpret the `left` argument as the coordinates of the centers of the bars.

log : boolean, optional, default: False
If true, sets the axis to be log scale

See also:

bar Plot a vertical bar plot.

Notes

The optional arguments `color`, `edgecolor`, `linewidth`, `xerr`, and `yerr` can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: `xerr` and `yerr` are passed directly to `errorbar()`, so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default

Continued on

Table 43.8 – continued from previous page

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color</td><td>matplotlib color spec</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or ec</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td>facecolor or fc</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fill</td><td>[True | False]</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>hatch</td><td>['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']</td></tr>
<tr><td>joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'</td></tr>
<tr><td>linewidth or lw</td><td>float or None for default</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>transform</td><td>Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

boxplot(*x*, *notch=None*, *sym=None*, *vert=None*, *whis=None*, *positions=None*, *widths=None*, *patch_artist=None*, *bootstrap=None*, *usermedians=None*, *conf_intervals=None*, *meanline=None*, *showmeans=None*, *showcaps=None*, *showbox=None*, *showfliers=None*, *boxprops=None*, *labels=None*, *flierprops=None*, *medianprops=None*, *meanprops=None*, *capprops=None*, *whiskerprops=None*, *manage_xticks=True*, *autorange=False*)

Make a box and whisker plot.

Call signature:

```
boxplot(self, x, notch=None, sym=None, vert=None, whis=None,
        positions=None, widths=None, patch_artist=False,
        bootstrap=None, usermedians=None, conf_intervals=None,
        meanline=False, showmeans=False, showcaps=True,
        showbox=True, showfliers=True, boxprops=None,
        labels=None, flierprops=None, medianprops=None,
        meanprops=None, capprops=None, whiskerprops=None,
        manage_xticks=True, autorange=False):
```

Make a box and whisker plot for each column of `x` or each vector in sequence `x`. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

Parameters **x** : Array or a sequence of vectors.

The input data.

notch : bool, optional (False)

If `True`, will produce a notched box plot. Otherwise, a rectangular boxplot is produced.

sym : str, optional

The default symbol for flier points. Enter an empty string (‘’) if you don’t want to show fliers. If `None`, then the fliers default to ‘b+’ If you want more control use the flierprops kwarg.

vert : bool, optional (True)

If `True` (default), makes the boxes vertical. If `False`, everything is drawn horizontally.

whis : float, sequence, or string (default = 1.5)

As a float, determines the reach of the whiskers past the first and third quartiles (e.g., Q3 + whis*IQR, IQR = interquartile range, Q3-Q1). Beyond the whiskers, data are considered outliers and are plotted as individual points. Set this to an unreasonably high value to force the whiskers to show the min and max values. Alternatively, set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, `whis` can be the string `'range'` to force the whiskers to the min and max of the data.

bootstrap : int, optional

Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If `bootstrap` is None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.

usermedians : array-like, optional

An array or sequence whose first dimension (or length) is compatible with `x`. This overrides the medians computed by matplotlib for each element of `usermedians` that is not `None`. When an element of `usermedians` is None, the median will be computed by matplotlib as normal.

conf_intervals : array-like, optional

Array or sequence whose first dimension (or length) is compatible with `x` and whose second dimension is 2. When the an element of `conf_intervals` is not None, the notch locations computed by matplotlib are overridden (provided `notch` is `True`). When an element of `conf_intervals` is `None`, the notches are computed by the method specified by the other kwargs (e.g., `bootstrap`).

positions : array-like, optional
Sets the positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to `range(1, N+1)` where N is the number of boxes to be drawn.

widths : scalar or array-like
Sets the width of each box either with a scalar or a sequence. The default is 0.5, or `0.15*(distance between extreme positions)`, if that is smaller.

patch_artist : bool, optional (False)
If `False` produces boxes with the Line2D artist. Otherwise, boxes and drawn with Patch artists.

labels : sequence, optional
Labels for each dataset. Length must be compatible with dimensions of `x`.

manage_xticks : bool, optional (True)
If the function should adjust the xlim and xtick locations.

autorange : bool, optional (False)
When `True` and the data are distributed such that the 25th and 75th percentiles are equal, `whis` is set to `'range'` such that the whisker ends are at the minimum and maximum of the data.

meanline : bool, optional (False)
If `True` (and `showmeans` is `True`), will try to render the mean as a line spanning the full width of the box according to `meanprops` (see below). Not recommended if `shownotches` is also True. Otherwise, means will be shown as points.

Returns result : dict
A dictionary mapping each component of the boxplot to a list of the `matplotlib.lines.Line2D` instances created. That dictionary has the following keys (assuming vertical boxplots):

- `boxes`: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: the vertical lines extending to the most extreme, non-outlier data points.
- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

Other Parameters The following boolean options toggle the drawing of individual

components of the boxplots:

- showcaps: the caps on the ends of whiskers (default is True)
- showbox: the central box (default is True)
- showfliers: the outliers beyond the caps (default is True)

- showmeans: the arithmetic means (default is False)

The remaining options can accept dictionaries that specify the

style of the individual artists:

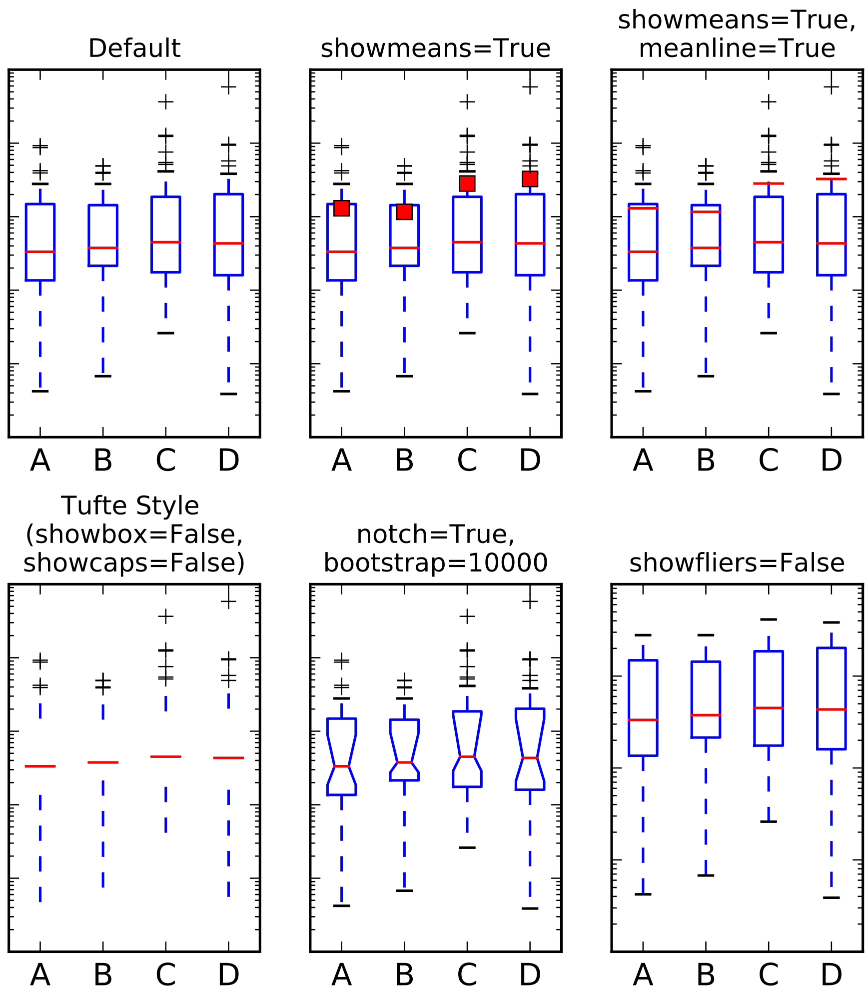
- capprops
- boxprops
- whiskerprops
- flierprops
- medianprops
- meanprops

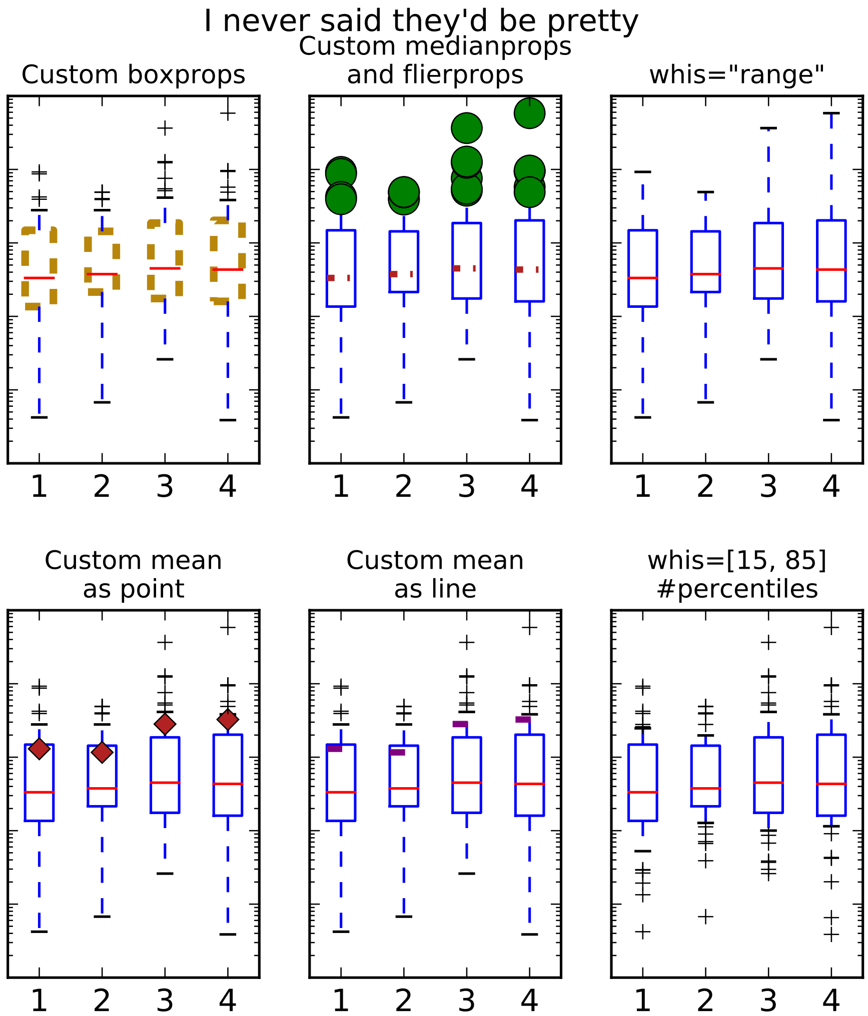
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

Examples


Default
showmeans=True
showmeans=True,
meanline=True
Tufte Style
(showbox=False,
showcaps=False)
notch=True,
bootstrap=10000
showfliers=False
A
B
C
D



broken_barh(*xranges*, *yrange*, ***kwargs*)

Plot horizontal bars.

Call signature:

```
broken_barh(self, xranges, yrange, **kwargs)
```

A collection of horizontal bars spanning *yrange* with a sequence of *xranges*.

Required arguments:

Argument	Description
xranges	sequence of (*xmin*, *xwidth*)
yrange	sequence of (*ymin*, *ywidth*)

kwargs are `matplotlib.collections.BrokenBarHCollection` properties:

Property	Description
`agg_filter`	unknown

Table 43.9 – continued from previous page

Property	Description
alpha	float or None
animated	[True | False]
antialiased or antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an *Axes* instance
clim	a length 2 sequence of floats
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
contains	a callable function
edgecolor or edgecolors	matplotlib color spec or sequence of specs
facecolor or facecolors	matplotlib color spec or sequence of specs
figure	a *matplotlib.figure.Figure* instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` |
linewidth or lw or linewidths	float or sequence of floats
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
path_effects	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

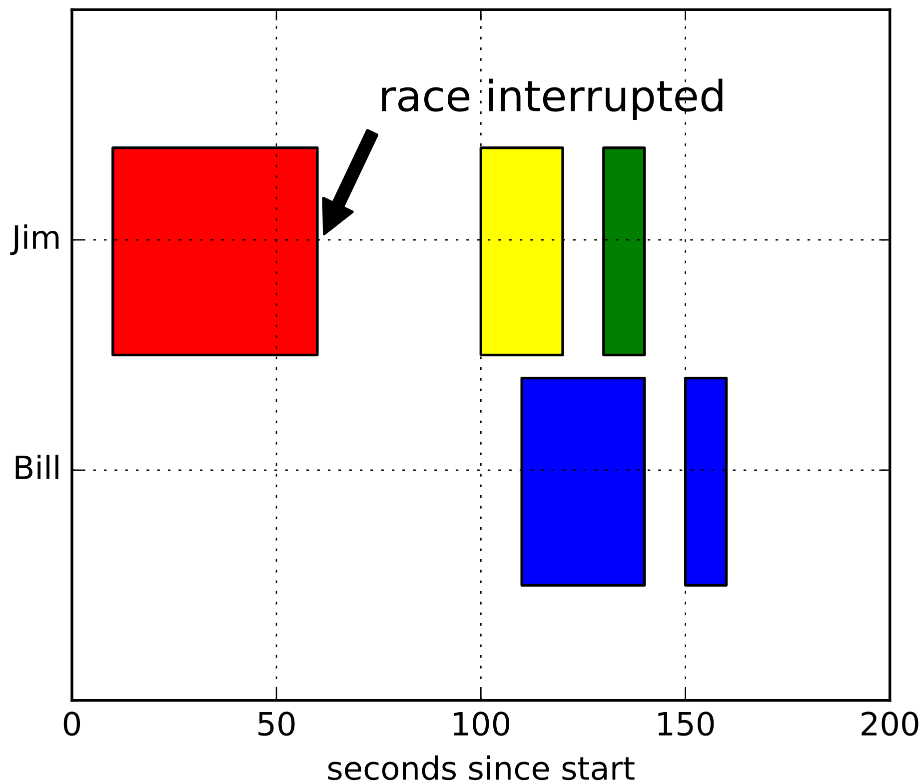
these can either be a single argument, i.e.,:

```
facecolors = 'black'
```

or a sequence of arguments for the various bars, i.e.,:

```
facecolors = ('black', 'red', 'green')
```

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

bxp(*bxpstats*, *positions=None*, *widths=None*, *vert=True*, *patch_artist=False*, *shownotches=False*, *showmeans=False*, *showcaps=True*, *showbox=True*, *showfliers=True*, *boxprops=None*, *whiskerprops=None*, *flierprops=None*, *medianprops=None*, *capprops=None*, *meanprops=None*, *meanline=False*, *manage_xticks=True*)

Drawing function for box and whisker plots.

Call signature:

```
bxp(self, bxpstats, positions=None, widths=None, vert=True,
    patch_artist=False, shownotches=False, showmeans=False,
    showcaps=True, showbox=True, showfliers=True,
    boxprops=None, whiskerprops=None, flierprops=None,
    medianprops=None, capprops=None, meanprops=None,
    meanline=False, manage_xticks=True):
```

Make a box and whisker plot for each column of *x* or each vector in sequence *x*. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

Parameters bxpstats : list of dicts

A list of dictionaries containing stats for each boxplot. Required keys are:

- `med`: The median (scalar float).
- `q1`: The first quartile (25th percentile) (scalar float).
- `q3`: The third quartile (75th percentile) (scalar float).
- `whislo`: Lower bound of the lower whisker (scalar float).
- `whishi`: Upper bound of the upper whisker (scalar float).

Optional keys are:

- `mean`: The mean (scalar float). Needed if `showmeans=True`.
- `fliers`: Data beyond the whiskers (sequence of floats). Needed if `showfliers=True`.
- `cilo` & `cihi`: Lower and upper confidence intervals about the median. Needed if `shownotches=True`.
- `label`: Name of the dataset (string). If available, this will be used a tick label for the boxplot

positions : array-like, default = [1, 2, ..., n]

Sets the positions of the boxes. The ticks and limits are automatically set to match the positions.

widths : array-like, default = 0.5

Either a scalar or a vector and sets the width of each box. The default is 0.5, or `0.15*(distance between extreme positions)` if that is smaller.

vert : bool, default = False

If `True` (default), makes the boxes vertical. If `False`, makes horizontal boxes.

patch_artist : bool, default = False

If `False` produces boxes with the *`Line2D`* artist. If `True` produces boxes with the *`Patch`* artist.

shownotches : bool, default = False

If `False` (default), produces a rectangular box plot. If `True`, will produce a notched box plot

showmeans : bool, default = False

If `True`, will toggle on the rendering of the means

showcaps : bool, default = True

If `True`, will toggle on the rendering of the caps

showbox : bool, default = True

If `True`, will toggle on the rendering of the box

showfliers : bool, default = True

If `True`, will toggle on the rendering of the fliers

boxprops : dict or None (default)

If provided, will set the plotting style of the boxes

whiskerprops : dict or None (default)

If provided, will set the plotting style of the whiskers

capprops : dict or None (default)

If provided, will set the plotting style of the caps

flierprops : dict or None (default)

If provided will set the plotting style of the fliers

medianprops : dict or None (default)

If provided, will set the plotting style of the medians

meanprops : dict or None (default)

If provided, will set the plotting style of the means

meanline : bool, default = False

If `True` (and *showmeans* is `True`), will try to render the mean as a line spanning the full width of the box according to *meanprops*. Not recommended if *shownotches* is also True. Otherwise, means will be shown as points.

manage_xticks : bool, default = True

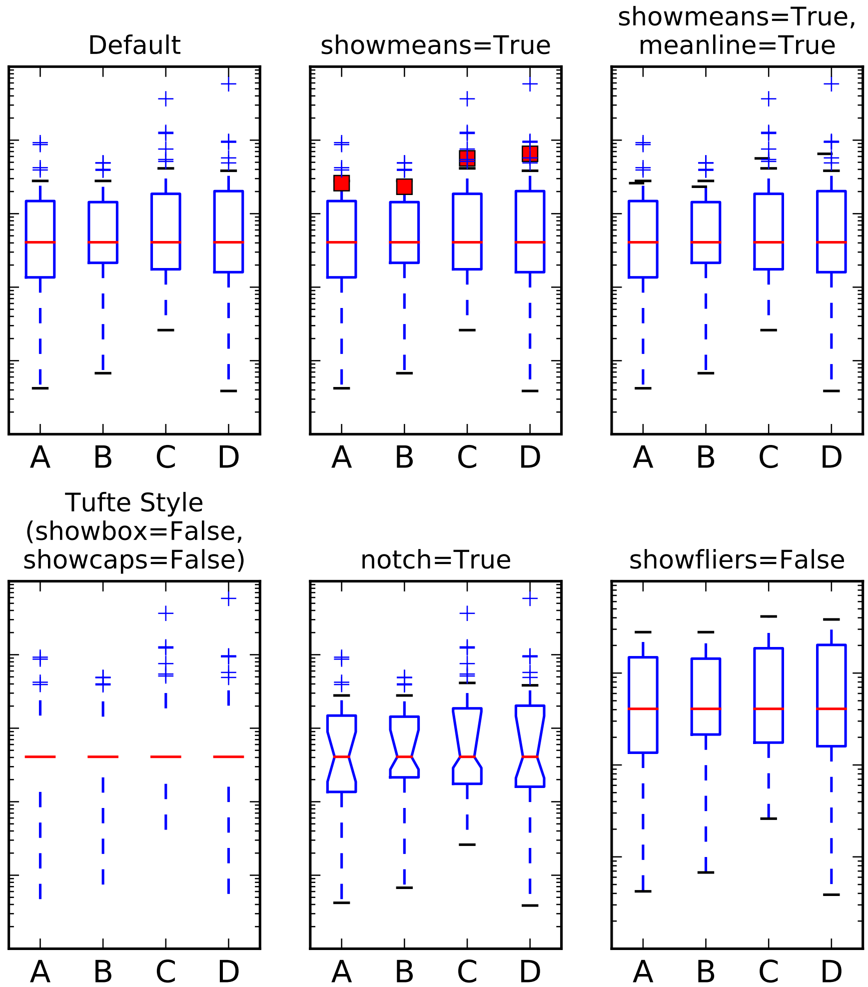
If the function should adjust the xlim and xtick locations.

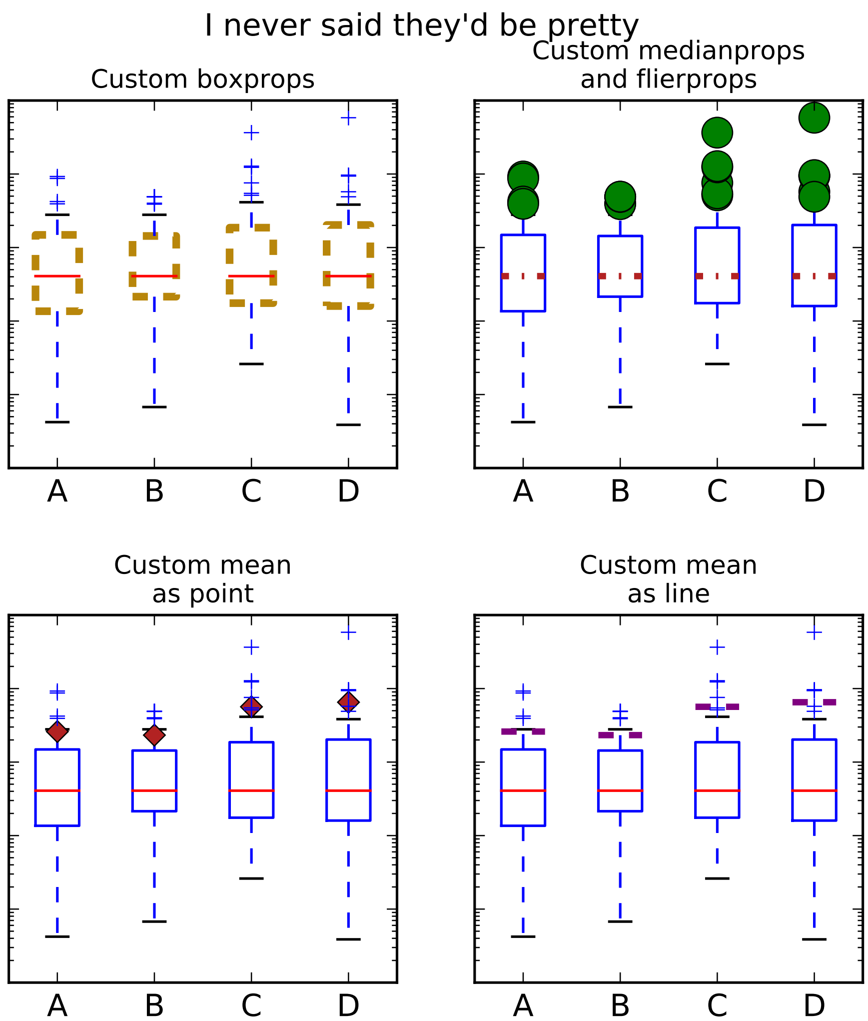
Returns result : dict

A dictionary mapping each component of the boxplot to a list of the *matplotlib.lines.Line2D* instances created. That dictionary has the following keys (assuming vertical boxplots):

- `boxes`: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: the vertical lines extending to the most extreme, non-outlier data points.
- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

Examples


Default
showmeans=True
showmeans=True,
meanline=True
Tufte Style
(showbox=False,
showcaps=False)
notch=True
showfliers=False
A
B
C
D



can_pan()

Return *True* if this axes supports any pan/zoom button functionality.

can_zoom()

Return *True* if this axes supports the zoom box button functionality.

cla()

Clear the current axes.

clabel(*CS*, **args*, ***kwargs*)

Label a contour plot.

Call signature:

```
clabel(cs, **kwargs)
```

Adds labels to line contours in *cs*, where *cs* is a `ContourSet` object returned by contour.

```
clabel(cs, v, **kwargs)
```

only labels contours listed in *v*.

Optional keyword arguments:

fontsize: size in points or relative size e.g., 'smaller', 'x-large'

colors:

- if *None*, the color of each label matches the color of the corresponding contour
- if one string color, e.g., *colors* = 'r' or *colors* = 'red', all labels will be plotted in this color
- if a tuple of matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified

inline: controls whether the underlying contour is removed or not. Default is *True*.

inline_spacing: space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

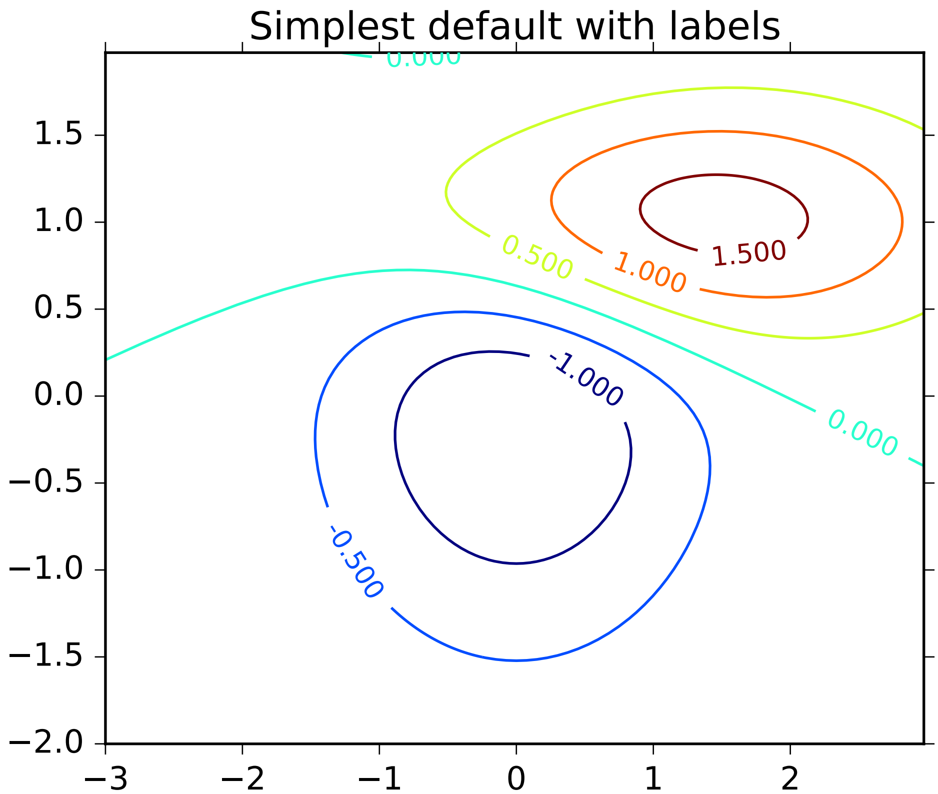
fmt: a format string for the label. Default is '%1.3f' Alternatively, this can be a dictionary matching contour levels with arbitrary strings to use for each contour level (i.e., fmt[level]=string), or it can be any callable, such as a `Formatter` instance, that returns a string when called with a numeric contour level.

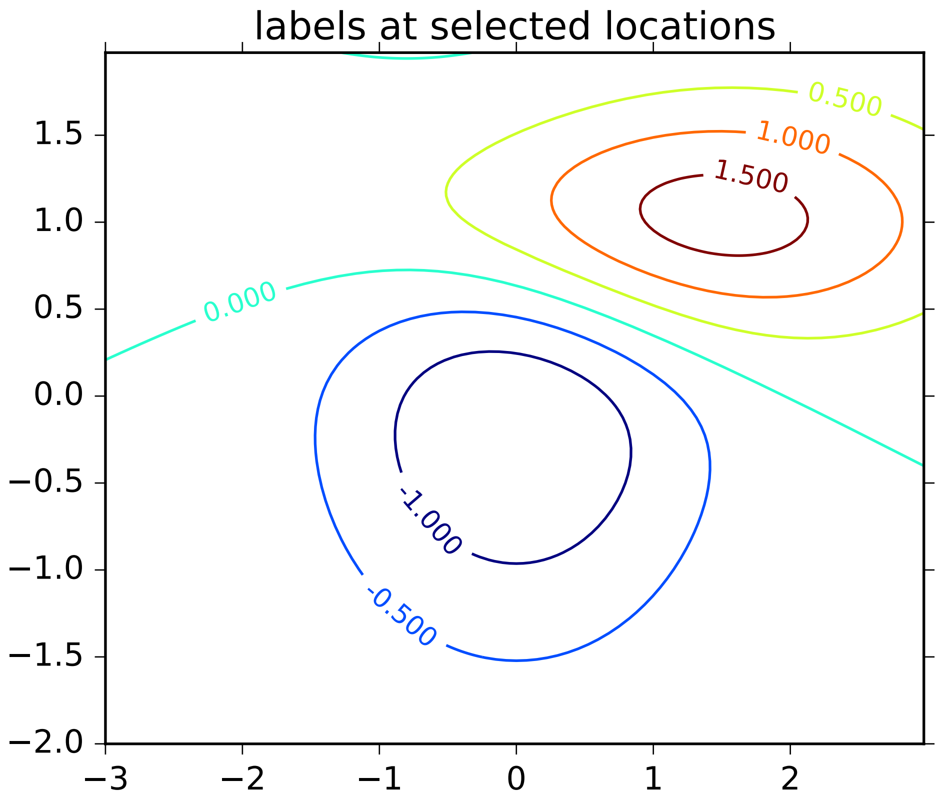
manual: if *True*, contour labels will be placed manually using mouse clicks. Click the first button near a contour to add a label, click the second button (or potentially both mouse buttons at once) to finish adding labels. The third button can be used to remove the last label added, but only if labels are not inline. Alternatively, the keyboard can be used to select label locations (enter to end label placement, delete or backspace act like the third mouse button, and any other key will select a label location).

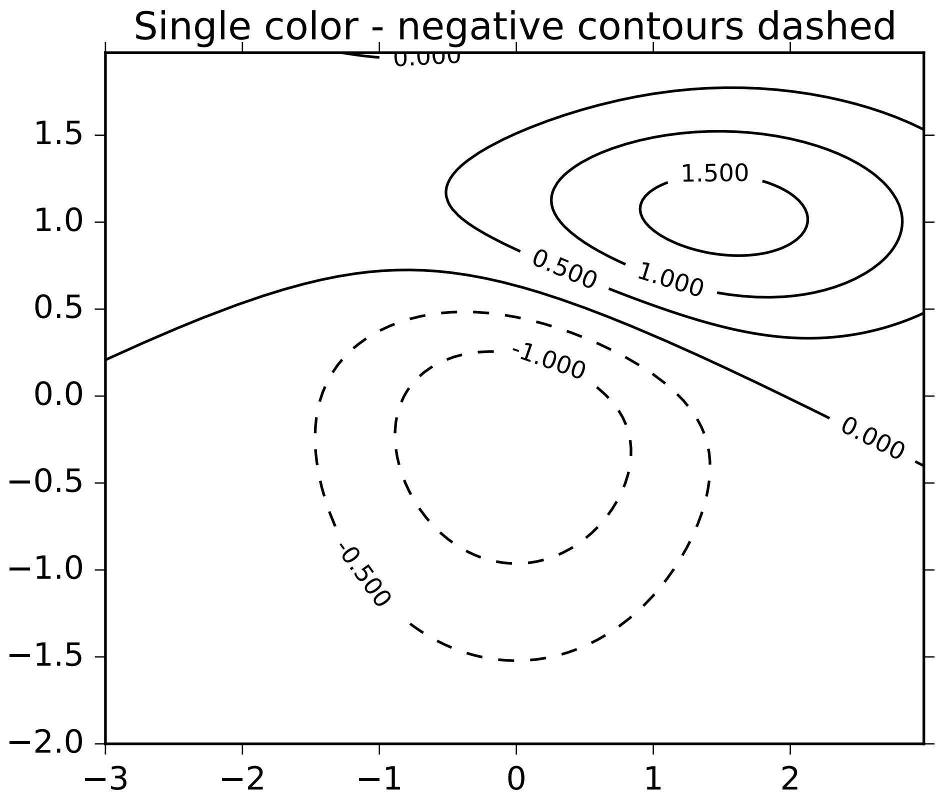
manual can be an iterable object of x,y tuples. Contour labels will be created as if mouse is clicked at each x,y positions.

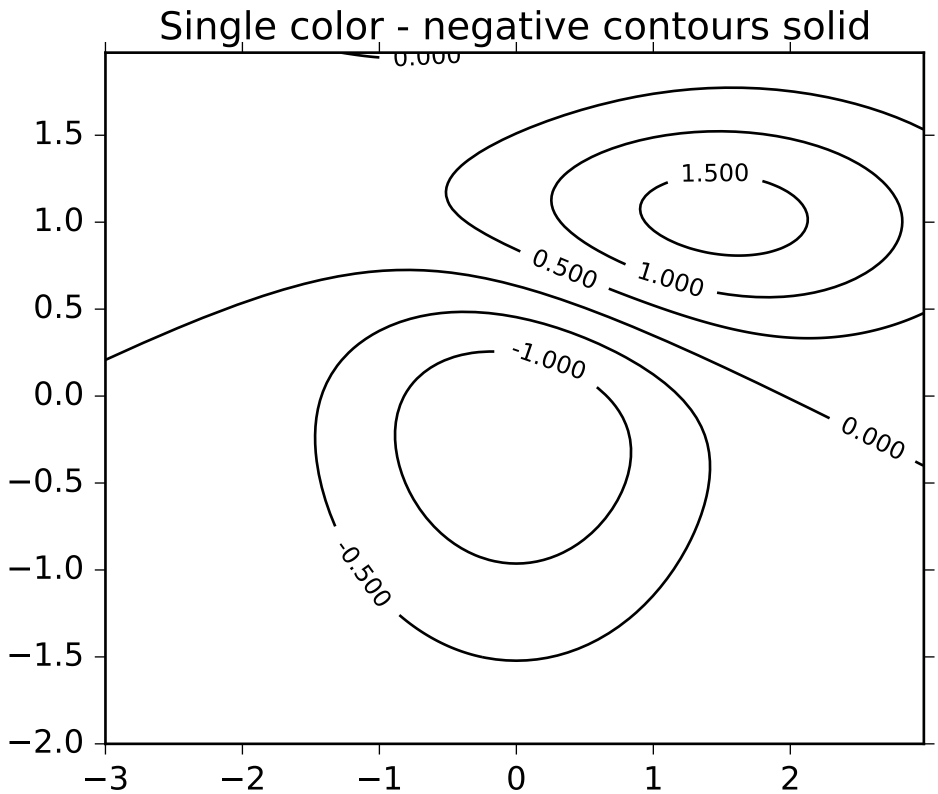
rightside_up: if *True* (default), label rotations will always be plus or minus 90 degrees from level.

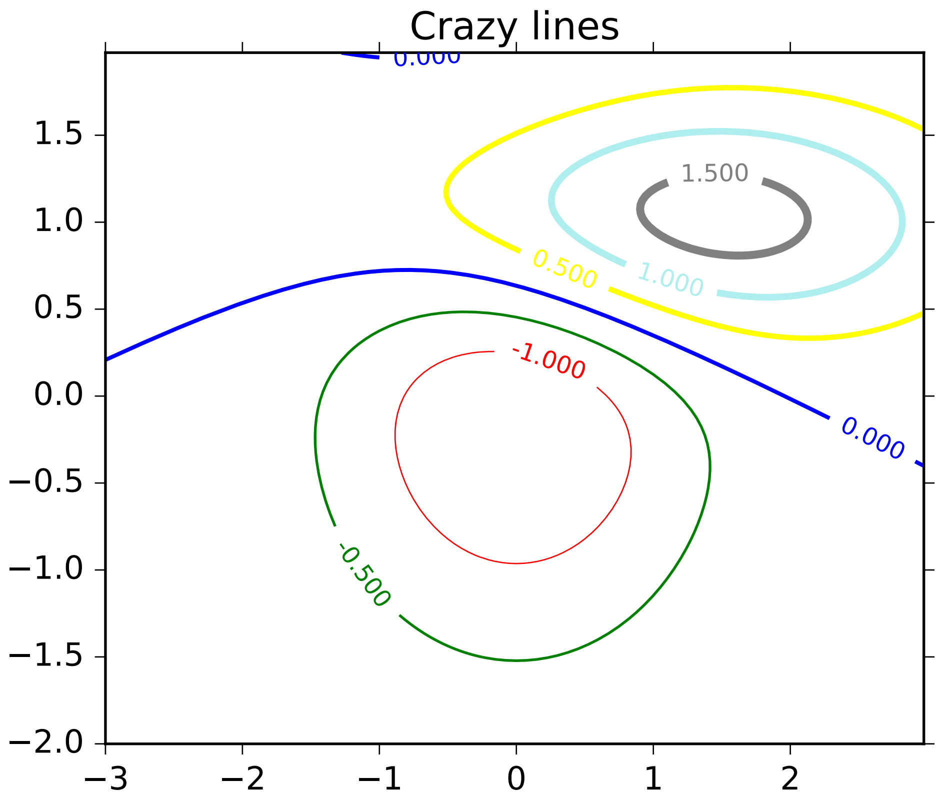
use_clabeltext: if *True* (default is False), ClabelText class (instead of matplotlib.Text) is used to create labels. ClabelText recalculates rotation angles of texts during the drawing time, therefore this can be used if aspect of the axes changes.

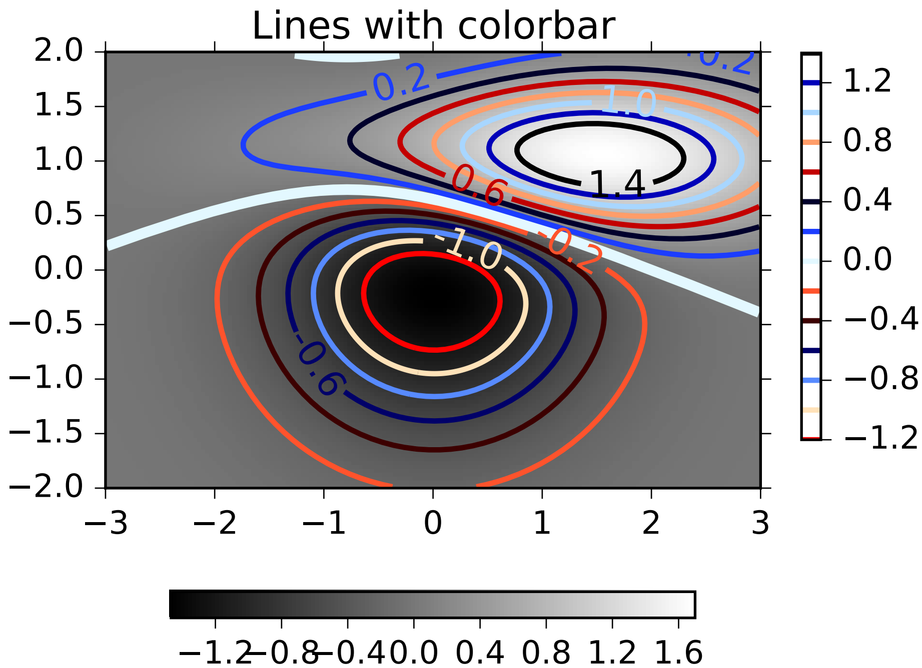

Simplest default with labels
0.000
0.500
1.000
1.500
-1.000
-0.500
0.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


labels at selected locations
0.500
1.000
1.500
0.000
-1.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Single color - negative contours dashed
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Single color - negative contours solid
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Crazy lines
0.000
1.500
1.000
0.500
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2



clear()
clear the axes

cohere(*x*, *y*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none>*, *window=<function window_hanning>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, ***kwargs*)

Plot the coherence between *x* and *y*.

Call signature:

```
cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend = mlab.detrend_none,
       window = mlab.window_hanning, noverlap=0, pad_to=None,
       sides='default', scale_by_freq=None, **kwargs)
```

Plot the coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \tag{43.1}$$

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is window_hanning(). If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or**
callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The pylab module defines detrend_none(), detrend_mean(), and detrend_linear(), but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call detrend_mean(). 'linear' calls detrend_linear(). 'none' calls detrend_none().

***scale_by_freq*: boolean**
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***noverlap*: integer** The number of points of overlap between blocks. The default value is 0 (no overlap).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

The return value is a tuple (Cxy, f), where f are the frequencies of the coherence vector.

kwargs are applied to the lines.

References:

- Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

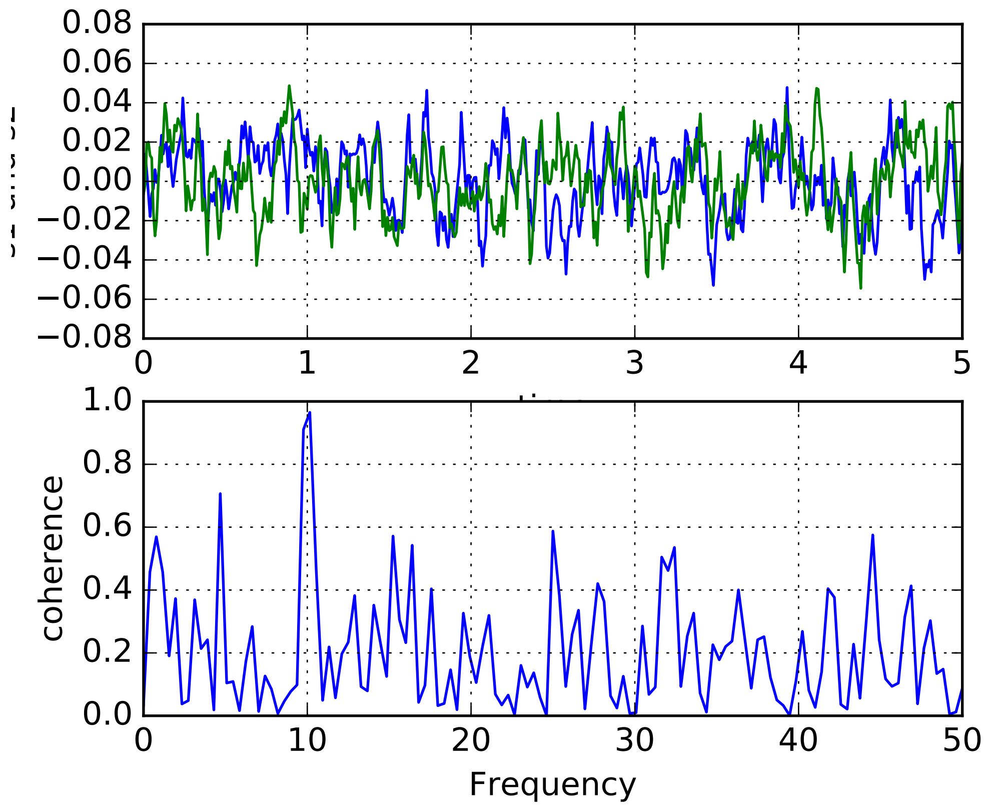
kwargs control the Line2D properties of the coherence plot:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
	C

Table 43.10 – continued from previous page

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>antialiased or aa</td><td>[True | False]</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color or c</td><td>any matplotlib color</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>dash_capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>dash_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>dashes</td><td>sequence of on/off ink in points</td></tr>
<tr><td>drawstyle</td><td>['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '</td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>marker</td><td>A valid marker style</td></tr>
<tr><td>markeredgecolor or mec</td><td>any matplotlib color</td></tr>
<tr><td>markeredgewidth or mew</td><td>float value in points</td></tr>
<tr><td>markerfacecolor or mfc</td><td>any matplotlib color</td></tr>
<tr><td>markerfacecoloralt or mfcalt</td><td>any matplotlib color</td></tr>
<tr><td>markersize or ms</td><td>float</td></tr>
<tr><td>markevery</td><td>[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>float distance in points or callable pick function fn(artist, event)</td></tr>
<tr><td>pickradius</td><td>float distance in points</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>solid_capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>solid_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>transform</td><td>a matplotlib.transforms.Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>xdata</td><td>1D array</td></tr>
<tr><td>ydata</td><td>1D array</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

contains(*mouseevent*)

Test whether the mouse event occured in the axes.

Returns *True* / *False*, {}

contains_point(*point*)

Returns *True* if the point (tuple of x,y) is inside the axes (the area defined by the its patch). A pixel coordinate is required.

contour(**args*, ***kwargs*)

Plot contours.

`contour()` and `contourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

`contourf()` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to `contour()`.

Call signatures:

```
contour(Z)
```

make a contour plot of an array *Z*. The level values are chosen automatically.

```
contour(X,Y,Z)
```

X, *Y* specify the (x, y) coordinates of the surface

```
contour(Z,N)
contour(X,Y,Z,N)
```

contour up to *N* automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence *V*, which must be in increasing order.

```
contourf(..., V)
```

fill the `len(V)-1` regions between the values in *V*, which must be in increasing order.

```
contour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

X and *Y* must both be 2-D with the same shape as *Z*, or they must both be 1-D such that `len(X)` is the number of columns in *Z* and `len(Y)` is the number of rows in *Z*.

`C = contour(...)` returns a `QuadContourSet` object.

Optional keyword arguments:

***corner_mask*: [*True* | *False* | 'legacy']** Enable/disable corner masking, which only has an effect if *Z* is a masked array. If *False*, any quad touching a masked point is masked out. If *True*, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual. If 'legacy', the old contouring algorithm is used, which is equivalent to *False* and is deprecated, only remaining whilst the new algorithm is tested fully.

If not specified, the default is taken from rcParams['contour.corner_mask'], which is True unless it has been modified.

***colors*: [*None* | string | (mpl_colors)]** If *None*, the colormap specified by cmap will be used.

If a string, like 'r' or 'red', all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

***alpha*: float** The alpha blending value

***cmap*: [*None* | Colormap]** A cm `Colormap` instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

***vmin*, *vmax*: [*None* | scalar]** If not *None*, either or both of these values will be supplied to the `matplotlib.colors.Normalize` instance, overriding the default color scaling based on *levels*.

***levels*: [level0, level1, ..., leveln]** A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

***origin*: [*None* | 'upper' | 'lower' | 'image']** If *None*, the first value of *Z* will correspond to the lower left corner, location (0,0). If 'image', the rc value for `image.origin` will be used.

This keyword is not active if *X* and *Y* are specified in the call to contour.

extent: [*None* | (x0,x1,y0,y1)]

If *origin* is not *None*, then *extent* is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0], and (*x1*, *y1*) is the position of Z[-1,-1].

This keyword is not active if *X* and *Y* are specified in the call to contour.

***locator*: [*None* | ticker.Locator subclass]** If *locator* is *None*, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

***extend*: ['neither' | 'both' | 'min' | 'max']** Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

***xunits*, *yunits*: [*None* | registered units]** Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

***antialiased*: [*True* | *False*]** enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from rcParams['lines.antialiased'].

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

contour-only keyword arguments:

***linewidths*: [*None* | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

***linestyles*: [*None* | 'solid' | 'dashed' | 'dashdot' | 'dotted']** If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc` `contour.negative_linestyle` setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

contourf-only keyword arguments:

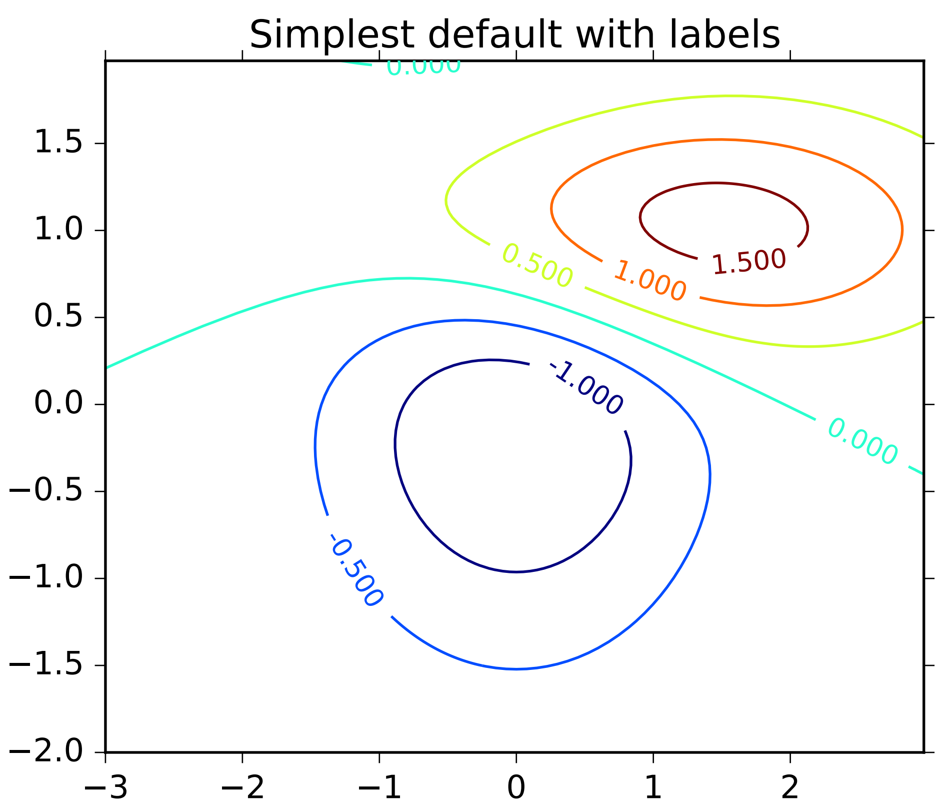
***hatches*:** A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

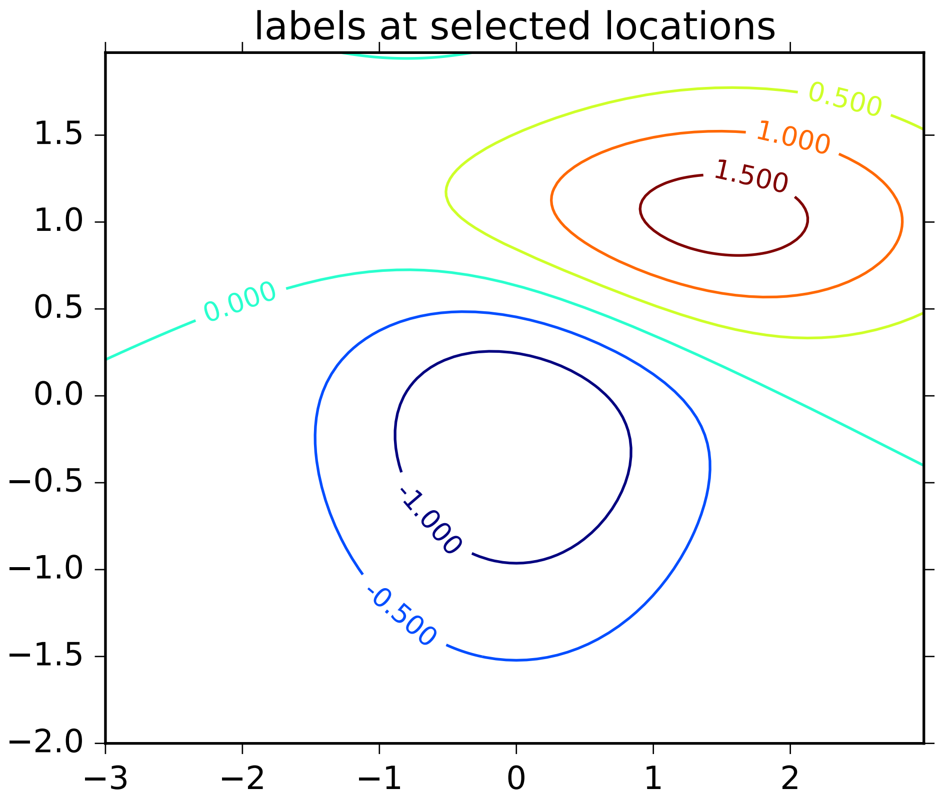
Note: contourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

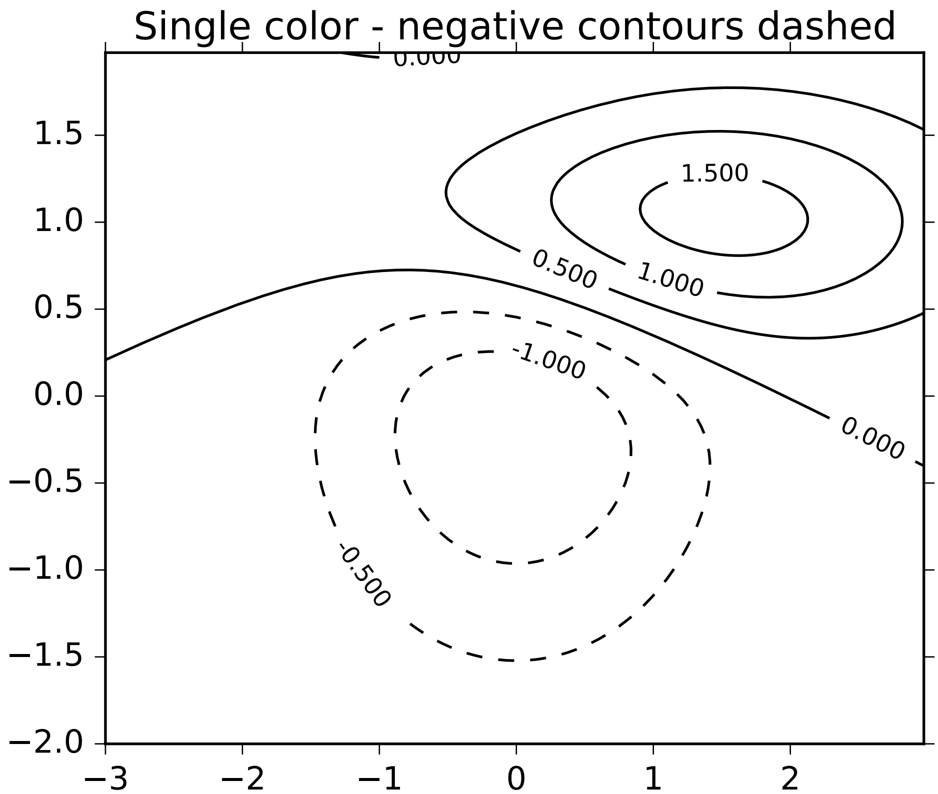
```
z1 < z <= z2
```

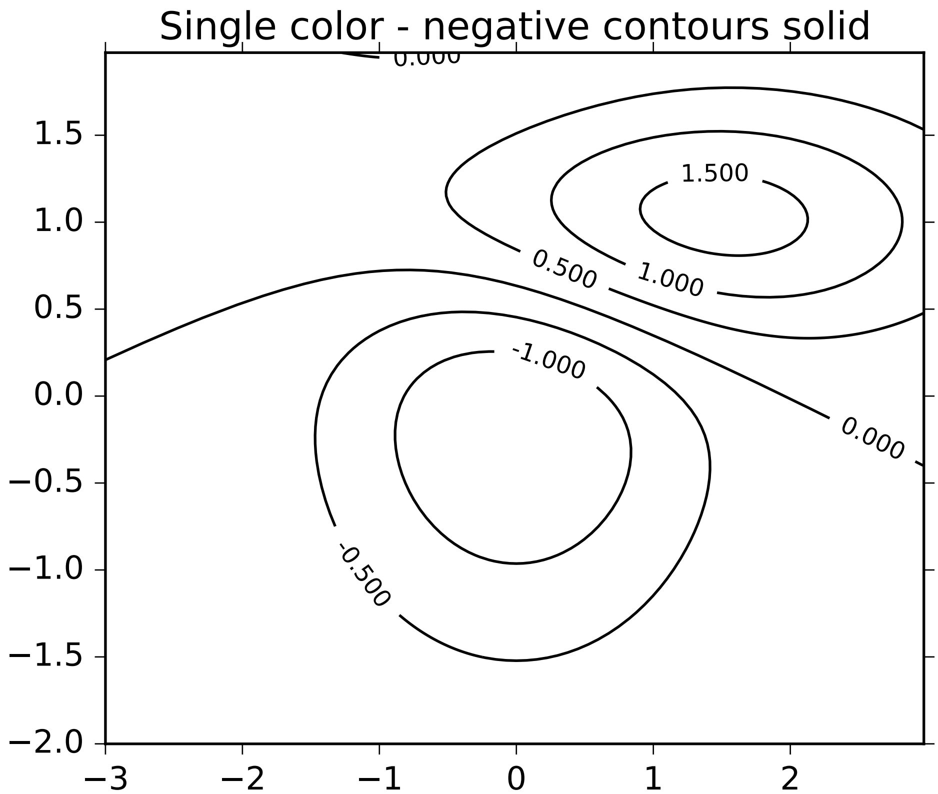
There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

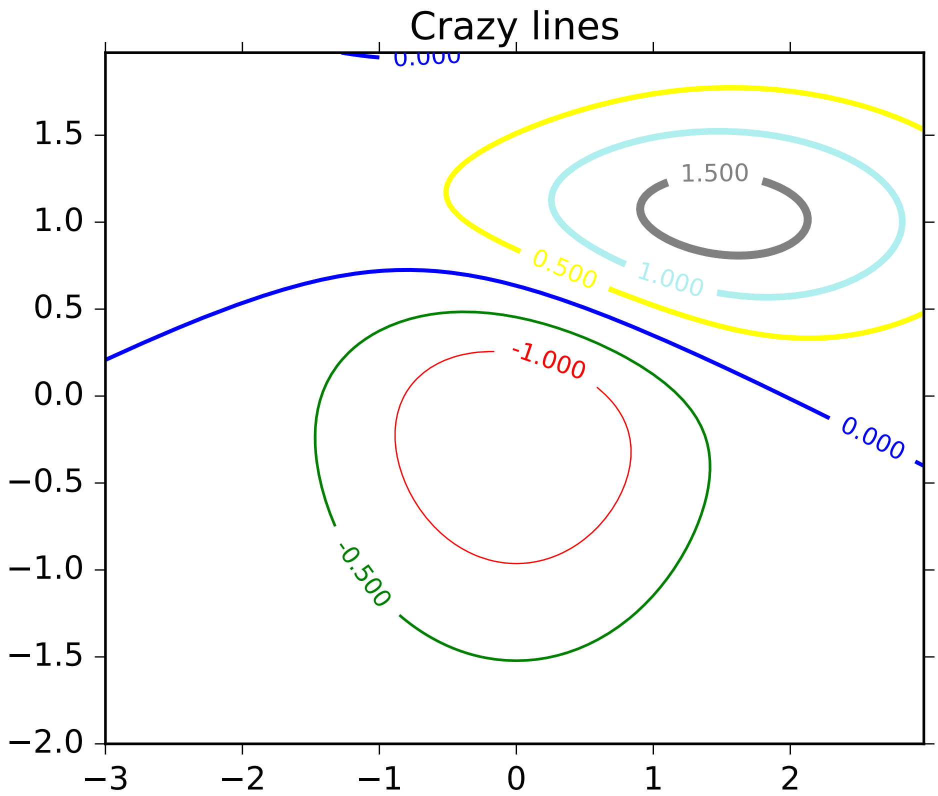
Examples:

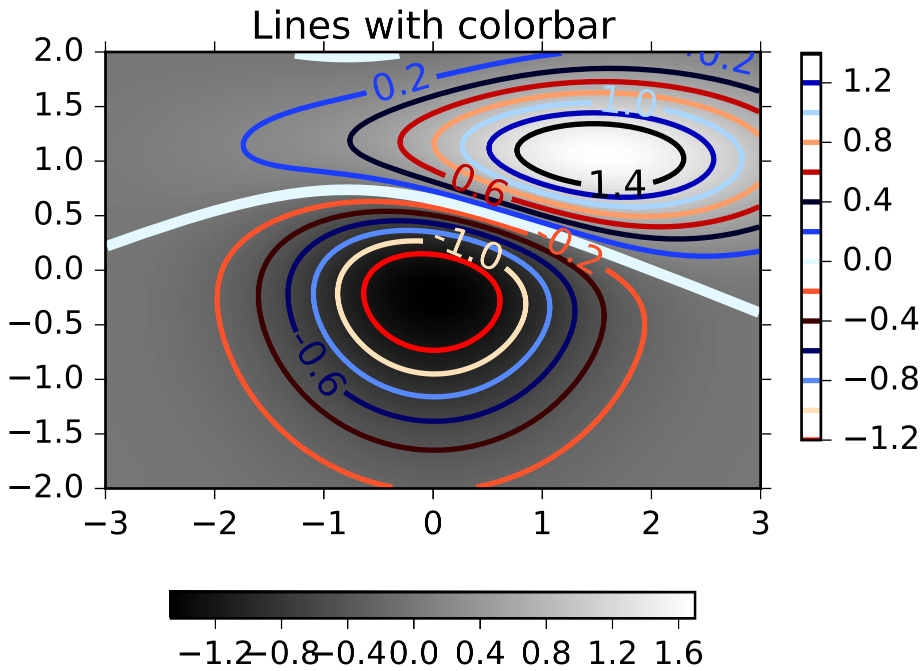


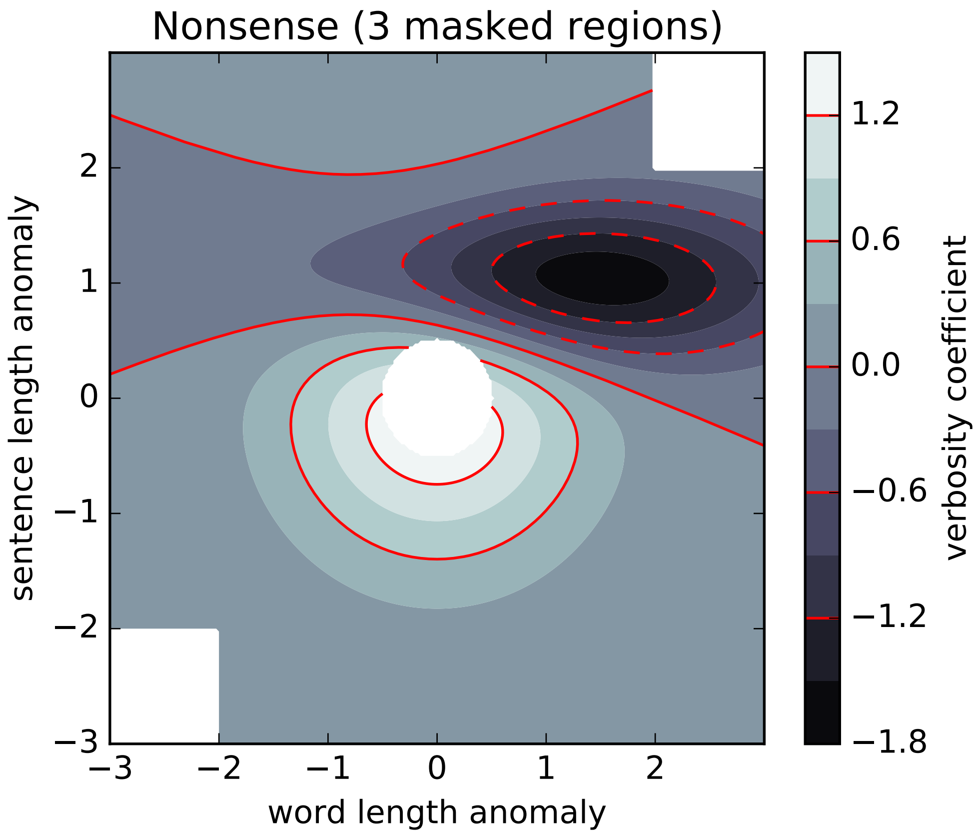

labels at selected locations
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
0.500
1.000
1.500
0.000
-1.000
-0.500


Single color - negative contours dashed
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

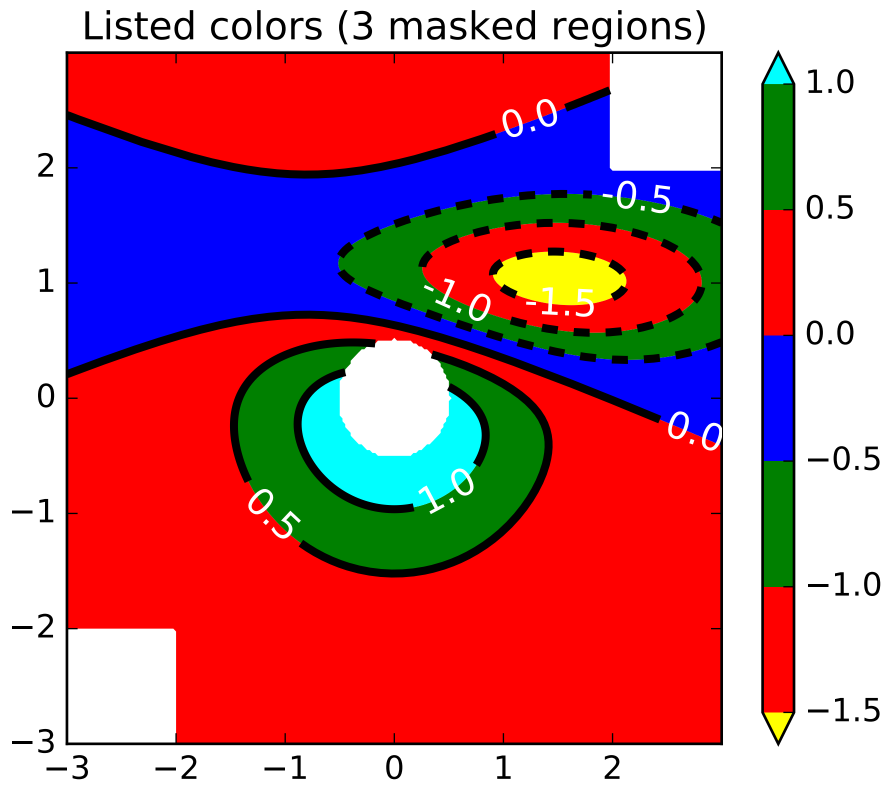

Single color - negative contours solid
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

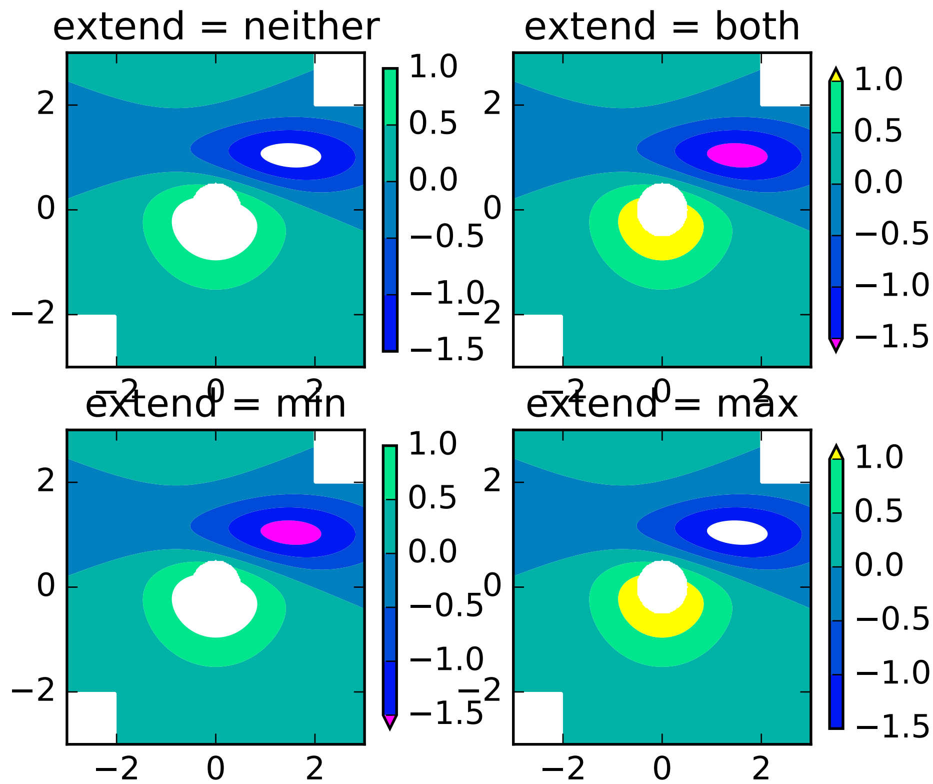

Crazy lines
0.000
1.500
1.000
0.500
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

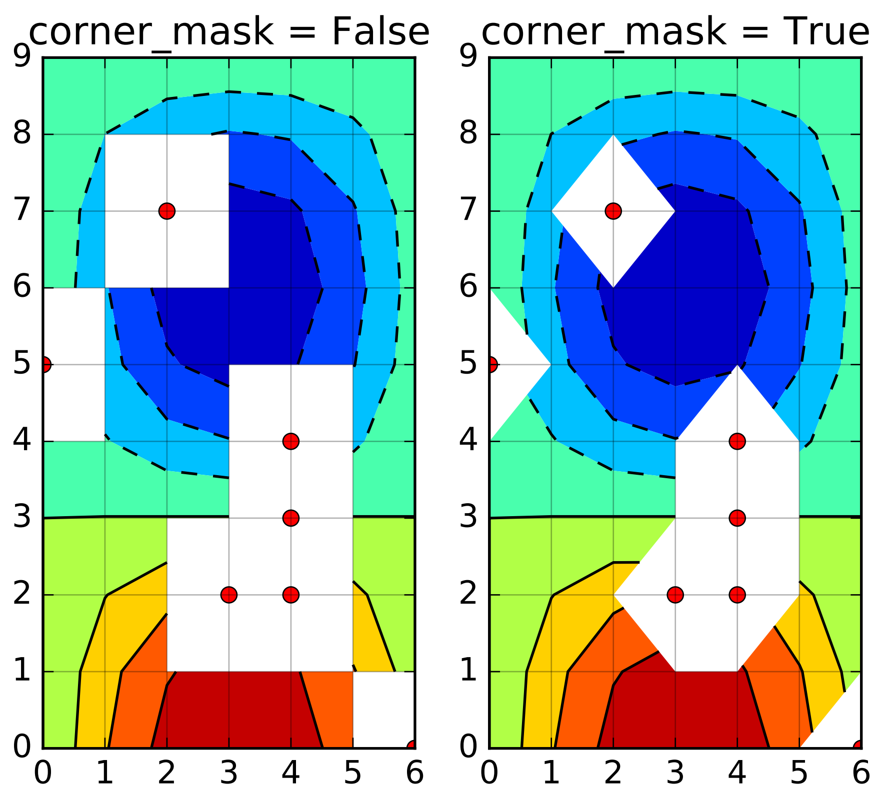

Lines with colorbar
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
3
0.2
1.0
1.4
0.6
-1.0
-0.2
-0.6
1.2
0.8
0.4
0.0
−0.4
−0.8
−1.2
−1.2
−0.8
−0.4
0.0
0.4
0.8
1.2
1.6



Nonsense (3 masked regions)
sentence length anomaly
word length anomaly
verbosity coefficient
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
1.2
0.6
0.0
−0.6
−1.2
−1.8


Listed colors (3 masked regions)
0.0
-0.5
-1.0
-1.5
0.0
1.0
0.5
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
1.0
0.5
0.0
−0.5
−1.0
−1.5


extend = neither
extend = both
extend = min
extend = max
2
0
−2
−2
0
2
1.0
0.5
0.0
−0.5
−1.0
−1.5



contourf(**args*, ***kwargs*)

Plot contours.

contour() and *contourf()* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

contourf() differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to *contour()*.

Call signatures:

```
contour(Z)
```

make a contour plot of an array *Z*. The level values are chosen automatically.

```
contour(X,Y,Z)
```

X, *Y* specify the (x, y) coordinates of the surface

```
contour(Z,N)
contour(X,Y,Z,N)
```

contour up to *N* automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence *V*, which must be in increasing order.

```
contourf(..., V)
```

fill the `len(V)-1` regions between the values in *V*, which must be in increasing order.

```
contour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

X and *Y* must both be 2-D with the same shape as *Z*, or they must both be 1-D such that `len(X)` is the number of columns in *Z* and `len(Y)` is the number of rows in *Z*.

`C = contour(...)` returns a `QuadContourSet` object.

Optional keyword arguments:

***corner_mask*: [*True* | *False* | 'legacy']** Enable/disable corner masking, which only has an effect if *Z* is a masked array. If *False*, any quad touching a masked point is masked out. If *True*, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual. If 'legacy', the old contouring algorithm is used, which is equivalent to *False* and is deprecated, only remaining whilst the new algorithm is tested fully.

If not specified, the default is taken from rcParams['contour.corner_mask'], which is True unless it has been modified.

***colors*: [*None* | string | (mpl_colors)]** If *None*, the colormap specified by cmap will be used.

If a string, like 'r' or 'red', all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

***alpha*: float** The alpha blending value

***cmap*: [*None* | Colormap]** A cm `Colormap` instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

***vmin*, *vmax*: [*None* | scalar]** If not *None*, either or both of these values will be supplied to the `matplotlib.colors.Normalize` instance, overriding the default color scaling based on *levels*.

***levels*: [level0, level1, ..., leveln]** A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

***origin*: [*None* | 'upper' | 'lower' | 'image']** If *None*, the first value of *Z* will correspond to the lower left corner, location (0,0). If 'image', the rc value for `image.origin` will be used.

This keyword is not active if *X* and *Y* are specified in the call to contour.

extent: [*None* | (x0,x1,y0,y1)]

If *origin* is not *None*, then *extent* is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0], and (*x1*, *y1*) is the position of Z[-1,-1].

This keyword is not active if *X* and *Y* are specified in the call to contour.

***locator*: [*None* | ticker.Locator subclass]** If *locator* is *None*, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

***extend*: ['neither' | 'both' | 'min' | 'max']** Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

***xunits*, *yunits*: [*None* | registered units]** Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

***antialiased*: [*True* | *False*]** enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from rcParams['lines.antialiased'].

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

contour-only keyword arguments:

***linewidths*: [*None* | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

***linestyles*: [*None* | 'solid' | 'dashed' | 'dashdot' | 'dotted']** If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc` `contour.negative_linestyle` setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

contourf-only keyword arguments:

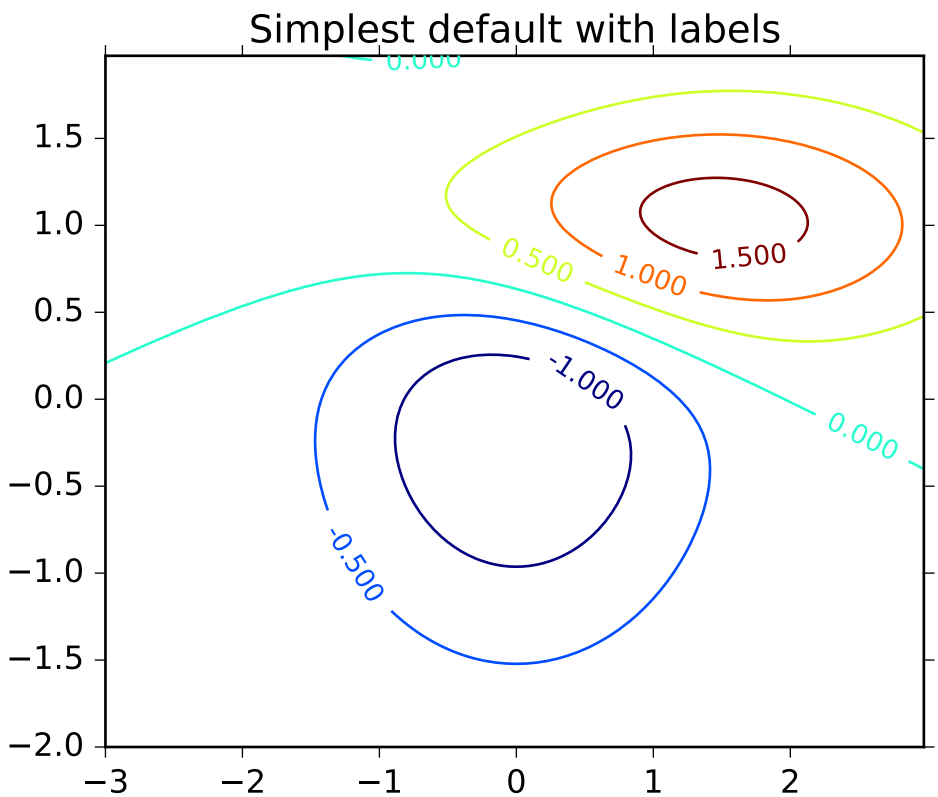
***hatches*:** A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

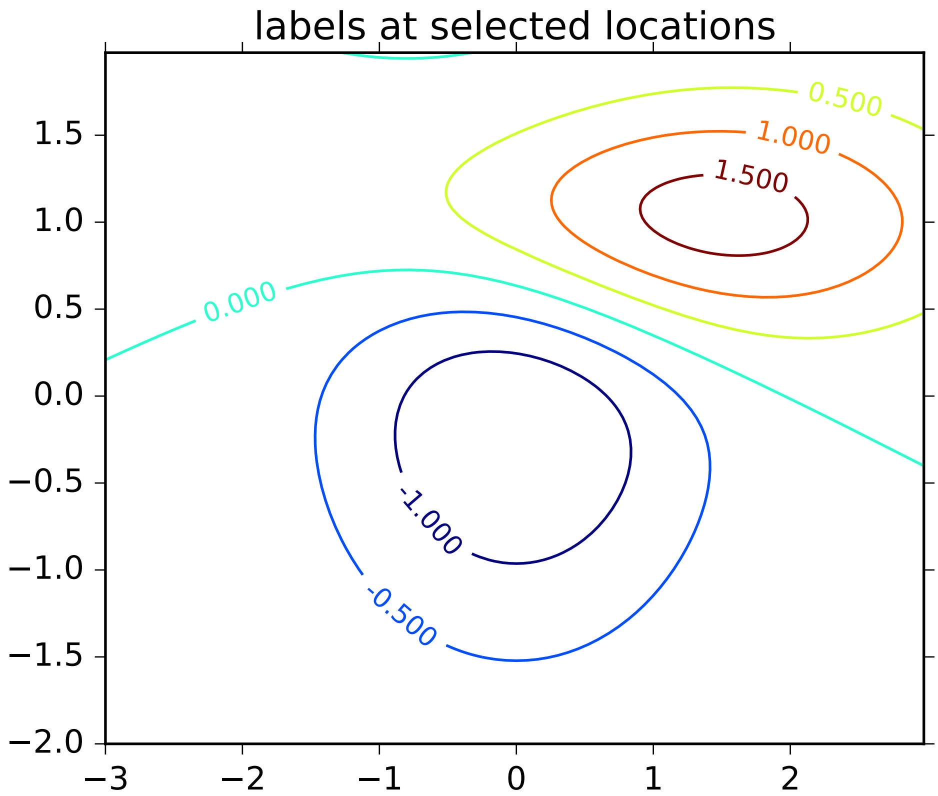
Note: contourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

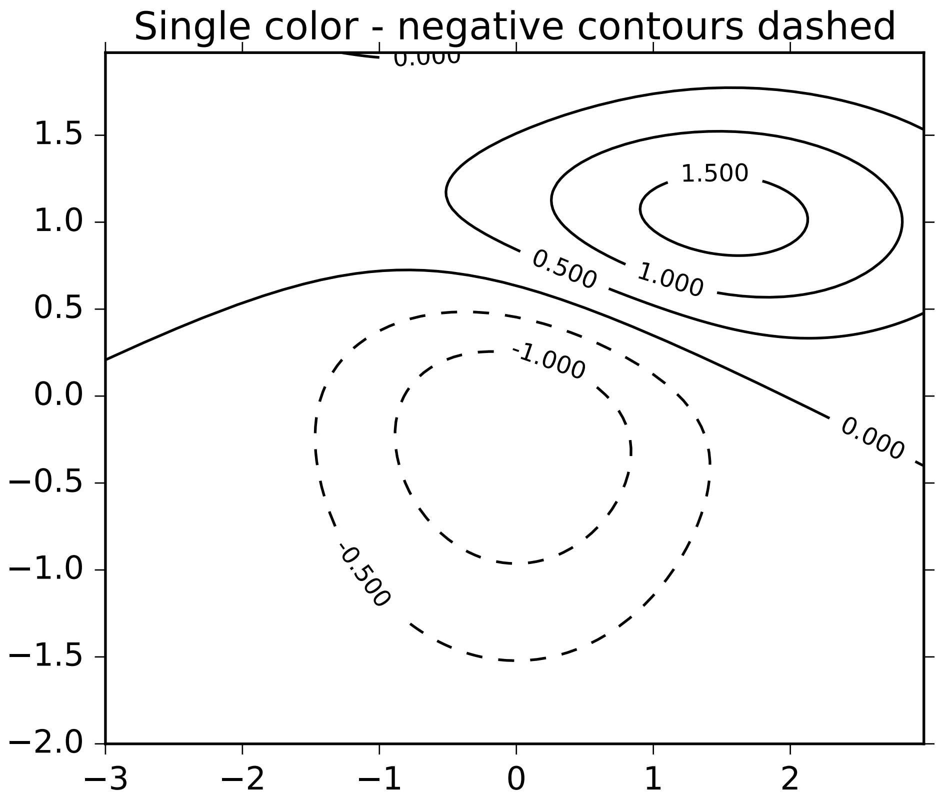
```
z1 < z <= z2
```

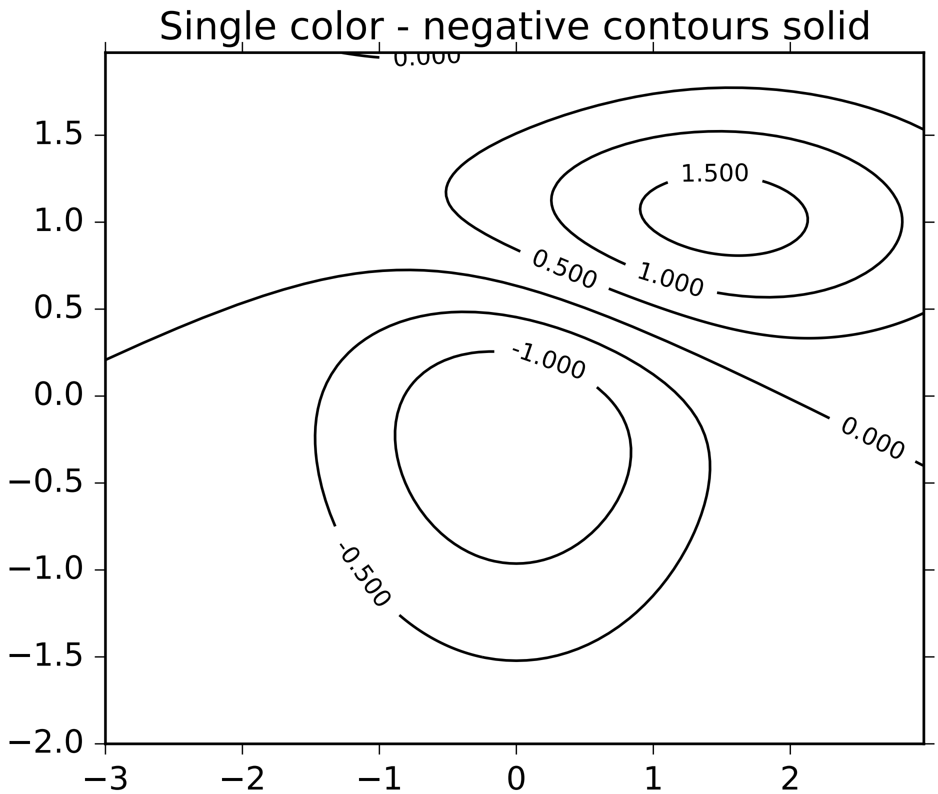
There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

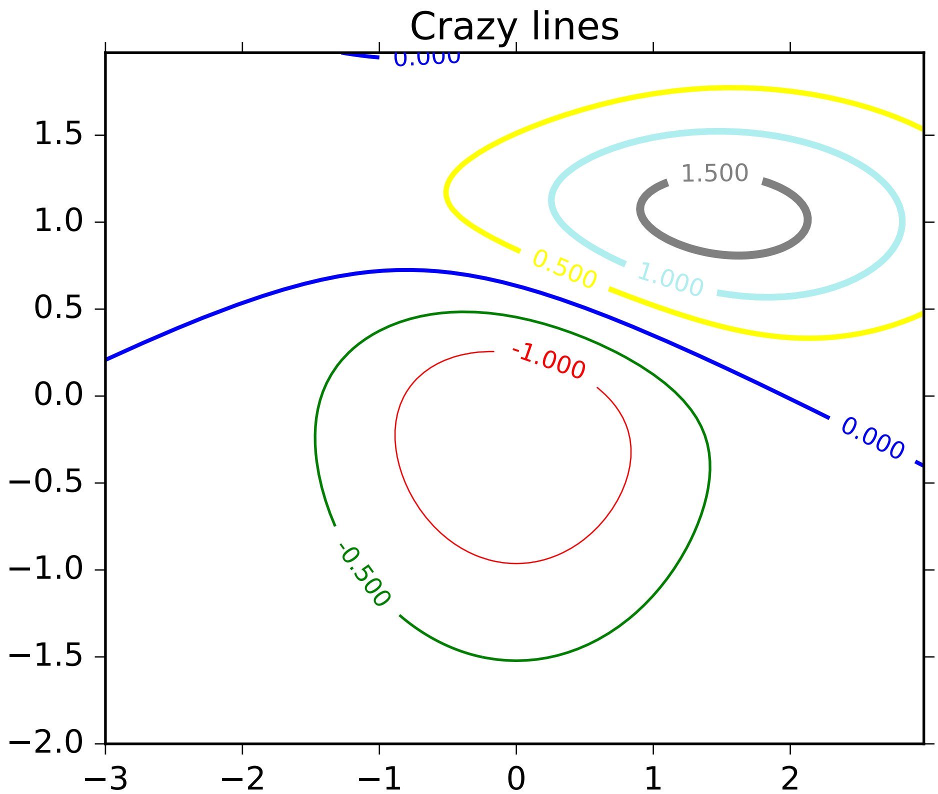
Examples:

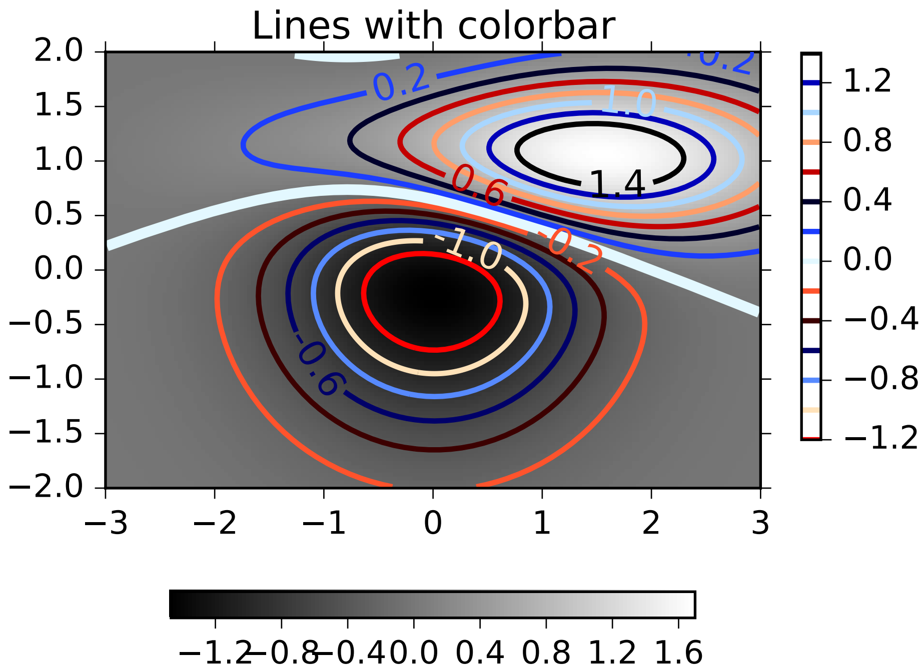


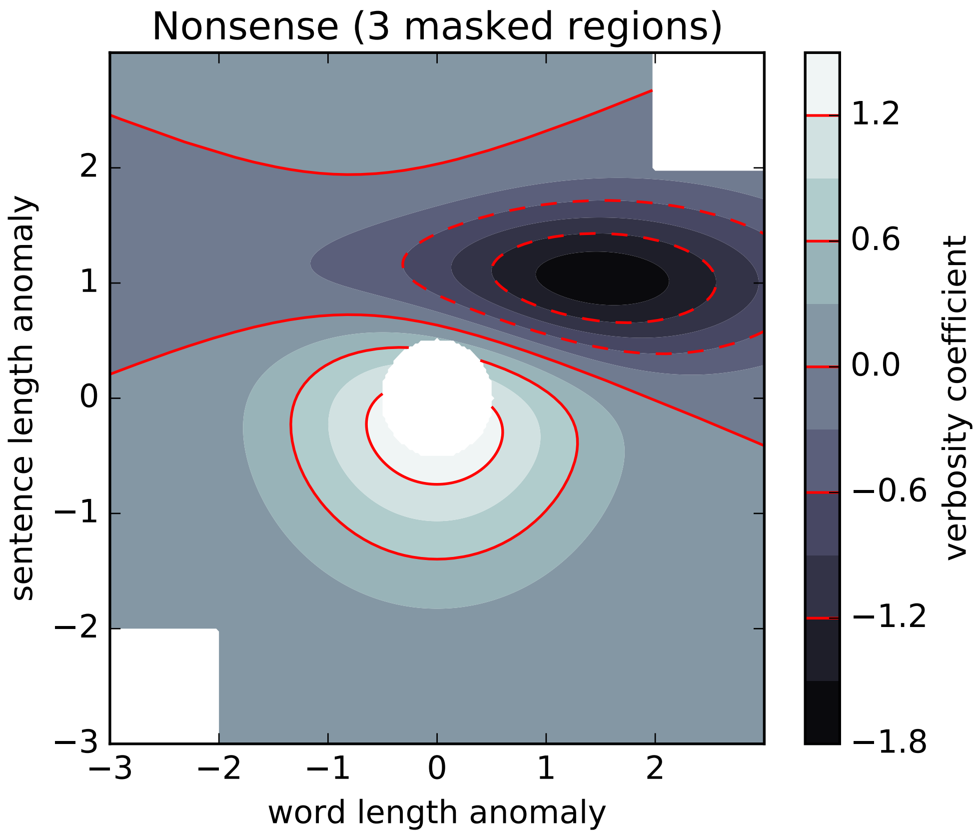

labels at selected locations
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
0.500
1.000
1.500
0.000
-1.000
-0.500

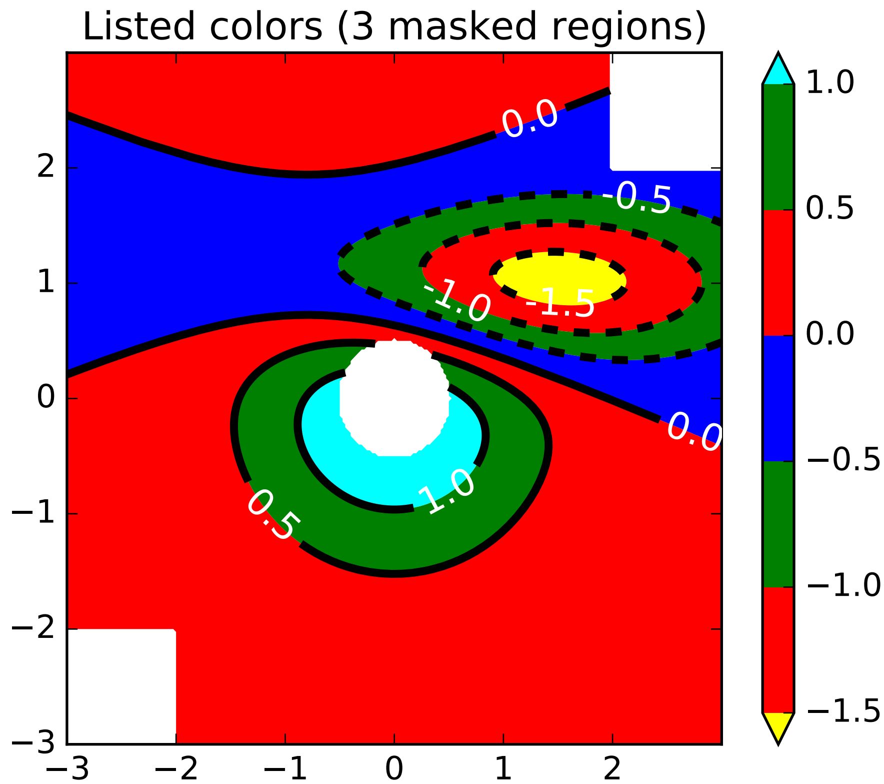

Single color - negative contours dashed
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

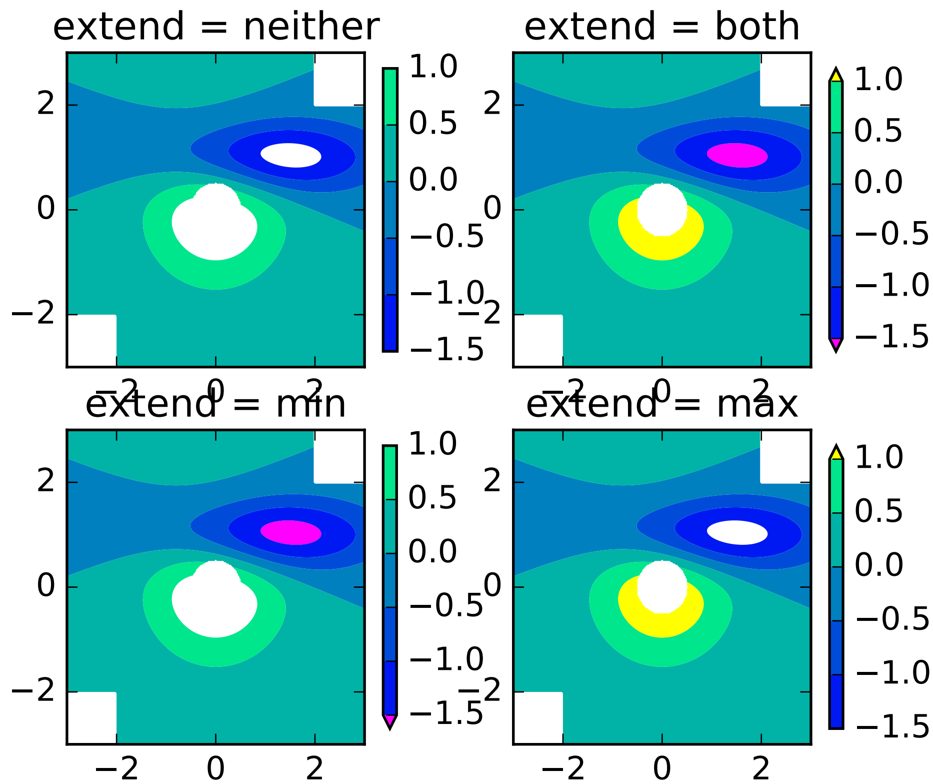

Single color - negative contours solid
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

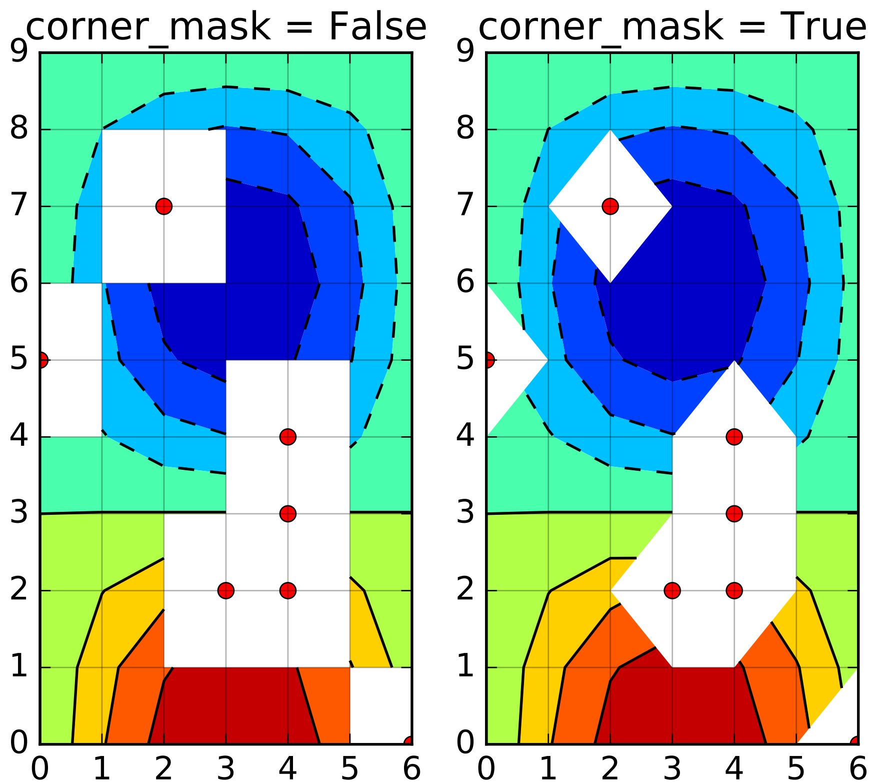

Crazy lines
1.500
1.000
0.500
0.000
-1.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Lines with colorbar
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
3
0.2
1.0
0.6
1.4
-1.0
-0.2
-0.6
1.2
0.8
0.4
0.0
−0.4
−0.8
−1.2
−1.2
−0.8
−0.4
0.0
0.4
0.8
1.2
1.6


Nonsense (3 masked regions)
sentence length anomaly
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
word length anomaly
1.2
0.6
0.0
−0.6
−1.2
−1.8
verbosity coefficient


Listed colors (3 masked regions)
0.0
-0.5
-1.0
-1.5
0.0
0.5
1.0
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
1.0
0.5
0.0
−0.5
−1.0
−1.5


extend = neither
extend = both
extend = min
extend = max
1.0
0.5
0.0
−0.5
−1.0
−1.5
2
0
−2
−2
0
2



convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

csd(*x*, *y*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, ***kwargs*)
Plot the cross-spectral density.

Call signature:

```
csd(x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None, return_line=None, **kwargs)
```

The cross spectral density P_{xy} by Welch's average periodogram method. The vectors *x* and *y* are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of *x* and *y* are averaged over each segment to compute P_{xy}, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero padded to *NFFT*.

***x*, *y*: 1-D arrays or sequences** Arrays or sequences containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or** callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***noverlap*: integer** The number of points of overlap between segments. The default value is 0 (no overlap).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

***return_line*: bool** Whether to include the line object plotted in the returned values. Default is False.

If *return_line* is False, returns the tuple (*Pxy*, *freqs*). If *return_line* is True, returns the tuple (*Pxy*, *freqs*. *line*):

***Pxy*: 1-D array** The values for the cross spectrum `P_{xy}` before scaling (complex valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *Pxy*

***line*: a `Line2D` instance** The line created by this function. Only returend if *return_line* is True.

For plotting, the power is plotted as $10\log_{10}(P_{xy})$ for decibels, though `P_{xy}` itself is returned.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the Line2D properties:

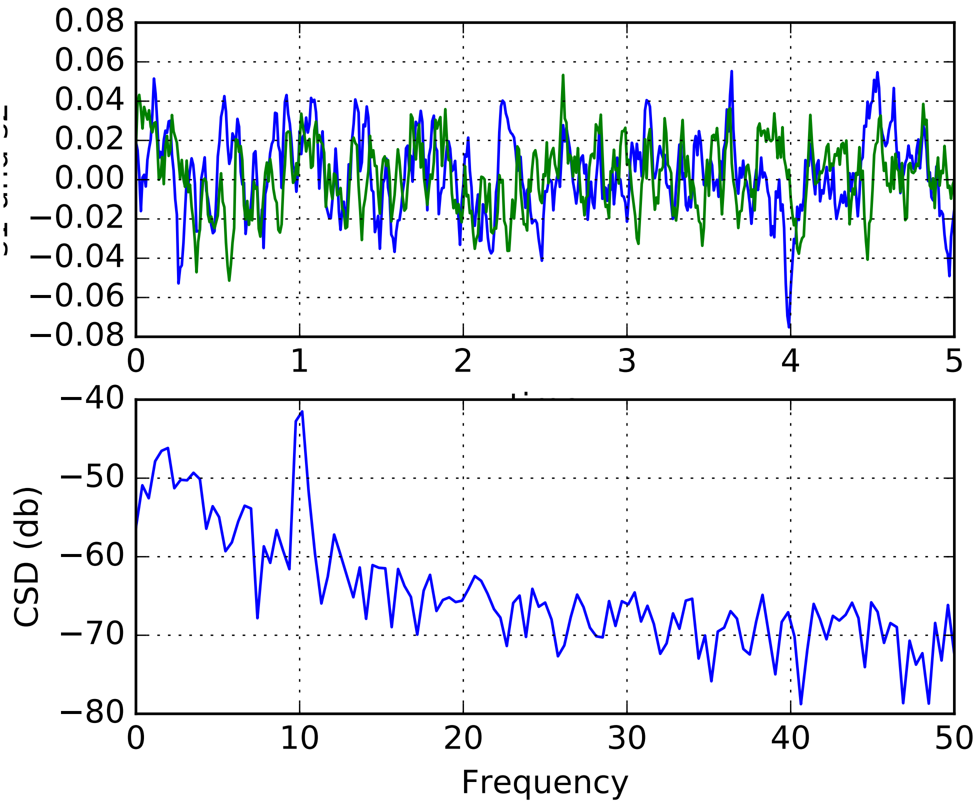
Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
`path_effects`	unknown
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string

C

Table 43.11 – continued from previous page

Property	Description
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:



See also:

psd() psd() is the equivalent to setting y=x.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

drag_pan(*button*, *key*, *x*, *y*)

Called when the mouse moves during a pan operation.

button is the mouse button number:

- 1: LEFT
- 2: MIDDLE
- 3: RIGHT

key is a "shift" key

x, *y* are the mouse coordinates in display coords.

Note: Intended to be overridden by new projection types.

draw(*artist*, *renderer*, **args*, ***kwargs*)
Draw everything (plot lines, axes, labels)

draw_artist(*a*)
This method can only be used after an initial draw which caches the renderer. It is used to efficiently update Axes data (axis ticks, labels, etc are not updated)

end_pan()
Called when a pan operation completes (when the mouse button is up.)

Note: Intended to be overridden by new projection types.

errorbar(*x*, *y*, *yerr=None*, *xerr=None*, *fmt=''*, *ecolor=None*, *elinewidth=None*, *capsize=None*, *barsabove=False*, *lolims=False*, *uplims=False*, *xlolims=False*, *xuplims=False*, *errorevery=1*, *capthick=None*, ***kwargs*)
Plot an errorbar graph.

Call signature:

```
errorbar(x, y, yerr=None, xerr=None,
         fmt='', ecolor=None, elinewidth=None, capsize=None,
         barsabove=False, lolims=False, uplims=False,
         xlolims=False, xuplims=False, errorevery=1,
         capthick=None)
```

Plot *x* versus *y* with error deltas in *yerr* and *xerr*. Vertical errorbars are plotted if *yerr* is not *None*. Horizontal errorbars are plotted if *xerr* is not *None*.

x, *y*, *xerr*, and *yerr* can all be scalars, which plots a single error bar at *x*, *y*.

Optional keyword arguments:

***xerr/yerr*: [scalar | N, Nx1, or 2xN array-like]** If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn at +/-value relative to the data.

If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2 relative to the data.

***fmt*: ['' | 'none' | plot format string]** The plot format symbol. If *fmt* is 'none' (case-insensitive), only the errorbars are plotted. This is used for adding errorbars to a bar plot, for example. Default is '', an empty plot format string; properties are then identical to the defaults for `plot()`.

***ecolor*: [*None* | mpl color]** A matplotlib color arg which gives the color the errorbar lines; if *None*, use the color of the line connecting the markers.

***elinewidth*: scalar** The linewidth of the errorbar lines. If *None*, use the linewidth.

***capsize*: scalar** The length of the error bar caps in points; if *None*, it will take the value from `errorbar.capsize` *rcParam*.

***capthick*: scalar** An alias kwarg to *markeredgewidth* (a.k.a. - *mew*). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if *mew* or *markeredgewidth* are given, then they will over-ride *capthick*. This may change in future releases.

***barsabove*: [*True* | *False*]** if *True*, will plot the errorbars above the plot symbols. Default is below.

***lolims* / *uplims* / *xlolims* / *xuplims*: [*False* | *True*]** These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. lims-arguments may be of the same type as *xerr* and *yerr*. To use limits with inverted axes, `set_xlim()` or `set_ylim()` must be called before `errorbar()`.

***errorevery*: positive integer** subsamples the errorbars. e.g., if errorevery=5, errorbars for every 5-th datapoint will be plotted. The data plot itself still shows all data points.

All other keyword arguments are passed on to the plot command for the markers. For example, this code makes big red squares with thick green edges:

```
x,y,yerr = rand(3,10)
errorbar(x, y, yerr, marker='s',
         mfc='red', mec='green', ms=20, mew=4)
```

where *mfc*, *mec*, *ms* and *mew* are aliases for the longer property names, *markerfacecolor*, *markeredgecolor*, *markersize* and *markeredgewith*.

valid kwargs for the marker properties are

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an *Axes* instance
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a *matplotlib.figure.Figure* instance
	C

Table 43.12 – continued from previous page

Property	Description
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '
linewidth or lw	float value in points
marker	*A valid marker style*
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a *matplotlib.transforms.Transform* instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

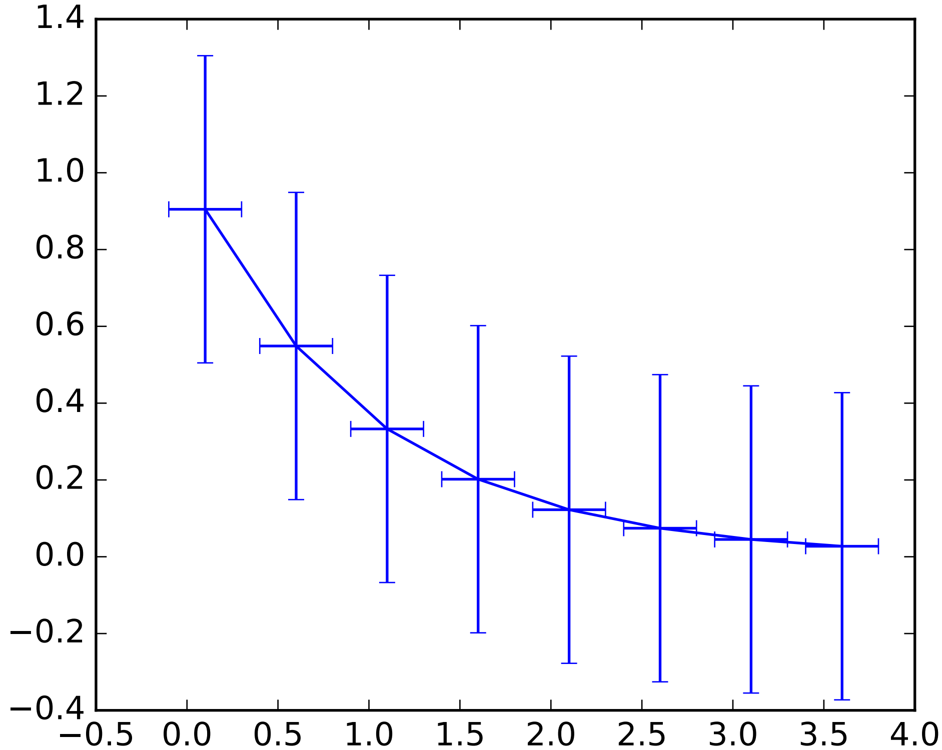
Returns (*plotline*, *caplines*, *barlinecols*):

***plotline*: `Line2D` instance** *x*, *y* plot markers and/or line

***caplines*: list of error bar cap** *Line2D* instances

***barlinecols*: list of** *LineCollection* instances for the horizontal and vertical error ranges.

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'yerr', 'y', 'xerr', 'x'.

eventplot(*positions*, *orientation='horizontal'*, *lineoffsets=1*, *linelengths=1*, *linewidths=None*, *colors=None*, *linestyles='solid'*, ***kwargs*)

Plot identical parallel lines at specific positions.

Call signature:

```
eventplot(positions, orientation='horizontal', lineoffsets=0,
          linelengths=1, linewidths=None, color =None,
          linestyles='solid'
```

Plot parallel lines at the given positions. positions should be a 1D or 2D array-like object, with each row corresponding to a row or column of lines.

This type of plot is commonly used in neuroscience for representing neural events, where it is commonly called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

orientation [['horizontal' | 'vertical']] 'horizontal' : the lines will be vertical and arranged in rows "vertical' : lines will be horizontal and arranged in columns

lineoffsets **:** A float or array-like containing floats.

linelengths **:** A float or array-like containing floats.

linewidths **:** A float or array-like containing floats.

colors must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed) or a list of such sequences

linestyles **:** ['solid' | 'dashed' | 'dashdot' | 'dotted'] or an array of these values

For linelengths, linewidths, colors, and linestyles, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as positions, and each value will be applied to the corresponding row or column in positions.

Returns a list of `matplotlib.collections.EventCollection` objects that were added.

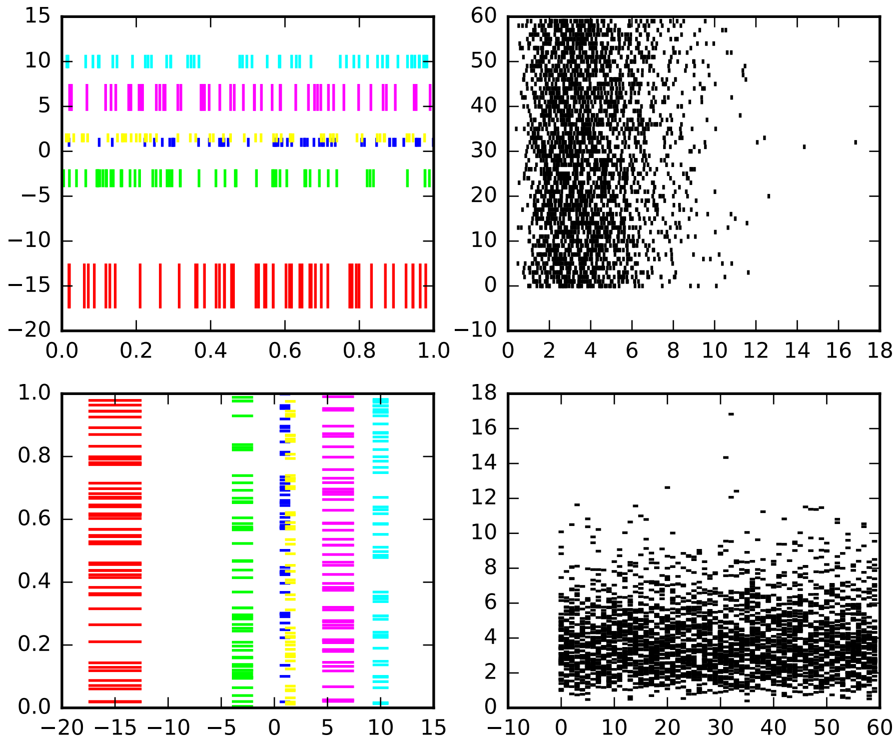
kwargs are `LineCollection` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color spec or sequence of specs
`facecolor` or facecolors	matplotlib color spec or sequence of specs
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` |
`linewidth` or lw or linewidths	float or sequence of floats
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`path_effects`	unknown
`paths`	unknown
`picker`	[None|float|boolean|callable]
`pickradius`	unknown
`rasterized`	[True | False | None]
	C

Table 43.13 – continued from previous page

Property	Description
segments	unknown
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
urls	unknown
verts	unknown
visible	[True | False]
zorder	any number

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'linelengths', 'linestyles', 'linewidths', 'positions', 'lineoffsets', 'colors'.

fill(**args*, ***kwargs*)

Plot filled polygons.

Call signature:

```
fill(*args, **kwargs)
```

args is a variable length argument, allowing for multiple *x*, *y* pairs with an optional color format string; see `plot()` for details on the argument parsing. For example, to plot a polygon with vertices at *x*, *y* in blue.:

```
ax.fill(x,y, 'b' )
```

An arbitrary number of *x*, *y*, *color* groups can be specified:

```
ax.fill(x1, y1, 'g', x2, y2, 'r')
```

Return value is a list of `Patch` instances that were added.

The same color strings that `plot()` supports are supported by the fill format string.

If you would like to fill below a curve, e.g., shade a region between 0 and *y* along *x*, use `fill_between()`

The *closed* kwarg will close the polygon when *True* (default).

kwargs control the `Polygon` properties:

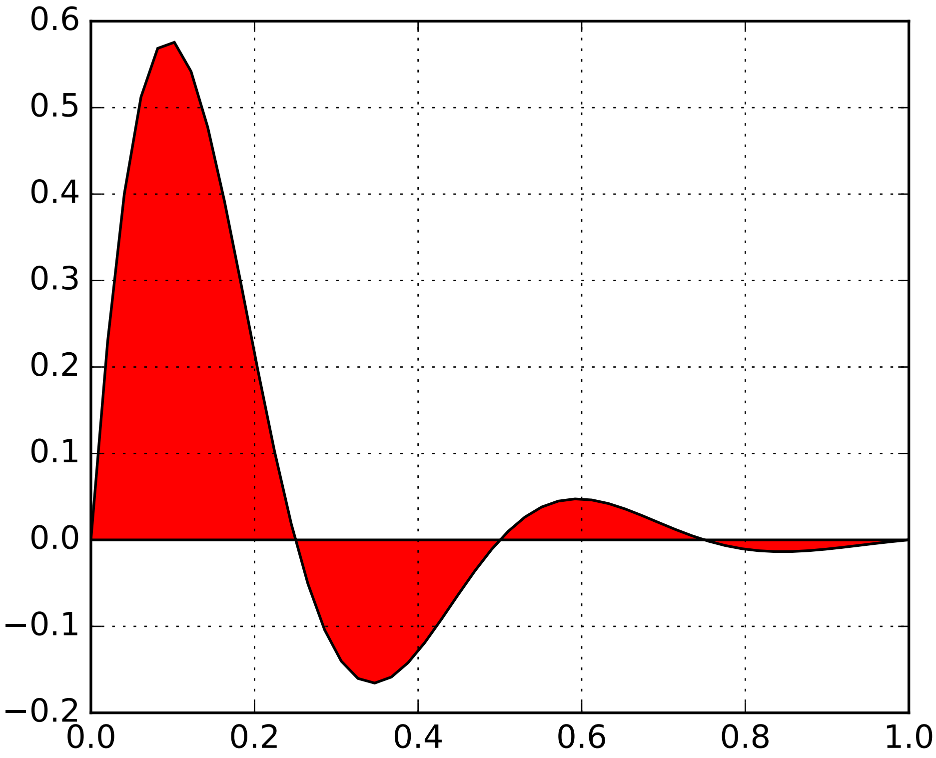
Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default

Continued on

Table 43.14 – continued from previous page

Property	Description
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

`fill_between`(*x*, *y1*, *y2=0*, *where=None*, *interpolate=False*, *step=None*, ***kwargs*)
Make filled polygons between two curves.

Create a *PolyCollection* filling the regions between *y1* and *y2* where `where==True`

Parameters **x** : array
An N-length array of the x data

y1 : array
An N-length array (or scalar) of the y data

y2 : array
An N-length array (or scalar) of the y data

where : array, optional
If `None`, default to fill between everywhere. If not `None`, it is an N-length numpy boolean array and the fill will only happen over the regions where `where==True`.

interpolate : bool, optional
If `True`, interpolate between the two lines to find the precise point of intersection. Otherwise, the start and end points of the filled region will only occur on explicit values in the *x* array.

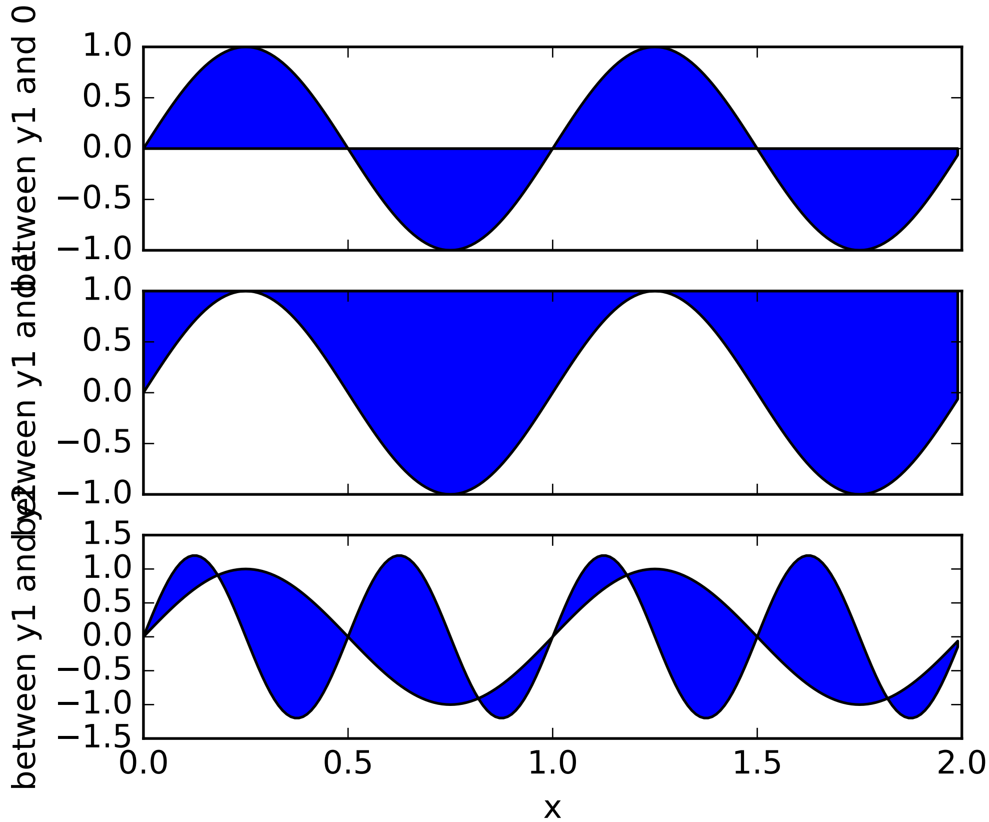
step : {'pre', 'post', 'mid'}, optional
If not None, fill with step logic.

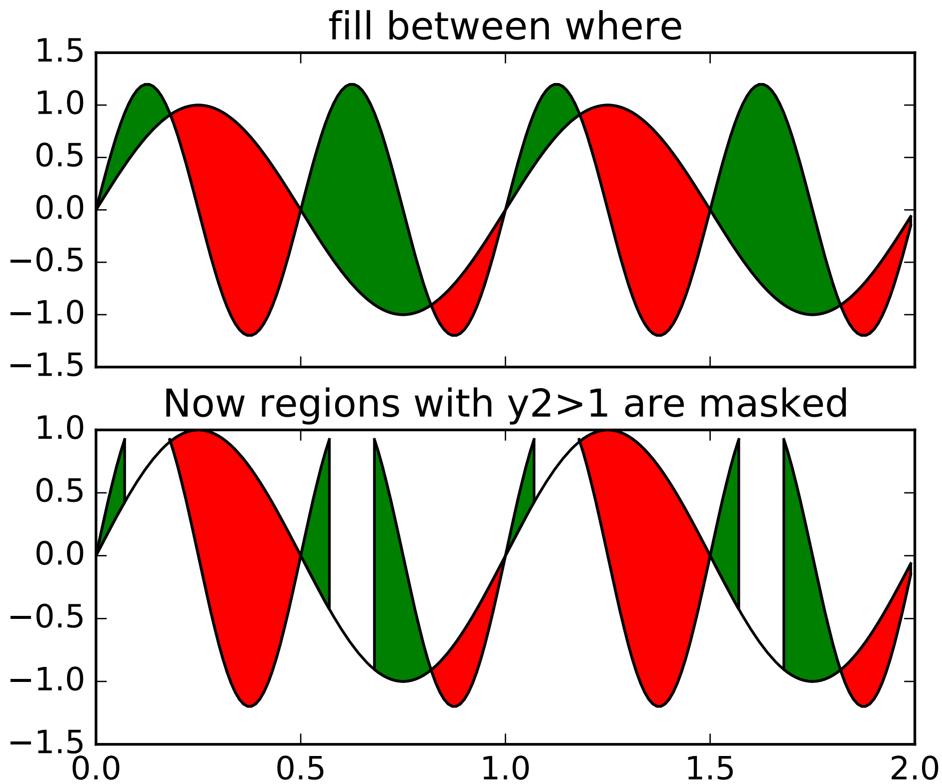
Notes

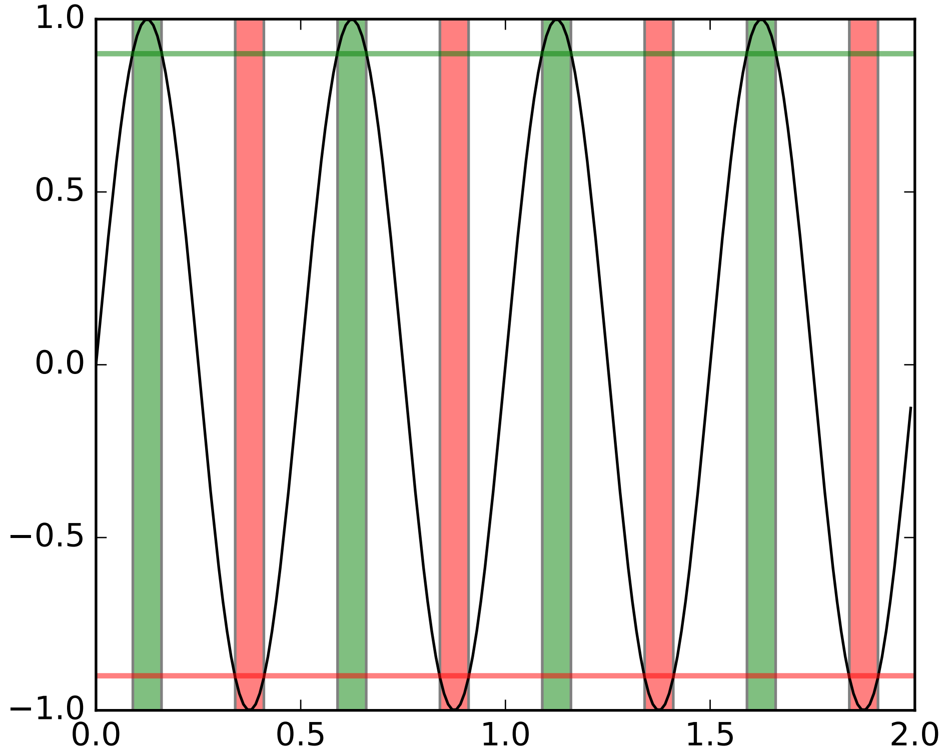
In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y1', 'y2', 'where', 'x'.

Examples


between y1 and 0
between y1 and 1
between y1 and y2
1.0
0.5
0.0
−0.5
−1.0
1.5
−1.5
0.0
0.5
1.0
1.5
2.0
x


fill between where
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Now regions with y2>1 are masked
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.5
1.0
1.5
2.0



fill_betweenx(*y*, *x1*, *x2=0*, *where=None*, *step=None*, ***kwargs*)

Make filled polygons between two horizontal curves.

Call signature:

```
fill_betweenx(y, x1, x2=0, where=None, **kwargs)
```

Create a *PolyCollection* filling the regions between *x1* and *x2* where `where==True`

Parameters y : array

An N-length array of the y data

x1 : array

An N-length array (or scalar) of the x data

x2 : array, optional

An N-length array (or scalar) of the x data

where : array, optional

If *None*, default to fill between everywhere. If not *None*, it is a N length numpy boolean array and the fill will only happen over the regions where `where==True`

step : {'pre', 'post', 'mid'}, optional

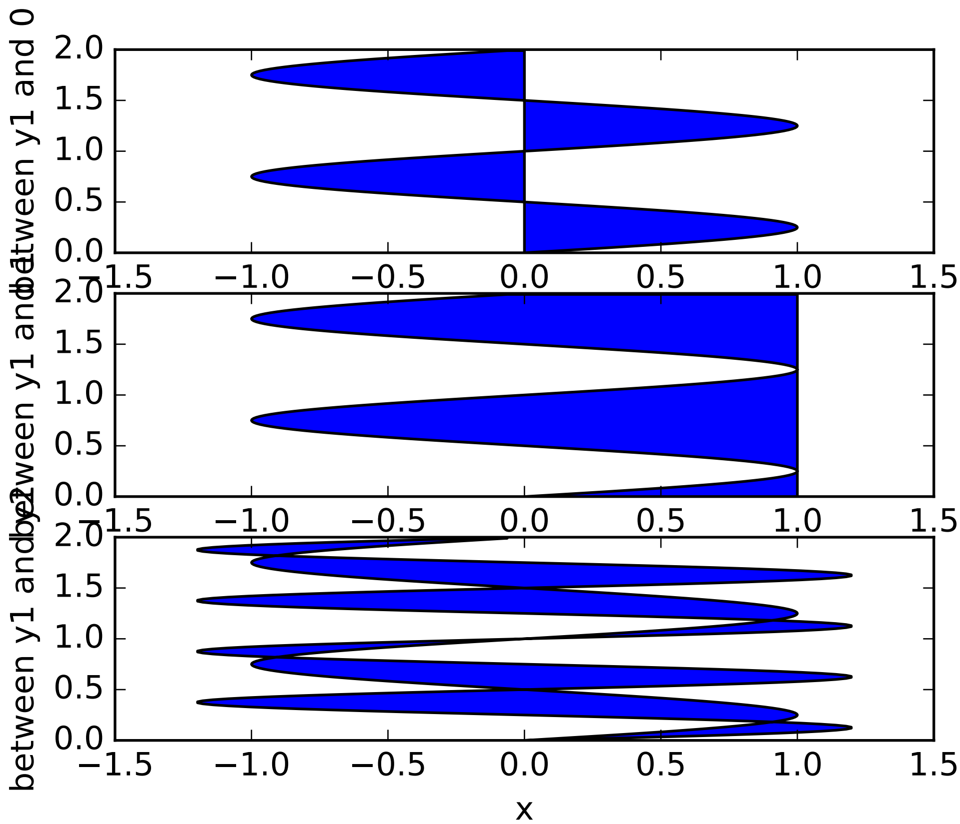
If not None, fill with step logic.

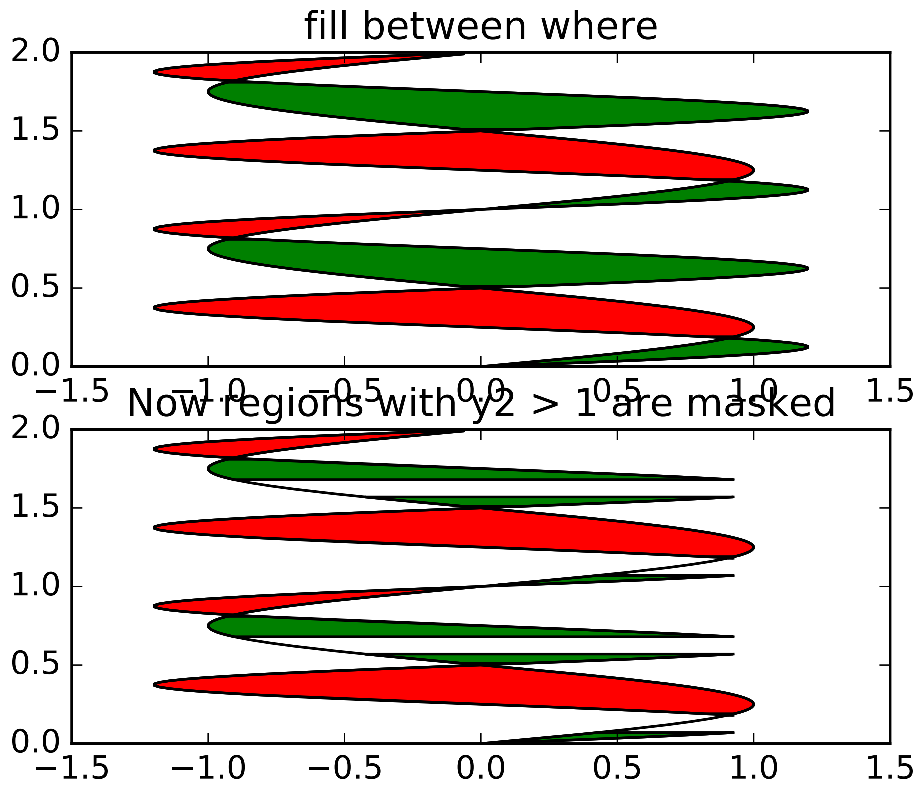
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x1', 'where', 'y', 'x2'.

Examples





findobj(*match=None*, *include_self=True*)

Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_coord(*x*, *y*)

Return a format string formatting the *x*, *y* coord

format_cursor_data(*data*)

Return *cursor data* string formatted.

format_xdata(*x*)

Return *x* string formatted. This function will use the attribute self.fmt_xdata if it is callable, else will fall back on the xaxis major formatter

format_ydata(*y*)

Return y string formatted. This function will use the `fmt_ydata` attribute if it is callable, else will fall back on the yaxis major formatter

get_adjustable()

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_anchor()

get_animated()
Return the artist's animated state

get_aspect()

get_autoscale_on()
Get whether autoscaling is applied for both axes on plot commands

get_autoscalex_on()
Get whether autoscaling for the x-axis is applied on plot commands

get_autoscaley_on()
Get whether autoscaling for the y-axis is applied on plot commands

get_axes()
Return the *Axes* instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_axes_locator()
return axes_locator

get_axis_bgcolor()
Return the axis background color

get_axisbelow()
Get whether axis below is true or not

get_children()
return a list of child artists

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_cursor_props()
Return the cursor propertiess as a (*linewidth*, *color*) tuple, where *linewidth* is a float and *color* is an RGBA tuple

get_data_ratio()
Returns the aspect ratio of the raw data.

This method is intended to be overridden by new projection types.

get_data_ratio_log()
Returns the aspect ratio of the raw data in log scale. Will be used when both axis scales are in log.

get_default_bbox_extra_artists()

get_figure()
Return the `Figure` instance the artist belongs to.

get_frame_on()
Get whether the axes rectangle patch is drawn

get_gid()
Returns the group id

get_images()
return a list of Axes images contained by the Axes

get_label()
Get the label used for this artist in the legend.

get_legend()
Return the legend.Legend instance, or None if no legend is defined

get_legend_handles_labels(*legend_handler_map=None*)
Return handles and labels for legend

`ax.legend()` is equivalent to

```
h, l = ax.get_legend_handles_labels()
ax.legend(h, l)
```

get_lines()
Return a list of lines contained by the Axes

get_navigate()
Get whether the axes responds to navigation commands

get_navigate_mode()
Get the navigation toolbar button status: 'PAN', 'ZOOM', or None

get_path_effects()

get_picker()
Return the picker object used by this artist

get_position(*original=False*)
Return the a copy of the axes rectangle as a Bbox

get_rasterization_zorder()
Get zorder value below which artists will be rasterized

get_rasterized()
return True if the artist is to be rasterized

get_renderer_cache()

get_shared_x_axes()
Return a copy of the shared axes Grouper object for x axes

get_shared_y_axes()
Return a copy of the shared axes Grouper object for y axes

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_tightbbox(*renderer*, *call_axes_locator=True*)
Return the tight bounding box of the axes. The dimension of the Bbox in canvas coordinate.

If *call_axes_locator* is *False*, it does not call the _axes_locator attribute, which is necessary to get the correct bounding box. `call_axes_locator==False` can be used if the caller is only intereted in the relative size of the tightbbox compared to the axes bbox.

get_title(*loc='center'*)
Get an axes title.

Get one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

Parameters loc : {'center', 'left', 'right'}, str, optional

Which title to get, defaults to 'center'

Returns title: str
The title text string.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_url()
Returns the url

get_visible()
Return the artist's visiblity

get_window_extent(**args*, ***kwargs*)
get the axes bounding box in display space; *args* and *kwargs* are empty

get_xaxis()
Return the XAxis instance

get_xaxis_text1_transform(*pad_points*)
Get the transformation used for drawing x-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates. Returns a 3-tuple of the form:

```
(transform, valign, halign)
```

where *valign* and *halign* are requested alignments for the text.

Note: This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

get_xaxis_text2_transform(*pad_points*)
Get the transformation used for drawing the secondary x-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in data coordinates and the y-direction is in axis coordinates. Returns a 3-tuple of the form:

```
(transform, valign, halign)
```

where *valign* and *halign* are requested alignments for the text.

Note: This transformation is primarily used by the *Axis* class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

get_xaxis_transform(*which='grid'*)
Get the transformation used for drawing x-axis labels, ticks and gridlines. The x-direction is in data coordinates and the y-direction is in axis coordinates.

Note: This transformation is primarily used by the `Axis` class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

get_xbound()
Returns the x-axis numerical bounds where:

```
lowerBound < upperBound
```

get_xgridlines()
Get the x grid lines as a list of Line2D instances

get_xlabel()
Get the xlabel text string.

get_xlim()
Get the x-axis range [*left*, *right*]

get_xmajorticklabels()
Get the xtick labels as a list of `Text` instances.

get_xminorticklabels()
Get the x minor tick labels as a list of `matplotlib.text.Text` instances.

get_xscale()
Return the xaxis scale string: linear, log, logit, symlog

get_xticklabels(*minor=False*, *which=None*)
Get the x tick labels as a list of `Text` instances.

Parameters minor : bool
If True return the minor ticklabels, else return the major ticklabels

which : None, ('minor', 'major', 'both')
Overrides `minor`.

Selects which ticklabels to return

Returns ret : list
List of `Text` instances.

get_xticklines()
Get the xtick lines as a list of Line2D instances

get_xticks(*minor=False*)
Return the x ticks as a list of locations

get_yaxis()
Return the YAxis instance

get_yaxis_text1_transform(*pad_points*)
Get the transformation used for drawing y-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates. Returns a 3-tuple of the form:

```
(transform, valign, halign)
```

where *valign* and *halign* are requested alignments for the text.

Note: This transformation is primarily used by the `Axis` class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

get_yaxis_text2_transform(*pad_points*)
Get the transformation used for drawing the secondary y-axis labels, which will add the given amount of padding (in points) between the axes and the label. The x-direction is in axis coordinates and the y-direction is in data coordinates. Returns a 3-tuple of the form:

```
(transform, valign, halign)
```

where *valign* and *halign* are requested alignments for the text.

Note: This transformation is primarily used by the `Axis` class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

get_yaxis_transform(*which='grid'*)
Get the transformation used for drawing y-axis labels, ticks and gridlines. The x-direction is in axis coordinates and the y-direction is in data coordinates.

Note: This transformation is primarily used by the `Axis` class, and is meant to be overridden by new kinds of projections that may need to place axis elements in different locations.

get_ybound()
Return y-axis numerical bounds in the form of `lowerBound < upperBound`

get_ygridlines()
Get the y grid lines as a list of Line2D instances

get_ylabel()
Get the ylabel text string.

get_ylim()
Get the y-axis range [*bottom*, *top*]

get_ymajorticklabels()
Get the major y tick labels as a list of `Text` instances.

get_yminorticklabels()
Get the minor y tick labels as a list of `Text` instances.

get_yscale()
Return the yaxis scale string: linear, log, logit, symlog

get_yticklabels(*minor=False*, *which=None*)
Get the x tick labels as a list of *Text* instances.

Parameters minor : bool
If True return the minor ticklabels, else return the major ticklabels

which : None, ('minor', 'major', 'both')
Overrides `minor`.

Selects which ticklabels to return

Returns ret : list
List of *Text* instances.

get_yticklines()
Get the ytick lines as a list of Line2D instances

get_yticks(*minor=False*)
Return the y ticks as a list of locations

get_zorder()
Return the `Artist`'s zorder.

grid(*b=None*, *which='major'*, *axis='both'*, ***kwargs*)
Turn the axes grids on or off.

Call signature:

```
grid(self, b=None, which='major', axis='both', **kwargs)
```

Set the axes grids on or off; *b* is a boolean. (For MATLAB compatibility, *b* may also be a string, 'on' or 'off'.)

If *b* is *None* and `len(kwargs)==0`, toggle the grid state. If *kwargs* are supplied, it is assumed that you want a grid and *b* is thus set to *True*.

which can be 'major' (default), 'minor', or 'both' to control whether major tick grids, minor tick grids, or both are affected.

axis can be 'both' (default), 'x', or 'y' to control which set of gridlines are drawn.

kwargs are used to set the grid line properties, e.g.,:

```
ax.grid(color='r', linestyle='-', linewidth=2)
```

Valid *Line2D* kwargs are

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an *Axes* instance
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
	C

Table 43.15 – continued from previous page

Property	Description
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a *matplotlib.figure.Figure* instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
linewidth or lw	float value in points
marker	*A valid marker style*
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a *matplotlib.transforms.Transform* instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

has_data()

Return *True* if any artists have been added to axes.

This should not be used to determine whether the *dataLim* need to be updated, and may not actually be useful for anything.

have_units()

Return *True* if units are set on the *x* or *y* axes

hexbin(*x*, *y*, *C=None*, *gridsize=100*, *bins=None*, *xscale='linear'*, *yscale='linear'*, *extent=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *edgecolors='none'*, *reduce_C_function=<function mean>*, *mincnt=None*, *marginals=False*, ***kwargs*)

Make a hexagonal binning plot.

Call signature:

```
hexbin(x, y, C = None, gridsize = 100, bins = None,
       xscale = 'linear', yscale = 'linear',
       cmap=None, norm=None, vmin=None, vmax=None,
       alpha=None, linewidths=None, edgecolors='none'
       reduce_C_function = np.mean, mincnt=None, marginals=True
       **kwargs)
```

Make a hexagonal binning plot of *x* versus *y*, where *x*, *y* are 1-D sequences of the same length, *N*. If *C* is *None* (the default), this is a histogram of the number of occurences of the observations at (x[i],y[i]).

If *C* is specified, it specifies values at the coordinate (x[i],y[i]). These values are accumulated for each hexagonal bin and then reduced according to *reduce_C_function*, which defaults to numpy's mean function (np.mean). (If *C* is specified, it must also be a 1-D sequence of the same length as *x* and *y*.)

x, *y* and/or *C* may be masked arrays, in which case only unmasked points will be plotted.

Optional keyword arguments:

gridsize: **[100 | integer]** The number of hexagons in the *x*-direction, default is 100. The corresponding number of hexagons in the *y*-direction is chosen such that the hexagons are approximately regular. Alternatively, gridsize can be a tuple with two elements specifying the number of hexagons in the *x*-direction and the *y*-direction.

bins: **[*None* | 'log' | integer | sequence]** If *None*, no binning is applied; the color of each hexagon directly corresponds to its count value.

If 'log', use a logarithmic scale for the color map. Internally, $log_{10}(i + 1)$ is used to determine the hexagon color.

If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.

If a sequence of values, the values of the lower bound of the bins to be used.

xscale: **['linear' | 'log']** Use a linear or log10 scale on the horizontal axis.

scale: **['linear' | 'log']** Use a linear or log10 scale on the vertical axis.

mincnt: **[*None* | a positive integer]** If not *None*, only display cells with more than *mincnt* number of points in the cell

marginals: **[*True* | *False*]** if marginals is *True*, plot the marginal density as colormapped rectagles along the bottom of the x-axis and left of the y-axis

extent: **[*None* | scalars (left, right, bottom, top)]** The limits of the bins. The default assigns the limits based on gridsize, x, y, xscale and yscale.

Other keyword arguments controlling color mapping and normalization arguments:

cmap: **[*None* | Colormap]** a `matplotlib.colors.Colormap` instance. If *None*, defaults to rc `image.cmap`.

***norm*: [*None* | Normalize]** `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1.

***vmin* / *vmax*: scalar** *vmin* and *vmax* are used in conjunction with *norm* to normalize luminance data. If either are *None*, the min and max of the color array *C* is used. Note if you pass a norm instance, your settings for *vmin* and *vmax* will be ignored.

alpha*: scalar between 0 and 1, or *None the alpha value for the patches

***linewidths*: [*None* | scalar]** If *None*, defaults to rc lines.linewidth. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by `RegularPolyCollection`.

Other keyword arguments controlling the Collection properties:

***edgecolors*: [*None* | 'none' | mpl color | color sequence]** If `'none'`, draws the edges in the same color as the fill color. This is the default, as it avoids unsightly unpainted pixels between the hexagons.

If *None*, draws the outlines in the default color.

If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.

Here are the standard descriptions of all the `Collection` kwargs:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color spec or sequence of specs
`facecolor` or facecolors	matplotlib color spec or sequence of specs
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` |
`linewidth` or lw or linewidths	float or sequence of floats
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`path_effects`	unknown
`picker`	[None|float|boolean|callable]

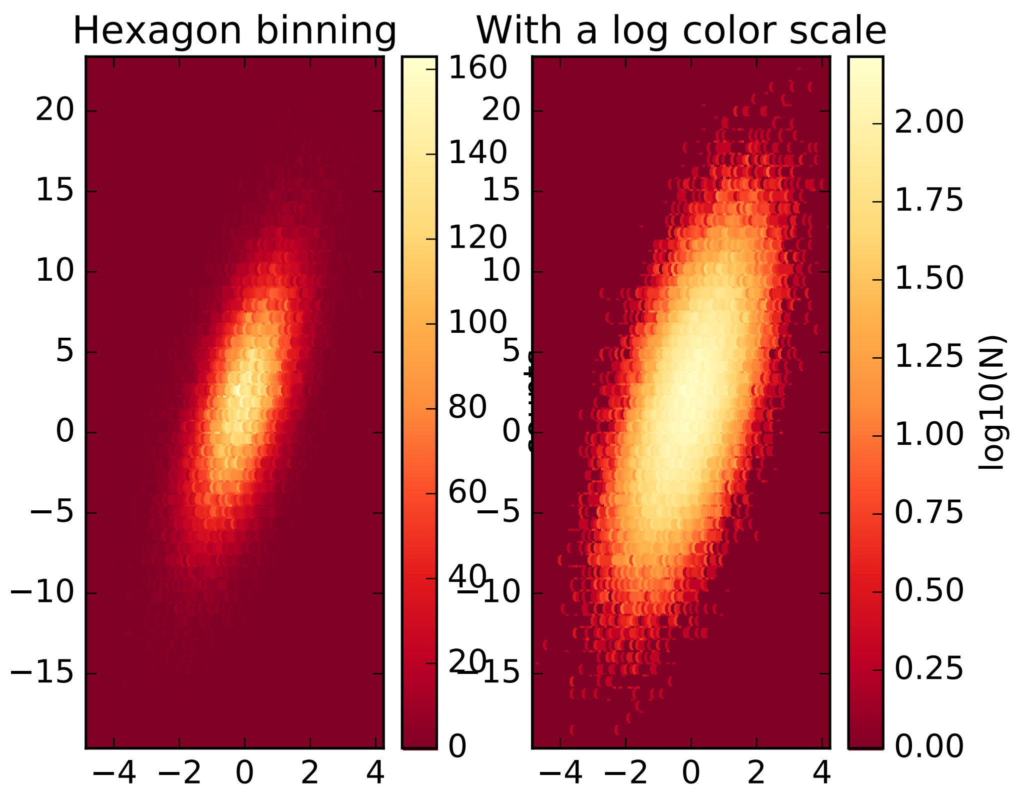
C

Table 43.16 – continued from previous page

Property	Description
pickradius	unknown
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

The return value is a PolyCollection instance; use get_array() on this PolyCollection to get the counts in each hexagon. If *marginals* is *True*, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes *hbar* and *vbar*.

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

hist(*x*, *bins=10*, *range=None*, *normed=False*, *weights=None*, *cumulative=False*, *bottom=None*, *histtype='bar'*, *align='mid'*, *orientation='vertical'*, *rwidth=None*, *log=False*, *color=None*, *label=None*, *stacked=False*, ***kwargs*)

Plot a histogram.

Compute and draw the histogram of *x*. The return value is a tuple (*n*, *bins*, *patches*) or ([*n0*, *n1*, ...], *bins*, [*patches0*, *patches1*,...]) if the input contains multiple data.

Multiple data can be provided via *x* as a list of datasets of potentially different length ([*x0*, *x1*, ...]), or as a 2-D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form.

Masked arrays are not supported at present.

Parameters x : (n,) array or sequence of (n,) arrays

Input values, this takes either a single array or a sequency of arrays which are not required to be of the same length

bins : integer or array_like, optional

If an integer is given, `bins + 1` bin edges are returned, consistently with `numpy.histogram()` for numpy version >= 1.3.

Unequally spaced bins are supported if `bins` is a sequence.

default is 10

range : tuple or None, optional

The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, `range` is (x.min(), x.max()). Range has no effect if `bins` is a sequence.

If `bins` is a sequence or `range` is specified, autoscaling is based on the specified bin range instead of the range of x.

Default is `None`

normed : boolean, optional

If `True`, the first element of the return tuple will be the counts normalized to form a probability density, i.e., `n/(len(x)'dbin)`, i.e., the integral of the histogram will sum to 1. If *stacked* is also *True*, the sum of the histograms is normalized to 1.

Default is `False`

weights : (n,) array_like or None, optional

An array of weights, of the same shape as `x`. Each value in `x` only contributes its associated weight towards the bin count (instead of 1). If `normed` is True, the weights are normalized, so that the integral of the density over the range remains 1.

Default is `None`

cumulative : boolean, optional

If `True`, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints. If `normed` is also `True` then the histogram is normalized such that the last bin equals 1. If `cumulative` evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if `normed` is also `True`, then the histogram is normalized such that the first bin equals 1.

Default is `False`

bottom : array_like, scalar, or None

Location of the bottom baseline of each bin. If a scalar, the base line for each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

Default is `None`

histtype : {'bar', 'barstacked', 'step', 'stepfilled'}, optional

The type of histogram to draw.

- 'bar' is a traditional bar-type histogram. If multiple data are given the bars are aranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

Default is 'bar'

align : {'left', 'mid', 'right'}, optional

Controls how the histogram is plotted.

- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

Default is 'mid'

orientation : {'horizontal', 'vertical'}, optional

If 'horizontal', `barh` will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

rwidth : scalar or None, optional

The relative width of the bars as a fraction of the bin width. If `None`, automatically compute the width.

Ignored if `histtype` is 'step' or 'stepfilled'.

Default is `None`

log : boolean, optional

If `True`, the histogram axis will be set to a log scale. If `log` is `True` and `x` is a 1D array, empty bins will be filtered out and only the non-empty (`n`, `bins`, `patches`) will be returned.

Default is `False`

color : color or array_like of colors or None, optional

Color spec or sequence of color specs, one per dataset. Default (`None`) uses the standard line color sequence.

Default is `None`

label : string or None, optional

String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that the legend command will work as expected.

default is `None`

stacked : boolean, optional

If `True`, multiple data are stacked on top of each other If `False` multiple data are aranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

Default is `False`

Returns n : array or list of arrays

The values of the histogram bins. See **normed** and **weights** for a description of the possible semantics. If input **x** is an array, then this is an array of length **nbins**. If input is a sequence arrays `[data1, data2,..]`, then this is a list of arrays with the values of the histograms for each of the arrays in the same order.

bins : array

The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

patches : list or list of lists

Silent list of individual patches used to create the histogram or list of such list if multiple input datasets.

Other Parameters kwargs : *`Patch`* properties

See also:

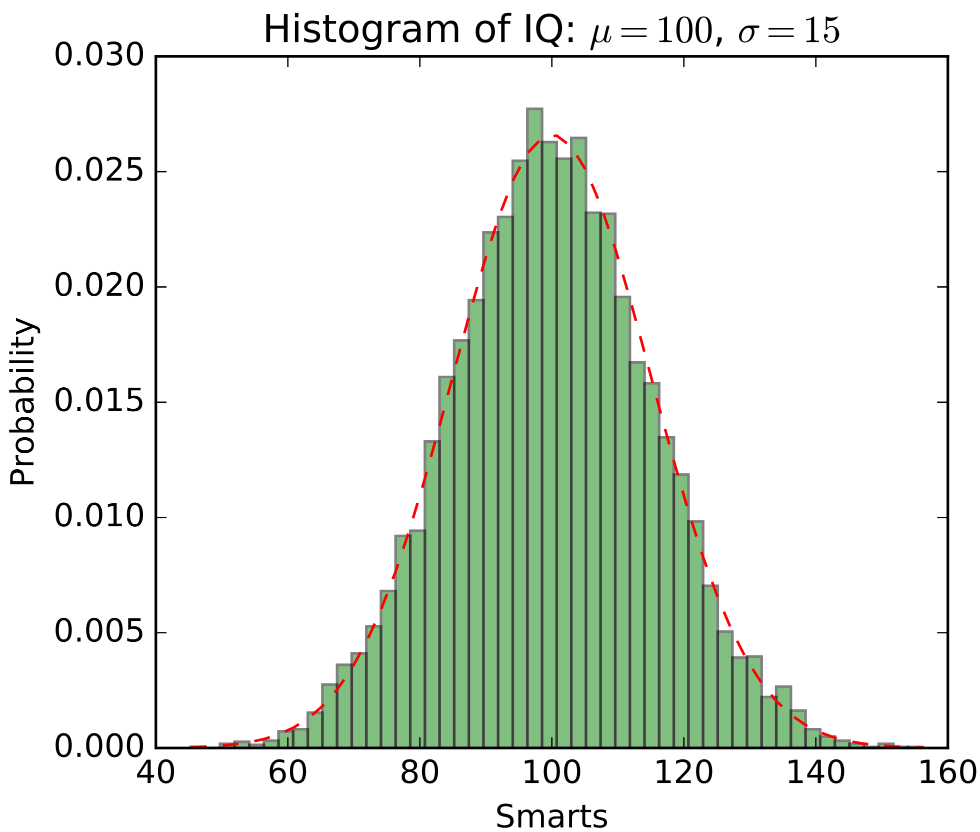
`hist2d` 2D histograms

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'weights', 'x'.

Examples



hist2d(*x*, *y*, *bins=10*, *range=None*, *normed=False*, *weights=None*, *cmin=None*, *cmax=None*, ***kwargs*)

Make a 2D histogram plot.

Parameters x, y: array_like, shape (n,)

Input values

bins: [None | int | [int, int] | array_like | [array, array]]

The bin specification:

- If int, the number of bins for the two dimensions (nx=ny=bins).
- If [int, int], the number of bins in each dimension (nx, ny = bins).
- If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).
- If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

The default value is 10.

range : array_like shape(2, 2), optional, default: None

The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): [[xmin, xmax], [ymin, ymax]]. All values outside of this range will be considered outliers and not tallied in the histogram.

normed : boolean, optional, default: False
Normalize histogram.

weights : array_like, shape (n,), optional, default: None
An array of values w_i weighing each sample (x_i, y_i).

cmin : scalar, optional, default: None
All bins that has count less than cmin will not be displayed and these count values in the return value count histogram will also be set to nan upon return

cmax : scalar, optional, default: None
All bins that has count more than cmax will not be displayed (set to none before passing to imshow) and these count values in the return value count histogram will also be set to nan upon return

Returns The return value is `(counts, xedges, yedges, Image)`.

Other Parameters kwargs : `pcolorfast()` properties.

See also:

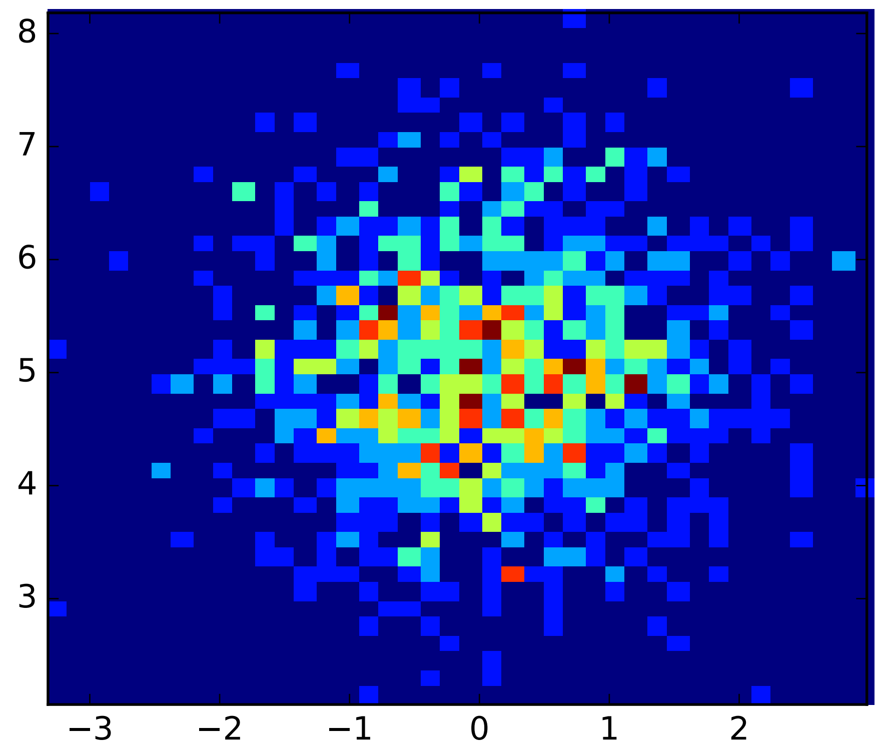
`hist` 1D histogram

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'weights', 'x'.

Examples



hitlist(*event*)

List the children of the artist which contain the mouse event *event*.

hlines(*y*, *xmin*, *xmax*, *colors='k'*, *linestyles='solid'*, *label='‘*, ***kwargs*)

Plot horizontal lines at each `y` from `xmin` to `xmax`.

Parameters y : scalar or sequence of scalar

y-indexes where to plot the lines.

xmin, xmax : scalar or 1D array_like

Respective beginning and end of each line. If scalars are provided, all lines will have same length.

colors : array_like of colors, optional, default: ‘k’

linestyles : ['solid' | ‘dashed’ | ‘dashdot’ | ‘dotted’], optional

label : string, optional, default: ‘’

Returns lines : `LineCollection`

Other Parameters kwargs : `LineCollection` properties.

See also:

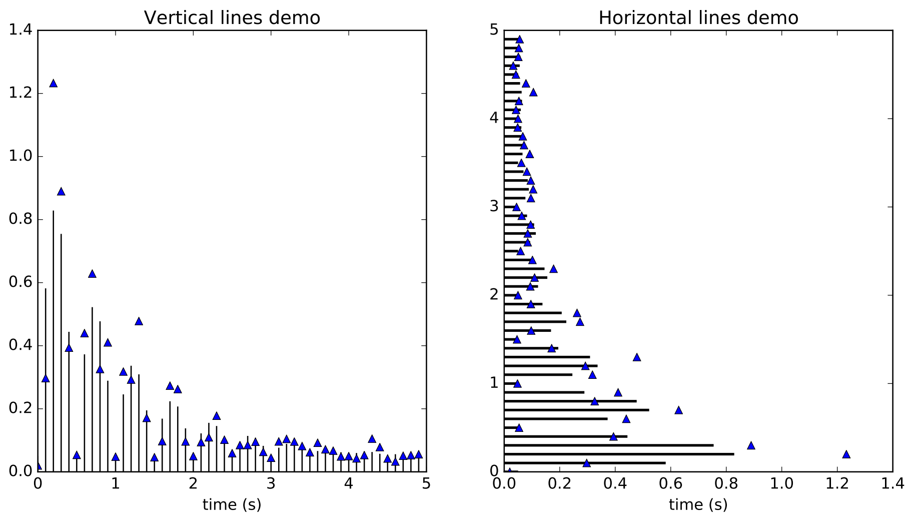
vlines vertical lines

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'xmin', 'xmax', 'y'.

Examples



hold(*b=None*)

Call signature:

```
hold(b=None)
```

Set the hold state. If *hold* is *None* (default), toggle the *hold* state. Else set the *hold* state to boolean value *b*.

Examples:

```
# toggle hold
hold()

# turn hold on
hold(True)

# turn hold off
hold(False)
```

When hold is *True*, subsequent plot commands will be added to the current axes. When hold is *False*, the current axes and figure will be cleared on the next plot command

imshow(*X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None, resample=None, url=None, **kwargs*)

Display an image on the axes.

Parameters X : array_like, shape (n, m) or (n, m, 3) or (n, m, 4)

Display the image in `X` to current axes. `X` may be a float array, a uint8 array or a PIL image. If `X` is an array, it can have the following shapes:

- MxN – luminance (grayscale, float array only)
- MxNx3 – RGB (float or uint8 array)
- MxNx4 – RGBA (float or uint8 array)

The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0; MxN float arrays may be normalised.

cmap : *Colormap*, optional, default: None

If None, default to rc `image.cmap` value. `cmap` is ignored when `X` has RGB(A) information

aspect : ['auto' | 'equal' | scalar], optional, default: None

If 'auto', changes the image aspect ratio to match that of the axes.

If 'equal', and `extent` is None, changes the axes aspect ratio to match that of the image. If `extent` is not `None`, the axes aspect ratio is changed to match that of the extent.

If None, default to rc `image.aspect` value.

interpolation : string, optional, default: None

Acceptable values are 'none', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos'

If `interpolation` is None, default to rc `image.interpolation`. See also the `filternorm` and `filterrad` parameters. If `interpolation` is 'none', then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to 'nearest'.

norm : *Normalize*, optional, default: None

A *Normalize* instance is used to scale luminance data to 0, 1. If `None`, use the default func:`normalize`. `norm` is only used if `X` is an array of floats.

vmin, vmax : scalar, optional, default: None

`vmin` and `vmax` are used in conjunction with norm to normalize luminance data. Note if you pass a `norm` instance, your settings for `vmin` and `vmax` will be ignored.

alpha : scalar, optional, default: None

The alpha blending value, between 0 (transparent) and 1 (opaque)

origin : ['upper' | 'lower'], optional, default: None

Place the [0,0] index of the array in the upper left or lower left corner of the axes. If None, default to rc `image.origin`.

extent : scalars (left, right, bottom, top), optional, default: None

The location, in data-coordinates, of the lower-left and upper-right corners. If `None`, the image is positioned such that the pixel centers fall on zero-based (row, column) indices.

shape : scalars (columns, rows), optional, default: None
For raw buffer images

filternorm : scalar, optional, default: 1
A parameter for the antigrain image resize filter. From the antigrain documentation, if `filternorm` = 1, the filter normalizes integer values and corrects the rounding errors. It doesn't do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

filterrad : scalar, optional, default: 4.0
The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: 'sinc', 'lanczos' or 'blackman'

Returns image : `AxesImage`

Other Parameters kwargs : `Artist` properties.

See also:

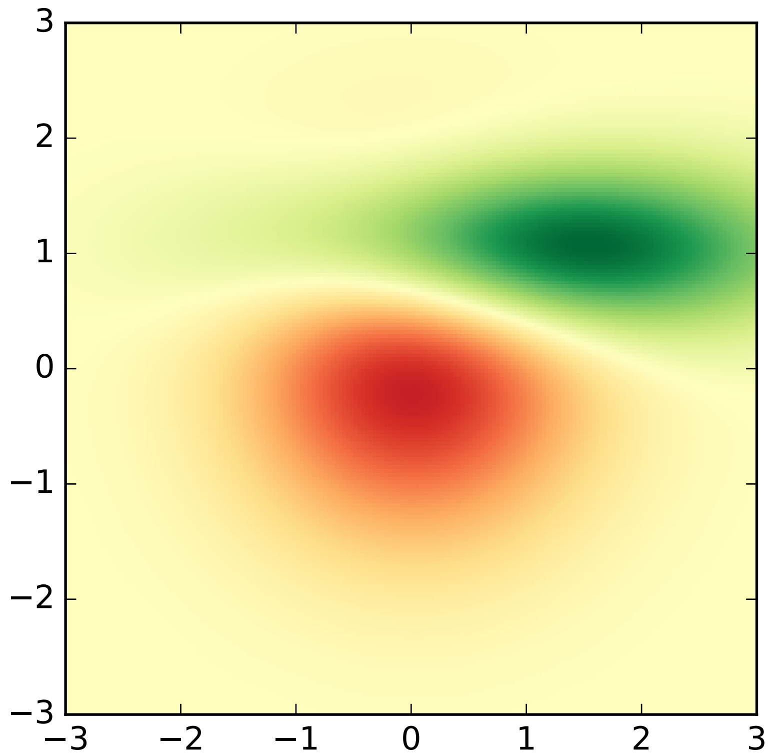
`matshow` Plot a matrix or an array as an image.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

Examples



in_axes(*mouseevent*)
Return *True* if the given *mouseevent* (in display coords) is in the Axes

invert_xaxis()
Invert the x-axis.

invert_yaxis()
Invert the y-axis.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

ishold()
return the HOLD status of the axes

legend(**args*, ***kwargs*)
Places a legend on the axes.

To make a legend for lines which already exist on the axes (via plot for instance), simply call this function with an iterable of strings, one for each legend item. For example:

```
ax.plot([1, 2, 3])
ax.legend(['A simple line'])
```

However, in order to keep the “label” and the legend element instance together, it is preferable to specify the label either at artist creation, or by calling the `set_label()` method on the artist:

```
line, = ax.plot([1, 2, 3], label='Inline label')
# Overwrite the label by calling the method.
line.set_label('Label via method')
ax.legend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling `legend()` without any arguments and without setting the labels manually will result in no legend being drawn.

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
legend((line1, line2, line3), ('label1', 'label2', 'label3'))
```

Parameters loc : int or string or pair of floats, default: ‘upper right’

The location of the legend. Possible codes are:

Location String	Location Code
‘best’	0
‘upper right’	1
‘upper left’	2
‘lower left’	3
‘lower right’	4
‘right’	5
‘center left’	6
‘center right’	7
‘lower center’	8
‘upper center’	9
‘center’	10

Alternatively can be a 2-tuple giving `x, y` of the lower-left corner of the legend in axes coordinates (in which case `bbox_to_anchor` will be ignored).

bbox_to_anchor : `matplotlib.transforms.BboxBase` instance or tuple of floats

Specify any arbitrary location for the legend in `bbox_transform` coordinates (default Axes coordinates).

For example, to put the legend’s upper right hand corner in the center of the axes the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

ncol : integer

The number of columns that the legend has. Default is 1.

prop : None or `matplotlib.font_manager.FontProperties` or dict

The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

fontsize : int or float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}
Controls the font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if `prop` is not specified.

numpoints : None or int
The number of marker points in the legend when creating a legend entry for a line/*matplotlib.lines.Line2D*. Default is `None` which will take the value from the `legend.numpoints` *rcParam*.

scatterpoints : None or int
The number of marker points in the legend when creating a legend entry for a scatter plot/ *matplotlib.collections.PathCollection*. Default is `None` which will take the value from the `legend.scatterpoints` *rcParam*.

scatteryoffsets : iterable of floats
The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`. Default `[0.375, 0.5, 0.3125]`.

markerscale : None or int or float
The relative size of legend markers compared with the originally drawn ones. Default is `None` which will take the value from the `legend.markerscale` *rcParam*.

***markerfirst*: [*True* | *False*]**
if *True*, legend marker is placed to the left of the legend label if *False*, legend marker is placed to the right of the legend label

frameon : None or bool
Control whether a frame should be drawn around the legend. Default is `None` which will take the value from the `legend.frameon` *rcParam*.

fancybox : None or bool
Control whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background. Default is `None` which will take the value from the `legend.fancybox` *rcParam*.

shadow : None or bool
Control whether to draw a shadow behind the legend. Default is `None` which will take the value from the `legend.shadow` *rcParam*.

framealpha : None or float
Control the alpha transparency of the legend's frame. Default is `None` which will take the value from the `legend.framealpha` *rcParam*.

mode : {"expand", None}
    If `mode` is set to `"expand"` the legend will be horizontally expanded to fill the axes area (or `bbox_to_anchor` if defines the legend's size).

bbox_transform : None or *matplotlib.transforms.Transform*
    The transform for the bounding box (`bbox_to_anchor`). For a value of `None` (default) the Axes' `transAxes` transform will be used.

title : str or None
    The legend's title. Default is no title (`None`).

borderpad : float or None
    The fractional whitespace inside the legend border. Measured in font-size units. Default is `None` which will take the value from the `legend.borderpad` *rcParam*.

labelspacing : float or None
    The vertical space between the legend entries. Measured in font-size units. Default is `None` which will take the value from the `legend.labelspacing` *rcParam*.

handlelength : float or None
    The length of the legend handles. Measured in font-size units. Default is `None` which will take the value from the `legend.handlelength` *rcParam*.

handletextpad : float or None
    The pad between the legend handle and text. Measured in font-size units. Default is `None` which will take the value from the `legend.handletextpad` *rcParam*.

borderaxespad : float or None
    The pad between the axes and legend border. Measured in font-size units. Default is `None` which will take the value from the `legend.borderaxespad` *rcParam*.

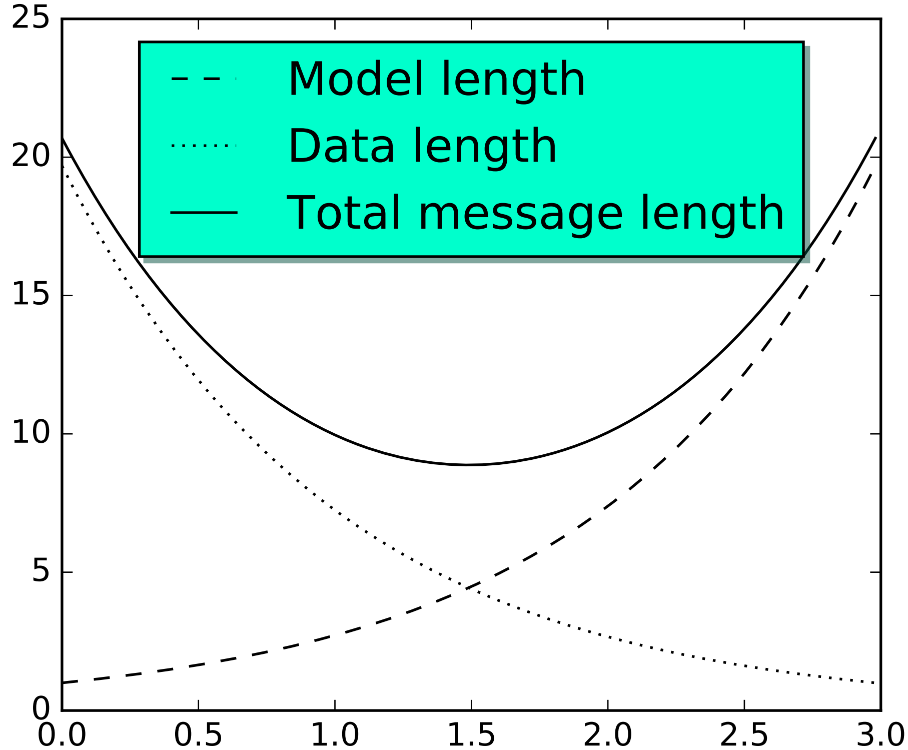
columnspacing : float or None
    The spacing between columns. Measured in font-size units. Default is `None` which will take the value from the `legend.columnspacing` *rcParam*.

handler_map : dict or None
    The custom dictionary mapping instances or types to a legend handler. This `handler_map` updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map()*.

Notes

Not all kinds of artist are supported by the legend command. See *Legend guide* for details.

Examples



locator_params(*axis='both'*, *tight=None*, ***kwargs*)

Control behavior of tick locators.

Keyword arguments:

axis ['x' | 'y' | 'both'] Axis on which to operate; default is 'both'.

tight [True | False | None] Parameter passed to `autoscale_view()`. Default is None, for no change.

Remaining keyword arguments are passed to directly to the `set_params()` method.

Typically one might want to reduce the maximum number of ticks and use tight bounds when plotting small subplots, for example:

```
ax.locator_params(tight=True, nbins=4)
```

Because the locator is involved in autoscaling, `autoscale_view()` is called automatically after the parameters are changed.

This presently works only for the `MaxNLocator` used by default on linear axes, but it may be generalized.

loglog(**args*, ***kwargs*)

Make a plot with log scaling on both the *x* and *y* axis.

Call signature:

```
loglog(*args, **kwargs)
```

loglog() supports all the keyword arguments of *plot()* and *matplotlib.axes.Axes.set_xscale()* / *matplotlib.axes.Axes.set_yscale()*.

Notable keyword arguments:

***basex/basey*: scalar > 1** Base of the *x*/*y* logarithm

***subsx/subsy*: [*None* | sequence]** The location of the minor *x*/*y* ticks; *None* defaults to autosubs, which depend on the number of decades in the plot; see *matplotlib.axes.Axes.set_xscale()* / *matplotlib.axes.Axes.set_yscale()* for details

***nonposx/nonposy*: ['mask' | 'clip']** Non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are *Line2D* properties:

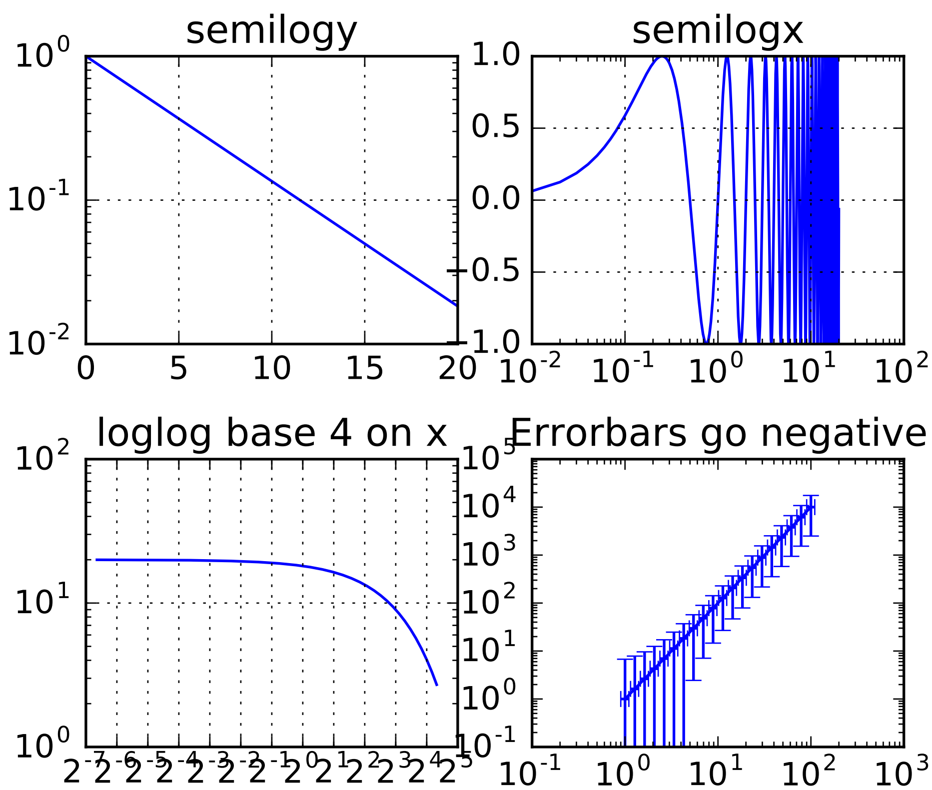
Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an *Axes* instance
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a *matplotlib.figure.Figure* instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
linewidth or lw	float value in points
marker	*A valid marker style*
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`

C

Table 43.17 – continued from previous page

Property	Description
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:



magnitude_spectrum(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *scale=None*, ***kwargs*)

Plot the magnitude spectrum.

Call signature:

```
magnitude_spectrum(x, Fs=2, Fc=0,  window=mlab.window_hanning,
                       pad_to=None, sides='default', **kwargs)
```

Compute the magnitude spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer**

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

***scale*: ['default' | 'linear' | 'dB']** The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale. When *mode* is 'density', this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). 'default' is 'linear'.

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns the tuple (*spectrum*, *freqs*, *line*):

***spectrum*: 1-D array** The values for the magnitude spectrum before scaling (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

***line*: a `Line2D` instance** The line created by this function

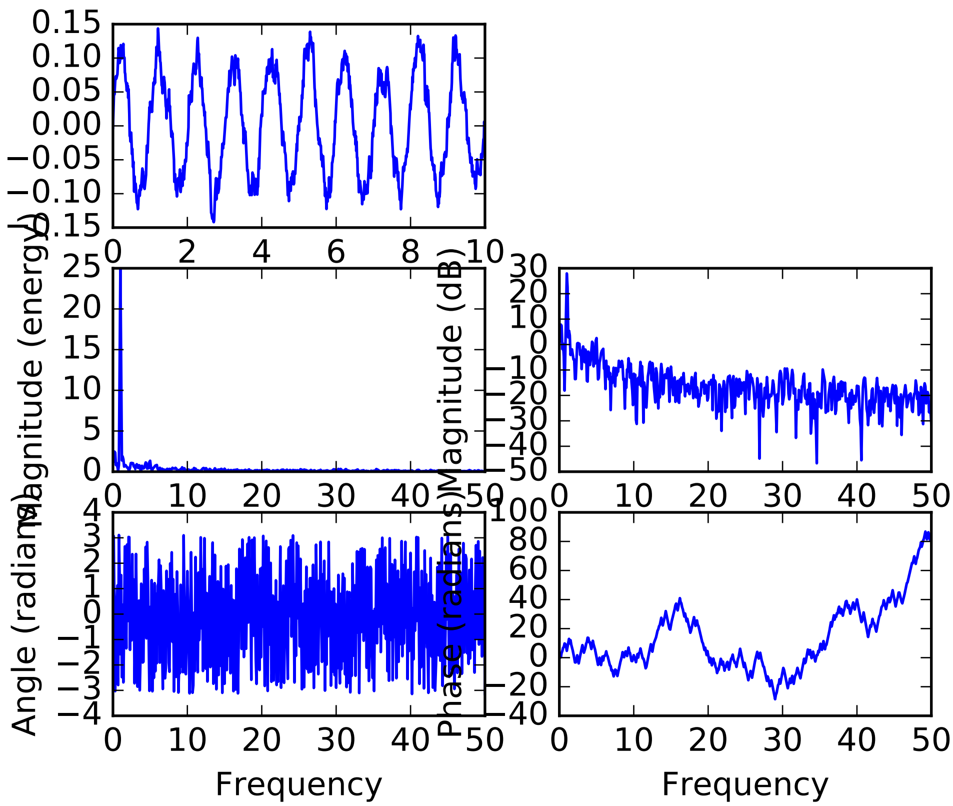
kwargs control the `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
C	

Table 43.18 – continued from previous page

Property	Description
antialiased or aa	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '
linewidth or lw	float value in points
marker	A valid marker style
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:



See also:

psd() psd() plots the power spectral density.‘.

angle_spectrum() angle_spectrum() plots the angles of the corresponding frequencies.

phase_spectrum() phase_spectrum() plots the phase (unwrapped angle) of the corresponding frequencies.

specgram() specgram() can plot the magnitude spectrum of segments within the signal in a colormap.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: ‘x’.

margins(**args*, ***kw*)

Set or retrieve autoscaling margins.

signatures:

```
margins()
```

returns xmargin, ymargin

```
margins(margin)

margins(xmargin, ymargin)

margins(x=xmargin, y=ymargin)

margins(..., tight=False)
```

All three forms above set the xmargin and ymargin parameters. All keyword parameters are optional. A single argument specifies both xmargin and ymargin. The *tight* parameter is passed to `autoscale_view()`, which is executed after a margin is changed; the default here is *True*, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting *tight* to *None* will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if *xmargin* is not None, then *xmargin* times the X data interval will be added to each end of that interval before it is used in autoscaling.

matshow(*Z*, ***kwargs*)

Plot a matrix or array as an image.

The matrix will be shown the way it would be printed, with the first row at the top. Row and column numbering is zero-based.

Parameters Z : array_like shape (n, m)

The matrix to be displayed.

Returns image : `AxesImage`

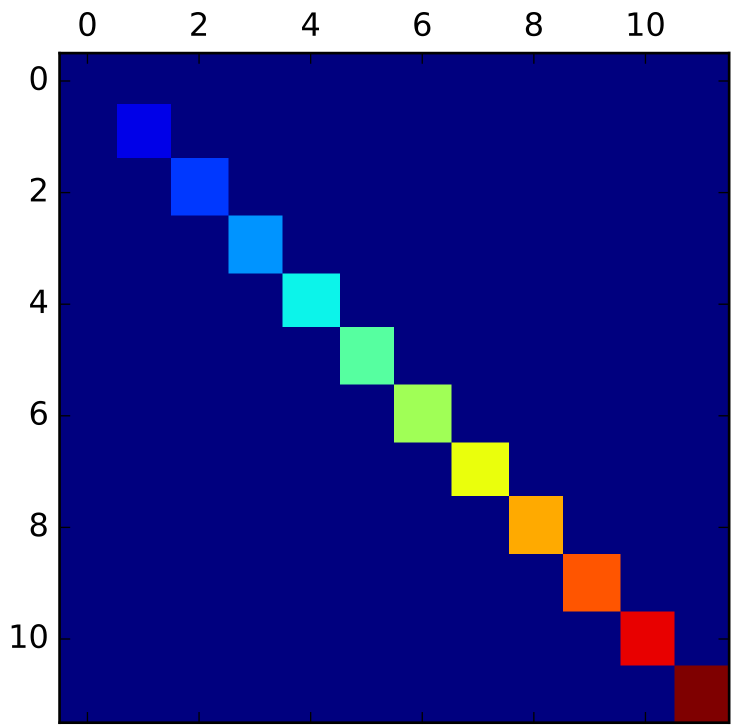
Other Parameters kwargs : `imshow` arguments

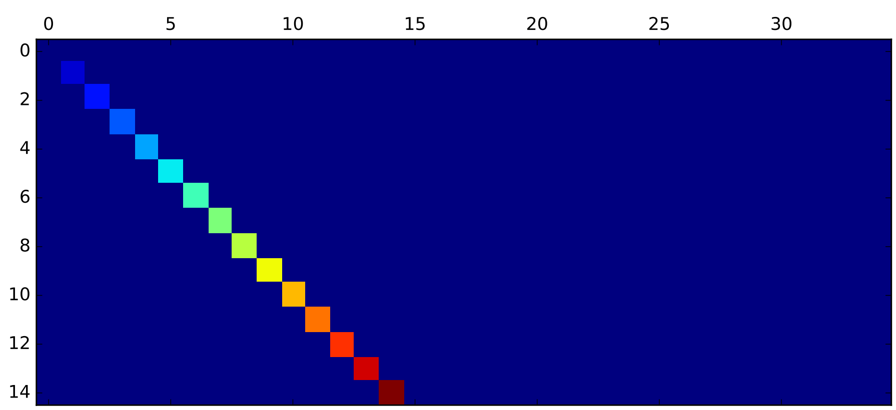
Sets `origin` to 'upper', 'interpolation' to 'nearest' and 'aspect' to equal.

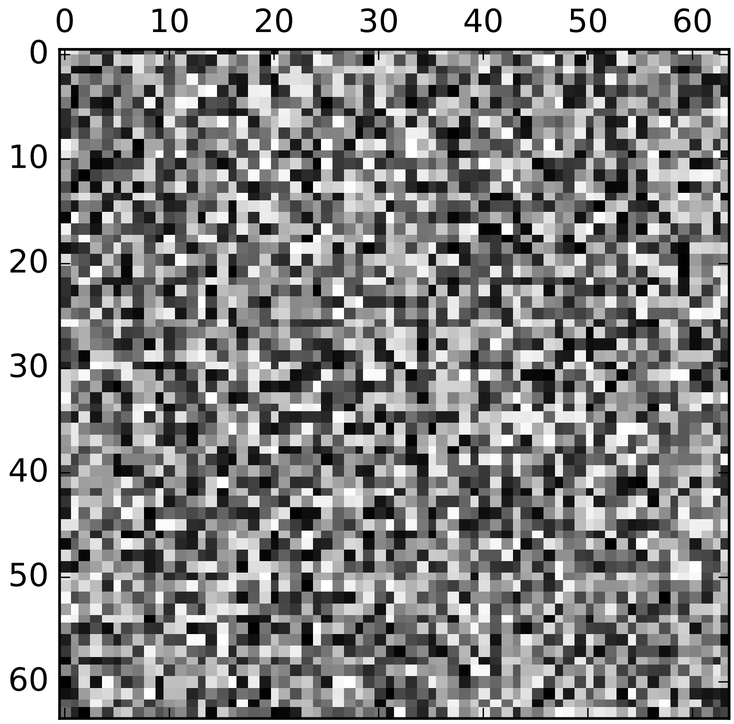
See also:

imshow plot an image

Examples







minorticks_off()
Remove minor ticks from the axes.

minorticks_on()
Add autoscaling minor ticks to the axes.

mouseover

name = 'rectilinear'

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pcolor(**args*, ***kwargs*)
Create a pseudocolor plot of a 2-D array.

Note: pcolor can be very slow for large arrays; consider using the similar but much faster *pcolormesh()* instead.

Call signatures:

```
pcolor(C, **kwargs)
pcolor(X, Y, C, **kwargs)
```

C is the array of color values.

X and *Y*, if given, specify the (*x*, *y*) coordinates of the colored quadrilaterals; the quadrilateral for C[i,j] has corners at:

```
(X[i,   j],   Y[i,   j]),
(X[i,   j+1], Y[i,   j+1]),
(X[i+1, j],   Y[i+1, j]),
(X[i+1, j+1], Y[i+1, j+1]).
```

Ideally the dimensions of *X* and *Y* should be one greater than those of *C*; if the dimensions are the same, then the last row and column of *C* will be ignored.

Note that the column index corresponds to the *x*-coordinate, and the row index corresponds to *y*; for details, see the *Grid Orientation* section below.

If either or both of *X* and *Y* are 1-D arrays or column vectors, they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

X, *Y* and *C* may be masked arrays. If either C[i, j], or one of the vertices surrounding C[i,j] (*X* or *Y* at [i, j], [i+1, j], [i, j+1],[i+1, j+1]) is masked, nothing is plotted.

Keyword arguments:

***cmap*: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance. If *None*, use rc settings.

***norm*: [*None* | Normalize]** An `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If *None*, defaults to `normalize()`.

***vmin*/*vmax*: [*None* | scalar]** *vmin* and *vmax* are used in conjunction with *norm* to normalize luminance data. If either is *None*, it is autoscaled to the respective min or max of the color array *C*. If not *None*, *vmin* or *vmax* passed in here override any pre-existing values supplied in the *norm* instance.

***shading*: ['flat' | 'faceted']** If 'faceted', a black grid is drawn around each rectangle; if 'flat', edges are not drawn. Default is 'flat', contrary to MATLAB.

This kwarg is deprecated; please use 'edgecolors' instead:

- shading='flat' – edgecolors='none'
- shading='faceted – edgecolors='k'

***edgecolors*: [*None* | `'none'` | color | color sequence]** If *None*, the rc setting is used by default.

If `'none'`, edges will not be visible.

An mpl color or sequence of colors will set the edge color

alpha*: `0 <= scalar <= 1` or *None the alpha blending value

***snap*: bool** Whether to snap the mesh to pixel boundaries.

Return value is a `matplotlib.collections.Collection` instance. The grid orientation follows the MATLAB convention: an array *C* with shape (*nrows*, *ncolumns*) is plotted with the column number as *X* and the row number as *Y*, increasing up; hence it is plotted the way the array would be printed, except that the *Y* axis is reversed. That is, *C* is taken as *C*(*y*, *x*).

Similarly for `meshgrid()`:

```
x = np.arange(5)
y = np.arange(3)
X, Y = np.meshgrid(x, y)
```

is equivalent to:

```
X = array([[0, 1, 2, 3, 4],
           [0, 1, 2, 3, 4],
           [0, 1, 2, 3, 4]])

Y = array([[0, 0, 0, 0, 0],
           [1, 1, 1, 1, 1],
           [2, 2, 2, 2, 2]])
```

so if you have:

```
C = rand(len(x), len(y))
```

then you need to transpose C:

```
pcolor(X, Y, C.T)
```

or:

```
pcolor(C.T)
```

MATLAB `pcolor()` always discards the last row and column of *C*, but matplotlib displays the last row and column if *X* and *Y* are not specified, or if *X* and *Y* have one more row and column than *C*.

kwargs can be used to control the *PolyCollection* properties:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an *Axes* instance
clim	a length 2 sequence of floats
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
contains	a callable function
edgecolor or edgecolors	matplotlib color spec or sequence of specs
facecolor or facecolors	matplotlib color spec or sequence of specs
figure	a *matplotlib.figure.Figure* instance

Table 43.19 – continued from previous page

Property	Description
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' |
linewidth or lw or linewidths	float or sequence of floats
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
path_effects	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

Note: The default *antialiaseds* is False if the default *edgecolors*="none" is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If *edgecolors* is not "none", then the default *antialiaseds* is taken from rcParams['patch.antialiased'], which defaults to *True*. Stroking the edges may be preferred if *alpha* is 1, but will cause artifacts otherwise.

See also:

pcolormesh() For an explanation of the differences between pcolor and pcolormesh.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

pcolorfast(**args*, ***kwargs*)

pseudocolor plot of a 2-D array

Experimental; this is a pcolor-type method that provides the fastest possible rendering with the Agg backend, and that can handle any quadrilateral grid. It supports only flat shading (no outlines), it lacks support for log scaling of the axes, and it does not have a pyplot wrapper.

Call signatures:

```
ax.pcolorfast(C, **kwargs)
ax.pcolorfast(xr, yr, C, **kwargs)
ax.pcolorfast(x, y, C, **kwargs)
ax.pcolorfast(X, Y, C, **kwargs)
```

C is the 2D array of color values corresponding to quadrilateral cells. Let (nr, nc) be its shape. C may be a masked array.

`ax.pcolorfast(C, **kwargs)` is equivalent to `ax.pcolorfast([0,nc], [0,nr], C, **kwargs)`

xr, *yr* specify the ranges of *x* and *y* corresponding to the rectangular region bounding *C*. If:

```
xr = [x0, x1]
```

and:

```
yr = [y0,y1]
```

then *x* goes from *x0* to *x1* as the second index of *C* goes from 0 to *nc*, etc. (*x0*, *y0*) is the outermost corner of cell (0,0), and (*x1*, *y1*) is the outermost corner of cell (*nr*-1, *nc*-1). All cells are rectangles of the same size. This is the fastest version.

x, *y* are 1D arrays of length *nc* +1 and *nr* +1, respectively, giving the x and y boundaries of the cells. Hence the cells are rectangular but the grid may be nonuniform. The speed is intermediate. (The grid is checked, and if found to be uniform the fast version is used.)

X and *Y* are 2D arrays with shape (*nr* +1, *nc* +1) that specify the (x,y) coordinates of the corners of the colored quadrilaterals; the quadrilateral for C[i,j] has corners at (X[i,j],Y[i,j]), (X[i,j+1],Y[i,j+1]), (X[i+1,j],Y[i+1,j]), (X[i+1,j+1],Y[i+1,j+1]). The cells need not be rectangular. This is the most general, but the slowest to render. It may produce faster and more compact output using ps, pdf, and svg backends, however.

Note that the column index corresponds to the x-coordinate, and the row index corresponds to y; for details, see the "Grid Orientation" section below.

Optional keyword arguments:

***cmap*: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance from cm. If *None*, use rc settings.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If *None*, defaults to normalize()

***vmin*/*vmax*: [*None* | scalar]** *vmin* and *vmax* are used in conjunction with norm to normalize luminance data. If either are *None*, the min and max of the color array *C* is used. If you pass a norm instance, *vmin* and *vmax* will be *None*.

alpha*: `0 <= scalar <= 1` or *None the alpha blending value

Return value is an image if a regular or rectangular grid is specified, and a `QuadMesh` collection in the general quadrilateral case.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

pcolormesh(**args*, ***kwargs*)

Plot a quadrilateral mesh.

Call signatures:

```
pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

pcolormesh is similar to `pcolor()`, but uses a different mechanism and returns a different object; pcolor returns a `PolyCollection` but pcolormesh returns a `QuadMesh`. It is much faster, so it is almost always preferred for large arrays.

C may be a masked array, but *X* and *Y* may not. Masked array support is implemented via *cmap* and *norm*; in contrast, `pcolor()` simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

***cmap*: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance. If *None*, use rc settings.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If *None*, defaults to `normalize()`.

***vmin*/*vmax*: [*None* | scalar]** *vmin* and *vmax* are used in conjunction with *norm* to normalize luminance data. If either is *None*, it is autoscaled to the respective min or max of the color array *C*. If not *None*, *vmin* or *vmax* passed in here override any pre-existing values supplied in the *norm* instance.

***shading*: ['flat' | 'gouraud']** 'flat' indicates a solid color for each quad. When 'gouraud', each quad will be Gouraud shaded. When gouraud shading, edgecolors is ignored.

***edgecolors*: [*None* | `'None'` | `'face'` | color | color sequence]**

If *None*, the rc setting is used by default.

If `'None'`, edges will not be visible.

If `'face'`, edges will have the same color as the faces.

An mpl color or sequence of colors will set the edge color

alpha*: `0 <= scalar <= 1` or *None the alpha blending value

Return value is a `matplotlib.collections.QuadMesh` object.

kwargs can be used to control the `matplotlib.collections.QuadMesh` properties:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or antialiaseds	Boolean or sequence of booleans
array	unknown
axes	an Axes instance
clim	a length 2 sequence of floats
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
cmap	a colormap or registered colormap name
color	matplotlib color arg or sequence of rgba tuples
contains	a callable function
edgecolor or edgecolors	matplotlib color spec or sequence of specs
facecolor or facecolors	matplotlib color spec or sequence of specs
figure	a matplotlib.figure.Figure instance
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
label	string or anything printable with '%s' conversion.
linestyle or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' |
linewidth or lw or linewidths	float or sequence of floats
norm	unknown
offset_position	unknown
offsets	float or sequence of floats
path_effects	unknown
picker	[None|float|boolean|callable]
pickradius	unknown
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	Transform instance
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

See also:

pcolor() For an explanation of the grid orientation and the expansion of 1-D *X* and/or *Y* to 2-D arrays.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

phase_spectrum(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, ***kwargs*)

Plot the phase spectrum.

Call signature:

```
phase_spectrum(x, Fs=2, Fc=0,  window=mlab.window_hanning,
               pad_to=None, sides='default', **kwargs)
```

Compute the phase spectrum (unwrapped angle spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer**

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns the tuple (*spectrum*, *freqs*, *line*):

***spectrum*: 1-D array** The values for the phase spectrum in radians (real valued)

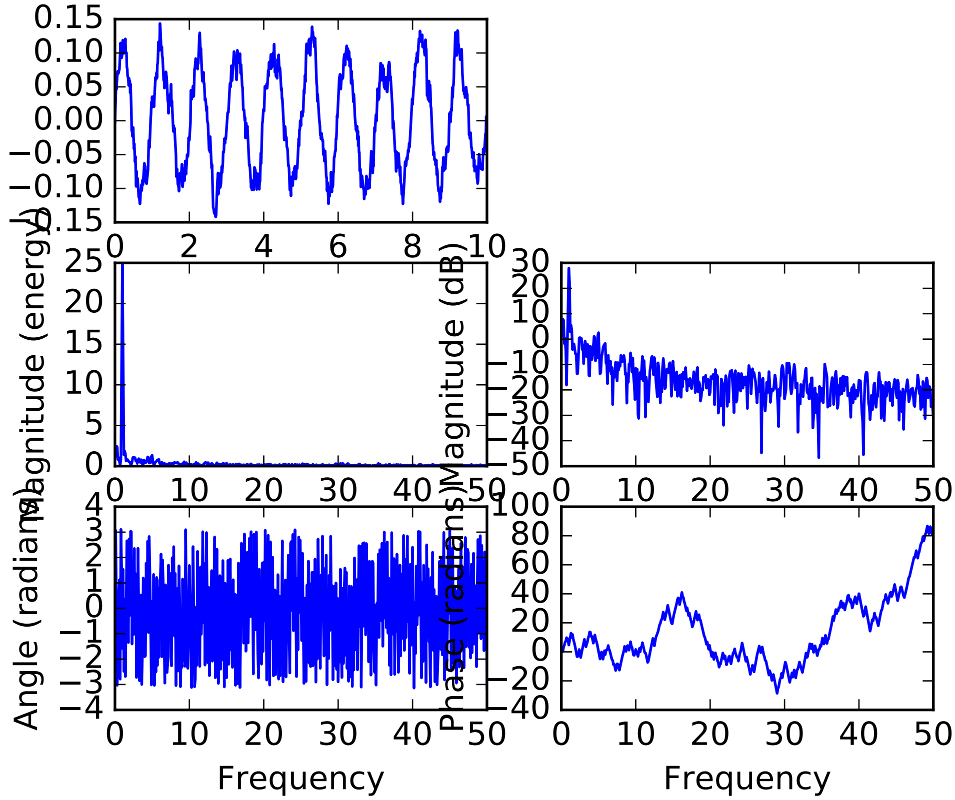
***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

***line*: a `Line2D` instance** The line created by this function

kwargs control the `Line2D` properties:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '
linewidth or lw	float value in points
marker	A valid marker style
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:



See also:

magnitude_spectrum() `magnitude_spectrum()` plots the magnitudes of the corresponding frequencies.

angle_spectrum() `angle_spectrum()` plots the wrapped version of this function.

specgram() `specgram()` can plot the phase spectrum of segments within the signal in a colormap.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

pick(**args*)

Call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if mouseevent is over the artist and the artist has picker set

pickable()

Return *True* if `Artist` is pickable.

pie(*x*, *explode=None*, *labels=None*, *colors=None*, *autopct=None*, *pctdistance=0.6*, *shadow=False*, *labeldistance=1.1*, *startangle=None*, *radius=None*, *counterclock=True*, *wedgeprops=None*, *textprops=None*, *center=(0, 0)*, *frame=False*)

Plot a pie chart.

Call signature:

```
pie(x, explode=None, labels=None,
    colors=('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'),
    autopct=None, pctdistance=0.6, shadow=False,
    labeldistance=1.1, startangle=None, radius=None,
    counterclock=True, wedgeprops=None, textprops=None,
    center = (0, 0), frame = False )
```

Make a pie chart of array *x*. The fractional area of each wedge is given by x/sum(x). If sum(x) <= 1, then the values of x give the fractional area directly and the array will not be normalized. The wedges are plotted counterclockwise, by default starting from the x-axis.

Keyword arguments:

***explode*: [*None* | len(x) sequence]** If not *None*, is a `len(x)` array which specifies the fraction of the radius with which to offset each wedge.

***colors*: [*None* | color sequence]** A sequence of matplotlib color args through which the pie chart will cycle.

***labels*: [*None* | len(x) sequence of strings]** A sequence of strings providing the labels for each wedge

***autopct*: [*None* | format string | format function]** If not *None*, is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be `fmt%pct`. If it is a function, it will be called.

***pctdistance*: scalar** The ratio between the center of each pie slice and the start of the text generated by *autopct*. Ignored if *autopct* is *None*; default is 0.6.

***labeldistance*: scalar** The radial distance at which the pie labels are drawn

***shadow*: [*False* | *True*]** Draw a shadow beneath the pie.

***startangle*: [*None* | Offset angle]** If not *None*, rotates the start of the pie chart by *angle* degrees counterclockwise from the x-axis.

radius: [*None* | scalar] The radius of the pie, if *radius* is *None* it will be set to 1.

***counterclock*: [*False* | *True*]** Specify fractions direction, clockwise or counterclockwise.

***wedgeprops*: [*None* | dict of key value pairs]** Dict of arguments passed to the wedge objects making the pie. For example, you can pass in wedgeprops = { 'linewidth' : 3 } to set the width of the wedge border lines equal to 3. For more details, look at the doc/arguments of the wedge object. By default `clip_on=False`.

***textprops*: [*None* | dict of key value pairs]** Dict of arguments to pass to the text objects.

center: [(0,0) | sequence of 2 scalars] Center position of the chart.

***frame*: [*False* | *True*]** Plot axes frame with the chart.

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal. e.g.:

```
figure(figsize=(8,8))
ax = axes([0.1, 0.1, 0.8, 0.8])
```

or:

```
axes(aspect=1)
```

Return value: If *autopct* is *None*, return the tuple (*patches*, *texts*):

- *patches* is a sequence of `matplotlib.patches.Wedge` instances
- *texts* is a list of the label `matplotlib.text.Text` instances.

If *autopct* is not *None*, return the tuple (*patches*, *texts*, *autotexts*), where *patches* and *texts* are as above, and *autotexts* is a list of `Text` instances for the numeric labels.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'colors', 'labels', 'x', 'explode'.

plot(**args*, ***kwargs*)

Plot lines and/or markers to the `Axes`. *args* is a variable length argument, allowing for multiple *x*, *y* pairs with an optional format string. For example, each of the following is legal:

```
plot(x, y)        # plot x and y using default line style and color
plot(x, y, 'bo')  # plot x and y using blue circle markers
plot(y)           # plot y using x as index array 0..N-1
plot(y, 'r+')     # ditto, but with red plusses
```

If *x* and/or *y* is 2-dimensional, then the corresponding columns will be plotted.

If used with labeled data, make sure that the color spec is not included as an element in data, as otherwise the last case `plot("v","r", data={"v":..., "r":...)` can be interpreted as the first case which would do `plot(v, r)` using the default line style and color.

If not used with labeled data (i.e., without a data argument), an arbitrary number of *x*, *y*, *fmt* groups can be specified, as in:

```
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

Return value is a list of lines that were added.

By default, each line is assigned a different style specified by a 'style cycle'. To change this behavior, you can edit the axes.prop_cycle rcParam.

The following format string characters are accepted to control the line style or marker:

character	description
`'-'`	solid line style
`'--'`	dashed line style
`'-.'`	dash-dot line style
`':'`	dotted line style
`'.'`	point marker
`','`	pixel marker
`'o'`	circle marker
`'v'`	triangle_down marker
`'^'`	triangle_up marker
`'<'`	triangle_left marker
`'>'`	triangle_right marker
`'1'`	tri_down marker
`'2'`	tri_up marker
`'3'`	tri_left marker
`'4'`	tri_right marker
`'s'`	square marker
`'p'`	pentagon marker
`'*'`	star marker
`'h'`	hexagon1 marker
`'H'`	hexagon2 marker
`'+'`	plus marker
`'x'`	x marker
`'D'`	diamond marker
`'d'`	thin_diamond marker
`'|'`	vline marker
`'_'`	hline marker

The following color abbreviations are supported:

character	color
‘b’	blue
‘g’	green
‘r’	red
‘c’	cyan
‘m’	magenta
‘y’	yellow
‘k’	black
‘w’	white

In addition, you can specify colors in many weird and wonderful ways, including full names (`'green'`), hex strings (`'#008000'`), RGB or RGBA tuples (`(0,1,0,1)`) or grayscale intensities as a string (`'0.8'`). Of these, the string specifications can be used in place of a `fmt` group, but the tuple forms can be used only as `kwargs`.

Line styles and colors are combined in a single format string, as in `'bo'` for blue circles.

The *kwargs* can be used to set line properties (any property that has a `set_*` method). You can use this to set a line label (for auto legends), linewidth, anitialising, marker face color, etc. Here is an example:

```
plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plot([1,2,3], [1,4,9], 'rs',  label='line 2')
axis([0, 4, 0, 10])
legend()
```

If you make multiple lines with one plot command, the kwargs apply to all those lines, e.g.:

```
plot(x1, y1, x2, y2, antialiased=False)
```

Neither line will be antialiased.

You do not need to use format strings, which are just abbreviations. All of the line properties can be controlled by keyword arguments. For example, you can set the color, marker, linestyle, and markercolor with:

```
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See *Line2D* for details.

The kwargs are *Line2D* properties:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an *Axes* instance
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a *matplotlib.figure.Figure* instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
linewidth or lw	float value in points
marker	*A valid marker style*
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
	C

Table 43.22 – continued from previous page

Property	Description
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
`path_effects`	unknown
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
`ydata`	1D array
`zorder`	any number

kwargs *scalex* and *scaley*, if defined, are passed on to `autoscale_view()` to determine whether the *x* and *y* axes are autoscaled; the default is *True*.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

plot_date(*x*, *y*, *fmt='o'*, *tz=None*, *xdate=True*, *ydate=False*, ***kwargs*)

Plot with data with dates.

Call signature:

```
plot_date(x, y, fmt='bo', tz=None, xdate=True,
          ydate=False, **kwargs)
```

Similar to the `plot()` command, except the *x* or *y* (or both) data is considered to be dates, and the axis is labeled accordingly.

x and/or *y* can be a sequence of dates represented as float days since 0001-01-01 UTC.

Keyword arguments:

***fmt*: string** The plot format string.

***tz*: [*None* | timezone string | `tzinfo` instance]** The time zone to use in labeling dates. If *None*, defaults to rc value.

***xdate*: [*True* | *False*]** If *True*, the *x*-axis will be labeled with dates.

***ydate*: [*False* | *True*]** If *True*, the *y*-axis will be labeled with dates.

Note if you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to `plot_date()` since `plot_date()` will set the default tick locator to `matplotlib.dates.AutoDateLocator` (if the tick locator is not already set to a `matplotlib.dates.DateLocator` instance) and the default tick formatter to `matplotlib.dates.AutoDateFormatter` (if the tick formatter is not already set to a `matplotlib.dates.DateFormatter` instance).

Valid kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
`path_effects`	unknown
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance

C

Table 43.23 – continued from previous page

Property	Description
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
`ydata`	1D array
`zorder`	any number

See also:

`dates` for helper functions

`date2num()`, `num2date()` and `drange()` for help on creating the required floating point dates.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

properties()

return a dictionary mapping property name -> value for all Artist props

psd(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, ***kwargs*)

Plot the power spectral density.

Call signature:

```
psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None, return_line=None, **kwargs)
```

The power spectral density P_{xx} by Welch's average periodogram method. The vector *x* is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The $|\mathrm{fft}(i)|^2$ of each segment i are averaged to compute P_{xx}, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.

x: **1-D array or sequence** Array or sequence containing the data

Keyword arguments:

Fs: **scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

window: **callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or** callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***noverlap*: integer** The number of points of overlap between segments. The default value is 0 (no overlap).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

***return_line*: bool** Whether to include the line object plotted in the returned values. Default is False.

If *return_line* is False, returns the tuple (*Pxx*, *freqs*). If *return_line* is True, returns the tuple (*Pxx*, *freqs*. *line*):

***Pxx*: 1-D array** The values for the power spectrum `P_{xx}` before scaling (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *Pxx*

***line*: a `Line2D` instance** The line created by this function. Only returend if *return_line* is True.

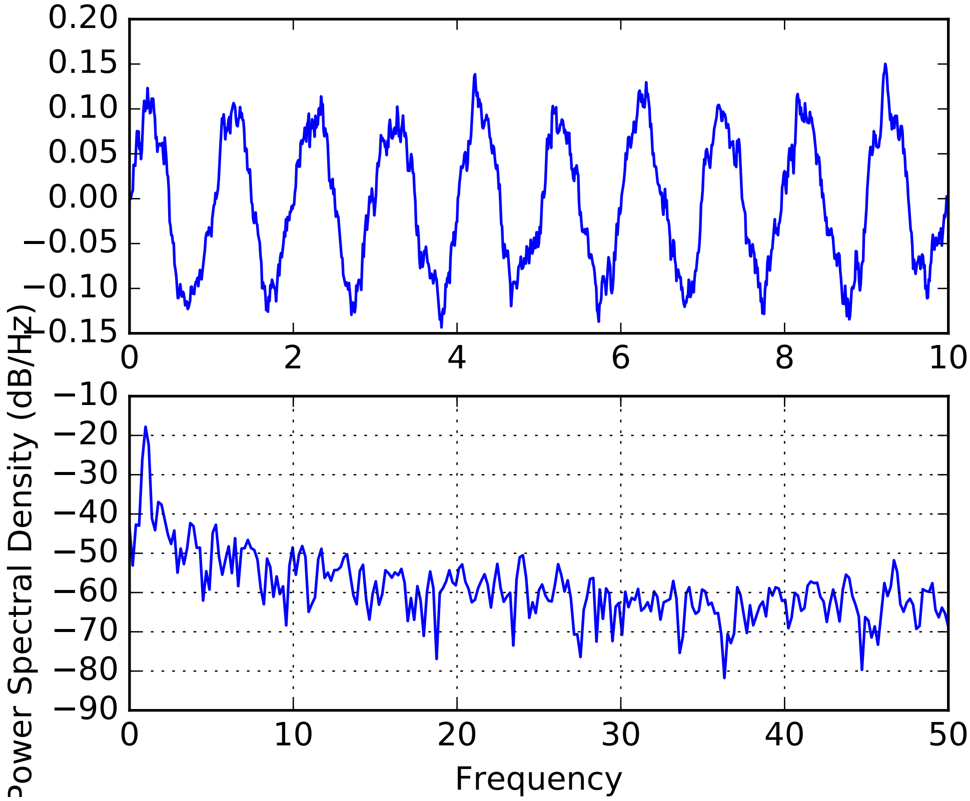
For plotting, the power is plotted as $10\log_{10}(P_{xx})$ for decibels, though *Pxx* itself is returned.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the `Line2D` properties:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '
linewidth or lw	float value in points
marker	A valid marker style
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:



See also:

specgram() specgram() differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

magnitude_spectrum() magnitude_spectrum() plots the magnitude spectrum.

csd() csd() plots the spectral density between two signals.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

quiver(**args*, ***kw*)

Plot a 2-D field of arrows.

call signatures:

```
quiver(U, V, **kw)
quiver(U, V, C, **kw)
quiver(X, Y, U, V, **kw)
quiver(X, Y, U, V, C, **kw)
```

Arguments:

***X*, *Y*:** The x and y coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg)

***U*, *V*:** Give the x and y components of the arrow vectors

***C*:** An optional array used to map colors to the arrows

All arguments may be 1-D or 2-D arrays or sequences. If *X* and *Y* are absent, they will be generated as a uniform grid. If *U* and *V* are 2-D arrays but *X* and *Y* are 1-D, and if `len(X)` and `len(Y)` match the column and row dimensions of *U*, then *X* and *Y* will be expanded with `numpy.meshgrid()`.

U, *V*, *C* may be masked arrays, but masked *X*, *Y* are not supported at present.

Keyword arguments:

***units*: ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'xy']** Arrow units; the arrow dimensions *except for length* are in multiples of this unit.

- 'width' or 'height': the width or height of the axes
- 'dots' or 'inches': pixels or inches, based on the figure dpi
- 'x', 'y', or 'xy': *X*, *Y*, or sqrt(X^2+Y^2) data units

The arrows scale differently depending on the units. For 'x' or 'y', the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For 'width or 'height', the arrow size increases with the width and height of the axes, respectively, when the window is resized; for 'dots' or 'inches', resizing does not change the arrows.

***angles*: ['uv' | 'xy' | array]** With the default 'uv', the arrow axis aspect ratio is 1, so that if *U*==*V* the orientation of the arrow on the plot is 45 degrees CCW from the horizontal axis (positive to the right). With 'xy', the arrow points from (x,y) to (x+u, y+v). Use this for plotting a gradient field, for example. Alternatively, arbitrary angles may be specified as an array of values in degrees, CCW from the horizontal axis. Note: inverting a data axis will correspondingly invert the arrows *only* with `angles='xy'`.

***scale*: [*None* | float]** Data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. If *None*, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the *scale_units* parameter

***scale_units*: *None*, or any of the *units* options.** For example, if *scale_units* is 'inches', *scale* is 2.0, and `(u,v) = (1,0)`, then the vector will be 0.5 inches long. If *scale_units* is 'width', then the vector will be half the width of the axes.

If *scale_units* is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use "angles='xy', scale_units='xy', scale=1".

***width*:** Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

***headwidth*: scalar** Head width as multiple of shaft width, default is 3

***headlength*: scalar** Head length as multiple of shaft width, default is 5

***headaxislength*: scalar** Head length at shaft intersection, default is 4.5

***minshaft*: scalar** Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible! Default is 1

***minlength*: scalar** Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead. Default is 1.

***pivot*: ['tail' | 'mid' | 'middle' | 'tip']** The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

***color*: [color | color sequence]** This is a synonym for the `PolyCollection` facecolor kwarg. If *C* has been set, *color* has no effect.

The defaults give a slightly swept-back arrow; to make the head a triangle, make *headaxislength* the same as *headlength*. To make the arrow more pointed, reduce *headwidth* or increase *headlength* and *headaxislength*. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave minshaft alone.

linewidths and edgecolors can be used to customize the arrow outlines. Additional `PolyCollection` keyword arguments:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td><code>agg_filter</code></td><td>unknown</td></tr>
<tr><td><code>alpha</code></td><td>float or None</td></tr>
<tr><td><code>animated</code></td><td>[True | False]</td></tr>
<tr><td><code>antialiased</code> or antialiaseds</td><td>Boolean or sequence of booleans</td></tr>
<tr><td><code>array</code></td><td>unknown</td></tr>
<tr><td><code>axes</code></td><td>an <code>Axes</code> instance</td></tr>
<tr><td><code>clim</code></td><td>a length 2 sequence of floats</td></tr>
<tr><td><code>clip_box</code></td><td>a <code>matplotlib.transforms.Bbox</code> instance</td></tr>
<tr><td><code>clip_on</code></td><td>[True | False]</td></tr>
<tr><td><code>clip_path</code></td><td>[(<code>Path</code>, <code>Transform</code>) | <code>Patch</code> | None]</td></tr>
<tr><td><code>cmap</code></td><td>a colormap or registered colormap name</td></tr>
<tr><td><code>color</code></td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td><code>contains</code></td><td>a callable function</td></tr>
<tr><td><code>edgecolor</code> or edgecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>facecolor</code> or facecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>figure</code></td><td>a <code>matplotlib.figure.Figure</code> instance</td></tr>
<tr><td><code>gid</code></td><td>an id string</td></tr>
<tr><td><code>hatch</code></td><td>['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']</td></tr>
<tr><td><code>label</code></td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td><code>linestyle</code> or linestyles or dashes</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> |</td></tr>
<tr><td><code>linewidth</code> or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td><code>norm</code></td><td>unknown</td></tr>
<tr><td><code>offset_position</code></td><td>unknown</td></tr>
<tr><td><code>offsets</code></td><td>float or sequence of floats</td></tr>
<tr><td><code>path_effects</code></td><td>unknown</td></tr>
<tr><td><code>picker</code></td><td>[None|float|boolean|callable]</td></tr>
<tr><td><code>pickradius</code></td><td>unknown</td></tr>
<tr><td><code>rasterized</code></td><td>[True | False | None]</td></tr>
<tr><td><code>sketch_params</code></td><td>unknown</td></tr>
<tr><td><code>snap</code></td><td>unknown</td></tr>
<tr><td><code>transform</code></td><td><code>Transform</code> instance</td></tr>
</table>

Table 43.25 – continued from previous page

Property	Description
url	a url string
urls	unknown
visible	[True | False]
zorder	any number

quiverkey(**args*, ***kw*)

Add a key to a quiver plot.

Call signature:

```
quiverkey(Q, X, Y, U, label, **kw)
```

Arguments:

Q: The Quiver instance returned by a call to quiver.

X*, *Y: The location of the key; additional explanation follows.

U: The length of the key

label: A string with the length and units of the key

Keyword arguments:

***coordinates* = ['axes' | 'figure' | 'data' | 'inches']** Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with 0,0 in the lower left and 1,1 in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with 0,0 at the lower left corner.

color: overrides face and edge colors from *Q*.

***labelpos* = ['N' | 'S' | 'E' | 'W']** Position the label above, below, to the right, to the left of the arrow, respectively.

labelsep: Distance in inches between the arrow and the label. Default is 0.1

labelcolor: defaults to default `Text` color.

fontproperties: A dictionary with keyword arguments accepted by the `FontProperties` initializer: *family*, *style*, *variant*, *size*, *weight*

Any additional keyword arguments are used to override vector properties taken from *Q*.

The positioning of the key depends on *X*, *Y*, *coordinates*, and *labelpos*. If *labelpos* is 'N' or 'S', *X*, *Y* give the position of the middle of the key arrow. If *labelpos* is 'E', *X*, *Y* positions the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in either of these two cases, *X*, *Y* is somewhere in the middle of the arrow+label key object.

redraw_in_frame()

This method can only be used after an initial draw which caches the renderer. It is used to efficiently update Axes data (axis ticks, labels, etc are not updated)

relim(*visible_only=False*)

Recompute the data limits based on current artists. If you want to exclude invisible artists from the calculation, set `visible_only=True`

At present, `Collection` instances are not supported.

remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)

Remove a callback based on its *id*.

See also:

add_callback() For adding callbacks

reset_position()

Make the original position the active position

scatter(*x*, *y*, *s=20*, *c=None*, *marker='o'*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *verts=None*, *edgecolors=None*, ***kwargs*)

Make a scatter plot of x vs y, where x and y are sequence like objects of the same length.

Parameters x, y : array_like, shape (n,)

Input data

s : scalar or array_like, shape (n,), optional, default: 20

size in points^2.

c : color, sequence, or sequence of color, optional, default: 'b'

`c` can be a single color format string, or a sequence of color specifications of length `N`, or a sequence of `N` numbers to be mapped to colors using the `cmap` and `norm` specified via kwargs (see below). Note that `c` should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. `c` can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.

marker : `MarkerStyle`, optional, default: 'o'

See `markers` for more information on the different styles of markers scatter supports. `marker` can be either an instance of the class or the text shorthand for a particular marker.

cmap : `Colormap`, optional, default: None

A `Colormap` instance or registered name. `cmap` is only used if `c` is an array of floats. If None, defaults to rc `image.cmap`.

norm : `Normalize`, optional, default: None

A `Normalize` instance is used to scale luminance data to 0, 1. `norm` is only used if `c` is an array of floats. If `None`, use the default `normalize()`.

vmin, vmax : scalar, optional, default: None

`vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. If either are `None`, the min and max of the color array is used. Note if you pass a `norm` instance, your settings for `vmin` and `vmax` will be ignored.

alpha : scalar, optional, default: None
The alpha blending value, between 0 (transparent) and 1 (opaque)

linewidths : scalar or array_like, optional, default: None
If None, defaults to (lines.linewidth,).

edgecolors : color or sequence of color, optional, default: None
If None, defaults to (patch.edgecolor). If 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn. For non-filled markers, the `edgecolors` kwarg is ignored; color is determined by `c`.

Returns paths : *PathCollection*

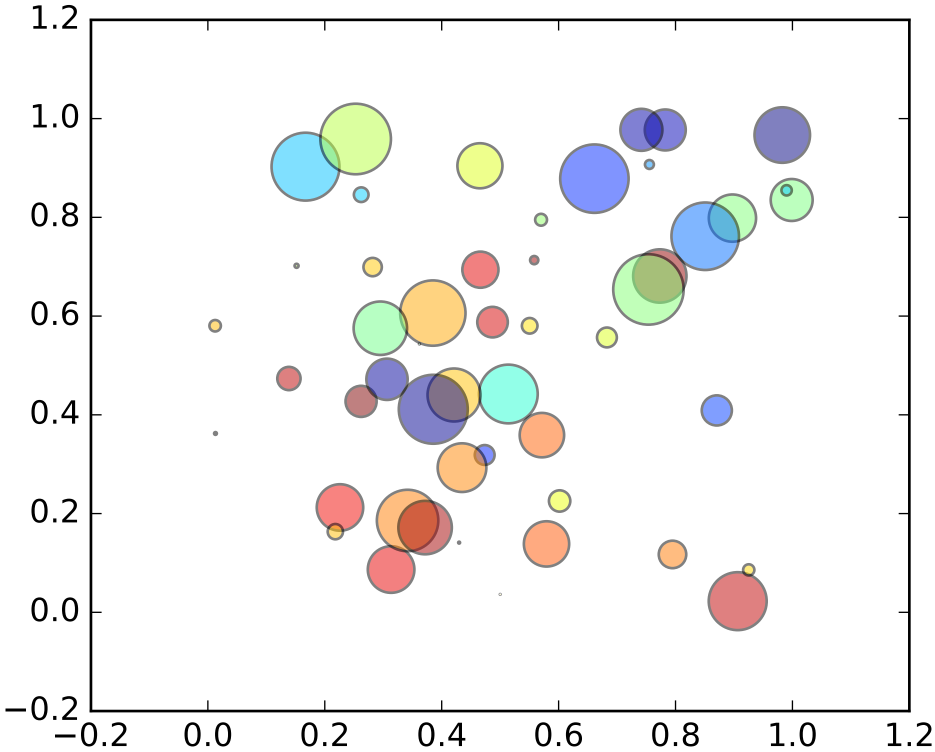
Other Parameters kwargs : *Collection* properties

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 's', 'y', 'linewidths', 'color', 'c', 'facecolors', 'x', 'facecolor', 'edgecolors'.

Examples



semilogx(**args*, ***kwargs*)

Make a plot with log scaling on the *x* axis.

Call signature:

```
semilogx(*args, **kwargs)
```

`semilogx()` supports all the keyword arguments of `plot()` and `matplotlib.axes.Axes.set_xscale()`.

Notable keyword arguments:

***basex*: scalar > 1** Base of the *x* logarithm

***subsx*: [*None* | sequence]** The location of the minor xticks; *None* defaults to autosubs, which depend on the number of decades in the plot; see `set_xscale()` for details.

***nonposx*: ['mask' | 'clip']** Non-positive values in *x* can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
	C

Table 43.26 – continued from previous page

Property	Description
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an *Axes* instance
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a *matplotlib.figure.Figure* instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
linewidth or lw	float value in points
marker	*A valid marker style*
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a *matplotlib.transforms.Transform* instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See also:

`loglog()` For example code and figure

semilogy(**args*, ***kwargs*)

Make a plot with log scaling on the *y* axis.

call signature:

```
semilogy(*args, **kwargs)
```

`semilogy()` supports all the keyword arguments of `plot()` and *matplotlib.axes.Axes.set_yscale()*.

Notable keyword arguments:

***basey*: scalar > 1** Base of the *y* logarithm

***subsy*: [*None* | sequence]** The location of the minor yticks; *None* defaults to autosubs, which depend on the number of decades in the plot; see *set_yscale()* for details.

***nonposy*: ['mask' | 'clip']** Non-positive values in *y* can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are *Line2D* properties:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an *Axes* instance
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a *matplotlib.figure.Figure* instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
linewidth or lw	float value in points
marker	*A valid marker style*
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown

C

Table 43.27 – continued from previous page

Property	Description
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a *matplotlib.transforms.Transform* instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See also:

loglog() For example code and figure

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_adjustable(*adjustable*)
ACCEPTS: ['box' | 'datalim' | 'box-forced']

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha value used for blending - not supported on all backends.

ACCEPTS: float (0.0 transparent through 1.0 opaque)

set_anchor(*anchor*)
anchor

value	description
'C'	Center
'SW'	bottom left
'S'	bottom
'SE'	bottom right
'E'	right
'NE'	top right
'N'	top
'NW'	top left
'W'	left

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_aspect(*aspect*, *adjustable=None*, *anchor=None*)

aspect

value	description
'auto'	automatic; fill position rectangle with data
'normal'	same as 'auto'; deprecated
'equal'	same scaling from data to plot units for x and y
num	a circle will be stretched such that the height is num times the width. aspect=1 is the same as aspect='equal'.

adjustable

value	description
'box'	change physical size of axes
'datalim'	change xlim or ylim
'box-forced'	same as 'box', but axes can be shared

'box' does not allow axes sharing, as this can cause unintended side effect. For cases when sharing axes is fine, use 'box-forced'.

anchor

value	description
'C'	centered
'SW'	lower left corner
'S'	middle of bottom edge
'SE'	lower right corner
etc.	

Deprecated since version 1.2: the option 'normal' for aspect is deprecated. Use 'auto' instead.

set_autoscale_on(*b*)
Set whether autoscaling is applied on plot commands

accepts: [*True* | *False*]

set_autoscalex_on(*b*)
Set whether autoscaling for the x-axis is applied on plot commands

accepts: [*True* | *False*]

set_autoscaley_on(*b*)
Set whether autoscaling for the y-axis is applied on plot commands

accepts: [*True* | *False*]

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_axes_locator(*locator*)
set axes_locator
ACCEPT: a callable object which takes an axes instance and renderer and returns a bbox.

set_axis_bgcolor(*color*)
set the axes background color

ACCEPTS: any matplotlib color - see `colors()`

set_axis_off()
turn off the axis

set_axis_on()
turn on the axis

set_axisbelow(*b*)
Set whether the axis ticks and gridlines are above or below most artists

ACCEPTS: [*True* | *False*]

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_color_cycle(*clist*)
Set the color cycle for any future plot commands on this Axes.

clist is a list of mpl color specifiers.

Deprecated since version 1.5.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_cursor_props(**args*)
Set the cursor property as:

```
ax.set_cursor_props(linewidth, color)
```

or:

```
ax.set_cursor_props((linewidth, color))
```

ACCEPTS: a (*float*, *color*) tuple

set_figure(*fig*)
Set the class:*Axes* figure

accepts a class:*Figure* instance

set_frame_on(*b*)
Set whether the axes rectangle patch is drawn

ACCEPTS: [*True* | *False*]

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_navigate(*b*)
Set whether the axes responds to navigation toolbar commands

ACCEPTS: [*True* | *False*]

set_navigate_mode(*b*)
Set the navigation toolbar button status;

Warning: this is not a user-API function.

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

`set_position`(*pos*, *which='both'*)

Set the axes position with:

```
pos = [left, bottom, width, height]
```

in relative 0,1 coords, or *pos* can be a `Bbox`

There are two position variables: one which is ultimately used, but which may be modified by `apply_aspect()`, and a second which is the starting point for `apply_aspect()`.

Optional keyword arguments: *which*

value	description
'active'	to change the first
'original'	to change the second
'both'	to change both

`set_prop_cycle`(**args*, ***kwargs*)

Set the property cycle for any future plot commands on this Axes.

set_prop_cycle(arg) set_prop_cycle(label, itr) set_prop_cycle(label1=itr1[, label2=itr2[, ...]])

Form 1 simply sets given `Cycler` object.

Form 2 creates and sets a `Cycler` from a label and an iterable.

Form 3 composes and sets a `Cycler` as an inner product of the pairs of keyword arguments. In other words, all of the iterables are cycled simultaneously, as if through zip().

Parameters arg : Cycler

Set the given Cycler. Can also be `None` to reset to the cycle defined by the current style.

label : str

The property key. Must be a valid `Artist` property. For example, 'color' or 'linestyle'. Aliases are allowed, such as 'c' for 'color' and 'lw' for 'linewidth'.

itr : iterable

Finite-length iterable of the property values. These values are validated and will raise a ValueError if invalid.

set_rasterization_zorder(*z*)
Set zorder value below which artists will be rasterized. Set to `None` to disable rasterizing of artists below a particular zorder.

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_title(*label*, *fontdict=None*, *loc='center'*, ***kwargs*)
Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

Parameters label : str
Text to use for the title

fontdict : dict
A dictionary controlling the appearance of the title text, the default `fontdict` is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight' : rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

loc : {'center', 'left', 'right'}, str, optional
Which title to set, defaults to 'center'

Returns text : `Text`
The matplotlib text instance representing the title

Other Parameters kwargs : text properties

Other keyword arguments are text properties, see `Text` for a list of valid text properties.

set_transform(*t*)
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_xbound(*lower=None*, *upper=None*)
Set the lower and upper numerical bounds of the x-axis. This method will honor axes inversion regardless of parameter order. It will not change the _autoscaleXon attribute.

set_xlabel(*xlabel*, *fontdict=None*, *labelpad=None*, ***kwargs*)
Set the label for the xaxis.

Parameters xlabel : string
x label

labelpad : scalar, optional, default: None
spacing in points between the label and the x-axis

Other Parameters kwargs : `Text` properties

See also:

text for information on how override and the optional args work

set_xlim(*left=None*, *right=None*, *emit=True*, *auto=False*, ***kw*)
Call signature:

```
set_xlim(self, *args, **kwargs):
```

Set the data limits for the xaxis

Examples:

```
set_xlim((left, right))
set_xlim(left, right)
set_xlim(left=1) # right unchanged
set_xlim(right=1) # left unchanged
```

Keyword arguments:

***left*: scalar** The left xlim; *xmin*, the previous name, may still be used

***right*: scalar** The right xlim; *xmax*, the previous name, may still be used

***emit*: [*True* | *False*]** Notify observers of limit change

***auto*: [*True* | *False* | *None*]** Turn *x* autoscaling on (*True*), off (*False*; default), or leave unchanged (*None*)

Note, the *left* (formerly *xmin*) value may be greater than the *right* (formerly *xmax*). For example, suppose *x* is years before present. Then one might use:

```
set_ylim(5000, 0)
```

so 5000 years ago is on the left of the plot and the present is on the right.

Returns the current xlimits as a length 2 tuple

ACCEPTS: length 2 sequence of floats

set_xmargin(*m*)

Set padding of X data limits prior to autoscaling.

m times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

set_xscale(*value*, ***kwargs*)

Call signature:

```
set_xscale(value)
```

Set the scaling of the x-axis: 'linear' | 'log' | 'logit' | 'symlog'

ACCEPTS: ['linear' | 'log' | 'logit' | 'symlog']

Different kwargs are accepted, depending on the scale: 'linear'

'log'

***basex/basey*:** The base of the logarithm

***nonposx/nonposy*: ['mask' | 'clip']** non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

'logit'

***nonpos*: ['mask' | 'clip']** values beyond]0, 1[can be masked as invalid, or clipped to a number very close to 0 or 1

'symlog'

***basex/basey*:** The base of the logarithm

***linthreshx/linthreshy*:** The range (-*x*, *x*) within which the plot is linear (to avoid having the plot go to infinity around zero).

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

***linscalex/linscaley*:** This allows the linear range (*-linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

set_xticklabels(*labels*, *fontdict=None*, *minor=False*, ***kwargs*)

Call signature:

```
set_xticklabels(labels, fontdict=None, minor=False, **kwargs)
```

Set the xtick labels with list of strings *labels*. Return a list of axis text instances.

kwargs set the `Text` properties. Valid properties are

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	FancyBboxPatch prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or name or fontname	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']
`rotation_mode`	unknown
`size` or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large'
`sketch_params`	unknown
`snap`	unknown
`stretch` or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'c
`style` or fontstyle	['normal' | 'italic' | 'oblique']
`text`	string or anything printable with '%s' conversion.

Table 43.28 – continued fro

Property	Description
transform	Transform instance
url	a url string
usetex	unknown
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' |
wrap	unknown
x	float
y	float
zorder	any number

ACCEPTS: sequence of strings

set_xticks(*ticks*, *minor=False*)

Set the x ticks with list of *ticks*

ACCEPTS: sequence of floats

set_ybound(*lower=None*, *upper=None*)

Set the lower and upper numerical bounds of the y-axis. This method will honor axes inversion regardless of parameter order. It will not change the _autoscaleYon attribute.

set_ylabel(*ylabel*, *fontdict=None*, *labelpad=None*, ***kwargs*)

Set the label for the yaxis

Parameters ylabel : string

y label

labelpad : scalar, optional, default: None

spacing in points between the label and the x-axis

Other Parameters kwargs : Text properties

See also:

text for information on how override and the optional args work

set_ylim(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***kw*)

Call signature:

```
set_ylim(self, *args, **kwargs):
```

Set the data limits for the yaxis

Examples:

```
set_ylim((bottom, top))
set_ylim(bottom, top)
set_ylim(bottom=1) # top unchanged
set_ylim(top=1) # bottom unchanged
```

Keyword arguments:

***bottom*: scalar** The bottom ylim; the previous name, *ymin*, may still be used

***top*: scalar** The top ylim; the previous name, *ymax*, may still be used

***emit*: [*True* | *False*]** Notify observers of limit change

***auto*: [*True* | *False* | *None*]** Turn *y* autoscaling on (*True*), off (*False*; default), or leave unchanged (*None*)

Note, the *bottom* (formerly *ymin*) value may be greater than the *top* (formerly *ymax*). For example, suppose *y* is depth in the ocean. Then one might use:

```
set_ylim(5000, 0)
```

so 5000 m depth is at the bottom of the plot and the surface, 0 m, is at the top.

Returns the current ylimits as a length 2 tuple

ACCEPTS: length 2 sequence of floats

set_ymargin(*m*)

Set padding of Y data limits prior to autoscaling.

m times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

set_yscale(*value*, ***kwargs*)

Call signature:

```
set_yscale(value)
```

Set the scaling of the y-axis: 'linear' | 'log' | 'logit' | 'symlog'

ACCEPTS: ['linear' | 'log' | 'logit' | 'symlog']

Different kwargs are accepted, depending on the scale: 'linear'

'log'

***basex*/*basey*:** The base of the logarithm

***nonposx*/*nonposy*: ['mask' | 'clip']** non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number

***subsx*/*subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

'logit'

***nonpos*: ['mask' | 'clip']** values beyond]0, 1[can be masked as invalid, or clipped to a number very close to 0 or 1

'symlog'

***basex*/*basey*:** The base of the logarithm

***linthreshx*/*linthreshy*:** The range (-*x*, *x*) within which the plot is linear (to avoid having the plot go to infinity around zero).

subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

linscalex/linscaley: This allows the linear range (*-linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

set_yticklabels(*labels*, *fontdict=None*, *minor=False*, ***kwargs*)

Call signature:

```
set_yticklabels(labels, fontdict=None, minor=False, **kwargs)
```

Set the y tick labels with list of strings *labels*. Return a list of `Text` instances.

kwargs set `Text` properties for the labels. Valid properties are

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	FancyBboxPatch prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or name or fontname	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']
`rotation_mode`	unknown

Table 43.29 – continued fro

Property	Description
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large'
sketch_params	unknown
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'c
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	*Transform* instance
url	a url string
usetex	unknown
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' |
wrap	unknown
x	float
y	float
zorder	any number

ACCEPTS: sequence of strings

set_yticks(*ticks*, *minor=False*)

Set the y ticks with list of *ticks*

ACCEPTS: sequence of floats

Keyword arguments:

***minor*: [*False* | *True*]** Sets the minor ticks if *True*

set_zorder(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

specgram(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *cmap=None*, *xextent=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *mode=None*, *scale=None*, *vmin=None*, *vmax=None*, ***kwargs*)

Plot a spectrogram.

Call signature:

```
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
         window=mlab.window_hanning, noverlap=128,
         cmap=None, xextent=None, pad_to=None, sides='default',
         scale_by_freq=None, mode='default', scale='default',
         **kwargs)
```

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*. The spectrogram is plotted as a colormap (using imshow).

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or** callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***mode*: ['default' | 'psd' | 'magnitude' | 'angle' | 'phase']** What sort of spectrum to use. Default is 'psd'. which takes the power spectral density. 'complex' returns the complex-valued frequency spectrum. 'magnitude' returns the magnitude spectrum. 'angle' returns the phase spectrum without unwrapping. 'phase' returns the phase spectrum with unwrapping.

***noverlap*: integer** The number of points of overlap between blocks. The default value is

128.

***scale*: ['default' | 'linear' | 'dB']** The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale. When *mode* is 'psd', this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). 'default' is 'dB' if *mode* is 'psd' or 'magnitude' and 'linear' otherwise. This must be 'linear' if *mode* is 'angle' or 'phase'.

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

***cmap*:** A `matplotlib.colors.Colormap` instance; if *None*, use default determined by rc

***xextent*:** The image extent along the x-axis. xextent = (xmin,xmax) The default is (0,max(bins)), where bins is the return value from `specgram()`

***kwargs*:** Additional kwargs are passed on to imshow which makes the specgram image

Note: *detrend* and *scale_by_freq* only apply when *mode* is set to 'psd'

Returns the tuple (*spectrum*, *freqs*, *t*, *im*):

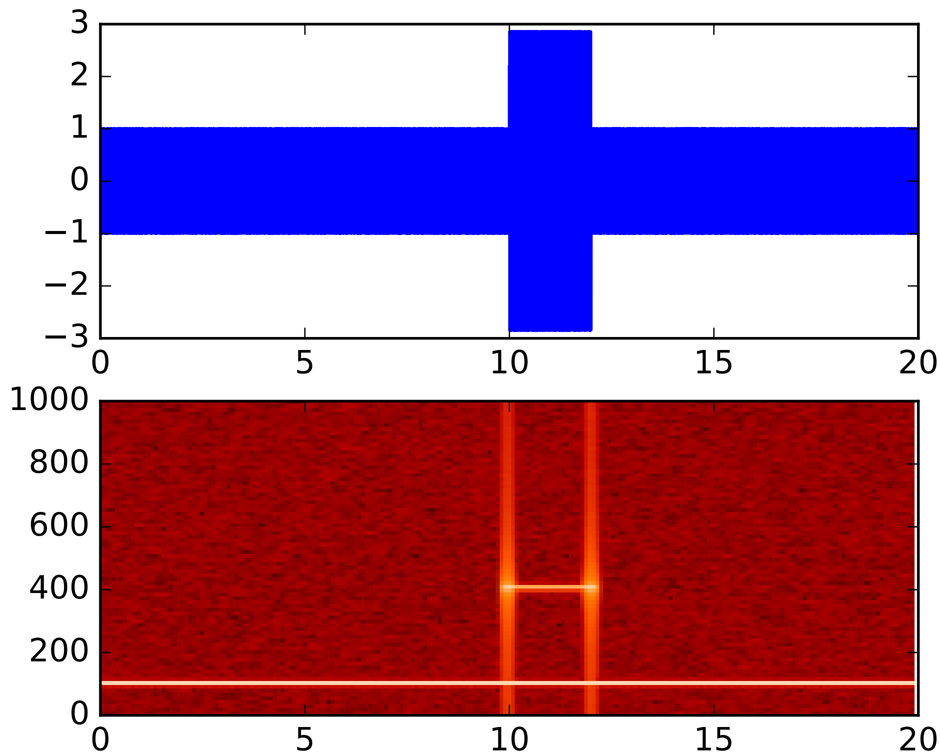
***spectrum*: 2-D array** columns are the periodograms of successive segments

***freqs*: 1-D array** The frequencies corresponding to the rows in *spectrum*

***t*: 1-D array** The times corresponding to midpoints of segments (i.e the columns in *spectrum*)

***im*: instance of class `AxesImage`** The image created by imshow containing the spectrogram

Example:



See also:

psd() psd() differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

magnitude_spectrum() A single spectrum, similar to having a single segment when *mode* is 'magnitude'. Plots a line instead of a colormap.

angle_spectrum() A single spectrum, similar to having a single segment when *mode* is 'angle'. Plots a line instead of a colormap.

phase_spectrum() A single spectrum, similar to having a single segment when *mode* is 'phase'. Plots a line instead of a colormap.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

spy(*Z*, *precision=0*, *marker=None*, *markersize=None*, *aspect='equal'*, *origin='upper'*, ***kwargs*)

Plot the sparsity pattern on a 2-D array.

spy(Z) plots the sparsity pattern of the 2-D array *Z*.

Parameters Z : sparse array (n, m)

The array to be plotted.

precision : float, optional, default: 0
If *precision* is 0, any non-zero value will be plotted; else, values of |*Z*| > *precision* will be plotted.

For `scipy.sparse.spmatrix` instances, there is a special case: if *precision* is 'present', any value present in the array will be plotted, even if it is identically zero.

origin : ["upper", "lower"], optional, default: "upper"
Place the [0,0] index of the array in the upper left or lower left corner of the axes.

aspect : ['auto' | 'equal' | scalar], optional, default: "equal"
If 'equal', and `extent` is None, changes the axes aspect ratio to match that of the image. If `extent` is not `None`, the axes aspect ratio is changed to match that of the extent.

If 'auto', changes the image aspect ratio to match that of the axes.

If None, default to rc `image.aspect` value.

Two plotting styles are available: image or marker. Both

are available for full arrays, but only the marker style

works for :class:'scipy.sparse.spmatrix' instances.

If *marker* and *markersize* are *None*, an image will be

returned and any remaining kwargs are passed to

:func:'~matplotlib.pyplot.imshow'; else, a

:class:'~matplotlib.lines.Line2D' object will be returned with

the value of marker determining the marker type, and any

remaining kwargs passed to the

:meth:'~matplotlib.axes.Axes.plot' method.

If *marker* and *markersize* are *None*, useful kwargs include:

*** *cmap***

*** *alpha***

See also:

`imshow` for image options.
`plot` for plotting options

`stackplot`(*x*, **args*, ***kwargs*)
Draws a stacked area plot.

x : 1d array of dimension N
y [2d array of dimension MxN, OR any number 1d arrays each of dimension] 1xN. The data is assumed to be unstacked. Each of the following calls is legal:

```
stackplot(x, y)               # where y is MxN
stackplot(x, y1, y2, y3, y4)  # where y1, y2, y3, y4, are all 1xNm
```

Keyword arguments:

baseline [['zero', 'sym', 'wiggle', 'weighted_wiggle']] Method used to calculate the baseline. 'zero' is just a simple stacked plot. 'sym' is symmetric around zero and is sometimes called `ThemeRiver`. 'wiggle' minimizes the sum of the squared slopes. 'weighted_wiggle' does the same but weights to account for size of each layer. It is also called `Streamgraph`-layout. More details can be found at http://www.leebyron.com/else/streamgraph/.

labels : A list or tuple of labels to assign to each data series.

colors [A list or tuple of colors. These will be cycled through and] used to colour the stacked areas. All other keyword arguments are passed to `fill_between()`

Returns *r* : A list of `PolyCollection`, one for each element in the stacked area plot.

stale

If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

start_pan(*x*, *y*, *button*)

Called when a pan operation has started.

x, *y* are the mouse coordinates in display coords. button is the mouse button number:

- 1: LEFT
- 2: MIDDLE
- 3: RIGHT

Note: Intended to be overridden by new projection types.

stem(**args*, ***kwargs*)

Create a stem plot.

Call signatures:

```
stem(y, linefmt='b-', markerfmt='bo', basefmt='r-')
stem(x, y, linefmt='b-', markerfmt='bo', basefmt='r-')
```

A stem plot plots vertical lines (using *linefmt*) at each *x* location from the baseline to *y*, and places a marker there using *markerfmt*. A horizontal line at 0 is is plotted using *basefmt*.

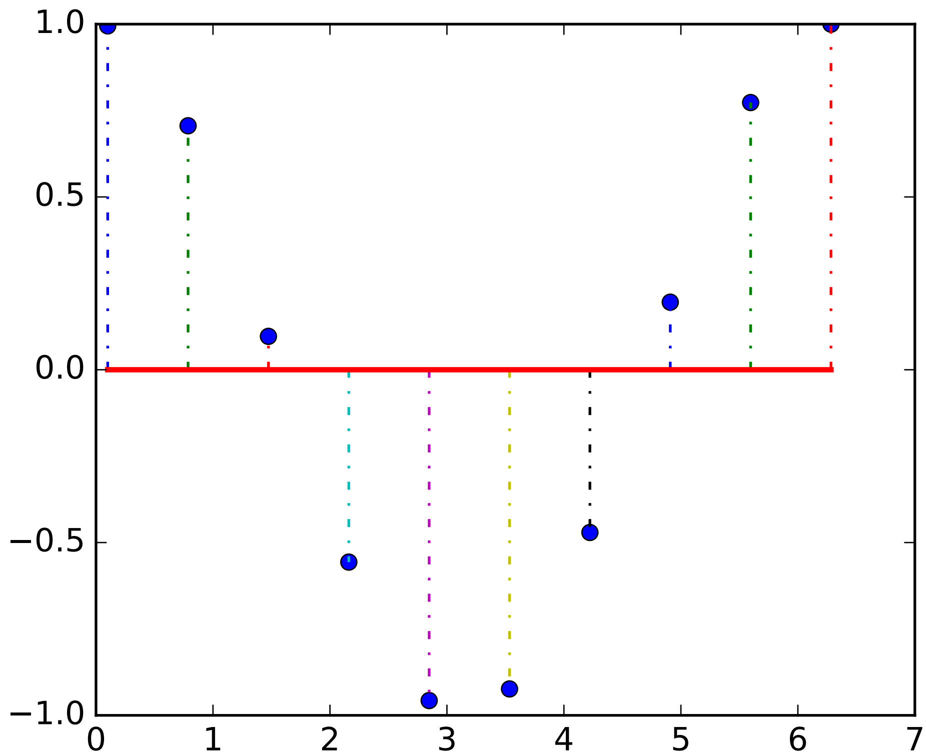
If no *x* values are provided, the default is (0, 1, ..., len(y) - 1)

Return value is a tuple (*markerline*, *stemlines*, *baseline*).

See also:

This document for details.

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

step(*x*, *y*, **args*, ***kwargs*)

Make a step plot.

Call signature:

```
step(x, y, *args, **kwargs)
```

Additional keyword args to `step()` are the same as those for `plot()`.

x and *y* must be 1-D sequences, and it is assumed, but not checked, that *x* is uniformly increasing.

Keyword arguments:

***where*: ['pre' | 'post' | 'mid']** If 'pre' (the default), the interval from x[i] to x[i+1] has level y[i+1].

If 'post', that interval has level y[i].

If 'mid', the jumps in *y* occur half-way between the *x*-values.

Return value is a list of lines that were added.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

streamplot(*x*, *y*, *u*, *v*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='-|>'*, *minlength=0.1*, *transform=None*, *zorder=1*, *start_points=None*)

Draws streamlines of a vector flow.

x*, *y [1d arrays] an *evenly spaced* grid.

u*, *v [2d arrays] x and y-velocities. Number of rows should match length of y, and the number of columns should match x.

density [float or 2-tuple] Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid—*density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use [density_x, density_y].

linewidth [numeric or 2d array] vary linewidth when given a 2d array with the same shape as velocities.

color [matplotlib color code, or 2d array] Streamline color. When given an array with the same shape as velocities, *color* values are converted to colors using *cmap*.

cmap [`Colormap`] Colormap used to plot streamlines and arrows. Only necessary when using an array input for *color*.

norm [`Normalize`] Normalize object used to scale luminance data to 0, 1. If None, stretch (min, max) to (0, 1). Only necessary when *color* is an array.

arrowsize [float] Factor scale arrow size.

arrowstyle [str] Arrow style specification. See `FancyArrowPatch`.

minlength [float] Minimum length of streamline in axes coordinates.

***start_points*: Nx2 array** Coordinates of starting points for the streamlines. In data coordinates, the same as the `x` and `y` arrays.

zorder [int] any number

Returns:

stream_container [StreamplotSet] Container object with attributes

- lines: `matplotlib.collections.LineCollection` of streamlines
- arrows: collection of `matplotlib.patches.FancyArrowPatch` objects representing arrows half-way along stream lines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

table(***kwargs*)

Add a table to the current axes.

Call signature:

```
table(cellText=None, cellColours=None,
      cellLoc='right', colWidths=None,
```

```
rowLabels=None, rowColours=None, rowLoc='left',
colLabels=None, colColours=None, colLoc='center',
loc='bottom', bbox=None):
```

Returns a `matplotlib.table.Table` instance. For finer grained control over tables, use the `Table` class and add it to the axes with `add_table()`.

Thanks to John Gill for providing the class and table.

kwargs control the `Table` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`contains`	a callable function
`figure`	a `matplotlib.figure.Figure` instance
`fontsize`	a float in points
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

text(*x*, *y*, *s*, *fontdict=None*, *withdash=False*, ***kwargs*)

Add text to the axes.

Add text in string `s` to axis at location `x`, `y`, data coordinates.

Parameters x, y : scalars

data coordinates

s : string

text

fontdict : dictionary, optional, default: None

A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.

withdash : boolean, optional, default: False

Creates a `TextWithDash` instance instead of a `Text` instance.

Other Parameters kwargs : `Text` properties.

Other miscellaneous text parameters.

Examples

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the axes:

```
>>> text(0.5, 0.5,'matplotlib', horizontalalignment='center',
...      verticalalignment='center',
...      transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword `bbox`. `bbox` is a dictionary of `Rectangle` properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

tick_params(*axis='both'*, ***kwargs*)
Change the appearance of ticks and tick labels.

Keyword arguments:
axis [['x' | 'y' | 'both']] Axis on which to operate; default is 'both'.
reset [[True | False]] If *True*, set all parameters to defaults before processing other keyword arguments. Default is *False*.
which [['major' | 'minor' | 'both']] Default is 'major'; apply arguments to *which* ticks.
direction [['in' | 'out' | 'inout']] Puts ticks inside the axes, outside the axes, or both.
length Tick length in points.
width Tick width in points.
color Tick color; accepts any mpl color spec.
pad Distance in points between tick and label.
labelsize Tick label font size in points or as a string (e.g., 'large').
labelcolor Tick label color; mpl color spec.
colors Changes the tick color and the label color to the same value: mpl color spec.
zorder Tick and label zorder.
bottom, top, left, right [[bool | 'on' | 'off']] controls whether to draw the respective ticks.
labelbottom, labeltop, labelleft, labelright Boolean or ['on' | 'off'], controls whether to draw the respective tick labels.
Example:

```
ax.tick_params(direction='out', length=6, width=2, colors='r')
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red.

ticklabel_format(***kwargs*)
Change the `ScalarFormatter` used by default for linear axes.

Optional keyword arguments:

Keyword	Description
style	['sci' (or 'scientific') | 'plain'] plain turns off scientific notation
scilimits	(m, n), pair of integers; if *style* is 'sci', scientific notation will be used for numbers outside the range 10'm':sup: to 10'n':sup:. Use (0,0) to include all numbers.
useOffset	[True | False | offset]; if True, the offset will be calculated as needed; if False, no offset will be used; if a numeric offset is specified, it will be used.
axis	['x' | 'y' | 'both']
useLocale	If True, format the number according to the current locale. This affects things such as the character used for the decimal separator. If False, use C-style (English) formatting. The default setting is controlled by the axes.formatter.use_locale rcparam.

Only the major ticks are affected. If the method is called when the `ScalarFormatter` is not the `Formatter` being used, an `AttributeError` will be raised.

tricontour(**args*, ***kwargs*)

Draw contours on an unstructured triangular grid. `tricontour()` and `tricontourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

```
tricontour(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```
tricontour(x, y, ...)
tricontour(x, y, triangles, ...)
tricontour(x, y, triangles=triangles, ...)
tricontour(x, y, mask=mask, ...)
tricontour(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The remaining arguments may be:

```
tricontour(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontour(..., Z, N)
```

contour *N* automatically-chosen levels.

```
tricontour(..., Z, V)
```

draw contour lines at the values specified in sequence *V*, which must be in increasing order.

```
tricontourf(..., Z, V)
```

fill the (len(*V*)-1) regions between the values in *V*, which must be in increasing order.

```
tricontour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

`C = tricontour(...)` returns a `TriContourSet` object.

Optional keyword arguments:

***colors*: [*None* | string | (mpl_colors)]** If *None*, the colormap specified by cmap will be used.

If a string, like 'r' or 'red', all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

***alpha*: float** The alpha blending value

***cmap*: [*None* | Colormap]** A cm `Colormap` instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

***levels* [level0, level1, ..., leveln]** A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

***origin*: [*None* | 'upper' | 'lower' | 'image']** If *None*, the first value of *Z* will correspond to the lower left corner, location (0,0). If 'image', the rc value for `image.origin` will be used.

This keyword is not active if *X* and *Y* are specified in the call to contour.

extent: [*None* | (x0,x1,y0,y1)]

If *origin* is not *None*, then *extent* is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0], and (*x1*, *y1*) is the position of Z[-1,-1].

This keyword is not active if *X* and *Y* are specified in the call to contour.

***locator*: [*None* | ticker.Locator subclass]** If *locator* is None, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

***extend*: ['neither' | 'both' | 'min' | 'max']** Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

***xunits*, *yunits*: [*None* | registered units]** Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

tricontour-only keyword arguments:

***linewidths*: [*None* | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified

***linestyles*: [*None* | 'solid' | 'dashed' | 'dashdot' | 'dotted']** If *linestyles* is *None*, the 'solid' is used.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in `contour.negative_linestyle` in `matplotlibrc` will be used.

tricontourf-only keyword arguments:

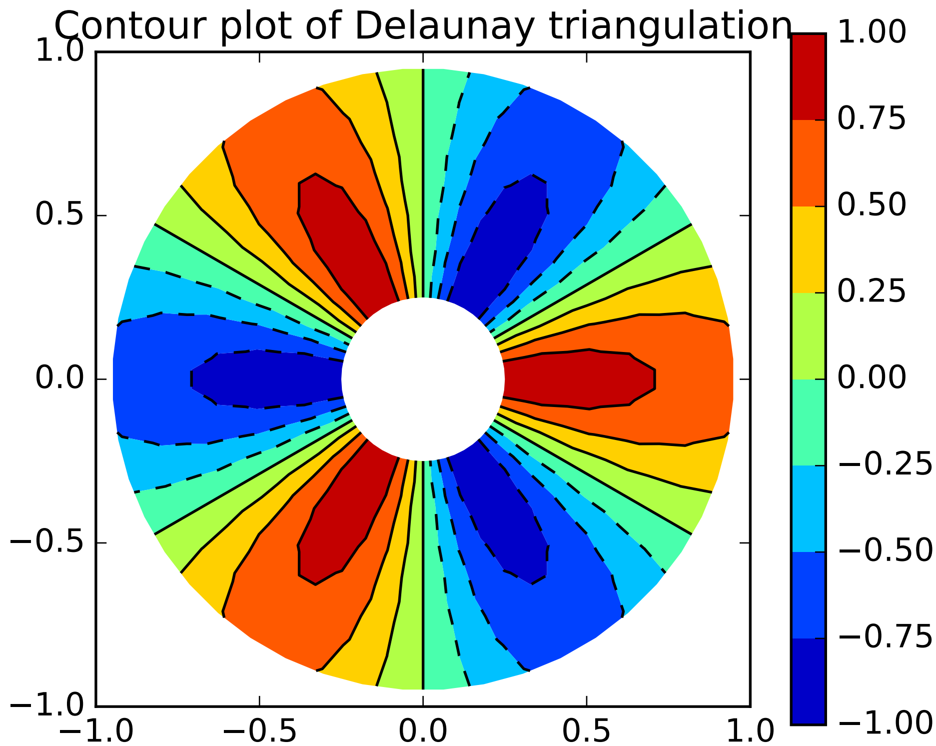
***antialiased*: [*True* | *False*]** enable antialiasing

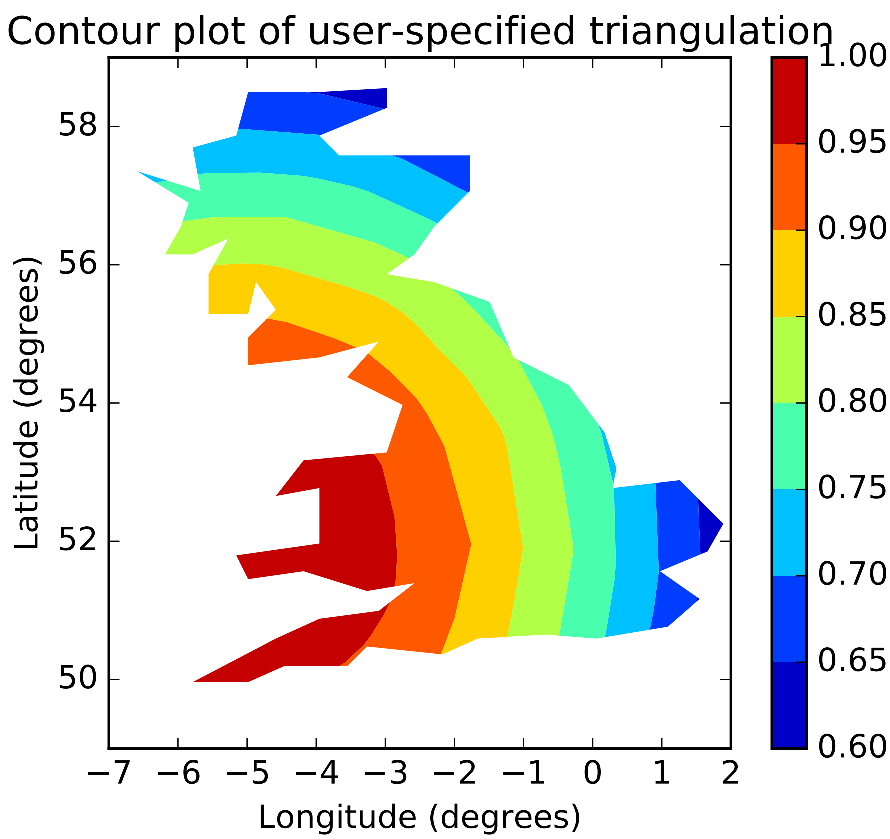
Note: tricontourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < z <= z2
```

There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

Examples:


Contour plot of Delaunay triangulation
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



tricontourf(**args*, ***kwargs*)

Draw contours on an unstructured triangular grid. `tricontour()` and `tricontourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

```
tricontour(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```
tricontour(x, y, ...)
tricontour(x, y, triangles, ...)
tricontour(x, y, triangles=triangles, ...)
tricontour(x, y, mask=mask, ...)
tricontour(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The remaining arguments may be:

```
tricontour(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontour(..., Z, N)
```

contour *N* automatically-chosen levels.

```
tricontour(..., Z, V)
```

draw contour lines at the values specified in sequence *V*, which must be in increasing order.

```
tricontourf(..., Z, V)
```

fill the (len(*V*)-1) regions between the values in *V*, which must be in increasing order.

```
tricontour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

`C = tricontour(...)` returns a `TriContourSet` object.

Optional keyword arguments:

***colors*: [*None* | string | (mpl_colors)]** If *None*, the colormap specified by cmap will be used.

If a string, like 'r' or 'red', all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

***alpha*: float** The alpha blending value

***cmap*: [*None* | Colormap]** A cm `Colormap` instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

***levels* [level0, level1, ..., leveln]** A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

***origin*: [*None* | 'upper' | 'lower' | 'image']** If *None*, the first value of *Z* will correspond to the lower left corner, location (0,0). If 'image', the rc value for `image.origin` will be used.

This keyword is not active if *X* and *Y* are specified in the call to contour.

extent: [*None* | (x0,x1,y0,y1)]

If *origin* is not *None*, then *extent* is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0], and (*x1*, *y1*) is the position of Z[-1,-1].

This keyword is not active if *X* and *Y* are specified in the call to contour.

***locator*: [*None* | ticker.Locator subclass]** If *locator* is None, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

***extend*: ['neither' | 'both' | 'min' | 'max']** Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

***xunits*, *yunits*: [*None* | registered units]** Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

tricontour-only keyword arguments:

***linewidths*: [*None* | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified

***linestyles*: [*None* | 'solid' | 'dashed' | 'dashdot' | 'dotted']** If *linestyles* is *None*, the 'solid' is used.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in `contour.negative_linestyle` in `matplotlibrc` will be used.

tricontourf-only keyword arguments:

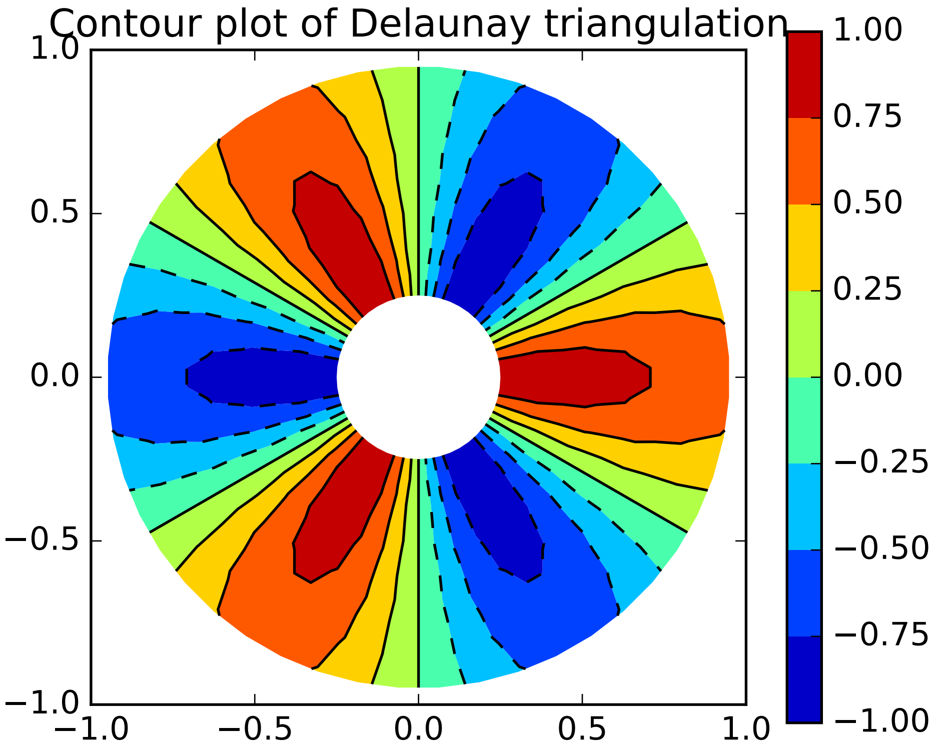
***antialiased*: [*True* | *False*]** enable antialiasing

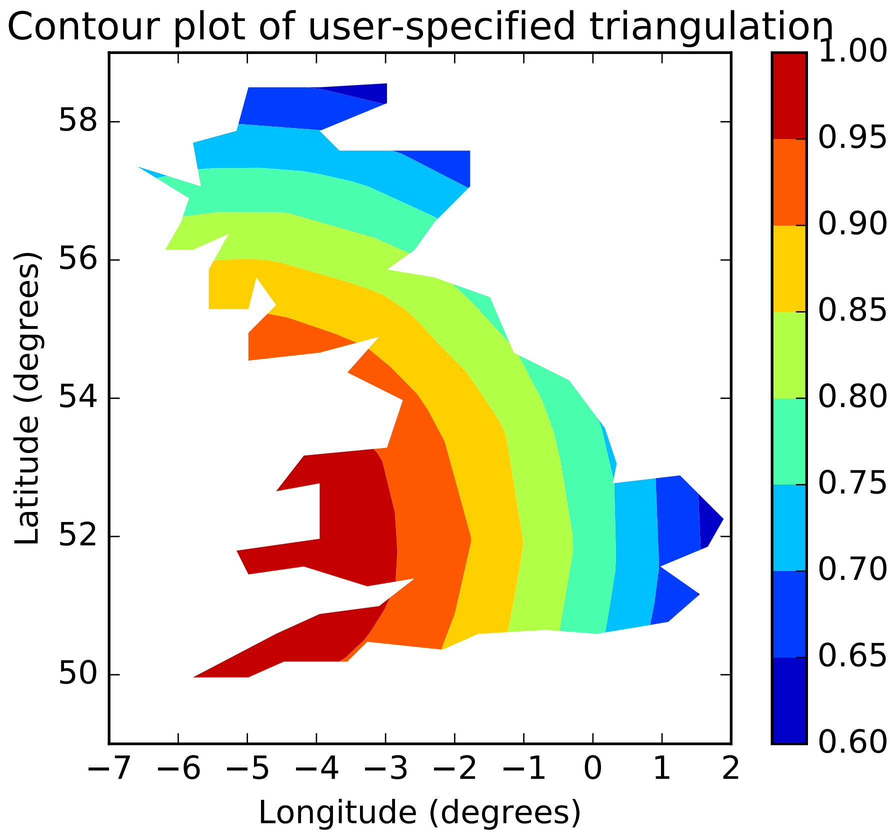
Note: tricontourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < z <= z2
```

There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

Examples:


Contour plot of Delaunay triangulation
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



tripcolor(**args*, ***kwargs*)

Create a pseudocolor plot of an unstructured triangular grid.

The triangulation can be specified in one of two ways; either:

```
tripcolor(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```
tripcolor(x, y, ...)
tripcolor(x, y, triangles, ...)
tripcolor(x, y, triangles=triangles, ...)
tripcolor(x, y, mask=mask, ...)
tripcolor(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

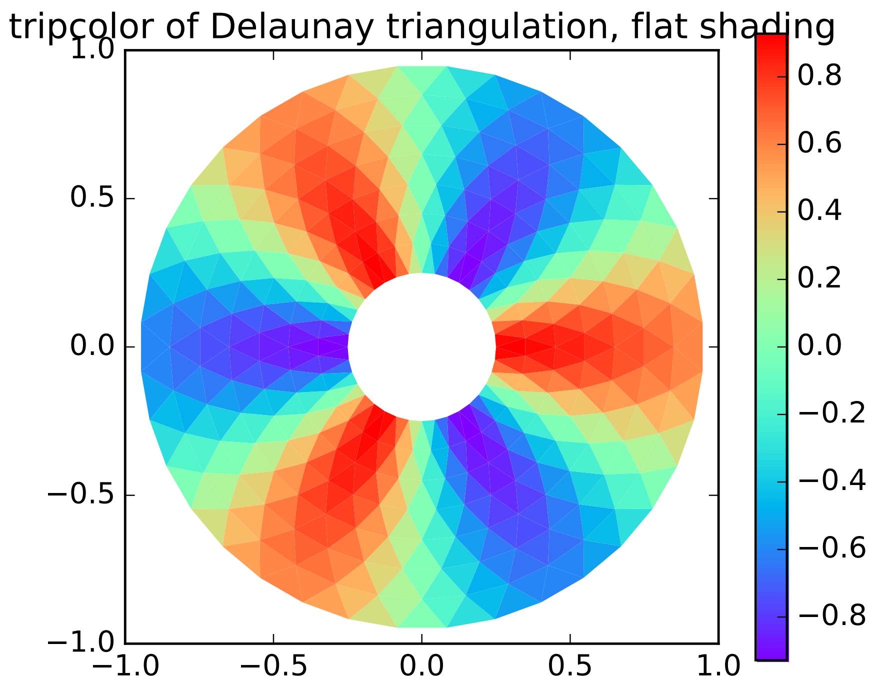
The next argument must be *C*, the array of color values, either one per point in the triangulation if color values are defined at points, or one per triangle in the triangulation if color values are defined at triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the kwarg *facecolors*=C instead of just *C*.

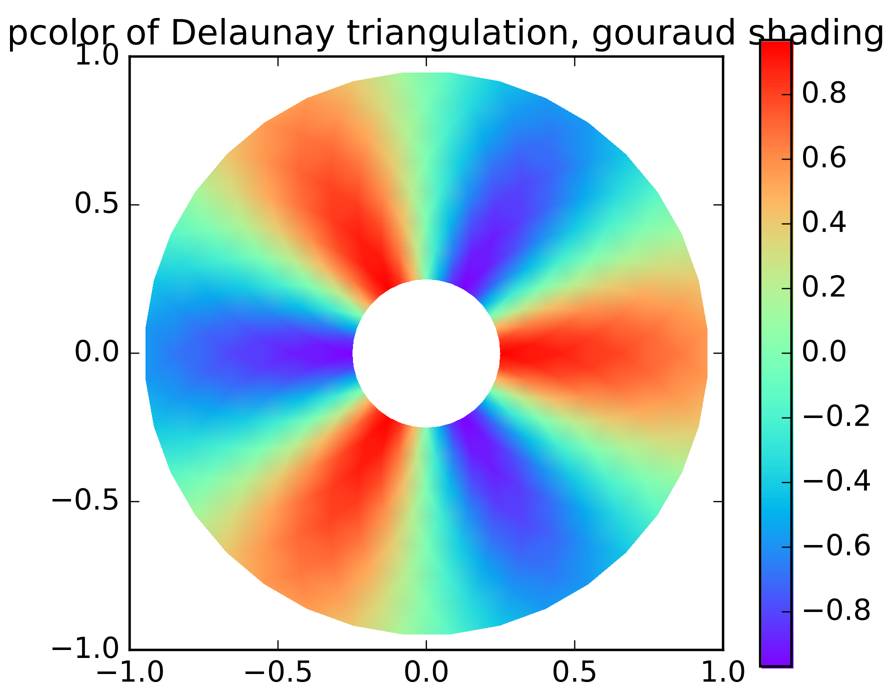
shading may be 'flat' (the default) or 'gouraud'. If *shading* is 'flat' and C values are defined at points, the color values used for each triangle are from the mean C of the triangle's three points.

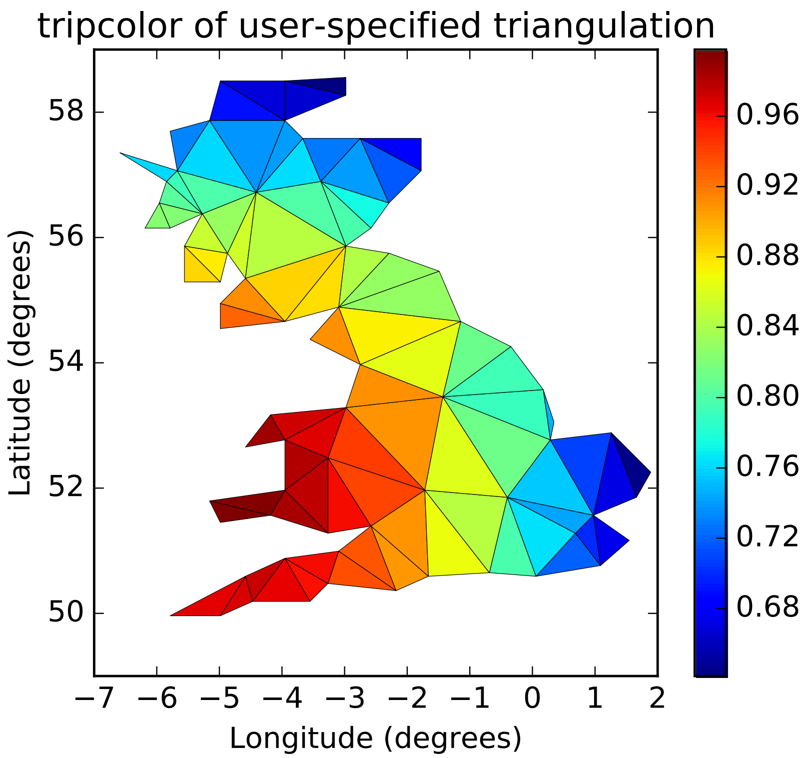
If *shading* is ‘gouraud’ then color values must be defined at points.

The remaining kwargs are the same as for `pcolor()`.

Example:







triplot(**args*, ***kwargs*)

Draw a unstructured triangular grid as lines and/or markers.

The triangulation to plot can be specified in one of two ways; either:

```
triplot(triangulation, ...)
```

where triangulation is a *matplotlib.tri.Triangulation* object, or

```
triplot(x, y, ...)
triplot(x, y, triangles, ...)
triplot(x, y, triangles=triangles, ...)
triplot(x, y, mask=mask, ...)
triplot(x, y, triangles, mask=mask, ...)
```

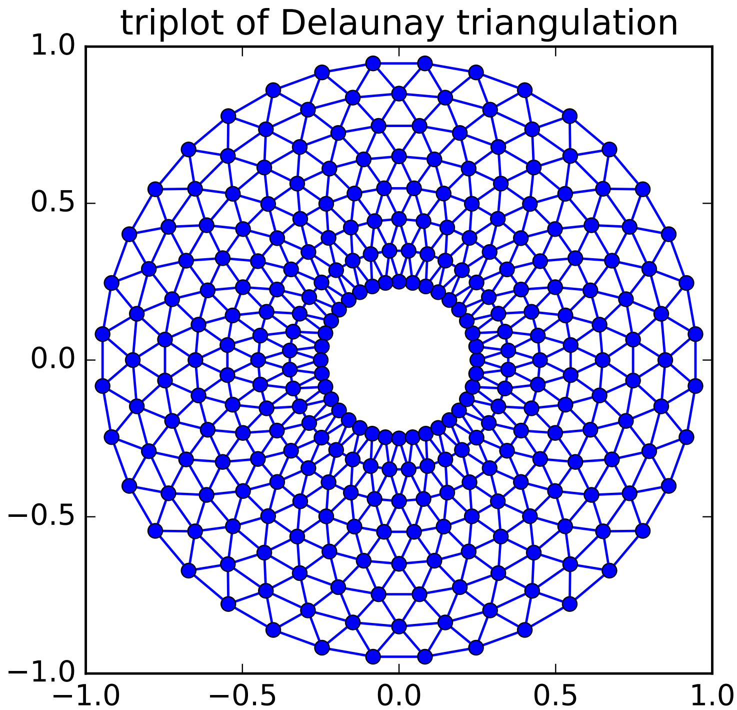
in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

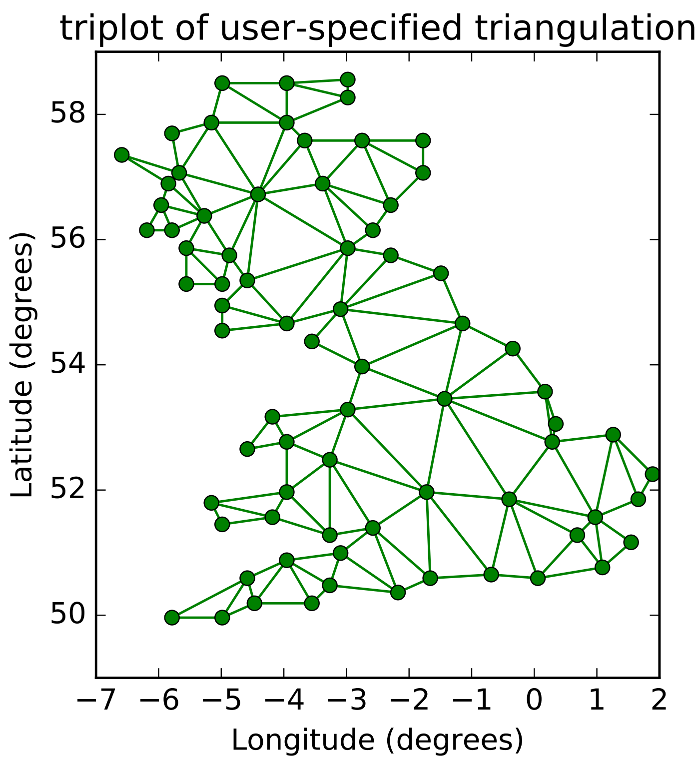
The remaining args and kwargs are the same as for *plot()*.

Return a list of 2 *Line2D* containing respectively:

- the lines plotted for triangles edges
- the markers plotted for triangles nodes

Example:





twinx()

Call signature:

```
ax = twinx()
```

create a twin of Axes for generating a plot with a sharex x-axis but independent y axis. The y-axis of self will have ticks on left and the returned axes will have ticks on the right.

Note: For those who are 'picking' artists while using twinx, pick events are only called for the artists in the top-most axes.

twiny()

Call signature:

```
ax = twiny()
```

create a twin of Axes for generating a plot with a shared y-axis but independent x axis. The x-axis of self will have ticks on bottom and the returned axes will have ticks on the top.

Note: For those who are 'picking' artists while using twiny, pick events are only called for the artists in the top-most axes.

update(*props*)

Update the properties of this `Artist` from the dictionary *prop*.

update_datalim(*xys*, *updatex=True*, *updatey=True*)

Update the data lim bbox with seq of xy tups or equiv. 2-D array

update_datalim_bounds(*bounds*)

Update the datalim to include the given `Bbox` *bounds*

update_datalim_numerix(*x*, *y*)

Update the data lim bbox with seq of xy tups

update_from(*other*)

Copy properties from *other* to *self*.

violin(*vpstats*, *positions=None*, *vert=True*, *widths=0.5*, *showmeans=False*, *showextrema=True*, *showmedians=False*)

Drawing function for violin plots.

Call signature:

```
violin(vpstats, positions=None, vert=True, widths=0.5,
       showmeans=False, showextrema=True, showmedians=False):
```

Draw a violin plot for each column of `vpstats`. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, and the maximum.

Parameters vpstats : list of dicts

A list of dictionaries containing stats for each violin plot. Required keys are:

- `coords`: A list of scalars containing the coordinates that the violin's kernel density estimate were evaluated at.
- `vals`: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in *coords*.
- `mean`: The mean value for this violin's dataset.
- `median`: The median value for this violin's dataset.
- `min`: The minimum value for this violin's dataset.
- `max`: The maximum value for this violin's dataset.

positions : array-like, default = [1, 2, ..., n]
Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

vert : bool, default = True.
If true, plots the violins veritcally. Otherwise, plots the violins horizontally.

widths : array-like, default = 0.5
Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.

showmeans : bool, default = False
If true, will toggle rendering of the means.

showextrema : bool, default = True
If true, will toggle rendering of the extrema.

showmedians : bool, default = False
If true, will toggle rendering of the medians.

Returns result : dict
A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the *`matplotlib.collections.PolyCollection`* instances containing the filled area of each violin.
- `cmeans`: A *`matplotlib.collections.LineCollection`* instance created to identify the mean values of each of the violin's distribution.
- `cmins`: A *`matplotlib.collections.LineCollection`* instance created to identify the bottom of each violin's distribution.
- `cmaxes`: A *`matplotlib.collections.LineCollection`* instance created to identify the top of each violin's distribution.
- `cbars`: A *`matplotlib.collections.LineCollection`* instance created to identify the centers of each violin's distribution.
- `cmedians`: A *`matplotlib.collections.LineCollection`* instance created to identify the median values of each of the violin's distribution.

violinplot(*dataset*, *positions=None*, *vert=True*, *widths=0.5*, *showmeans=False*, *showextrema=True*, *showmedians=False*, *points=100*, *bw_method=None*)

Make a violin plot.

Call signature:

```
violinplot(dataset, positions=None, vert=True, widths=0.5,
           showmeans=False, showextrema=True, showmedians=False,
           points=100, bw_method=None):
```

Make a violin plot for each column of *dataset* or each vector in sequence *dataset*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, and the maximum.

Parameters dataset : Array or a sequence of vectors.

The input data.

positions [array-like, default = [1, 2, ..., n]] Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

vert [bool, default = True.] If true, creates a vertical violin plot. Otherwise, creates a horizontal violin plot.

widths [array-like, default = 0.5] Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.

showmeans [bool, default = False] If `True`, will toggle rendering of the means.

showextrema [bool, default = True] If `True`, will toggle rendering of the extrema.

showmedians [bool, default = False] If `True`, will toggle rendering of the medians.

points [scalar, default = 100] Defines the number of points to evaluate each of the gaussian kernel density estimations at.

bw_method [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If a scalar, this will be used directly as `kde.factor`. If a callable, it should take a `GaussianKDE` instance as its only parameter and return a scalar. If None (default), 'scott' is used.

Returns result : dict

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the *matplotlib.collections.PolyCollection* instances containing the filled area of each violin.
- `cmeans`: A *matplotlib.collections.LineCollection* instance created to identify the mean values of each of the violin's distribution.

- `cmins`: A *matplotlib.collections.LineCollection* instance created to identify the bottom of each violin's distribution.
- `cmaxes`: A *matplotlib.collections.LineCollection* instance created to identify the top of each violin's distribution.
- `cbars`: A *matplotlib.collections.LineCollection* instance created to identify the centers of each violin's distribution.
- `cmedians`: A *matplotlib.collections.LineCollection* instance created to identify the median values of each of the violin's distribution.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'dataset'.

vlines(*x*, *ymin*, *ymax*, *colors='k'*, *linestyles='solid'*, *label=''*, ***kwargs*)

Plot vertical lines.

Plot vertical lines at each `x` from `ymin` to `ymax`.

Parameters x : scalar or 1D array_like

x-indexes where to plot the lines.

ymin, ymax : scalar or 1D array_like

Respective beginning and end of each line. If scalars are provided, all lines will have same length.

colors : array_like of colors, optional, default: 'k'

linestyles : ['solid' | 'dashed' | 'dashdot' | 'dotted'], optional

label : string, optional, default: ''

Returns lines : *LineCollection*

Other Parameters kwargs : *LineCollection* properties.

See also:

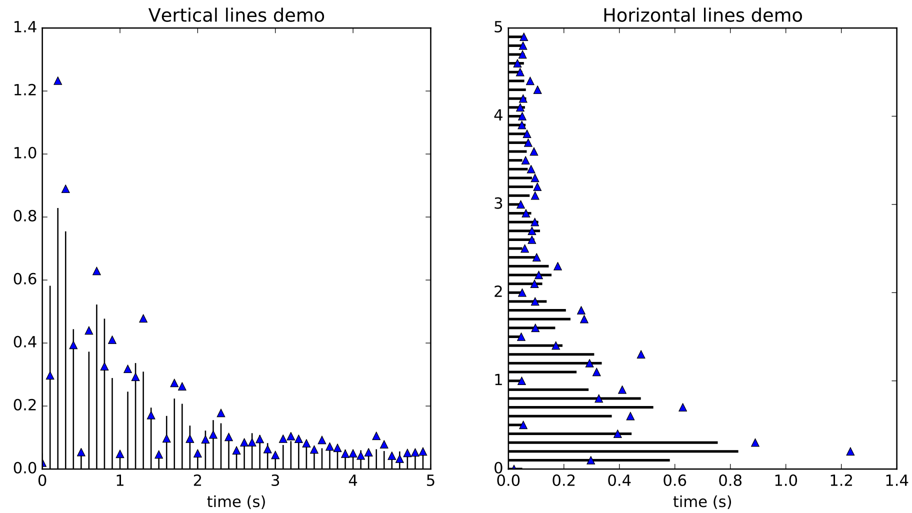
hlines horizontal lines

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'ymax', 'x', 'colors', 'ymin'.

Examples



xaxis_date(*tz=None*)
Sets up x-axis ticks and labels that treat the x data as dates.

tz is a timezone string or `tzinfo` instance. Defaults to rc value.

xaxis_inverted()
Returns *True* if the x-axis is inverted.

xcorr(*x*, *y*, *normed=True*, *detrend=<function detrend_none>*, *usevlines=True*, *maxlags=10*, ***kwargs*)
Plot the cross correlation between *x* and *y*.

Parameters x : sequence of scalars of length n

y : sequence of scalars of length n

hold : boolean, optional, default: True

detrend : callable, optional, default: `mlab.detrend_none`
x is detrended by the `detrend` callable. Default is no normalization.

normed : boolean, optional, default: True
if True, normalize the data by the autocorrelation at the 0-th lag.

usevlines : boolean, optional, default: True
if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.

maxlags : integer, optional, default: 10
number of lags to show. If None, will return all 2 * len(x) - 1 lags.

Returns (lags, c, line, b) : where:
- `lags` are a length 2‘maxlags+1 lag vector.
- `c` is the 2‘maxlags+1 auto correlation vectorI
- `line` is a *Line2D* instance returned by `plot`.
- `b` is the x-axis (none, if plot is used).

Other Parameters linestyle : `Line2D` prop, optional, default: None
Only used if usevlines is False.

marker : string, optional, default: 'o'

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

yaxis_date(*tz=None*)
Sets up y-axis ticks and labels that treat the y data as dates.

tz is a timezone string or `tzinfo` instance. Defaults to rc value.

yaxis_inverted()
Returns *True* if the y-axis is inverted.

zorder = 0

CHAPTER

FORTYFOUR

AXIS

44.1 matplotlib.axis

Classes for the ticks and x and y axis

class matplotlib.axis.**Axis**(*axes*, *pickradius=15*)

Bases: *matplotlib.artist.Artist*

Public attributes

- axes.transData - transform data coords to display coords
- axes.transAxes - transform axis coords to display coords
- labelpad - number of points between the axis and its label

Init the axis with the parent Axes instance

OFFSETTEXTPAD = 3

axis_date(*tz=None*)

Sets up x-axis ticks and labels that treat the x data as dates. *tz* is a tzinfo instance or a timezone string. This timezone is used to create date labels.

cla()

clear the current axis

convert_units(*x*)

draw(*artist*, *renderer*, **args*, ***kwargs*)

Draw the axis lines, grid lines, tick lines and labels

get_children()

get_data_interval()

return the Interval instance for this axis data limits

get_gridlines()

Return the grid lines as a list of Line2D instance

get_label()

Return the axis label as a Text instance

get_label_text()
Get the text of the label

get_major_formatter()
Get the formatter of the major ticker

get_major_locator()
Get the locator of the major ticker

get_major_ticks(*numticks=None*)
get the tick instances; grow as necessary

get_majorticklabels()
Return a list of Text instances for the major ticklabels

get_majorticklines()
Return the major tick lines as a list of Line2D instances

get_majorticklocs()
Get the major tick locations in data coordinates as a numpy array

get_minor_formatter()
Get the formatter of the minor ticker

get_minor_locator()
Get the locator of the minor ticker

get_minor_ticks(*numticks=None*)
get the minor tick instances; grow as necessary

get_minorticklabels()
Return a list of Text instances for the minor ticklabels

get_minorticklines()
Return the minor tick lines as a list of Line2D instances

get_minorticklocs()
Get the minor tick locations in data coordinates as a numpy array

get_offset_text()
Return the axis offsetText as a Text instance

get_pickradius()
Return the depth of the axis used by the picker

get_scale()

get_smart_bounds()
get whether the axis has smart bounds

get_ticklabel_extents(*renderer*)
Get the extents of the tick labels on either side of the axes.

get_ticklabels(*minor=False*, *which=None*)
Get the x tick labels as a list of `Text` instances.

Parameters minor : bool

If True return the minor ticklabels, else return the major ticklabels

which : None, ('minor', 'major', 'both')

Overrides `minor`.

Selects which ticklabels to return

Returns ret : list

List of `Text` instances.

get_ticklines(*minor=False*)

Return the tick lines as a list of Line2D instances

get_ticklocs(*minor=False*)

Get the tick locations in data coordinates as a numpy array

get_tightbbox(*renderer*)

Return a bounding box that encloses the axis. It only accounts tick labels, axis label, and offsetText.

get_transform()

get_units()

return the units for axis

get_view_interval()

return the Interval instance for this axis view limits

grid(*b=None*, *which='major'*, ***kwargs*)

Set the axis grid on or off; b is a boolean. Use *which* = 'major' | 'minor' | 'both' to set the grid for major or minor ticks.

If *b* is *None* and len(kwargs)==0, toggle the grid state. If *kwargs* are supplied, it is assumed you want the grid on and *b* will be set to True.

kwargs are used to set the line properties of the grids, e.g.,

xax.grid(color='r', linestyle='-', linewidth=2)

have_units()

iter_ticks()

Iterate through all of the major and minor ticks.

limit_range_for_scale(*vmin*, *vmax*)

pan(*numsteps*)

Pan *numsteps* (can be positive or negative)

reset_ticks()

set_clip_path(*clippath*, *transform=None*)

set_data_interval()
set the axis data limits

set_default_intervals()
set the default limits for the axis data and view interval if they are not mutated

set_label_coords(*x*, *y*, *transform=None*)
Set the coordinates of the label. By default, the x coordinate of the y label is determined by the tick label bounding boxes, but this can lead to poor alignment of multiple ylabels if there are multiple axes. Ditto for the y coodinate of the x label.

You can also specify the coordinate system of the label with the transform. If None, the default coordinate system will be the axes coordinate system (0,0) is (left,bottom), (0.5, 0.5) is middle, etc

set_label_text(*label*, *fontdict=None*, ***kwargs*)
Sets the text value of the axis label

ACCEPTS: A string value for the label

set_major_formatter(*formatter*)
Set the formatter of the major ticker

ACCEPTS: A `Formatter` instance

set_major_locator(*locator*)
Set the locator of the major ticker

ACCEPTS: a `Locator` instance

set_minor_formatter(*formatter*)
Set the formatter of the minor ticker

ACCEPTS: A `Formatter` instance

set_minor_locator(*locator*)
Set the locator of the minor ticker

ACCEPTS: a `Locator` instance

set_pickradius(*pickradius*)
Set the depth of the axis used by the picker

ACCEPTS: a distance in points

set_smart_bounds(*value*)
set the axis to have smart bounds

set_tick_params(*which='major'*, *reset=False*, ***kw*)
Set appearance parameters for ticks and ticklabels.

For documentation of keyword arguments, see `matplotlib.axes.Axes.tick_params()`.

set_ticklabels(*ticklabels*, **args*, ***kwargs*)
Set the text values of the tick labels. Return a list of Text instances. Use *kwarg minor=True* to select minor ticks. All other kwargs are used to update the text object properties. As for get_ticklabels, label1 (left or bottom) is affected for a given tick only if its label1On attribute is

True, and similarly for label2. The list of returned label text objects consists of all such label1 objects followed by all such label2 objects.

The input *ticklabels* is assumed to match the set of tick locations, regardless of the state of label1On and label2On.

ACCEPTS: sequence of strings or Text objects

set_ticks(*ticks*, *minor=False*)
Set the locations of the tick marks from sequence ticks

ACCEPTS: sequence of floats

set_units(*u*)
set the units for axis

ACCEPTS: a units tag

set_view_interval(*vmin*, *vmax*, *ignore=False*)

update_units(*data*)
introspect *data* for units converter and update the axis.converter instance if necessary. Return *True* if *data* is registered for unit conversion.

zoom(*direction*)
Zoom in/out on axis; if *direction* is >0 zoom in, else zoom out

class matplotlib.axis.**Tick**(*axes*, *loc*, *label*, *size=None*, *width=None*, *color=None*, *tickdir=None*, *pad=None*, *labelsize=None*, *labelcolor=None*, *zorder=None*, *gridOn=None*, *tick1On=True*, *tick2On=True*, *label1On=True*, *label2On=False*, *major=True*)
Bases: `matplotlib.artist.Artist`

Abstract base class for the axis ticks, grid lines and labels

1 refers to the bottom of the plot for xticks and the left for yticks 2 refers to the top of the plot for xticks and the right for yticks

Publicly accessible attributes:
tick1line a Line2D instance
tick2line a Line2D instance
gridline a Line2D instance
label1 a Text instance
label2 a Text instance
gridOn a boolean which determines whether to draw the tickline
tick1On a boolean which determines whether to draw the 1st tickline
tick2On a boolean which determines whether to draw the 2nd tickline
label1On a boolean which determines whether to draw tick label
label2On a boolean which determines whether to draw tick label

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

apply_tickdir(*tickdir*)
Calculate self._pad and self._tickmarkers

contains(*mouseevent*)
Test whether the mouse event occurred in the Tick marks.

This function always returns false. It is more useful to test if the axis as a whole contains the mouse rather than the set of tick marks.

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_children()

get_loc()
Return the tick location (data coords) as a scalar

get_pad()
Get the value of the tick label pad in points

get_pad_pixels()

get_view_interval()
return the view Interval instance for the axis this tick is ticking

set_clip_path(*clippath*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_label(*s*)
Set the text of ticklabel

ACCEPTS: str

set_label1(*s*)
Set the text of ticklabel

ACCEPTS: str

set_label2(*s*)
Set the text of ticklabel2

ACCEPTS: str

set_pad(*val*)
Set the tick label pad in points

ACCEPTS: float

class `matplotlib.axis.`**Ticker**
Bases: `object`

formatter = None

locator = None

class matplotlib.axis.**XAxis**(*axes*, *pickradius=15*)
Bases: `matplotlib.axis.Axis`

Init the axis with the parent Axes instance

axis_name = 'x'

contains(*mouseevent*)
Test whether the mouse event occured in the x axis.

get_data_interval()
return the Interval instance for this axis data limits

get_label_position()
Return the label position (top or bottom)

get_minpos()

get_text_heights(*renderer*)
Returns the amount of space one should reserve for text above and below the axes. Returns a tuple (above, below)

get_ticks_position()
Return the ticks position (top, bottom, default or unknown)

get_view_interval()
return the Interval instance for this axis view limits

set_data_interval(*vmin*, *vmax*, *ignore=False*)
set the axis data limits

set_default_intervals()
set the default limits for the axis interval if they are not mutated

set_label_position(*position*)
Set the label position (top or bottom)

ACCEPTS: ['top' | 'bottom']

set_ticks_position(*position*)
Set the ticks position (top, bottom, both, default or none) both sets the ticks to appear on both positions, but does not change the tick labels. 'default' resets the tick positions to the default: ticks on both positions, labels at bottom. 'none' can be used if you don't want any ticks. 'none' and 'both' affect only the ticks, not the labels.

ACCEPTS: ['top' | 'bottom' | 'both' | 'default' | 'none']

set_view_interval(*vmin*, *vmax*, *ignore=False*)
If *ignore* is *False*, the order of vmin, vmax does not matter; the original axis orientation will be

preserved. In addition, the view limits can be expanded, but will not be reduced. This method is for mpl internal use; for normal use, see `set_xlim()`.

tick_bottom()
use ticks only on bottom

tick_top()
use ticks only on top

class matplotlib.axis.**XTick**(*axes*, *loc*, *label*, *size=None*, *width=None*, *color=None*, *tickdir=None*, *pad=None*, *labelsize=None*, *labelcolor=None*, *zorder=None*, *gridOn=None*, *tick1On=True*, *tick2On=True*, *label1On=True*, *label2On=False*, *major=True*)
Bases: `matplotlib.axis.Tick`

Contains all the Artists needed to make an x tick - the tick line, the label text and the grid line

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

apply_tickdir(*tickdir*)

get_view_interval()
return the Interval instance for this axis view limits

update_position(*loc*)
Set the location of tick in data coords with scalar *loc*

class matplotlib.axis.**YAxis**(*axes*, *pickradius=15*)
Bases: `matplotlib.axis.Axis`

Init the axis with the parent Axes instance

axis_name = 'y'

contains(*mouseevent*)
Test whether the mouse event occurred in the y axis.

Returns *True* | *False*

get_data_interval()
return the Interval instance for this axis data limits

get_label_position()
Return the label position (left or right)

get_minpos()

get_text_widths(*renderer*)

get_ticks_position()
Return the ticks position (left, right, both or unknown)

get_view_interval()
return the Interval instance for this axis view limits

set_data_interval(*vmin*, *vmax*, *ignore=False*)
set the axis data limits

set_default_intervals()
set the default limits for the axis interval if they are not mutated

set_label_position(*position*)
Set the label position (left or right)

ACCEPTS: ['left' | 'right']

set_offset_position(*position*)

set_ticks_position(*position*)
Set the ticks position (left, right, both, default or none) 'both' sets the ticks to appear on both positions, but does not change the tick labels. 'default' resets the tick positions to the default: ticks on both positions, labels at left. 'none' can be used if you don't want any ticks. 'none' and 'both' affect only the ticks, not the labels.

ACCEPTS: ['left' | 'right' | 'both' | 'default' | 'none']

set_view_interval(*vmin*, *vmax*, *ignore=False*)
If *ignore* is *False*, the order of vmin, vmax does not matter; the original axis orientation will be preserved. In addition, the view limits can be expanded, but will not be reduced. This method is for mpl internal use; for normal use, see `set_ylim()`.

tick_left()
use ticks only on left

tick_right()
use ticks only on right

class matplotlib.axis.**YTick**(*axes*, *loc*, *label*, *size=None*, *width=None*, *color=None*, *tickdir=None*, *pad=None*, *labelsize=None*, *labelcolor=None*, *zorder=None*, *gridOn=None*, *tick1On=True*, *tick2On=True*, *label1On=True*, *label2On=False*, *major=True*)
Bases: `matplotlib.axis.Tick`

Contains all the Artists needed to make a Y tick - the tick line, the label text and the grid line

bbox is the Bound2D bounding box in display coords of the Axes loc is the tick location in data coords size is the tick size in points

apply_tickdir(*tickdir*)

get_view_interval()
return the Interval instance for this axis view limits

update_position(*loc*)
Set the location of tick in data coords with scalar loc

CHAPTER

FORTYFIVE

BACKENDS

45.1 matplotlib.backend_bases

Abstract base classes define the primitives that renderers and graphics contexts must implement to serve as a matplotlib backend

RendererBase An abstract base class to handle drawing/rendering operations.

FigureCanvasBase The abstraction layer that separates the *matplotlib.figure.Figure* from the backend specific details like a user interface drawing area

GraphicsContextBase An abstract base class that provides color, line styles, etc...

Event The base class for all of the matplotlib event handling. Derived classes such as *KeyEvent* and *MouseEvent* store the meta data like keys and buttons pressed, x and y locations in pixel and *Axes* coordinates.

ShowBase The base class for the Show class of each interactive backend; the 'show' callable is then set to Show.__call__, inherited from ShowBase.

ToolContainerBase The base class for the Toolbar class of each interactive backend.

StatusbarBase The base class for the messaging area.

class `matplotlib.backend_bases.`**`CloseEvent`**(*name*, *canvas*, *guiEvent=None*)
Bases: *matplotlib.backend_bases.Event*

An event triggered by a figure being closed

In addition to the *Event* attributes, the following event attributes are defined:

class `matplotlib.backend_bases.`**`DrawEvent`**(*name*, *canvas*, *renderer*)
Bases: *matplotlib.backend_bases.Event*

An event triggered by a draw operation on the canvas

In addition to the *Event* attributes, the following event attributes are defined:
renderer the *RendererBase* instance for the draw event

class `matplotlib.backend_bases.`**`Event`**(*name*, *canvas*, *guiEvent=None*)
Bases: `object`

A matplotlib event. Attach additional attributes as defined in `FigureCanvasBase.mpl_connect()`.
The following attributes are defined and shown with their default values
name the event name
canvas the FigureCanvas instance generating the event
guiEvent the GUI event that triggered the matplotlib event

class `matplotlib.backend_bases.`**`FigureCanvasBase`**(*figure*)
Bases: `object`

The canvas the figure renders into.

Public attributes
figure A `matplotlib.figure.Figure` instance

`blit`(*bbox=None*)
blit the canvas in bbox (default entire canvas)

`button_press_event`(*x*, *y*, *button*, *dblclick=False*, *guiEvent=None*)
Backend derived classes should call this function on any mouse button press. x,y are the canvas coords: 0,0 is lower, left. button and key are as defined in `MouseEvent`.

This method will be call all functions connected to the 'button_press_event' with a `MouseEvent` instance.

`button_release_event`(*x*, *y*, *button*, *guiEvent=None*)
Backend derived classes should call this function on any mouse button release.
x the canvas coordinates where 0=left
y the canvas coordinates where 0=bottom
guiEvent the native UI event that generated the mpl event
This method will be call all functions connected to the 'button_release_event' with a `MouseEvent` instance.

`close_event`(*guiEvent=None*)
This method will be called by all functions connected to the 'close_event' with a `CloseEvent`

`draw`(**args*, ***kwargs*)
Render the `Figure`

`draw_cursor`(*event*)
Draw a cursor in the event.axes if inaxes is not None. Use native GUI drawing for efficiency if possible

`draw_event`(*renderer*)
This method will be call all functions connected to the 'draw_event' with a `DrawEvent`

`draw_idle`(**args*, ***kwargs*)
`draw()` only if idle; defaults to draw but backends can overrride

`enter_notify_event`(*guiEvent=None*, *xy=None*)
Backend derived classes should call this function when entering canvas
guiEvent the native UI event that generated the mpl event
xy the coordinate location of the pointer when the canvas is entered

`events` **= ['resize_event', 'draw_event', 'key_press_event', 'key_release_event', 'button_press_event', 'button_**

filetypes = {'tiff': 'Tagged Image File Format', 'png': 'Portable Network Graphics', 'svg': 'Scalable Vector

fixed_dpi = None

flush_events()
Flush the GUI events for the figure. Implemented only for backends with GUIs.

get_default_filename()
Return a string, which includes extension, suitable for use as a default filename.

classmethod get_default_filetype()
Get the default savefig file format as specified in rcParam `savefig.format`. Returned string excludes period. Overridden in backends that only support a single file type.

classmethod get_supported_filetypes()
Return dict of savefig file formats supported by this backend

classmethod get_supported_filetypes_grouped()
Return a dict of savefig file formats supported by this backend, where the keys are a file type name, such as 'Joint Photographic Experts Group', and the values are a list of filename extensions used for that filetype, such as ['jpg', 'jpeg'].

get_width_height()
Return the figure width and height in points or pixels (depending on the backend), truncated to integers

get_window_title()
Get the title text of the window containing the figure. Return None if there is no window (e.g., a PS backend).

grab_mouse(*ax*)
Set the child axes which are currently grabbing the mouse events. Usually called by the widgets themselves. It is an error to call this if the mouse is already grabbed by another axes.

idle_event(*guiEvent=None*)
Called when GUI is idle.

is_saving()
Returns `True` when the renderer is in the process of saving to a file, rather than rendering for an on-screen buffer.

key_press_event(*key*, *guiEvent=None*)
This method will be call all functions connected to the 'key_press_event' with a *KeyEvent*

key_release_event(*key*, *guiEvent=None*)
This method will be call all functions connected to the 'key_release_event' with a *KeyEvent*

leave_notify_event(*guiEvent=None*)
Backend derived classes should call this function when leaving canvas
guiEvent the native UI event that generated the mpl event

motion_notify_event(*x*, *y*, *guiEvent=None*)
Backend derived classes should call this function on any motion-notify-event.

x the canvas coordinates where 0=left
y the canvas coordinates where 0=bottom
guiEvent the native UI event that generated the mpl event
This method will be call all functions connected to the 'motion_notify_event' with a `MouseEvent` instance.

mpl_connect(*s*, *func*)
Connect event with string *s* to *func*. The signature of *func* is:

```
def func(event)
```

where event is a `matplotlib.backend_bases.Event`. The following events are recognized

- 'button_press_event'
- 'button_release_event'
- 'draw_event'
- 'key_press_event'
- 'key_release_event'
- 'motion_notify_event'
- 'pick_event'
- 'resize_event'
- 'scroll_event'
- 'figure_enter_event',
- 'figure_leave_event',
- 'axes_enter_event',
- 'axes_leave_event'
- 'close_event'

For the location events (button and key press/release), if the mouse is over the axes, the variable `event.inaxes` will be set to the `Axes` the event occurs is over, and additionally, the variables `event.xdata` and `event.ydata` will be defined. This is the mouse location in data coords. See `KeyEvent` and `MouseEvent` for more info.

Return value is a connection id that can be used with `mpl_disconnect()`.

Example usage:

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

mpl_disconnect(*cid*)
Disconnect callback id cid

Example usage:

```
cid = canvas.mpl_connect('button_press_event', on_press)
#...later
canvas.mpl_disconnect(cid)
```

new_timer(**args*, ***kwargs*)
Creates a new backend-specific subclass of `backend_bases.Timer`. This is useful for getting

periodic events through the backend's native event loop. Implemented only for backends with GUIs.

optional arguments:
interval Timer interval in milliseconds
callbacks Sequence of (func, args, kwargs) where func(*args, **kwargs) will be executed by the timer every *interval.*

`onHilite`(*ev*)
Mouse event processor which highlights the artists under the cursor. Connect this to the 'motion_notify_event' using:

```
canvas.mpl_connect('motion_notify_event',canvas.onHilite)
```

`onRemove`(*ev*)
Mouse event processor which removes the top artist under the cursor. Connect this to the 'mouse_press_event' using:

```
canvas.mpl_connect('mouse_press_event',canvas.onRemove)
```

`pick`(*mouseevent*)

`pick_event`(*mouseevent*, *artist*, ***kwargs*)
This method will be called by artists who are picked and will fire off `PickEvent` callbacks registered listeners

`print_figure`(*filename*, *dpi=None*, *facecolor='w'*, *edgecolor='w'*, *orientation='portrait'*, *format=None*, ***kwargs*)
Render the figure to hardcopy. Set the figure patch face and edge colors. This is useful because some of the GUIs have a gray figure face color background and you'll probably want to override this on hardcopy.

Arguments are:
filename can also be a file object on image backends
orientation only currently applies to PostScript printing.
dpi the dots per inch to save the figure in; if None, use savefig.dpi
facecolor the facecolor of the figure
edgecolor the edgecolor of the figure
orientation landscape' | 'portrait' (not supported on all backends)
format when set, forcibly set the file format to save to
bbox_inches Bbox in inches. Only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure. If None, use savefig.bbox
pad_inches Amount of padding around the figure when bbox_inches is 'tight'. If None, use savefig.pad_inches
bbox_extra_artists A list of extra artists that will be considered when the tight bbox is calculated.

`release_mouse`(*ax*)
Release the mouse grab held by the axes, ax. Usually called by the widgets. It is ok to call this even if you ax doesn't have the mouse grab currently.

resize(*w*, *h*)
set the canvas size in pixels

resize_event()
This method will be call all functions connected to the 'resize_event' with a *ResizeEvent*

scroll_event(*x*, *y*, *step*, *guiEvent=None*)
Backend derived classes should call this function on any scroll wheel event. x,y are the canvas coords: 0,0 is lower, left. button and key are as defined in MouseEvent.

This method will be call all functions connected to the 'scroll_event' with a *MouseEvent* instance.

set_window_title(*title*)
Set the title text of the window containing the figure. Note that this has no effect if there is no window (e.g., a PS backend).

start_event_loop(*timeout*)
Start an event loop. This is used to start a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events. This should not be confused with the main GUI event loop, which is always running and has nothing to do with this.

This is implemented only for backends with GUIs.

start_event_loop_default(*timeout=0*)
Start an event loop. This is used to start a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events. This should not be confused with the main GUI event loop, which is always running and has nothing to do with this.

This function provides default event loop functionality based on time.sleep that is meant to be used until event loop functions for each of the GUI backends can be written. As such, it throws a deprecated warning.

Call signature:

```
start_event_loop_default(self,timeout=0)
```

This call blocks until a callback function triggers stop_event_loop() or *timeout* is reached. If *timeout* is <=0, never timeout.

stop_event_loop()
Stop an event loop. This is used to stop a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events.

This is implemented only for backends with GUIs.

stop_event_loop_default()
Stop an event loop. This is used to stop a blocking event loop so that interactive functions, such as ginput and waitforbuttonpress, can wait for events.

Call signature:

```
stop_event_loop_default(self)
```

supports_blit = True

switch_backends(*FigureCanvasClass*)

Instantiate an instance of FigureCanvasClass

This is used for backend switching, e.g., to instantiate a FigureCanvasPS from a FigureCanvasGTK. Note, deep copying is not done, so any changes to one of the instances (e.g., setting figure size or line props), will be reflected in the other

class matplotlib.backend_bases.**FigureManagerBase**(*canvas*, *num*)

Bases: object

Helper class for pyplot mode, wraps everything up into a neat bundle

Public attibutes:
canvas A FigureCanvasBase instance
num The figure number

destroy()

full_screen_toggle()

get_window_title()

Get the title text of the window containing the figure. Return None for non-GUI backends (e.g., a PS backend).

key_press(*event*)

Implement the default mpl key bindings defined at *Navigation Keyboard Shortcuts*

resize(*w*, *h*)

"For gui backends, resize the window (in pixels).

set_window_title(*title*)

Set the title text of the window containing the figure. Note that this has no effect for non-GUI backends (e.g., a PS backend).

show()

For GUI backends, show the figure window and redraw. For non-GUI backends, raise an exception to be caught by show(), for an optional warning.

show_popup(*msg*)

Display message in a popup – GUI only

class matplotlib.backend_bases.**GraphicsContextBase**

Bases: object

An abstract base class that provides color, line styles, etc...

copy_properties(*gc*)

Copy properties from gc to self

dashd = {'solid': (None, None), 'dashdot': (0, (3.0, 5.0, 1.0, 5.0)), 'dashed': (0, (6.0, 6.0)), 'dotted': (0, (1.0, 3.0))

`get_alpha()`
Return the alpha value used for blending - not supported on all backends

`get_antialiased()`
Return true if the object should try to do antialiased rendering

`get_capstyle()`
Return the capstyle as a string in ('butt', 'round', 'projecting')

`get_clip_path()`
Return the clip path in the form (path, transform), where path is a *Path* instance, and transform is an affine transform to apply to the path before clipping.

`get_clip_rectangle()`
Return the clip rectangle as a *Bbox* instance

`get_dashes()`
Return the dash information as an offset dashlist tuple.

The dash list is a even size list that gives the ink on, ink off in pixels.

See p107 of to PostScript BLUEBOOK for more info.

Default value is None

`get_forced_alpha()`
Return whether the value given by get_alpha() should be used to override any other alpha-channel values.

`get_gid()`
Return the object identifier if one is set, None otherwise.

`get_hatch()`
Gets the current hatch style

`get_hatch_path(density=6.0)`
Returns a Path for the current hatch.

`get_joinstyle()`
Return the line join style as one of ('miter', 'round', 'bevel')

`get_linestyle(style)`
Return the linestyle: one of ('solid', 'dashed', 'dashdot', 'dotted').

`get_linewidth()`
Return the line width in points as a scalar

`get_rgb()`
returns a tuple of three or four floats from 0-1.

`get_sketch_params()`
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:
- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.

- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

get_url()
returns a url if one is set, None otherwise

restore()
Restore the graphics context from the stack - needed only for backends that save graphics contexts on a stack

set_alpha(*alpha*)
Set the alpha value used for blending - not supported on all backends. If `alpha=None` (the default), the alpha components of the foreground and fill colors will be used to set their respective transparencies (where applicable); otherwise, `alpha` will override them.

set_antialiased(*b*)
True if object should be drawn with antialiased rendering

set_capstyle(*cs*)
Set the capstyle as a string in ('butt', 'round', 'projecting')

set_clip_path(*path*)
Set the clip path and transformation. Path should be a *TransformedPath* instance.

set_clip_rectangle(*rectangle*)
Set the clip rectangle with sequence (left, bottom, width, height)

set_dashes(*dash_offset*, *dash_list*)
Set the dash style for the gc.
dash_offset is the offset (usually 0).
dash_list specifies the on-off sequence as points. `(None, None)` specifies a solid line

set_foreground(*fg*, *isRGBA=False*)
Set the foreground color. fg can be a MATLAB format string, a html hex color string, an rgb or rgba unit tuple, or a float between 0 and 1. In the latter case, grayscale is used.

If you know fg is rgba, set `isRGBA=True` for efficiency.

set_gid(*id*)
Sets the id.

set_graylevel(*frac*)
Set the foreground color to be a gray level with *frac*

set_hatch(*hatch*)
Sets the hatch style for filling

set_joinstyle(*js*)
Set the join style to be one of ('miter', 'round', 'bevel')

set_linestyle(*style*)
Set the linestyle to be one of ('solid', 'dashed', 'dashdot', 'dotted'). One may specify customized dash styles by providing a tuple of (offset, dash pairs). For example, the predefiend linestyles have following values.:
'dashed' : (0, (6.0, 6.0)), 'dashdot' : (0, (3.0, 5.0, 1.0, 5.0)), 'dotted' : (0, (1.0, 3.0)),

set_linewidth(*w*)
Set the linewidth in points

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

set_url(*url*)
Sets the url for links in compatible backends

class `matplotlib.backend_bases.`**IdleEvent**(*name*, *canvas*, *guiEvent=None*)
Bases: `matplotlib.backend_bases.Event`

An event triggered by the GUI backend when it is idle – useful for passive animation

class `matplotlib.backend_bases.`**KeyEvent**(*name*, *canvas*, *key*, *x=0*, *y=0*, *guiEvent=None*)
Bases: `matplotlib.backend_bases.LocationEvent`

A key event (key press, key release).

Attach additional attributes as defined in `FigureCanvasBase.mpl_connect()`.

In addition to the `Event` and `LocationEvent` attributes, the following attributes are defined:

key the key(s) pressed. Could be **None**, a single case sensitive ascii character ("g", "G", "#", etc.), a special key ("control", "shift", "f1", "up", etc.) or a combination of the above (e.g., "ctrl+alt+g", "ctrl+alt+G").

Note: Modifier keys will be prefixed to the pressed key and will be in the order "ctrl", "alt", "super".

The exception to this rule is when the pressed key is itself a modifier key, therefore "ctrl+alt" and "alt+control" can both be valid key values.

Example usage:

```
def on_key(event):
    print('you pressed', event.key, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('key_press_event', on_key)
```

class `matplotlib.backend_bases.`**`LocationEvent`**(*name*, *canvas*, *x*, *y*, *guiEvent=None*)
Bases: `matplotlib.backend_bases.Event`

An event that has a screen location

The following additional attributes are defined and shown with their default values.

In addition to the `Event` attributes, the following event attributes are defined:
x x position - pixels from left of canvas
y y position - pixels from bottom of canvas
inaxes the `Axes` instance if mouse is over axes
xdata x coord of mouse in data coords
ydata y coord of mouse in data coords
x, *y* in figure coords, 0,0 = bottom, left

`inaxes` **= None**

`lastevent` **= None**

`x` **= None**

`xdata` **= None**

`y` **= None**

`ydata` **= None**

class `matplotlib.backend_bases.`**`MouseEvent`**(*name*, *canvas*, *x*, *y*, *button=None*, *key=None*, *step=0*, *dblclick=False*, *guiEvent=None*)
Bases: `matplotlib.backend_bases.LocationEvent`
A mouse event ('button_press_event', 'button_release_event', 'scroll_event', 'motion_notify_event').
In addition to the `Event` and `LocationEvent` attributes, the following attributes are defined:
button button pressed None, 1, 2, 3, 'up', 'down' (up and down are used for scroll events). Note that in the nbagg backend, both the middle and right clicks return 3 since right clicking will bring up the context menu in some browsers.
key the key depressed when the mouse event triggered (see `KeyEvent`)

step number of scroll steps (positive for 'up', negative for 'down')
Example usage:

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('button_press_event', on_press)
```

x, y in figure coords, 0,0 = bottom, left button pressed None, 1, 2, 3, 'up', 'down'

button = None

dblclick = None

inaxes = None

step = None

x = None

xdata = None

y = None

ydata = None

class `matplotlib.backend_bases.`**`NavigationToolbar2`**(*canvas*)
Bases: `object`

Base class for the navigation cursor, version 2

backends must implement a canvas that handles connections for 'button_press_event' and 'button_release_event'. See *`FigureCanvasBase.mpl_connect()`* for more information

They must also define
`save_figure()` save the current figure
`set_cursor()` if you want the pointer icon to change
`_init_toolbar()` create your toolbar widget
`draw_rubberband()` **(optional)** draw the zoom to rect "rubberband" rectangle
`press()` **(optional)** whenever a mouse button is pressed, you'll be notified with the event
`release()` **(optional)** whenever a mouse button is released, you'll be notified with the event
`dynamic_update()` **(optional)** dynamically update the window while navigating
`set_message()` **(optional)** display message
`set_history_buttons()` **(optional)** you can change the history back / forward buttons to indicate disabled / enabled state.

That's it, we'll do the rest!

back(**args*)
: move back up the view lim stack

drag_pan(*event*)
: the drag callback in pan/zoom mode

drag_zoom(*event*)
: the drag callback in zoom mode

draw()
: Redraw the canvases, update the locators

draw_rubberband(*event*, *x0*, *y0*, *x1*, *y1*)
: Draw a rectangle rubberband to indicate zoom limits

dynamic_update()

forward(**args*)
: Move forward in the view lim stack

home(**args*)
: Restore the original view

mouse_move(*event*)

pan(**args*)
: Activate the pan/zoom tool. pan with left button, zoom with right

press(*event*)
: Called whenver a mouse button is pressed.

press_pan(*event*)
: the press mouse button in pan/zoom mode callback

press_zoom(*event*)
: the press mouse button in zoom to rect mode callback

push_current()
: push the current view limits and position onto the stack

release(*event*)
: this will be called whenever mouse button is released

release_pan(*event*)
: the release mouse button callback in pan/zoom mode

release_zoom(*event*)
: the release mouse button callback in zoom to rect mode

remove_rubberband()
: Remove the rubberband

save_figure(**args*)
Save the current figure

set_cursor(*cursor*)
Set the current cursor to one of the Cursors enums values

set_history_buttons()
Enable or disable back/forward button

set_message(*s*)
Display a message on toolbar or in status bar

toolitems = ((‘Home’, ‘Reset original view’, ‘home’, ‘home’), (‘Back’, ‘Back to previous view’, ‘back’, ‘back’

update()
Reset the axes stack

zoom(**args*)
Activate zoom to rect mode

exception matplotlib.backend_bases.**NonGuiException**
Bases: Exception

class matplotlib.backend_bases.**PickEvent**(*name*, *canvas*, *mouseevent*, *artist*, *guiEvent=None*, ***kwargs*)
Bases: *matplotlib.backend_bases.Event*

a pick event, fired when the user picks a location on the canvas sufficiently close to an artist.

Attrs: all the *Event* attributes plus
mouseevent the *MouseEvent* that generated the pick
artist the *Artist* picked
other extra class dependent attrs – e.g., a *Line2D* pick may define different extra attributes than a *PatchCollection* pick event
Example usage:

```
line, = ax.plot(rand(100), 'o', picker=5)  # 5 points tolerance

def on_pick(event):
    thisline = event.artist
    xdata, ydata = thisline.get_data()
    ind = event.ind
    print('on pick line:', zip(xdata[ind], ydata[ind]))

cid = fig.canvas.mpl_connect('pick_event', on_pick)
```

class matplotlib.backend_bases.**RendererBase**
Bases: object

An abstract base class to handle drawing/rendering operations.

The following methods must be implemented in the backend for full functionality (though just implementing *draw_path()* alone would give a highly capable backend):

- *draw_path()*

- `draw_image()`
- `draw_gouraud_triangle()`

The following methods *should* be implemented in the backend for optimization reasons:

- `draw_text()`
- `draw_markers()`
- `draw_path_collection()`
- `draw_quad_mesh()`

close_group(*s*)
Close a grouping element with label *s* Is only currently used by `backend_svg`

draw_gouraud_triangle(*gc*, *points*, *colors*, *transform*)
Draw a Gouraud-shaded triangle.

points is a 3x2 array of (x, y) points for the triangle.

colors is a 3x4 array of RGBA colors for each point of the triangle.

transform is an affine transform to apply to the points.

draw_gouraud_triangles(*gc*, *triangles_array*, *colors_array*, *transform*)
Draws a series of Gouraud triangles.

points is a Nx3x2 array of (x, y) points for the trianglex.

colors is a Nx3x4 array of RGBA colors for each point of the triangles.

transform is an affine transform to apply to the points.

draw_image(*gc*, *x*, *y*, *im*)
Draw the image instance into the current axes;
gc a GraphicsContext containing clipping information
x is the distance in pixels from the left hand side of the canvas.
y the distance from the origin. That is, if origin is upper, y is the distance from top. If origin is lower, y is the distance from bottom
im the `matplotlib._image.Image` instance

draw_markers(*gc*, *marker_path*, *marker_trans*, *path*, *trans*, *rgbFace=None*)
Draws a marker at each of the vertices in path. This includes all vertices, including control points on curves. To avoid that behavior, those vertices should be removed before calling this function.
gc the `GraphicsContextBase` instance
marker_trans is an affine transform applied to the marker.
trans is an affine transform applied to the path.
This provides a fallback implementation of draw_markers that makes multiple calls to `draw_path()`. Some backends may want to override this method in order to draw the marker only once and reuse it multiple times.

draw_path(*gc*, *path*, *transform*, *rgbFace=None*)
Draws a `Path` instance using the given affine transform.

draw_path_collection(*gc*, *master_transform*, *paths*, *all_transforms*, *offsets*, *offsetTrans*, *facecolors*, *edgecolors*, *linewidths*, *linestyles*, *antialiaseds*, *urls*, *offset_position*)
Draws a collection of paths selecting drawing properties from the lists *facecolors*, *edgecolors*,

linewidths, *linestyles* and *antialiaseds*. *offsets* is a list of offsets to apply to each of the paths. The offsets in *offsets* are first transformed by *offsetTrans* before being applied. *offset_position* may be either "screen" or "data" depending on the space that the offsets are in.

This provides a fallback implementation of `draw_path_collection()` that makes multiple calls to `draw_path()`. Some backends may want to override this in order to render each set of path data only once, and then reference that path multiple times with the different offsets, colors, styles etc. The generator methods `_iter_collection_raw_paths()` and `_iter_collection()` are provided to help with (and standardize) the implementation across backends. It is highly recommended to use those generators, so that changes to the behavior of `draw_path_collection()` can be made globally.

draw_quad_mesh(*gc*, *master_transform*, *meshWidth*, *meshHeight*, *coordinates*, *offsets*, *offsetTrans*, *facecolors*, *antialiased*, *edgecolors*)
This provides a fallback implementation of `draw_quad_mesh()` that generates paths and then calls `draw_path_collection()`.

draw_tex(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath='TeX!'*, *mtext=None*)

draw_text(*gc*, *x*, *y*, *s*, *prop*, *angle*, *ismath=False*, *mtext=None*)
Draw the text instance
gc the `GraphicsContextBase` instance
x the x location of the text in display coords
y the y location of the text baseline in display coords
s the text string
prop a `matplotlib.font_manager.FontProperties` instance
angle the rotation angle in degrees
mtext a `matplotlib.text.Text` instance
backend implementers note

When you are trying to determine if you have gotten your bounding box right (which is what enables the text layout/alignment to work properly), it helps to change the line in text.py:

```
if 0: bbox_artist(self, renderer)
```

to if 1, and then the actual bounding box will be plotted along with your text.

flipy()
Return true if y small numbers are top for renderer Is used for drawing text (`matplotlib.text`) and images (`matplotlib.image`) only

get_canvas_width_height()
return the canvas width and height in display coords

get_image_magnification()
Get the factor by which to magnify images passed to `draw_image()`. Allows a backend to have images at a different resolution to other artists.

get_texmanager()
return the `matplotlib.texmanager.TexManager` instance

get_text_width_height_descent(*s*, *prop*, *ismath*)
get the width and height, and the offset from the bottom to the baseline (descent), in display coords of the string s with `FontProperties` prop

new_gc()
Return an instance of a `GraphicsContextBase`

open_group(*s*, *gid=None*)
Open a grouping element with label *s*. If *gid* is given, use *gid* as the id of the group. Is only currently used by `backend_svg`.

option_image_nocomposite()
override this method for renderers that do not necessarily always want to rescale and composite raster images. (like SVG, PDF, or PS)

option_scale_image()
override this method for renderers that support arbitrary scaling of image (most of the vector backend).

points_to_pixels(*points*)
Convert points to display units
points a float or a numpy array of float
return points converted to pixels

You need to override this function (unless your backend doesn't have a dpi, e.g., postscript or svg). Some imaging systems assume some value for pixels per inch:

```
points to pixels = points * pixels_per_inch/72.0 * dpi/72.0
```

start_filter()
Used in AggRenderer. Switch to a temporary renderer for image filtering effects.

start_rasterizing()
Used in MixedModeRenderer. Switch to the raster renderer.

stop_filter(*filter_func*)
Used in AggRenderer. Switch back to the original renderer. The contents of the temporary renderer is processed with the *filter_func* and is drawn on the original renderer as an image.

stop_rasterizing()
Used in MixedModeRenderer. Switch back to the vector renderer and draw the contents of the raster renderer as an image on the vector renderer.

strip_math(*s*)

class `matplotlib.backend_bases.`**ResizeEvent**(*name*, *canvas*)
Bases: `matplotlib.backend_bases.Event`

An event triggered by a canvas resize

In addition to the `Event` attributes, the following event attributes are defined:
width width of the canvas in pixels
height height of the canvas in pixels

class `matplotlib.backend_bases.`**`ShowBase`**

Bases: `object`

Simple base class to generate a show() callable in backends.

Subclass must override mainloop() method.

`mainloop`()

class `matplotlib.backend_bases.`**`StatusbarBase`**(*toolmanager*)

Bases: `object`

Base class for the statusbar

`set_message`(*s*)

Display a message on toolbar or in status bar

Parameters s : str

Message text

class `matplotlib.backend_bases.`**`TimerBase`**(*interval=None*, *callbacks=None*)

Bases: `object`

A base class for providing timer events, useful for things animations. Backends need to implement a few specific methods in order to use their own timing mechanisms so that the timer events are integrated into their event loops.

Mandatory functions that must be implemented:

- `_timer_start`: Contains backend-specific code for starting the timer
- `_timer_stop`: Contains backend-specific code for stopping the timer

Optional overrides:

- `_timer_set_single_shot`: Code for setting the timer to single shot operating mode, if supported by the timer object. If not, the `Timer` class itself will store the flag and the `_on_timer` method should be overridden to support such behavior.
- `_timer_set_interval`: Code for setting the interval on the timer, if there is a method for doing so on the timer object.
- `_on_timer`: This is the internal function that any timer object should call, which will handle the task of running all callbacks that have been set.

Attributes:

- *`interval`*: The time between timer events in milliseconds. Default is 1000 ms.
- *`single_shot`*: Boolean flag indicating whether this timer should operate as single shot (run once and then stop). Defaults to `False`.
- `callbacks`: Stores list of (func, args) tuples that will be called upon timer events. This list can be manipulated directly, or the functions *`add_callback`* and *`remove_callback`* can be used.

`add_callback`(*func*, **args*, ***kwargs*)

Register `func` to be called by timer when the event fires. Any additional arguments provided will be passed to `func`.

`interval`

`remove_callback`(*func*, **args*, ***kwargs*)

Remove `func` from list of callbacks. `args` and `kwargs` are optional and used to distinguish between copies of the same function registered to be called with different arguments.

single_shot

start(*interval=None*)

Start the timer object. *interval* is optional and will be used to reset the timer interval first if provided.

stop()

Stop the timer.

class matplotlib.backend_bases.**ToolContainerBase**(*toolmanager*)

Bases: `object`

Base class for all tool containers, e.g. toolbars.

Attributes

toolmanager	(`ToolManager` object that holds the tools that) this `ToolContainer` wants to communicate with.

add_tool(*tool*, *group*, *position=-1*)

Adds a tool to this container

Parameters tool : tool_like

The tool to add, see `ToolManager.get_tool`.

group : str

The name of the group to add this tool to.

position : int (optional)

The position within the group to place this tool. Defaults to end.

add_toolitem(*name*, *group*, *position*, *image*, *description*, *toggle*)

Add a toolitem to the container

This method must get implemented per backend

The callback associated with the button click event, must be **EXACTLY** `self.trigger_tool(name)`

Parameters name : string

Name of the tool to add, this gets used as the tool's ID and as the default label of the buttons

group : String

Name of the group that this tool belongs to

position : Int

Position of the tool within its group, if -1 it goes at the End

image_file : String

Filename of the image for the button or `None`

description : String

Description of the tool, used for the tooltips

toggle : Bool

- `True` : The button is a toggle (change the pressed/unpressed state between consecutive clicks)
- `False` : The button is a normal button (returns to unpressed state after release)

remove_toolitem(*name*)

Remove a toolitem from the `ToolContainer`

This method must get implemented per backend

Called when `ToolManager` emits a `tool_removed_event`

Parameters name : string
Name of the tool to remove

toggle_toolitem(*name*, *toggled*)

Toggle the toolitem without firing event

Parameters name : String
Id of the tool to toggle

toggled : bool
Whether to set this tool as toggled or not.

trigger_tool(*name*)

Trigger the tool

Parameters name : String
Name(id) of the tool triggered from within the container

`matplotlib.backend_bases.`**get_registered_canvas_class**(*format*)

Return the registered default canvas for given file format. Handles deferred import of required backend.

`matplotlib.backend_bases.`**key_press_handler**(*event*, *canvas*, *toolbar=None*)

Implement the default mpl key bindings for the canvas and toolbar described at *Navigation Keyboard Shortcuts*

event a `KeyEvent` instance
canvas a `FigureCanvasBase` instance
toolbar a `NavigationToolbar2` instance

`matplotlib.backend_bases.`**register_backend**(*format*, *backend*, *description=None*)

Register a backend for saving to a given file format.
format [str] File extention
backend [module string or canvas class] Backend for handling file output
description [str, optional] Description of the file type. Defaults to an empty string

45.2 matplotlib.backend_managers

ToolManager Class that makes the bridge between user interaction (key press, toolbar clicks, ..) and the actions in response to the user inputs.

class `matplotlib.backend_managers.`**ToolEvent**(*name*, *sender*, *tool*, *data=None*)

Bases: `object`

Event for tool manipulation (add/remove)

class `matplotlib.backend_managers.ToolManager`(*canvas*)

Bases: `object`

Helper class that groups all the user interactions for a FigureManager

Attributes

manager: `FigureManager`	
keypresslock: `widgets.LockDraw`	`LockDraw` object to know if the `canvas` key_press_event is locked
messagelock: `widgets.LockDraw`	`LockDraw` object to know if the message is available to write

`active_toggle`

Currently toggled tools

`add_tool`(*name*, *tool*, **args*, ***kwargs*)

Add *tool* to `ToolManager`

If successful adds a new event `tool_trigger_name` where **name** is the **name** of the tool, this event is fired everytime the tool is triggered.

Parameters name : str

Name of the tool, treated as the ID, has to be unique

tool : class_like, i.e. str or type

Reference to find the class of the Tool to added.

See also:

`matplotlib.backend_tools.ToolBase` The base class for tools.

Notes

args and kwargs get passed directly to the tools constructor.

`get_tool`(*name*, *warn=True*)

Return the tool object, also accepts the actual tool for convenience

Parameters name : str, ToolBase

Name of the tool, or the tool itself

warn : bool, optional

If this method should give warnings.

`get_tool_keymap`(*name*)

Get the keymap associated with the specified tool

Parameters name : string

Name of the Tool

Returns list : list of keys associated with the Tool

`message_event`(*message*, *sender=None*)

Emit a `ToolManagerMessageEvent`

remove_tool(*name*)
Remove tool from *ToolManager*

Parameters name : string
Name of the Tool

toolmanager_connect(*s*, *func*)
Connect event with string *s* to *func*.

Parameters s : String
Name of the event

The following events are recognized

- 'tool_message_event'
- 'tool_removed_event'
- 'tool_added_event'

For every tool added a new event is created

- 'tool_trigger_TOOLNAME' Where TOOLNAME is the id of the tool.

func : function
Function to be called with signature def func(event)

toolmanager_disconnect(*cid*)
Disconnect callback id *cid*

Example usage:

```
cid = toolmanager.toolmanager_connect('tool_trigger_zoom',
                                      on_press)
#...later
toolmanager.toolmanager_disconnect(cid)
```

tools
Return the tools controlled by *ToolManager*

trigger_tool(*name*, *sender=None*, *canvasevent=None*, *data=None*)
Trigger a tool and emit the tool_trigger_[name] event

Parameters name : string
Name of the tool

sender: object
Object that wishes to trigger the tool

canvasevent : Event
Original Canvas event or None

data : Object
Extra data to pass to the tool when triggering

update_keymap(*name*, **keys*)
Set the keymap to associate with the specified tool

Parameters name : string
Name of the Tool

keys : keys to associate with the Tool

class matplotlib.backend_managers.**ToolManagerMessageEvent**(*name*, *sender*, *message*)
Bases: `object`

Event carrying messages from toolmanager

Messages usually get displayed to the user by the toolbar

class `matplotlib.backend_managers.`**`ToolTriggerEvent`**(*name*, *sender*, *tool*, *canvasevent=None*, *data=None*)

Bases: *`matplotlib.backend_managers.ToolEvent`*

Event to inform that a tool has been triggered

45.3 matplotlib.backend_tools

Abstract base classes define the primitives for Tools. These tools are used by *`matplotlib.backend_managers.ToolManager`*

`ToolBase` Simple stateless tool

`ToolToggleBase` Tool that has two states, only one Toggle tool can be active at any given time for the same *`matplotlib.backend_managers.ToolManager`*

class `matplotlib.backend_tools.`**`AxisScaleBase`**(**args*, ***kwargs*)

Bases: *`matplotlib.backend_tools.ToolToggleBase`*

Base Tool to toggle between linear and logarithmic

`disable`(*event*)

`enable`(*event*)

`trigger`(*sender*, *event*, *data=None*)

class `matplotlib.backend_tools.`**`ConfigureSubplotsBase`**(*toolmanager*, *name*)

Bases: *`matplotlib.backend_tools.ToolBase`*

Base tool for the configuration of subplots

`description` **= 'Configure subplots'**

`image` **= 'subplots.png'**

class `matplotlib.backend_tools.`**`Cursors`**

Bases: `object`

Simple namespace for cursor reference

`HAND` **= 0**

`MOVE` **= 3**

POINTER = 1

SELECT_REGION = 2

class matplotlib.backend_tools.**RubberbandBase**(*toolmanager*, *name*)
Bases: *matplotlib.backend_tools.ToolBase*

Draw and remove rubberband

draw_rubberband(**data*)
Draw rubberband

This method must get implemented per backend

remove_rubberband()
Remove rubberband

This method should get implemented per backend

trigger(*sender*, *event*, *data*)
Call *draw_rubberband* or *remove_rubberband* based on data

class matplotlib.backend_tools.**SaveFigureBase**(*toolmanager*, *name*)
Bases: *matplotlib.backend_tools.ToolBase*

Base tool for figure saving

default_keymap = ['s', 'ctrl+s']

description = 'Save the figure'

image = 'filesave.png'

class matplotlib.backend_tools.**SetCursorBase**(**args*, ***kwargs*)
Bases: *matplotlib.backend_tools.ToolBase*

Change to the current cursor while inaxes

This tool, keeps track of all *ToolToggleBase* derived tools, and calls set_cursor when a tool gets triggered

set_cursor(*cursor*)
Set the cursor

This method has to be implemented per backend

class matplotlib.backend_tools.**ToolBack**(*toolmanager*, *name*)
Bases: *matplotlib.backend_tools.ViewsPositionsBase*

Move back up the view lim stack

default_keymap = ['left', 'c', 'backspace']

description = 'Back to previous view'

image = 'back.png'

class `matplotlib.backend_tools.`**`ToolBase`**(*toolmanager*, *name*)

Bases: `object`

Base tool class

A base tool, only implements *`trigger`* method or not method at all. The tool is instantiated by *`matplotlib.backend_managers.ToolManager`*

Attributes

toolmanager: *`matplotlib.backend_managers.ToolManager`*	ToolManager that controls this Tool
figure: `FigureCanvas`	Figure instance that is affected by this Tool
name: String	Used as **Id** of the tool, has to be unique among tools of the same ToolManager

`default_keymap` **= None**

Keymap to associate with this tool

String: List of comma separated keys that will be used to call this tool when the keypress event of *self.figure.canvas* is emited

`description` **= None**

Description of the Tool

String: If the Tool is included in the Toolbar this text is used as a Tooltip

`destroy`()

Destroy the tool

This method is called when the tool is removed by *`matplotlib.backend_managers.ToolManager.remove_tool`*

`figure`

`image` **= None**

Filename of the image

String: Filename of the image to use in the toolbar. If None, the *`name`* is used as a label in the toolbar button

`name`

Tool Id

`trigger`(*sender*, *event*, *data=None*)

Called when this tool gets used

This method is called by *`matplotlib.backend_managers.ToolManager.trigger_tool`*

Parameters event: 'Event'
The Canvas event that caused this tool to be called

sender: object
Object that requested the tool to be triggered

data: object
Extra data

class `matplotlib.backend_tools.`**`ToolCursorPosition`**(**args*, ***kwargs*)

Bases: `matplotlib.backend_tools.ToolBase`

Send message with the current pointer position

This tool runs in the background reporting the position of the cursor

`send_message`(*event*)

Call `matplotlib.backend_managers.ToolManager.message_event`

class `matplotlib.backend_tools.`**`ToolEnableAllNavigation`**(*toolmanager*, *name*)

Bases: `matplotlib.backend_tools.ToolBase`

Tool to enable all axes for toolmanager interaction

`default_keymap` **= ['a']**

`description` **= 'Enables all axes toolmanager'**

`trigger`(*sender*, *event*, *data=None*)

class `matplotlib.backend_tools.`**`ToolEnableNavigation`**(*toolmanager*, *name*)

Bases: `matplotlib.backend_tools.ToolBase`

Tool to enable a specific axes for toolmanager interaction

`default_keymap` **= (1, 2, 3, 4, 5, 6, 7, 8, 9)**

`description` **= 'Enables one axes toolmanager'**

`trigger`(*sender*, *event*, *data=None*)

class `matplotlib.backend_tools.`**`ToolForward`**(*toolmanager*, *name*)

Bases: `matplotlib.backend_tools.ViewsPositionsBase`

Move forward in the view lim stack

`default_keymap` **= ['right', 'v']**

`description` **= 'Forward to next view'**

image = 'forward.png'

class `matplotlib.backend_tools.`**`ToolFullScreen`**(**args*, ***kwargs*)

Bases: [`matplotlib.backend_tools.ToolToggleBase`]

Tool to toggle full screen

`default_keymap` = ['f', 'ctrl+f']

`description` = 'Toogle Fullscreen mode'

`disable`(*event*)

`enable`(*event*)

class `matplotlib.backend_tools.`**`ToolGrid`**(**args*, ***kwargs*)

Bases: [`matplotlib.backend_tools.ToolToggleBase`]

Tool to toggle the grid of the figure

`default_keymap` = ['g']

`description` = 'Toogle Grid'

`disable`(*event*)

`enable`(*event*)

`trigger`(*sender*, *event*, *data=None*)

class `matplotlib.backend_tools.`**`ToolHome`**(*toolmanager*, *name*)

Bases: [`matplotlib.backend_tools.ViewsPositionsBase`]

Restore the original view lim

`default_keymap` = ['h', 'r', 'home']

`description` = 'Reset original view'

`image` = 'home.png'

class `matplotlib.backend_tools.`**`ToolPan`**(**args*)

Bases: [`matplotlib.backend_tools.ZoomPanBase`]

Pan axes with left mouse, zoom with right

cursor = 3

default_keymap = ['p']

description = 'Pan axes with left mouse, zoom with right'

image = 'move.png'

radio_group = 'default'

class matplotlib.backend_tools.**ToolQuit**(*toolmanager*, *name*)

Bases: *matplotlib.backend_tools.ToolBase*

Tool to call the figure manager destroy method

default_keymap = ['ctrl+w', 'cmd+w']

description = 'Quit the figure'

trigger(*sender*, *event*, *data=None*)

class matplotlib.backend_tools.**ToolToggleBase**(**args*, ***kwargs*)

Bases: *matplotlib.backend_tools.ToolBase*

Toggleable tool

Every time it is triggered, it switches between enable and disable

cursor = None

Cursor to use when the tool is active

disable(*event=None*)

Disable the toggle tool

trigger call this method when *toggled* is True.

This can happen in different circumstances

- Click on the toolbar tool button
- Call to *matplotlib.backend_managers.ToolManager.trigger_tool*
- Another *ToolToggleBase* derived tool is triggered (from the same `ToolManager`)

enable(*event=None*)

Enable the toggle tool

trigger calls this method when *toggled* is False

radio_group = None

Attribute to group 'radio' like tools (mutually exclusive)

String that identifies the group or **None** if not belonging to a group

toggled
State of the toggled tool

trigger(*sender*, *event*, *data=None*)
Calls *enable* or *disable* based on *toggled* value

class matplotlib.backend_tools.**ToolViewsPositions**(**args*, ***kwargs*)
Bases: *matplotlib.backend_tools.ToolBase*

Auxiliary Tool to handle changes in views and positions

Runs in the background and should get used by all the tools that need to access the figure's history of views and positions, e.g.

- *ToolZoom*
- *ToolPan*
- *ToolHome*
- *ToolBack*
- *ToolForward*

add_figure()
Add the current figure to the stack of views and positions

back()
Back one step in the stack of views and positions

clear(*figure*)
Reset the axes stack

forward()
Forward one step in the stack of views and positions

home()
Recall the first view and position from the stack

push_current()
push the current view limits and position onto the stack

refresh_locators()
Redraw the canvases, update the locators

update_view()
Update the viewlim and position from the view and position stack for each axes

class matplotlib.backend_tools.**ToolXScale**(**args*, ***kwargs*)
Bases: *matplotlib.backend_tools.AxisScaleBase*

Tool to toggle between linear and logarithmic scales on the X axis

default_keymap = ['k', 'L']

description = 'Toogle Scale X axis'

set_scale(*ax*, *scale*)

class `matplotlib.backend_tools.`**`ToolYScale`**(**args*, ***kwargs*)

Bases: `matplotlib.backend_tools.AxisScaleBase`

Tool to toggle between linear and logarithmic scales on the Y axis

`default_keymap` = ['l']

`description` = 'Toogle Scale Y axis'

`set_scale`(*ax*, *scale*)

class `matplotlib.backend_tools.`**`ToolZoom`**(**args*)

Bases: `matplotlib.backend_tools.ZoomPanBase`

Zoom to rectangle

`cursor` = 2

`default_keymap` = ['o']

`description` = 'Zoom to rectangle'

`image` = 'zoom_to_rect.png'

`radio_group` = 'default'

class `matplotlib.backend_tools.`**`ViewsPositionsBase`**(*toolmanager*, *name*)

Bases: `matplotlib.backend_tools.ToolBase`

Base class for `ToolHome`, `ToolBack` and `ToolForward`

`trigger`(*sender*, *event*, *data=None*)

class `matplotlib.backend_tools.`**`ZoomPanBase`**(**args*)

Bases: `matplotlib.backend_tools.ToolToggleBase`

Base class for `ToolZoom` and `ToolPan`

`disable`(*event*)

Release the canvas and disconnect press/release events

`enable`(*event*)

Connect press/release events and lock the canvas

`scroll_zoom`(*event*)

`trigger`(*sender*, *event*, *data=None*) `matplotlib.backend_tools.add_tools_to_container`(*container, tools=[['navigation', ['home', 'back', 'forward']], ['zoompan', ['pan', 'zoom']], ['layout', ['subplots']], ['io', ['save']]]*)

Add multiple tools to the container.

Parameters container: Container

`backend_bases.ToolContainerBase` object that will get the tools added

tools : list, optional

List in the form [[group1, [tool1, tool2 ...]], [group2, [...]]] Where the tools given by tool1, and tool2 will display in group1. See `add_tool` for details.

`matplotlib.backend_tools.add_tools_to_manager`(*toolmanager, tools={'position': <class 'matplotlib.backend_tools.ToolCursorPosition'>, 'allnav': <class 'matplotlib.backend_tools.ToolEnableAllNavigation'>, 'pan': <class 'matplotlib.backend_tools.ToolPan'>, 'back': <class 'matplotlib.backend_tools.ToolBack'>, 'xscale': <class 'matplotlib.backend_tools.ToolXScale'>, 'grid': <class 'matplotlib.backend_tools.ToolGrid'>, 'nav': <class 'matplotlib.backend_tools.ToolEnableNavigation'>, 'cursor': 'ToolSetCursor', 'home': <class 'matplotlib.backend_tools.ToolHome'>, 'viewpos': <class 'matplotlib.backend_tools.ToolViewsPositions'>, 'yscale': <class 'matplotlib.backend_tools.ToolYScale'>, 'quit': <class 'matplotlib.backend_tools.ToolQuit'>, 'zoom': <class 'matplotlib.backend_tools.ToolZoom'>, 'save': 'ToolSaveFigure', 'fullscreen': <class 'matplotlib.backend_tools.ToolFullScreen'>, 'subplots': 'ToolConfigureSubplots', 'rubberband': 'ToolRubberband', 'forward': <class 'matplotlib.backend_tools.ToolForward'>}*)

Add multiple tools to `ToolManager`

Parameters toolmanager: ToolManager

backend_managers.ToolManager object that will get the tools added

tools : {str: class_like}, optional
The tools to add in a {name: tool} dict, see add_tool for more info.

matplotlib.backend_tools.**default_toolbar_tools = [['navigation', ['home', 'back', 'forward']], ['zoompan', [**
Default tools in the toolbar

matplotlib.backend_tools.**default_tools = {'position': <class 'matplotlib.backend_tools.ToolCursorPosition'>**
Default tools

45.4 matplotlib.backends.backend_gtkagg

TODO We'll add this later, importing the gtk backends requires an active X-session, which is not compatible with cron jobs.

45.5 matplotlib.backends.backend_qt4agg

Render to qt from agg

matplotlib.backends.backend_qt4agg.**FigureCanvas**
alias of *FigureCanvasQTAgg*

class matplotlib.backends.backend_qt4agg.**FigureCanvasQTAgg**(*figure*)
Bases: *matplotlib.backends.backend_qt4agg.FigureCanvasQTAggBase*, matplotlib.backends.backend_qt4.FigureCanvasQT, matplotlib.backends.backend_agg.FigureCanva

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...

Public attribute
figure - A Figure instance

class matplotlib.backends.backend_qt4agg.**FigureCanvasQTAggBase**(*figure*)
Bases: matplotlib.backends.backend_qt5agg.FigureCanvasQTAggBase

matplotlib.backends.backend_qt4agg.**new_figure_manager**(*num*, **args*, ***kwargs*)
Create a new figure manager instance

matplotlib.backends.backend_qt4agg.**new_figure_manager_given_figure**(*num*, *figure*)
Create a new figure manager instance for the given figure.

45.6 matplotlib.backends.backend_wxagg

matplotlib.backends.backend_wxagg.**FigureCanvas**
alias of *FigureCanvasWxAgg*

class matplotlib.backends.backend_wxagg.**FigureCanvasWxAgg**(*parent*, *id*, *figure*)
Bases: matplotlib.backends.backend_agg.FigureCanvasAgg, matplotlib.backends.backend_wx.FigureCanvasWx

The FigureCanvas contains the figure and does event handling.

In the wxPython backend, it is derived from wxPanel, and (usually) lives inside a frame instantiated by a FigureManagerWx. The parent window probably implements a wxSizer to control the displayed control size - but we give a hint as to our preferred minimum size.

Initialise a FigureWx instance.
- Initialise the FigureCanvasBase and wxPanel parents.
- Set event handlers for: EVT_SIZE (Resize event) EVT_PAINT (Paint event)

blit(*bbox=None*)
Transfer the region of the agg buffer defined by bbox to the display. If bbox is None, the entire buffer is transferred.

draw(*drawDC=None*)
Render the figure using agg.

filetypes = {'tiff': 'Tagged Image File Format', 'png': 'Portable Network Graphics', 'svg': 'Scalable Vector

print_figure(*filename, *args, **kwargs*)

class matplotlib.backends.backend_wxagg.**FigureFrameWxAgg**(*num, fig*)
Bases: matplotlib.backends.backend_wx.FigureFrameWx

get_canvas(*fig*)

class matplotlib.backends.backend_wxagg.**NavigationToolbar2WxAgg**(*canvas*)
Bases: matplotlib.backends.backend_wx.NavigationToolbar2Wx

get_canvas(*frame, fig*)

matplotlib.backends.backend_wxagg.**new_figure_manager**(*num, *args, **kwargs*)
Create a new figure manager instance

matplotlib.backends.backend_wxagg.**new_figure_manager_given_figure**(*num, figure*)
Create a new figure manager instance for the given figure.

45.7 matplotlib.backends.backend_pdf

A PDF matplotlib backend Author: Jouni K Seppänen <jks@iki.fi>

matplotlib.backends.backend_pdf.**FigureCanvas**
alias of *FigureCanvasPdf*

class matplotlib.backends.backend_pdf.**FigureCanvasPdf**(*figure*)
Bases: *matplotlib.backend_bases.FigureCanvasBase*

The canvas the figure renders into. Calls the draw and print fig methods, creates the renderers, etc...

Public attribute
figure - A Figure instance

class `matplotlib.backends.backend_pdf.`**`Name`**`(`*name*`)`

Bases: `object`

PDF name object.

class `matplotlib.backends.backend_pdf.`**`Operator`**`(`*op*`)`

Bases: `object`

PDF operator object.

class `matplotlib.backends.backend_pdf.`**`PdfFile`**`(`*filename*`)`

Bases: `object`

PDF file object.

`alphaState``(`*alpha*`)`

Return name of an ExtGState that sets alpha to the given value

`embedTTF``(`*filename*, *characters*`)`

Embed the TTF font from the named file into the document.

`fontName``(`*fontprop*`)`

Select a font based on fontprop and return a name suitable for Op.selectfont. If fontprop is a string, it will be interpreted as the filename (or dvi name) of the font.

`imageObject``(`*image*`)`

Return name of an image XObject representing the given image.

`markerObject``(`*path*, *trans*, *fill*, *stroke*, *lw*, *joinstyle*, *capstyle*`)`

Return name of a marker XObject representing the given path.

`reserveObject``(`*name=''*`)`

Reserve an ID for an indirect object. The name is used for debugging in case we forget to print out the object with writeObject.

`writeInfoDict``()`

Write out the info dictionary, checking it for good form

`writeTrailer``()`

Write out the PDF trailer.

`writeXref``()`

Write out the xref table.

class `matplotlib.backends.backend_pdf.`**`PdfPages`**`(`*filename*, *keep_empty=True*`)`

Bases: `object`

A multi-page PDF file.

Notes

In reality `PdfPages` is a thin wrapper around `PdfFile`, in order to avoid confusion when using `savefig()` and forgetting the format argument.

Examples

```
>>> import matplotlib.pyplot as plt
>>> # Initialize:
>>> with PdfPages('foo.pdf') as pdf:
...     # As many times as you like, create a figure fig and save it:
...     fig = plt.figure()
...     pdf.savefig(fig)
...     # When no figure is specified the current figure is saved
...     pdf.savefig()
```

Create a new PdfPages object.

Parameters filename: str

Plots using `PdfPages.savefig()` will be written to a file at this location. The file is opened at once and any older file with the same name is overwritten.

keep_empty: bool, optional

If set to False, then empty pdf files will be deleted automatically when closed.

attach_note(*text, positionRect=[-100, -100, 0, 0]*)

Add a new text note to the page to be saved next. The optional positionRect specifies the position of the new note on the page. It is outside the page per default to make sure it is invisible on printouts.

close()

Finalize this object, making the underlying file a complete PDF file.

get_pagecount()

Returns the current number of pages in the multipage pdf file.

infodict()

Return a modifiable information dictionary object (see PDF reference section 10.2.1 'Document Information Dictionary').

savefig(*figure=None*, ***kwargs*)

Saves a `Figure` to this file as a new page.

Any other keyword arguments are passed to `savefig()`.

Parameters figure: :class:'~matplotlib.figure.Figure' or int, optional

Specifies what figure is saved to file. If not specified, the active figure is saved. If a `Figure` instance is provided, this figure is saved. If an int is specified, the figure instance to save is looked up by number.

class `matplotlib.backends.backend_pdf.`**Reference**(*id*)

Bases: `object`

PDF reference object. Use PdfFile.reserveObject() to create References.

class `matplotlib.backends.backend_pdf.`**Stream**(*id*, *len*, *file*, *extra=None*, *png=None*)

Bases: `object`

PDF stream object.

This has no pdfRepr method. Instead, call begin(), then output the contents of the stream by calling write(), and finally call end().

id: object id of stream; len: an unused Reference object for the length of the stream, or None (to use a memory buffer); file: a PdfFile; extra: a dictionary of extra key-value pairs to include in the stream header; png: if the data is already png compressed, the decode parameters

end()
Finalize stream.

write(*data*)
Write some data on the stream.

class `matplotlib.backends.backend_pdf.`**Verbatim**(*x*)
Bases: `object`

Store verbatim PDF command content for later inclusion in the stream.

`matplotlib.backends.backend_pdf.`**fill**(*strings*, *linelen=75*)
Make one string from sequence of strings, with whitespace in between. The whitespace is chosen to form lines of at most linelen characters, if possible.

`matplotlib.backends.backend_pdf.`**new_figure_manager**(*num*, **args*, ***kwargs*)
Create a new figure manager instance

`matplotlib.backends.backend_pdf.`**new_figure_manager_given_figure**(*num*, *figure*)
Create a new figure manager instance for the given figure.

`matplotlib.backends.backend_pdf.`**pdfRepr**(*obj*)
Map Python objects to PDF syntax.

CHAPTER

FORTYSIX

CBOOK

46.1 matplotlib.cbook

A collection of utility functions and classes. Originally, many (but not all) were from the Python Cookbook – hence the name cbook.

This module is safe to import from anywhere within matplotlib; it imports matplotlib only at runtime.

class matplotlib.cbook.**Bunch**(***kwds*)

Bases: object

Often we want to just collect a bunch of stuff together, naming each item of the bunch; a dictionary's OK for that, but a small do- nothing class is even handier, and prettier to use. Whenever you want to group a few variables:

```
>>> point = Bunch(datum=2, squared=4, coord=12)
>>> point.datum

By: Alex Martelli
From: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52308
```

class matplotlib.cbook.**CallbackRegistry**

Bases: object

Handle registering and disconnecting for a set of signals and callbacks:

```
>>> def oneat(x):
...     print('eat', x)
>>> def ondrink(x):
...     print('drink', x)
```

```
>>> from matplotlib.cbook import CallbackRegistry
>>> callbacks = CallbackRegistry()
```

```
>>> id_eat = callbacks.connect('eat', oneat)
>>> id_drink = callbacks.connect('drink', ondrink)
```

```
>>> callbacks.process('drink', 123)
drink 123
```

```
>>> callbacks.process('eat', 456)
eat 456
>>> callbacks.process('be merry', 456) # nothing will be called
>>> callbacks.disconnect(id_eat)
>>> callbacks.process('eat', 456)      # nothing will be called
```

In practice, one should always disconnect all callbacks when they are no longer needed to avoid dangling references (and thus memory leaks). However, real code in matplotlib rarely does so, and due to its design, it is rather difficult to place this kind of code. To get around this, and prevent this class of memory leaks, we instead store weak references to bound methods only, so when the destination object needs to die, the CallbackRegistry won't keep it alive. The Python stdlib weakref module can not create weak references to bound methods directly, so we need to create a proxy object to handle weak references to bound methods (or regular free functions). This technique was shared by Peter Parente on his "Mindtrove" blog.

connect(*s*, *func*)
register *func* to be called when a signal *s* is generated func will be called

disconnect(*cid*)
disconnect the callback registered with callback id *cid*

process(*s*, **args*, ***kwargs*)
process signal *s*. All of the functions registered to receive callbacks on *s* will be called with **args* and ***kwargs*

class `matplotlib.cbook.`**GetRealpathAndStat**
Bases: `object`

class `matplotlib.cbook.`**Grouper**(*init=()*)
Bases: `object`

This class provides a lightweight way to group arbitrary objects together into disjoint sets when a full-blown graph data structure would be overkill.

Objects can be joined using `join()`, tested for connectedness using `joined()`, and all disjoint sets can be retreived by using the object as an iterator.

The objects being joined must be hashable and weak-referenceable.

For example:

```
>>> from matplotlib.cbook import Grouper
>>> class Foo(object):
...     def __init__(self, s):
...         self.s = s
...     def __repr__(self):
...         return self.s
...
>>> a, b, c, d, e, f = [Foo(x) for x in 'abcdef']
>>> grp = Grouper()
>>> grp.join(a, b)
>>> grp.join(b, c)
>>> grp.join(d, e)
>>> sorted(map(tuple, grp))
```

```
[(a, b, c), (d, e)]
>>> grp.joined(a, b)
True
>>> grp.joined(a, c)
True
>>> grp.joined(a, d)
False
```

clean()
Clean dead weak references from the dictionary

get_siblings(*a*)
Returns all of the items joined with *a*, including itself.

join(*a*, **args*)
Join given arguments into the same set. Accepts one or more arguments.

joined(*a*, *b*)
Returns True if *a* and *b* are members of the same set.

remove(*a*)

exception matplotlib.cbook.**IgnoredKeywordWarning**
Bases: UserWarning

A class for issuing warnings about keyword arguments that will be ignored by matplotlib

exception matplotlib.cbook.**MatplotlibDeprecationWarning**
Bases: UserWarning

A class for issuing deprecation warnings for Matplotlib users.

In light of the fact that Python builtin DeprecationWarnings are ignored by default as of Python 2.7 (see link below), this class was put in to allow for the signaling of deprecation, but via UserWarnings which are not ignored by default.

http://docs.python.org/dev/whatsnew/2.7.html#the-future-for-python-2-x

class matplotlib.cbook.**MemoryMonitor**(*nmax=20000*)
Bases: object

clear()

plot(*i0=0*, *isub=1*, *fig=None*)

report(*segments=4*)

xy(*i0=0*, *isub=1*)

class matplotlib.cbook.**Null**(**args*, ***kwargs*)
Bases: object

Null objects always and reliably "do nothing."

class `matplotlib.cbook.`**`RingBuffer`**(*size_max*)

Bases: `object`

class that implements a not-yet-full buffer

`append`(*x*)

append an element at the end of the buffer

`get`()

Return a list of elements from the oldest to the newest.

class `matplotlib.cbook.`**`Sorter`**

Bases: `object`

Sort by attribute or item

Example usage:

```
sort = Sorter()

list = [(1, 2), (4, 8), (0, 3)]
dict = [{'a': 3, 'b': 4}, {'a': 5, 'b': 2}, {'a': 0, 'b': 0},
        {'a': 9, 'b': 9}]

sort(list)       # default sort
sort(list, 1)    # sort by index 1
sort(dict, 'a')  # sort a list of dicts by key 'a'
```

`byAttribute`(*data*, *attributename*, *inplace=1*)

`byItem`(*data*, *itemindex=None*, *inplace=1*)

`sort`(*data*, *itemindex=None*, *inplace=1*)

class `matplotlib.cbook.`**`Stack`**(*default=None*)

Bases: `object`

Implement a stack where elements can be pushed on and you can move back and forth. But no pop. Should mimic home / back / forward in a browser

`back`()

move the position back and return the current element

`bubble`(*o*)

raise *o* to the top of the stack and return *o*. *o* must be in the stack

`clear`()

empty the stack

`empty`()

forward()
move the position forward and return the current element

home()
push the first element onto the top of the stack

push(*o*)
push object onto stack at current position - all elements occurring later than the current position are discarded

remove(*o*)
remove element *o* from the stack

class matplotlib.cbook.**Xlator**
Bases: dict

All-in-one multiple-string-substitution class

Example usage:

```
text = "Larry Wall is the creator of Perl"
adict = {
"Larry Wall" : "Guido van Rossum",
"creator" : "Benevolent Dictator for Life",
"Perl" : "Python",
}

print(multiple_replace(adict, text))

xlat = Xlator(adict)
print(xlat.xlat(text))
```

xlat(*text*)
Translate *text*, returns the modified text.

matplotlib.cbook.**align_iterators**(*func*, **iterables*)
This generator takes a bunch of iterables that are ordered by func It sends out ordered tuples:

```
(func(row), [rows from all iterators matching func(row)])
```

It is used by *matplotlib.mlab.recs_join()* to join record arrays

matplotlib.cbook.**allequal**(*seq*)
Return *True* if all elements of *seq* compare equal. If *seq* is 0 or 1 length, return *True*

matplotlib.cbook.**allpairs**(*x*)
return all possible pairs in sequence *x*

Condensed by Alex Martelli from this thread on c.l.python

matplotlib.cbook.**alltrue**(*seq*)
Return *True* if all elements of *seq* evaluate to *True*. If *seq* is empty, return *False*.

matplotlib.cbook.**boxplot_stats**(*X*, *whis=1.5*, *bootstrap=None*, *labels=None*, *autorange=False*)
Returns list of dictionaries of statistics used to draw a series of box and whisker plots. The Returns

section enumerates the required keys of the dictionary. Users can skip this function and pass a user-defined set of dictionaries to the new `axes.bxp` method instead of relying on MPL to do the calculations.

Parameters X : array-like

Data that will be represented in the boxplots. Should have 2 or fewer dimensions.

whis : float, string, or sequence (default = 1.5)

As a float, determines the reach of the whiskers past the first and third quartiles (e.g., Q3 + whis*IQR, QR = interquartile range, Q3-Q1). Beyond the whiskers, data are considered outliers and are plotted as individual points. This can be set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, `whis` can be the string `'range'` to force the whiskers to the minimum and maximum of the data. In the edge case that the 25th and 75th percentiles are equivalent, `whis` can be automatically set to `'range'` via the `autorange` option.

bootstrap : int, optional

Number of times the confidence intervals around the median should be bootstrapped (percentile method).

labels : array-like, optional

Labels for each dataset. Length must be compatible with dimensions of `X`.

autorange : bool, optional (False)

When `True` and the data are distributed such that the 25th and 75th percentiles are equal, `whis` is set to `'range'` such that the whisker ends are at the minimum and maximum of the data.

Returns bxpstats : list of dict

A list of dictionaries containing the results for each column of data. Keys of each dictionary are the following:

Key	Value Description
label	tick label for the boxplot
mean	arithemetic mean value
med	50th percentile
q1	first quartile (25th percentile)
q3	third quartile (75th percentile)
cilo	lower notch around the median
cihi	upper notch around the median
whislo	end of the lower whisker
whishi	end of the upper whisker
fliers	outliers

Notes

Non-bootstrapping approach to confidence interval uses Gaussian- based asymptotic approximation:

$$\mathrm{med} \pm 1.57 \times \frac{\mathrm{iqr}}{\sqrt{N}} \tag{46.1}$$

General approach from: McGill, R., Tukey, J.W., and Larsen, W.A. (1978) “Variations of Boxplots”, The American Statistician, 32:12-16.

class `matplotlib.cbook.converter`(*missing='Null'*, *missingval=None*)

Bases: `object`

Base class for handling string -> python type with support for missing values

`is_missing`(*s*)

`matplotlib.cbook.dedent`(*s*)

Remove excess indentation from docstring *s*.

Discards any leading blank lines, then removes up to n whitespace characters from each line, where n is the number of leading whitespace characters in the first line. It differs from textwrap.dedent in its deletion of leading blank lines and its use of the first non-blank line to determine the indentation.

It is also faster in most cases.

`matplotlib.cbook.delete_masked_points`(**args*)

Find all masked and/or non-finite points in a set of arguments, and return the arguments with only the unmasked points remaining.

Arguments can be in any of 5 categories:

1. 1-D masked arrays
2. 1-D ndarrays
3. ndarrays with more than one dimension
4. other non-string iterables
5. anything else

The first argument must be in one of the first four categories; any argument with a length differing from that of the first argument (and hence anything in category 5) then will be passed through unchanged.

Masks are obtained from all arguments of the correct length in categories 1, 2, and 4; a point is bad if masked in a masked array or if it is a nan or inf. No attempt is made to extract a mask from categories 2, 3, and 4 if `np.isfinite()` does not yield a Boolean array.

All input arguments that are not passed unchanged are returned as ndarrays after removing the points or rows corresponding to masks in any of the arguments.

A vastly simpler version of this function was originally written as a helper for Axes.scatter().

`matplotlib.cbook.deprecated`(*since*, *message=''*, *name=''*, *alternative=''*, *pending=False*, *obj_type='function'*)

Decorator to mark a function as deprecated.

Parameters since : str

The release at which this API became deprecated. This is required.

message : str, optional

Override the default deprecation message. The format specifier `%(func)s` may be used for the name of the function, and `%(alternative)s` may be used in the deprecation message to insert the name of an alternative to the deprecated function. `%(obj_type)` may be used to insert a friendly name for the type of object being deprecated.

name : str, optional
The name of the deprecated function; if not provided the name is automatically determined from the passed in function, though this is useful in the case of renamed functions, where the new function is just assigned to the name of the deprecated function. For example:

```
def new_function():
    ...
oldFunction = new_function
```

alternative : str, optional
An alternative function that the user may use in place of the deprecated function. The deprecation warning will tell the user about this alternative if provided.

pending : bool, optional
If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.

Examples

Basic example:

```
@deprecated('1.4.0')
def the_function_to_deprecate():
    pass
```

matplotlib.cbook.**dict_delall**(*d*, *keys*)
delete all of the *keys* from the `dict` *d*

matplotlib.cbook.**exception_to_str**(*s=None*)

matplotlib.cbook.**file_requires_unicode**(*x*)
Returns `True` if the given writable file-like object requires Unicode to be written to it.

matplotlib.cbook.**finddir**(*o*, *match*, *case=False*)
return all attributes of *o* which match string in match. if case is True require an exact case match.

matplotlib.cbook.**flatten**(*seq*, *scalarp=<function is_scalar_or_string>*)
Returns a generator of flattened nested containers

For example:

```
>>> from matplotlib.cbook import flatten
>>> l = (('John', ['Hunter']), (1, 23), [[([42, (5, 23)], )]])
>>> print(list(flatten(l)))
['John', 'Hunter', 1, 23, 42, 5, 23]
```

By: Composite of Holger Krekel and Luther Blissett From: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/121294 and Recipe 1.12 in cookbook `matplotlib.cbook.get_label`(*y*, *default_name*)

`matplotlib.cbook.get_recursive_filelist`(*args*)
Recurse all the files and dirs in *args* ignoring symbolic links and return the files as a list of strings

`matplotlib.cbook.get_sample_data`(*fname*, *asfileobj=True*)
Return a sample data file. *fname* is a path relative to the `mpl-data/sample_data` directory. If *asfileobj* is `True` return a file object, otherwise just a file path.

Set the rc parameter examples.directory to the directory where we should look, if sample_data files are stored in a location different than default (which is 'mpl-data/sample_data' at the same level of 'matplotlib' Python module files).

If the filename ends in .gz, the file is implicitly ungzipped.

`matplotlib.cbook.get_split_ind`(*seq*, *N*)
seq is a list of words. Return the index into seq such that:

```
len(' '.join(seq[:ind])<=N
```

.

`matplotlib.cbook.index_of`(*y*)
A helper function to get the index of an input to plot against if x values are not explicitly given.

Tries to get `y.index` (works if this is a pd.Series), if that fails, return np.arange(y.shape[0]).

This will be extended in the future to deal with more types of labeled data.

Parameters y : scalar or array-like
The proposed y-value

Returns x, y : ndarray
The x and y values to plot.

`matplotlib.cbook.is_hashable`(*obj*)
Returns true if *obj* can be hashed

`matplotlib.cbook.is_math_text`(*s*)

`matplotlib.cbook.is_numlike`(*obj*)
return true if *obj* looks like a number

`matplotlib.cbook.is_scalar`(*obj*)
return true if *obj* is not string like and is not iterable

`matplotlib.cbook.is_scalar_or_string`(*val*)
Return whether the given object is a scalar or string like.

`matplotlib.cbook.is_sequence_of_strings`(*obj*)
Returns true if *obj* is iterable and contains strings

`matplotlib.cbook.is_string_like`(*obj*)
Return True if *obj* looks like a string

`matplotlib.cbook.is_writable_file_like(obj)`
: return true if *obj* looks like a file object with a *write* method

`matplotlib.cbook.issubclass_safe(x, klass)`
: return issubclass(x, klass) and return False on a TypeError

`matplotlib.cbook.iterable(obj)`
: return true if *obj* is iterable

`matplotlib.cbook.listFiles(root, patterns='*', recurse=1, return_folders=0)`
: Recursively list files

from Parmar and Martelli in the Python Cookbook

`matplotlib.cbook.local_over_kwdict(local_var, kwargs, *keys)`
: Enforces the priority of a local variable over potentially conflicting argument(s) from a kwargs dict. The following possible output values are considered in order of priority:

local_var > kwargs[keys[0]] > ... > kwargs[keys[-1]]

The first of these whose value is not None will be returned. If all are None then None will be returned. Each key in keys will be removed from the kwargs dict in place.

Parameters local_var: any object
The local variable (highest priority)

kwargs: dict Dictionary of keyword arguments; modified in place
keys: str(s) Name(s) of keyword arguments to process, in descending order of priority

Returns out: any object
Either local_var or one of kwargs[key] for key in keys

Raises IgnoredKeywordWarning
For each key in keys that is removed from kwargs but not used as the output value

class `matplotlib.cbook.maxdict(maxsize)`
: Bases: `dict`

A dictionary with a maximum size; this doesn't override all the relevant methods to contrain size, just setitem, so use with caution

`matplotlib.cbook.mkdirs(newdir, mode=511)`
: make directory *newdir* recursively, and set *mode*. Equivalent to

```
> mkdir -p NEWDIR
> chmod MODE NEWDIR
```

`matplotlib.cbook.mplDeprecation`
: alias of `MatplotlibDeprecationWarning`

`matplotlib.cbook.normalize_kwargs(kw, alias_mapping=None, required=(), forbidden=(), allowed=None)`
: Helper function to normalize kwarg inputs

The order they are resolved are:

1.aliasing
2.required

3.forbidden

4.allowed

This order means that only the canonical names need appear in `allowed`, `forbidden`, `required`

Parameters alias_mapping, dict, optional

A mapping between a canonical name to a list of aliases, in order of precedence from lowest to highest.

If the canonical value is not in the list it is assumed to have the highest priority.

required : iterable, optional

A tuple of fields that must be in kwargs.

forbidden : iterable, optional

A list of keys which may not be in kwargs

allowed : tuple, optional

A tuple of allowed fields. If this not None, then raise if `kw` contains any keys not in the union of `required` and `allowed`. To allow only the required fields pass in `()` for `allowed`

Raises TypeError

To match what python raises if invalid args/kwargs are passed to a callable.

`matplotlib.cbook.`**`onetrue`**(*seq*)

Return *True* if one element of *seq* is *True*. It *seq* is empty, return *False*.

`matplotlib.cbook.`**`pieces`**(*seq*, *num=2*)

Break up the *seq* into *num* tuples

`matplotlib.cbook.`**`popall`**(*seq*)

empty a list

`matplotlib.cbook.`**`print_cycles`**(*objects*, *outstream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *show_progress=False*)

objects A list of objects to find cycles in. It is often useful to pass in gc.garbage to find the cycles that are preventing some objects from being garbage collected.

outstream The stream for output.

show_progress If True, print the number of objects reached as they are found.

`matplotlib.cbook.`**`pts_to_midstep`**(*x*, **args*)

Covert continuous line to pre-steps

Given a set of N points convert to 2 N -1 points which when connected linearly give a step function which changes values at the begining the intervals.

Parameters x : array

The x location of the steps

y1, y2, ... : array

Any number of y arrays to be turned into steps. All must be the same length as `x`

Returns x, y1, y2, .. : array

The x and y values converted to steps in the same order as the input. If the input is length `N`, each of these arrays will be length `2N + 1`

Examples

>> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)

`matplotlib.cbook.pts_to_poststep(x, *args)`
Covert continuous line to pre-steps

Given a set of N points convert to 2 N -1 points which when connected linearly give a step function which changes values at the begining the intervals.

Parameters x : array
The x location of the steps

y1, y2, ... : array
Any number of y arrays to be turned into steps. All must be the same length as `x`

Returns x, y1, y2, .. : array
The x and y values converted to steps in the same order as the input. If the input is length `N`, each of these arrays will be length `2N + 1`

Examples

>> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)

`matplotlib.cbook.pts_to_prestep(x, *args)`
Covert continuous line to pre-steps

Given a set of N points convert to 2 N -1 points which when connected linearly give a step function which changes values at the begining the intervals.

Parameters x : array
The x location of the steps

y1, y2, ... : array
Any number of y arrays to be turned into steps. All must be the same length as `x`

Returns x, y1, y2, .. : array
The x and y values converted to steps in the same order as the input. If the input is length `N`, each of these arrays will be length `2N + 1`

Examples

>> x_s, y1_s, y2_s = pts_to_prestep(x, y1, y2)

`matplotlib.cbook.recursive_remove(path)`

`matplotlib.cbook.report_memory(i=0)`
return the memory consumed by process

`matplotlib.cbook.restrict_dict(d, keys)`
Return a dictionary that contains those keys that appear in both d and keys, with values from d.

matplotlib.cbook.**reverse_dict**(*d*)
reverse the dictionary – may lose data if values are not unique!

matplotlib.cbook.**safe_first_element**(*obj*)

matplotlib.cbook.**safe_masked_invalid**(*x*, *copy=False*)

matplotlib.cbook.**safezip**(**args*)
make sure *args* are equal len before zipping

class matplotlib.cbook.**silent_list**(*type*, *seq=None*)
Bases: `list`

override repr when returning a list of matplotlib artists to prevent long, meaningless output. This is meant to be used for a homogeneous list of a given type

matplotlib.cbook.**simple_linear_interpolation**(*a*, *steps*)

matplotlib.cbook.**soundex**(*name*, *len=4*)
soundex module conforming to Odell-Russell algorithm

matplotlib.cbook.**strip_math**(*s*)
remove latex formatting from mathtext

matplotlib.cbook.**to_filehandle**(*fname*, *flag='rU'*, *return_opened=False*)
fname can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in .gz. *flag* is a read/write flag for `file()`

class matplotlib.cbook.**todate**(*fmt='%Y-%m-%d'*, *missing='Null'*, *missingval=None*)
Bases: `matplotlib.cbook.converter`

convert to a date or None

use a `time.strptime()` format string for conversion

class matplotlib.cbook.**todatetime**(*fmt='%Y-%m-%d'*, *missing='Null'*, *missingval=None*)
Bases: `matplotlib.cbook.converter`

convert to a datetime or None

use a `time.strptime()` format string for conversion

class matplotlib.cbook.**tofloat**(*missing='Null'*, *missingval=None*)
Bases: `matplotlib.cbook.converter`

convert to a float or None

class matplotlib.cbook.**toint**(*missing='Null'*, *missingval=None*)
Bases: `matplotlib.cbook.converter`

convert to an int or None

class matplotlib.cbook.**tostr**(*missing='Null'*, *missingval=''*)
Bases: `matplotlib.cbook.converter`

convert to string or None

`matplotlib.cbook.unicode_safe(s)`

`matplotlib.cbook.unique(x)`
Return a list of unique elements of *x*

`matplotlib.cbook.unmasked_index_ranges(mask, compressed=True)`
Find index ranges where *mask* is *False*.

mask will be flattened if it is not already 1-D.

Returns Nx2 `numpy.ndarray` with each row the start and stop indices for slices of the compressed `numpy.ndarray` corresponding to each of *N* uninterrupted runs of unmasked values. If optional argument *compressed* is *False*, it returns the start and stop indices into the original `numpy.ndarray`, not the compressed `numpy.ndarray`. Returns *None* if there are no unmasked values.

Example:

```
y = ma.array(np.arange(5), mask = [0,0,1,0,0])
ii = unmasked_index_ranges(ma.getmaskarray(y))
# returns array [[0,2,] [2,4,]]

y.compressed()[ii[1,0]:ii[1,1]]
# returns array [3,4,]

ii = unmasked_index_ranges(ma.getmaskarray(y), compressed=False)
# returns array [[0, 2], [3, 5]]

y.filled()[ii[1,0]:ii[1,1]]
# returns array [3,4,]
```

Prior to the transforms refactoring, this was used to support masked arrays in Line2D.

`matplotlib.cbook.violin_stats(X, method, points=100)`
Returns a list of dictionaries of data which can be used to draw a series of violin plots. See the `Returns` section below to view the required keys of the dictionary. Users can skip this function and pass a user-defined set of dictionaries to the `axes.vplot` method instead of using MPL to do the calculations.

Parameters X : array-like
Sample data that will be used to produce the gaussian kernel density estimates. Must have 2 or fewer dimensions.

method : callable
The method used to calculate the kernel density estimate for each column of data. When called via `method(v, coords)`, it should return a vector of the values of the KDE evaluated at the values specified in coords.

points : scalar, default = 100
Defines the number of points to evaluate each of the gaussian kernel density estimates at.

Returns A list of dictionaries containing the results for each column of data.

The dictionaries contain at least the following:

- coords: A list of scalars containing the coordinates this particular kernel density estimate was evaluated at.
- vals: A list of scalars containing the values of the kernel density estimate at each of the coordinates given in `coords`.
- mean: The mean value for this column of data.
- median: The median value for this column of data.
- min: The minimum value for this column of data.
- max: The maximum value for this column of data.

`matplotlib.cbook.`**`warn_deprecated`**(*since*, *message=''*, *name=''*, *alternative=''*, *pending=False*, *obj_type='attribute'*)

Used to display deprecation warning in a standard way.

Parameters since : str

The release at which this API became deprecated.

message : str, optional

Override the default deprecation message. The format specifier `%(func)s` may be used for the name of the function, and `%(alternative)s` may be used in the deprecation message to insert the name of an alternative to the deprecated function. `%(obj_type)` may be used to insert a friendly name for the type of object being deprecated.

name : str, optional

The name of the deprecated function; if not provided the name is automatically determined from the passed in function, though this is useful in the case of renamed functions, where the new function is just assigned to the name of the deprecated function. For example:

```
def new_function():
    ...
oldFunction = new_function
```

alternative : str, optional

An alternative function that the user may use in place of the deprecated function. The deprecation warning will tell the user about this alternative if provided.

pending : bool, optional

If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.

obj_type : str, optional

The object type being deprecated.

Examples

Basic example:

```
# To warn of the deprecation of "matplotlib.name_of_module"
warn_deprecated('1.4.0', name='matplotlib.name_of_module',
                obj_type='module')
```

`matplotlib.cbook.wrap`(*prefix*, *text*, *cols*)

wrap *text* with *prefix* at length *cols*

CHAPTER

FORTYSEVEN

CM (COLORMAP)

47.1 matplotlib.cm

This module provides a large set of colormaps, functions for registering new colormaps and for getting a colormap by name, and a mixin class for adding color mapping functionality.

class matplotlib.cm.**ScalarMappable**(*norm=None*, *cmap=None*)

Bases: object

This is a mixin class to support scalar data to RGBA mapping. The ScalarMappable makes use of data normalization before returning RGBA colors from the given colormap.

Parameters norm : *matplotlib.colors.Normalize* instance

The normalizing object which scales data, typically into the interval [0, 1]. If *None*, *norm* defaults to a *colors.Normalize* object which initializes its scaling based on the first data processed.

cmap : str or *Colormap* instance

The colormap used to map normalized data values to RGBA colors.

add_checker(*checker*)

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

autoscale()

Autoscale the scalar limits on the norm instance using the current array

autoscale_None()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

changed()

Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)

If mappable has changed since the last check, return True; else return False

cmap = None

The Colormap instance of this ScalarMappable.

colorbar = None

The last colorbar associated with this ScalarMappable. May be None.

get_array()
Return the array

get_clim()
return the min, max of the color limits for image scaling

get_cmap()
return the colormap

norm = None
The Normalization instance of this ScalarMappable.

set_array(*A*)
Set the image array from numpy array *A*

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_norm(*norm*)
set the normalization instance

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

matplotlib.cm.**get_cmap**(*name=None*, *lut=None*)
Get a colormap instance, defaulting to rc values if *name* is None.

Colormaps added with `register_cmap()` take precedence over built-in colormaps.

If *name* is a `matplotlib.colors.Colormap` instance, it will be returned.

If *lut* is not None it must be an integer giving the number of entries desired in the lookup table, and *name* must be a standard mpl colormap name.

matplotlib.cm.**register_cmap**(*name=None*, *cmap=None*, *data=None*, *lut=None*)

Add a colormap to the set recognized by `get_cmap()`.

It can be used in two ways:

```
register_cmap(name='swirly', cmap=swirly_cmap)

register_cmap(name='choppy', data=choppydata, lut=128)
```

In the first case, *cmap* must be a `matplotlib.colors.Colormap` instance. The *name* is optional; if absent, the name will be the `name` attribute of the *cmap*.

In the second case, the three arguments are passed to the `LinearSegmentedColormap` initializer, and the resulting colormap is registered.

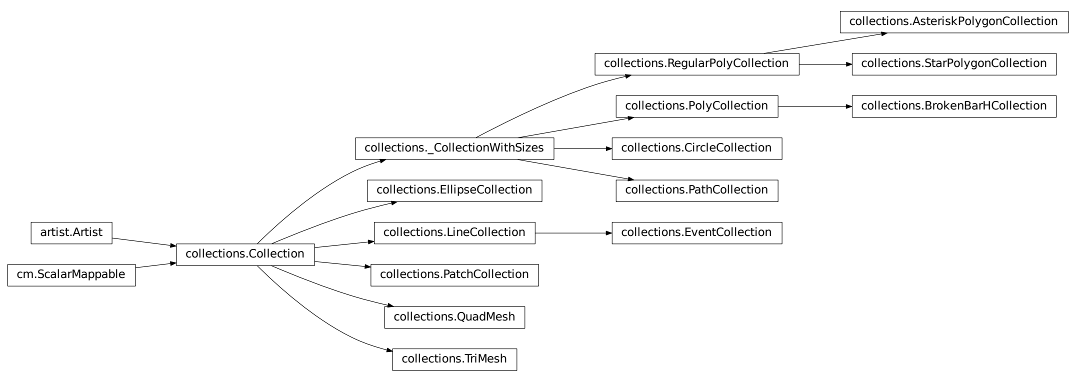
matplotlib.cm.**revcmap**(*data*)

Can only handle specification *data* in dictionary format.

CHAPTER

FORTYEIGHT

COLLECTIONS



48.1 `matplotlib.collections`

Classes for the efficient drawing of large collections of objects that share most properties, e.g., a large number of line segments or polygons.

The classes are not meant to be as flexible as their single element counterparts (e.g., you may not be able to select all line styles) but they are meant to be fast for common use cases (e.g., a large set of solid line segemnts)

class `matplotlib.collections.`**`AsteriskPolygonCollection`**(*numsides*, *rotation=0*, *sizes=(1,)*, ***kwargs*)

Bases: *`matplotlib.collections.RegularPolyCollection`*

Draw a collection of regular asterisks with *numsides* points.
numsides the number of sides of the polygon
rotation the rotation of the polygon in radians
sizes gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:

- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None

- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for `matplotlib.cm.ScalarMappable`)
- *cmap*: None (optional for `matplotlib.cm.ScalarMappable`)

offsets and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

Example: see `examples/dynamic_collection.py` for complete example:

```
offsets = np.random.rand(20,2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0,0,0,1)

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors = facecolors,
    edgecolors = (black,),
    linewidths = (1,),
    offsets = offsets,
    transOffset = ax.transData,
    )
```

add_callback(*func*)
: Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

  Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)
: Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
: Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
: Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
: The `Axes` instance the artist resides in, or *None*.

changed()
: Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
: If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the `Axes` instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the `Figure` instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

`get_numsides()`

`get_offset_position()`
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`get_offset_transform()`

`get_offsets()`
Return the offsets for the collection.

`get_path_effects()`

`get_paths()`

`get_picker()`
Return the picker object used by this artist

`get_pickradius()`

`get_rasterized()`
return True if the artist is to be rasterized

`get_rotation()`

`get_sizes()`
Returns the sizes of the elements in the collection. The value represents the 'area' of the element.

Returns sizes : array

The 'area' of each element.

`get_sketch_params()`
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

`get_snap()`
Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is

- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()

return a dictionary mapping property name -> value for all Artist props

remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)

Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)

A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)

set agg_filter fuction.

set_alpha(*alpha*)

Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)

Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)

Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)

alias for set_antialiased

set_array(*A*)

Set the image array from numpy array *A*

set_axes(*axes*)

Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)

set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)

Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)

Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)

Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)

set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)

Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

set_contains(*picker*)

Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)

alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.
ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)
alias for set_linestyle

set_linewidth(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)
alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths()

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sizes(*sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters sizes : ndarray or None
The size to set for each element of the collection. The value is the 'area' of the element.

dpi : float
The dpi of the canvas. Defaults to 72.0.

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)

Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)

Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_visible(*b*)

Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)

Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale

If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)

Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class `matplotlib.collections.`**BrokenBarHCollection**(*xranges*, *yrange*, ***kwargs*)
Bases: `matplotlib.collections.PolyCollection`

A collection of horizontal bars spanning *yrange* with a sequence of *xranges*.
xranges sequence of (*xmin*, *xwidth*)
yrange *ymin*, *ywidth*

Valid Collection keyword arguments:
- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for `matplotlib.cm.ScalarMappable`)
- *cmap*: None (optional for `matplotlib.cm.ScalarMappable`)

offsets and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

add_callback(*func*)
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The *Axes* instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in self.

match can be
- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the *Axes* instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist`s this :class:`Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the `Figure` instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

`get_linestyle()`

`get_linestyles()`

`get_linewidth()`

`get_linewidths()`

`get_offset_position()`

Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`get_offset_transform()`

`get_offsets()`

Return the offsets for the collection.

`get_path_effects()`

`get_paths()`

`get_picker()`

Return the picker object used by this artist

`get_pickradius()`

`get_rasterized()`

return True if the artist is to be rasterized

`get_sizes()`

Returns the sizes of the elements in the collection. The value represents the 'area' of the element.

Returns sizes : array

The 'area' of each element.

`get_sketch_params()`

Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the `Transform` instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a `Figure`.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

add_callback() For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, **`set_edgecolor()`** For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)

Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)

Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)

alias for set_linestyle

set_linewidth(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)

alias for set_linewidth

set_lw(*lw*)

alias for set_linewidth

set_norm(*norm*)

set the normalization instance

set_offset_position(*offset_position*)

Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths(*verts*, *closed=True*)
This allows one to delay initialization of the vertices.

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sizes(*sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters sizes : ndarray or None
The size to set for each element of the collection. The value is the 'area' of the element.

dpi : float
The dpi of the canvas. Defaults to 72.0.

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_verts(*verts*, *closed=True*)
This allows one to delay initialization of the vertices.

set_verts_and_codes(*verts*, *codes*)
This allows one to initialize vertices with path codes.

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

static span_where(*x*, *ymin*, *ymax*, *where*, ***kwargs*)
Create a BrokenBarHCollection to plot horizontal bars from over the regions in *x* where *where* is True. The bars range on the y-axis from *ymin* to *ymax*

A `BrokenBarHCollection` is returned. *kwargs* are passed on to the collection.

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class `matplotlib.collections.`**CircleCollection**(*sizes*, ***kwargs*)
Bases: `matplotlib.collections._CollectionWithSizes`

A collection of circles, drawn using splines.

sizes Gives the area of the circle in points^2

Valid Collection keyword arguments:

- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for `matplotlib.cm.ScalarMappable`)
- *cmap*: None (optional for `matplotlib.cm.ScalarMappable`)

offsets and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

add_callback(*func*)
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The `Axes` instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be
- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

`format_cursor_data(data)`
Return *cursor data* string formatted.

`get_agg_filter()`
return filter function to be used for agg filter

`get_alpha()`
Return the alpha value used for blending - not supported on all backends

`get_animated()`
Return the artist's animated state

`get_array()`
Return the array

`get_axes()`
Return the `Axes` instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

`get_children()`
Return a list of the child `Artist's this :class:'Artist` contains.

`get_clim()`
return the min, max of the color limits for image scaling

`get_clip_box()`
Return artist clipbox

`get_clip_on()`
Return whether artist uses clipping

`get_clip_path()`
Return artist clip path

`get_cmap()`
return the colormap

`get_contains()`
Return the _contains test used by the artist, or *None* for default.

`get_cursor_data(event)`
Get the cursor data for a given event.

`get_dashes()`

`get_datalim(transData)`

`get_edgecolor()`

`get_edgecolors()`

get_facecolor()

get_facecolors()

get_figure()
Return the *Figure* instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_sizes()
Returns the sizes of the elements in the collection. The value represents the 'area' of the element.

Returns sizes : array
The 'area' of each element.

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:
- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()

Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)

List the children of the artist which contain the mouse event *event*.

is_figure_set()

Returns True if the artist is assigned to a `Figure`.

is_transform_set()

Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()

Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)

call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()

Return *True* if `Artist` is pickable.

properties()

return a dictionary mapping property name -> value for all Artist props

remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)

Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)

A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)

set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

set_facecolor(), *set_edgecolor()* For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the *Figure* instance the artist belongs to.

ACCEPTS: a *matplotlib.figure.Figure* instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)
alias for set_linestyle

set_linewidth(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)
alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths()

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:
- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sizes(*sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters sizes : ndarray or None
The size to set for each element of the collection. The value is the 'area' of the element.

dpi : float
The dpi of the canvas. Defaults to 72.0.

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class `matplotlib.collections.`**Collection**(*edgecolors=None*, *facecolors=None*, *linewidths=None*, *linestyles='solid'*, *antialiaseds=None*, *offsets=None*, *transOffset=None*, *norm=None*, *cmap=None*, *pickradius=5.0*, *hatch=None*, *urls=None*, *offset_position='screen'*, *zorder=1*, ***kwargs*)
Bases: `matplotlib.artist.Artist`, `matplotlib.cm.ScalarMappable`

Base class for Collections. Must be subclassed to be usable.

All properties in a collection must be sequences or scalars; if scalars, they will be converted to sequences. The property of the ith element of the collection is:

```
prop[i % len(props)]
```

Keyword arguments and default values:

- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *offset_position*: 'screen' (default) or 'data'
- *norm*: None (optional for `matplotlib.cm.ScalarMappable`)
- *cmap*: None (optional for `matplotlib.cm.ScalarMappable`)
- *hatch*: None
- *zorder*: 1

offsets and *transOffset* are used to translate the patch after rendering (default no offsets). If offset_position is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

The use of `ScalarMappable` is optional. If the `ScalarMappable` matrix _A is not None (i.e., a call to set_array has been made), at draw time a call to scalar mappable will be made to set the face colors.

Create a Collection

%(Collection)s

add_callback(*func*)

Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()

Autoscale the scalar limits on the norm instance using the current array

autoscale_None()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes

The `Axes` instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the `Axes` instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the `Figure` instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

`get_linestyles()`

`get_linewidth()`

`get_linewidths()`

`get_offset_position()`
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`get_offset_transform()`

`get_offsets()`
Return the offsets for the collection.

`get_path_effects()`

`get_paths()`

`get_picker()`
Return the picker object used by this artist

`get_pickradius()`

`get_rasterized()`
return True if the artist is to be rasterized

`get_sketch_params()`
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

`get_snap()`
Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)

Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)

A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)

set agg_filter fuction.

set_alpha(*alpha*)

Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)

Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)

Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)

alias for set_antialiased

set_array(*A*)

Set the image array from numpy array *A*

set_axes(*axes*)

Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)

set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit = True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)

Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description

The line style.

set_linestyles(*ls*)

alias for set_linestyle

set_linewidth(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)

alias for set_linewidth

set_lw(*lw*)

alias for set_linewidth

set_norm(*norm*)

set the normalization instance

set_offset_position(*offset_position*)

Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)

Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)

set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

`set_paths()`

`set_picker(`*picker*`)`

Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

`set_pickradius(`*pr*`)`

`set_rasterized(`*rasterized*`)`

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

`set_sketch_params(`*scale=None*, *length=None*, *randomness=None*`)`

Sets the sketch parameters.

Parameters scale : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap(`*snap*`)`

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class `matplotlib.collections.`**`EllipseCollection`**(*widths*, *heights*, *angles*, *units='points'*, ***kwargs*)

Bases: `matplotlib.collections.Collection`

A collection of ellipses, drawn using splines.

***widths*: sequence** lengths of first axes (e.g., major axis lengths)

***heights*: sequence** lengths of second axes

***angles*: sequence** angles of first axes, degrees CCW from the X-axis

units: ['points' | 'inches' | 'dots' | 'width' | 'height' | 'x' | 'y' | 'xy']
units in which majors and minors are given; 'width' and 'height' refer to the dimensions of the axes, while 'x' and 'y' refer to the *offsets* data units. 'xy' differs from all others in that the angle as plotted varies with the aspect ratio, and equals the specified angle only when the aspect ratio is unity. Hence it behaves the same as the `Ellipse` with axes.transData as its transform.

Additional kwargs inherited from the base `Collection`:

Valid Collection keyword arguments:

- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for `matplotlib.cm.ScalarMappable`)
- *cmap*: None (optional for `matplotlib.cm.ScalarMappable`)

offsets and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

`add_callback`(*func*)

Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

`add_checker`(*checker*)

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

`autoscale`()

Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

`axes`

The `Axes` instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be
- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the `Axes` instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the `Figure` instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

`get_linestyles()`

`get_linewidth()`

`get_linewidths()`

`get_offset_position()`
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`get_offset_transform()`

`get_offsets()`
Return the offsets for the collection.

`get_path_effects()`

`get_paths()`

`get_picker()`
Return the picker object used by this artist

`get_pickradius()`

`get_rasterized()`
return True if the artist is to be rasterized

`get_sketch_params()`
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:
- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

`get_snap()`
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()

Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)

Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)

A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)

set agg_filter fuction.

set_alpha(*alpha*)

Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)

Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)

Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)

alias for set_antialiased

set_array(*A*)

Set the image array from numpy array *A*

set_axes(*axes*)

Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)

set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, **`set_edgecolor()`** For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit = True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)
alias for set_linestyle

set_linewidth(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)
alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

`set_paths()`

`set_picker(picker)`

Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

`set_pickradius(pr)`

`set_rasterized(rasterized)`

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

`set_sketch_params(scale=None, length=None, randomness=None)`

Sets the sketch parameters.

Parameters scale : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

`set_snap(snap)`

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the *Transform* instance used by this artist.

ACCEPTS: *Transform* instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.**EventCollection**(*positions*, *orientation=None*, *lineoffset=0*, *linelength=1*, *linewidth=None*, *color=None*, *linestyle='solid'*, *antialiased=None*, ***kwargs*)

Bases: `matplotlib.collections.LineCollection`

A collection of discrete events.

An event is a 1-dimensional value, usually the position of something along an axis, such as time or length. Events do not have an amplitude. They are displayed as v

positions a sequence of numerical values or a 1D numpy array. Can be None

orientation **['horizontal' | 'vertical' | None]** defaults to 'horizontal' if not specified or None

lineoffset a single numerical value, corresponding to the offset of the center of the markers from the origin

linelength a single numerical value, corresponding to the total height of the marker (i.e. the marker stretches from lineoffset+linelength/2 to lineoffset-linelength/2). Defaults to 1

linewidth a single numerical value

color must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).

linestyle ['solid' | 'dashed' | 'dashdot' | 'dotted']

antialiased 1 or 2

If *linewidth*, *color*, or *antialiased* is None, they default to their rcParams setting, in sequence form.

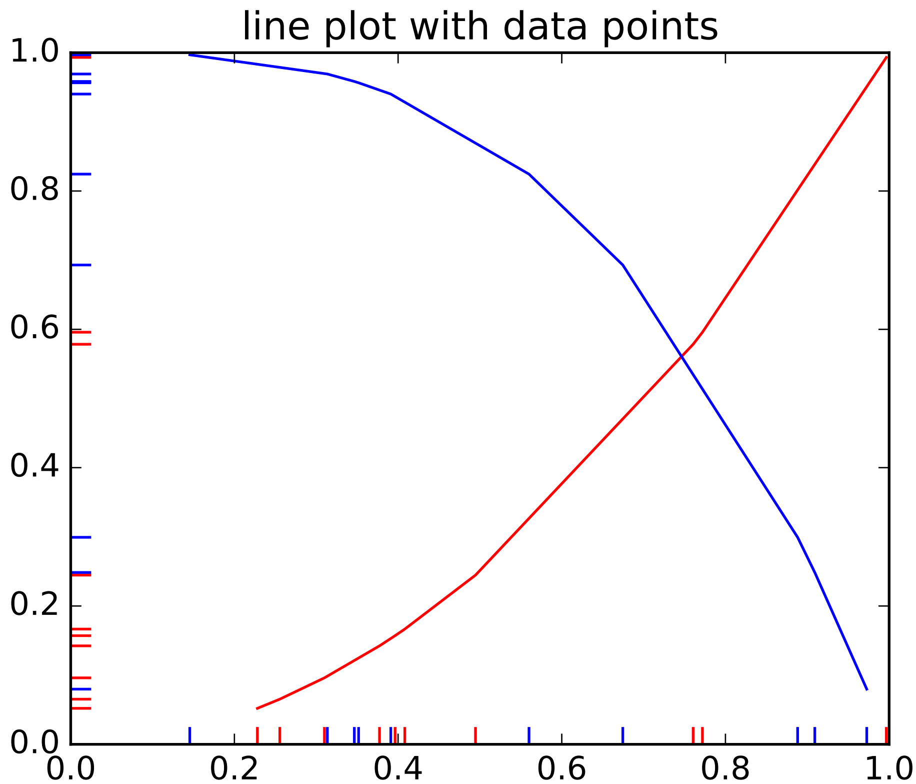
norm None (optional for `matplotlib.cm.ScalarMappable`)

cmap None (optional for `matplotlib.cm.ScalarMappable`)

pickradius is the tolerance for mouse clicks picking a line. The default is 5 pt.

The use of `ScalarMappable` is optional. If the `ScalarMappable` array `_A` is not None (i.e., a call to `set_array()` has been made), at draw time a call to scalar mappable will be made to set the colors.

Example:



add_callback(*func*)
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

add_positions(*position*)
add one or more events at the specified positions

aname = 'Artist'

append_positions(*position*)
add one or more events at the specified positions

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The `Axes` instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

extend_positions(*position*)
add one or more events at the specified positions

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be
- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the `Axes` instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist`s this :class:`Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_color()
get the color of the lines used to mark each event

get_colors()

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the `Figure` instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linelength()
get the length of the lines used to mark each event

get_lineoffset()
get the offset of the lines used to mark each event

get_linestyle()
get the style of the lines used to mark each event ['solid' | 'dashed' | 'dashdot' | 'dotted']

get_linestyles()

get_linewidth()
get the width of the lines used to mark each event

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_orientation()
get the orientation of the event line, may be: ['horizontal' | 'vertical']

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_positions()
return an array containing the floating-point values of the positions

get_rasterized()
return True if the artist is to be rasterized

get_segments()

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:
- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *`Transform`* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *`Figure`*.

is_horizontal()
True if the eventcollection is horizontal, False if vertical

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set the color(s) of the line collection. *c* can be a matplotlib color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linelength(*linelength*)
set the length of the lines used to mark each event

set_lineoffset(*lineoffset*)
set the offset of the lines used to mark each event

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.
ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)
alias for set_linestyle

set_linewidth(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)
alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_orientation(*orientation=None*)
set the orientation of the event line ['horizontal' | 'vertical' | None] defaults to 'horizontal' if not specified or None

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths(*segments*)

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_positions(*positions*)
set the positions of the events to the specified value

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_segments(*segments*)

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_verts(*segments*)

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

switch_orientation()
switch the orientation of the event line, either from vertical to horizontal or vice versus

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class `matplotlib.collections.LineCollection`(*segments*, *linewidths=None*, *colors=None*, *antialiaseds=None*, *linestyles='solid'*, *offsets=None*, *transOffset=None*, *norm=None*, *cmap=None*, *pickradius=5*, *zorder=2*, *facecolors='none'*, ***kwargs*)

Bases: `matplotlib.collections.Collection`

All parameters must be sequences or scalars; if scalars, they will be converted to sequences. The property of the ith line segment is:

```
prop[i % len(props)]
```

i.e., the properties cycle if the `len` of props is less than the number of segments.

segments a sequence of (*line0*, *line1*, *line2*), where:

```
linen = (x0, y0), (x1, y1), ... (xm, ym)
```

or the equivalent numpy array with two columns. Each line can be a different length.

colors must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed).

antialiaseds must be a sequence of ones or zeros

linestyles **['solid' | 'dashed' | 'dashdot' | 'dotted']** a string or dash tuple. The dash tuple is:

```
(offset, onoffseq),
```

where *onoffseq* is an even length tuple of on and off ink in points.

If *linewidths*, *colors*, or *antialiaseds* is None, they default to their rcParams setting, in sequence form.

If *offsets* and *transOffset* are not None, then *offsets* are transformed by *transOffset* and applied after the segments have been transformed to display coordinates.

If *offsets* is not None but *transOffset* is None, then the *offsets* are added to the segments before any transformation. In this case, a single offset can be specified as:

```
offsets=(xo,yo)
```

and this value will be added cumulatively to each successive segment, so as to produce a set of successively offset curves.

norm None (optional for `matplotlib.cm.ScalarMappable`)

cmap None (optional for `matplotlib.cm.ScalarMappable`)

pickradius is the tolerance for mouse clicks picking a line. The default is 5 pt.

zorder The zorder of the LineCollection. Default is 2

facecolors The facecolors of the LineCollection. Default is 'none' Setting to a value other than 'none' will lead to a filled polygon being drawn between points on each line.

The use of `ScalarMappable` is optional. If the `ScalarMappable` array `_A` is not None (i.e., a call to `set_array()` has been made), at draw time a call to scalar mappable will be made to set the colors.

add_callback(*func*)

Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The `Axes` instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the *Axes* instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_color()

get_colors()

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the *Figure* instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_segments()

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:
- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set the color(s) of the line collection. *c* can be a matplotlib color arg (all patches have same color), or a sequence or rgba tuples; if it is a sequence the patches will cycle through the sequence.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit = True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
```

```
-    - horizontal
+    - crossed
x    - crossed diagonal
o    - small circle
O    - large circle
.    - dots
*    - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)

Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)

Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description

The line style.

set_linestyles(*ls*)

alias for set_linestyle

set_linewidth(*lw*)

Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)

alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths(*segments*)

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_segments(*segments*)

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_verts(*segments*)

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, x is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If x is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as x being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class `matplotlib.collections.`**PatchCollection**(*patches*, *match_original=False*, ***kwargs*)
Bases: `matplotlib.collections.Collection`

A generic collection of patches.

This makes it easier to assign a color map to a heterogeneous collection of patches.

This also may improve plotting speed, since PatchCollection will draw faster than a large number of patches.

patches a sequence of Patch objects. This list may include a heterogeneous assortment of different patch types.

match_original If True, use the colors and linewidths of the original patches. If False, new colors may be assigned by providing the standard collection arguments, facecolor, edgecolor, linewidths, norm or cmap.

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

The use of `ScalarMappable` is optional. If the `ScalarMappable` matrix _A is not None (i.e., a call to set_array has been made), at draw time a call to scalar mappable will be made to set the face colors.

add_callback(*func*)
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The *Axes* instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in self.

match can be
- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the *Axes* instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist`s this :class:`Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the *Figure* instance the artist belongs to.

`get_fill()`
return whether fill is set

`get_gid()`
Returns the group id

`get_hatch()`
Return the current hatching pattern

`get_label()`
Get the label used for this artist in the legend.

`get_linestyle()`

`get_linestyles()`

`get_linewidth()`

`get_linewidths()`

`get_offset_position()`
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`get_offset_transform()`

`get_offsets()`
Return the offsets for the collection.

`get_path_effects()`

`get_paths()`

`get_picker()`
Return the picker object used by this artist

`get_pickradius()`

`get_rasterized()`
return True if the artist is to be rasterized

`get_sketch_params()`
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
```

```
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)
alias for set_linestyle

set_linewidth(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)
alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths(*patches*)

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is None, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the Transform instance used by this artist.

ACCEPTS: Transform instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class `matplotlib.collections.`**PathCollection**(*paths*, *sizes=None*, ***kwargs*)
Bases: `matplotlib.collections._CollectionWithSizes`

This is the most basic `Collection` subclass.

paths is a sequence of `matplotlib.path.Path` instances.
Valid Collection keyword arguments:
- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for `matplotlib.cm.ScalarMappable`)
- *cmap*: None (optional for `matplotlib.cm.ScalarMappable`)

offsets and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

add_callback(*func*)
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The `Axes` instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the *Axes* instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the *Figure* instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

`get_hatch()`
Return the current hatching pattern

`get_label()`
Get the label used for this artist in the legend.

`get_linestyle()`

`get_linestyles()`

`get_linewidth()`

`get_linewidths()`

`get_offset_position()`
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`get_offset_transform()`

`get_offsets()`
Return the offsets for the collection.

`get_path_effects()`

`get_paths()`

`get_picker()`
Return the picker object used by this artist

`get_pickradius()`

`get_rasterized()`
return True if the artist is to be rasterized

`get_sizes()`
Returns the sizes of the elements in the collection. The value represents the 'area' of the element.

Returns sizes : array
The 'area' of each element.

`get_sketch_params()`
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit = True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
```

```
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)
alias for set_linestyle

set_linewidth(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)
alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths(*paths*)

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sizes(*sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters sizes : ndarray or None
The size to set for each element of the collection. The value is the 'area' of the element.

dpi : float
The dpi of the canvas. Defaults to 72.0.

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class `matplotlib.collections.`**PolyCollection**(*verts*, *sizes=None*, *closed=True*, ***kwargs*)
Bases: `matplotlib.collections._CollectionWithSizes`

verts is a sequence of (*verts0*, *verts1*, ...) where *verts_i* is a sequence of *xy* tuples of vertices, or an equivalent `numpy` array of shape (*nv*, 2).

sizes is *None* (default) or a sequence of floats that scale the corresponding *verts_i*. The scaling is applied before the Artist master transform; if the latter is an identity transform, then the overall scaling is such that if *verts_i* specify a unit square, then *sizes_i* is the area of that square in points^2. If len(*sizes*) < *nv*, the additional values will be taken cyclically from the array.

closed, when *True*, will explicitly close the polygon.

Valid Collection keyword arguments:

- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for `matplotlib.cm.ScalarMappable`)
- *cmap*: None (optional for `matplotlib.cm.ScalarMappable`)

offsets and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

add_callback(*func*)
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The `Axes` instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the `Axes` instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the *Figure* instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_sizes()
Returns the sizes of the elements in the collection. The value represents the 'area' of the element.

Returns sizes : array
The 'area' of each element.

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:
- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if Artist has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if Artist is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call *matplotlib.axes.Axes.relim()* to update the axes limits if desired.

Note: *relim()* will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

add_callback() For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

set_facecolor(), *set_edgecolor()* For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the *Figure* instance the artist belongs to.

ACCEPTS: a *matplotlib.figure.Figure* instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)
alias for set_linestyle

set_linewidth(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)
alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths(*verts*, *closed=True*)
This allows one to delay initialization of the vertices.

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sizes(*sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters sizes : ndarray or None
The size to set for each element of the collection. The value is the 'area' of the element.

dpi : float
The dpi of the canvas. Defaults to 72.0.

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_verts(*verts*, *closed=True*)
This allows one to delay initialization of the vertices.

set_verts_and_codes(*verts*, *codes*)
This allows one to initialize vertices with path codes.

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class `matplotlib.collections.`**QuadMesh**(*meshWidth*, *meshHeight*, *coordinates*, *antialiased=True*, *shading='flat'*, ***kwargs*)
Bases: `matplotlib.collections.Collection`

Class for the efficient drawing of a quadrilateral mesh.

A quadrilateral mesh consists of a grid of vertices. The dimensions of this array are (*meshWidth* + 1, *meshHeight* + 1). Each vertex in the mesh has a different set of "mesh coordinates" representing its position in the topology of the mesh. For any values (m, n) such that 0 <= m <= *meshWidth* and 0 <= n <= *meshHeight*, the vertices at mesh coordinates (m, n), (m, n + 1), (m + 1, n + 1), and (m + 1, n) form one of the quadrilaterals in the mesh. There are thus (*meshWidth* * *meshHeight*) quadrilaterals in the mesh. The mesh need not be regular and the polygons need not be convex.

A quadrilateral mesh is represented by a (2 x ((*meshWidth* + 1) * (*meshHeight* + 1))) numpy array *coordinates*, where each row is the x and y coordinates of one of the vertices. To define the function that maps from a data point to its corresponding color, use the `set_cmap()` method. Each of these arrays is indexed in row-major order by the mesh coordinates of the vertex (or the mesh coordinates of the lower left vertex, in the case of the colors).

For example, the first entry in *coordinates* is the coordinates of the vertex at mesh coordinates (0, 0), then the one at (0, 1), then at (0, 2) .. (0, meshWidth), (1, 0), (1, 1), and so on.

shading may be 'flat', or 'gouraud'

add_callback(*func*)
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The `Axes` instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

static **convert_mesh_to_paths**(*meshWidth*, *meshHeight*, *coordinates*)
Converts a given mesh into a sequence of `matplotlib.path.Path` objects for easier rendering by backends that do not directly support quadmeshes.

This function is primarily of use to backend implementers.

convert_mesh_to_triangles(*meshWidth*, *meshHeight*, *coordinates*)
Converts a given mesh into a sequence of triangles, each point with its own color. This is useful for experiments using `draw_qouraud_triangle`.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be
- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the `Axes` instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the `Figure` instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining

affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call *matplotlib.axes.Axes.relim()* to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
'-' or 'solid'	solid line
'--' or 'dashed'	dashed line
'-.' or 'dashdot'	dash-dotted line
':' or 'dotted'	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None' | ' ' | '']

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)
alias for set_linestyle

set_linewidth(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)
alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths()

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)

Sets the sketch parameters.

Parameters scale : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)

Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

`set_url(url)`
Sets the url for the artist

ACCEPTS: a url string

`set_urls(urls)`

`set_visible(b)`
Set the artist's visiblity.

ACCEPTS: [True | False]

`set_zorder(level)`
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

`stale`
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

`to_rgba(x, alpha=None, bytes=False)`
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

`update(props)`
Update the properties of this `Artist` from the dictionary *prop*.

`update_from(other)`
copy properties from other to self

`update_scalarmappable()`
If the scalar mappable array is not none, update colors from scalar data

`zorder = 0`

class `matplotlib.collections.`**`RegularPolyCollection`**(*numsides*, *rotation=0*, *sizes=(1,)*, ***kwargs*)

Bases: `matplotlib.collections._CollectionWithSizes`

Draw a collection of regular polygons with *numsides*.

numsides the number of sides of the polygon

rotation the rotation of the polygon in radians

sizes gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:

- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for `matplotlib.cm.ScalarMappable`)
- *cmap*: None (optional for `matplotlib.cm.ScalarMappable`)

offsets and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

Example: see `examples/dynamic_collection.py` for complete example:

```
offsets = np.random.rand(20,2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0,0,0,1)

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors = facecolors,
    edgecolors = (black,),
    linewidths = (1,),
    offsets = offsets,
    transOffset = ax.transData,
    )
```

`add_callback`(*func*)

Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with `remove_callback()` later.

`add_checker`(*checker*)

Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

`autoscale`()

Autoscale the scalar limits on the norm instance using the current array

`autoscale_None`()

Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The *Axes* instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all *Artist* instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the *Axes* instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the `Figure` instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

`get_linestyle()`

`get_linestyles()`

`get_linewidth()`

`get_linewidths()`

`get_numsides()`

`get_offset_position()`
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`get_offset_transform()`

`get_offsets()`
Return the offsets for the collection.

`get_path_effects()`

`get_paths()`

`get_picker()`
Return the picker object used by this artist

`get_pickradius()`

`get_rasterized()`
return True if the artist is to be rasterized

`get_rotation()`

`get_sizes()`
Returns the sizes of the elements in the collection. The value represents the 'area' of the element.

Returns sizes : array

The 'area' of each element.

`get_sketch_params()`
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit = True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
```

```
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linestyles(*ls*)
alias for set_linestyle

set_linewidth(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

set_linewidths(*lw*)
alias for set_linewidth

set_lw(*lw*)
alias for set_linewidth

set_norm(*norm*)
set the normalization instance

set_offset_position(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

set_offsets(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

set_path_effects(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

set_paths()

set_picker(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)
Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sizes(*sizes*, *dpi=72.0*)
Set the sizes of each member of the collection.

Parameters sizes : ndarray or None
The size to set for each element of the collection. The value is the 'area' of the element.

dpi : float
The dpi of the canvas. Defaults to 72.0.

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)
Sets the sketch parameters.

Parameters scale : float, optional
The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional
The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional
The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)
Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)

Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)

Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)

copy properties from other to self

update_scalarmappable()

If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.**StarPolygonCollection**(*numsides*, *rotation=0*, *sizes=(1*, *)*, ***kwargs*)

Bases: `matplotlib.collections.RegularPolyCollection`

Draw a collection of regular stars with *numsides* points.
numsides the number of sides of the polygon
rotation the rotation of the polygon in radians
sizes gives the area of the circle circumscribing the regular polygon in points^2

Valid Collection keyword arguments:

- *edgecolors*: None
- *facecolors*: None
- *linewidths*: None
- *antialiaseds*: None
- *offsets*: None
- *transOffset*: transforms.IdentityTransform()
- *norm*: None (optional for `matplotlib.cm.ScalarMappable`)
- *cmap*: None (optional for `matplotlib.cm.ScalarMappable`)

offsets and *transOffset* are used to translate the patch after rendering (default no offsets)

If any of *edgecolors*, *facecolors*, *linewidths*, *antialiaseds* are None, they default to their `matplotlib.rcParams` patch setting, in sequence form.

Example: see `examples/dynamic_collection.py` for complete example:

```
offsets = np.random.rand(20,2)
facecolors = [cm.jet(x) for x in np.random.rand(20)]
black = (0,0,0,1)

collection = RegularPolyCollection(
    numsides=5, # a pentagon
    rotation=0, sizes=(50,),
    facecolors = facecolors,
    edgecolors = (black,),
    linewidths = (1,),
    offsets = offsets,
    transOffset = ax.transData,
    )
```

add_callback(*func*)
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with *remove_callback()* later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The *Axes* instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

`draw`(*artist*, *renderer*, **args*, ***kwargs*)

`findobj`(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be
- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

`format_cursor_data`(*data*)
Return *cursor data* string formatted.

`get_agg_filter`()
return filter function to be used for agg filter

`get_alpha`()
Return the alpha value used for blending - not supported on all backends

`get_animated`()
Return the artist's animated state

`get_array`()
Return the array

`get_axes`()
Return the `Axes` instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

`get_children`()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

`get_clim`()
return the min, max of the color limits for image scaling

`get_clip_box`()
Return artist clipbox

`get_clip_on`()
Return whether artist uses clipping

`get_clip_path`()
Return artist clip path

`get_cmap`()
return the colormap

`get_contains`()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the *Figure* instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_numsides()

get_offset_position()
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_rotation()

get_sizes()
Returns the sizes of the elements in the collection. The value represents the 'area' of the element.

Returns sizes : array
The 'area' of each element.

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:
- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:
- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *`Transform`* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining

affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the `Artist`'s zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a `Figure`.

is_transform_set()
Returns *True* if `Artist` has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if `Artist` is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with `matplotlib.axes.Axes.draw_idle()`. Call `matplotlib.axes.Axes.relim()` to update the axes limits if desired.

Note: `relim()` will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

`add_callback()` For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions (e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a *matplotlib.transforms.Bbox* instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a *Patch* (or subclass) instance
- **a *Path* instance, in which case** an optional *Transform* instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(*Path*, *Transform*) | *Patch* | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

set_facecolor(), ***set_edgecolor()*** For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

`set_linestyles`(*ls*)
alias for set_linestyle

`set_linewidth`(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

`set_linewidths`(*lw*)
alias for set_linewidth

`set_lw`(*lw*)
alias for set_linewidth

`set_norm`(*norm*)
set the normalization instance

`set_offset_position`(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`set_offsets`(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

`set_path_effects`(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

`set_paths`()

`set_picker`(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

  to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sizes(*sizes*, *dpi=72.0*)

Set the sizes of each member of the collection.

Parameters sizes : ndarray or None

The size to set for each element of the collection. The value is the 'area' of the element.

dpi : float

The dpi of the canvas. Defaults to 72.0.

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)

Sets the sketch parameters.

Parameters scale : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is

- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)
Set the *Transform* instance used by this artist.

ACCEPTS: *Transform* instance

set_url(*url*)
Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_visible(*b*)
Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

class matplotlib.collections.**TriMesh**(*triangulation*, ***kwargs*)
Bases: *matplotlib.collections.Collection*

Class for the efficient drawing of a triangular mesh using Gouraud shading.

A triangular mesh is a *Triangulation* object.

add_callback(*func*)
Adds a callback function that will be called whenever one of the `Artist`'s properties changes.

Returns an *id* that is useful for removing the callback with *remove_callback()* later.

add_checker(*checker*)
Add an entry to a dictionary of boolean flags that are set to True when the mappable is changed.

aname = 'Artist'

autoscale()
Autoscale the scalar limits on the norm instance using the current array

autoscale_None()
Autoscale the scalar limits on the norm instance using the current array, changing only limits that are None

axes
The *Axes* instance the artist resides in, or *None*.

changed()
Call this whenever the mappable is changed to notify all the callbackSM listeners to the 'changed' signal

check_update(*checker*)
If mappable has changed since the last check, return True; else return False

contains(*mouseevent*)
Test whether the mouse event occurred in the collection.

Returns True | False, `dict(ind=itemlist)`, where every item in itemlist contains the event.

static **convert_mesh_to_paths**(*tri*)
Converts a given mesh into a sequence of *matplotlib.path.Path* objects for easier rendering by backends that do not directly support meshes.

This function is primarily of use to backend implementers.

convert_xunits(*x*)
For artists in an axes, if the xaxis has units support, convert *x* using xaxis unit type

convert_yunits(*y*)
For artists in an axes, if the yaxis has units support, convert *y* using yaxis unit type

draw(*artist*, *renderer*, **args*, ***kwargs*)

findobj(*match=None*, *include_self=True*)
Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be
- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

format_cursor_data(*data*)
Return *cursor data* string formatted.

get_agg_filter()
return filter function to be used for agg filter

get_alpha()
Return the alpha value used for blending - not supported on all backends

get_animated()
Return the artist's animated state

get_array()
Return the array

get_axes()
Return the `Axes` instance the artist resides in, or *None*.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

get_children()
Return a list of the child `Artist‘s this :class:‘Artist` contains.

get_clim()
return the min, max of the color limits for image scaling

get_clip_box()
Return artist clipbox

get_clip_on()
Return whether artist uses clipping

get_clip_path()
Return artist clip path

get_cmap()
return the colormap

get_contains()
Return the _contains test used by the artist, or *None* for default.

get_cursor_data(*event*)
Get the cursor data for a given event.

get_dashes()

get_datalim(*transData*)

get_edgecolor()

get_edgecolors()

get_facecolor()

get_facecolors()

get_figure()
Return the `Figure` instance the artist belongs to.

get_fill()
return whether fill is set

get_gid()
Returns the group id

get_hatch()
Return the current hatching pattern

get_label()
Get the label used for this artist in the legend.

get_linestyle()

get_linestyles()

get_linewidth()

get_linewidths()

get_offset_position()
Returns how offsets are applied for the collection. If *offset_position* is 'screen', the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

get_offset_transform()

get_offsets()
Return the offsets for the collection.

get_path_effects()

get_paths()

get_picker()
Return the picker object used by this artist

get_pickradius()

get_rasterized()
return True if the artist is to be rasterized

get_sketch_params()
Returns the sketch parameters for the artist.

Returns sketch_params : tuple or `None`

A 3-tuple with the following elements:

- `scale`: The amplitude of the wiggle perpendicular to the source line.
- `length`: The length of the wiggle along the line.
- `randomness`: The scale factor by which the length is shrunken or expanded.

May return `None` if no sketch parameters were set.

get_snap()
Returns the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

get_transform()
Return the *Transform* instance used by this artist.

get_transformed_clip_path_and_affine()
Return the clip path with the non-affine part of its transformation applied, and the remaining affine part of its transformation.

get_transforms()

get_url()
Returns the url

get_urls()

get_visible()
Return the artist's visiblity

get_window_extent(*renderer*)

get_zorder()
Return the Artist's zorder.

have_units()
Return *True* if units are set on the *x* or *y* axes

hitlist(*event*)
List the children of the artist which contain the mouse event *event*.

is_figure_set()
Returns True if the artist is assigned to a *Figure*.

is_transform_set()
Returns *True* if Artist has a transform explicitly set.

mouseover

pchanged()
Fire an event when property changed, calling all of the registered callbacks.

pick(*mouseevent*)
call signature:

```
pick(mouseevent)
```

each child artist will fire a pick event if *mouseevent* is over the artist and the artist has picker set

pickable()
Return *True* if Artist is pickable.

properties()
return a dictionary mapping property name -> value for all Artist props

remove()
Remove the artist from the figure if possible. The effect will not be visible until the figure is redrawn, e.g., with matplotlib.axes.Axes.draw_idle(). Call *matplotlib.axes.Axes.relim()* to update the axes limits if desired.

Note: *relim()* will not see collections even if the collection was added to axes with *autolim* = True.

Note: there is no support for removing the artist's legend entry.

remove_callback(*oid*)
Remove a callback based on its *id*.

See also:

add_callback() For adding callbacks

set(***kwargs*)
A property batch setter. Pass *kwargs* to set properties. Will handle property name collisions

(e.g., if both 'color' and 'facecolor' are specified, the property with higher priority gets set last).

set_agg_filter(*filter_func*)
set agg_filter fuction.

set_alpha(*alpha*)
Set the alpha tranparencies of the collection. *alpha* must be a float or *None*.

ACCEPTS: float or None

set_animated(*b*)
Set the artist's animation state.

ACCEPTS: [True | False]

set_antialiased(*aa*)
Set the antialiasing state for rendering.

ACCEPTS: Boolean or sequence of booleans

set_antialiaseds(*aa*)
alias for set_antialiased

set_array(*A*)
Set the image array from numpy array *A*

set_axes(*axes*)
Set the `Axes` instance in which the artist resides, if any.

This has been deprecated in mpl 1.5, please use the axes property. Will be removed in 1.7 or 2.0.

ACCEPTS: an `Axes` instance

set_clim(*vmin=None*, *vmax=None*)
set the norm limits for image scaling; if *vmin* is a length2 sequence, interpret it as `(vmin, vmax)` which is used to support setp

ACCEPTS: a length 2 sequence of floats

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.

- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_cmap(*cmap*)
set the colormap for luminance data

ACCEPTS: a colormap or registered colormap name

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

`set_facecolor()`, `set_edgecolor()` For setting the edge or face color individually.

set_contains(*picker*)
Replace the contains test used by this artist. The new picker should be a callable function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

If the mouse event is over the artist, return *hit* = *True* and *props* is a dictionary of properties you want returned with the contains test.

ACCEPTS: a callable function

set_dashes(*ls*)
alias for set_linestyle

set_edgecolor(*c*)
Set the edgecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn.

ACCEPTS: matplotlib color spec or sequence of specs

set_edgecolors(*c*)
alias for set_edgecolor

set_facecolor(*c*)
Set the facecolor(s) of the collection. *c* can be a matplotlib color spec (all patches have same color), or a sequence of specs; if it is a sequence the patches will cycle through the sequence.

If *c* is 'none', the patch will not be filled.

ACCEPTS: matplotlib color spec or sequence of specs

set_facecolors(*c*)
alias for set_facecolor

set_figure(*fig*)
Set the `Figure` instance the artist belongs to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_gid(*gid*)
Sets the (group) id for the artist

ACCEPTS: an id string

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

Unlike other properties such as linewidth and colors, hatching can only be specified for the collection as a whole, not separately for each member.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_label(*s*)
Set the label to *s* for auto legend.

ACCEPTS: string or anything printable with '%s' conversion.

set_linestyle(*ls*)
Set the linestyle(s) for the collection.

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

`set_linestyles`(*ls*)
alias for set_linestyle

`set_linewidth`(*lw*)
Set the linewidth(s) for the collection. *lw* can be a scalar or a sequence; if it is a sequence the patches will cycle through the sequence

ACCEPTS: float or sequence of floats

`set_linewidths`(*lw*)
alias for set_linewidth

`set_lw`(*lw*)
alias for set_linewidth

`set_norm`(*norm*)
set the normalization instance

`set_offset_position`(*offset_position*)
Set how offsets are applied. If *offset_position* is 'screen' (default) the offset is applied after the master transform has been applied, that is, the offsets are in screen coordinates. If offset_position is 'data', the offset is applied before the master transform, i.e., the offsets are in data coordinates.

`set_offsets`(*offsets*)
Set the offsets for the collection. *offsets* can be a scalar or a sequence.

ACCEPTS: float or sequence of floats

`set_path_effects`(*path_effects*)
set path_effects, which should be a list of instances of matplotlib.patheffect._Base class or its derivatives.

`set_paths`()

`set_picker`(*picker*)
Set the epsilon for picking used by this artist

picker can be one of the following:

- *None*: picking is disabled for this artist (default)
- A boolean: if *True* then picking will be enabled and the artist will fire a pick event if the mouse event is over the artist
- A float: if picker is a number it is interpreted as an epsilon tolerance in points and the artist will fire off an event if it's data is within epsilon of the mouse event. For some artists like lines and patch collections, the artist may provide additional data to the pick event that is generated, e.g., the indices of the data within epsilon of the pick event
- A function: if picker is callable, it is a user supplied function which determines whether the artist is hit by the mouse event:

```
hit, props = picker(artist, mouseevent)
```

to determine the hit test. if the mouse event is over the artist, return *hit=True* and props is a dictionary of properties you want added to the PickEvent attributes.

ACCEPTS: [None|float|boolean|callable]

set_pickradius(*pr*)

set_rasterized(*rasterized*)

Force rasterized (bitmap) drawing in vector backend output.

Defaults to None, which implies the backend's default behavior

ACCEPTS: [True | False | None]

set_sketch_params(*scale=None*, *length=None*, *randomness=None*)

Sets the sketch parameters.

Parameters scale : float, optional

The amplitude of the wiggle perpendicular to the source line, in pixels. If scale is `None`, or not provided, no sketch filter will be provided.

length : float, optional

The length of the wiggle along the line, in pixels (default 128.0)

randomness : float, optional

The scale factor by which the length is shrunken or expanded (default 16.0)

set_snap(*snap*)

Sets the snap setting which may be:

- True: snap vertices to the nearest pixel center
- False: leave vertices as-is
- None: (auto) If the path contains only rectilinear line segments, round to the nearest pixel center

Only supported by the Agg and MacOSX backends.

set_transform(*t*)

Set the `Transform` instance used by this artist.

ACCEPTS: `Transform` instance

set_url(*url*)

Sets the url for the artist

ACCEPTS: a url string

set_urls(*urls*)

set_visible(*b*)

Set the artist's visiblity.

ACCEPTS: [True | False]

set_zorder(*level*)
Set the zorder for the artist. Artists with lower zorder values are drawn first.

ACCEPTS: any number

stale
If the artist is 'stale' and needs to be re-drawn for the output to match the internal state of the artist.

to_rgba(*x*, *alpha=None*, *bytes=False*)
Return a normalized rgba array corresponding to *x*.

In the normal case, *x* is a 1-D or 2-D sequence of scalars, and the corresponding ndarray of rgba values will be returned, based on the norm and colormap set for this ScalarMappable.

There is one special case, for handling images that are already rgb or rgba, such as might have been read from an image file. If *x* is an ndarray with 3 dimensions, and the last dimension is either 3 or 4, then it will be treated as an rgb or rgba array, and no mapping will be done. If the last dimension is 3, the *alpha* kwarg (defaulting to 1) will be used to fill in the transparency. If the last dimension is 4, the *alpha* kwarg is ignored; it does not replace the pre-existing alpha. A ValueError will be raised if the third dimension is other than 3 or 4.

In either case, if *bytes* is *False* (default), the rgba array will be floats in the 0-1 range; if it is *True*, the returned rgba array will be uint8 in the 0 to 255 range.

Note: this method assumes the input is well-behaved; it does not check for anomalies such as *x* being a masked rgba array, or being an integer type other than uint8, or being a floating point rgba array with values outside the 0-1 range.

update(*props*)
Update the properties of this `Artist` from the dictionary *prop*.

update_from(*other*)
copy properties from other to self

update_scalarmappable()
If the scalar mappable array is not none, update colors from scalar data

zorder = 0

CHAPTER

FORTYNINE

COLORBAR

49.1 matplotlib.colorbar

Colorbar toolkit with two classes and a function:

ColorbarBase the base class with full colorbar drawing functionality. It can be used as-is to make a colorbar for a given colormap; a mappable object (e.g., image) is not needed.

Colorbar the derived class for use with images or contour plots.

make_axes() a function for resizing an axes and adding a second axes suitable for a colorbar

The *colorbar()* method uses *make_axes()* and *Colorbar*; the *colorbar()* function is a thin wrapper over *colorbar()*.

class matplotlib.colorbar.**Colorbar**(*ax*, *mappable*, ***kw*)

Bases: *matplotlib.colorbar.ColorbarBase*

This class connects a *ColorbarBase* to a *ScalarMappable* such as a *AxesImage* generated via *imshow()*.

It is not intended to be instantiated directly; instead, use *colorbar()* or *colorbar()* to make your colorbar.

add_lines(*CS*, *erase=True*)

Add the lines from a non-filled ContourSet to the colorbar.

Set *erase* to False if these lines should be added to any pre-existing lines.

on_mappable_changed(*mappable*)

Updates this colorbar to match the mappable's properties.

Typically this is automatically registered as an event handler by *colorbar_factory()* and should not be called manually.

remove()

Remove this colorbar from the figure. If the colorbar was created with use_gridspec=True then restore the gridspec to its previous value.

update_bruteforce(*mappable*)

Destroy and rebuild the colorbar. This is intended to become obsolete, and will probably be deprecated and then removed. It is not called when the pyplot.colorbar function or the Figure.colorbar method are used to create the colorbar.

update_normal(*mappable*)
update solid, lines, etc. Unlike update_bruteforce, it does not clear the axes. This is meant to be called when the image or contour plot to which this colorbar belongs is changed.

class matplotlib.colorbar.**ColorbarBase**(*ax*, *cmap=None*, *norm=None*, *alpha=None*, *values=None*, *boundaries=None*, *orientation='vertical'*, *ticklocation='auto'*, *extend='neither'*, *spacing='uniform'*, *ticks=None*, *format=None*, *drawedges=False*, *filled=True*, *extendfrac=None*, *extendrect=False*, *label=''*)
Bases: `matplotlib.cm.ScalarMappable`

Draw a colorbar in an existing axes.

This is a base class for the `Colorbar` class, which is the basis for the `colorbar()` function and the `colorbar()` method, which are the usual ways of creating a colorbar.

It is also useful by itself for showing a colormap. If the *cmap* kwarg is given but *boundaries* and *values* are left as None, then the colormap will be displayed on a 0-1 scale. To show the under- and over-value colors, specify the *norm* as:

```
colors.Normalize(clip=False)
```

To show the colors versus index instead of on the 0-1 scale, use:

```
norm=colors.NoNorm.
```

Useful attributes:
ax the Axes instance in which the colorbar is drawn
lines a list of LineCollection if lines were drawn, otherwise an empty list
dividers a LineCollection if *drawedges* is True, otherwise None

Useful public methods are `set_label()` and `add_lines()`.

add_lines(*levels*, *colors*, *linewidths*, *erase=True*)
Draw lines on the colorbar.

colors and *linewidths* must be scalars or sequences the same length as *levels*.

Set *erase* to False to add lines without first removing any previously added lines.

ax = None
The axes that this colorbar lives in.

config_axis()

draw_all()
Calculate any free parameters based on the current cmap and norm, and do all the drawing.

n_rasterize = 50

remove()
Remove this colorbar from the figure

set_alpha(*alpha*)

set_label(*label*, ***kw*)
Label the long axis of the colorbar

set_ticklabels(*ticklabels*, *update_ticks=True*)
set tick labels. Tick labels are updated immediately unless update_ticks is *False*. To manually update the ticks, call *update_ticks* method explicitly.

set_ticks(*ticks*, *update_ticks=True*)
set tick locations. Tick locations are updated immediately unless update_ticks is *False*. To manually update the ticks, call *update_ticks* method explicitly.

update_ticks()
Force the update of the ticks and ticklabels. This must be called whenever the tick locator and/or tick formatter changes.

class matplotlib.colorbar.**ColorbarPatch**(*ax*, *mappable*, ***kw*)
Bases: `matplotlib.colorbar.Colorbar`

A Colorbar which is created using `Patch` rather than the default `pcolor()`.

It uses a list of Patch instances instead of a `PatchCollection` because the latter does not allow the hatch pattern to vary among the members of the collection.

matplotlib.colorbar.**colorbar_factory**(*cax*, *mappable*, ***kwargs*)
Creates a colorbar on the given axes for the given mappable.

Typically, for automatic colorbar placement given only a mappable use `colorbar()`.

matplotlib.colorbar.**make_axes**(*parents*, *location=None*, *orientation=None*, *fraction=0.15*, *shrink=1.0*, *aspect=20*, ***kw*)
Resize and reposition parent axes, and return a child axes suitable for a colorbar:

```
cax, kw = make_axes(parent, **kw)
```

Keyword arguments may include the following (with defaults):

location [[None|'left'|'right'|'top'|'bottom']] The position, relative to **parents**, where the colorbar axes should be created. If None, the value will either come from the given `orientation`, else it will default to 'right'.

orientation [[None|'vertical'|'horizontal']] The orientation of the colorbar. Typically, this keyword shouldn't be used, as it can be derived from the `location` keyword.

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions
anchor	(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes
panchor	(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes' anchor will be unchanged

Returns (cax, kw), the child axes and the reduced kw dictionary to be passed when creating the colorbar instance.

`matplotlib.colorbar.`**`make_axes_gridspec`**(*parent*, ***kw*)

Resize and reposition a parent axes, and return a child axes suitable for a colorbar. This function is similar to make_axes. Prmary differences are

- *make_axes_gridspec* only handles the *orientation* keyword and cannot handle the "location" keyword.
- *make_axes_gridspec* should only be used with a subplot parent.
- ***make_axes*** **creates an instance of Axes.** ***make_axes_gridspec*** creates an instance of Subplot.
- ***make_axes*** **updates the position of the** parent. *make_axes_gridspec* replaces the grid_spec attribute of the parent with a new one.

While this function is meant to be compatible with *make_axes*, there could be some minor differences.:

```
cax, kw = make_axes_gridspec(parent, **kw)
```

Keyword arguments may include the following (with defaults):

orientation 'vertical' or 'horizontal'

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions
anchor	(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes
panchor	(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes' anchor will be unchanged

All but the first of these are stripped from the input kw set.

Returns (cax, kw), the child axes and the reduced kw dictionary to be passed when creating the colorbar instance.

CHAPTER

FIFTY

COLORS

For a visual representation of the matplotlib colormaps, see the "Color" section in the gallery.

50.1 matplotlib.colors

A module for converting numbers or color arguments to *RGB* or *RGBA*

RGB and *RGBA* are sequences of, respectively, 3 or 4 floats in the range 0-1.

This module includes functions and classes for color specification conversions, and for mapping numbers to colors in a 1-D array of colors called a colormap. Colormapping typically involves two steps: a data array is first mapped onto the range 0-1 using an instance of `Normalize` or of a subclass; then this number in the 0-1 range is mapped to a color using an instance of a subclass of `Colormap`. Two are provided here: `LinearSegmentedColormap`, which is used to generate all the built-in colormap instances, but is also useful for making custom colormaps, and `ListedColormap`, which is used for generating a custom colormap from a list of color specifications.

The module also provides a single instance, *colorConverter*, of the `ColorConverter` class providing methods for converting single color specifications or sequences of them to *RGB* or *RGBA*.

Commands which take color arguments can use several formats to specify the colors. For the basic built-in colors, you can use a single letter

- b: blue
- g: green
- r: red
- c: cyan
- m: magenta
- y: yellow
- k: black
- w: white

Gray shades can be given as a string encoding a float in the 0-1 range, e.g.:

```
color = '0.75'
```

For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

```
color = '#eeefff'
```

or you can pass an R , G , B tuple, where each of R , G , B are in the range [0,1].

Finally, legal html names for colors, like 'red', 'burlywood' and 'chartreuse' are supported.

class `matplotlib.colors.BoundaryNorm`(*boundaries*, *ncolors*, *clip=False*)

Bases: `matplotlib.colors.Normalize`

Generate a colormap index based on discrete intervals.

Unlike `Normalize` or `LogNorm`, `BoundaryNorm` maps values to integers instead of to the interval 0-1.

Mapping to the 0-1 interval could have been done via piece-wise linear interpolation, but using integers seems simpler, and reduces the number of conversions back and forth between integer and floating point.

boundaries a monotonically increasing sequence
ncolors number of colors in the colormap to be used
If:

```
b[i] <= v < b[i+1]
```

then v is mapped to color j; as i varies from 0 to len(boundaries)-2, j goes from 0 to ncolors-1.

Out-of-range values are mapped to -1 if low and ncolors if high; these are converted to valid indices by `Colormap.__call__()` . If clip == True, out-of-range values are mapped to 0 if low and ncolors-1 if high.

`inverse`(*value*)

class `matplotlib.colors.ColorConverter`

Bases: `object`

Provides methods for converting color specifications to *RGB* or *RGBA*

Caching is used for more efficient conversion upon repeated calls with the same argument.

Ordinarily only the single instance instantiated in this module, *colorConverter*, is needed.

cache = {'#348ABD': (0.20392156862745098, 0.5411764705882353, 0.7411764705882353), '#6d904f': (0.427450

colors = {'b': (0.0, 0.0, 1.0), 'c': (0.0, 0.75, 0.75), 'm': (0.75, 0, 0.75), 'y': (0.75, 0.75, 0), 'w': (1.0, 1.0, 1.0), 'r':

`to_rgb`(*arg*)

Returns an *RGB* tuple of three floats from 0-1.

arg can be an *RGB* or *RGBA* sequence or a string in any of several forms:

1. a letter from the set 'rgbcmykw'
2. a hex color string, like '#00FFFF'
3. a standard name, like 'aqua'
4. a string representation of a float, like '0.4', indicating gray on a 0-1 scale

if *arg* is *RGBA*, the *A* will simply be discarded.

to_rgba(*arg*, *alpha=None*)

Returns an *RGBA* tuple of four floats from 0-1.

For acceptable values of *arg*, see `to_rgb()`. In addition, if *arg* is "none" (case-insensitive), then (0,0,0,0) will be returned. If *arg* is an *RGBA* sequence and *alpha* is not *None*, *alpha* will replace the original *A*.

to_rgba_array(*c*, *alpha=None*)

Returns a numpy array of *RGBA* tuples.

Accepts a single mpl color spec or a sequence of specs.

Special case to handle "no color": if *c* is "none" (case-insensitive), then an empty array will be returned. Same for an empty list.

class `matplotlib.colors.`**Colormap**(*name*, *N=256*)

Bases: `object`

Baseclass for all scalar to RGBA mappings.

Typically Colormap instances are used to convert data values (floats) from the interval `[0, 1]` to the RGBA color that the respective Colormap represents. For scaling of data into the `[0, 1]` interval see `matplotlib.colors.Normalize`. It is worth noting that `matplotlib.cm.ScalarMappable` subclasses make heavy use of this `data->normalize->map-to-color` processing chain.

Parameters name : str

The name of the colormap.

N : int

The number of rgb quantization levels.

colorbar_extend = None

When this colormap exists on a scalar mappable and colorbar_extend is not False, colorbar creation will pick up `colorbar_extend` as the default value for the `extend` keyword in the `matplotlib.colorbar.Colorbar` constructor.

is_gray()

set_bad(*color='k'*, *alpha=None*)

Set color to be used for masked values.

set_over(*color='k'*, *alpha=None*)

Set color to be used for high out-of-range values. Requires norm.clip = False

set_under(*color='k'*, *alpha=None*)

Set color to be used for low out-of-range values. Requires norm.clip = False

class `matplotlib.colors.`**LightSource**(*azdeg=315*, *altdeg=45*, *hsv_min_val=0*, *hsv_max_val=1*, *hsv_min_sat=1*, *hsv_max_sat=0*)

Bases: `object`

Create a light source coming from the specified azimuth and elevation. Angles are in degrees, with the azimuth measured clockwise from north and elevation up from the zero plane of the surface.

The `shade()` is used to produce "shaded" rgb values for a data array. `shade_rgb()` can be used to combine an rgb image with The `shade_rgb()` The `hillshade()` produces an illumination map of a surface.

Specify the azimuth (measured clockwise from south) and altitude (measured up from the plane of the surface) of the light source in degrees.

Parameters azdeg : number, optional

The azimuth (0-360, degrees clockwise from North) of the light source. Defaults to 315 degrees (from the northwest).

altdeg : number, optional

The altitude (0-90, degrees up from horizontal) of the light source. Defaults to 45 degrees from horizontal.

Notes

For backwards compatibility, the parameters *hsv_min_val*, *hsv_max_val*, *hsv_min_sat*, and *hsv_max_sat* may be supplied at initialization as well. However, these parameters will only be used if "blend_mode='hsv'" is passed into `shade()` or `shade_rgb()`. See the documentation for `blend_hsv()` for more details.

blend_hsv(*rgb*, *intensity*, *hsv_max_sat=None*, *hsv_max_val=None*, *hsv_min_val=None*, *hsv_min_sat=None*)

Take the input data array, convert to HSV values in the given colormap, then adjust those color values to give the impression of a shaded relief map with a specified light source. RGBA values are returned, which can then be used to plot the shaded image with imshow.

The color of the resulting image will be darkened by moving the (s,v) values (in hsv colorspace) toward (hsv_min_sat, hsv_min_val) in the shaded regions, or lightened by sliding (s,v) toward (hsv_max_sat hsv_max_val) in regions that are illuminated. The default extremes are chose so that completely shaded points are nearly black (s = 1, v = 0) and completely illuminated points are nearly white (s = 0, v = 1).

Parameters rgb : ndarray

An MxNx3 RGB array of floats ranging from 0 to 1 (color image).

intensity : ndarray

An MxNx1 array of floats ranging from 0 to 1 (grayscale image).

hsv_max_sat : number, optional

The maximum saturation value that the *intensity* map can shift the output image to. Defaults to 1.

hsv_min_sat : number, optional

The minimum saturation value that the *intensity* map can shift the output image to. Defaults to 0.

hsv_max_val : number, optional

The maximum value ("v" in "hsv") that the *intensity* map can shift the output image to. Defaults to 1.

hsv_min_val: number, optional

The minimum value ("v" in "hsv") that the *intensity* map can shift the output image to. Defaults to 0.

Returns rgb : ndarray

An MxNx3 RGB array representing the combined images.

`blend_overlay`(*rgb*, *intensity*)

Combines an rgb image with an intensity map using "overlay" blending.

Parameters rgb : ndarray

An MxNx3 RGB array of floats ranging from 0 to 1 (color image).

intensity : ndarray

An MxNx1 array of floats ranging from 0 to 1 (grayscale image).

Returns rgb : ndarray

An MxNx3 RGB array representing the combined images.

`blend_soft_light`(*rgb*, *intensity*)

Combines an rgb image with an intensity map using "soft light" blending. Uses the "pegtop" formula.

Parameters rgb : ndarray

An MxNx3 RGB array of floats ranging from 0 to 1 (color image).

intensity : ndarray

An MxNx1 array of floats ranging from 0 to 1 (grayscale image).

Returns rgb : ndarray

An MxNx3 RGB array representing the combined images.

`hillshade`(*elevation*, *vert_exag=1*, *dx=1*, *dy=1*, *fraction=1.0*)

Calculates the illumination intensity for a surface using the defined azimuth and elevation for the light source.

Imagine an artificial sun placed at infinity in some azimuth and elevation position illuminating our surface. The parts of the surface that slope toward the sun should brighten while those sides facing away should become darker.

Parameters elevation : array-like

A 2d array (or equivalent) of the height values used to generate an illumination map

vert_exag : number, optional

The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs meters) or to exaggerate or de-emphasize topographic effects.

dx : number, optional

The x-spacing (columns) of the input *elevation* grid.

dy : number, optional

The y-spacing (rows) of the input *elevation* grid.

fraction : number, optional

Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

Returns

——-

intensity : ndarray
A 2d array of illumination values between 0-1, where 0 is completely in shadow and 1 is completely illuminated.

shade(*data*, *cmap*, *norm=None*, *blend_mode='hsv'*, *vmin=None*, *vmax=None*, *vert_exag=1*, *dx=1*, *dy=1*, *fraction=1*, ***kwargs*)

Combine colormapped data values with an illumination intensity map (a.k.a. "hillshade") of the values.

Parameters data : array-like
A 2d array (or equivalent) of the height values used to generate a shaded map.

cmap : `Colormap` instance
The colormap used to color the *data* array. Note that this must be a `Colormap` instance. For example, rather than passing in `cmap='gist_earth'`, use `cmap=plt.get_cmap('gist_earth')` instead.

norm : `Normalize` instance, optional
The normalization used to scale values before colormapping. If None, the input will be linearly scaled between its min and max.

blend_mode : {'hsv', 'overlay', 'soft'} or callable, optional
The type of blending used to combine the colormapped data values with the illumination intensity. For backwards compatibility, this defaults to "hsv". Note that for most topographic surfaces, "overlay" or "soft" appear more visually realistic. If a user-defined function is supplied, it is expected to combine an MxNx3 RGB array of floats (ranging 0 to 1) with an MxNx1 hillshade array (also 0 to 1). (Call signature `func(rgb, illum, **kwargs)`) Additional kwargs supplied to this function will be passed on to the *blend_mode* function.

vmin : scalar or None, optional
The minimum value used in colormapping *data*. If *None* the minimum value in *data* is used. If *norm* is specified, then this argument will be ignored.

vmax : scalar or None, optional
The maximum value used in colormapping *data*. If *None* the maximum value in *data* is used. If *norm* is specified, then this argument will be ignored.

vert_exag : number, optional
The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for dif-

ferences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs meters) or to exaggerate or de-emphasize topography.

dx : number, optional
: The x-spacing (columns) of the input *elevation* grid.

dy : number, optional
: The y-spacing (rows) of the input *elevation* grid.

fraction : number, optional
: Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

Additional kwargs are passed on to the *blend_mode* function.

Returns rgba : ndarray
: An MxNx4 array of floats ranging between 0-1.

`shade_rgb`(*rgb*, *elevation*, *fraction=1.0*, *blend_mode='hsv'*, *vert_exag=1*, *dx=1*, *dy=1*, ***kwargs*)

Take the input RGB array (ny*nx*3) adjust their color values to given the impression of a shaded relief map with a specified light source using the elevation (ny*nx). A new RGB array ((ny*nx*3)) is returned.

Parameters rgb : array-like
: An MxNx3 RGB array, assumed to be in the range of 0 to 1.

elevation : array-like
: A 2d array (or equivalent) of the height values used to generate a shaded map.

fraction : number
: Increases or decreases the contrast of the hillshade. Values greater than one will cause intermediate values to move closer to full illumination or shadow (and clipping any values that move beyond 0 or 1). Note that this is not visually or mathematically the same as vertical exaggeration.

blend_mode : {'hsv', 'overlay', 'soft'} or callable, optional
: The type of blending used to combine the colormapped data values with the illumination intensity. For backwards compatibility, this defaults to "hsv". Note that for most topographic surfaces, "overlay" or "soft" appear more visually realistic. If a user-defined function is supplied, it is expected to combine an MxNx3 RGB array of floats (ranging 0 to 1) with an MxNx1 hillshade array (also 0 to 1). (Call signature `func(rgb, illum, **kwargs)`) Additional kwargs supplied to this function will be passed on to the *blend_mode* function.

vert_exag : number, optional
: The amount to exaggerate the elevation values by when calculating illumination. This can be used either to correct for differences in units between the x-y coordinate system and the elevation coordinate system (e.g. decimal degrees vs meters) or to

exaggerate or de-emphasize topography.

dx : number, optional

The x-spacing (columns) of the input *elevation* grid.

dy : number, optional

The y-spacing (rows) of the input *elevation* grid.

Additional kwargs are passed on to the *blend_mode* function.

Returns shaded_rgb : ndarray

An MxNx3 array of floats ranging between 0-1.

class `matplotlib.colors.LinearSegmentedColormap`(*name*, *segmentdata*, *N=256*, *gamma=1.0*)

Bases: *matplotlib.colors.Colormap*

Colormap objects based on lookup tables using linear segments.

The lookup table is generated using linear interpolation for each primary color, with the 0-1 domain divided into any number of segments.

Create color map from linear mapping segments

segmentdata argument is a dictionary with a red, green and blue entries. Each entry should be a list of *x*, *y0*, *y1* tuples, forming rows in a table. Entries for alpha are optional.

Example: suppose you want red to increase from 0 to 1 over the bottom half, green to do the same over the middle half, and blue over the top half. Then you would use:

```
cdict = {'red':   [(0.0,  0.0, 0.0),
                   (0.5,  1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'green': [(0.0,  0.0, 0.0),
                   (0.25, 0.0, 0.0),
                   (0.75, 1.0, 1.0),
                   (1.0,  1.0, 1.0)],

         'blue':  [(0.0,  0.0, 0.0),
                   (0.5,  0.0, 0.0),
                   (1.0,  1.0, 1.0)]}
```

Each row in the table for a given color is a sequence of *x*, *y0*, *y1* tuples. In each sequence, *x* must increase monotonically from 0 to 1. For any input value *z* falling between *x[i]* and *x[i+1]*, the output value of a given color will be linearly interpolated between *y1[i]* and *y0[i+1]*:

```
row i:   x  y0  y1
               /
              /
row i+1: x  y0  y1
```

Hence y0 in the first row and y1 in the last row are never used.

See also:

LinearSegmentedColormap.from_list() Static method; factory function for generating a smoothly-varying LinearSegmentedColormap.

`makeMappingArray()` For information about making a mapping array.

static from_list(*name*, *colors*, *N=256*, *gamma=1.0*)
Make a linear segmented colormap with *name* from a sequence of *colors* which evenly transitions from colors[0] at val=0 to colors[-1] at val=1. *N* is the number of rgb quantization levels. Alternatively, a list of (value, color) tuples can be given to divide the range unevenly.

set_gamma(*gamma*)
Set a new gamma value and regenerate color map.

class matplotlib.colors.**ListedColormap**(*colors*, *name='from_list'*, *N=None*)
Bases: `matplotlib.colors.Colormap`

Colormap object generated from a list of colors.

This may be most useful when indexing directly into a colormap, but it can also be used to generate special colormaps for ordinary mapping.

Make a colormap from a list of colors.

colors a list of matplotlib color specifications, or an equivalent Nx3 or Nx4 floating point array (*N* rgb or rgba values)

name a string to identify the colormap

N the number of entries in the map. The default is *None*, in which case there is one colormap entry for each element in the list of colors. If:

```
N < len(colors)
```

the list will be truncated at *N*. If:

```
N > len(colors)
```

the list will be extended by repetition.

class matplotlib.colors.**LogNorm**(*vmin=None*, *vmax=None*, *clip=False*)
Bases: `matplotlib.colors.Normalize`

Normalize a given value to the 0-1 range on a log scale

If *vmin* or *vmax* is not given, they are initialized from the minimum and maximum value respectively of the first input processed. That is, *__call__(A)* calls *autoscale_None(A)*. If *clip* is *True* and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

```
vmin==vmax
```

Works with scalars or arrays, including masked arrays. If *clip* is *True*, masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and masked colors in the colormap, so it is likely to lead to surprises; therefore the default is *clip* = *False*.

autoscale(*A*)
Set *vmin*, *vmax* to min, max of *A*.

autoscale_None(*A*)
autoscale only None-valued vmin or vmax

inverse(*value*)

class matplotlib.colors.**NoNorm**(*vmin=None*, *vmax=None*, *clip=False*)

Bases: `matplotlib.colors.Normalize`

Dummy replacement for Normalize, for the case where we want to use indices directly in a `ScalarMappable` .

If *vmin* or *vmax* is not given, they are initialized from the minimum and maximum value respectively of the first input processed. That is, *__call__(A)* calls *autoscale_None(A)*. If *clip* is *True* and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

```
vmin==vmax
```

Works with scalars or arrays, including masked arrays. If *clip* is *True*, masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and masked colors in the colormap, so it is likely to lead to surprises; therefore the default is *clip* = *False*.

inverse(*value*)

class matplotlib.colors.**Normalize**(*vmin=None*, *vmax=None*, *clip=False*)

Bases: `object`

A class which, when called, can normalize data into the `[0.0, 1.0]` interval.

If *vmin* or *vmax* is not given, they are initialized from the minimum and maximum value respectively of the first input processed. That is, *__call__(A)* calls *autoscale_None(A)*. If *clip* is *True* and the given value falls outside the range, the returned value will be 0 or 1, whichever is closer. Returns 0 if:

```
vmin==vmax
```

Works with scalars or arrays, including masked arrays. If *clip* is *True*, masked values are set to 1; otherwise they remain masked. Clipping silently defeats the purpose of setting the over, under, and masked colors in the colormap, so it is likely to lead to surprises; therefore the default is *clip* = *False*.

autoscale(*A*)

Set *vmin*, *vmax* to min, max of *A*.

autoscale_None(*A*)

autoscale only None-valued vmin or vmax

inverse(*value*)

static **process_value**(*value*)

Homogenize the input *value* for easy and efficient normalization.

value can be a scalar or sequence.

Returns *result*, *is_scalar*, where *result* is a masked array matching *value*. Float dtypes are preserved; integer types with two bytes or smaller are converted to np.float32, and larger types are converted to np.float. Preserving float32 when possible, and using in-place operations, can greatly improve speed for large arrays.

Experimental; we may want to add an option to force the use of float32.

scaled()
return true if vmin and vmax set

class matplotlib.colors.**PowerNorm**(*gamma*, *vmin=None*, *vmax=None*, *clip=False*)
Bases: `matplotlib.colors.Normalize`

Normalize a given value to the `[0, 1]` interval with a power-law scaling. This will clip any negative data points to 0.

autoscale(*A*)
Set *vmin*, *vmax* to min, max of *A*.

autoscale_None(*A*)
autoscale only None-valued vmin or vmax

inverse(*value*)

class matplotlib.colors.**SymLogNorm**(*linthresh*, *linscale=1.0*, *vmin=None*, *vmax=None*, *clip=False*)
Bases: `matplotlib.colors.Normalize`

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter *linthresh* allows the user to specify the size of this range (-*linthresh*, *linthresh*).

linthresh: The range within which the plot is linear (to avoid having the plot go to infinity around zero).

linscale: This allows the linear range (-*linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range. Defaults to 1.

autoscale(*A*)
Set *vmin*, *vmax* to min, max of *A*.

autoscale_None(*A*)
autoscale only None-valued vmin or vmax

inverse(*value*)

matplotlib.colors.**from_levels_and_colors**(*levels*, *colors*, *extend='neither'*)
A helper routine to generate a cmap and a norm instance which behave similar to contourf's levels and colors arguments.

Parameters levels : sequence of numbers
The quantization levels used to construct the `BoundaryNorm`. Values `v` are quantizied to level `i` if `lev[i] <= v < lev[i+1]`.

colors : sequence of colors
: The fill color to use for each level. If `extend` is "neither" there must be `n_level - 1` colors. For an `extend` of "min" or "max" add one extra color, and for an `extend` of "both" add two colors.

extend : {'neither', 'min', 'max', 'both'}, optional
: The behaviour when a value falls out of range of the given levels. See `contourf()` for details.

Returns (cmap, norm) : tuple containing a `Colormap` and a `Normalize` instance

`matplotlib.colors.`**`hex2color`**(*s*)
: Take a hex string *s* and return the corresponding rgb 3-tuple Example: #efefef -> (0.93725, 0.93725, 0.93725)

`matplotlib.colors.`**`hsv_to_rgb`**(*hsv*)
: convert hsv values in a numpy array to rgb values all values assumed to be in range [0, 1]

  Parameters hsv : (..., 3) array-like
  : All values assumed to be in range [0, 1]

  Returns rgb : (..., 3) ndarray
  : Colors converted to RGB values in range [0, 1]

`matplotlib.colors.`**`is_color_like`**(*c*)
: Return *True* if *c* can be converted to *RGB*

`matplotlib.colors.`**`makeMappingArray`**(*N*, *data*, *gamma=1.0*)
: Create an *N* -element 1-d lookup table

  data represented by a list of x,y0,y1 mapping correspondences. Each element in this list represents how a value between 0 and 1 (inclusive) represented by x is mapped to a corresponding value between 0 and 1 (inclusive). The two values of y are to allow for discontinuous mapping functions (say as might be found in a sawtooth) where y0 represents the value of y for values of x <= to that given, and y1 is the value to be used for x > than that given). The list must start with x=0, end with x=1, and all values of x must be in increasing order. Values between the given mapping points are determined by simple linear interpolation.

  Alternatively, data can be a function mapping values between 0 - 1 to 0 - 1.

  The function returns an array "result" where `result[x*(N-1)]` gives the closest value for values of x between 0 and 1.

`matplotlib.colors.`**`rgb2hex`**(*rgb*)
: Given an rgb or rgba sequence of 0-1 floats, return the hex string

`matplotlib.colors.`**`rgb_to_hsv`**(*arr*)
: convert float rgb values (in the range [0, 1]), in a numpy array to hsv values.

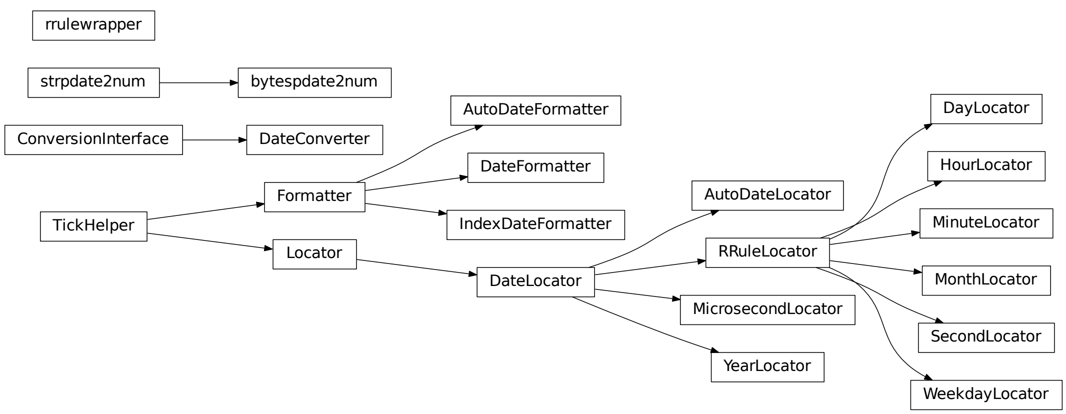
  Parameters arr : (..., 3) array-like
  : All values must be in the range [0, 1]

  Returns hsv : (..., 3) ndarray
  : Colors converted to hsv values in range [0, 1]

CHAPTER

FIFTYONE

DATES



51.1 matplotlib.dates

Matplotlib provides sophisticated date plotting capabilities, standing on the shoulders of python `datetime`, the add-on modules `pytz` and `dateutil`. `datetime` objects are converted to floating point numbers which represent time in days since 0001-01-01 UTC, plus 1. For example, 0001-01-01, 06:00 is 1.25, not 0.25. The helper functions *date2num()*, *num2date()* and *drange()* are used to facilitate easy conversion to and from `datetime` and numeric ranges.

Note: Like Python's datetime, mpl uses the Gregorian calendar for all conversions between dates and floating point numbers. This practice is not universal, and calendar differences can cause confusing differences between what Python and mpl give as the number of days since 0001-01-01 and what other software and databases yield. For example, the US Naval Observatory uses a calendar that switches from Julian to Gregorian in October, 1582. Hence, using their calculator, the number of days between 0001-01-01 and 2006-04-01 is 732403, whereas using the Gregorian calendar via the datetime module we find:

```
In [31]:date(2006,4,1).toordinal() - date(1,1,1).toordinal()
Out[31]:732401
```

A wide range of specific and general purpose date tick locators and formatters are provided in this module. See *matplotlib.ticker* for general information on tick locators and formatters. These are described below.

All the matplotlib date converters, tickers and formatters are timezone aware, and the default timezone is given by the timezone parameter in your `matplotlibrc` file. If you leave out a `tz` timezone instance, the default from your rc file will be assumed. If you want to use a custom time zone, pass a `pytz.timezone` instance with the tz keyword argument to *num2date()*, `plot_date()`, and any custom date tickers or locators you create. See pytz for information on `pytz` and timezone handling.

The dateutil module provides additional code to handle date ticking, making it easy to place ticks on any kinds of dates. See examples below.

51.1.1 Date tickers

Most of the date tickers can locate single or multiple values. For example:

```
# import constants for the days of the week
from matplotlib.dates import MO, TU, WE, TH, FR, SA, SU

# tick on mondays every week
loc = WeekdayLocator(byweekday=MO, tz=tz)

# tick on mondays and saturdays
loc = WeekdayLocator(byweekday=(MO, SA))
```

In addition, most of the constructors take an interval argument:

```
# tick on mondays every second week
loc = WeekdayLocator(byweekday=MO, interval=2)
```

The rrule locator allows completely general date ticking:

```
# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
```

Here are all the date tickers:

- *MinuteLocator*: locate minutes
- *HourLocator*: locate hours
- *DayLocator*: locate specifed days of the month
- *WeekdayLocator*: Locate days of the week, e.g., MO, TU
- *MonthLocator*: locate months, e.g., 7 for july
- *YearLocator*: locate years that are multiples of base
- *RRuleLocator*: locate using a `matplotlib.dates.rrulewrapper`. The `rrulewrapper` is a simple wrapper around a `dateutil.rrule` (dateutil) which allow almost arbitrary date tick specifications. See rrule example.

- `AutoDateLocator`: On autoscale, this class picks the best `MultipleDateLocator` to set the view limits and the tick locations.

51.1.2 Date formatters

Here all all the date formatters:

- `AutoDateFormatter`: attempts to figure out the best format to use. This is most useful when used with the `AutoDateLocator`.
- `DateFormatter`: use `strftime()` format strings
- `IndexDateFormatter`: date plots with implicit *x* indexing.

`matplotlib.dates.`**`date2num`**(*d*)

d is either a `datetime` instance or a sequence of datetimes.

Return value is a floating point number (or sequence of floats) which gives the number of days (fraction part represents hours, minutes, seconds) since 0001-01-01 00:00:00 UTC, *plus one*. The addition of one here is a historical artifact. Also, note that the Gregorian calendar is assumed; this is not universal practice. For details, see the module docstring.

`matplotlib.dates.`**`num2date`**(*x*, *tz=None*)

x is a float value which gives the number of days (fraction part represents hours, minutes, seconds) since 0001-01-01 00:00:00 UTC *plus one*. The addition of one here is a historical artifact. Also, note that the Gregorian calendar is assumed; this is not universal practice. For details, see the module docstring.

Return value is a `datetime` instance in timezone *tz* (default to rcparams TZ value).

If *x* is a sequence, a sequence of `datetime` objects will be returned.

`matplotlib.dates.`**`drange`**(*dstart*, *dend*, *delta*)

Return a date range as float Gregorian ordinals. *dstart* and *dend* are `datetime` instances. *delta* is a `datetime.timedelta` instance.

`matplotlib.dates.`**`epoch2num`**(*e*)

Convert an epoch or sequence of epochs to the new date format, that is days since 0001.

`matplotlib.dates.`**`num2epoch`**(*d*)

Convert days since 0001 to epoch. *d* can be a number or sequence.

`matplotlib.dates.`**`mx2num`**(*mxdates*)

Convert mx `datetime` instance (or sequence of mx instances) to the new date format.

class `matplotlib.dates.`**`DateFormatter`**(*fmt*, *tz=None*)

Bases: `matplotlib.ticker.Formatter`

Tick location is seconds since the epoch. Use a `strftime()` format string.

Python only supports `datetime` `strftime()` formatting for years greater than 1900. Thanks to Andrew Dalke, Dalke Scientific Software who contributed the `strftime()` code below to include dates earlier than this year.
***fmt* is a `strftime()` format string; *tz* is the** `tzinfo` instance.

illegal_s = re.compile('((^|[^%])(%%)*%s)')

set_tzinfo(*tz*)

strftime(*dt*, *fmt=None*)

Refer to documentation for datetime.strftime.

fmt is a `strftime()` format string.

Warning: For years before 1900, depending upon the current locale it is possible that the year displayed with %x might be incorrect. For years before 100, %y and %Y will yield zero-padded strings.

strftime_pre_1900(*dt*, *fmt=None*)

Call time.strftime for years before 1900 by rolling forward a multiple of 28 years.

fmt is a `strftime()` format string.

Dalke: I hope I did this math right. Every 28 years the calendar repeats, except through century leap years excepting the 400 year leap years. But only if you're using the Gregorian calendar.

class matplotlib.dates.**IndexDateFormatter**(*t*, *fmt*, *tz=None*)

Bases: `matplotlib.ticker.Formatter`

Use with `IndexLocator` to cycle format strings by index.

t is a sequence of dates (floating point days). *fmt* is a `strftime()` format string.

class matplotlib.dates.**AutoDateFormatter**(*locator*, *tz=None*, *defaultfmt='%Y-%m-%d'*)

Bases: `matplotlib.ticker.Formatter`

This class attempts to figure out the best format to use. This is most useful when used with the `AutoDateLocator`.

The AutoDateFormatter has a scale dictionary that maps the scale of the tick (the distance in days between one major tick) and a format string. The default looks like this:

```
self.scaled = {
   365.0  : '%Y',
   30.    : '%b %Y',
   1.0    : '%b %d %Y',
   1./24. : '%H:%M:%S',
   1. / (24. * 60.): '%H:%M:%S.%f',
   }
```

The algorithm picks the key in the dictionary that is >= the current scale and uses that format string. You can customize this dictionary by doing:

```
>>> locator = AutoDateLocator()
>>> formatter = AutoDateFormatter(locator)
>>> formatter.scaled[1/(24.*60.)] = '%M:%S' # only show min and sec
```

A custom `FuncFormatter` can also be used. The following example shows how to use a custom format function to strip trailing zeros from decimal seconds and adds the date to the first ticklabel:

```
>>> def my_format_function(x, pos=None):
...     x = matplotlib.dates.num2date(x)
...     if pos == 0:
...         fmt = '%D %H:%M:%S.%f'
...     else:
...         fmt = '%H:%M:%S.%f'
...     label = x.strftime(fmt)
...     label = label.rstrip("0")
...     label = label.rstrip(".")
...     return label
>>> from matplotlib.ticker import FuncFormatter
>>> formatter.scaled[1/(24.*60.)] = FuncFormatter(my_format_function)
```

Autoformat the date labels. The default format is the one to use if none of the values in `self.scaled` are greater than the unit returned by `locator._get_unit()`.

class `matplotlib.dates.`**`DateLocator`**(*tz=None*)

Bases: `matplotlib.ticker.Locator`

Determines the tick locations when plotting dates.

tz is a `tzinfo` instance.

`datalim_to_dt`()

Convert axis data interval to datetime objects.

`hms0d` = {'byminute': 0, 'bysecond': 0, 'byhour': 0}

`nonsingular`(*vmin*, *vmax*)

Given the proposed upper and lower extent, adjust the range if it is too close to being singular (i.e. a range of ~0).

`set_tzinfo`(*tz*)

Set time zone info.

`viewlim_to_dt`()

Converts the view interval to datetime objects.

class `matplotlib.dates.`**`RRuleLocator`**(*o*, *tz=None*)

Bases: `matplotlib.dates.DateLocator`

`autoscale`()

Set the view limits to include the data range.

static `get_unit_generic`(*freq*)

`tick_values`(*vmin*, *vmax*)

class `matplotlib.dates.`**`AutoDateLocator`**(*tz=None*, *minticks=5*, *maxticks=None*, *interval_multiples=False*)

Bases: `matplotlib.dates.DateLocator`

On autoscale, this class picks the best `DateLocator` to set the view limits and the tick locations.

minticks is the minimum number of ticks desired, which is used to select the type of ticking (yearly, monthly, etc.).

maxticks is the maximum number of ticks desired, which controls any interval between ticks (ticking every other, every 3, etc.). For really fine-grained control, this can be a dictionary mapping individual rrule frequency constants (YEARLY, MONTHLY, etc.) to their own maximum number of ticks. This can be used to keep the number of ticks appropriate to the format chosen in `AutoDateFormatter`. Any frequency not specified in this dictionary is given a default value.

tz is a `tzinfo` instance.

interval_multiples is a boolean that indicates whether ticks should be chosen to be multiple of the interval. This will lock ticks to 'nicer' locations. For example, this will force the ticks to be at hours 0,6,12,18 when hourly ticking is done at 6 hour intervals.

The AutoDateLocator has an interval dictionary that maps the frequency of the tick (a constant from dateutil.rrule) and a multiple allowed for that ticking. The default looks like this:

```
self.intervald = {
  YEARLY  : [1, 2, 4, 5, 10, 20, 40, 50, 100, 200, 400, 500,
            1000, 2000, 4000, 5000, 10000],
  MONTHLY : [1, 2, 3, 4, 6],
  DAILY   : [1, 2, 3, 7, 14],
  HOURLY  : [1, 2, 3, 4, 6, 12],
  MINUTELY: [1, 5, 10, 15, 30],
  SECONDLY: [1, 5, 10, 15, 30],
  MICROSECONDLY: [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000,
                 5000, 10000, 20000, 50000, 100000, 200000, 500000,
                 1000000],
  }
```

The interval is used to specify multiples that are appropriate for the frequency of ticking. For instance, every 7 days is sensible for daily ticks, but for minutes/seconds, 15 or 30 make sense. You can customize this dictionary by doing:

```
locator = AutoDateLocator()
locator.intervald[HOURLY] = [3] # only show every 3 hours
```

autoscale()
: Try to choose the view limits intelligently.

get_locator(*dmin*, *dmax*)
: Pick the best locator based on a distance.

nonsingular(*vmin*, *vmax*)

refresh()
: Refresh internal information based on current limits.

set_axis(*axis*)

tick_values(*vmin*, *vmax*)

class matplotlib.dates.**YearLocator**(*base=1*, *month=1*, *day=1*, *tz=None*)
Bases: *matplotlib.dates.DateLocator*

Make ticks on a given day of each year that is a multiple of base.

Examples:

```
# Tick every year on Jan 1st
locator = YearLocator()

# Tick every 5 years on July 4th
locator = YearLocator(5, month=7, day=4)
```

Mark years that are multiple of base on a given month and day (default jan 1).

autoscale()
Set the view limits to include the data range.

tick_values(*vmin*, *vmax*)

class matplotlib.dates.**MonthLocator**(*bymonth=None*, *bymonthday=1*, *interval=1*, *tz=None*)
Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurances of each month month, e.g., 1, 3, 12.

Mark every month in *bymonth*; *bymonth* can be an int or sequence. Default is `range(1,13)`, i.e. every month.

interval is the interval between each iteration. For example, if `interval=2`, mark every second occurance.

class matplotlib.dates.**WeekdayLocator**(*byweekday=1*, *interval=1*, *tz=None*)
Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurances of each weekday.

Mark every weekday in *byweekday*; *byweekday* can be a number or sequence.

Elements of *byweekday* must be one of MO, TU, WE, TH, FR, SA, SU, the constants from `dateutil.rrule`, which have been imported into the *matplotlib.dates* namespace.

interval specifies the number of weeks to skip. For example, `interval=2` plots every second week.

class matplotlib.dates.**DayLocator**(*bymonthday=None*, *interval=1*, *tz=None*)
Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurances of each day of the month. For example, 1, 15, 30.

Mark every day in *bymonthday*; *bymonthday* can be an int or sequence.

Default is to tick every day of the month: `bymonthday=range(1,32)`

class matplotlib.dates.**HourLocator**(*byhour=None*, *interval=1*, *tz=None*)
Bases: *matplotlib.dates.RRuleLocator*

Make ticks on occurances of each hour.

Mark every hour in *byhour*; *byhour* can be an int or sequence. Default is to tick every hour: `byhour=range(24)`

interval is the interval between each iteration. For example, if `interval=2`, mark every second occurrence.

class `matplotlib.dates.`**`MinuteLocator`**(*byminute=None*, *interval=1*, *tz=None*)

Bases: `matplotlib.dates.RRuleLocator`

Make ticks on occurances of each minute.

Mark every minute in *byminute*; *byminute* can be an int or sequence. Default is to tick every minute: `byminute=range(60)`

interval is the interval between each iteration. For example, if `interval=2`, mark every second occurrence.

class `matplotlib.dates.`**`SecondLocator`**(*bysecond=None*, *interval=1*, *tz=None*)

Bases: `matplotlib.dates.RRuleLocator`

Make ticks on occurances of each second.

Mark every second in *bysecond*; *bysecond* can be an int or sequence. Default is to tick every second: `bysecond = range(60)`

interval is the interval between each iteration. For example, if `interval=2`, mark every second occurrence.

class `matplotlib.dates.`**`MicrosecondLocator`**(*interval=1*, *tz=None*)

Bases: `matplotlib.dates.DateLocator`

Make ticks on occurances of each microsecond.

interval is the interval between each iteration. For example, if `interval=2`, mark every second microsecond.

`set_axis`(*axis*)

`set_data_interval`(*vmin*, *vmax*)

`set_view_interval`(*vmin*, *vmax*)

`tick_values`(*vmin*, *vmax*)

class `matplotlib.dates.`**`rrule`**(*freq*, *dtstart=None*, *interval=1*, *wkst=None*, *count=None*, *until=None*, *bysetpos=None*, *bymonth=None*, *bymonthday=None*, *byyearday=None*, *byeaster=None*, *byweekno=None*, *byweekday=None*, *byhour=None*, *byminute=None*, *bysecond=None*, *cache=False*)

Bases: `dateutil.rrule.rrulebase`

That's the base of the rrule operation. It accepts all the keywords defined in the RFC as its constructor parameters (except byday, which was renamed to byweekday) and more. The constructor prototype is:

```
rrule(freq)
```

Where freq must be one of YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, or SECONDLY.

Note: Per RFC section 3.3.10, recurrence instances falling on invalid dates and times are ignored rather than coerced:

> Recurrence rules may generate recurrence instances with an invalid date (e.g., February 30) or nonexistent local time (e.g., 1:30 AM on a day where the local time is moved forward by an hour at 1:00 AM). Such recurrence instances MUST be ignored and MUST NOT be counted as part of the recurrence set.

This can lead to possibly surprising behavior when, for example, the start date occurs at the end of the month:

```
>>> from dateutil.rrule import rrule, MONTHLY
>>> from datetime import datetime
>>> start_date = datetime(2014, 12, 31)
>>> list(rrule(freq=MONTHLY, count=4, dtstart=start_date))
...
[datetime.datetime(2014, 12, 31, 0, 0),
 datetime.datetime(2015, 1, 31, 0, 0),
 datetime.datetime(2015, 3, 31, 0, 0),
 datetime.datetime(2015, 5, 31, 0, 0)]
```

Additionally, it supports the following keyword arguments:

Parameters

- **cache** – If given, it must be a boolean value specifying to enable or disable caching of results. If you will use the same rrule instance multiple times, enabling caching will improve the performance considerably.
- **dtstart** – The recurrence start. Besides being the base for the recurrence, missing parameters in the final recurrence instances will also be extracted from this date. If not given, datetime.now() will be used instead.
- **interval** – The interval between each freq iteration. For example, when using YEARLY, an interval of 2 means once every two years, but with HOURLY, it means once every two hours. The default interval is 1.
- **wkst** – The week start day. Must be one of the MO, TU, WE constants, or an integer, specifying the first day of the week. This will affect recurrences based on weekly periods. The default week start is got from calendar.firstweekday(), and may be modified by calendar.setfirstweekday().
- **count** – How many occurrences will be generated.

Note: As of version 2.5.0, the use of the `until` keyword together with the `count` keyword is deprecated per RFC-2445 Sec. 4.3.10.

- **until** – If given, this must be a datetime instance, that will specify the limit of the recurrence. The last recurrence in the rule is the greatest datetime that is less than or equal to the value specified in the `until` parameter.

  Note: As of version 2.5.0, the use of the `until` keyword together with the `count` keyword is deprecated per RFC-2445 Sec. 4.3.10.

- **bysetpos** – If given, it must be either an integer, or a sequence of integers, positive or negative. Each given integer will specify an occurrence number, corresponding to the nth occurrence of the rule inside the frequency period. For example, a bysetpos of -1 if combined with a MONTHLY frequency, and a byweekday of (MO, TU, WE, TH, FR), will result in the last work day of every month.
- **bymonth** – If given, it must be either an integer, or a sequence of integers, meaning the months to apply the recurrence to.
- **bymonthday** – If given, it must be either an integer, or a sequence of integers, meaning the month days to apply the recurrence to.
- **byyearday** – If given, it must be either an integer, or a sequence of integers, meaning the year days to apply the recurrence to.
- **byweekno** – If given, it must be either an integer, or a sequence of integers, meaning the week numbers to apply the recurrence to. Week numbers have the meaning described in ISO8601, that is, the first week of the year is that containing at least four days of the new year.
- **byweekday** – If given, it must be either an integer (0 == MO), a sequence of integers, one of the weekday constants (MO, TU, etc), or a sequence of these constants. When given, these variables will define the weekdays where the recurrence will be applied. It's also possible to use an argument n for the weekday instances, which will mean the nth occurrence of this weekday in the period. For example, with MONTHLY, or with YEARLY and BYMONTH, using FR(+1) in byweekday will specify the first friday of the month where the recurrence happens. Notice that in the RFC documentation, this is specified as BYDAY, but was renamed to avoid the ambiguity of that keyword.
- **byhour** – If given, it must be either an integer, or a sequence of integers, meaning the hours to apply the recurrence to.
- **byminute** – If given, it must be either an integer, or a sequence of integers, meaning the minutes to apply the recurrence to.
- **bysecond** – If given, it must be either an integer, or a sequence of integers, meaning the seconds to apply the recurrence to.
- **byeaster** – If given, it must be either an integer, or a sequence of integers, positive or negative. Each integer will define an offset from the Easter Sunday. Passing the offset 0 to byeaster will yield the Easter Sunday itself. This is an extension to the RFC specification.

replace(***kwargs*)

Return new rrule with same attributes except for those attributes given new values by whichever keyword arguments are specified.

class `matplotlib.dates.`**`relativedelta`**(*dt1=None*, *dt2=None*, *years=0*, *months=0*, *days=0*, *leapdays=0*, *weeks=0*, *hours=0*, *minutes=0*, *seconds=0*, *microseconds=0*, *year=None*, *month=None*, *day=None*, *weekday=None*, *yearday=None*, *nlyearday=None*, *hour=None*, *minute=None*, *second=None*, *microsecond=None*)

Bases: `object`

The relativedelta type is based on the specification of the excellent work done by M.-A. Lemburg in his mx.DateTime extension. However, notice that this type does *NOT* implement the same algorithm as his work. Do *NOT* expect it to behave like mx.DateTime's counterpart.

There are two different ways to build a relativedelta instance. The first one is passing it two date/datetime classes:

```
relativedelta(datetime1, datetime2)
```

The second one is passing it any number of the following keyword arguments:

```
relativedelta(arg1=x,arg2=y,arg3=z...)

year, month, day, hour, minute, second, microsecond:
    Absolute information (argument is singular); adding or subtracting a
    relativedelta with absolute information does not perform an aritmetic
    operation, but rather REPLACES the corresponding value in the
    original datetime with the value(s) in relativedelta.

years, months, weeks, days, hours, minutes, seconds, microseconds:
    Relative information, may be negative (argument is plural); adding
    or subtracting a relativedelta with relative information performs
    the corresponding aritmetic operation on the original datetime value
    with the information in the relativedelta.

weekday:
    One of the weekday instances (MO, TU, etc). These instances may
    receive a parameter N, specifying the Nth weekday, which could
    be positive or negative (like MO(+1) or MO(-2). Not specifying
    it is the same as specifying +1. You can also use an integer,
    where 0=MO.

leapdays:
    Will add given days to the date found, if year is a leap
    year, and the date found is post 28 of february.

yearday, nlyearday:
    Set the yearday or the non-leap year day (jump leap days).
    These are converted to day/month/leapdays information.
```

Here is the behavior of operations with relativedelta:

1. Calculate the absolute year, using the 'year' argument, or the original datetime year, if the argument is not present.

2.Add the relative 'years' argument to the absolute year.
3.Do steps 1 and 2 for month/months.
4.Calculate the absolute day, using the 'day' argument, or the original datetime day, if the argument is not present. Then, subtract from the day until it fits in the year and month found after their operations.
5.Add the relative 'days' argument to the absolute day. Notice that the 'weeks' argument is multiplied by 7 and added to 'days'.
6.Do steps 1 and 2 for hour/hours, minute/minutes, second/seconds, microsecond/microseconds.
7.If the 'weekday' argument is present, calculate the weekday, with the given (wday, nth) tuple. wday is the index of the weekday (0-6, 0=Mon), and nth is the number of weeks to add forward or backward, depending on its signal. Notice that if the calculated date is already Monday, for example, using (0, 1) or (0, -1) won't change the day.

normalized()

Return a version of this object represented entirely using integer values for the relative attributes.

```
>>> relativedelta(days=1.5, hours=2).normalized()
relativedelta(days=1, hours=14)
```

Returns Returns a `dateutil.relativedelta.relativedelta` object.

weeks

matplotlib.dates.**seconds**(*s*)

Return seconds as days.

matplotlib.dates.**minutes**(*m*)

Return minutes as days.

matplotlib.dates.**hours**(*h*)

Return hours as days.

matplotlib.dates.**weeks**(*w*)

Return weeks as days.

CHAPTER

FIFTYTWO

DVIREAD

52.1 matplotlib.dviread

An experimental module for reading dvi files output by TeX. Several limitations make this not (currently) useful as a general-purpose dvi preprocessor, but it is currently used by the pdf backend for processing usetex text.

Interface:

```
dvi = Dvi(filename, 72)
# iterate over pages (but only one page is supported for now):
for page in dvi:
    w, h, d = page.width, page.height, page.descent
    for x,y,font,glyph,width in page.text:
        fontname = font.texname
        pointsize = font.size
        ...
    for x,y,height,width in page.boxes:
        ...
```

class matplotlib.dviread.**Dvi**(*filename*, *dpi*)

Bases: object

A dvi ("device-independent") file, as produced by TeX. The current implementation only reads the first page and does not even attempt to verify the postamble.

Initialize the object. This takes the filename as input and opens the file; actually reading the file happens when iterating through the pages of the file.

close()

Close the underlying file if it is open.

class matplotlib.dviread.**DviFont**(*scale*, *tfm*, *texname*, *vf*)

Bases: object

Object that holds a font's texname and size, supports comparison, and knows the widths of glyphs in the same units as the AFM file. There are also internal attributes (for use by dviread.py) that are *not* used for comparison.

The size is in Adobe points (converted from TeX points).

texname
Name of the font as used internally by TeX and friends. This is usually very different from any external font names, and `dviread.PsfontsMap` can be used to find the external name of the font.

size
Size of the font in Adobe points, converted from the slightly smaller TeX points.

widths
Widths of glyphs in glyph-space units, typically 1/1000ths of the point size.

size

texname

widths

class `matplotlib.dviread.`**`Encoding`**(*filename*)
Bases: `object`

Parses a *.enc file referenced from a psfonts.map style file. The format this class understands is a very limited subset of PostScript.

Usage (subject to change):

```
for name in Encoding(filename):
    whatever(name)
```

encoding

class `matplotlib.dviread.`**`PsfontsMap`**(*filename*)
Bases: `object`

A psfonts.map formatted file, mapping TeX fonts to PS fonts. Usage:

```
>>> map = PsfontsMap(find_tex_file('pdftex.map'))
>>> entry = map['ptmbo8r']
>>> entry.texname
'ptmbo8r'
>>> entry.psname
'Times-Bold'
>>> entry.encoding
'/usr/local/texlive/2008/texmf-dist/fonts/enc/dvips/base/8r.enc'
>>> entry.effects
{'slant': 0.16700000000000001}
>>> entry.filename
```

For historical reasons, TeX knows many Type-1 fonts by different names than the outside world. (For one thing, the names have to fit in eight characters.) Also, TeX's native fonts are not Type-1 but Metafont, which is nontrivial to convert to PostScript except as a bitmap. While high-quality conversions to Type-1 format exist and are shipped with modern TeX distributions, we need to know which Type-1 fonts are the counterparts of which native fonts. For these reasons a mapping is needed from internal font names to font file names.

A texmf tree typically includes mapping files called e.g. psfonts.map, pdftex.map, dvipdfm.map. psfonts.map is used by dvips, pdftex.map by pdfTeX, and dvipdfm.map by dvipdfm. psfonts.map might avoid embedding the 35 PostScript fonts (i.e., have no filename for them, as in the Times-Bold example above), while the pdf-related files perhaps only avoid the "Base 14" pdf fonts. But the user may have configured these files differently.

class `matplotlib.dviread.Tfm`(*filename*)

Bases: `object`

A TeX Font Metric file. This implementation covers only the bare minimum needed by the Dvi class.

checksum
Used for verifying against the dvi file.

design_size
Design size of the font (in what units?)

width
Width of each character, needs to be scaled by the factor specified in the dvi file. This is a dict because indexing may not start from 0.

height
Height of each character.

depth
Depth of each character.

checksum

depth

design_size

height

width

class `matplotlib.dviread.Vf`(*filename*)

Bases: `matplotlib.dviread.Dvi`

A virtual font (*.vf file) containing subroutines for dvi files.

Usage:

```
vf = Vf(filename)
glyph = vf[code]
glyph.text, glyph.boxes, glyph.width
```

`matplotlib.dvіread.`**`find_tex_file`**(*filename*, *format=None*)

Call **`kpsewhich`** to find a file in the texmf tree. If *format* is not None, it is used as the value for the `--format` option.

Apparently most existing TeX distributions on Unix-like systems use kpathsea. I hear MikTeX (a popular distribution on Windows) doesn't use kpathsea, so what do we do? (TODO)

See also:

Kpathsea documentation The library that **`kpsewhich`** is part of.

`matplotlib.dviread.`**`ord`**(*x*)

CHAPTER

FIFTYTHREE

FIGURE

53.1 matplotlib.figure

The figure module provides the top-level *Artist*, the *Figure*, which contains all the plot elements. The following classes are defined

SubplotParams control the default spacing of the subplots

Figure top level container for all plot elements

class matplotlib.figure.**AxesStack**

Bases: *matplotlib.cbook.Stack*

Specialization of the Stack to handle all tracking of Axes in a Figure. This stack stores `key, (ind, axes)` pairs, where:

- **key** should be a hash of the args and kwargs used in generating the Axes.
- **ind** is a serial number for tracking the order in which axes were added.

The AxesStack is a callable, where `ax_stack()` returns the current axes. Alternatively the *current_key_axes()* will return the current key and associated axes.

add(*key*, *a*)

Add Axes *a*, with key *key*, to the stack, and return the stack.

If *a* is already on the stack, don't add it again, but return *None*.

as_list()

Return a list of the Axes instances that have been added to the figure

bubble(*a*)

Move the given axes, which must already exist in the stack, to the top.

current_key_axes()

Return a tuple of `(key, axes)` for the active axes.

If no axes exists on the stack, then returns `(None, None)`.

get(*key*)

Return the Axes instance that was added with *key*. If it is not present, return None.

remove(*a*)

Remove the axes from the stack.

class `matplotlib.figure.Figure`(*figsize=None*, *dpi=None*, *facecolor=None*, *edgecolor=None*, *linewidth=0.0*, *frameon=None*, *subplotpars=None*, *tight_layout=None*)

Bases: `matplotlib.artist.Artist`

The Figure instance supports callbacks through a *callbacks* attribute which is a `matplotlib.cbook.CallbackRegistry` instance. The events you can connect to are 'dpi_changed', and the callback will be called with `func(fig)` where fig is the `Figure` instance.

patch The figure patch is drawn by a `matplotlib.patches.Rectangle` instance

suppressComposite For multiple figure images, the figure will make composite images depending on the renderer option_image_nocomposite function. If suppressComposite is True|False, this will override the renderer.

figsize w,h tuple in inches

dpi Dots per inch

facecolor The figure patch facecolor; defaults to rc `figure.facecolor`

edgecolor The figure patch edge color; defaults to rc `figure.edgecolor`

linewidth The figure patch edge linewidth; the default linewidth of the frame

frameon If *False*, suppress drawing the figure frame

subplotpars A `SubplotParams` instance, defaults to rc

tight_layout If *False* use *subplotpars*; if *True* adjust subplot parameters using `tight_layout()` with default padding. When providing a dict containing the keys `pad`, `w_pad`, `h_pad` and `rect`, the default `tight_layout()` paddings will be overridden. Defaults to rc `figure.autolayout`.

`add_axes`(**args*, ***kwargs*)

Add an axes at position *rect* [*left*, *bottom*, *width*, *height*] where all quantities are in fractions of figure width and height. kwargs are legal `Axes` kwargs plus *projection* which sets the projection type of the axes. (For backward compatibility, `polar=True` may also be provided, which is equivalent to `projection='polar'`). Valid values for *projection* are: ['aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear']. Some of these projections support additional kwargs, which may be provided to `add_axes()`. Typical usage:

```
rect = l,b,w,h
fig.add_axes(rect)
fig.add_axes(rect, frameon=False, axisbg='g')
fig.add_axes(rect, polar=True)
fig.add_axes(rect, projection='polar')
fig.add_axes(ax)
```

If the figure already has an axes with the same parameters, then it will simply make that axes current and return it. If you do not want this behavior, e.g., you want to force the creation of a new Axes, you must use a unique set of args and kwargs. The axes `label` attribute has been exposed for this purpose. e.g., if you want two axes that are otherwise identical to be added to the figure, make sure you give them unique labels:

```
fig.add_axes(rect, label='axes1')
fig.add_axes(rect, label='axes2')
```

In rare circumstances, add_axes may be called with a single argument, an Axes instance already created in the present figure but not in the figure's list of axes. For example, if an axes has been removed with `delaxes()`, it can be restored with:

```
fig.add_axes(ax)
```

In all cases, the `Axes` instance will be returned.

In addition to *projection*, the following kwargs are supported:

Property	Description
`adjustable`	['box' | 'datalim' | 'box-forced']
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`anchor`	unknown
`animated`	[True | False]
`aspect`	unknown
`autoscale_on`	unknown
`autoscalex_on`	unknown
`autoscaley_on`	unknown
`axes`	an `Axes` instance
`axes_locator`	unknown
`axis_bgcolor`	any matplotlib color - see `colors()`
`axisbelow`	[*True* | *False*]
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color_cycle`	unknown
`contains`	a callable function
`figure`	unknown
`frame_on`	[*True* | *False*]
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`navigate`	[*True* | *False*]
`navigate_mode`	unknown
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	unknown
`rasterization_zorder`	unknown
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`title`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`xbound`	unknown
`xlabel`	unknown
`xlim`	length 2 sequence of floats
`xmargin`	unknown

Continued on next page

Table 53.1 – continued from previous page

Property	Description
xscale	['linear' | 'log' | 'logit' | 'symlog']
xticklabels	sequence of strings
xticks	sequence of floats
ybound	unknown
ylabel	unknown
ylim	length 2 sequence of floats
ymargin	unknown
yscale	['linear' | 'log' | 'logit' | 'symlog']
yticklabels	sequence of strings
yticks	sequence of floats
zorder	any number

add_axobserver(*func*)

whenever the axes state change, `func(self)` will be called

add_subplot(**args*, ***kwargs*)

Add a subplot. Examples:

```
fig.add_subplot(111)

# equivalent but more general
fig.add_subplot(1,1,1)

# add subplot with red background
fig.add_subplot(212, axisbg='r')

# add a polar subplot
fig.add_subplot(111, projection='polar')

# add Subplot instance sub
fig.add_subplot(sub)
```

kwargs are legal `Axes` kwargs plus *projection*, which chooses a projection type for the axes. (For backward compatibility, *polar=True* may also be provided, which is equivalent to *projection='polar'*). Valid values for *projection* are: ['aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear']. Some of these projections support additional *kwargs*, which may be provided to `add_axes()`.

The `Axes` instance will be returned.

If the figure already has a subplot with key (*args*, *kwargs*) then it will simply make that subplot current and return it.

See also:

`subplot()` for an explanation of the args.

The following kwargs are supported:

Property	Description
adjustable	['box' | 'datalim' | 'box-forced']
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
anchor	unknown
animated	[True | False]
aspect	unknown
autoscale_on	unknown
autoscalex_on	unknown
autoscaley_on	unknown
axes	an *Axes* instance
axes_locator	unknown
axis_bgcolor	any matplotlib color - see *colors()*
axisbelow	[*True* | *False*]
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color_cycle	unknown
contains	a callable function
figure	unknown
frame_on	[*True* | *False*]
gid	an id string
label	string or anything printable with '%s' conversion.
navigate	[*True* | *False*]
navigate_mode	unknown
path_effects	unknown
picker	[None|float|boolean|callable]
position	unknown
rasterization_zorder	unknown
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
title	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
xbound	unknown
xlabel	unknown
xlim	length 2 sequence of floats
xmargin	unknown
xscale	['linear' | 'log' | 'logit' | 'symlog']
xticklabels	sequence of strings
xticks	sequence of floats
ybound	unknown
ylabel	unknown

Continued on next page

Table 53.2 – continued from previous page

Property	Description
ylim	length 2 sequence of floats
ymargin	unknown
yscale	['linear' | 'log' | 'logit' | 'symlog']
yticklabels	sequence of strings
yticks	sequence of floats
zorder	any number

autofmt_xdate(*bottom=0.2*, *rotation=30*, *ha='right'*)

Date ticklabels often overlap, so it is useful to rotate them and right align them. Also, a common use case is a number of subplots with shared xaxes where the x-axis is date data. The ticklabels are often long, and it helps to rotate them on the bottom subplot and turn them off on other subplots, as well as turn off xlabels.

bottom The bottom of the subplots for *subplots_adjust()*

rotation The rotation of the xtick labels

ha The horizontal alignment of the xticklabels

axes

Read-only: list of axes in Figure

clear()

Clear the figure – synonym for *clf()*.

clf(*keep_observers=False*)

Clear the figure.

Set *keep_observers* to True if, for example, a gui widget is tracking the axes in the figure.

colorbar(*mappable*, *cax=None*, *ax=None*, *use_gridspec=True*, ***kw*)

Create a colorbar for a ScalarMappable instance, *mappable*.

Documentation for the pylab thin wrapper:

Add a colorbar to a plot.

Function signatures for the *pyplot* interface; all but the first are also method signatures for the *colorbar()* method:

```
colorbar(**kwargs)
colorbar(mappable, **kwargs)
colorbar(mappable, cax=cax, **kwargs)
colorbar(mappable, ax=ax, **kwargs)
```

arguments:

mappable the Image, ContourSet, etc. to which the colorbar applies; this argument is mandatory for the *colorbar()* method but optional for the *colorbar()* function, which sets the default to the current image.

keyword arguments:

cax None | axes object into which the colorbar will be drawn

ax None | parent axes object(s) from which space for a new colorbar axes will be stolen. If a list of axes is given they will all be resized to make room for the colorbar axes.

use_gridspec False | If *cax* is None, a new *cax* is created as an instance of Axes. If *ax* is an instance of Subplot and *use_gridspec* is True, *cax* is created as an instance of Subplot using the grid_spec module.

Additional keyword arguments are of two kinds:

axes properties:

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions
anchor	(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes
panchor	(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes' anchor will be unchanged

colorbar properties:

Property	Description
extend	['neither' | 'both' | 'min' | 'max'] If not 'neither', make pointed end(s) for out-of- range values. These are set for a given colormap using the colormap set_under and set_over methods.
extendfrac	[*None* | 'auto' | length | lengths] If set to *None*, both the minimum and maximum triangular colorbar extensions with have a length of 5% of the interior colorbar length (this is the default setting). If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional'). If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.
extendrect	[*False* | *True*] If *False* the minimum and maximum colorbar extensions will be triangular (the default). If *True* the extensions will be rectangular.
spacing	['uniform' | 'proportional'] Uniform spacing gives each discrete color the same space; proportional makes the space proportional to the data interval.
ticks	[None | list of ticks | Locator object] If None, ticks are determined automatically from the input.
format	[None | format string | Formatter object] If None, the `ScalarFormatter` is used. If a format string is given, e.g., '%.3f', that is used. An alternative `Formatter` object may be given instead.
drawedges	[False | True] If true, draw lines at color boundaries.

The following will probably be useful only in the context of indexed colors (that is, when the mappable has norm=NoNorm()), or other unusual circumstances.

Property	Description
boundaries	None or a sequence
values	None or a sequence which must be of length 1 less than the sequence of *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used.

If *mappable* is a `ContourSet`, its *extend* kwarg is included automatically.

Note that the *shrink* kwarg provides a simple way to keep a vertical colorbar, for example, from being taller than the axes of the mappable to which the colorbar is attached; but it is a manual method requiring some trial and error. If the colorbar is too tall (or a horizontal colorbar is too wide) use a smaller value of *shrink*.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the axes properties kwargs.

It is known that some vector graphics viewer (svg and pdf) renders white gaps between segments of the colorbar. This is due to bugs in the viewers not matplotlib. As a workaround the colorbar can be rendered with overlapping segments:

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However this has negative consequences in other circumstances. Particularly with semi transparent images (alpha < 1) and colorbar extensions and is not enabled by default see (issue #1188).

returns: `Colorbar` instance; see also its base class, `ColorbarBase`. Call the `set_label()` method to label the colorbar.

contains(*mouseevent*)
Test whether the mouse event occurred on the figure.

Returns True,{}

delaxes(*a*)
remove a from the figure and update the current axes

dpi

draw(*artist*, *renderer*, **args*, ***kwargs*)
Render the figure using `matplotlib.backend_bases.RendererBase` instance *renderer*.

draw_artist(*a*)
draw `matplotlib.artist.Artist` instance *a* only – this is available only after the figure is drawn

figimage(*X*, *xo=0*, *yo=0*, *alpha=None*, *norm=None*, *cmap=None*, *vmin=None*, *vmax=None*, *origin=None*, *resize=False*, ***kwargs*)
Adds a non-resampled image to the figure.

call signatures:

```
figimage(X, **kwargs)
```

adds a non-resampled array *X* to the figure.

```
figimage(X, xo, yo)
```

with pixel offsets *xo*, *yo*,

X must be a float array:

- If *X* is MxN, assume luminance (grayscale)

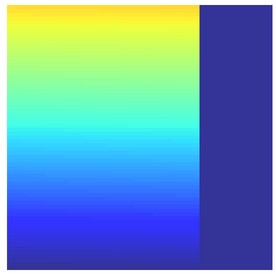
- If *X* is MxNx3, assume RGB
- If *X* is MxNx4, assume RGBA

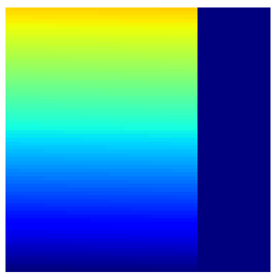
Optional keyword arguments:

Keyword	Description
resize	a boolean, True or False. If "True", then re-size the Figure to match the given image size.
xo or yo	An integer, the *x* and *y* image offset in pixels
cmap	a `matplotlib.colors.Colormap` instance, e.g., cm.jet. If *None*, default to the rc `image.cmap` value
norm	a `matplotlib.colors.Normalize` instance. The default is normalization(). This scales luminance -> 0-1
vmin|vmax	are used to scale a luminance image to 0-1. If either is *None*, the min and max of the luminance values will be used. Note if you pass a norm instance, the settings for *vmin* and *vmax* will be ignored.
alpha	the alpha blending value, default is *None*
origin	['upper' | 'lower'] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc image.origin value

figimage complements the axes image (`imshow()`) which will be resampled to fit the current axes. If you want a resampled image to fill the entire figure, you can define an `Axes` with size [0,1,0,1].

An `matplotlib.image.FigureImage` instance is returned.





Additional kwargs are Artist kwargs passed on to *FigureImage*

gca(****kwargs**)

Get the current axes, creating one if necessary

The following kwargs are supported for ensuring the returned axes adheres to the given projection etc., and for axes creation if the active axes does not exist:

Property	Description
adjustable	['box' | 'datalim' | 'box-forced']
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
anchor	unknown
animated	[True | False]
aspect	unknown
autoscale_on	unknown
autoscalex_on	unknown
autoscaley_on	unknown
axes	an *Axes* instance
axes_locator	unknown
axis_bgcolor	any matplotlib color - see *colors()*
axisbelow	[*True* | *False*]
clip_box	a *matplotlib.transforms.Bbox* instance

Continued on next page

Table 53.3 – continued from previous page

Property	Description
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color_cycle	unknown
contains	a callable function
figure	unknown
frame_on	[*True* | *False*]
gid	an id string
label	string or anything printable with '%s' conversion.
navigate	[*True* | *False*]
navigate_mode	unknown
path_effects	unknown
picker	[None|float|boolean|callable]
position	unknown
rasterization_zorder	unknown
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
title	unknown
transform	Transform instance
url	a url string
visible	[True | False]
xbound	unknown
xlabel	unknown
xlim	length 2 sequence of floats
xmargin	unknown
xscale	['linear' | 'log' | 'logit' | 'symlog']
xticklabels	sequence of strings
xticks	sequence of floats
ybound	unknown
ylabel	unknown
ylim	length 2 sequence of floats
ymargin	unknown
yscale	['linear' | 'log' | 'logit' | 'symlog']
yticklabels	sequence of strings
yticks	sequence of floats
zorder	any number

get_axes()

get_children()
get a list of artists contained in the figure

get_default_bbox_extra_artists()

get_dpi()
Return the dpi as a float

get_edgecolor()
Get the edge color of the Figure rectangle

get_facecolor()
Get the face color of the Figure rectangle

get_figheight()
Return the figheight as a float

get_figwidth()
Return the figwidth as a float

get_frameon()
get the boolean indicating frameon

get_size_inches()
Returns the current size of the figure in inches (1in == 2.54cm) as an numpy array.

Returns size : ndarray
The size of the figure in inches

See also:

`matplotlib.Figure.set_size_inches`

get_tight_layout()
Return the Boolean flag, True to use :meth'tight_layout' when drawing.

get_tightbbox(*renderer*)
Return a (tight) bounding box of the figure in inches.

It only accounts axes title, axis labels, and axis ticklabels. Needs improvement.

get_window_extent(**args*, ***kwargs*)
get the figure bounding box in display space; kwargs are void

ginput(*n=1*, *timeout=30*, *show_clicks=True*, *mouse_add=1*, *mouse_pop=3*, *mouse_stop=2*)
Call signature:

```
ginput(self, n=1, timeout=30, show_clicks=True,
       mouse_add=1, mouse_pop=3, mouse_stop=2)
```

Blocking call to interact with the figure.

This will wait for *n* clicks from the user and return a list of the coordinates of each click.

If *timeout* is zero or negative, does not timeout.

If *n* is zero or negative, accumulate clicks until a middle click (or potentially both mouse buttons at once) terminates the input.

Right clicking cancels last input.

The buttons used for the various actions (adding points, removing points, terminating the inputs) can be overriden via the arguments *mouse_add*, *mouse_pop* and *mouse_stop*, that give the associated mouse button: 1 for left, 2 for middle, 3 for right.

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

hold(*b=None*)

Set the hold state. If hold is None (default), toggle the hold state. Else set the hold state to boolean value b.

e.g.:

```
hold()      # toggle hold
hold(True)  # hold is on
hold(False) # hold is off
```

legend(*handles*, *labels*, **args*, ***kwargs*)

Place a legend in the figure. Labels are a sequence of strings, handles is a sequence of `Line2D` or `Patch` instances, and loc can be a string or an integer specifying the legend location

USAGE:

```
legend( (line1, line2, line3),
        ('label1', 'label2', 'label3'),
        'upper right')
```

The *loc* location codes are:

```
'best' : 0,          (currently not supported for figure legends)
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10,
```

loc can also be an (x,y) tuple in figure coords, which specifies the lower left of the legend box. figure coords are (0,0) is the left, bottom of the figure and 1,1 is the right, top.

Keyword arguments:

***prop*: [*None* | FontProperties | dict]** A `matplotlib.font_manager.FontProperties` instance. If *prop* is a dictionary, a new instance will be created with *prop*. If *None*, use rc settings.

***numpoints*: integer** The number of points in the legend line, default is 4

***scatterpoints*: integer** The number of points in the legend line, default is 4

***scatteryoffsets*: list of floats** a list of yoffsets for scatter symbols in legend

***markerscale*: [*None* | scalar]** The relative size of legend markers vs. original. If *None*, use rc settings.

***markerfirst*: [*True* | *False*]** if *True*, legend marker is placed to the left of the legend label if *False*, legend marker is placed to the right of the legend label

***fancybox*: [*None* | *False* | *True*]** if *True*, draw a frame with a round fancybox. If *None*, use rc

***shadow*: [*None* | *False* | *True*]** If *True*, draw a shadow behind legend. If *None*, use rc settings.

ncol [integer] number of columns. default is 1

mode [["expand" | *None*]] if mode is "expand", the legend will be horizontally expanded to fill the axes area (or *bbox_to_anchor*)

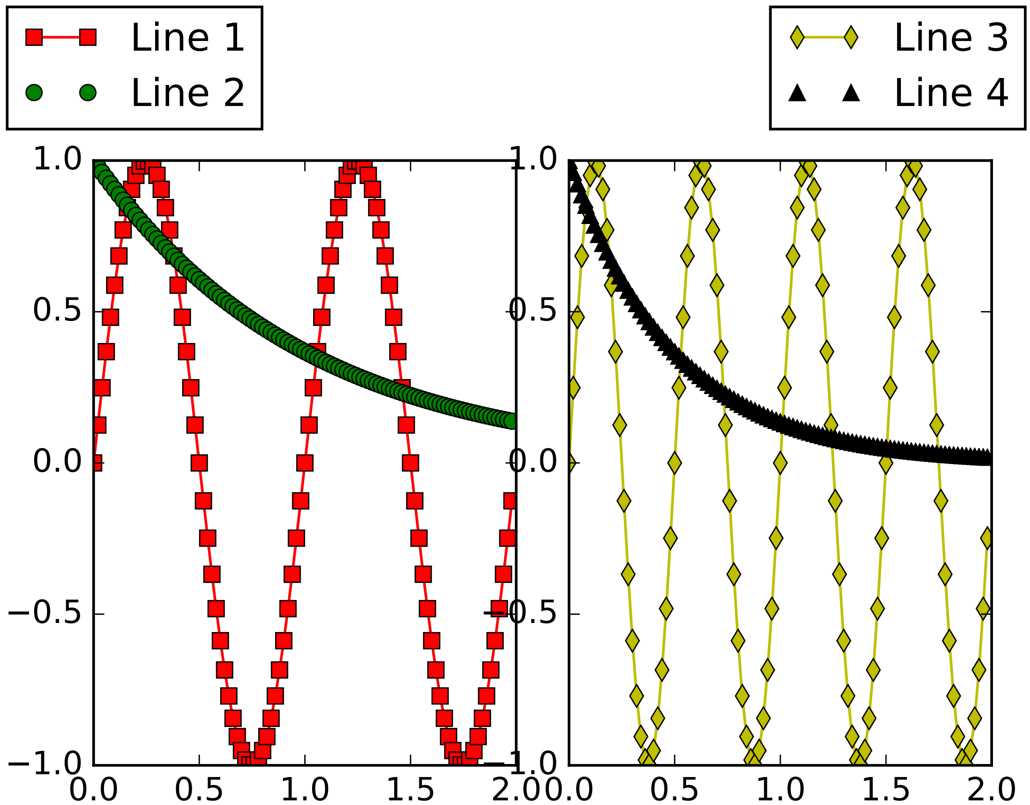
title [string] the legend title

Padding and spacing between various elements use following keywords parameters. The dimensions of these values are given as a fraction of the fontsize. Values from rcParams will be used if None.

Keyword	Description
borderpad	the fractional whitespace inside the legend border
labelspacing	the vertical space between the legend entries
handlelength	the length of the legend handles
handletextpad	the pad between the legend handle and text
borderaxespad	the pad between the axes and legend border
columnspacing	the spacing between columns

Note: Not all kinds of artist are supported by the legend. See LINK (FIXME) for details.

Example:



savefig(**args*, ***kwargs*)

Save the current figure.

Call signature:

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False, bbox_inches=None, pad_inches=0.1,
        frameon=None)
```

The output formats available depend on the backend being used.

Arguments:

***fname*:** A string containing a path to a filename, or a Python file-like object, or possibly some backend-dependent object such as `PdfPages`.

If *format* is *None* and *fname* is a string, the output format is deduced from the extension of the filename. If the filename has no extension, the value of the rc parameter `savefig.format` is used.

If *fname* is not a string, remember to specify *format* to ensure that the correct backend is used.

Keyword arguments:

***dpi*: [*None* | `scalar > 0` | 'figure']** The resolution in dots per inch. If *None* it will default to the value `savefig.dpi` in the matplotlibrc file. If 'figure' it will set the dpi to be the value of the figure.

***facecolor, edgecolor*:** the colors of the figure rectangle

***orientation*: ['landscape' | 'portrait']** not supported on all backends; currently only on postscript output

***papertype*:** One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

***format*:** One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

***transparent*:** If *True*, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

***frameon*:** If *True*, the figure patch will be colored, if *False*, the figure background will be transparent. If not provided, the rcParam 'savefig.frameon' will be used.

***bbox_inches*:** Bbox in inches. Only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

***pad_inches*:** Amount of padding around the figure when bbox_inches is 'tight'.

***bbox_extra_artists*:** A list of extra artists that will be considered when the tight bbox is calculated.

`sca`(*a*)
Set the current axes to be a and return a

`set_canvas`(*canvas*)
Set the canvas that contains the figure

ACCEPTS: a FigureCanvas instance

`set_dpi`(*val*)
Set the dots-per-inch of the figure

ACCEPTS: float

`set_edgecolor`(*color*)
Set the edge color of the Figure rectangle

ACCEPTS: any matplotlib color - see help(colors)

`set_facecolor`(*color*)
Set the face color of the Figure rectangle

ACCEPTS: any matplotlib color - see help(colors)

`set_figheight`(*val*, *forward=False*)
Set the height of the figure in inches

ACCEPTS: float

`set_figwidth`(*val*, *forward=False*)
Set the width of the figure in inches

ACCEPTS: float

set_frameon(*b*)

Set whether the figure frame (background) is displayed or invisible

ACCEPTS: boolean

set_size_inches(*w*, *h*, *forward=False*)

Set the figure size in inches (1in == 2.54cm)

Usage:

```
fig.set_size_inches(w,h)  # OR
fig.set_size_inches((w,h) )
```

optional kwarg *forward=True* will cause the canvas size to be automatically updated; e.g., you can resize the figure window from the shell

ACCEPTS: a w,h tuple with w,h in inches

See also:

`matplotlib.Figure.get_size_inches`

set_tight_layout(*tight*)

Set whether `tight_layout()` is used upon drawing. If None, the rcParams['figure.autolayout'] value will be set.

When providing a dict containing the keys `pad`, `w_pad`, `h_pad` and `rect`, the default `tight_layout()` paddings will be overridden.

ACCEPTS: [True | False | dict | None]

show(*warn=True*)

If using a GUI backend with pyplot, display the figure window.

If the figure was not created using `figure()`, it will lack a `FigureManagerBase`, and will raise an AttributeError.

For non-GUI backends, this does nothing, in which case a warning will be issued if *warn* is True (default).

subplots_adjust(**args*, ***kwargs*)

Call signature:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                    wspace=None, hspace=None)
```

Update the `SubplotParams` with *kwargs* (defaulting to rc when *None*) and update the subplot locations

suptitle(*t*, ***kwargs*)

Add a centered title to the figure.

kwargs are `matplotlib.text.Text` properties. Using figure coordinates, the defaults are:

x [0.5] The x location of the text in figure coords

y [0.98] The y location of the text in figure coords

horizontalalignment ['center'] The horizontal alignment of the text

verticalalignment ['top'] The vertical alignment of the text

A `matplotlib.text.Text` instance is returned.

Example:

```
fig.suptitle('this is the figure title', fontsize=12)
```

text(*x*, *y*, *s*, **args*, ***kwargs*)

Add text to figure.

Call signature:

```
text(x, y, s, fontdict=None, **kwargs)
```

Add text to figure at location *x*, *y* (relative 0-1 coords). See `text()` for the meaning of the other arguments.

kwargs control the `Text` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	FancyBboxPatch prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or name or fontname or fontfamily	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']
`rotation_mode`	unknown
`size` or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large'
`sketch_params`	unknown
`snap`	unknown

Table 53.4 – continued fror

Property	Description
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'c
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	*Transform* instance
url	a url string
usetex	unknown
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' |
wrap	unknown
x	float
y	float
zorder	any number

tight_layout(*renderer=None*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Adjust subplot parameters to give specified padding.

Parameters:

pad [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.

h_pad*, *w_pad [float] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.

rect [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

waitforbuttonpress(*timeout=-1*)

Call signature:

```
waitforbuttonpress(self, timeout=-1)
```

Blocking call to interact with the figure.

This will return True is a key was pressed, False if a mouse button was pressed and None if *timeout* was reached without either being pressed.

If *timeout* is negative, does not timeout.

class matplotlib.figure.**SubplotParams**(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)

Bases: `object`

A class to hold the parameters for a subplot

All dimensions are fraction of the figure width or height. All values default to their rc params

The following attributes are available

left [0.125] The left side of the subplots of the figure
right [0.9] The right side of the subplots of the figure
bottom [0.1] The bottom of the subplots of the figure
top [0.9] The top of the subplots of the figure
wspace [0.2] The amount of width reserved for blank space between subplots
hspace [0.2] The amount of height reserved for white space between subplots

update(*left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*)
Update the current values. If any kwarg is None, default to the current value, if set, otherwise to rc

matplotlib.figure.**figaspect**(*arg*)
Create a figure with specified aspect ratio. If *arg* is a number, use that aspect ratio. If *arg* is an array, figaspect will determine the width and height for a figure that would fit array preserving aspect ratio. The figure width, height in inches are returned. Be sure to create an axes with equal with and height, e.g.,

Example usage:

```
# make a figure twice as tall as it is wide
w, h = figaspect(2.)
fig = Figure(figsize=(w,h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)

# make a figure with the proper aspect for an array
A = rand(5,3)
w, h = figaspect(A)
fig = Figure(figsize=(w,h))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.imshow(A, **kwargs)
```

Thanks to Fernando Perez for this function

CHAPTER

FIFTYFOUR

FINANCE

54.1 matplotlib.finance

A collection of functions for collecting, analyzing and plotting financial data. User contributions welcome!

This module is deprecated in 1.4 and will be moved to `mpl_toolkits` or it's own project in the future.

`matplotlib.finance.`**`candlestick2_ochl`**(*ax*, *opens*, *closes*, *highs*, *lows*, *width=4*, *colorup='k'*, *colordown='r'*, *alpha=0.75*)

Represent the open, close as a bar line and high low range as a vertical line.

Preserves the original argument order.

Parameters ax : `Axes`

an Axes instance to plot to

opens : sequence

sequence of opening values

closes : sequence

sequence of closing values

highs : sequence

sequence of high values

lows : sequence

sequence of low values

ticksize : int

size of open and close ticks in points

colorup : color

the color of the lines where close >= open

colordown : color

the color of the lines where close < open

alpha : float

bar transparency

Returns ret : tuple

(lineCollection, barCollection)

`matplotlib.finance.`**`candlestick2_ohlc`**(*ax*, *opens*, *highs*, *lows*, *closes*, *width=4*, *colorup='k'*, *colordown='r'*, *alpha=0.75*)

Represent the open, close as a bar line and high low range as a vertical line.

NOTE: this code assumes if any value open, low, high, close is missing they all are missing

Parameters ax : `Axes`

an Axes instance to plot to

opens : sequence
sequence of opening values

highs : sequence
sequence of high values

lows : sequence
sequence of low values

closes : sequence
sequence of closing values

ticksize : int
size of open and close ticks in points

colorup : color
the color of the lines where close >= open

colordown : color
the color of the lines where close < open

alpha : float
bar transparency

Returns ret : tuple
(lineCollection, barCollection)

`matplotlib.finance.`**`candlestick_ochl`**(*ax*, *quotes*, *width=0.2*, *colorup='k'*, *colordown='r'*, *alpha=1.0*)

Plot the time, open, close, high, low as a vertical line ranging from low to high. Use a rectangular bar to represent the open-close span. If close >= open, use colorup to color the bar, otherwise use colordown

Parameters ax : `Axes`
an Axes instance to plot to

quotes : sequence of (time, open, close, high, low, ...) sequences
As long as the first 5 elements are these values, the record can be as long as you want (e.g., it may store volume).

time must be in float days format - see date2num

width : float
fraction of a day for the rectangle width

colorup : color
the color of the rectangle where close >= open

colordown : color
the color of the rectangle where close < open

alpha : float
the rectangle alpha level

Returns ret : tuple
returns (lines, patches) where lines is a list of lines added and patches is a list of the rectangle patches added

`matplotlib.finance.`**`candlestick_ohlc`**(*ax*, *quotes*, *width=0.2*, *colorup='k'*, *colordown='r'*, *alpha=1.0*)

Plot the time, open, high, low, close as a vertical line ranging from low to high. Use a rectangular bar to represent the open-close span. If close >= open, use colorup to color the bar, otherwise use colordown

Parameters ax : `Axes`

an Axes instance to plot to

quotes : sequence of (time, open, high, low, close, ...) sequences

As long as the first 5 elements are these values, the record can be as long as you want (e.g., it may store volume).

time must be in float days format - see date2num

width : float

fraction of a day for the rectangle width

colorup : color

the color of the rectangle where close >= open

colordown : color

the color of the rectangle where close < open

alpha : float

the rectangle alpha level

Returns ret : tuple

returns (lines, patches) where lines is a list of lines added and patches is a list of the rectangle patches added

`matplotlib.finance.`**`fetch_historical_yahoo`**(*ticker*, *date1*, *date2*, *cachename=None*, *dividends=False*)

Fetch historical data for ticker between date1 and date2. date1 and date2 are date or datetime instances, or (year, month, day) sequences.

Parameters ticker : str

ticker

date1 : sequence of form (year, month, day), `datetime`, or `date`

start date

date2 : sequence of form (year, month, day), `datetime`, or `date`

end date

cachename : str

cachename is the name of the local file cache. If None, will default to the md5 hash or the url (which incorporates the ticker and date range)

dividends : bool

set dividends=True to return dividends instead of price data. With this option set, parse functions will not work

Returns file_handle : file handle

a file handle is returned

Examples

```
>>> fh = fetch_historical_yahoo('^GSPC', (2000, 1, 1), (2001, 12, 31))
```

`matplotlib.finance.`**`index_bar`**(*ax*, *vals*, *facecolor='b'*, *edgecolor='l'*, *width=4*, *alpha=1.0*)

Add a bar collection graph with height vals (-1 is missing).

Parameters ax : `Axes`

an Axes instance to plot to

vals : sequence

a sequence of values

facecolor : color
the color of the bar face

edgecolor : color
the color of the bar edges

width : int
the bar width in points

alpha : float
bar transparency

Returns ret : `barCollection`
The `barrCollection` added to the axes

`matplotlib.finance.`**`md5`**`(`*x*`)`

`matplotlib.finance.`**`parse_yahoo_historical_ochl`**`(`*fh*, *adjusted=True*, *asobject=False*`)`

Parse the historical data in file handle fh from yahoo finance.

Parameters adjusted : bool
If True (default) replace open, close, high, low prices with their adjusted values. The adjustment is by a scale factor, S = adjusted_close/close. Adjusted prices are actual prices multiplied by S.

Volume is not adjusted as it is already backward split adjusted by Yahoo. If you want to compute dollars traded, multiply volume by the adjusted close, regardless of whether you choose adjusted = True|False.

asobject : bool or None
If False (default for compatibility with earlier versions) return a list of tuples containing

d, open, close, high, low, volume

If None (preferred alternative to False), return a 2-D ndarray corresponding to the list of tuples.

Otherwise return a numpy recarray with

date, year, month, day, d, open, close, high, low, volume, adjusted_close

where d is a floating poing representation of date, as returned by date2num, and date is a python standard library datetime.date instance.

The name of this kwarg is a historical artifact. Formerly, True returned a cbook Bunch holding 1-D ndarrays. The behavior of a numpy recarray is very similar to the Bunch.

`matplotlib.finance.`**`parse_yahoo_historical_ohlc`**`(`*fh*, *adjusted=True*, *asobject=False*`)`

Parse the historical data in file handle fh from yahoo finance.

Parameters adjusted : bool
If True (default) replace open, high, low, close prices with their adjusted values. The adjustment is by a scale factor, S = adjusted_close/close. Adjusted prices are actual prices multiplied by S.

Volume is not adjusted as it is already backward split adjusted by Yahoo. If you want to compute dollars traded, multiply volume by the adjusted close, regardless of whether you choose adjusted = True|False.

asobject : bool or None

If False (default for compatibility with earlier versions) return a list of tuples containing

d, open, high, low, close, volume

If None (preferred alternative to False), return a 2-D ndarray corresponding to the list of tuples.

Otherwise return a numpy recarray with

date, year, month, day, d, open, high, low, close, volume, adjusted_close

where d is a floating poing representation of date, as returned by date2num, and date is a python standard library datetime.date instance.

The name of this kwarg is a historical artifact. Formerly, True returned a cbook Bunch holding 1-D ndarrays. The behavior of a numpy recarray is very similar to the Bunch.

`matplotlib.finance.`**`plot_day_summary2_ochl`**(*ax*, *opens*, *closes*, *highs*, *lows*, *ticksize=4*, *colorup='k'*, *colordown='r'*)

Represent the time, open, close, high, low, as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

Parameters ax : `Axes`

an Axes instance to plot to

opens : sequence

sequence of opening values

closes : sequence

sequence of closing values

highs : sequence

sequence of high values

lows : sequence

sequence of low values

ticksize : int

size of open and close ticks in points

colorup : color

the color of the lines where close >= open

colordown : color

the color of the lines where close < open

Returns ret : list

a list of lines added to the axes

`matplotlib.finance.`**`plot_day_summary2_ohlc`**(*ax*, *opens*, *highs*, *lows*, *closes*, *ticksize=4*, *colorup='k'*, *colordown='r'*)

Represent the time, open, high, low, close as a vertical line ranging from low to high. The left tick is the open and the right tick is the close. *opens*, *highs*, *lows* and *closes* must have the same length. NOTE: this code assumes if any value open, high, low, close is missing (*-1*) they all are missing

Parameters ax : `Axes`

an Axes instance to plot to

opens : sequence

sequence of opening values

highs : sequence

sequence of high values

lows : sequence
sequence of low values

closes : sequence
sequence of closing values

ticksize : int
size of open and close ticks in points

colorup : color
the color of the lines where close >= open

colordown : color
the color of the lines where close < open

Returns ret : list
a list of lines added to the axes

`matplotlib.finance.`**`plot_day_summary_oclh`**(*ax*, *quotes*, *ticksize=3*, *colorup='k'*, *colordown='r'*)

Plots day summary

Represent the time, open, close, high, low as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

Parameters ax : `Axes`
an `Axes` instance to plot to

quotes : sequence of (time, open, close, high, low, ...) sequences
data to plot. time must be in float date format - see date2num

ticksize : int
open/close tick marker in points

colorup : color
the color of the lines where close >= open

colordown : color
the color of the lines where close < open

Returns lines : list
list of tuples of the lines added (one tuple per quote)

`matplotlib.finance.`**`plot_day_summary_ohlc`**(*ax*, *quotes*, *ticksize=3*, *colorup='k'*, *colordown='r'*)

Plots day summary

Represent the time, open, high, low, close as a vertical line ranging from low to high. The left tick is the open and the right tick is the close.

Parameters ax : `Axes`
an `Axes` instance to plot to

quotes : sequence of (time, open, high, low, close, ...) sequences
data to plot. time must be in float date format - see date2num

ticksize : int
open/close tick marker in points

colorup : color
the color of the lines where close >= open

colordown : color
the color of the lines where close < open

Returns lines : list

list of tuples of the lines added (one tuple per quote)

matplotlib.finance.**quotes_historical_yahoo_ochl**(*ticker*, *date1*, *date2*, *asobject=False*, *adjusted=True*, *cachename=None*)

Get historical data for ticker between date1 and date2.

See `parse_yahoo_historical()` for explanation of output formats and the *asobject* and *adjusted* kwargs.

Parameters ticker : str
stock ticker

date1 : sequence of form (year, month, day), `datetime`, or `date`
start date

date2 : sequence of form (year, month, day), `datetime`, or `date`
end date

cachename : str or `None`
is the name of the local file cache. If None, will default to the md5 hash or the url (which incorporates the ticker and date range)

Examples

```
>>> sp = f.quotes_historical_yahoo_ochl('^GSPC', d1, d2,
                             asobject=True, adjusted=True)
>>> returns = (sp.open[1:] - sp.open[:-1])/sp.open[1:]
>>> [n,bins,patches] = hist(returns, 100)
>>> mu = mean(returns)
>>> sigma = std(returns)
>>> x = normpdf(bins, mu, sigma)
>>> plot(bins, x, color='red', lw=2)
```

matplotlib.finance.**quotes_historical_yahoo_ohlc**(*ticker*, *date1*, *date2*, *asobject=False*, *adjusted=True*, *cachename=None*)

Get historical data for ticker between date1 and date2.

See `parse_yahoo_historical()` for explanation of output formats and the *asobject* and *adjusted* kwargs.

Parameters ticker : str
stock ticker

date1 : sequence of form (year, month, day), `datetime`, or `date`
start date

date2 : sequence of form (year, month, day), `datetime`, or `date`
end date

cachename : str or `None`
is the name of the local file cache. If None, will default to the md5 hash or the url (which incorporates the ticker and date range)

Examples

```
>>> sp = f.quotes_historical_yahoo_ohlc('^GSPC', d1, d2,
                             asobject=True, adjusted=True)
>>> returns = (sp.open[1:] - sp.open[:-1])/sp.open[1:]
>>> [n,bins,patches] = hist(returns, 100)
>>> mu = mean(returns)
>>> sigma = std(returns)
>>> x = normpdf(bins, mu, sigma)
>>> plot(bins, x, color='red', lw=2)
```

`matplotlib.finance.`**`volume_overlay`**(*ax*, *opens*, *closes*, *volumes*, *colorup='k'*, *colordown='r'*, *width=4*, *alpha=1.0*)

Add a volume overlay to the current axes. The opens and closes are used to determine the color of the bar. -1 is missing. If a value is missing on one it must be missing on all

Parameters ax : `Axes`
: an Axes instance to plot to

opens : sequence
: a sequence of opens

closes : sequence
: a sequence of closes

volumes : sequence
: a sequence of volumes

width : int
: the bar width in points

colorup : color
: the color of the lines where close >= open

colordown : color
: the color of the lines where close < open

alpha : float
: bar transparency

Returns ret : `barCollection`
: The `barrCollection` added to the axes

`matplotlib.finance.`**`volume_overlay2`**(*ax*, *closes*, *volumes*, *colorup='k'*, *colordown='r'*, *width=4*, *alpha=1.0*)

Add a volume overlay to the current axes. The closes are used to determine the color of the bar. -1 is missing. If a value is missing on one it must be missing on all

nb: first point is not displayed - it is used only for choosing the right color

Parameters ax : `Axes`
: an Axes instance to plot to

closes : sequence
: a sequence of closes

volumes : sequence
: a sequence of volumes

width : int
: the bar width in points

colorup : color

the color of the lines where close >= open

colordown : color

the color of the lines where close < open

alpha : float

bar transparency

Returns ret : `barCollection`

The `barrCollection` added to the axes

`matplotlib.finance.`**`volume_overlay3`**(*ax*, *quotes*, *colorup='k'*, *colordown='r'*, *width=4*, *alpha=1.0*)

Add a volume overlay to the current axes. quotes is a list of (d, open, high, low, close, volume) and close-open is used to determine the color of the bar

Parameters ax : `Axes`

an Axes instance to plot to

quotes : sequence of (time, open, high, low, close, ...) sequences

data to plot. time must be in float date format - see date2num

width : int

the bar width in points

colorup : color

the color of the lines where close1 >= close0

colordown : color

the color of the lines where close1 < close0

alpha : float

bar transparency

Returns ret : `barCollection`

The `barrCollection` added to the axes

CHAPTER

FIFTYFIVE

FONT_MANAGER

55.1 matplotlib.font_manager

A module for finding, managing, and using fonts across platforms.

This module provides a single *FontManager* instance that can be shared across backends and platforms. The *findfont()* function returns the best TrueType (TTF) font file in the local or system font path that matches the specified *FontProperties* instance. The *FontManager* also handles Adobe Font Metrics (AFM) font files for use by the PostScript backend.

The design is based on the W3C Cascading Style Sheet, Level 1 (CSS1) font specification. Future versions may implement the Level 2 or 2.1 specifications.

Experimental support is included for using `fontconfig` on Unix variant platforms (Linux, OS X, Solaris). To enable it, set the constant `USE_FONTCONFIG` in this file to `True`. Fontconfig has the advantage that it is the standard way to look up fonts on X11 platforms, so if a font is installed, it is much more likely to be found.

class `matplotlib.font_manager.`**`FontEntry`**(*fname=''*, *name=''*, *style='normal'*, *variant='normal'*, *weight='normal'*, *stretch='normal'*, *size='medium'*)

Bases: `object`

A class for storing Font properties. It is used when populating the font lookup dictionary.

class `matplotlib.font_manager.`**`FontManager`**(*size=None*, *weight='normal'*)

Bases: `object`

On import, the *FontManager* singleton instance creates a list of TrueType fonts based on the font properties: name, style, variant, weight, stretch, and size. The *findfont()* method does a nearest neighbor search to find the font that most closely matches the specification. If no good enough match is found, a default font is returned.

`findfont`(*prop*, *fontext='ttf'*, *directory=None*, *fallback_to_default=True*, *rebuild_if_missing=True*)

Search the font list for the font that most closely matches the *FontProperties* *prop*.

findfont() performs a nearest neighbor search. Each font is given a similarity score to the target font properties. The first font with the highest score is returned. If no matches below a certain threshold are found, the default font (usually Vera Sans) is returned.

`directory`, is specified, will only return fonts from the given directory (or subdirectory of that directory).

The result is cached, so subsequent lookups don't have to perform the O(n) nearest neighbor search.

If `fallback_to_default` is True, will fallback to the default font family (usually "Bitstream Vera Sans" or "Helvetica") if the first lookup hard-fails.

See the W3C Cascading Style Sheet, Level 1 documentation for a description of the font finding algorithm.

static **get_default_size**()
Return the default font size.

get_default_weight()
Return the default font weight.

score_family(*families*, *family2*)
Returns a match score between the list of font families in *families* and the font family name *family2*.

An exact match at the head of the list returns 0.0.

A match further down the list will return between 0 and 1.

No match will return 1.0.

score_size(*size1*, *size2*)
Returns a match score between *size1* and *size2*.

If *size2* (the size specified in the font file) is 'scalable', this function always returns 0.0, since any font size can be generated.

Otherwise, the result is the absolute distance between *size1* and *size2*, normalized so that the usual range of font sizes (6pt - 72pt) will lie between 0.0 and 1.0.

score_stretch(*stretch1*, *stretch2*)
Returns a match score between *stretch1* and *stretch2*.

The result is the absolute value of the difference between the CSS numeric values of *stretch1* and *stretch2*, normalized between 0.0 and 1.0.

score_style(*style1*, *style2*)
Returns a match score between *style1* and *style2*.

An exact match returns 0.0.

A match between 'italic' and 'oblique' returns 0.1.

No match returns 1.0.

score_variant(*variant1*, *variant2*)
Returns a match score between *variant1* and *variant2*.

An exact match returns 0.0, otherwise 1.0.

score_weight(*weight1*, *weight2*)
Returns a match score between *weight1* and *weight2*.

The result is the absolute value of the difference between the CSS numeric values of *weight1* and *weight2*, normalized between 0.0 and 1.0.

set_default_weight(*weight*)
Set the default font weight. The initial value is 'normal'.

update_fonts(*filenames*)
Update the font dictionary with new font files. Currently not implemented.

class matplotlib.font_manager.**FontProperties**(*family=None*, *style=None*, *variant=None*, *weight=None*, *stretch=None*, *size=None*, *fname=None*, *_init=None*)

Bases: `object`

A class for storing and manipulating font properties.

The font properties are those described in the W3C Cascading Style Sheet, Level 1 font specification. The six properties are:

- family: A list of font names in decreasing order of priority. The items may include a generic font family name, either 'serif', 'sans-serif', 'cursive', 'fantasy', or 'monospace'. In that case, the actual font to be used will be looked up from the associated rcParam in `matplotlibrc`.
- style: Either 'normal', 'italic' or 'oblique'.
- variant: Either 'normal' or 'small-caps'.
- stretch: A numeric value in the range 0-1000 or one of 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded'
- weight: A numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'
- size: Either an relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 12

The default font property for TrueType fonts (as specified in the default `matplotlibrc` file) is:

```
sans-serif, normal, normal, normal, normal, scalable.
```

Alternatively, a font may be specified using an absolute path to a .ttf file, by using the *fname* kwarg.

The preferred usage of font sizes is to use the relative values, e.g., 'large', instead of absolute font sizes, e.g., 12. This approach allows all text sizes to be made larger or smaller based on the font manager's default font size.

This class will also accept a fontconfig pattern, if it is the only argument provided. See the documentation on fontconfig patterns. This support does not require fontconfig to be installed. We are merely borrowing its pattern syntax for use here.

Note that matplotlib's internal font manager and fontconfig use a different algorithm to lookup fonts, so the results of the same pattern may be different in matplotlib than in other applications that use fontconfig.

`copy()`
Return a deep copy of self

`get_family()`
Return a list of font names that comprise the font family.

`get_file()`
Return the filename of the associated font.

`get_fontconfig_pattern()`
Get a fontconfig pattern suitable for looking up the font as specified with fontconfig's `fc-match` utility.

See the documentation on fontconfig patterns.

This support does not require fontconfig to be installed or support for it to be enabled. We are merely borrowing its pattern syntax for use here.

`get_name()`
Return the name of the font that best matches the font properties.

`get_size()`
Return the font size.

`get_size_in_points()`

`get_slant()`
Return the font style. Values are: 'normal', 'italic' or 'oblique'.

`get_stretch()`
Return the font stretch or width. Options are: 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'.

`get_style()`
Return the font style. Values are: 'normal', 'italic' or 'oblique'.

`get_variant()`
Return the font variant. Values are: 'normal' or 'small-caps'.

`get_weight()`
Set the font weight. Options are: A numeric value in the range 0-1000 or one of 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'

`set_family`(*family*)
Change the font family. May be either an alias (generic name is CSS parlance), such as: 'serif', 'sans-serif', 'cursive', 'fantasy', or 'monospace', a real font name or a list of real font names. Real font names are not supported when `text.usetex` is `True`.

`set_file`(*file*)
Set the filename of the fontfile to use. In this case, all other properties will be ignored.

`set_fontconfig_pattern`(*pattern*)
Set the properties by parsing a fontconfig *pattern*.

See the documentation on fontconfig patterns.

This support does not require fontconfig to be installed or support for it to be enabled. We are merely borrowing its pattern syntax for use here.

set_name(*family*)
Change the font family. May be either an alias (generic name is CSS parlance), such as: 'serif', 'sans-serif', 'cursive', 'fantasy', or 'monospace', a real font name or a list of real font names. Real font names are not supported when `text.usetex` is `True`.

set_size(*size*)
Set the font size. Either an relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 12.

set_slant(*style*)
Set the font style. Values are: 'normal', 'italic' or 'oblique'.

set_stretch(*stretch*)
Set the font stretch or width. Options are: 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded', or a numeric value in the range 0-1000.

set_style(*style*)
Set the font style. Values are: 'normal', 'italic' or 'oblique'.

set_variant(*variant*)
Set the font variant. Values are: 'normal' or 'small-caps'.

set_weight(*weight*)
Set the font weight. May be either a numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'

matplotlib.font_manager.**OSXInstalledFonts**(*directories=None*, *fontext='ttf'*)
Get list of font files on OS X - ignores font suffix by default.

class matplotlib.font_manager.**TempCache**
Bases: `object`

A class to store temporary caches that are (a) not saved to disk and (b) invalidated whenever certain font-related rcParams—namely the family lookup lists—are changed or the font cache is reloaded. This avoids the expensive linear search through all fonts every time a font is looked up.

get(*prop*)

invalidating_rcparams = ('font.serif', 'font.sans-serif', 'font.cursive', 'font.fantasy', 'font.monospace')

make_rcparams_key()

set(*prop*, *value*)

`matplotlib.font_manager.afmFontProperty(fontpath, font)`
A function for populating a `FontKey` instance by extracting information from the AFM font file.

font is a class:`AFM` instance.

`matplotlib.font_manager.createFontList(fontfiles, fontext='ttf')`
A function to create a font lookup list. The default is to create a list of TrueType fonts. An AFM font list can optionally be created.

`matplotlib.font_manager.findSystemFonts(fontpaths=None, fontext='ttf')`
Search for fonts in the specified font paths. If no paths are given, will use a standard set of system paths, as well as the list of fonts tracked by fontconfig if fontconfig is installed and available. A list of TrueType fonts are returned by default with AFM fonts as an option.

`matplotlib.font_manager.findfont(prop, **kw)`

`matplotlib.font_manager.get_fontconfig_fonts(fontext='ttf')`
Grab a list of all the fonts that are being tracked by fontconfig by making a system call to `fc-list`. This is an easy way to grab all of the fonts the user wants to be made available to applications, without needing knowing where all of them reside.

`matplotlib.font_manager.get_fontext_synonyms(fontext)`
Return a list of file extensions extensions that are synonyms for the given file extension *fileext*.

`matplotlib.font_manager.is_opentype_cff_font(filename)`
Returns True if the given font is a Postscript Compact Font Format Font embedded in an OpenType wrapper. Used by the PostScript and PDF backends that can not subset these fonts.

`matplotlib.font_manager.list_fonts(directory, extensions)`
Return a list of all fonts matching any of the extensions, possibly upper-cased, found recursively under the directory.

`matplotlib.font_manager.pickle_dump(data, filename)`
Equivalent to pickle.dump(data, open(filename, 'w')) but closes the file to prevent filehandle leakage.

`matplotlib.font_manager.pickle_load(filename)`
Equivalent to pickle.load(open(filename, 'r')) but closes the file to prevent filehandle leakage.

`matplotlib.font_manager.ttfFontProperty(font)`
A function for populating the `FontKey` by extracting information from the TrueType font file.

font is a `FT2Font` instance.

`matplotlib.font_manager.ttfdict_to_fnames(d)`
flatten a ttfdict to all the filenames it contains

`matplotlib.font_manager.weight_as_number(weight)`
Return the weight property as a numeric value. String values are converted to their corresponding numeric value.

`matplotlib.font_manager.win32FontDirectory()`
Return the user-specified font directory for Win32. This is looked up from the registry key:

```
\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell␣
↪Folders\Fonts
```

If the key is not found, $WINDIR/Fonts will be returned.

`matplotlib.font_manager.`**`win32InstalledFonts`**(*directory=None*, *fontext='ttf'*)

Search for fonts in the specified font directory, or use the system directories if none given. A list of TrueType font filenames are returned by default, or AFM fonts if *fontext* == 'afm'.

55.2 matplotlib.fontconfig_pattern

A module for parsing and generating fontconfig patterns.

See the fontconfig pattern specification for more information.

class `matplotlib.fontconfig_pattern.`**`FontconfigPatternParser`**

Bases: `object`

A simple pyparsing-based parser for fontconfig-style patterns.

See the fontconfig pattern specification for more information.

`parse`(*pattern*)

Parse the given fontconfig *pattern* and return a dictionary of key/value pairs useful for initializing a `font_manager.FontProperties` object.

`matplotlib.fontconfig_pattern.`**`family_escape`**()

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

`matplotlib.fontconfig_pattern.`**`family_unescape`**()

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

`matplotlib.fontconfig_pattern.`**`generate_fontconfig_pattern`**(*d*)

Given a dictionary of key/value pairs, generates a fontconfig pattern string.

`matplotlib.fontconfig_pattern.`**`value_escape`**()

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

`matplotlib.fontconfig_pattern.`**`value_unescape`**()

Return the string obtained by replacing the leftmost non-overlapping occurrences of pattern in string by the replacement repl.

CHAPTER

FIFTYSIX

GRIDSPEC

56.1 matplotlib.gridspec

gridspec is a module which specifies the location of the subplot in the figure.

GridSpec specifies the geometry of the grid that a subplot will be placed. The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

SubplotSpec specifies the location of the subplot in the given *GridSpec*.

class matplotlib.gridspec.**GridSpec**(*nrows*, *ncols*, *left=None*, *bottom=None*, *right=None*, *top=None*, *wspace=None*, *hspace=None*, *width_ratios=None*, *height_ratios=None*)

Bases: *matplotlib.gridspec.GridSpecBase*

A class that specifies the geometry of the grid that a subplot will be placed. The location of grid is determined by similar way as the SubplotParams.

The number of rows and number of columns of the grid need to be set. Optionally, the subplot layout parameters (e.g., left, right, etc.) can be tuned.

get_subplot_params(*fig=None*)

return a dictionary of subplot layout parameters. The default parameters are from rcParams unless a figure attribute is set.

locally_modified_subplot_params()

tight_layout(*fig*, *renderer=None*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Adjust subplot parameters to give specified padding.

Parameters:

pad [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.

h_pad, w_pad [float] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.

rect [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

update(***kwargs*)
Update the current values. If any kwarg is None, default to the current value, if set, otherwise to rc.

class matplotlib.gridspec.**GridSpecBase**(*nrows*, *ncols*, *height_ratios=None*, *width_ratios=None*)
Bases: object

A base class of GridSpec that specifies the geometry of the grid that a subplot will be placed.

The number of rows and number of columns of the grid need to be set. Optionally, the ratio of heights and widths of rows and columns can be specified.

get_geometry()
get the geometry of the grid, e.g., 2,3

get_grid_positions(*fig*)
return lists of bottom and top position of rows, left and right positions of columns.

get_height_ratios()

get_subplot_params(*fig=None*)

get_width_ratios()

new_subplotspec(*loc*, *rowspan=1*, *colspan=1*)
create and return a SuplotSpec instance.

set_height_ratios(*height_ratios*)

set_width_ratios(*width_ratios*)

class matplotlib.gridspec.**GridSpecFromSubplotSpec**(*nrows*, *ncols*, *subplot_spec*, *wspace=None*, *hspace=None*, *height_ratios=None*, *width_ratios=None*)
Bases: matplotlib.gridspec.GridSpecBase

GridSpec whose subplot layout parameters are inherited from the location specified by a given SubplotSpec.

The number of rows and number of columns of the grid need to be set. An instance of SubplotSpec is also needed to be set from which the layout parameters will be inherited. The wspace and hspace of the layout can be optionally specified or the default values (from the figure or rcParams) will be used.

get_subplot_params(*fig=None*)
return a dictionary of subplot layout parameters.

get_topmost_subplotspec()
get the topmost SubplotSpec instance associated with the subplot

class `matplotlib.gridspec.SubplotSpec`(*gridspec*, *num1*, *num2=None*)
Bases: `object`

specifies the location of the subplot in the given *GridSpec*.

The subplot will occupy the num1-th cell of the given gridspec. If num2 is provided, the subplot will span between num1-th cell and num2-th cell.

The index stars from 0.

`get_geometry()`
get the subplot geometry, e.g., 2,2,3. Unlike SuplorParams, index is 0-based

`get_gridspec()`

`get_position`(*fig*, *return_all=False*)
update the subplot position from fig.subplotpars

`get_topmost_subplotspec()`
get the topmost SubplotSpec instance associated with the subplot

CHAPTER

FIFTYSEVEN

IMAGE

57.1 matplotlib.image

The image module supports basic image loading, rescaling and display operations.

class `matplotlib.image.`**`AxesImage`**(*ax*, *cmap=None*, *norm=None*, *interpolation=None*, *origin=None*, *extent=None*, *filternorm=1*, *filterrad=4.0*, *resample=False*, ***kwargs*)

Bases: `matplotlib.image._AxesImageBase`

interpolation and cmap default to their rc settings

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

extent is data axes (left, right, bottom, top) for making image plots registered with data plots. Default is to label the pixel centers with the zero-based row and column indices.

Additional kwargs are matplotlib.artist properties

`get_cursor_data`(*event*)

Get the cursor data for a given event

`get_extent`()

Get the image extent: left, right, bottom, top

`get_window_extent`(*renderer=None*)

`make_image`(*magnification=1.0*)

`set_extent`(*extent*)

extent is data axes (left, right, bottom, top) for making image plots

This updates ax.dataLim, and, if autoscaling, sets viewLim to tightly fit the image, regardless of dataLim. Autoscaling state is not changed, so following this with ax.autoscale_view will redo the autoscaling in accord with dataLim.

class `matplotlib.image.`**`BboxImage`**(*bbox*, *cmap=None*, *norm=None*, *interpolation=None*, *origin=None*, *filternorm=1*, *filterrad=4.0*, *resample=False*, *interp_at_native=True*, ***kwargs*)

Bases: `matplotlib.image._AxesImageBase`

The Image class whose size is determined by the given bbox.

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

interp_at_native is a flag that determines whether or not interpolation should still be applied when the image is displayed at its native resolution. A common use case for this is when displaying an image for annotational purposes; it is treated similarly to Photoshop (interpolation is only used when displaying the image at non-native resolutions).

kwargs are an optional list of Artist keyword args

contains(*mouseevent*)
Test whether the mouse event occured within the image.

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_size()
Get the numrows, numcols of the input image

get_window_extent(*renderer=None*)

make_image(*renderer*, *magnification=1.0*)

class matplotlib.image.**FigureImage**(*fig*, *cmap=None*, *norm=None*, *offsetx=0*, *offsety=0*, *origin=None*, ***kwargs*)
Bases: matplotlib.artist.Artist, matplotlib.cm.ScalarMappable

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

kwargs are an optional list of Artist keyword args

contains(*mouseevent*)
Test whether the mouse event occured within the image.

draw(*artist*, *renderer*, **args*, ***kwargs*)

get_extent()
Get the image extent: left, right, bottom, top

get_size()
Get the numrows, numcols of the input image

make_image(*magnification=1.0*)

set_array(*A*)
Deprecated; use set_data for consistency with other image types.

set_data(*A*)
Set the image array.

write_png(*fname*)
Write the image to png file with fname

zorder = 0

class matplotlib.image.**NonUniformImage**(*ax*, ***kwargs*)

Bases: `matplotlib.image.AxesImage`

kwargs are identical to those for AxesImage, except that 'interpolation' defaults to 'nearest', and 'bilinear' is the only alternative.

get_extent()

make_image(*magnification=1.0*)

set_array(**args*)

set_cmap(*cmap*)

set_data(*x*, *y*, *A*)

Set the grid for the pixel centers, and the pixel values.

x and y are 1-D ndarrays of lengths N and M, respectively, specifying pixel centers

A is an (M,N) ndarray or masked array of values to be colormapped, or a (M,N,3) RGB array, or a (M,N,4) RGBA array.

set_filternorm(*s*)

set_filterrad(*s*)

set_interpolation(*s*)

set_norm(*norm*)

class matplotlib.image.**PcolorImage**(*ax*, *x=None*, *y=None*, *A=None*, *cmap=None*, *norm=None*, ***kwargs*)

Bases: `matplotlib.artist.Artist`, `matplotlib.cm.ScalarMappable`

Make a pcolor-style plot with an irregular rectangular grid.

This uses a variation of the original irregular image code, and it is used by pcolorfast for the corresponding grid type.

cmap defaults to its rc setting

cmap is a colors.Colormap instance norm is a colors.Normalize instance to map luminance to 0-1

Additional kwargs are matplotlib.artist properties

changed()

draw(*artist*, *renderer*, **args*, ***kwargs*)

make_image(*magnification=1.0*)

set_alpha(*alpha*)
Set the alpha value used for blending - not supported on all backends

ACCEPTS: float

set_array(**args*)

set_data(*x*, *y*, *A*)

matplotlib.image.**imread**(*fname*, *format=None*)
Read an image from a file into an array.

fname may be a string path, a valid URL, or a Python file-like object. If using a file object, it must be opened in binary mode.

If *format* is provided, will try to read file of that type, otherwise the format is deduced from the filename. If nothing can be deduced, PNG is tried.

Return value is a `numpy.array`. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4.

matplotlib can only read PNGs natively, but if PIL is installed, it will use it to load the image and return an array (if possible) which can be used with `imshow()`. Note, URL strings may not be compatible with PIL. Check the PIL documentation for more information.

matplotlib.image.**imsave**(*fname*, *arr*, *vmin=None*, *vmax=None*, *cmap=None*, *format=None*, *origin=None*, *dpi=100*)
Save an array as in image file.

The output formats available depend on the backend being used.

Arguments:

fname: A string containing a path to a filename, or a Python file-like object. If *format* is *None* and *fname* is a string, the output format is deduced from the extension of the filename.

arr: An MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA) array.

Keyword arguments:

vmin/vmax: **[None | scalar]** *vmin* and *vmax* set the color scaling for the image by fixing the values that map to the colormap color limits. If either *vmin* or *vmax* is None, that limit is determined from the *arr* min/max value.

cmap: cmap is a colors.Colormap instance, e.g., cm.jet. If None, default to the rc image.cmap value.

format: One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

origin ['upper' | 'lower'] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc image.origin value.

dpi The DPI to store in the metadata of the file. This does not affect the resolution of the output image.

`matplotlib.image.pil_to_array(pilImage)`

Load a PIL image and return it as a numpy array. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4

`matplotlib.image.thumbnail(infile, thumbfile, scale=0.1, interpolation='bilinear', preview=False)`

make a thumbnail of image in *infile* with output filename *thumbfile*.

infile **the image file – must be PNG or Pillow-readable if you** have Pillow installed

thumbfile the thumbnail filename

scale the scale factor for the thumbnail

interpolation the interpolation scheme used in the resampling

preview if True, the default backend (presumably a user interface backend) will be used which will cause a figure to be raised if `show()` is called. If it is False, a pure image backend will be used depending on the extension, 'png'->FigureCanvasAgg, 'pdf'->FigureCanvasPdf, 'svg'->FigureCanvasSVG

See examples/misc/image_thumbnail.py.

Return value is the figure instance containing the thumbnail

CHAPTER

FIFTYEIGHT

LEGEND AND LEGEND_HANDLER

58.1 matplotlib.legend

The legend module defines the Legend class, which is responsible for drawing legends associated with axes and/or figures.

Important: It is unlikely that you would ever create a Legend instance manually. Most users would normally create a legend via the `legend()` function. For more details on legends there is also a *legend guide*.

The Legend class can be considered as a container of legend handles and legend texts. Creation of corresponding legend handles from the plot elements in the axes or figures (e.g., lines, patches, etc.) are specified by the handler map, which defines the mapping between the plot elements and the legend handlers to be used (the default legend handlers are defined in the `legend_handler` module). Note that not all kinds of artist are supported by the legend yet by default but it is possible to extend the legend handler's capabilities to support arbitrary objects. See the *legend guide* for more information.

class `matplotlib.legend.`**`DraggableLegend`**(*legend*, *use_blit=False*, *update='loc'*)

Bases: `matplotlib.offsetbox.DraggableOffsetBox`

update [If "loc", update *loc* parameter of] legend upon finalizing. If "bbox", update *bbox_to_anchor* parameter.

`artist_picker`(*legend*, *evt*)

`finalize_offset`()

class `matplotlib.legend.`**`Legend`**(*parent*, *handles*, *labels*, *loc=None*, *numpoints=None*, *markerscale=None*, *markerfirst=True*, *scatterpoints=None*, *scatteryoffsets=None*, *prop=None*, *fontsize=None*, *borderpad=None*, *labelspacing=None*, *handlelength=None*, *handleheight=None*, *handletextpad=None*, *borderaxespad=None*, *columnspacing=None*, *ncol=1*, *mode=None*, *fancybox=None*, *shadow=None*, *title=None*, *framealpha=None*, *bbox_to_anchor=None*, *bbox_transform=None*, *frameon=None*, *handler_map=None*)

Bases: `matplotlib.artist.Artist`

Place a legend on the axes at location loc. Labels are a sequence of strings and loc can be a string or an integer specifying the legend location

The location codes are:

```
'best'         : 0, (only implemented for axes legends)
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10,
```

loc can be a tuple of the normalized coordinate values with respect its parent.

- *parent*: the artist that contains the legend
- ***handles*: a list of artists (lines, patches) to be added to the** legend
- *labels*: a list of strings to label the legend

Optional keyword arguments:

Keyword	Description
loc	a location code
prop	the font property
fontsize	the font size (used only if prop is not specified)
markerscale	the relative size of legend markers vs. original
markerfirst	If true, place legend marker to left of label If false, place legend marker to right of label
numpoints	the number of points in the legend for line
scatterpoints	the number of points in the legend for scatter plot
scatteryoffsets	a list of yoffsets for scatter symbols in legend
frameon	if True, draw a frame around the legend. If None, use rc
fancybox	if True, draw a frame with a round fancybox. If None, use rc
shadow	if True, draw a shadow behind legend
framealpha	If not None, alpha channel for the frame.
ncol	number of columns
borderpad	the fractional whitespace inside the legend border
labelspacing	the vertical space between the legend entries
handlelength	the length of the legend handles
handleheight	the height of the legend handles
handletextpad	the pad between the legend handle and text
borderaxespad	the pad between the axes and legend border
columnspacing	the spacing between columns
title	the legend title
bbox_to_anchor	the bbox that the legend will be anchored.
bbox_transform	the transform for the bbox. transAxes if None.

The pad and spacing parameters are measured in font-size units. e.g., a fontsize of 10 points and a handlelength=5 implies a handlelength of 50 points. Values from rcParams will be used if None.

Users can specify any arbitrary location for the legend using the *bbox_to_anchor* keyword argument. bbox_to_anchor can be an instance of BboxBase(or its derivatives) or a tuple of 2 or 4 floats. See `set_bbox_to_anchor()` for more detail.

The legend location can be specified by setting *loc* with a tuple of 2 floats, which is interpreted as the lower-left corner of the legend in the normalized axes coordinate.

codes = {'upper right': 1, 'upper left': 2, 'best': 0, 'center right': 7, 'center left': 6, 'right': 5, 'upper center':

contains(*event*)

draggable(*state=None*, *use_blit=False*, *update='loc'*)
Set the draggable state – if state is

- None : toggle the current state
- True : turn draggable on
- False : turn draggable off

If draggable is on, you can drag the legend on the canvas with the mouse. The DraggableLegend helper instance is returned if draggable is on.

The update parameter control which parameter of the legend changes when dragged. If update is "loc", the *loc* paramter of the legend is changed. If "bbox", the *bbox_to_anchor* parameter is changed.

draw(*artist*, *renderer*, **args*, ***kwargs*)
Draw everything that belongs to the legend

draw_frame(*b*)
b is a boolean. Set draw frame to b

get_bbox_to_anchor()
return the bbox that the legend will be anchored

get_children()
return a list of child artists

classmethod get_default_handler_map()
A class method that returns the default handler map.

get_frame()
return the Rectangle instance used to frame the legend

get_frame_on()
Get whether the legend box patch is drawn

static get_legend_handler(*legend_handler_map*, *orig_handle*)
return a legend handler from *legend_handler_map* that corresponds to *orig_handler*.

legend_handler_map should be a dictionary object (that is returned by the get_legend_handler_map method).

It first checks if the *orig_handle* itself is a key in the *legend_hanler_map* and return the associated value. Otherwise, it checks for each of the classes in its method-resolution-order. If no matching key is found, it returns None.

get_legend_handler_map()
return the handler map.

get_lines()
return a list of lines.Line2D instances in the legend

get_patches()
return a list of patch instances in the legend

get_texts()
return a list of text.Text instance in the legend

get_title()
return Text instance for the legend title

get_window_extent(**args*, ***kwargs*)
return a extent of the legend

set_bbox_to_anchor(*bbox*, *transform=None*)
set the bbox that the legend will be anchored.

bbox can be a BboxBase instance, a tuple of [left, bottom, width, height] in the given transform (normalized axes coordinate if None), or a tuple of [left, bottom] where the width and height will be assumed to be zero.

classmethod set_default_handler_map(*handler_map*)
A class method to set the default handler map.

set_frame_on(*b*)
Set whether the legend box patch is drawn

ACCEPTS: [*True* | *False*]

set_title(*title*, *prop=None*)
set the legend title. Fontproperties can be optionally set with *prop* parameter.

classmethod update_default_handler_map(*handler_map*)
A class method to update the default handler map.

zorder = 5

58.2 matplotlib.legend_handler

This module defines default legend handlers.

It is strongly encouraged to have read the *legend guide* before this documentation.

Legend handlers are expected to be a callable object with a following signature.

```
legend_handler(legend, orig_handle, fontsize, handlebox)
```

Where *legend* is the legend itself, *orig_handle* is the original plot, *fontsize* is the fontsize in pixles, and *handlebox* is a OffsetBox instance. Within the call, you should create relevant artists (using relevant properties from the *legend* and/or *orig_handle*) and add them into the handlebox. The artists needs to be scaled according to the fontsize (note that the size is in pixel, i.e., this is dpi-scaled value).

This module includes definition of several legend handler classes derived from the base class (HandlerBase) with the following method.

> def legend_artist(self, legend, orig_handle, fontsize, handlebox):

class `matplotlib.legend_handler.`**`HandlerBase`**(*xpad=0.0*, *ypad=0.0*, *update_func=None*)

A Base class for default legend handlers.

The derived classes are meant to override *create_artists* method, which has a following signature.:

```
def create_artists(self, legend, orig_handle,
                   xdescent, ydescent, width, height, fontsize,
                   trans):
```

The overridden method needs to create artists of the given transform that fits in the given dimension (xdescent, ydescent, width, height) that are scaled by fontsize if necessary.

`adjust_drawing_area`(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

`create_artists`(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

`legend_artist`(*legend*, *orig_handle*, *fontsize*, *handlebox*)

Return the artist that this HandlerBase generates for the given original artist/handle.

Parameters legend : `matplotlib.legend.Legend` instance
: The legend for which these legend artists are being created.

orig_handle : `matplotlib.artist.Artist` or similar
: The object for which these legend artists are being created.

fontsize : float or int
: The fontsize in pixels. The artists being created should be scaled according to the given fontsize.

handlebox : `matplotlib.offsetbox.OffsetBox` instance
: The box which has been created to hold this legend entry's artists. Artists created in the `legend_artist` method must be added to this handlebox inside this method.

`update_prop`(*legend_handle*, *orig_handle*, *legend*)

class `matplotlib.legend_handler.`**`HandlerCircleCollection`**(*yoffsets=None*, *sizes=None*, ***kw*)

Handler for CircleCollections

`create_collection`(*orig_handle*, *sizes*, *offsets*, *transOffset*)

class `matplotlib.legend_handler.`**`HandlerErrorbar`**(*xerr_size=0.5*, *yerr_size=None*, *marker_pad=0.3*, *numpoints=None*, ***kw*)

Handler for Errorbars

`create_artists`(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

`get_err_size`(*legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

class `matplotlib.legend_handler.`**`HandlerLine2D`**(*marker_pad=0.3*, *numpoints=None*, ***kw*)

Handler for Line2D instances.

`create_artists`(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

class `matplotlib.legend_handler.`**`HandlerLineCollection`**(*marker_pad=0.3*, *numpoints=None*, ***kw*)

Handler for LineCollection instances.

`create_artists`(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

`get_numpoints`(*legend*)

class `matplotlib.legend_handler.`**`HandlerNpoints`**(*marker_pad=0.3*, *numpoints=None*, ***kw*)

`get_numpoints`(*legend*)

`get_xdata`(*legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

class `matplotlib.legend_handler.`**`HandlerNpointsYoffsets`**(*numpoints=None*, *yoffsets=None*, ***kw*)

`get_ydata`(*legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

class `matplotlib.legend_handler.`**`HandlerPatch`**(*patch_func=None*, ***kw*)

Handler for Patch instances.

The HandlerPatch class optionally takes a function `patch_func` who's responsibility is to create the legend key artist. The `patch_func` should have the signature:

```
def patch_func(legend=legend, orig_handle=orig_handle,
               xdescent=xdescent, ydescent=ydescent,
               width=width, height=height, fontsize=fontsize)
```

Subsequently the created artist will have its `update_prop` method called and the appropriate transform will be applied.

create_artists(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

class matplotlib.legend_handler.**HandlerPathCollection**(*yoffsets=None*, *sizes=None*, ***kw*)
Handler for PathCollections, which are used by scatter

create_collection(*orig_handle*, *sizes*, *offsets*, *transOffset*)

class matplotlib.legend_handler.**HandlerPolyCollection**(*xpad=0.0*, *ypad=0.0*, *update_func=None*)
Handler for PolyCollection used in fill_between and stackplot.

create_artists(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

class matplotlib.legend_handler.**HandlerRegularPolyCollection**(*yoffsets=None*, *sizes=None*, ***kw*)
Handler for RegularPolyCollections.

create_artists(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

create_collection(*orig_handle*, *sizes*, *offsets*, *transOffset*)

get_numpoints(*legend*)

get_sizes(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

update_prop(*legend_handle*, *orig_handle*, *legend*)

class matplotlib.legend_handler.**HandlerStem**(*marker_pad=0.3*, *numpoints=None*, *bottom=None*, *yoffsets=None*, ***kw*)
Handler for Errorbars

create_artists(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

get_ydata(*legend*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*)

class matplotlib.legend_handler.**HandlerTuple**(***kwargs*)
Handler for Tuple

create_artists(*legend*, *orig_handle*, *xdescent*, *ydescent*, *width*, *height*, *fontsize*, *trans*)

matplotlib.legend_handler.**update_from_first_child**(*tgt*, *src*)

CHAPTER

FIFTYNINE

LINES

59.1 matplotlib.lines

This module contains all the 2D line class which can draw with a variety of line styles, markers and colors.

class `matplotlib.lines.Line2D`(*xdata*, *ydata*, *linewidth=None*, *linestyle=None*, *color=None*, *marker=None*, *markersize=None*, *markeredgewidth=None*, *markeredgecolor=None*, *markerfacecolor=None*, *markerfacecoloralt='none'*, *fillstyle=None*, *antialiased=None*, *dash_capstyle=None*, *solid_capstyle=None*, *dash_joinstyle=None*, *solid_joinstyle=None*, *pickradius=5*, *drawstyle=None*, *markevery=None*, ***kwargs*)

Bases: `matplotlib.artist.Artist`

A line - the line can have both a solid linestyle connecting all the vertices, and a marker at each vertex. Additionally, the drawing of the solid line is influenced by the drawstyle, e.g., one can create "stepped" lines in various styles.

Create a `Line2D` instance with *x* and *y* data in sequences *xdata*, *ydata*.

The kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
C	

Table 59.1 – continued from previous page

Property	Description
figure	a *matplotlib.figure.Figure* instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
linewidth or lw	float value in points
marker	*A valid marker style*
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a *matplotlib.transforms.Transform* instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See *set_linestyle()* for a decription of the line styles, *set_marker()* for a description of the markers, and *set_drawstyle()* for a description of the draw styles.

axes
The *Axes* instance the artist resides in, or *None*.

contains(*mouseevent*)
Test whether the mouse event occurred on the line. The pick radius determines the precision of the location test (usually within five points of the value). Use *get_pickradius()* or *set_pickradius()* to view or modify it.

Returns *True* if any values are within the radius along with `{'ind': pointlist}`, where *pointlist* is the set of points within the radius.

TODO: sort returned indices by distance

draw(*artist*, *renderer*, **args*, ***kwargs*)
draw the Line with `renderer` unless visibility is False

drawStyleKeys = ['steps-pre', 'default', 'steps-mid', 'steps-post', 'steps']

drawStyles = {'steps-pre': '_draw_steps_pre', 'steps': '_draw_steps_pre', 'default': '_draw_lines', 'steps-mic

fillStyles = ('full', 'left', 'right', 'bottom', 'top', 'none')

filled_markers = ('o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd')

get_aa()
alias for get_antialiased

get_antialiased()

get_c()
alias for get_color

get_color()

get_dash_capstyle()
Get the cap style for dashed linestyles

get_dash_joinstyle()
Get the join style for dashed linestyles

get_data(*orig=True*)
Return the xdata, ydata.

If *orig* is *True*, return the original data.

get_drawstyle()

get_fillstyle()
return the marker fillstyle

get_linestyle()

get_linewidth()

get_ls()
alias for get_linestyle

get_lw()
alias for get_linewidth

get_marker()

`get_markeredgecolor()`

`get_markeredgewidth()`

`get_markerfacecolor()`

`get_markerfacecoloralt()`

`get_markersize()`

`get_markevery()`
: return the markevery setting

`get_mec()`
: alias for get_markeredgecolor

`get_mew()`
: alias for get_markeredgewidth

`get_mfc()`
: alias for get_markerfacecolor

`get_mfcalt`(*alt=False*)
: alias for get_markerfacecoloralt

`get_ms()`
: alias for get_markersize

`get_path()`
: Return the `Path` object associated with this line.

`get_pickradius()`
: return the pick radius used for containment tests

`get_solid_capstyle()`
: Get the cap style for solid linestyles

`get_solid_joinstyle()`
: Get the join style for solid linestyles

`get_window_extent`(*renderer*)

`get_xdata`(*orig=True*)
: Return the xdata.

  If *orig* is *True*, return the original data, else the processed data.

`get_xydata()`
: Return the *xy* data as a Nx2 numpy array.

get_ydata(*orig=True*)
Return the ydata.

If *orig* is *True*, return the original data, else the processed data.

is_dashed()
return True if line is dashstyle

lineStyles = {'': '_draw_nothing', ':': '_draw_dotted', '–': '_draw_dashed', '-.': '_draw_dash_dot', '-': '_dr

markers = {0: 'tickleft', 1: 'tickright', 2: 'tickup', '8': 'octagon', 4: 'caretleft', 5: 'caretright', ',': 'pixel', 7: 'c

recache(*always=False*)

recache_always()

set_aa(*val*)
alias for set_antialiased

set_antialiased(*b*)
True if line should be drawin with antialiased rendering

ACCEPTS: [True | False]

set_c(*val*)
alias for set_color

set_color(*color*)
Set the color of the line

ACCEPTS: any matplotlib color

set_dash_capstyle(*s*)
Set the cap style for dashed linestyles

ACCEPTS: ['butt' | 'round' | 'projecting']

set_dash_joinstyle(*s*)
Set the join style for dashed linestyles ACCEPTS: ['miter' | 'round' | 'bevel']

set_dashes(*seq*)
Set the dash sequence, sequence of dashes with on off ink in points. If seq is empty or if seq = (None, None), the linestyle will be set to solid.

ACCEPTS: sequence of on/off ink in points

set_data(**args*)
Set the x and y data

ACCEPTS: 2D array (rows are x, y) or two 1D arrays

set_drawstyle(*drawstyle*)
Set the drawstyle of the plot

'default' connects the points with lines. The steps variants produce step-plots. 'steps' is equivalent to 'steps-pre' and is maintained for backward-compatibility.
ACCEPTS: ['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']

set_fillstyle(*fs*)
Set the marker fill style; 'full' means fill the whole marker. 'none' means no filling; other options are for half-filled markers.

ACCEPTS: ['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']

set_linestyle(*ls*)
Set the linestyle of the line (also accepts drawstyles, e.g., `'steps--'`)

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line
`'None'`	draw nothing
`' '`	draw nothing
`''`	draw nothing

'steps' is equivalent to 'steps-pre' and is maintained for backward-compatibility.

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where `onoffseq` is an even length tuple of on and off ink in points.
ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'` | `' '` | `''`]
See also:

`set_drawstyle()` To set the drawing style (stepping) of the plot.

Parameters ls : { `'-'`, `'--'`, `'-.'`, `':'`} and more see description
The line style.

set_linewidth(*w*)
Set the line width in points

ACCEPTS: float value in points

set_ls(*val*)
alias for set_linestyle

set_lw(*val*)
alias for set_linewidth

set_marker(*marker*)
Set the line marker

ACCEPTS: *`A valid marker style`*
Parameters marker: marker style
See *`markers`* for full description of possible argument

`set_markeredgecolor`(*ec*)

Set the marker edge color

ACCEPTS: any matplotlib color

`set_markeredgewidth`(*ew*)

Set the marker edge width in points

ACCEPTS: float value in points

`set_markerfacecolor`(*fc*)

Set the marker face color.

ACCEPTS: any matplotlib color

`set_markerfacecoloralt`(*fc*)

Set the alternate marker face color.

ACCEPTS: any matplotlib color

`set_markersize`(*sz*)

Set the marker size in points

ACCEPTS: float

`set_markevery`(*every*)

Set the markevery property to subsample the plot when using markers.

e.g., if `every=5`, every 5-th marker will be plotted.

ACCEPTS: [None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]

Parameters every: None | int | length-2 tuple of int | slice | list/array of int |

float | length-2 tuple of float

Which markers to plot.

- every=None, every point will be plotted.
- every=N, every N-th marker will be plotted starting with marker 0.
- every=(start, N), every N-th marker, starting at point start, will be plotted.
- every=slice(start, end, N), every N-th marker, starting at point start, upto but not including point end, will be plotted.
- every=[i, j, m, n], only markers at points i, j, m, and n will be plotted.
- every=0.1, (i.e. a float) then markers will be spaced at approximately equal distances along the line; the distance along the line between markers is determined by multiplying the display-coordinate distance of the axes bounding-box diagonal by the value of every.
- every=(0.5, 0.1) (i.e. a length-2 tuple of float), the same functionality as every=0.1 is exhibited but the first marker

will be 0.5 multiplied by the display-cordinate-diagonal-distance along the line.

Notes

Setting the markevery property will only show markers at actual data points. When using float arguments to set the markevery property on irregularly spaced data, the markers will likely not appear evenly spaced because the actual data points do not coincide with the theoretical spacing between markers.

When using a start offset to specify the first marker, the offset will be from the first data point which may be different from the first the visible data point if the plot is zoomed in.

If zooming in on a plot when using float arguments then the actual data points that have markers will change because the distance between markers is always determined from the display-coordinates axes-bounding-box-diagonal regardless of the actual axes data limits.

set_mec(*val*)
alias for set_markeredgecolor

set_mew(*val*)
alias for set_markeredgewidth

set_mfc(*val*)
alias for set_markerfacecolor

set_mfcalt(*val*)
alias for set_markerfacecoloralt

set_ms(*val*)
alias for set_markersize

set_picker(*p*)
Sets the event picker details for the line.

ACCEPTS: float distance in points or callable pick function `fn(artist, event)`

set_pickradius(*d*)
Sets the pick radius used for containment tests

ACCEPTS: float distance in points

set_solid_capstyle(*s*)
Set the cap style for solid linestyles

ACCEPTS: ['butt' | 'round' | 'projecting']

set_solid_joinstyle(*s*)
Set the join style for solid linestyles ACCEPTS: ['miter' | 'round' | 'bevel']

set_transform(*t*)
set the Transformation instance used by this artist

ACCEPTS: a `matplotlib.transforms.Transform` instance

set_xdata(*x*)
Set the data np.array for x

ACCEPTS: 1D array

set_ydata(*y*)
Set the data np.array for y

ACCEPTS: 1D array

update_from(*other*)
copy properties from other to self

validCap = ('butt', 'round', 'projecting')

validJoin = ('miter', 'round', 'bevel')

zorder = 2

class matplotlib.lines.**VertexSelector**(*line*)
Bases: object

Manage the callbacks to maintain a list of selected vertices for *matplotlib.lines.Line2D*. Derived classes should override *process_selected()* to do something with the picks.

Here is an example which highlights the selected verts with red circles:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines

class HighlightSelected(lines.VertexSelector):
    def __init__(self, line, fmt='ro', **kwargs):
        lines.VertexSelector.__init__(self, line)
        self.markers, = self.axes.plot([], [], fmt, **kwargs)

    def process_selected(self, ind, xs, ys):
        self.markers.set_data(xs, ys)
        self.canvas.draw()

fig = plt.figure()
ax = fig.add_subplot(111)
x, y = np.random.rand(2, 30)
line, = ax.plot(x, y, 'bs-', picker=5)

selector = HighlightSelected(line)
plt.show()
```

Initialize the class with a *matplotlib.lines.Line2D* instance. The line should already be added to some *matplotlib.axes.Axes* instance and should have the picker property set.

onpick(*event*)
When the line is picked, update the set of selected indicies.

process_selected(*ind*, *xs*, *ys*)

Default "do nothing" implementation of the `process_selected()` method.

ind are the indices of the selected vertices. *xs* and *ys* are the coordinates of the selected vertices.

matplotlib.lines.**segment_hits**(*cx*, *cy*, *x*, *y*, *radius*)

Determine if any line segments are within radius of a point. Returns the list of line segments that are within that radius.

CHAPTER

SIXTY

MARKERS

60.1 matplotlib.markers

This module contains functions to handle markers. Used by both the marker functionality of *plot* and *scatter*.

All possible markers are defined here:

marker	description
”.”	point
”,”	pixel
“o”	circle
“v”	triangle_down
“^”	triangle_up
“<”	triangle_left
“>”	triangle_right
“1”	tri_down
“2”	tri_up
“3”	tri_left
“4”	tri_right
“8”	octagon
“s”	square
“p”	pentagon
“*”	star
“h”	hexagon1
“H”	hexagon2
“+”	plus
“x”	x
“D”	diamond
“d”	thin_diamond
“|”	vline
“_”	hline
TICKLEFT	tickleft
TICKRIGHT	tickright
TICKUP	tickup

Con

Table 60.1 – continued from previous page

marker	description
TICKDOWN	tickdown
CARETLEFT	caretleft
CARETRIGHT	caretright
CARETUP	caretup
CARETDOWN	caretdown
“None”	nothing
None	nothing
” “	nothing
“”	nothing
`'$...$'`	render the string using mathtext.
`verts`	a list of (x, y) pairs used for Path vertices. The center of the marker is located at (0,0) and th
path	a *Path* instance.
(`numsides`, `style`, `angle`)	see below

The marker can also be a tuple (`numsides`, `style`, `angle`), which will create a custom, regular symbol.

`numsides`: the number of sides

`style`: the style of the regular symbol:

Value	Description
0	a regular polygon
1	a star-like symbol
2	an asterisk
3	a circle (`numsides` and `angle` is ignored)

`angle`: the angle of rotation of the symbol, in degrees

For backward compatibility, the form (`verts`, 0) is also accepted, but it is equivalent to just `verts` for giving a raw set of vertices that define the shape.

class `matplotlib.markers.`**`MarkerStyle`**(*marker=None*, *fillstyle=None*)

Bases: `object`

Parameters marker : string or array_like, optional, default: None

See the descriptions of possible markers in the module docstring.

fillstyle : string, optional, default: ‘full’

‘full’, ‘left”, ‘right’, ‘bottom’, ‘top’, ‘none’

Attributes

markers	(list of known markes)
fillstyles	(list of known fillstyles)
filled_markers	(list of known filled markers.)

`filled_markers` = (‘o’, ‘v’, ‘^’, ‘<’, ‘>’, ‘8’, ‘s’, ‘p’, ‘*’, ‘h’, ‘H’, ‘D’, ‘d’)

fillstyles = ('full', 'left', 'right', 'bottom', 'top', 'none')

get_alt_path()

get_alt_transform()

get_capstyle()

get_fillstyle()

get_joinstyle()

get_marker()

get_path()

get_snap_threshold()

get_transform()

is_filled()

markers = {0: 'tickleft', 1: 'tickright', 2: 'tickup', '8': 'octagon', 4: 'caretleft', 5: 'caretright', ',': 'pixel', 7: 'c

set_fillstyle(*fillstyle*)
Sets fillstyle

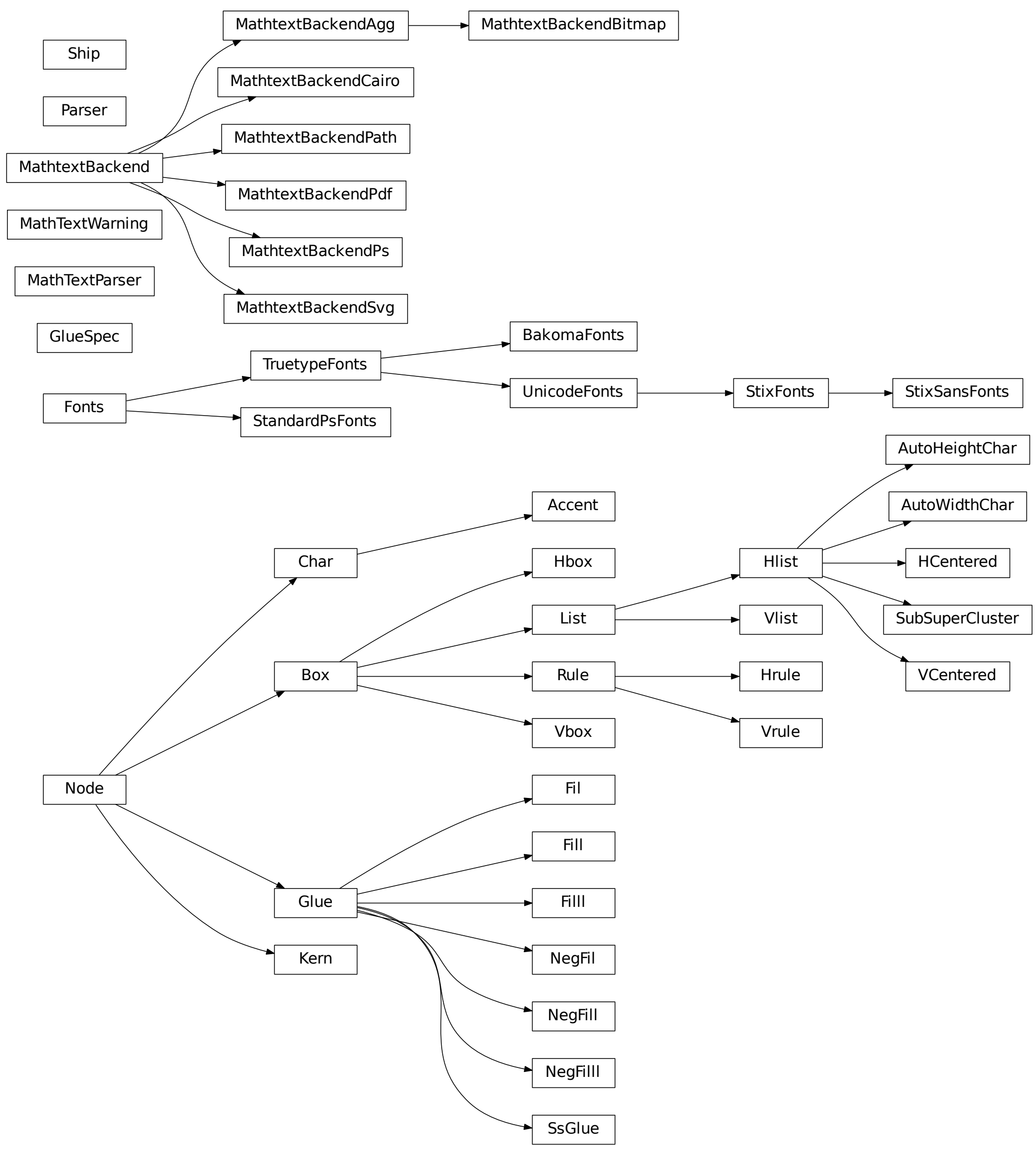
Parameters fillstyle : string amongst known fillstyles

set_marker(*marker*)

CHAPTER

SIXTYONE

MATHTEXT



61.1 matplotlib.mathtext

mathtext is a module for parsing a subset of the TeX math syntax and drawing them to a matplotlib backend.

For a tutorial of its usage see *Writing mathematical expressions*. This document is primarily concerned with implementation details.

The module uses pyparsing to parse the TeX expression.

The Bakoma distribution of the TeX Computer Modern fonts, and STIX fonts are supported. There is experimental support for using arbitrary fonts, but results may vary without proper tweaking and metrics for those fonts.

class matplotlib.mathtext.**Accent**(*c*, *state*)

Bases: *matplotlib.mathtext.Char*

The font metrics need to be dealt with differently for accents, since they are already offset correctly from the baseline in TrueType fonts.

grow()

render(*x*, *y*)

Render the character to the canvas.

shrink()

class matplotlib.mathtext.**AutoHeightChar**(*c*, *height*, *depth*, *state*, *always=False*, *factor=None*)

Bases: *matplotlib.mathtext.Hlist*

AutoHeightChar will create a character as close to the given height and depth as possible. When using a font with multiple height versions of some characters (such as the BaKoMa fonts), the correct glyph will be selected, otherwise this will always just return a scaled version of the glyph.

class matplotlib.mathtext.**AutoWidthChar**(*c*, *width*, *state*, *always=False*, *char_class=<class 'matplotlib.mathtext.Char'>*)

Bases: *matplotlib.mathtext.Hlist*

AutoWidthChar will create a character as close to the given width as possible. When using a font with multiple width versions of some characters (such as the BaKoMa fonts), the correct glyph will be selected, otherwise this will always just return a scaled version of the glyph.

class matplotlib.mathtext.**BakomaFonts**(**args*, ***kwargs*)

Bases: *matplotlib.mathtext.TruetypeFonts*

Use the Bakoma TrueType fonts for rendering.

Symbols are strewn about a number of font files, each of which has its own proprietary 8-bit encoding.

alias = '\\]'

get_sized_alternatives_for_symbol(*fontname*, *sym*)

target = ']'

class matplotlib.mathtext.**Box**(*width*, *height*, *depth*)
Bases: *matplotlib.mathtext.Node*

Represents any node with a physical location.

grow()

render(*x1*, *y1*, *x2*, *y2*)

shrink()

class matplotlib.mathtext.**Char**(*c*, *state*)
Bases: *matplotlib.mathtext.Node*

Represents a single character. Unlike TeX, the font information and metrics are stored with each *Char* to make it easier to lookup the font metrics when needed. Note that TeX boxes have a width, height, and depth, unlike Type1 and Truetype which use a full bounding box and an advance in the x-direction. The metrics must be converted to the TeX way, and the advance (if different from width) must be converted into a *Kern* node when the *Char* is added to its parent *Hlist*.

get_kerning(*next*)
Return the amount of kerning between this and the given character. Called when characters are strung together into *Hlist* to create *Kern* nodes.

grow()

is_slanted()

render(*x*, *y*)
Render the character to the canvas

shrink()

matplotlib.mathtext.**Error**(*msg*)
Helper class to raise parser errors.

class matplotlib.mathtext.**Fil**
Bases: *matplotlib.mathtext.Glue*

class matplotlib.mathtext.**Fill**
Bases: *matplotlib.mathtext.Glue*

class matplotlib.mathtext.**Filll**
Bases: *matplotlib.mathtext.Glue*

class `matplotlib.mathtext.`**`Fonts`**(*default_font_prop*, *mathtext_backend*)

Bases: `object`

An abstract base class for a system of fonts to use for mathtext.

The class must be able to take symbol keys and font file names and return the character metrics. It also delegates to a backend class to do the actual drawing.

default_font_prop: A `FontProperties` object to use for the default non-math font, or the base font for Unicode (generic) font rendering.

mathtext_backend: A subclass of `MathTextBackend` used to delegate the actual rendering.

`destroy`()

Fix any cyclical references before the object is about to be destroyed.

`get_kern`(*font1*, *fontclass1*, *sym1*, *fontsize1*, *font2*, *fontclass2*, *sym2*, *fontsize2*, *dpi*)

Get the kerning distance for font between *sym1* and *sym2*.

fontX: one of the TeX font names:

```
tt, it, rm, cal, sf, bf or default/regular (non-math)
```

fontclassX: TODO

symX: a symbol in raw TeX form. e.g., '1', 'x' or 'sigma'

fontsizeX: the fontsize in points

dpi: the current dots-per-inch

`get_metrics`(*font*, *font_class*, *sym*, *fontsize*, *dpi*)

font: one of the TeX font names:

```
tt, it, rm, cal, sf, bf or default/regular (non-math)
```

font_class: TODO

sym: a symbol in raw TeX form. e.g., '1', 'x' or 'sigma'

fontsize: font size in points

dpi: current dots-per-inch

Returns an object with the following attributes:

- *advance*: The advance distance (in points) of the glyph.
- *height*: The height of the glyph in points.
- *width*: The width of the glyph in points.
- *xmin*, *xmax*, *ymin*, *ymax* - the ink rectangle of the glyph
- *iceberg* - the distance from the baseline to the top of the glyph. This corresponds to TeX's definition of "height".

`get_results`(*box*)

Get the data needed by the backend to render the math expression. The return value is backend-specific.

get_sized_alternatives_for_symbol(*fontname*, *sym*)
Override if your font provides multiple sizes of the same symbol. Should return a list of symbols matching *sym* in various sizes. The expression renderer will select the most appropriate size for a given situation from this list.

get_underline_thickness(*font*, *fontsize*, *dpi*)
Get the line thickness that matches the given font. Used as a base unit for drawing lines such as in a fraction or radical.

get_used_characters()
Get the set of characters that were used in the math expression. Used by backends that need to subset fonts so they know which glyphs to include.

get_xheight(*font*, *fontsize*, *dpi*)
Get the xheight for the given *font* and *fontsize*.

render_glyph(*ox*, *oy*, *facename*, *font_class*, *sym*, *fontsize*, *dpi*)
Draw a glyph at

- *ox*, *oy*: position
- *facename*: One of the TeX face names
- *font_class*:
- *sym*: TeX symbol name or single character
- *fontsize*: fontsize in points
- *dpi*: The dpi to draw at.

render_rect_filled(*x1*, *y1*, *x2*, *y2*)
Draw a filled rectangle from (*x1*, *y1*) to (*x2*, *y2*).

set_canvas_size(*w*, *h*, *d*)
Set the size of the buffer used to render the math expression. Only really necessary for the bitmap backends.

class `matplotlib.mathtext.`**Glue**(*glue_type*, *copy=False*)
Bases: `matplotlib.mathtext.Node`

Most of the information in this object is stored in the underlying `GlueSpec` class, which is shared between multiple glue objects. (This is a memory optimization which probably doesn't matter anymore, but it's easier to stick to what TeX does.)

grow()

shrink()

class `matplotlib.mathtext.`**GlueSpec**(*width=0.0*, *stretch=0.0*, *stretch_order=0*, *shrink=0.0*, *shrink_order=0*)
Bases: `object`

See `Glue`.

copy()

classmethod factory(*glue_type*)

class `matplotlib.mathtext.`**HCentered**(*elements*)
Bases: `matplotlib.mathtext.Hlist`

A convenience class to create an `Hlist` whose contents are centered within its enclosing box.

class `matplotlib.mathtext.`**Hbox**(*width*)
Bases: `matplotlib.mathtext.Box`

A box with only width (zero height and depth).

class `matplotlib.mathtext.`**Hlist**(*elements*, *w=0.0*, *m='additional'*, *do_kern=True*)
Bases: `matplotlib.mathtext.List`

A horizontal list of boxes.

hpack(*w=0.0*, *m='additional'*)
The main duty of `hpack()` is to compute the dimensions of the resulting boxes, and to adjust the glue if one of those dimensions is pre-specified. The computed sizes normally enclose all of the material inside the new box; but some items may stick out if negative glue is used, if the box is overfull, or if a `\vbox` includes other boxes that have been shifted left.

- *w*: specifies a width
- *m*: is either 'exactly' or 'additional'.

Thus, `hpack(w, 'exactly')` produces a box whose width is exactly *w*, while `hpack(w, 'additional')` yields a box whose width is the natural width plus *w*. The default values produce a box with the natural width.

kern()
Insert `Kern` nodes between `Char` nodes to set kerning. The `Char` nodes themselves determine the amount of kerning they need (in `get_kerning()`), and this function just creates the linked list in the correct way.

class `matplotlib.mathtext.`**Hrule**(*state*, *thickness=None*)
Bases: `matplotlib.mathtext.Rule`

Convenience class to create a horizontal rule.

class `matplotlib.mathtext.`**Kern**(*width*)
Bases: `matplotlib.mathtext.Node`

A `Kern` node has a width field to specify a (normally negative) amount of spacing. This spacing correction appears in horizontal lists between letters like A and V when the font designer said that it looks better to move them closer together or further apart. A kern node can also appear in a vertical list, when its *width* denotes additional spacing in the vertical direction.

depth = 0

grow()

height = 0

shrink()

class matplotlib.mathtext.**List**(*elements*)
Bases: `matplotlib.mathtext.Box`

A list of nodes (either horizontal or vertical).

grow()

shrink()

class matplotlib.mathtext.**MathTextParser**(*output*)
Bases: `object`

Create a MathTextParser for the given backend *output*.

get_depth(*texstr*, *dpi=120*, *fontsize=14*)
Returns the offset of the baseline from the bottom of the image in pixels.
texstr A valid mathtext string, e.g., r'IQ: $sigma_i=15$'
dpi The dots-per-inch to render the text
fontsize The font size in points

parse(*s*, *dpi=72*, *prop=None*)
Parse the given math expression *s* at the given *dpi*. If *prop* is provided, it is a `FontProperties` object specifying the "default" font to use in the math expression, used for all non-math text.

The results are cached, so multiple calls to `parse()` with the same expression should be fast.

to_mask(*texstr*, *dpi=120*, *fontsize=14*)

texstr A valid mathtext string, e.g., r'IQ: $sigma_i=15$'
dpi The dots-per-inch to render the text
fontsize The font size in points
Returns a tuple (*array*, *depth*)
- *array* is an NxM uint8 alpha ubyte mask array of rasterized tex.
- depth is the offset of the baseline from the bottom of the image in pixels.

to_png(*filename*, *texstr*, *color='black'*, *dpi=120*, *fontsize=14*)
Writes a tex expression to a PNG file.

Returns the offset of the baseline from the bottom of the image in pixels.
filename A writable filename or fileobject
texstr A valid mathtext string, e.g., r'IQ: $sigma_i=15$'
color A valid matplotlib color argument
dpi The dots-per-inch to render the text
fontsize The font size in points
Returns the offset of the baseline from the bottom of the image in pixels.

to_rgba(*texstr*, *color='black'*, *dpi=120*, *fontsize=14*)

texstr A valid mathtext string, e.g., r'IQ: $sigma_i=15$'

color Any matplotlib color argument
dpi The dots-per-inch to render the text
fontsize The font size in points
Returns a tuple (*array*, *depth*)

- *array* is an NxM uint8 alpha ubyte mask array of rasterized tex.
- depth is the offset of the baseline from the bottom of the image in pixels.

exception `matplotlib.mathtext.`**`MathTextWarning`**
Bases: `Warning`

class `matplotlib.mathtext.`**`MathtextBackend`**
Bases: `object`

The base class for the mathtext backend-specific code. The purpose of *`MathtextBackend`* subclasses is to interface between mathtext and a specific matplotlib graphics backend.

Subclasses need to override the following:

- *`render_glyph()`*
- *`render_rect_filled()`*
- *`get_results()`*

And optionally, if you need to use a Freetype hinting style:

- *`get_hinting_type()`*

`get_hinting_type`()
Get the Freetype hinting type to use with this particular backend.

`get_results`(*box*)
Return a backend-specific tuple to return to the backend after all processing is done.

`render_glyph`(*ox*, *oy*, *info*)
Draw a glyph described by *info* to the reference point (*ox*, *oy*).

`render_rect_filled`(*x1*, *y1*, *x2*, *y2*)
Draw a filled black rectangle from (*x1*, *y1*) to (*x2*, *y2*).

`set_canvas_size`(*w*, *h*, *d*)
Dimension the drawing canvas

class `matplotlib.mathtext.`**`MathtextBackendAgg`**
Bases: *`matplotlib.mathtext.MathtextBackend`*

Render glyphs and rectangles to an FTImage buffer, which is later transferred to the Agg image by the Agg backend.

`get_hinting_type`()

`get_results`(*box*, *used_characters*)

`render_glyph`(*ox*, *oy*, *info*)

`render_rect_filled`(*x1*, *y1*, *x2*, *y2*)

set_canvas_size(*w*, *h*, *d*)

class matplotlib.mathtext.**MathtextBackendBitmap**

Bases: *matplotlib.mathtext.MathtextBackendAgg*

get_results(*box*, *used_characters*)

class matplotlib.mathtext.**MathtextBackendCairo**

Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the Cairo backend.

get_results(*box*, *used_characters*)

render_glyph(*ox*, *oy*, *info*)

render_rect_filled(*x1*, *y1*, *x2*, *y2*)

class matplotlib.mathtext.**MathtextBackendPath**

Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the text path machinery.

get_results(*box*, *used_characters*)

render_glyph(*ox*, *oy*, *info*)

render_rect_filled(*x1*, *y1*, *x2*, *y2*)

class matplotlib.mathtext.**MathtextBackendPdf**

Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the PDF backend.

get_results(*box*, *used_characters*)

render_glyph(*ox*, *oy*, *info*)

render_rect_filled(*x1*, *y1*, *x2*, *y2*)

class matplotlib.mathtext.**MathtextBackendPs**

Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the PostScript backend.

get_results(*box*, *used_characters*)

render_glyph(*ox*, *oy*, *info*)

render_rect_filled(*x1*, *y1*, *x2*, *y2*)

class matplotlib.mathtext.**MathtextBackendSvg**

Bases: *matplotlib.mathtext.MathtextBackend*

Store information to write a mathtext rendering to the SVG backend.

get_results(*box*, *used_characters*)

render_glyph(*ox*, *oy*, *info*)

render_rect_filled(*x1*, *y1*, *x2*, *y2*)

class matplotlib.mathtext.**NegFil**

Bases: *matplotlib.mathtext.Glue*

class matplotlib.mathtext.**NegFill**

Bases: *matplotlib.mathtext.Glue*

class matplotlib.mathtext.**NegFilll**

Bases: *matplotlib.mathtext.Glue*

class matplotlib.mathtext.**Node**

Bases: object

A node in the TeX box model

get_kerning(*next*)

grow()

Grows one level larger. There is no limit to how big something can get.

render(*x*, *y*)

shrink()

Shrinks one level smaller. There are only three levels of sizes, after which things will no longer get smaller.

class matplotlib.mathtext.**Parser**

Bases: object

This is the pyparsing-based parser for math expressions. It actually parses full strings *containing* math expressions, in that raw text may also appear outside of pairs of $.

The grammar is based directly on that in TeX, though it cuts a few corners.

class State(*font_output*, *font*, *font_class*, *fontsize*, *dpi*)

Bases: object

Stores the state of the parser.

States are pushed and popped from a stack as necessary, and the "current" state is always at the top of the stack.

copy()

font

Parser.**accent**(*s*, *loc*, *toks*)

Parser.**auto_delim**(*s*, *loc*, *toks*)

Parser.**binom**(*s*, *loc*, *toks*)

Parser.**c_over_c**(*s*, *loc*, *toks*)

Parser.**customspace**(*s*, *loc*, *toks*)

Parser.**end_group**(*s*, *loc*, *toks*)

Parser.**font**(*s*, *loc*, *toks*)

Parser.**frac**(*s*, *loc*, *toks*)

Parser.**function**(*s*, *loc*, *toks*)

Parser.**genfrac**(*s*, *loc*, *toks*)

Parser.**get_state**()

Get the current *State* of the parser.

Parser.**group**(*s*, *loc*, *toks*)

Parser.**is_dropsub**(*nucleus*)

Parser.**is_overunder**(*nucleus*)

Parser.**is_slanted**(*nucleus*)

`Parser.main(s, loc, toks)`

`Parser.math(s, loc, toks)`

`Parser.math_string(s, loc, toks)`

`Parser.non_math(s, loc, toks)`

`Parser.operatorname(s, loc, toks)`

`Parser.overline(s, loc, toks)`

`Parser.parse(s, fonts_object, fontsize, dpi)`
Parse expression *s* using the given *fonts_object* for output, at the given *fontsize* and *dpi*.

Returns the parse tree of `Node` instances.

`Parser.pop_state()`
Pop a `State` off of the stack.

`Parser.push_state()`
Push a new `State` onto the stack which is just a copy of the current state.

`Parser.required_group(s, loc, toks)`

`Parser.simple_group(s, loc, toks)`

`Parser.snowflake(s, loc, toks)`

`Parser.space(s, loc, toks)`

`Parser.sqrt(s, loc, toks)`

`Parser.stackrel(s, loc, toks)`

`Parser.start_group(s, loc, toks)`

`Parser.subsuper(s, loc, toks)`

`Parser.symbol(s, loc, toks)`

Parser.**unknown_symbol**(*s*, *loc*, *toks*)

class matplotlib.mathtext.**Rule**(*width*, *height*, *depth*, *state*)
Bases: *matplotlib.mathtext.Box*

A *Rule* node stands for a solid black rectangle; it has *width*, *depth*, and *height* fields just as in an *Hlist*. However, if any of these dimensions is inf, the actual value will be determined by running the rule up to the boundary of the innermost enclosing box. This is called a "running dimension." The width is never running in an *Hlist*; the height and depth are never running in a *Vlist*.

render(*x*, *y*, *w*, *h*)

class matplotlib.mathtext.**Ship**
Bases: object

Once the boxes have been set up, this sends them to output. Since boxes can be inside of boxes inside of boxes, the main work of *Ship* is done by two mutually recursive routines, *hlist_out()* and *vlist_out()*, which traverse the *Hlist* nodes and *Vlist* nodes inside of horizontal and vertical boxes. The global variables used in TeX to store state as it processes have become member variables here.

static clamp(*value*)

hlist_out(*box*)

vlist_out(*box*)

class matplotlib.mathtext.**SsGlue**
Bases: *matplotlib.mathtext.Glue*

class matplotlib.mathtext.**StandardPsFonts**(*default_font_prop*)
Bases: *matplotlib.mathtext.Fonts*

Use the standard postscript fonts for rendering to backend_ps

Unlike the other font classes, BakomaFont and UnicodeFont, this one requires the Ps backend.

basepath = '/home/tcaswell/.virtualenvs/dd35/lib/python3.5/site-packages/matplotlib/mpl-data/fonts/afm'

fontmap = {'tt': 'pcrr8a', None: 'psyr', 'bf': 'pncb8a', 'sf': 'phvr8a', 'cal': 'pzcmi8a', 'it': 'pncri8a', 'rm': 'p

get_kern(*font1*, *fontclass1*, *sym1*, *fontsize1*, *font2*, *fontclass2*, *sym2*, *fontsize2*, *dpi*)

get_underline_thickness(*font*, *fontsize*, *dpi*)

get_xheight(*font*, *fontsize*, *dpi*)

class matplotlib.mathtext.**StixFonts**(**args*, ***kwargs*)
Bases: *matplotlib.mathtext.UnicodeFonts*

A font handling class for the STIX fonts.

In addition to what UnicodeFonts provides, this class:

- supports "virtual fonts" which are complete alpha numeric character sets with different font styles at special Unicode code points, such as "Blackboard".
- handles sized alternative characters for the STIXSizeX fonts.

cm_fallback = False

get_sized_alternatives_for_symbol(*fontname*, *sym*)

use_cmex = False

class matplotlib.mathtext.**StixSansFonts**(**args*, ***kwargs*)
Bases: *matplotlib.mathtext.StixFonts*

A font handling class for the STIX fonts (that uses sans-serif characters by default).

class matplotlib.mathtext.**SubSuperCluster**
Bases: *matplotlib.mathtext.Hlist*

SubSuperCluster is a sort of hack to get around that fact that this code do a two-pass parse like TeX. This lets us store enough information in the hlist itself, namely the nucleus, sub- and super-script, such that if another script follows that needs to be attached, it can be reconfigured on the fly.

class matplotlib.mathtext.**TruetypeFonts**(*default_font_prop*, *mathtext_backend*)
Bases: *matplotlib.mathtext.Fonts*

A generic base class for all font setups that use Truetype fonts (through FT2Font).

class **CachedFont**(*font*)
Bases: object

TruetypeFonts.**destroy**()

TruetypeFonts.**get_kern**(*font1*, *fontclass1*, *sym1*, *fontsize1*, *font2*, *fontclass2*, *sym2*, *fontsize2*, *dpi*)

TruetypeFonts.**get_underline_thickness**(*font*, *fontsize*, *dpi*)

TruetypeFonts.**get_xheight**(*font*, *fontsize*, *dpi*)

class matplotlib.mathtext.**UnicodeFonts**(**args*, ***kwargs*)
Bases: *matplotlib.mathtext.TruetypeFonts*

An abstract base class for handling Unicode fonts.

While some reasonably complete Unicode fonts (such as DejaVu) may work in some situations, the only Unicode font I'm aware of with a complete set of math symbols is STIX.

This class will "fallback" on the Bakoma fonts when a required symbol can not be found in the font.

get_sized_alternatives_for_symbol(*fontname*, *sym*)

use_cmex = True

class `matplotlib.mathtext.`**VCentered**(*elements*)
Bases: `matplotlib.mathtext.Hlist`

A convenience class to create a `Vlist` whose contents are centered within its enclosing box.

class `matplotlib.mathtext.`**Vbox**(*height*, *depth*)
Bases: `matplotlib.mathtext.Box`

A box with only height (zero width).

class `matplotlib.mathtext.`**Vlist**(*elements*, *h=0.0*, *m='additional'*)
Bases: `matplotlib.mathtext.List`

A vertical list of boxes.

vpack(*h=0.0*, *m='additional'*, *l=inf*)
The main duty of `vpack()` is to compute the dimensions of the resulting boxes, and to adjust the glue if one of those dimensions is pre-specified.

- *h*: specifies a height
- *m*: is either 'exactly' or 'additional'.
- *l*: a maximum height

Thus, `vpack(h, 'exactly')` produces a box whose height is exactly *h*, while `vpack(h, 'additional')` yields a box whose height is the natural height plus *h*. The default values produce a box with the natural width.

class `matplotlib.mathtext.`**Vrule**(*state*)
Bases: `matplotlib.mathtext.Rule`

Convenience class to create a vertical rule.

`matplotlib.mathtext.`**get_unicode_index**(*symbol*) → integer
Return the integer index (from the Unicode table) of symbol. *symbol* can be a single unicode character, a TeX command (i.e. r'pi'), or a Type1 symbol name (i.e. 'phi').

`matplotlib.mathtext.`**math_to_image**(*s*, *filename_or_obj*, *prop=None*, *dpi=None*, *format=None*)
Given a math expression, renders it in a closely-clipped bounding box to an image file.
s A math expression. The math portion should be enclosed in dollar signs.
filename_or_obj A filepath or writable file-like object to write the image data to.
prop If provided, a FontProperties() object describing the size and style of the text.
dpi Override the output dpi, otherwise use the default associated with the output format.
format The output format, e.g., 'svg', 'pdf', 'ps' or 'png'. If not provided, will be deduced from the filename.

`matplotlib.mathtext.unichr_safe(index)`

Return the Unicode character corresponding to the index, or the replacement character if this is a narrow build of Python and the requested character is outside the BMP.

CHAPTER

SIXTYTWO

MLAB

62.1 matplotlib.mlab

Numerical python functions written for compatability with MATLAB commands with the same names.

62.1.1 MATLAB compatible functions

cohere() Coherence (normalized cross spectral density)

csd() Cross spectral density uing Welch's average periodogram

detrend() Remove the mean or best fit line from an array

find() Return the indices where some condition is true; numpy.nonzero is similar but more general.

griddata() Interpolate irregularly distributed data to a regular grid.

prctile() Find the percentiles of a sequence

prepca() Principal Component Analysis

psd() Power spectral density uing Welch's average periodogram

rk4() A 4th order runge kutta integrator for 1D or ND systems

specgram() Spectrogram (spectrum over segments of time)

62.1.2 Miscellaneous functions

Functions that don't exist in MATLAB, but are useful anyway:

cohere_pairs() Coherence over all pairs. This is not a MATLAB function, but we compute coherence a lot in my lab, and we compute it for a lot of pairs. This function is optimized to do this efficiently by caching the direct FFTs.

rk4() A 4th order Runge-Kutta ODE integrator in case you ever find yourself stranded without scipy (and the far superior scipy.integrate tools)

contiguous_regions() Return the indices of the regions spanned by some logical mask

cross_from_below() Return the indices where a 1D array crosses a threshold from below *cross_from_above()* Return the indices where a 1D array crosses a threshold from above

complex_spectrum() Return the complex-valued frequency spectrum of a signal

magnitude_spectrum() Return the magnitude of the frequency spectrum of a signal

angle_spectrum() Return the angle (wrapped phase) of the frequency spectrum of a signal

phase_spectrum() Return the phase (unwrapped angle) of the frequency spectrum of a signal

detrend_mean() Remove the mean from a line.

demean() Remove the mean from a line. This function is the same as as *detrend_mean()* except for the default *axis*.

detrend_linear() Remove the best fit line from a line.

detrend_none() Return the original line.

stride_windows() Get all windows in an array in a memory-efficient manner

stride_repeat() Repeat an array in a memory-efficient manner

apply_window() Apply a window along a given axis

62.1.3 record array helper functions

A collection of helper methods for numpyrecord arrays

See *misc Examples*

rec2txt() Pretty print a record array

rec2csv() Store record array in CSV file

csv2rec() Import record array from CSV file with type inspection

rec_append_fields() Adds field(s)/array(s) to record array

rec_drop_fields() Drop fields from record array

rec_join() Join two record arrays on sequence of fields

recs_join() A simple join of multiple recarrays using a single column as a key

rec_groupby() Summarize data by groups (similar to SQL GROUP BY)

rec_summarize() Helper code to filter rec array fields into new fields

For the rec viewer functions(e rec2csv), there are a bunch of Format objects you can pass into the functions that will do things like color negative values red, set percent formatting and scaling, etc.

Example usage:

```
r = csv2rec('somefile.csv', checkrows=0)

formatd = dict(
    weight = FormatFloat(2),
    change = FormatPercent(2),
```

```
    cost   = FormatThousands(2),
    )


rec2excel(r, 'test.xls', formatd=formatd)
rec2csv(r, 'test.csv', formatd=formatd)
scroll = rec2gtk(r, formatd=formatd)

win = gtk.Window()
win.set_size_request(600,800)
win.add(scroll)
win.show_all()
gtk.main()
```

class matplotlib.mlab.**FormatBool**
Bases: *matplotlib.mlab.FormatObj*

fromstr(*s*)

toval(*x*)

class matplotlib.mlab.**FormatDate**(*fmt*)
Bases: *matplotlib.mlab.FormatObj*

fromstr(*x*)

toval(*x*)

class matplotlib.mlab.**FormatDatetime**(*fmt='%Y-%m-%d %H:%M:%S'*)
Bases: *matplotlib.mlab.FormatDate*

fromstr(*x*)

class matplotlib.mlab.**FormatFloat**(*precision=4*, *scale=1.0*)
Bases: *matplotlib.mlab.FormatFormatStr*

fromstr(*s*)

toval(*x*)

class matplotlib.mlab.**FormatFormatStr**(*fmt*)
Bases: *matplotlib.mlab.FormatObj*

tostr(*x*)

class matplotlib.mlab.**FormatInt**
Bases: *matplotlib.mlab.FormatObj*

fromstr(*s*)

tostr(*x*)

toval(*x*)

class matplotlib.mlab.**FormatMillions**(*precision=4*)
Bases: *matplotlib.mlab.FormatFloat*

class matplotlib.mlab.**FormatObj**
Bases: object

fromstr(*s*)

tostr(*x*)

toval(*x*)

class matplotlib.mlab.**FormatPercent**(*precision=4*)
Bases: *matplotlib.mlab.FormatFloat*

class matplotlib.mlab.**FormatString**
Bases: *matplotlib.mlab.FormatObj*

tostr(*x*)

class matplotlib.mlab.**FormatThousands**(*precision=4*)
Bases: *matplotlib.mlab.FormatFloat*

class matplotlib.mlab.**GaussianKDE**(*dataset*, *bw_method=None*)
Bases: object

Representation of a kernel-density estimate using Gaussian kernels.

Call signature:: kde = GaussianKDE(dataset, bw_method='silverman')

Parameters dataset : array_like
Datapoints to estimate from. In case of univariate data this is a 1-D array, otherwise a 2-D array with shape (# of dims, # of data).

bw_method : str, scalar or callable, optional
The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If a scalar, this will be used directly as `kde.factor`. If a callable, it should take a *GaussianKDE* instance as only parameter and return a scalar. If None (default), 'scott' is used.

Attributes

dataset	(ndarray) The dataset with which `gaussian_kde` was initialized.
dim	(int) Number of dimensions.
num_dp	(int) Number of datapoints.
factor	(float) The bandwidth factor, obtained from `kde.covariance_factor`, with which the covariance matrix is multiplied.
covariance	(ndarray) The covariance matrix of `dataset`, scaled by the calculated bandwidth (`kde.factor`).
inv_cov	(ndarray) The inverse of `covariance`.

Methods

kde.evaluate(points)	(ndarray) Evaluate the estimated pdf on a provided set of points.
kde(points)	(ndarray) Same as kde.evaluate(points)

covariance_factor()

evaluate(*points*)

Evaluate the estimated pdf on a set of points.

Parameters points : (# of dimensions, # of points)-array

Alternatively, a (# of dimensions,) vector can be passed in and treated as a single point.

Returns values : (# of points,)-array

The values at each point.

Raises ValueError : if the dimensionality of the input points is different than the dimensionality of the KDE.

scotts_factor()

silverman_factor()

class `matplotlib.mlab.`**PCA**(*a*, *standardize=True*)

Bases: `object`

compute the SVD of a and store data for PCA. Use project to project the data onto a reduced set of dimensions

Inputs:

a: a numobservations x numdims array *standardize*: True if input data are to be standardized. If False, only centering will be carried out.

Attrs:

a a centered unit sigma version of input a

numrows, *numcols*: the dimensions of a *mu*: a numdims array of means of a. This is the vector that points to the origin of PCA space.

sigma: a numdims array of standard deviation of a

fracs: the proportion of variance of each of the principal components

s: the actual eigenvalues of the decomposition

Wt: the weight vector for projecting a numdims point or array into PCA space

Y: a projected into PCA space

The factor loadings are in the Wt factor, i.e., the factor loadings for the 1st principal component are given by Wt[0]. This row is also the 1st eigenvector.

center(*x*)
center and optionally standardize the data using the mean and sigma from training set a

project(*x*, *minfrac=0.0*)
project x onto the principle axes, dropping any axes where fraction of variance<minfrac

matplotlib.mlab.**amap**(*function*, *sequence*[, *sequence*, *...*]) → array.
Works like `map()`, but it returns an array. This is just a convenient shorthand for `numpy.array(map(...))`.

matplotlib.mlab.**angle_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)
Compute the angle of the frequency spectrum (wrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Returns the tuple (*spectrum*, *freqs*):

***spectrum*: 1-D array** The values for the angle spectrum in radians (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

See also:

`complex_spectrum()` This function returns the angle value of `complex_spectrum()`.

`magnitude_spectrum()` `angle_spectrum()` returns the magnitudes of the corresponding frequencies.

`phase_spectrum()` `phase_spectrum()` returns the unwrapped version of this function.

`specgram()` `specgram()` can return the angle spectrum of segments within the signal.

`matplotlib.mlab.`**`apply_window`**(*x*, *window*, *axis=0*, *return_window=None*)

Apply the given window to the given 1D or 2D array along the given axis.

Call signature:

```
  apply_window(x, window, axis=0, return_window=False)

*x*: 1D or 2D array or sequence
  Array or sequence containing the data.

*winodw*: function or array.
  Either a function to generate a window or an array with length
  *x*.shape[*axis*]

*axis*: integer
  The axis over which to do the repetition.
  Must be 0 or 1.  The default is 0

*return_window*: bool
  If true, also return the 1D values of the window that was applied
```

`matplotlib.mlab.`**`base_repr`**(*number*, *base=2*, *padding=0*)

Return the representation of a *number* in any given *base*.

`matplotlib.mlab.`**`binary_repr`**(*number*, *max_length=1025*)

Return the binary representation of the input *number* as a string.

This is more efficient than using `base_repr()` with base 2.

Increase the value of max_length for very large numbers. Note that on 32-bit machines, 2**1023 is the largest integer power of 2 which can be converted to a Python float.

`matplotlib.mlab.`**`bivariate_normal`**(*X*, *Y*, *sigmax=1.0*, *sigmay=1.0*, *mux=0.0*, *muy=0.0*, *sigmaxy=0.0*)

Bivariate Gaussian distribution for equal shape *X*, *Y*.

See bivariate normal at mathworld.

`matplotlib.mlab.`**`center_matrix`**(*M*, *dim=0*)

Return the matrix *M* with each row having zero mean and unit std.

If *dim* = 1 operate on columns instead of rows. (*dim* is opposite to the numpy axis kwarg.)

`matplotlib.mlab.`**`cohere`**(*x*, *y*, *NFFT=256*, *Fs=2*, *detrend=<function detrend_none>*, *window=<function window_hanning>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*)

The coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \tag{62.1}$$

x, y Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or**
callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***noverlap*: integer** The number of points of overlap between blocks. The default value is 0 (no overlap).

The return value is the tuple (Cxy, f), where f are the frequencies of the coherence vector. For cohere, scaling the individual densities by the sampling frequency has no effect, since the factors cancel out.

See also:

`psd()` and `csd()` For information about the methods used to compute P_{xy}, P_{xx} and P_{yy}.

`matplotlib.mlab.`**`cohere_pairs`**(*X*, *ij*, *NFFT=256*, *Fs=2*, *detrend=<function detrend_none>*, *window=<function window_hanning>*, *noverlap=0*, *preferSpeedOverMemory=True*, *progressCallback=<function donothing_callback>*, *returnPxx=False*)

Call signature:

```
Cxy, Phase, freqs = cohere_pairs( X, ij, ...)
```

Compute the coherence and phase for all pairs *ij*, in *X*.

X is a *numSamples* * *numCols* array

ij is a list of tuples. Each tuple is a pair of indexes into the columns of X for which you want to compute coherence. For example, if *X* has 64 columns, and you want to compute all nonredundant pairs, define *ij* as:

```
ij = []
for i in range(64):
    for j in range(i+1,64):
        ij.append( (i,j) )
```

preferSpeedOverMemory is an optional bool. Defaults to true. If False, limits the caching by only making one, rather than two, complex cache arrays. This is useful if memory becomes critical. Even when *preferSpeedOverMemory* is False, `cohere_pairs()` will still give significant performace gains over calling `cohere()` for each pair, and will use subtantially less memory than if *preferSpeedOverMemory* is True. In my tests with a 43000,64 array over all nonredundant pairs, *preferSpeedOverMemory* = True delivered a 33% performance boost on a 1.7GHZ Athlon with 512MB RAM compared with *preferSpeedOverMemory* = False. But both solutions were more than 10x faster than naively crunching all possible pairs through `cohere()`.

Returns:

```
(Cxy, Phase, freqs)
```

where:

- *Cxy*: dictionary of (*i*, *j*) tuples -> coherence vector for that pair. i.e., `Cxy[(i,j) = cohere(X[:,i], X[:,j])`. Number of dictionary keys is `len(ij)`.
- *Phase*: dictionary of phases of the cross spectral density at each frequency for each pair. Keys are (*i*, *j*).
- ***freqs*: vector of frequencies, equal in length to either the** coherence or phase vectors for any (*i*, *j*) key.

e.g., to make a coherence Bode plot:

```
subplot(211)
plot( freqs, Cxy[(12,19)])
subplot(212)
plot( freqs, Phase[(12,19)])
```

For a large number of pairs, `cohere_pairs()` can be much more efficient than just calling `cohere()` for each pair, because it caches most of the intensive computations. If N is the number of pairs, this function is $O(N)$ for most of the heavy lifting, whereas calling cohere for each pair is $O(N^2)$. However, because of the caching, it is also more memory intensive, making 2 additional complex arrays with approximately the same number of elements as *X*.

See `test/cohere_pairs_test.py` in the src tree for an example script that shows that this `cohere_pairs()` and `cohere()` give the same results for a given pair.

See also:

`psd()` For information about the methods used to compute P_{xy}, P_{xx} and P_{yy}.

matplotlib.mlab.**complex_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the complex-valued frequency spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Returns the tuple (*spectrum*, *freqs*):

***spectrum*: 1-D array** The values for the complex spectrum (complex valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

See also:

`magnitude_spectrum()` `magnitude_spectrum()` returns the absolute value of this function.
`angle_spectrum()` `angle_spectrum()` returns the angle of this function.
`phase_spectrum()` `phase_spectrum()` returns the phase (unwrapped angle) of this function.
`specgram()` `specgram()` can return the complex spectrum of segments within the signal.

matplotlib.mlab.**contiguous_regions**(*mask*)

return a list of (ind0, ind1) such that mask[ind0:ind1].all() is True and we cover all such regions

matplotlib.mlab.**cross_from_above**(*x*, *threshold*)

return the indices into *x* where *x* crosses some threshold from below, e.g., the i's where:

```
x[i-1]>threshold and x[i]<=threshold
```

See also:

`cross_from_below()` and `contiguous_regions()`

matplotlib.mlab.**cross_from_below**(*x*, *threshold*)

return the indices into *x* where *x* crosses some threshold from below, e.g., the i's where:

```
x[i-1]<threshold and x[i]>=threshold
```

Example code:

```
import matplotlib.pyplot as plt

t = np.arange(0.0, 2.0, 0.1)
s = np.sin(2*np.pi*t)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(t, s, '-o')
ax.axhline(0.5)
ax.axhline(-0.5)

ind = cross_from_below(s, 0.5)
ax.vlines(t[ind], -1, 1)

ind = cross_from_above(s, -0.5)
ax.vlines(t[ind], -1, 1)

plt.show()
```

See also:

cross_from_above() and *contiguous_regions()*

matplotlib.mlab.**csd**(*x*, *y*, *NFFT=None*, *Fs=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*)

Compute the cross-spectral density.

Call signature:

```
csd(x, y, NFFT=256, Fs=2, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None)
```

The cross spectral density P_{xy} by Welch's average periodogram method. The vectors *x* and *y* are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of *x* and *y* are averaged over each segment to compute P_{xy}, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero padded to *NFFT*.

***x*, *y*: 1-D arrays or sequences** Arrays or sequences containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see *window_hanning()*, *window_none()*, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or** callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***noverlap*: integer** The number of points of overlap between segments. The default value is 0 (no overlap).

Returns the tuple (*Pxy*, *freqs*):

***Pxy*: 1-D array** The values for the cross spectrum `P_{xy}` before scaling (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *Pxy*

Refs: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

See also:

`psd()` `psd()` is the equivalent to setting y=x.

`matplotlib.mlab.csv2rec`(*fname*, *comments='#'*, *skiprows=0*, *checkrows=0*, *delimiter='*, *'*, *converterd=None*, *names=None*, *missing=''*, *missingd=None*, *use_mrecords=False*, *dayfirst=False*, *yearfirst=False*)

Load data from comma/space/tab delimited file in *fname* into a numpy record array and return the record array.

If *names* is *None*, a header row is required to automatically assign the recarray names. The headers will be lower cased, spaces will be converted to underscores, and illegal attribute name characters removed. If *names* is not *None*, it is a sequence of names to use for the column names. In this case, it is assumed there is no header row.

- *fname*: can be a filename or a file handle. Support for gzipped files is automatic, if the filename ends in '.gz'
- *comments*: the character used to indicate the start of a comment in the file, or *None* to switch off the removal of comments
- *skiprows*: is the number of rows from the top to skip
- *checkrows*: is the number of rows to check to validate the column data type. When set to zero all rows are validated.
- *converterd*: if not *None*, is a dictionary mapping column number or munged column name to a converter function.
- *names*: if not None, is a list of header names. In this case, no header will be read from the file
- *missingd* is a dictionary mapping munged column names to field values which signify that the field does not contain actual data and should be masked, e.g., '0000-00-00' or 'unused'
- *missing*: a string whose value signals a missing field regardless of the column it appears in
- *use_mrecords*: if True, return an mrecords.fromrecords record array if any of the data are missing
- *dayfirst*: default is False so that MM-DD-YY has precedence over DD-MM-YY. See http://labix.org/python-dateutil#head-b95ce2094d189a89f80f5ae52a05b4ab7b41af47 for further information.
- *yearfirst*: default is False so that MM-DD-YY has precedence over YY-MM-DD. See http://labix.org/python-dateutil#head-b95ce2094d189a89f80f5ae52a05b4ab7b41af47 for further information.

If no rows are found, *None* is returned – see `examples/loadrec.py`

`matplotlib.mlab.`**`csvformat_factory`**(*format*)

`matplotlib.mlab.`**`demean`**(*x*, *axis=0*)

Return x minus its mean along the specified axis.

Call signature:

```
  demean(x, axis=0)

*x*: array or sequence
  Array or sequence containing the data
  Can have any dimensionality

*axis*: integer
  The axis along which to take the mean.  See numpy.mean for a
  description of this argument.
```

See also:

`delinear()`
`denone()` `delinear()` and `denone()` are other detrend algorithms.
`detrend_mean()` This function is the same as as *`detrend_mean()`* except for the default *axis*.

`matplotlib.mlab.`**`detrend`**(*x*, *key=None*, *axis=None*)

Return x with its trend removed.

Call signature:

```
  detrend(x, key='mean')

*x*: array or sequence
  Array or sequence containing the data.

*key*: [ 'default' | 'constant' | 'mean' | 'linear' | 'none'] or function
    Specifies the detrend algorithm to use.  'default' is 'mean',
    which is the same as :func:`detrend_mean`.  'constant' is the same.
    'linear' is the same as :func:`detrend_linear`.  'none' is the same
    as :func:`detrend_none`.  The default is 'mean'.  See the
    corresponding functions for more details regarding the algorithms.
    Can also be a function that carries out the detrend operation.

*axis*: integer
  The axis along which to do the detrending.
```

See also:

detrend_mean() *detrend_mean()* implements the 'mean' algorithm.
detrend_linear() *detrend_linear()* implements the 'linear' algorithm.
detrend_none() *detrend_none()* implements the 'none' algorithm.

matplotlib.mlab.**detrend_linear**(*y*)
Return x minus best fit line; 'linear' detrending.

Call signature:

```
  detrend_linear(y)

*y*: 0-D or 1-D array or sequence
  Array or sequence containing the data

*axis*: integer
  The axis along which to take the mean.  See numpy.mean for a
  description of this argument.
```

See also:

delinear() This function is the same as as delinear() except for the default *axis*.

detrend_mean()
detrend_none() *detrend_mean()* and *detrend_none()* are other detrend algorithms.
detrend() *detrend()* is a wrapper around all the detrend algorithms.

matplotlib.mlab.**detrend_mean**(*x*, *axis=None*)
Return x minus the mean(x).

Call signature:

```
  detrend_mean(x, axis=None)

*x*: array or sequence
  Array or sequence containing the data
  Can have any dimensionality
```

```
*axis*: integer
  The axis along which to take the mean.  See numpy.mean for a
  description of this argument.
```

See also:

demean() This function is the same as as *demean()* except for the default *axis*.

detrend_linear()
detrend_none() *detrend_linear()* and *detrend_none()* are other detrend algorithms.
detrend() *detrend()* is a wrapper around all the detrend algorithms.

matplotlib.mlab.**detrend_none**(*x*, *axis=None*)
Return x: no detrending.

Call signature:

```
  detrend_none(x, axis=None)

*x*: any object
  An object containing the data

*axis*: integer
  This parameter is ignored.
  It is included for compatibility with detrend_mean
```

See also:

denone() This function is the same as as denone() except for the default *axis*, which has no effect.

detrend_mean()
detrend_linear() *detrend_mean()* and *detrend_linear()* are other detrend algorithms.
detrend() *detrend()* is a wrapper around all the detrend algorithms.

matplotlib.mlab.**dist**(*x*, *y*)
Return the distance between two points.

matplotlib.mlab.**dist_point_to_segment**(*p*, *s0*, *s1*)
Get the distance of a point to a segment.

p, *s0*, *s1* are *xy* sequences

This algorithm from http://softsurfer.com/Archive/algorithm_0102/algorithm_0102.htm#Distance%20to%20Ray%20or%20Segment

matplotlib.mlab.**distances_along_curve**(*X*)
Computes the distance between a set of successive points in *N* dimensions.

Where *X* is an *M* x *N* array or matrix. The distances between successive rows is computed. Distance is the standard Euclidean distance.

matplotlib.mlab.**donothing_callback**(**args*) `matplotlib.mlab.entropy(y, bins)`

Return the entropy of the data in *y* in units of nat.

$$-\sum p_i \ln(p_i) \tag{62.2}$$

where p_i is the probability of observing *y* in the i^{th} bin of *bins*. *bins* can be a number of bins or a range of bins; see `numpy.histogram()`.

Compare *S* with analytic calculation for a Gaussian:

```
x = mu + sigma * randn(200000)
Sanalytic = 0.5 * ( 1.0 + log(2*pi*sigma**2.0) )
```

`matplotlib.mlab.exp_safe(x)`

Compute exponentials which safely underflow to zero.

Slow, but convenient to use. Note that numpy provides proper floating point exception handling with access to the underlying hardware.

`matplotlib.mlab.fftsurr(x, detrend=<function detrend_none>, window=<function window_none>)`

Compute an FFT phase randomized surrogate of *x*.

`matplotlib.mlab.find(condition)`

Return the indices where ravel(condition) is true

`matplotlib.mlab.frange([start], stop[, step, keywords])` → array of floats

Return a numpy ndarray containing a progression of floats. Similar to `numpy.arange()`, but defaults to a closed interval.

`frange(x0, x1)` returns `[x0, x0+1, x0+2, ..., x1]`; *start* defaults to 0, and the endpoint *is included*. This behavior is different from that of `range()` and `numpy.arange()`. This is deliberate, since `frange()` will probably be more useful for generating lists of points for function evaluation, and endpoints are often desired in this use. The usual behavior of `range()` can be obtained by setting the keyword *closed* = 0, in this case, `frange()` basically becomes :func:numpy.arange‘.

When *step* is given, it specifies the increment (or decrement). All arguments can be floating point numbers.

`frange(x0,x1,d)` returns `[x0,x0+d,x0+2d,...,xfin]` where *xfin* <= *x1*.

`frange()` can also be called with the keyword *npts*. This sets the number of points the list should contain (and overrides the value *step* might have been given). `numpy.arange()` doesn't offer this option.

Examples:

```
>>> frange(3)
array([ 0.,  1.,  2.,  3.])
>>> frange(3,closed=0)
array([ 0.,  1.,  2.])
>>> frange(1,6,2)
array([1, 3, 5])   or 1,3,5,7, depending on floating point vagueries
>>> frange(1,6.5,npts=5)
array([ 1.   ,  2.375,  3.75 ,  5.125,  6.5  ])
```

`matplotlib.mlab.get_formatd`(*r*, *formatd=None*)
build a formatd guaranteed to have a key for every dtype name

`matplotlib.mlab.get_sparse_matrix`(*M*, *N*, *frac=0.1*)
Return a *M* x *N* sparse matrix with *frac* elements randomly filled.

`matplotlib.mlab.get_xyz_where`(*Z*, *Cond*)
Z and *Cond* are *M* x *N* matrices. *Z* are data and *Cond* is a boolean matrix where some condition is satisfied. Return value is (*x*, *y*, *z*) where *x* and *y* are the indices into *Z* and *z* are the values of *Z* at those indices. *x*, *y*, and *z* are 1D arrays.

`matplotlib.mlab.griddata`(*x*, *y*, *z*, *xi*, *yi*, *interp='nn'*)
Interpolates from a nonuniformly spaced grid to some other grid.

Fits a surface of the form z = f(`x`, `y`) to the data in the (usually) nonuniformly spaced vectors (`x`, `y`, `z`), then interpolates this surface at the points specified by (`xi`, `yi`) to produce `zi`.

Parameters x, y, z : 1d array_like
Coordinates of grid points to interpolate from.

xi, yi : 1d or 2d array_like
Coordinates of grid points to interpolate to.

interp : string key from {'nn', 'linear'}
Interpolation algorithm, either 'nn' for natural neighbor, or 'linear' for linear interpolation.

Returns 2d float array
Array of values interpolated at (`xi`, `yi`) points. Array will be masked is any of (`xi`, `yi`) are outside the convex hull of (`x`, `y`).

Notes

If `interp` is 'nn' (the default), uses natural neighbor interpolation based on Delaunay triangulation. This option is only available if the mpl_toolkits.natgrid module is installed. This can be downloaded from https://github.com/matplotlib/natgrid. The (`xi`, `yi`) grid must be regular and monotonically increasing in this case.

If `interp` is 'linear', linear interpolation is used via matplotlib.tri.LinearTriInterpolator.

Instead of using *`griddata`*, more flexible functionality and other interpolation options are available using a matplotlib.tri.Triangulation and a matplotlib.tri.TriInterpolator.

`matplotlib.mlab.identity`(*n*, *rank=2*, *dtype='l'*, *typecode=None*)
Returns the identity matrix of shape (*n*, *n*, ..., *n*) (rank *r*).

For ranks higher than 2, this object is simply a multi-index Kronecker delta:

```
                        /  1  if i0=i1=...=iR,
id[i0,i1,...,iR] = -|
                        \  0  otherwise.
```

Optionally a *dtype* (or typecode) may be given (it defaults to 'l').

Since rank defaults to 2, this function behaves in the default case (when only *n* is given) like `numpy.identity(n)` – but surprisingly, it is much faster.

matplotlib.mlab.**inside_poly**(*points*, *verts*)

points is a sequence of *x*, *y* points. *verts* is a sequence of *x*, *y* vertices of a polygon.

Return value is a sequence of indices into points for the points that are inside the polygon.

matplotlib.mlab.**is_closed_polygon**(*X*)

Tests whether first and last object in a sequence are the same. These are presumably coordinates on a polygonal curve, in which case this function tests if that curve is closed.

matplotlib.mlab.**ispower2**(*n*)

Returns the log base 2 of *n* if *n* is a power of 2, zero otherwise.

Note the potential ambiguity if *n* == 1: 2**0 == 1, interpret accordingly.

matplotlib.mlab.**isvector**(*X*)

Like the MATLAB function with the same name, returns *True* if the supplied numpy array or matrix *X* looks like a vector, meaning it has a one non-singleton axis (i.e., it can have multiple axes, but all must have length 1, except for one of them).

If you just want to see if the array has 1 axis, use X.ndim == 1.

matplotlib.mlab.**l1norm**(*a*)

Return the *l1* norm of *a*, flattened out.

Implemented as a separate function (not a call to `norm()` for speed).

matplotlib.mlab.**l2norm**(*a*)

Return the *l2* norm of *a*, flattened out.

Implemented as a separate function (not a call to `norm()` for speed).

matplotlib.mlab.**less_simple_linear_interpolation**(*x*, *y*, *xi*, *extrap=False*)

This function provides simple (but somewhat less so than `cbook.simple_linear_interpolation()`) linear interpolation. `simple_linear_interpolation()` will give a list of point between a start and an end, while this does true linear interpolation at an arbitrary set of points.

This is very inefficient linear interpolation meant to be used only for a small number of points in relatively non-intensive use cases. For real linear interpolation, use scipy.

matplotlib.mlab.**log2**(*x*, *ln2=0.6931471805599453*)

Return the log(*x*) in base 2.

This is a _slow_ function but which is guaranteed to return the correct integer value if the input is an integer exact power of 2.

matplotlib.mlab.**logspace**(*xmin*, *xmax*, *N*)

Return N values logarithmically spaced between xmin and xmax.

Call signature:

```
logspace(xmin, xmax, N)
```

matplotlib.mlab.**longest_contiguous_ones**(*x*)

Return the indices of the longest stretch of contiguous ones in *x*, assuming *x* is a vector of zeros and ones. If there are two equally long stretches, pick the first.

`matplotlib.mlab.longest_ones(x)`
alias for longest_contiguous_ones

`matplotlib.mlab.magnitude_spectrum(x, Fs=None, window=None, pad_to=None, sides=None)`
Compute the magnitude (absolute value) of the frequency spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Returns the tuple (*spectrum*, *freqs*):

***spectrum*: 1-D array** The values for the magnitude spectrum (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

See also:

`psd()` `psd()` returns the power spectral density.

`complex_spectrum()` This function returns the absolute value of `complex_spectrum()`.

`angle_spectrum()` `angle_spectrum()` returns the angles of the corresponding frequencies.

`phase_spectrum()` `phase_spectrum()` returns the phase (unwrapped angle) of the corresponding frequencies.

`specgram()` `specgram()` can return the magnitude spectrum of segments within the signal.

`matplotlib.mlab.movavg(x, n)`
Compute the len(*n*) moving average of *x*.

`matplotlib.mlab.norm_flat(a, p=2)`
norm(a,p=2) -> l-p norm of a.flat

Return the l-p norm of *a*, considered as a flat array. This is NOT a true matrix norm, since arrays of arbitrary rank are always flattened.

p can be a number or the string 'Infinity' to get the L-infinity norm.

`matplotlib.mlab.normpdf(x, *args)`

Return the normal pdf evaluated at *x*; args provides *mu*, *sigma*

matplotlib.mlab.**offset_line**(*y*, *yerr*)

Offsets an array *y* by +/- an error and returns a tuple (y - err, y + err).

The error term can be:

- A scalar. In this case, the returned tuple is obvious.
- A vector of the same length as *y*. The quantities y +/- err are computed component-wise.
- A tuple of length 2. In this case, yerr[0] is the error below *y* and yerr[1] is error above *y*. For example:

```
from pylab import *
x = linspace(0, 2*pi, num=100, endpoint=True)
y = sin(x)
y_minus, y_plus = mlab.offset_line(y, 0.1)
plot(x, y)
fill_between(x, ym, y2=yp)
show()
```

matplotlib.mlab.**path_length**(*X*)

Computes the distance travelled along a polygonal curve in *N* dimensions.

Where *X* is an *M* x *N* array or matrix. Returns an array of length *M* consisting of the distance along the curve at each point (i.e., the rows of *X*).

matplotlib.mlab.**phase_spectrum**(*x*, *Fs=None*, *window=None*, *pad_to=None*, *sides=None*)

Compute the phase of the frequency spectrum (unwrapped angle spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

Returns the tuple (*spectrum*, *freqs*):

***spectrum*: 1-D array** The values for the phase spectrum in radians (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

See also:

`complex_spectrum()` This function returns the angle value of `complex_spectrum()`.

`magnitude_spectrum()` `magnitude_spectrum()` returns the magnitudes of the corresponding frequencies.

`angle_spectrum()` `angle_spectrum()` returns the wrapped version of this function.

`specgram()` `specgram()` can return the phase spectrum of segments within the signal.

matplotlib.mlab.**poly_below**(*xmin*, *xs*, *ys*)

Given a sequence of *xs* and *ys*, return the vertices of a polygon that has a horizontal base at *xmin* and an upper bound at the *ys*. *xmin* is a scalar.

Intended for use with `matplotlib.axes.Axes.fill()`, e.g.,:

```
xv, yv = poly_below(0, x, y)
ax.fill(xv, yv)
```

matplotlib.mlab.**poly_between**(*x*, *ylower*, *yupper*)

Given a sequence of *x*, *ylower* and *yupper*, return the polygon that fills the regions between them. *ylower* or *yupper* can be scalar or iterable. If they are iterable, they must be equal in length to *x*.

Return value is *x*, *y* arrays for use with `matplotlib.axes.Axes.fill()`.

matplotlib.mlab.**prctile**(*x*, *p=(0.0, 25.0, 50.0, 75.0, 100.0)*)

Return the percentiles of *x*. *p* can either be a sequence of percentile values or a scalar. If *p* is a sequence, the ith element of the return sequence is the *p*(i)-th percentile of *x*. If *p* is a scalar, the largest value of *x* less than or equal to the *p* percentage point in the sequence is returned.

matplotlib.mlab.**prctile_rank**(*x*, *p*)

Return the rank for each element in *x*, return the rank 0..len(*p*). e.g., if *p* = (25, 50, 75), the return value will be a len(*x*) array with values in [0,1,2,3] where 0 indicates the value is less than the 25th percentile, 1 indicates the value is >= the 25th and < 50th percentile, ... and 3 indicates the value is above the 75th percentile cutoff.

p is either an array of percentiles in [0..100] or a scalar which indicates how many quantiles of data you want ranked.

matplotlib.mlab.**psd**(*x*, *NFFT=None*, *Fs=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*)

Compute the power spectral density.

Call signature:

```
psd(x, NFFT=256, Fs=2, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None)
```

The power spectral density P_{xx} by Welch's average periodogram method. The vector *x* is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The $|\mathrm{fft}(i)|^2$ of each segment *i* are averaged to compute P_{xx}.

If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or** callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean** Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

noverlap: integer The number of points of overlap between segments. The default value is 0 (no overlap).

Returns the tuple (*Pxx*, *freqs*).

***Pxx*: 1-D array** The values for the power spectrum `P_{xx}` (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *Pxx*

Refs:

Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

See also:

`specgram()` `specgram()` differs in the default overlap; in not returning the mean of the segment periodograms; and in returning the times of the segments.

`magnitude_spectrum()` `magnitude_spectrum()` returns the magnitude spectrum.

`csd()` `csd()` returns the spectral density between two signals.

`matplotlib.mlab.quad2cubic`(*q0x*, *q0y*, *q1x*, *q1y*, *q2x*, *q2y*)
Converts a quadratic Bezier curve to a cubic approximation.

The inputs are the *x* and *y* coordinates of the three control points of a quadratic curve, and the output is a tuple of *x* and *y* coordinates of the four control points of the cubic curve.

`matplotlib.mlab.rec2csv`(*r*, *fname*, *delimiter=', '*, *formatd=None*, *missing=''*, *missingd=None*, *withheader=True*)
Save the data from numpy recarray *r* into a comma-/space-/tab-delimited file. The record array dtype names will be used for column headers.

***fname*: can be a filename or a file handle. Support for gzipped** files is automatic, if the filename ends in '.gz'

***withheader*: if withheader is False, do not write the attribute** names in the first row

for formatd type FormatFloat, we override the precision to store full precision floats in the CSV file

See also:

`csv2rec()` For information about *missing* and *missingd*, which can be used to fill in masked values into your CSV file.

`matplotlib.mlab.rec2txt`(*r*, *header=None*, *padding=3*, *precision=3*, *fields=None*)
Returns a textual representation of a record array.

r: numpy recarray

header: list of column headers

padding: space between each column

***precision*: number of decimal places to use for floats.** Set to an integer to apply to all floats. Set to a list of integers to apply precision individually. Precision for non-floats is simply ignored.

fields : if not None, a list of field names to print. fields can be a list of strings like ['field1', 'field2'] or a single comma separated string like 'field1,field2'

Example:

```
precision=[0,2,3]
```

Output:

```
ID    Price   Return
ABC   12.54    0.234
XYZ    6.32   -0.076
```

`matplotlib.mlab.rec_append_fields`(*rec*, *names*, *arrs*, *dtypes=None*)
Return a new record array with field names populated with data from arrays in *arrs*. If appending a single field, then *names*, *arrs* and *dtypes* do not have to be lists. They can just be the values themselves.

`matplotlib.mlab.rec_drop_fields`(*rec*, *names*)
Return a new numpy record array with fields in *names* dropped.

`matplotlib.mlab.rec_groupby`(*r*, *groupby*, *stats*)
r is a numpy record array

groupby is a sequence of record array attribute names that together form the grouping key. e.g., ('date', 'productcode')

stats is a sequence of (*attr*, *func*, *outname*) tuples which will call `x = func(attr)` and assign *x* to the record array output with attribute *outname*. For example:

```
stats = ( ('sales', len, 'numsales'), ('sales', np.mean, 'avgsale') )
```

Return record array has *dtype* names for each attribute name in the *groupby* argument, with the associated group values, and for each outname name in the *stats* argument, with the associated stat summary output.

`matplotlib.mlab.`**`rec_join`**(*key*, *r1*, *r2*, *jointype='inner'*, *defaults=None*, *r1postfix='1'*, *r2postfix='2'*)

Join record arrays *r1* and *r2* on *key*; *key* is a tuple of field names – if *key* is a string it is assumed to be a single attribute name. If *r1* and *r2* have equal values on all the keys in the *key* tuple, then their fields will be merged into a new record array containing the intersection of the fields of *r1* and *r2*.

r1 (also *r2*) must not have any duplicate keys.

The *jointype* keyword can be 'inner', 'outer', 'leftouter'. To do a rightouter join just reverse *r1* and *r2*.

The *defaults* keyword is a dictionary filled with `{column_name:default_value}` pairs.

The keywords *r1postfix* and *r2postfix* are postfixed to column names (other than keys) that are both in *r1* and *r2*.

`matplotlib.mlab.`**`rec_keep_fields`**(*rec*, *names*)

Return a new numpy record array with only fields listed in names

`matplotlib.mlab.`**`rec_summarize`**(*r*, *summaryfuncs*)

r is a numpy record array

summaryfuncs is a list of (*attr*, *func*, *outname*) tuples which will apply *func* to the array *r**[attr] and assign the output to a new attribute name *outname*. The returned record array is identical to *r*, with extra arrays for each element in *summaryfuncs*.

`matplotlib.mlab.`**`recs_join`**(*key*, *name*, *recs*, *jointype='outer'*, *missing=0.0*, *postfixes=None*)

Join a sequence of record arrays on single column key.

This function only joins a single column of the multiple record arrays
key is the column name that acts as a key
name is the name of the column that we want to join
recs is a list of record arrays to join
jointype is a string 'inner' or 'outer'
missing is what any missing field is replaced by
postfixes if not None, a len recs sequence of postfixes
returns a record array with columns [rowkey, name0, name1, ... namen-1]. or if postfixes [PF0, PF1, ..., PFN-1] are supplied, [rowkey, namePF0, namePF1, ... namePFN-1].

Example:

```
r = recs_join("date", "close", recs=[r0, r1], missing=0.)
```

matplotlib.mlab.**rk4**(*derivs*, *y0*, *t*)

Integrate 1D or ND system of ODEs using 4-th order Runge-Kutta. This is a toy implementation which may be useful if you find yourself stranded on a system w/o scipy. Otherwise use `scipy.integrate()`.

y0 initial state vector

t sample times

derivs returns the derivative of the system and has the signature `dy = derivs(yi, ti)`

Example 1

```
## 2D system

def derivs6(x,t):
    d1 =  x[0] + 2*x[1]
    d2 =  -3*x[0] + 4*x[1]
    return (d1, d2)
dt = 0.0005
t = arange(0.0, 2.0, dt)
y0 = (1,2)
yout = rk4(derivs6, y0, t)
```

Example 2:

```
## 1D system
alpha = 2
def derivs(x,t):
    return -alpha*x + exp(-t)

y0 = 1
yout = rk4(derivs, y0, t)
```

If you have access to scipy, you should probably be using the scipy.integrate tools rather than this function.

matplotlib.mlab.**rms_flat**(*a*)

Return the root mean square of all the elements of *a*, flattened out.

matplotlib.mlab.**safe_isinf**(*x*)

`numpy.isinf()` for arbitrary types

matplotlib.mlab.**safe_isnan**(*x*)

`numpy.isnan()` for arbitrary types

matplotlib.mlab.**segments_intersect**(*s1*, *s2*)

Return *True* if *s1* and *s2* intersect. *s1* and *s2* are defined as:

```
s1: (x1, y1), (x2, y2)
s2: (x3, y3), (x4, y4)
```

matplotlib.mlab.**slopes**(*x*, *y*)

`slopes()` calculates the slope *y*‘(*x*)

The slope is estimated using the slope obtained from that of a parabola through any three consecutive points.

This method should be superior to that described in the appendix of A CONSISTENTLY WELL BEHAVED METHOD OF INTERPOLATION by Russel W. Stineman (Creative Computing July 1980) in at least one aspect:

> Circles for interpolation demand a known aspect ratio between *x*- and *y*-values. For many functions, however, the abscissa are given in different dimensions, so an aspect ratio is completely arbitrary.

The parabola method gives very similar results to the circle method for most regular cases but behaves much better in special cases.

Norbert Nemec, Institute of Theoretical Physics, University or Regensburg, April 2006 Norbert.Nemec at physik.uni-regensburg.de

(inspired by a original implementation by Halldor Bjornsson, Icelandic Meteorological Office, March 2006 halldor at vedur.is)

`matplotlib.mlab.`**`specgram`**(*x*, *NFFT=None*, *Fs=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *mode=None*)

Compute a spectrogram.

Call signature:

```
specgram(x, NFFT=256, Fs=2,detrend=mlab.detrend_none,
        window=mlab.window_hanning, noverlap=128,
        cmap=None, xextent=None, pad_to=None, sides='default',
        scale_by_freq=None, mode='default')
```

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or**
callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***mode*: ['default' | 'psd' | 'complex' | 'magnitude'**
'angle' | 'phase']

What sort of spectrum to use. Default is 'psd'. which takes the power spectral density. 'complex' returns the complex-valued frequency spectrum. 'magnitude' returns the magnitude spectrum. 'angle' returns the phase spectrum without unwrapping. 'phase' returns the phase spectrum with unwrapping.

***noverlap*: integer** The number of points of overlap between blocks. The default value is 128.

Returns the tuple (*spectrum*, *freqs*, *t*):

***spectrum*: 2-D array** columns are the periodograms of successive segments

***freqs*: 1-D array** The frequencies corresponding to the rows in *spectrum*

***t*: 1-D array** The times corresponding to midpoints of segments (i.e the columns in *spectrum*).

Note: *detrend* and *scale_by_freq* only apply when *mode* is set to 'psd'

See also:

`psd()` `psd()` differs in the default overlap; in returning the mean of the segment periodograms; and in not returning times.

`complex_spectrum()` A single spectrum, similar to having a single segment when *mode* is 'complex'.

`magnitude_spectrum()` A single spectrum, similar to having a single segment when *mode* is 'magnitude'.

`angle_spectrum()` A single spectrum, similar to having a single segment when *mode* is 'angle'.

`phase_spectrum()` A single spectrum, similar to having a single segment when *mode* is 'phase'.

`matplotlib.mlab.`**`stineman_interp`**(*xi*, *x*, *y*, *yp=None*)

Given data vectors *x* and *y*, the slope vector *yp* and a new abscissa vector *xi*, the function `stineman_interp()` uses Stineman interpolation to calculate a vector *yi* corresponding to *xi*.

Here's an example that generates a coarse sine curve, then interpolates over a finer abscissa:

```
x = linspace(0,2*pi,20);  y = sin(x); yp = cos(x)
xi = linspace(0,2*pi,40);
yi = stineman_interp(xi,x,y,yp);
plot(x,y,'o',xi,yi)
```

The interpolation method is described in the article A CONSISTENTLY WELL BEHAVED METHOD OF INTERPOLATION by Russell W. Stineman. The article appeared in the July 1980 issue of Creative Computing with a note from the editor stating that while they were:

> not an academic journal but once in a while something serious and original comes in adding that this was "apparently a real solution" to a well known problem.

For *yp* = *None*, the routine automatically determines the slopes using the `slopes()` routine.

x is assumed to be sorted in increasing order.

For values `xi[j] < x[0]` or `xi[j] > x[-1]`, the routine tries an extrapolation. The relevance of the data obtained from this, of course, is questionable...

Original implementation by Halldor Bjornsson, Icelandic Meteorolocial Office, March 2006 halldor at vedur.is

Completely reworked and optimized for Python by Norbert Nemec, Institute of Theoretical Physics, University or Regensburg, April 2006 Norbert.Nemec at physik.uni-regensburg.de

`matplotlib.mlab.`**`stride_repeat`**`(`*x*, *n*, *axis=0*`)`

Repeat the values in an array in a memory-efficient manner. Array x is stacked vertically n times.

> **Warning:** It is not safe to write to the output array. Multiple elements may point to the same piece of memory, so modifying one value may change others.

Call signature:

```
  stride_repeat(x, n, axis=0)

*x*: 1D array or sequence
  Array or sequence containing the data.

*n*: integer
  The number of time to repeat the array.

*axis*: integer
  The axis along which the data will run.
```

Refs: stackoverflaw: Repeat NumPy array without replicating data?

`matplotlib.mlab.`**`stride_windows`**`(`*x*, *n*, *noverlap=None*, *axis=0*`)`

Get all windows of x with length n as a single array, using strides to avoid data duplication.

> **Warning:** It is not safe to write to the output array. Multiple elements may point to the same piece of memory, so modifying one value may change others.

Call signature:

```
  stride_windows(x, n, noverlap=0)

*x*: 1D array or sequence
  Array or sequence containing the data.

*n*: integer
  The number of data points in each window.

*noverlap*: integer
  The overlap between adjacent windows.
  Default is 0 (no overlap)

*axis*: integer
  The axis along which the windows will run.
```

Refs: stackoverflaw: Rolling window for 1D arrays in Numpy? stackoverflaw: Using strides for an efficient moving average filter

`matplotlib.mlab.vector_lengths`(*X*, *P=2.0*, *axis=None*)

Finds the length of a set of vectors in *n* dimensions. This is like the `numpy.norm()` function for vectors, but has the ability to work over a particular axis of the supplied array or matrix.

Computes `(sum((x_i)^P))^(1/P)` for each `{x_i}` being the elements of *X* along the given axis. If *axis* is *None*, compute over all elements of *X*.

`matplotlib.mlab.window_hanning`(*x*)

Return x times the hanning window of len(x).

Call signature:

```
window_hanning(x)
```

See also:

`window_none()` *`window_none()`* is another window algorithm.

`matplotlib.mlab.window_none`(*x*)

No window function; simply return x.

Call signature:

```
window_none(x)
```

See also:

`window_hanning()` *`window_hanning()`* is another window algorithm.

CHAPTER

SIXTYTHREE

OFFSETBOX

63.1 matplotlib.offsetbox

The OffsetBox is a simple container artist. The child artist are meant to be drawn at a relative position to its parent. The [VH]Packer, DrawingArea and TextArea are derived from the OffsetBox.

The [VH]Packer automatically adjust the relative postisions of their children, which should be instances of the OffsetBox. This is used to align similar artists together, e.g., in legend.

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed. The TextArea is contains a single Text instance. The width and height of the TextArea instance is the width and height of the its child text.

class `matplotlib.offsetbox.`**`AnchoredOffsetbox`**(*loc*, *pad=0.4*, *borderpad=0.5*, *child=None*, *prop=None*, *frameon=True*, *bbox_to_anchor=None*, *bbox_transform=None*, ***kwargs*)

Bases: *matplotlib.offsetbox.OffsetBox*

An offset box placed according to the legend location loc. AnchoredOffsetbox has a single child. When multiple children is needed, use other OffsetBox class to enclose them. By default, the offset box is anchored against its parent axes. You may explicitly specify the bbox_to_anchor.

loc is a string or an integer specifying the legend location. The valid location codes are:

```
'upper right'  : 1,
'upper left'   : 2,
'lower left'   : 3,
'lower right'  : 4,
'right'        : 5,
'center left'  : 6,
'center right' : 7,
'lower center' : 8,
'upper center' : 9,
'center'       : 10,
```

pad [pad around the child for drawing a frame. given in] fraction of fontsize.

borderpad : pad between offsetbox frame and the bbox_to_anchor,

child : OffsetBox instance that will be anchored.

prop : font property. This is only used as a reference for paddings.

frameon : draw a frame box if True.

bbox_to_anchor : bbox to anchor. Use self.axes.bbox if None.

bbox_transform : with which the bbox_to_anchor will be transformed.

draw(*renderer*)
draw the artist

get_bbox_to_anchor()
return the bbox that the legend will be anchored

get_child()
return the child

get_children()
return the list of children

get_extent(*renderer*)
return the extent of the artist. The extent of the child added with the pad is returned

get_window_extent(*renderer*)
get the bounding box in display space.

set_bbox_to_anchor(*bbox*, *transform=None*)
set the bbox that the child will be anchored.

bbox can be a Bbox instance, a list of [left, bottom, width, height], or a list of [left, bottom] where the width and height will be assumed to be zero. The bbox will be transformed to display coordinate by the given transform.

set_child(*child*)
set the child to be anchored

update_frame(*bbox*, *fontsize=None*)

zorder = 5

class matplotlib.offsetbox.**AnchoredText**(*s*, *loc*, *pad=0.4*, *borderpad=0.5*, *prop=None*, ***kwargs*)
Bases: *matplotlib.offsetbox.AnchoredOffsetbox*

AnchoredOffsetbox with Text.

Parameters s : string
Text.

loc : str
Location code.

pad : float, optional
Pad between the text and the frame as fraction of the font size.

borderpad : float, optional
Pad between the frame and the axes (or *bbox_to_anchor*).

prop : *matplotlib.font_manager.FontProperties*

Font properties.

Notes

Other keyword parameters of `AnchoredOffsetbox` are also allowed.

class `matplotlib.offsetbox.AnnotationBbox`(*offsetbox*, *xy*, *xybox=None*, *xycoords='data'*, *boxcoords=None*, *frameon=True*, *pad=0.4*, *annotation_clip=None*, *box_alignment=(0.5, 0.5)*, *bboxprops=None*, *arrowprops=None*, *fontsize=None*, ***kwargs*)

Bases: `matplotlib.artist.Artist`, `matplotlib.text._AnnotationBase`

Annotation-like class, but with offsetbox instead of Text.

offsetbox : OffsetBox instance

xycoords [same as Annotation but can be a tuple of two] strings which are interpreted as x and y coordinates.

boxcoords [similar to textcoords as Annotation but can be a] tuple of two strings which are interpreted as x and y coordinates.

box_alignment [a tuple of two floats for a vertical and] horizontal alignment of the offset box w.r.t. the *boxcoords*. The lower-left corner is (0.0) and upper-right corner is (1.1).

other parameters are identical to that of Annotation.

anncoords

contains(*event*)

draw(*renderer*)
Draw the `Annotation` object to the given *renderer*.

get_children()

get_fontsize(*s=None*)
return fontsize in points

set_figure(*fig*)

set_fontsize(*s=None*)
set fontsize in points

update_positions(*renderer*)
Update the pixel positions of the annotated point and the text.

xyann

zorder = 3

class `matplotlib.offsetbox.`**`AuxTransformBox`**(*aux_transform*)

Bases: `matplotlib.offsetbox.OffsetBox`

Offset Box with the aux_transform . Its children will be transformed with the aux_transform first then will be offseted. The absolute coordinate of the aux_transform is meaning as it will be automatically adjust so that the left-lower corner of the bounding box of children will be set to (0,0) before the offset transform.

It is similar to drawing area, except that the extent of the box is not predetermined but calculated from the window extent of its children. Furthermore, the extent of the children will be calculated in the transformed coordinate.

`add_artist`(*a*)

Add any `Artist` to the container box

`draw`(*renderer*)

Draw the children

`get_extent`(*renderer*)

`get_offset`()

return offset of the container.

`get_transform`()

Return the `Transform` applied to the children

`get_window_extent`(*renderer*)

get the bounding box in display space.

`set_offset`(*xy*)

set offset of the container.

Accept : tuple of x,y coordinate in disokay units.

`set_transform`(*t*)

set_transform is ignored.

class `matplotlib.offsetbox.`**`DraggableAnnotation`**(*annotation*, *use_blit=False*)

Bases: `matplotlib.offsetbox.DraggableBase`

`finalize_offset`()

`save_offset`()

`update_offset`(*dx*, *dy*)

class `matplotlib.offsetbox.`**`DraggableBase`**(*ref_artist*, *use_blit=False*)

Bases: `object`

helper code for a draggable artist (legend, offsetbox) The derived class must override following two method.

def saveoffset(self): pass

def update_offset(self, dx, dy): pass

saveoffset is called when the object is picked for dragging and it is meant to save reference position of the artist.

***update_offset* is called during the dragging. dx and dy is the pixel** offset from the point where the mouse drag started.

Optionally you may override following two methods.

def artist_picker(self, artist, evt): return self.ref_artist.contains(evt)

def finalize_offset(self): pass

***artist_picker* is a picker method that will be** used. *finalize_offset* is called when the mouse is released. In current implementaion of DraggableLegend and DraggableAnnotation, *update_offset* places the artists simply in display coordinates. And *finalize_offset* recalculate their position in the normalized axes coordinate and set a relavant attribute.

artist_picker(*artist*, *evt*)

disconnect()

disconnect the callbacks

finalize_offset()

on_motion(*evt*)

on_motion_blit(*evt*)

on_pick(*evt*)

on_release(*event*)

save_offset()

update_offset(*dx*, *dy*)

class matplotlib.offsetbox.**DraggableOffsetBox**(*ref_artist*, *offsetbox*, *use_blit=False*)

Bases: *matplotlib.offsetbox.DraggableBase*

get_loc_in_canvas()

save_offset()

update_offset(*dx*, *dy*)

class `matplotlib.offsetbox.DrawingArea`(*width*, *height*, *xdescent=0.0*, *ydescent=0.0*, *clip=False*)

Bases: `matplotlib.offsetbox.OffsetBox`

The DrawingArea can contain any Artist as a child. The DrawingArea has a fixed width and height. The position of children relative to the parent is fixed. The children can be clipped at the boundaries of the parent.

width, *height* : width and height of the container box. *xdescent*, *ydescent* : descent of the box in x- and y-direction. *clip* : Whether to clip the children

add_artist(*a*)
Add any `Artist` to the container box

clip_children
If the children of this DrawingArea should be clipped by DrawingArea bounding box.

draw(*renderer*)
Draw the children

get_extent(*renderer*)
Return with, height, xdescent, ydescent of box

get_offset()
return offset of the container.

get_transform()
Return the `Transform` applied to the children

get_window_extent(*renderer*)
get the bounding box in display space.

set_offset(*xy*)
set offset of the container.

Accept : tuple of x,y cooridnate in disokay units.

set_transform(*t*)
set_transform is ignored.

class `matplotlib.offsetbox.HPacker`(*pad=None*, *sep=None*, *width=None*, *height=None*, *align='baseline'*, *mode='fixed'*, *children=None*)

Bases: `matplotlib.offsetbox.PackerBase`

The HPacker has its children packed horizontally. It automatically adjusts the relative positions of children at draw time.

Parameters pad : float, optional
Boundary pad.

sep : float, optional
Spacing between items.

width : float, optional

height : float, optional
Width and height of the container box, calculated if `None`.

align : str

Alignment of boxes.

mode : str

Packing mode.

Notes

pad and *sep* need to given in points and will be scale with the renderer dpi, while *width* and *height* need to be in pixels.

get_extent_offsets(*renderer*)
update offset of children and return the extents of the box

class matplotlib.offsetbox.**OffsetBox**(**args*, ***kwargs*)
Bases: `matplotlib.artist.Artist`

The OffsetBox is a simple container artist. The child artist are meant to be drawn at a relative position to its parent.

axes
The `Axes` instance the artist resides in, or *None*.

contains(*mouseevent*)

draw(*renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

get_children()
Return a list of artists it contains.

get_extent(*renderer*)
Return with, height, xdescent, ydescent of box

get_extent_offsets(*renderer*)

get_offset(*width*, *height*, *xdescent*, *ydescent*, *renderer*)
Get the offset

accepts extent of the box

get_visible_children()
Return a list of visible artists it contains.

get_window_extent(*renderer*)
get the bounding box in display space.

set_figure(*fig*)
Set the figure

accepts a class:`Figure` instance

set_height(*height*)
Set the height

accepts float

set_offset(*xy*)
Set the offset

accepts x, y, tuple, or a callable object.

set_width(*width*)
Set the width

accepts float

class matplotlib.offsetbox.**OffsetImage**(*arr*, *zoom=1*, *cmap=None*, *norm=None*, *interpolation=None*, *origin=None*, *filternorm=1*, *filterrad=4.0*, *resample=False*, *dpi_cor=True*, ***kwargs*)
Bases: `matplotlib.offsetbox.OffsetBox`

draw(*renderer*)
Draw the children

get_children()

get_data()

get_extent(*renderer*)

get_offset()
return offset of the container.

get_window_extent(*renderer*)
get the bounding box in display space.

get_zoom()

set_data(*arr*)

set_zoom(*zoom*)

class matplotlib.offsetbox.**PackerBase**(*pad=None*, *sep=None*, *width=None*, *height=None*, *align=None*, *mode=None*, *children=None*)
Bases: `matplotlib.offsetbox.OffsetBox`

Parameters pad : float, optional
Boundary pad.

sep : float, optional
Spacing between items.

width : float, optional

height : float, optional
Width and height of the container box, calculated if `None`.

align : str, optional
Alignment of boxes. Can be one of `top`, `bottom`, `left`, `right`, `center` and `baseline`

mode : str, optional
Packing mode.

Notes

pad and *sep* need to given in points and will be scale with the renderer dpi, while *width* and *height* need to be in pixels.

class `matplotlib.offsetbox.`**`PaddedBox`**(*child*, *pad=None*, *draw_frame=False*, *patch_attrs=None*)
Bases: `matplotlib.offsetbox.OffsetBox`

pad : boundary pad

Note: *pad* need to given in points and will be scale with the renderer dpi, while *width* and *height* need to be in pixels.

`draw`(*renderer*)
Update the location of children if necessary and draw them to the given *renderer*.

`draw_frame`(*renderer*)

`get_extent_offsets`(*renderer*)
update offset of childrens and return the extents of the box

`update_frame`(*bbox*, *fontsize=None*)

class `matplotlib.offsetbox.`**`TextArea`**(*s*, *textprops=None*, *multilinebaseline=None*, *minimumdescent=True*)
Bases: `matplotlib.offsetbox.OffsetBox`

The TextArea is contains a single Text instance. The text is placed at (0,0) with baseline+left alignment. The width and height of the TextArea instance is the width and height of the its child text.

Parameters s : str
a string to be displayed.

textprops : `FontProperties`, optional

multilinebaseline : bool, optional
If `True`, baseline for multiline text is adjusted so that it is (approximatedly) center-aligned with singleline text.

minimumdescent : bool, optional
If `True`, the box has a minimum descent of "p".

`draw`(*renderer*)
Draw the children

`get_extent(renderer)`

`get_minimumdescent()`
get minimumdescent.

`get_multilinebaseline()`
get multilinebaseline .

`get_offset()`
return offset of the container.

`get_text()`
Returns the string representation of this area's text

`get_window_extent(renderer)`
get the bounding box in display space.

`set_minimumdescent(t)`
Set minimumdescent .

If True, extent of the single line text is adjusted so that it has minimum descent of "p"

`set_multilinebaseline(t)`
Set multilinebaseline .

If True, baseline for multiline text is adjusted so that it is (approximately) center-aligned with singleline text.

`set_offset(xy)`
set offset of the container.

Accept : tuple of x,y coordinates in display units.

`set_text(s)`
Set the text of this area as a string.

`set_transform(t)`
set_transform is ignored.

class `matplotlib.offsetbox.VPacker`(*pad=None, sep=None, width=None, height=None, align='baseline', mode='fixed', children=None*)

Bases: `matplotlib.offsetbox.PackerBase`

The VPacker has its children packed vertically. It automatically adjust the relative positions of children in the drawing time.

Parameters pad : float, optional
Boundary pad.

sep : float, optional
Spacing between items.

width : float, optional

height : float, optional
width and height of the container box, calculated if `None`.

align : str, optional
Alignment of boxes.

mode : str, optional
Packing mode.

Notes

pad and *sep* need to given in points and will be scale with the renderer dpi, while *width* and *height* need to be in pixels.

get_extent_offsets(*renderer*)
update offset of childrens and return the extents of the box

matplotlib.offsetbox.**bbox_artist**(**args*, ***kwargs*)

CHAPTER

SIXTYFOUR

PATCHES

64.1 matplotlib.patches

class matplotlib.patches.**Arc**(*xy*, *width*, *height*, *angle=0.0*, *theta1=0.0*, *theta2=360.0*, ***kwargs*)

Bases: *matplotlib.patches.Ellipse*

An elliptical arc. Because it performs various optimizations, it can not be filled.

The arc must be used in an *Axes* instance—it can not be added directly to a *Figure*—because it is optimized to only render the segments that are inside the axes bounding box with high resolution.

The following args are supported:
xy center of ellipse
width length of horizontal axis
height length of vertical axis
angle rotation in degrees (anti-clockwise)
theta1 starting angle of the arc in degrees
theta2 ending angle of the arc in degrees
If *theta1* and *theta2* are not provided, the arc will form a complete ellipse.

Valid kwargs are:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color

Continued on

Table 64.1 – continued from previous page

Property	Description
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	[’/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]
joinstyle	[’miter’ | ‘round’ | ‘bevel’]
label	string or anything printable with ‘%s’ conversion.
linestyle or ls	[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

draw(*artist*, *renderer*, **args*, ***kwargs*)

Ellipses are normally drawn using an approximation that uses eight cubic bezier splines. The error of this approximation is 1.89818e-6, according to this unverified source:

> Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.
>
> http://www.tinaja.com/glib/ellipse4.pdf

There is a use case where very large ellipses must be drawn with very high accuracy, and it is too expensive to render the entire ellipse with enough segments (either splines or line segments). Therefore, in the case where either radius of the ellipse is large enough that the error of the spline approximation will be visible (greater than one pixel offset from the ideal), a different technique is used.

In that case, only the visible parts of the ellipse are drawn, with each visible arc using a fixed number of spline segments (8). The algorithm proceeds as follows:

1. The points where the ellipse intersects the axes bounding box are located. (This is done be performing an inverse transformation on the axes bbox such that it is relative to the unit circle – this makes the intersection calculation much easier than doing rotated ellipse intersection directly).

   This uses the “line intersecting a circle” algorithm from:

   > Vince, John. Geometry for Computer Graphics: Formulae, Examples & Proofs. London: Springer-Verlag, 2005.

2. The angles of each of the intersection points are calculated.
3. Proceeding counterclockwise starting in the positive x-direction, each of the visible arc-segments between the pairs of vertices are drawn using the bezier arc approximation technique implemented in *matplotlib.path.Path.arc()*.

class `matplotlib.patches.Arrow`(*x*, *y*, *dx*, *dy*, *width=1.0*, ***kwargs*)

Bases: `matplotlib.patches.Patch`

An arrow patch.

Draws an arrow, starting at (*x*, *y*), direction and length given by (*dx*, *dy*) the width of the arrow is scaled by *width*.

Valid kwargs are:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

get_patch_transform()

get_path()

class `matplotlib.patches.ArrowStyle`

Bases: `matplotlib.patches._Style`

ArrowStyle is a container class which defines several arrowstyle classes, which is used to create an arrow path along a given path. These are mainly used with *FancyArrowPatch*.

A arrowstyle object can be either created as:

```
ArrowStyle.Fancy(head_length=.4, head_width=.4, tail_width=.4)
```

or:

```
ArrowStyle("Fancy", head_length=.4, head_width=.4, tail_width=.4)
```

or:

```
ArrowStyle("Fancy, head_length=.4, head_width=.4, tail_width=.4")
```

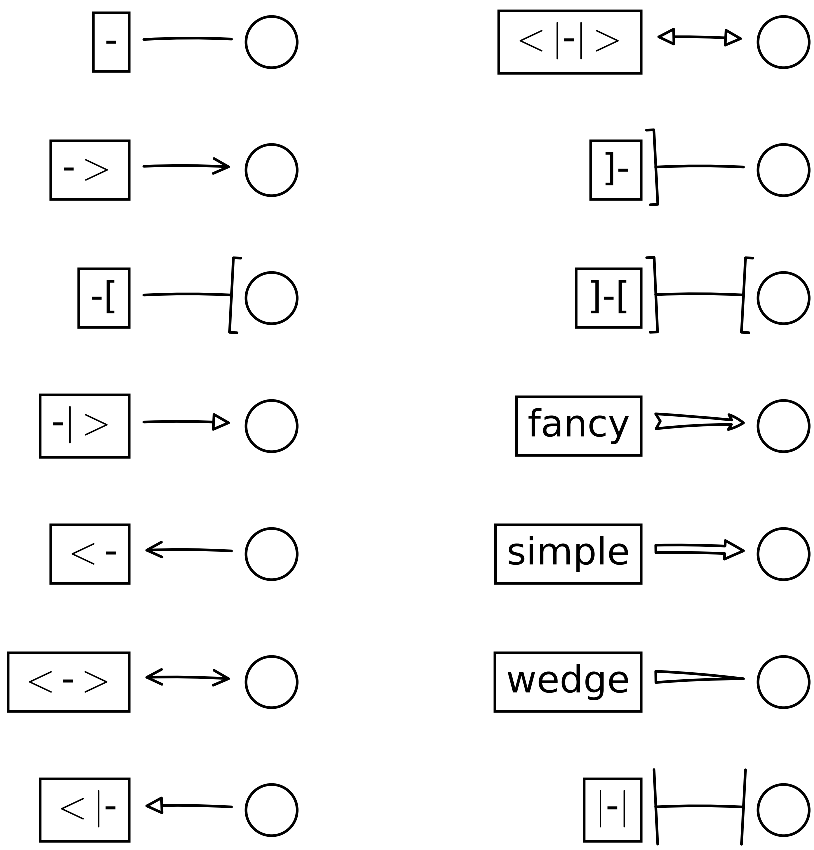
The following classes are defined

Class	Name	Attrs
Curve	-	None
CurveB	->	head_length=0.4,head_width=0.2
BracketB	-[	widthB=1.0,lengthB=0.2,angleB=None
Curve-FilledB	-|>	head_length=0.4,head_width=0.2
CurveA	<-	head_length=0.4,head_width=0.2
CurveAB	<->	head_length=0.4,head_width=0.2
Curve-FilledA	<|-	head_length=0.4,head_width=0.2
Curve-FilledAB	<|-|>	head_length=0.4,head_width=0.2
BracketA	]-	widthA=1.0,lengthA=0.2,angleA=None
BracketAB	]-[	widthA=1.0,lengthA=0.2,angleA=None,widthB=1.0,lengthB=0.2,angleB=None
Fancy	fancy	head_length=0.4,head_width=0.4,tail_width=0.4
Simple	simple	head_length=0.5,head_width=0.5,tail_width=0.2
Wedge	wedge	tail_width=0.3,shrink_factor=0.5
BarAB	|-|	widthA=1.0,angleA=None,widthB=1.0,angleB=None

An instance of any arrow style class is a callable object, whose call signature is:

```
__call__(self, path, mutation_size, linewidth, aspect_ratio=1.)
```

and it returns a tuple of a `Path` instance and a boolean value. *path* is a `Path` instance along which the arrow will be drawn. *mutation_size* and *aspect_ratio* have the same meaning as in *BoxStyle*. *linewidth* is a line width to be stroked. This is meant to be used to correct the location of the head so that it does not overshoot the destination point, but not all classes support it.



return the instance of the subclass with the given style name.

class `BarAB`(*widthA=1.0*, *angleA=None*, *widthB=1.0*, *angleB=None*)

Bases: `matplotlib.patches._Bracket`

An arrow with a bar(|) at both ends.
widthA width of the bracket
lengthA length of the bracket
angleA angle between the bracket and the line
widthB width of the bracket
lengthB length of the bracket
angleB angle between the bracket and the line

class `ArrowStyle.BracketA`(*widthA=1.0*, *lengthA=0.2*, *angleA=None*)

Bases: `matplotlib.patches._Bracket`

An arrow with a bracket(]) at its end.
widthA width of the bracket
lengthA length of the bracket
angleA angle between the bracket and the line

class `ArrowStyle.BracketAB`(*widthA=1.0*, *lengthA=0.2*, *angleA=None*, *widthB=1.0*, *lengthB=0.2*, *angleB=None*)

Bases: `matplotlib.patches._Bracket`

An arrow with a bracket(]) at both ends.
widthA width of the bracket
lengthA length of the bracket
angleA angle between the bracket and the line
widthB width of the bracket
lengthB length of the bracket
angleB angle between the bracket and the line

class `ArrowStyle.`**`BracketB`**(*widthB=1.0*, *lengthB=0.2*, *angleB=None*)
Bases: `matplotlib.patches._Bracket`

An arrow with a bracket([) at its end.
widthB width of the bracket
lengthB length of the bracket
angleB angle between the bracket and the line

class `ArrowStyle.`**`Curve`**
Bases: `matplotlib.patches._Curve`

A simple curve without any arrow head.

class `ArrowStyle.`**`CurveA`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with a head at its begin point.
head_length length of the arrow head
head_width width of the arrow head

class `ArrowStyle.`**`CurveAB`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with heads both at the begin and the end point.
head_length length of the arrow head
head_width width of the arrow head

class `ArrowStyle.`**`CurveB`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with a head at its end point.
head_length length of the arrow head
head_width width of the arrow head

class `ArrowStyle.`**`CurveFilledA`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with filled triangle head at the begin.
head_length length of the arrow head
head_width width of the arrow head

class `ArrowStyle.`**`CurveFilledAB`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with filled triangle heads both at the begin and the end point.

head_length length of the arrow head
head_width width of the arrow head

class `ArrowStyle.`**`CurveFilledB`**(*head_length=0.4*, *head_width=0.2*)
Bases: `matplotlib.patches._Curve`

An arrow with filled triangle head at the end.
head_length length of the arrow head
head_width width of the arrow head

class `ArrowStyle.`**`Fancy`**(*head_length=0.4*, *head_width=0.4*, *tail_width=0.4*)
Bases: `matplotlib.patches._Base`

A fancy arrow. Only works with a quadratic bezier curve.
head_length length of the arrow head
head_with width of the arrow head
tail_width width of the arrow tail
`transmute`(*path*, *mutation_size*, *linewidth*)

class `ArrowStyle.`**`Simple`**(*head_length=0.5*, *head_width=0.5*, *tail_width=0.2*)
Bases: `matplotlib.patches._Base`

A simple arrow. Only works with a quadratic bezier curve.
head_length length of the arrow head
head_with width of the arrow head
tail_width width of the arrow tail
`transmute`(*path*, *mutation_size*, *linewidth*)

class `ArrowStyle.`**`Wedge`**(*tail_width=0.3*, *shrink_factor=0.5*)
Bases: `matplotlib.patches._Base`

Wedge(?) shape. Only works with a quadratic bezier curve. The begin point has a width of the tail_width and the end point has a width of 0. At the middle, the width is shrink_factor*tail_width.
tail_width width of the tail
shrink_factor fraction of the arrow width at the middle point
`transmute`(*path*, *mutation_size*, *linewidth*)

class `matplotlib.patches.`**`BoxStyle`**
Bases: `matplotlib.patches._Style`

`BoxStyle` is a container class which defines several boxstyle classes, which are used for *`FancyBboxPatch`*.

A style object can be created as:

```
BoxStyle.Round(pad=0.2)
```

or:

```
BoxStyle("Round", pad=0.2)
```

or:

```
BoxStyle("Round, pad=0.2")
```

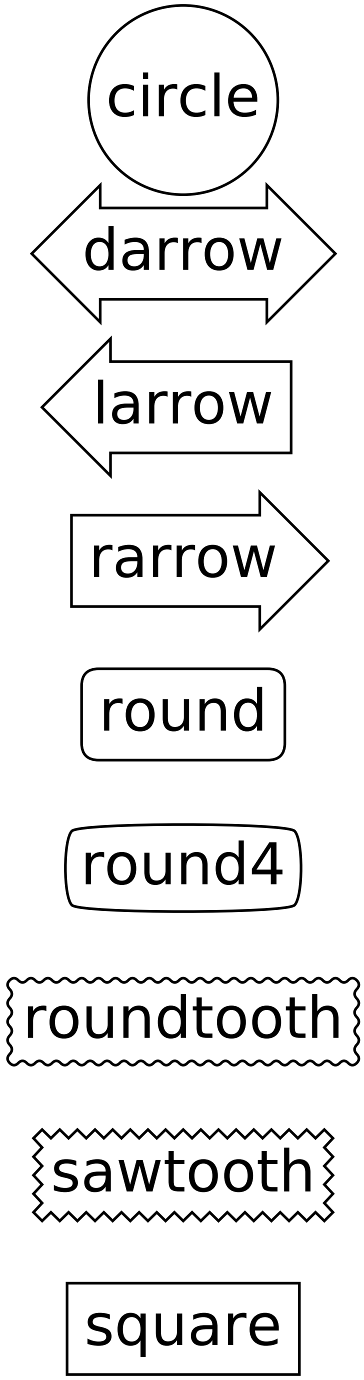
Following boxstyle classes are defined.

Class	Name	Attrs
Circle	`circle`	pad=0.3
DArrow	`darrow`	pad=0.3
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3,rounding_size=None
Round4	`round4`	pad=0.3,rounding_size=None
Roundtooth	`roundtooth`	pad=0.3,tooth_size=None
Sawtooth	`sawtooth`	pad=0.3,tooth_size=None
Square	`square`	pad=0.3

An instance of any boxstyle class is an callable object, whose call signature is:

```
__call__(self, x0, y0, width, height, mutation_size, aspect_ratio=1.)
```

and returns a `Path` instance. *x0*, *y0*, *width* and *height* specify the location and size of the box to be drawn. *mutation_scale* determines the overall size of the mutation (by which I mean the transformation of the rectangle to the fancy box). *mutation_aspect* determines the aspect-ratio of the mutation.



return the instance of the subclass with the given style name.

class `Circle`(*pad=0.3*)

Bases: `matplotlib.patches._Base`

A simple circle box.

Parameters pad : float

The amount of padding around the original box.

transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

class BoxStyle.**DArrow**(*pad=0.3*)
Bases: `matplotlib.patches._Base`

(Double) Arrow Box

transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

class BoxStyle.**LArrow**(*pad=0.3*)
Bases: `matplotlib.patches._Base`

(left) Arrow Box

transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

class BoxStyle.**RArrow**(*pad=0.3*)
Bases: `matplotlib.patches.LArrow`

(right) Arrow Box

transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

class BoxStyle.**Round**(*pad=0.3*, *rounding_size=None*)
Bases: `matplotlib.patches._Base`

A box with round corners.
pad amount of padding
rounding_size rounding radius of corners. *pad* if None
transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

class BoxStyle.**Round4**(*pad=0.3*, *rounding_size=None*)
Bases: `matplotlib.patches._Base`

Another box with round edges.
pad amount of padding
rounding_size rounding size of edges. *pad* if None
transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

class BoxStyle.**Roundtooth**(*pad=0.3*, *tooth_size=None*)
Bases: `matplotlib.patches.Sawtooth`

A rounded tooth box.
pad amount of padding
tooth_size size of the sawtooth. pad* if None
transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

class BoxStyle.**Sawtooth**(*pad=0.3*, *tooth_size=None*)
Bases: `matplotlib.patches._Base`

A sawtooth box.

pad amount of padding

tooth_size size of the sawtooth. pad* if None

transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

class BoxStyle.**Square**(*pad=0.3*)

Bases: `matplotlib.patches._Base`

A simple square box.

pad amount of padding

transmute(*x0*, *y0*, *width*, *height*, *mutation_size*)

class matplotlib.patches.**Circle**(*xy*, *radius=5*, ***kwargs*)

Bases: `matplotlib.patches.Ellipse`

A circle patch.

Create true circle at center *xy* = (*x*, *y*) with given *radius*. Unlike `CirclePolygon` which is a polygonal approximation, this uses Bézier splines and is much closer to a scale-free circle.

Valid kwargs are:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]

Continued on

Table 64.3 – continued from previous page

Property	Description
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

get_radius()
return the radius of the circle

radius
return the radius of the circle

set_radius(*radius*)
Set the radius of the circle

ACCEPTS: float

class matplotlib.patches.**CirclePolygon**(*xy*, *radius=5*, *resolution=20*, ***kwargs*)
Bases: *matplotlib.patches.RegularPolygon*

A polygon-approximation of a circle patch.

Create a circle at *xy* = (*x*, *y*) with given *radius*. This circle is approximated by a regular polygon with *resolution* sides. For a smoother circle drawn with splines, see *Circle*.

Valid kwargs are:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

Continued on

Table 64.4 – continued from previous page

Property	Description
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

class `matplotlib.patches.`**`ConnectionPatch`**(*xyA*, *xyB*, *coordsA*, *coordsB=None*, *axesA=None*, *axesB=None*, *arrowstyle='-'*, *arrow_transmuter=None*, *connectionstyle='arc3'*, *connector=None*, *patchA=None*, *patchB=None*, *shrinkA=0.0*, *shrinkB=0.0*, *mutation_scale=10.0*, *mutation_aspect=None*, *clip_on=False*, *dpi_cor=1.0*, ***kwargs*)

Bases: `matplotlib.patches.FancyArrowPatch`

A `ConnectionPatch` class is to make connecting lines between two points (possibly in different axes).

Connect point *xyA* in *coordsA* with point *xyB* in *coordsB*

Valid keys are

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for `matplotlib.patches.PathPatch`

coordsA and *coordsB* are strings that indicate the coordinates of *xyA* and *xyB*.

Property	Description
'figure points'	points from the lower left corner of the figure
'figure pixels'	pixels from the lower left corner of the figure
'figure fraction'	0,0 is lower left of figure and 1,1 is upper, right
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	0,1 is lower left of axes and 1,1 is upper right
'data'	use the coordinate system of the object being annotated (default)
'offset points'	Specify an offset (in points) from the *xy* value
'polar'	you can specify *theta*, *r* for the annotation, even in cartesian plots. Note that if you are using a polar axes, you do not need to specify polar for the coordinate system since that is the native "data" coordinate system.

draw(*renderer*)
Draw.

get_annotation_clip()
Return *annotation_clip* attribute. See `set_annotation_clip()` for the meaning of return values.

get_path_in_displaycoord()
Return the mutated path of the arrow in the display coord

set_annotation_clip(*b*)
set *annotation_clip* attribute.

- **True: the annotation will only be drawn when self.xy is inside the** axes.
- **False: the annotation will always be drawn regardless of its** position.
- None: the self.xy will be checked only if *xycoords* is "data"

class matplotlib.patches.**ConnectionStyle**
Bases: matplotlib.patches._Style

`ConnectionStyle` is a container class which defines several connectionstyle classes, which is used to create a path between two points. These are mainly used with `FancyArrowPatch`.

A connectionstyle object can be either created as:

```
ConnectionStyle.Arc3(rad=0.2)
```

or:

```
ConnectionStyle("Arc3", rad=0.2)
```

or:

```
ConnectionStyle("Arc3, rad=0.2")
```

The following classes are defined

Class	Name	Attrs
Angle	`angle`	angleA=90,angleB=0,rad=0.0
Angle3	`angle3`	angleA=90,angleB=0
Arc	`arc`	angleA=0,angleB=0,armA=None,armB=None,rad=0.0
Arc3	`arc3`	rad=0.0
Bar	`bar`	armA=0.0,armB=0.0,fraction=0.3,angle=None

An instance of any connection style class is an callable object, whose call signature is:

```
__call__(self, posA, posB,
         patchA=None, patchB=None,
         shrinkA=2., shrinkB=2.)
```

and it returns a `Path` instance. *posA* and *posB* are tuples of x,y coordinates of the two points to be connected. *patchA* (or *patchB*) is given, the returned path is clipped so that it start (or end) from the boundary of the patch. The path is further shrunk by *shrinkA* (or *shrinkB*) which is given in points.

return the instance of the subclass with the given style name.

class **`Angle`**(*angleA=90*, *angleB=0*, *rad=0.0*)

Bases: `matplotlib.patches._Base`

Creates a picewise continuous quadratic bezier path between two points. The path has a one passing-through point placed at the intersecting point of two lines which crosses the start (or end) point and has a angle of angleA (or angleB). The connecting edges are rounded with *rad*.

angleA starting angle of the path
angleB ending angle of the path
rad rounding radius of the edge
`connect`(*posA*, *posB*)

class `ConnectionStyle.`**`Angle3`**(*angleA=90*, *angleB=0*)

Bases: `matplotlib.patches._Base`

Creates a simple quadratic bezier curve between two points. The middle control points is placed at the intersecting point of two lines which crosses the start (or end) point and has a angle of angleA (or angleB).

angleA starting angle of the path
angleB ending angle of the path
`connect`(*posA*, *posB*)

class `ConnectionStyle.`**`Arc`**(*angleA=0*, *angleB=0*, *armA=None*, *armB=None*, *rad=0.0*)

Bases: `matplotlib.patches._Base`

Creates a picewise continuous quadratic bezier path between two points. The path can have two passing-through points, a point placed at the distance of armA and angle of angleA from point A, another point with respect to point B. The edges are rounded with *rad*.
angleA **:** starting angle of the path
angleB **:** ending angle of the path
armA **:** length of the starting arm
armB **:** length of the ending arm
rad **:** rounding radius of the edges
connect(*posA*, *posB*)

class `ConnectionStyle.`**`Arc3`**(*rad=0.0*)
Bases: `matplotlib.patches._Base`

Creates a simple quadratic bezier curve between two points. The curve is created so that the middle contol points (C1) is located at the same distance from the start (C0) and end points(C2) and the distance of the C1 to the line connecting C0-C2 is *rad* times the distance of C0-C2.
rad curvature of the curve.
connect(*posA*, *posB*)

class `ConnectionStyle.`**`Bar`**(*armA=0.0*, *armB=0.0*, *fraction=0.3*, *angle=None*)
Bases: `matplotlib.patches._Base`

A line with *angle* between A and B with *armA* and *armB*. One of the arms is extended so that they are connected in a right angle. The length of armA is determined by (*armA* + *fraction* x AB distance). Same for armB.

Parameters armA : float
minimum length of armA
armB : float
minimum length of armB
fraction : float
a fraction of the distance between two points that will be added to armA and armB.
angle : float or None
angle of the connecting line (if None, parallel to A and B)

connect(*posA*, *posB*)

class `matplotlib.patches.`**`Ellipse`**(*xy*, *width*, *height*, *angle=0.0*, ***kwargs*)
Bases: `matplotlib.patches.Patch`

A scale-free ellipse.
xy center of ellipse
width total length (diameter) of horizontal axis
height total length (diameter) of vertical axis
angle rotation in degrees (anti-clockwise)
Valid kwargs are:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>agg_filter</td><td>unknown</td></tr>
<tr><td>alpha</td><td>float or None</td></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or aa</td><td>[True | False] or None for default</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color</td><td>matplotlib color spec</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or ec</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td>facecolor or fc</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fill</td><td>[True | False]</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>hatch</td><td>['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']</td></tr>
<tr><td>joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> | <code>':'</code> | <code>'None'</code></td></tr>
<tr><td>linewidth or lw</td><td>float or None for default</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>transform</td><td>Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

get_patch_transform()

get_path()

Return the vertices of the rectangle

class matplotlib.patches.**FancyArrow**(*x*, *y*, *dx*, *dy*, *width=0.001*, *length_includes_head=False*, *head_width=None*, *head_length=None*, *shape='full'*, *overhang=0*, *head_starts_at_zero=False*, ***kwargs*)

Bases: *matplotlib.patches.Polygon*

Like Arrow, but lets you set head width and head height independently.

Constructor arguments

***width*: float (default: 0.001)** width of full arrow tail

***length_includes_head*: [True | False] (default: False)** True if head is to be counted in calculating the length.

***head_width*: float or None (default: 3*width)** total width of the full arrow head

***head_length*: float or None (default: 1.5 * head_width)** length of arrow head

***shape*: ['full', 'left', 'right'] (default: 'full')** draw the left-half, right-half, or full arrow

***overhang*: float (default: 0)** fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

***head_starts_at_zero*: [True | False] (default: False)** if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from `Patch`) are:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

class `matplotlib.patches.`**`FancyArrowPatch`**(*posA=None*, *posB=None*, *path=None*, *arrowstyle='simple'*, *arrow_transmuter=None*, *connectionstyle='arc3'*, *connector=None*, *patchA=None*, *patchB=None*, *shrinkA=2.0*, *shrinkB=2.0*, *mutation_scale=1.0*, *mutation_aspect=None*, *dpi_cor=1.0*, ***kwargs*)

Bases: `matplotlib.patches.Patch`

A fancy arrow patch. It draws an arrow using the :class:ArrowStyle.

If *posA* and *posB* is given, a path connecting two point are created according to the connectionstyle. The path will be clipped with *patchA* and *patchB* and further shrunken by *shrinkA* and *shrinkB*. An arrow is drawn along this resulting path using the *arrowstyle* parameter. If *path* provided, an arrow is drawn along this path and *patchA*, *patchB*, *shrinkA*, and *shrinkB* are ignored.

The *connectionstyle* describes how *posA* and *posB* are connected. It can be an instance of the ConnectionStyle class (matplotlib.patches.ConnectionStlye) or a string of the connectionstyle name, with optional comma-separated attributes. The following connection styles are available.

Class	Name	Attrs
Angle	`angle`	angleA=90,angleB=0,rad=0.0
Angle3	`angle3`	angleA=90,angleB=0
Arc	`arc`	angleA=0,angleB=0,armA=None,armB=None,rad=0.0
Arc3	`arc3`	rad=0.0
Bar	`bar`	armA=0.0,armB=0.0,fraction=0.3,angle=None

The *arrowstyle* describes how the fancy arrow will be drawn. It can be string of the available arrowstyle names, with optional comma-separated attributes, or one of the ArrowStyle instance. The optional attributes are meant to be scaled with the *mutation_scale*. The following arrow styles are available.

Class	Name	Attrs
Curve	`-`	None
CurveB	`->`	head_length=0.4,head_width=0.2
BracketB	`-[`	widthB=1.0,lengthB=0.2,angleB=None
CurveFilledB	`-|>`	head_length=0.4,head_width=0.2
CurveA	`<-`	head_length=0.4,head_width=0.2
CurveAB	`<->`	head_length=0.4,head_width=0.2
CurveFilledA	`<|-`	head_length=0.4,head_width=0.2
CurveFilledAB	`<|-|>`	head_length=0.4,head_width=0.2
BracketA	`]-`	widthA=1.0,lengthA=0.2,angleA=None
BracketAB	`]-[`	widthA=1.0,lengthA=0.2,angleA=None,widthB=1.0,lengthB=0.2,angleB=None
Fancy	`fancy`	head_length=0.4,head_width=0.4,tail_width=0.4
Simple	`simple`	head_length=0.5,head_width=0.5,tail_width=0.2
Wedge	`wedge`	tail_width=0.3,shrink_factor=0.5
BarAB	`|-|`	widthA=1.0,angleA=None,widthB=1.0,angleB=None

mutation_scale [a value with which attributes of arrowstyle] (e.g., head_length) will be scaled. default=1.

mutation_aspect [The height of the rectangle will be] squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it. default=None.

Valid kwargs are:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

draw(*renderer*)

get_arrowstyle()
Return the arrowstyle object

get_connectionstyle()
Return the ConnectionStyle instance

get_dpi_cor()
dpi_cor is currently used for linewidth-related things and shrink factor. Mutation scale is not

affected by this.

get_mutation_aspect()
Return the aspect ratio of the bbox mutation.

get_mutation_scale()
Return the mutation scale.

get_path()
return the path of the arrow in the data coordinate. Use get_path_in_displaycoord() method to retrieve the arrow path in the display coord.

get_path_in_displaycoord()
Return the mutated path of the arrow in the display coord

set_arrowstyle(*arrowstyle=None*, ***kw*)
Set the arrow style.

***arrowstyle* can be a string with arrowstyle name with optional** comma-separated attributes.
Alternatively, the attrs can be provided as keywords.

set_arrowstyle("Fancy,head_length=0.2") set_arrowstyle("fancy", head_length=0.2)

Old attrs simply are forgotten.

Without argument (or with arrowstyle=None), return available box styles as a list of strings.

set_connectionstyle(*connectionstyle*, ***kw*)
Set the connection style.

***connectionstyle* can be a string with connectionstyle name with** optional comma-separated attributes. Alternatively, the attrs can be provided as keywords.

set_connectionstyle("arc,angleA=0,armA=30,rad=10") set_connectionstyle("arc", angleA=0,armA=30,rad=10)

Old attrs simply are forgotten.

Without argument (or with connectionstyle=None), return available styles as a list of strings.

set_dpi_cor(*dpi_cor*)
dpi_cor is currently used for linewidth-related things and shrink factor. Mutation scale is not affected by this.

set_mutation_aspect(*aspect*)
Set the aspect ratio of the bbox mutation.

ACCEPTS: float

set_mutation_scale(*scale*)
Set the mutation scale.

ACCEPTS: float

set_patchA(*patchA*)
set the begin patch.

set_patchB(*patchB*)
set the begin patch

set_positions(*posA*, *posB*)
set the begin and end positions of the connecting path. Use current value if None.

class matplotlib.patches.**FancyBboxPatch**(*xy*, *width*, *height*, *boxstyle='round'*, *bbox_transmuter=None*, *mutation_scale=1.0*, *mutation_aspect=None*, ***kwargs*)
Bases: `matplotlib.patches.Patch`

Draw a fancy box around a rectangle with lower left at *xy*=(*x*, *y*) with specified width and height.

`FancyBboxPatch` class is similar to `Rectangle` class, but it draws a fancy box around the rectangle. The transformation of the rectangle box to the fancy box is delegated to the `BoxTransmuterBase` and its derived classes.

xy = lower left corner

width, *height*

boxstyle determines what kind of fancy box will be drawn. It can be a string of the style name with a comma separated attribute, or an instance of `BoxStyle`. Following box styles are available.

Class	Name	Attrs
Circle	`circle`	pad=0.3
DArrow	`darrow`	pad=0.3
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3,rounding_size=None
Round4	`round4`	pad=0.3,rounding_size=None
Roundtooth	`roundtooth`	pad=0.3,tooth_size=None
Sawtooth	`sawtooth`	pad=0.3,tooth_size=None
Square	`square`	pad=0.3

mutation_scale : a value with which attributes of boxstyle (e.g., pad) will be scaled. default=1.

mutation_aspect : The height of the rectangle will be squeezed by this value before the mutation and the mutated box will be stretched by the inverse of it. default=None.

Valid kwargs are:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color

Continued on

Table 64.8 – continued from previous page

Property	Description
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

get_bbox()

get_boxstyle()
Return the boxstyle object

get_height()
Return the height of the rectangle

get_mutation_aspect()
Return the aspect ratio of the bbox mutation.

get_mutation_scale()
Return the mutation scale.

get_path()
Return the mutated path of the rectangle

get_width()
Return the width of the rectangle

get_x()
Return the left coord of the rectangle

get_y()
Return the bottom coord of the rectangle

set_bounds(**args*)
Set the bounds of the rectangle: l,b,w,h

ACCEPTS: (left, bottom, width, height)

set_boxstyle(*boxstyle=None*, ***kw*)

Set the box style.

boxstyle can be a string with boxstyle name with optional comma-separated attributes. Alternatively, the attrs can be provided as keywords:

```
set_boxstyle("round,pad=0.2")
set_boxstyle("round", pad=0.2)
```

Old attrs simply are forgotten.

Without argument (or with *boxstyle* = None), it returns available box styles.

The following boxstyles are available:

Class	Name	Attrs
Circle	`circle`	pad=0.3
DArrow	`darrow`	pad=0.3
LArrow	`larrow`	pad=0.3
RArrow	`rarrow`	pad=0.3
Round	`round`	pad=0.3,rounding_size=None
Round4	`round4`	pad=0.3,rounding_size=None
Roundtooth	`roundtooth`	pad=0.3,tooth_size=None
Sawtooth	`sawtooth`	pad=0.3,tooth_size=None
Square	`square`	pad=0.3

ACCEPTS: ['circle' | 'darrow' | 'larrow' | 'rarrow' | 'round' | 'round4' | 'roundtooth' | 'sawtooth' | 'square']

set_height(*h*)

Set the width rectangle

ACCEPTS: float

set_mutation_aspect(*aspect*)

Set the aspect ratio of the bbox mutation.

ACCEPTS: float

set_mutation_scale(*scale*)

Set the mutation scale.

ACCEPTS: float

set_width(*w*)

Set the width rectangle

ACCEPTS: float

set_x(*x*)

Set the left coord of the rectangle

ACCEPTS: float

set_y(*y*)

Set the bottom coord of the rectangle

ACCEPTS: float

class `matplotlib.patches.`**`Patch`**(*edgecolor=None*, *facecolor=None*, *color=None*, *linewidth=None*, *linestyle=None*, *antialiased=None*, *hatch=None*, *fill=True*, *capstyle=None*, *joinstyle=None*, ***kwargs*)

Bases: `matplotlib.artist.Artist`

A patch is a 2D artist with a face color and an edge color.

If any of *edgecolor*, *facecolor*, *linewidth*, or *antialiased* are *None*, they default to their rc params setting.

The following kwarg properties are supported

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td><code>agg_filter</code></td><td>unknown</td></tr>
<tr><td><code>alpha</code></td><td>float or None</td></tr>
<tr><td><code>animated</code></td><td>[True | False]</td></tr>
<tr><td><code>antialiased</code> or aa</td><td>[True | False] or None for default</td></tr>
<tr><td><code>axes</code></td><td>an <code>Axes</code> instance</td></tr>
<tr><td><code>capstyle</code></td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td><code>clip_box</code></td><td>a <code>matplotlib.transforms.Bbox</code> instance</td></tr>
<tr><td><code>clip_on</code></td><td>[True | False]</td></tr>
<tr><td><code>clip_path</code></td><td>[(<code>Path</code>, <code>Transform</code>) | <code>Patch</code> | None]</td></tr>
<tr><td><code>color</code></td><td>matplotlib color spec</td></tr>
<tr><td><code>contains</code></td><td>a callable function</td></tr>
<tr><td><code>edgecolor</code> or ec</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td><code>facecolor</code> or fc</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td><code>figure</code></td><td>a <code>matplotlib.figure.Figure</code> instance</td></tr>
<tr><td><code>fill</code></td><td>[True | False]</td></tr>
<tr><td><code>gid</code></td><td>an id string</td></tr>
<tr><td><code>hatch</code></td><td>['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']</td></tr>
<tr><td><code>joinstyle</code></td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td><code>label</code></td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td><code>linestyle</code> or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> | <code>':'</code> | <code>'None'</code></td></tr>
<tr><td><code>linewidth</code> or lw</td><td>float or None for default</td></tr>
<tr><td><code>path_effects</code></td><td>unknown</td></tr>
<tr><td><code>picker</code></td><td>[None|float|boolean|callable]</td></tr>
<tr><td><code>rasterized</code></td><td>[True | False | None]</td></tr>
<tr><td><code>sketch_params</code></td><td>unknown</td></tr>
<tr><td><code>snap</code></td><td>unknown</td></tr>
<tr><td><code>transform</code></td><td><code>Transform</code> instance</td></tr>
<tr><td><code>url</code></td><td>a url string</td></tr>
<tr><td><code>visible</code></td><td>[True | False]</td></tr>
<tr><td><code>zorder</code></td><td>any number</td></tr>
</table>

`contains`(*mouseevent*, *radius=None*)

Test whether the mouse event occurred in the patch.

Returns T/F, {}

contains_point(*point*, *radius=None*)
Returns *True* if the given point is inside the path (transformed with its transform attribute).

draw(*artist*, *renderer*, **args*, ***kwargs*)
Draw the `Patch` to the given *renderer*.

fill
return whether fill is set

get_aa()
Returns True if the `Patch` is to be drawn with antialiasing.

get_antialiased()
Returns True if the `Patch` is to be drawn with antialiasing.

get_capstyle()
Return the current capstyle

get_data_transform()
Return the `Transform` instance which maps data coordinates to physical coordinates.

get_ec()
Return the edge color of the `Patch`.

get_edgecolor()
Return the edge color of the `Patch`.

get_extents()
Return a `Bbox` object defining the axis-aligned extents of the `Patch`.

get_facecolor()
Return the face color of the `Patch`.

get_fc()
Return the face color of the `Patch`.

get_fill()
return whether fill is set

get_hatch()
Return the current hatching pattern

get_joinstyle()
Return the current joinstyle

get_linestyle()
Return the linestyle. Will be one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

get_linewidth()
Return the line width in points.

get_ls()
Return the linestyle. Will be one of ['solid' | 'dashed' | 'dashdot' | 'dotted']

get_lw()
Return the line width in points.

get_patch_transform()
Return the *Transform* instance which takes patch coordinates to data coordinates.

For example, one may define a patch of a circle which represents a radius of 5 by providing coordinates for a unit circle, and a transform which scales the coordinates (the patch coordinate) by 5.

get_path()
Return the path of this patch

get_transform()
Return the *Transform* applied to the *Patch*.

get_verts()
Return a copy of the vertices used in this patch

If the patch contains Bezier curves, the curves will be interpolated by line segments. To access the curves as curves, use *get_path()*.

get_window_extent(*renderer=None*)

set_aa(*aa*)
alias for set_antialiased

set_alpha(*alpha*)
Set the alpha tranparency of the patch.

ACCEPTS: float or None

set_antialiased(*aa*)
Set whether to use antialiased rendering

ACCEPTS: [True | False] or None for default

set_capstyle(*s*)
Set the patch capstyle

ACCEPTS: ['butt' | 'round' | 'projecting']

set_color(*c*)
Set both the edgecolor and the facecolor.

ACCEPTS: matplotlib color spec

See also:

set_facecolor(), ***set_edgecolor()*** For setting the edge or face color individually.

set_ec(*color*)
alias for set_edgecolor

set_edgecolor(*color*)
Set the patch edge color

ACCEPTS: mpl color spec, or None for default, or 'none' for no color

set_facecolor(*color*)
Set the patch face color

ACCEPTS: mpl color spec, or None for default, or 'none' for no color

set_fc(*color*)
alias for set_facecolor

set_fill(*b*)
Set whether to fill the patch

ACCEPTS: [True | False]

set_hatch(*hatch*)
Set the hatching pattern

hatch can be one of:

```
/   - diagonal hatching
\   - back diagonal
|   - vertical
-   - horizontal
+   - crossed
x   - crossed diagonal
o   - small circle
O   - large circle
.   - dots
*   - stars
```

Letters can be combined, in which case all the specified hatchings are done. If same letter repeats, it increases the density of hatching of that pattern.

Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

ACCEPTS: ['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']

set_joinstyle(*s*)
Set the patch joinstyle

ACCEPTS: ['miter' | 'round' | 'bevel']

set_linestyle(*ls*)
Set the patch linestyle

linestyle	description
`'-'` or `'solid'`	solid line
`'--'` or `'dashed'`	dashed line
`'-.'` or `'dashdot'`	dash-dotted line
`':'` or `'dotted'`	dotted line

Alternatively a dash tuple of the following form can be provided:

```
(offset, onoffseq),
```

where onoffseq is an even length tuple of on and off ink in points.

ACCEPTS: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None' | ' ' | '']

Parameters ls : { '-', '–', '-.', ':'} and more see description
The line style.

set_linewidth(*w*)
Set the patch linewidth in points

ACCEPTS: float or None for default

set_ls(*ls*)
alias for set_linestyle

set_lw(*lw*)
alias for set_linewidth

update_from(*other*)
Updates this *Patch* from the properties of *other*.

validCap = ('butt', 'round', 'projecting')

validJoin = ('miter', 'round', 'bevel')

zorder = 1

class matplotlib.patches.**PathPatch**(*path*, ***kwargs*)
Bases: *matplotlib.patches.Patch*

A general polycurve path patch.

path is a *matplotlib.path.Path* object.

Valid kwargs are:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color

Continued on

Table 64.10 – continued from previous page

Property	Description
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

See also:

Patch For additional kwargs

get_path()

class matplotlib.patches.**Polygon**(*xy*, *closed=True*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

A general polygon patch.

xy is a numpy array with shape Nx2.

If *closed* is *True*, the polygon will be closed so the starting and ending points are the same.

Valid kwargs are:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]

Continued on

Table 64.11 – continued from previous page

Property	Description
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

See also:

Patch For additional kwargs

get_closed()

Returns if the polygon is closed

Returns closed : bool

If the path is closed

get_path()

Get the path of the polygon

Returns path : Path

The *Path* object for the polygon

get_xy()

Get the vertices of the path

Returns vertices : numpy array

The coordinates of the vertices as a Nx2 ndarray.

set_closed(*closed*)

Set if the polygon is closed

Parameters closed : bool

True if the polygon is closed

set_xy(*xy*)

Set the vertices of the polygon

Parameters xy : numpy array or iterable of pairs

The coordinates of the vertices as a Nx2 ndarray or iterable of pairs.

xy

Set/get the vertices of the polygon. This property is provided for backward compatibility with matplotlib 0.91.x only. New code should use `get_xy()` and `set_xy()` instead.

class `matplotlib.patches.Rectangle`(*xy*, *width*, *height*, *angle=0.0*, ***kwargs*)

Bases: `matplotlib.patches.Patch`

Draw a rectangle with lower left at *xy* = (*x*, *y*) with specified *width* and *height*.

angle rotation in degrees (anti-clockwise)

fill is a boolean indicating whether to fill the rectangle

Valid kwargs are:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string

Continued on

Table 64.12 – continued from previous page

Property	Description
visible	[True | False]
zorder	any number

get_bbox()

get_height()
Return the height of the rectangle

get_patch_transform()

get_path()
Return the vertices of the rectangle

get_width()
Return the width of the rectangle

get_x()
Return the left coord of the rectangle

get_xy()
Return the left and bottom coords of the rectangle

get_y()
Return the bottom coord of the rectangle

set_bounds(**args*)
Set the bounds of the rectangle: l,b,w,h

ACCEPTS: (left, bottom, width, height)

set_height(*h*)
Set the width rectangle

ACCEPTS: float

set_width(*w*)
Set the width rectangle

ACCEPTS: float

set_x(*x*)
Set the left coord of the rectangle

ACCEPTS: float

set_xy(*xy*)
Set the left and bottom coords of the rectangle

ACCEPTS: 2-item sequence

set_y(*y*)
Set the bottom coord of the rectangle

ACCEPTS: float

xy

Return the left and bottom coords of the rectangle

class matplotlib.patches.**RegularPolygon**(*xy*, *numVertices*, *radius=5*, *orientation=0*, ***kwargs*)

Bases: matplotlib.patches.Patch

A regular polygon patch.

Constructor arguments:
xy A length 2 tuple (*x*, *y*) of the center.
numVertices the number of vertices.
radius The distance from the center to each of the vertices.
orientation rotates the polygon (in radians).
Valid kwargs are:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an Axes instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]

Continued on

Table 64.13 – continued from previous page

Property	Description
zorder	any number

get_patch_transform()

get_path()

numvertices

orientation

radius

xy

class matplotlib.patches.**Shadow**(*patch*, *ox*, *oy*, *props=None*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

Create a shadow of the given *patch* offset by *ox*, *oy*. *props*, if not *None*, is a patch property update dictionary. If *None*, the shadow will have have the same color as the face, but darkened.

kwargs are

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
joinstyle	['miter' | 'round' | 'bevel']

Continued on

Table 64.14 – continued from previous page

Property	Description
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

draw(*renderer*)

get_patch_transform()

get_path()

class matplotlib.patches.**Wedge**(*center*, *r*, *theta1*, *theta2*, *width=None*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

Wedge shaped patch.

Draw a wedge centered at *x*, *y* center with radius *r* that sweeps *theta1* to *theta2* (in degrees). If *width* is given, then a partial wedge is drawn from inner radius *r* - *width* to outer radius *r*.

Valid kwargs are:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color

Continued on

Table 64.15 – continued from previous page

Property	Description
figure	a *matplotlib.figure.Figure* instance
fill	[True | False]
gid	an id string
hatch	[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]
joinstyle	[’miter’ | ‘round’ | ‘bevel’]
label	string or anything printable with ‘%s’ conversion.
linestyle or ls	[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
linewidth or lw	float or None for default
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
visible	[True | False]
zorder	any number

get_path()

set_center(*center*)

set_radius(*radius*)

set_theta1(*theta1*)

set_theta2(*theta2*)

set_width(*width*)

class matplotlib.patches.**YAArrow**(*figure*, *xytip*, *xybase*, *width=4*, *frac=0.1*, *headwidth=12*, ***kwargs*)

Bases: *matplotlib.patches.Patch*

Yet another arrow class.

This is an arrow that is defined in display space and has a tip at *x1*, *y1* and a base at *x2*, *y2*.

Constructor arguments:
xytip (*x*, *y*) location of arrow tip
xybase (*x*, *y*) location the arrow base mid point
figure The *Figure* instance (fig.dpi)

width The width of the arrow in points
frac The fraction of the arrow length occupied by the head
headwidth The width of the base of the arrow head in points
Valid kwargs are:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

get_patch_transform()

get_path()

getpoints(*x1*, *y1*, *x2*, *y2*, *k*)
For line segment defined by (*x1*, *y1*) and (*x2*, *y2*) return the points on the line that is perpendicular to the line and intersects (*x2*, *y2*) and the distance from (*x2*, *y2*) of the returned points is *k*.

matplotlib.patches.**bbox_artist**(*artist*, *renderer*, *props=None*, *fill=True*)

This is a debug function to draw a rectangle around the bounding box returned by `get_window_extent()` of an artist, to test whether the artist is returning the correct bbox.

props is a dict of rectangle props with the additional property 'pad' that sets the padding around the bbox in points.

matplotlib.patches.**draw_bbox**(*bbox*, *renderer*, *color='k'*, *trans=None*)

This is a debug function to draw a rectangle around the bounding box returned by `get_window_extent()` of an artist, to test whether the artist is returning the correct bbox.

CHAPTER

SIXTYFIVE

PATH

65.1 matplotlib.path

A module for dealing with the polylines used throughout matplotlib.

The primary class for polyline handling in matplotlib is `Path`. Almost all vector drawing makes use of Paths somewhere in the drawing pipeline.

Whilst a `Path` instance itself cannot be drawn, there exists `Artist` subclasses which can be used for convenient Path visualisation - the two most frequently used of these are `PathPatch` and `PathCollection`.

class `matplotlib.path.`**`Path`**(*vertices*, *codes=None*, *_interpolation_steps=1*, *closed=False*, *readonly=False*)

Bases: `object`

`Path` represents a series of possibly disconnected, possibly closed, line and curve segments.

The underlying storage is made up of two parallel numpy arrays:

- *vertices*: an Nx2 float array of vertices
- *codes*: an N-length uint8 array of vertex types

These two arrays always have the same length in the first dimension. For example, to represent a cubic curve, you must provide three vertices as well as three codes `CURVE3`.

The code types are:

- **`STOP`** [1 vertex (ignored)] A marker for the end of the entire path (currently not required and ignored)
- **`MOVETO`** [1 vertex] Pick up the pen and move to the given vertex.
- **`LINETO`** [1 vertex] Draw a line from the current position to the given vertex.
- **`CURVE3`** [1 control point, 1 endpoint] Draw a quadratic Bezier curve from the current position, with the given control point, to the given end point.
- **`CURVE4`** [2 control points, 1 endpoint] Draw a cubic Bezier curve from the current position, with the given control points, to the given end point.
- **`CLOSEPOLY`** [1 vertex (ignored)] Draw a line segment to the start point of the current polyline.

Users of Path objects should not access the vertices and codes arrays directly. Instead, they should use `iter_segments()` or `cleaned()` to get the vertex/code pairs. This is important, since many `Path` objects, as an optimization, do not store a *codes* at all, but have a default one provided for them by `iter_segments()`.

Note: The vertices and codes arrays should be treated as immutable – there are a number of optimizations and assumptions made up front in the constructor that will not change when the data changes.

Create a new path with the given vertices and codes.

Parameters vertices : array_like
: The `(n, 2)` float array, masked array or sequence of pairs representing the vertices of the path.

  If *vertices* contains masked values, they will be converted to NaNs which are then handled correctly by the Agg PathIterator and other consumers of path data, such as `iter_segments()`.

codes : {None, array_like}, optional
: n-length array integers representing the codes of the path. If not None, codes must be the same length as vertices. If None, *vertices* will be treated as a series of line segments.

_interpolation_steps : int, optional
: Used as a hint to certain projections, such as Polar, that this path should be linearly interpolated immediately before drawing. This attribute is primarily an implementation detail and is not intended for public use.

closed : bool, optional
: If *codes* is None and closed is True, vertices will be treated as line segments of a closed polygon.

readonly : bool, optional
: Makes the path behave in an immutable way and sets the vertices and codes as read-only arrays.

CLOSEPOLY = 79

CURVE3 = 3

CURVE4 = 4

LINETO = 2

MOVETO = 1

NUM_VERTICES_FOR_CODE = {0: 1, 1: 1, 2: 1, 3: 2, 4: 3, 79: 1}

A dictionary mapping Path codes to the number of vertices that the code expects.

STOP = 0

classmethod arc(*theta1*, *theta2*, *n=None*, *is_wedge=False*)

Return an arc on the unit circle from angle *theta1* to angle *theta2* (in degrees).

If *n* is provided, it is the number of spline segments to make. If *n* is not provided, the number of spline segments is determined based on the delta between *theta1* and *theta2*.

Masionobe, L. 2003. Drawing an elliptical arc using polylines, quadratic or cubic Bezier curves.

classmethod circle(*center=(0.0, 0.0)*, *radius=1.0*, *readonly=False*)

Return a Path representing a circle of a given radius and center.

Parameters center : pair of floats

The center of the circle. Default `(0, 0)`.

radius : float

The radius of the circle. Default is 1.

readonly : bool

Whether the created path should have the "readonly" argument set when creating the Path instance.

Notes

The circle is approximated using cubic Bezier curves. This uses 8 splines around the circle using the approach presented here:

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.

cleaned(*transform=None*, *remove_nans=False*, *clip=None*, *quantize=False*, *simplify=False*, *curves=False*, *stroke_width=1.0*, *snap=False*, *sketch=None*)

Cleans up the path according to the parameters returning a new Path instance.

See also:

See `iter_segments()` for details of the keyword arguments.

Returns Path instance with cleaned up vertices and codes.

clip_to_bbox(*bbox*, *inside=True*)

Clip the path to the given bounding box.

The path must be made up of one or more closed polygons. This algorithm will not behave correctly for unclosed paths.

If *inside* is `True`, clip to the inside of the box, otherwise to the outside of the box.

code_type

alias of `uint8`

codes

The list of codes in the `Path` as a 1-D numpy array. Each code is one of `STOP`, `MOVETO`, `LINETO`, `CURVE3`, `CURVE4` or `CLOSEPOLY`. For codes that correspond to more than one vertex (`CURVE3` and `CURVE4`), that code will be repeated so that the length of `self.vertices` and `self.codes` is always the same.

contains_path(*path*, *transform=None*)

Returns *True* if this path completely contains the given path.

If *transform* is not *None*, the path will be transformed before performing the test.

contains_point(*point*, *transform=None*, *radius=0.0*)
Returns *True* if the path contains the given point.

If *transform* is not *None*, the path will be transformed before performing the test.

radius allows the path to be made slightly larger or smaller.

contains_points(*points*, *transform=None*, *radius=0.0*)
Returns a bool array which is *True* if the path contains the corresponding point.

If *transform* is not *None*, the path will be transformed before performing the test.

radius allows the path to be made slightly larger or smaller.

copy()
Returns a shallow copy of the `Path`, which will share the vertices and codes with the source `Path`.

deepcopy()
Returns a deepcopy of the `Path`. The `Path` will not be readonly, even if the source `Path` is.

get_extents(*transform=None*)
Returns the extents (*xmin*, *ymin*, *xmax*, *ymax*) of the path.

Unlike computing the extents on the *vertices* alone, this algorithm will take into account the curves and deal with control points appropriately.

has_nonfinite
`True` if the vertices array has nonfinite values.

classmethod hatch(*hatchpattern*, *density=6*)
Given a hatch specifier, *hatchpattern*, generates a Path that can be used in a repeated hatching pattern. *density* is the number of lines per unit square.

interpolated(*steps*)
Returns a new path resampled to length N x steps. Does not currently handle interpolating curves.

intersects_bbox(*bbox*, *filled=True*)
Returns *True* if this path intersects a given `Bbox`.

filled, when True, treats the path as if it was filled. That is, if one path completely encloses the other, `intersects_path()` will return True.

intersects_path(*other*, *filled=True*)
Returns *True* if this path intersects another given path.

filled, when True, treats the paths as if they were filled. That is, if one path completely encloses the other, `intersects_path()` will return True.

iter_segments(*transform=None*, *remove_nans=True*, *clip=None*, *snap=False*, *stroke_width=1.0*, *simplify=None*, *curves=True*, *sketch=None*)
Iterates over all of the curve segments in the path. Each iteration returns a 2-tuple (*vertices*, *code*), where *vertices* is a sequence of 1 - 3 coordinate pairs, and *code* is one of the `Path` codes.

Additionally, this method can provide a number of standard cleanups and conversions to the path.

Parameters transform : None or `Transform` instance
If not None, the given affine transformation will be applied to the path.

remove_nans : {False, True}, optional
If True, will remove all NaNs from the path and insert MOVETO commands to skip over them.

clip : None or sequence, optional
If not None, must be a four-tuple (x1, y1, x2, y2) defining a rectangle in which to clip the path.

snap : None or bool, optional
If None, auto-snap to pixels, to reduce fuzziness of rectilinear lines. If True, force snapping, and if False, don't snap.

stroke_width : float, optional
The width of the stroke being drawn. Needed as a hint for the snapping algorithm.

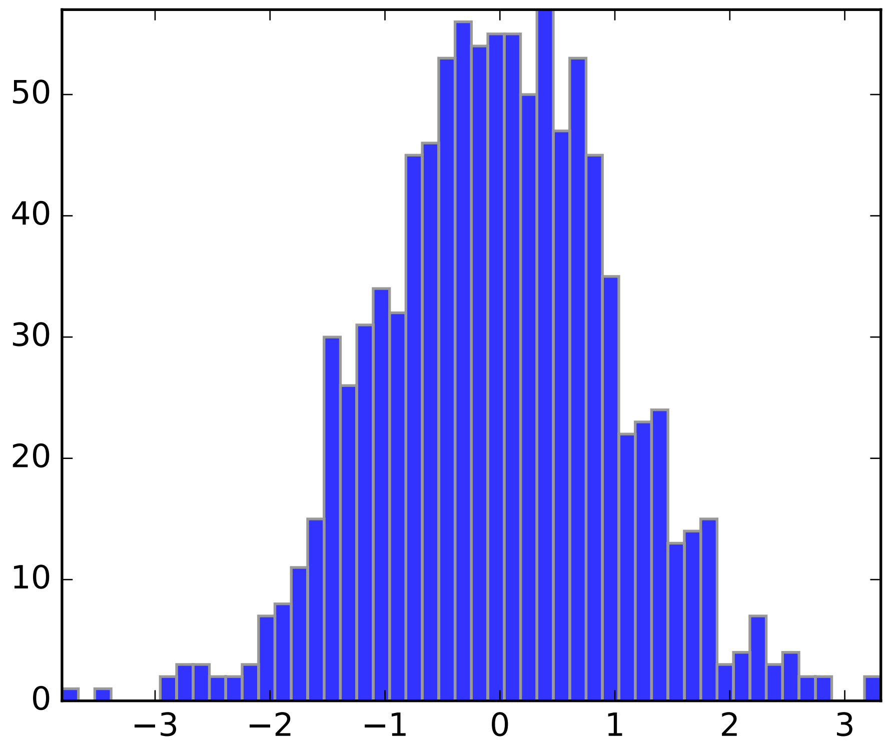
simplify : None or bool, optional
If True, perform simplification, to remove vertices that do not affect the appearance of the path. If False, perform no simplification. If None, use the should_simplify member variable.

curves : {True, False}, optional
If True, curve segments will be returned as curve segments. If False, all curves will be converted to line segments.

sketch : None or sequence, optional
If not None, must be a 3-tuple of the form (scale, length, randomness), representing the sketch parameters.

classmethod `make_compound_path`(**args*)
Make a compound path from a list of Path objects.

classmethod `make_compound_path_from_polys`(*XY*)
Make a compound path object to draw a number of polygons with equal numbers of sides XY is a (numpolys x numsides x 2) numpy array of vertices. Return object is a `Path`



readonly
True if the *Path* is read-only.

should_simplify
True if the vertices array should be simplified.

simplify_threshold
The fraction of a pixel difference below which vertices will be simplified out.

to_polygons(*transform=None*, *width=0*, *height=0*)
Convert this path to a list of polygons. Each polygon is an Nx2 array of vertices. In other words, each polygon has no MOVETO instructions or curves. This is useful for displaying in backends that do not support compound paths or Bezier curves, such as GDK.

If *width* and *height* are both non-zero then the lines will be simplified so that vertices outside of (0, 0), (width, height) will be clipped.

transformed(*transform*)
Return a transformed copy of the path.

See also:

matplotlib.transforms.TransformedPath A specialized path class that will cache the transformed result and automatically update when the transform changes.

classmethod unit_circle()
Return the readonly *Path* of the unit circle.

For most cases, `Path.circle()` will be what you want.

classmethod unit_circle_righthalf()

Return a `Path` of the right half of a unit circle. The circle is approximated using cubic Bezier curves. This uses 4 splines around the circle using the approach presented here:

Lancaster, Don. Approximating a Circle or an Ellipse Using Four Bezier Cubic Splines.

classmethod unit_rectangle()

Return a `Path` instance of the unit rectangle from (0, 0) to (1, 1).

classmethod unit_regular_asterisk(*numVertices*)

Return a `Path` for a unit regular asterisk with the given numVertices and radius of 1.0, centered at (0, 0).

classmethod unit_regular_polygon(*numVertices*)

Return a `Path` instance for a unit regular polygon with the given *numVertices* and radius of 1.0, centered at (0, 0).

classmethod unit_regular_star(*numVertices*, *innerCircle=0.5*)

Return a `Path` for a unit regular star with the given numVertices and radius of 1.0, centered at (0, 0).

vertices

The list of vertices in the `Path` as an Nx2 numpy array.

classmethod wedge(*theta1*, *theta2*, *n=None*)

Return a wedge of the unit circle from angle *theta1* to angle *theta2* (in degrees).

If *n* is provided, it is the number of spline segments to make. If *n* is not provided, the number of spline segments is determined based on the delta between *theta1* and *theta2*.

matplotlib.path.**get_path_collection_extents**(*master_transform*, *paths*, *transforms*, *offsets*, *offset_transform*)

Given a sequence of `Path` objects, `Transform` objects and offsets, as found in a `PathCollection`, returns the bounding box that encapsulates all of them.

master_transform is a global transformation to apply to all paths

paths is a sequence of `Path` instances.

transforms is a sequence of `Affine2D` instances.

offsets is a sequence of (x, y) offsets (or an Nx2 array)

offset_transform is a `Affine2D` to apply to the offsets before applying the offset to the path.

The way that *paths*, *transforms* and *offsets* are combined follows the same method as for collections. Each is iterated over independently, so if you have 3 paths, 2 transforms and 1 offset, their combinations are as follows:

(A, A, A), (B, B, A), (C, A, A)

matplotlib.path.**get_paths_extents**(*paths*, *transforms=[]*)

Given a sequence of `Path` objects and optional `Transform` objects, returns the bounding box that encapsulates all of them.

paths is a sequence of `Path` instances.

transforms is an optional sequence of `Affine2D` instances to apply to each path.

CHAPTER SIXTYSIX

PATHEFFECTS

66.1 matplotlib.patheffects

Defines classes for path effects. The path effects are supported in *Text*, *Line2D* and *Patch*.

class matplotlib.patheffects.**AbstractPathEffect**(*offset=(0.0, 0.0)*)

Bases: object

A base class for path effects.

Subclasses should override the draw_path method to add effect functionality.

Parameters offset : pair of floats
The offset to apply to the path, measured in points.

draw_path(*renderer*, *gc*, *tpath*, *affine*, *rgbFace=None*)

Derived should override this method. The arguments are the same as *matplotlib.backend_bases.RendererBase.draw_path()* except the first argument is a renderer.

class matplotlib.patheffects.**Normal**(*offset=(0.0, 0.0)*)

Bases: *matplotlib.patheffects.AbstractPathEffect*

The "identity" PathEffect.

The Normal PathEffect's sole purpose is to draw the original artist with no special path effect.

Parameters offset : pair of floats
The offset to apply to the path, measured in points.

class matplotlib.patheffects.**PathEffectRenderer**(*path_effects*, *renderer*)

Bases: *matplotlib.backend_bases.RendererBase*

Implements a Renderer which contains another renderer.

This proxy then intercepts draw calls, calling the appropriate *AbstractPathEffect* draw method.

Note: Not all methods have been overridden on this RendererBase subclass. It may be necessary to add further methods to extend the PathEffects capabilities further.

Parameters path_effects : iterable of *AbstractPathEffect*
The path effects which this renderer represents.

renderer : `matplotlib.backend_bases.RendererBase` instance

copy_with_path_effect(*path_effects*)

draw_markers(*gc*, *marker_path*, *marker_trans*, *path*, **args*, ***kwargs*)

draw_path(*gc*, *tpath*, *affine*, *rgbFace=None*)

draw_path_collection(*gc*, *master_transform*, *paths*, **args*, ***kwargs*)

new_gc()

points_to_pixels(*points*)

class matplotlib.patheffects.**PathPatchEffect**(*offset=(0, 0)*, ***kwargs*)

Bases: `matplotlib.patheffects.AbstractPathEffect`

Draws a `PathPatch` instance whose Path comes from the original PathEffect artist.

Parameters offset : pair of floats

The offset to apply to the path, in points.

****kwargs :**

All keyword arguments are passed through to the `PathPatch` constructor. The properties which cannot be overridden are "path", "clip_box" "transform" and "clip_path".

draw_path(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

class matplotlib.patheffects.**SimpleLineShadow**(*offset=(2, -2)*, *shadow_color='k'*, *alpha=0.3*, *rho=0.3*, ***kwargs*)

Bases: `matplotlib.patheffects.AbstractPathEffect`

A simple shadow via a line.

Parameters offset : pair of floats

The offset to apply to the path, in points.

shadow_color : color

The shadow color. Default is black. A value of `None` takes the original artist's color with a scale factor of `rho`.

alpha : float

The alpha transparency of the created shadow patch. Default is 0.3.

rho : float

A scale factor to apply to the rgbFace color if `shadow_rgbFace` is `None`. Default is 0.3.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

draw_path(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

class `matplotlib.patheffects.`**`SimplePatchShadow`**(*offset=(2, -2)*, *shadow_rgbFace=None*, *alpha=None*, *rho=0.3*, ***kwargs*)

Bases: `matplotlib.patheffects.AbstractPathEffect`

A simple shadow via a filled patch.

Parameters offset : pair of floats

The offset of the shadow in points.

shadow_rgbFace : color

The shadow color.

alpha : float

The alpha transparency of the created shadow patch. Default is 0.3. http://matplotlib.1069221.n5.nabble.com/path-effects-question-td27630.html

rho : float

A scale factor to apply to the rgbFace color if `shadow_rgbFace` is not specified. Default is 0.3.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

`draw_path`(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

Overrides the standard draw_path to add the shadow offset and necessary color changes for the shadow.

class `matplotlib.patheffects.`**`Stroke`**(*offset=(0, 0)*, ***kwargs*)

Bases: `matplotlib.patheffects.AbstractPathEffect`

A line based PathEffect which re-draws a stroke.

The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

`draw_path`(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

draw the path with updated gc.

class `matplotlib.patheffects.`**`withSimplePatchShadow`**(*offset=(2, -2)*, *shadow_rgbFace=None*, *alpha=None*, *rho=0.3*, ***kwargs*)

Bases: `matplotlib.patheffects.SimplePatchShadow`

Adds a simple `SimplePatchShadow` and then draws the original Artist to avoid needing to call `Normal`.

Parameters offset : pair of floats

The offset of the shadow in points.

shadow_rgbFace : color

The shadow color.

alpha : float

The alpha transparency of the created shadow patch. Default is 0.3. http://matplotlib.1069221.n5.nabble.com/path-effects-question-td27630.html

rho : float

A scale factor to apply to the rgbFace color if `shadow_rgbFace` is not specified. Default is 0.3.

****kwargs**

Extra keywords are stored and passed through to `AbstractPathEffect._update_gc()`.

draw_path(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

class matplotlib.patheffects.**withStroke**(*offset=(0, 0)*, ***kwargs*)

Bases: *matplotlib.patheffects.Stroke*

Adds a simple *Stroke* and then draws the original Artist to avoid needing to call *Normal*.

The path will be stroked with its gc updated with the given keyword arguments, i.e., the keyword arguments should be valid gc parameter values.

draw_path(*renderer*, *gc*, *tpath*, *affine*, *rgbFace*)

CHAPTER

SIXTYSEVEN

PROJECTIONS

67.1 matplotlib.projections

class `matplotlib.projections.`**`ProjectionRegistry`**

Bases: `object`

Manages the set of projections available to the system.

`get_projection_class`(*name*)

Get a projection class from its *name*.

`get_projection_names`()

Get a list of the names of all projections currently registered.

`register`(**projections*)

Register a new set of projection(s).

`matplotlib.projections.`**`get_projection_class`**(*projection=None*)

Get a projection class from its name.

If *projection* is None, a standard rectilinear projection is returned.

`matplotlib.projections.`**`get_projection_names`**()

Get a list of acceptable projection names.

`matplotlib.projections.`**`process_projection_requirements`**(*figure*, **args*, ***kwargs*)

Handle the args/kwargs to for add_axes/add_subplot/gca, returning:

```
(axes_proj_class, proj_class_kwargs, proj_stack_key)
```

Which can be used for new axes initialization/identification.

Note: **kwargs** is modified in place.

67.2 matplotlib.projections.polar

class matplotlib.projections.polar.**InvertedPolarTransform**(*axis=None*, *use_rmin=True*)

Bases: `matplotlib.transforms.Transform`

The inverse of the polar transform, mapping Cartesian coordinate space *x* and *y* back to *theta* and *r*.

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`x === self.inverted().transform(self.transform(x))`

transform_non_affine(*xy*)

Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

class matplotlib.projections.polar.**PolarAffine**(*scale_transform*, *limits*)

Bases: `matplotlib.transforms.Affine2DBase`

The affine part of the polar projection. Scales the output so that maximum radius rests on the edge of the axes circle.

limits is the view limit of the data. The only part of its bounds that is used is ymax (for the radius maximum). The theta range is always fixed to (0, 2pi).

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class matplotlib.projections.polar.**PolarAxes**(**args*, ***kwargs*)

Bases: `matplotlib.axes._axes.Axes`

A polar graph projection, where the input dimensions are *theta*, *r*.

Theta starts pointing east and goes anti-clockwise.

class **InvertedPolarTransform**(*axis=None*, *use_rmin=True*)

Bases: `matplotlib.transforms.Transform`

The inverse of the polar transform, mapping Cartesian coordinate space *x* and *y* back to *theta* and *r*.

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

x === self.inverted().transform(self.transform(x))

transform_non_affine(*xy*)

Performs only the non-affine part of the transformation.

transform(values) is always equivalent to transform_affine(transform_non_affine(values))

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

class PolarAxes.**PolarAffine**(*scale_transform*, *limits*)

Bases: *matplotlib.transforms.Affine2DBase*

The affine part of the polar projection. Scales the output so that maximum radius rests on the edge of the axes circle.

limits is the view limit of the data. The only part of its bounds that is used is ymax (for the radius maximum). The theta range is always fixed to (0, 2pi).

get_matrix()

Get the Affine transformation array for the affine part of this transform.

class PolarAxes.**PolarTransform**(*axis=None*, *use_rmin=True*)

Bases: *matplotlib.transforms.Transform*

The base polar transform. This handles projection *theta* and *r* into Cartesian coordinate space *x* and *y*, but does not perform the ultimate affine transformation into the correct position.

inverted()

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

x === self.inverted().transform(self.transform(x))

transform_non_affine(*tr*)

Performs only the non-affine part of the transformation.

transform(values) is always equivalent to transform_affine(transform_non_affine(values))

In non-affine transformations, this is generally equivalent to transform(values). In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x input_dims) and returns a numpy array of shape (N x output_dims).

Alternatively, accepts a numpy array of length input_dims and returns a numpy array of length output_dims.

transform_path_non_affine(*path*)
Returns a path, transformed only by the non-affine part of this transform.

path: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(`

class `PolarAxes.`**RadialLocator**(*base*)
Bases: `matplotlib.ticker.Locator`

Used to locate radius ticks.

Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base `Locator` (which may be different depending on the scale of the *r*-axis.

class `PolarAxes.`**ThetaFormatter**
Bases: `matplotlib.ticker.Formatter`

Used to format the *theta* tick labels. Converts the native unit of radians into degrees and adds a degree symbol.

`PolarAxes.`**can_pan**()
Return *True* if this axes supports the pan/zoom button functionality.

For polar axes, this is slightly misleading. Both panning and zooming are performed by the same button. Panning is performed in azimuth while zooming is done along the radial.

`PolarAxes.`**can_zoom**()
Return *True* if this axes supports the zoom box button functionality.

Polar axes do not support zoom boxes.

`PolarAxes.`**format_coord**(*theta*, *r*)
Return a format string formatting the coordinate using Unicode characters.

`PolarAxes.`**get_data_ratio**()
Return the aspect ratio of the data itself. For a polar plot, this should always be 1.0

`PolarAxes.`**get_rlabel_position**()

Returns float
The theta position of the radius labels in degrees.

`PolarAxes.`**get_theta_direction**()
Get the direction in which theta increases.
-1: Theta increases in the clockwise direction
1: Theta increases in the counterclockwise direction

`PolarAxes.`**get_theta_offset**()
Get the offset for the location of 0 in radians.

`PolarAxes.`**set_rgrids**(*radii*, *labels=None*, *angle=None*, *fmt=None*, ***kwargs*)
Set the radial locations and labels of the *r* grids.

The labels will appear at radial distances *radii* at the given *angle* in degrees.

labels, if not None, is a `len(radii)` list of strings of the labels to use at each radius.

If *labels* is None, the built-in formatter will be used.

Return value is a list of tuples (*line*, *label*), where *line* is `Line2D` instances and the *label* is `Text` instances.

kwargs are optional text properties for the labels:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	FancyBboxPatch prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or name or fontname or fontfamily	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']
`rotation_mode`	unknown
`size` or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large'
`sketch_params`	unknown
`snap`	unknown
`stretch` or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'c
`style` or fontstyle	['normal' | 'italic' | 'oblique']
`text`	string or anything printable with '%s' conversion.
`transform`	`Transform` instance
`url`	a url string
`usetex`	unknown
`variant` or fontvariant	['normal' | 'small-caps']
`verticalalignment` or va or ma	['center' | 'top' | 'bottom' | 'baseline']
`visible`	[True | False]
`weight` or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' |
`wrap`	unknown

Table 67.1 – continued fro

Property	Description
x	float
y	float
zorder	any number

ACCEPTS: sequence of floats

`PolarAxes.`**`set_rlabel_position`**`(`*value*`)`

Updates the theta position of the radius labels.

Parameters value : number

The angular position of the radius labels in degrees.

`PolarAxes.`**`set_theta_direction`**`(`*direction*`)`

Set the direction in which theta increases.

clockwise, -1: Theta increases in the clockwise direction

counterclockwise, anticlockwise, 1: Theta increases in the counterclockwise direction

`PolarAxes.`**`set_theta_offset`**`(`*offset*`)`

Set the offset for the location of 0 in radians.

`PolarAxes.`**`set_theta_zero_location`**`(`*loc*`)`

Sets the location of theta's zero. (Calls set_theta_offset with the correct value in radians under the hood.)

May be one of "N", "NW", "W", "SW", "S", "SE", "E", or "NE".

`PolarAxes.`**`set_thetagrids`**`(`*angles*, *labels=None*, *frac=None*, *fmt=None*, ***kwargs*`)`

Set the angles at which to place the theta grids (these gridlines are equal along the theta dimension). *angles* is in degrees.

labels, if not None, is a `len(angles)` list of strings of the labels to use at each angle.

If *labels* is None, the labels will be `fmt % angle`

frac is the fraction of the polar axes radius at which to place the label (1 is the edge). e.g., 1.05 is outside the axes and 0.95 is inside the axes.

Return value is a list of tuples (*line*, *label*), where *line* is `Line2D` instances and the *label* is `Text` instances.

kwargs are optional text properties for the labels:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
axes	an `Axes` instance
backgroundcolor	any matplotlib color
bbox	FancyBboxPatch prop dict
clip_box	a `matplotlib.transforms.Bbox` instance

Table 67.2 – continued fror

Property	Description
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	any matplotlib color
contains	a callable function
family or name or fontname or fontfamily	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a *matplotlib.figure.Figure* instance
fontproperties or font_properties	a *matplotlib.font_manager.FontProperties* instance
gid	an id string
horizontalalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large'
sketch_params	unknown
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'c
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	*Transform* instance
url	a url string
usetex	unknown
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' |
wrap	unknown
x	float
y	float
zorder	any number

ACCEPTS: sequence of floats

class `matplotlib.projections.polar.`**`PolarTransform`**(*axis=None*, *use_rmin=True*)

Bases: *matplotlib.transforms.Transform*

The base polar transform. This handles projection *theta* and *r* into Cartesian coordinate space *x* and *y*, but does not perform the ultimate affine transformation into the correct position.

`inverted()`

Return the corresponding inverse transformation.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

`x === self.inverted().transform(self.transform(x))`

transform_non_affine(*tr*)

Performs only the non-affine part of the transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

Accepts a numpy array of shape (N x `input_dims`) and returns a numpy array of shape (N x `output_dims`).

Alternatively, accepts a numpy array of length `input_dims` and returns a numpy array of length `output_dims`.

transform_path_non_affine(*path*)

Returns a path, transformed only by the non-affine part of this transform.

path: a `Path` instance.

`transform_path(path)` is equivalent to `transform_path_affine(transform_path_non_affine(value`

class `matplotlib.projections.polar.`**RadialLocator**(*base*)

Bases: `matplotlib.ticker.Locator`

Used to locate radius ticks.

Ensures that all ticks are strictly positive. For all other tasks, it delegates to the base `Locator` (which may be different depending on the scale of the *r*-axis.

class `matplotlib.projections.polar.`**ThetaFormatter**

Bases: `matplotlib.ticker.Formatter`

Used to format the *theta* tick labels. Converts the native unit of radians into degrees and adds a degree symbol.

CHAPTER

SIXTYEIGHT

PYPLOT

68.1 matplotlib.pyplot

Provides a MATLAB-like plotting framework.

pylab combines pyplot with numpy into a single namespace. This is convenient for interactive work, but for programming it is recommended that the namespaces be kept separate, e.g.:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5, 0.1);
y = np.sin(x)
plt.plot(x, y)
```

matplotlib.pyplot.**acorr**(*x*, *hold=None*, *data=None*, ***kwargs*)

Plot the autocorrelation of x.

Parameters x : sequence of scalar

hold : boolean, optional, default: True

detrend : callable, optional, default: mlab.detrend_none
x is detrended by the detrend callable. Default is no normalization.

normed : boolean, optional, default: True
if True, normalize the data by the autocorrelation at the 0-th lag.

usevlines : boolean, optional, default: True
if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.

maxlags : integer, optional, default: 10
number of lags to show. If None, will return all 2 * len(x) - 1 lags.

Returns (lags, c, line, b) : where:

- lags are a length 2‘maxlags+1 lag vector.
- c is the 2‘maxlags+1 auto correlation vectorI
- line is a *Line2D* instance returned by *plot*.
- b is the x-axis.

Other Parameters linestyle : *Line2D* prop, optional, default: None
Only used if usevlines is False.

marker : string, optional, default: ‘o’

Notes

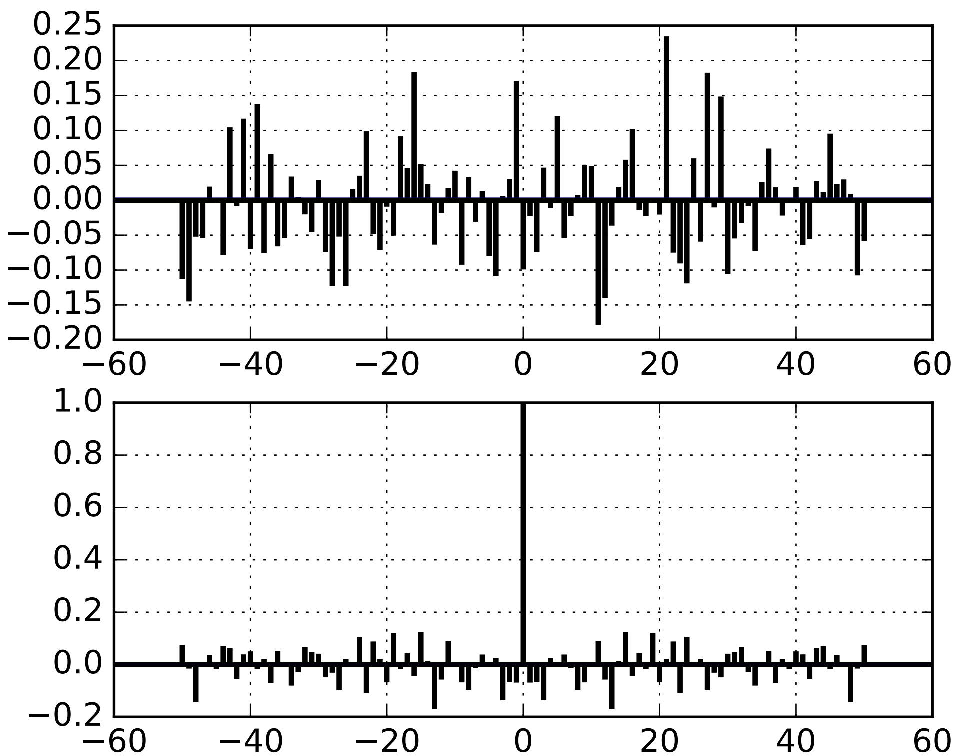
In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

Examples

xcorr is top graph, and *acorr* is bottom graph.



`matplotlib.pyplot.angle_spectrum`(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *hold=None*, *data=None*, ***kwargs*)

Plot the angle spectrum.

Call signature:

```
angle_spectrum(x, Fs=2, Fc=0,  window=mlab.window_hanning,
               pad_to=None, sides='default', **kwargs)
```

Compute the angle spectrum (wrapped phase spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer**

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns the tuple (*spectrum*, *freqs*, *line*):

***spectrum*: 1-D array** The values for the angle spectrum in radians (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

***line*: a `Line2D` instance** The line created by this function

kwargs control the `Line2D` properties:

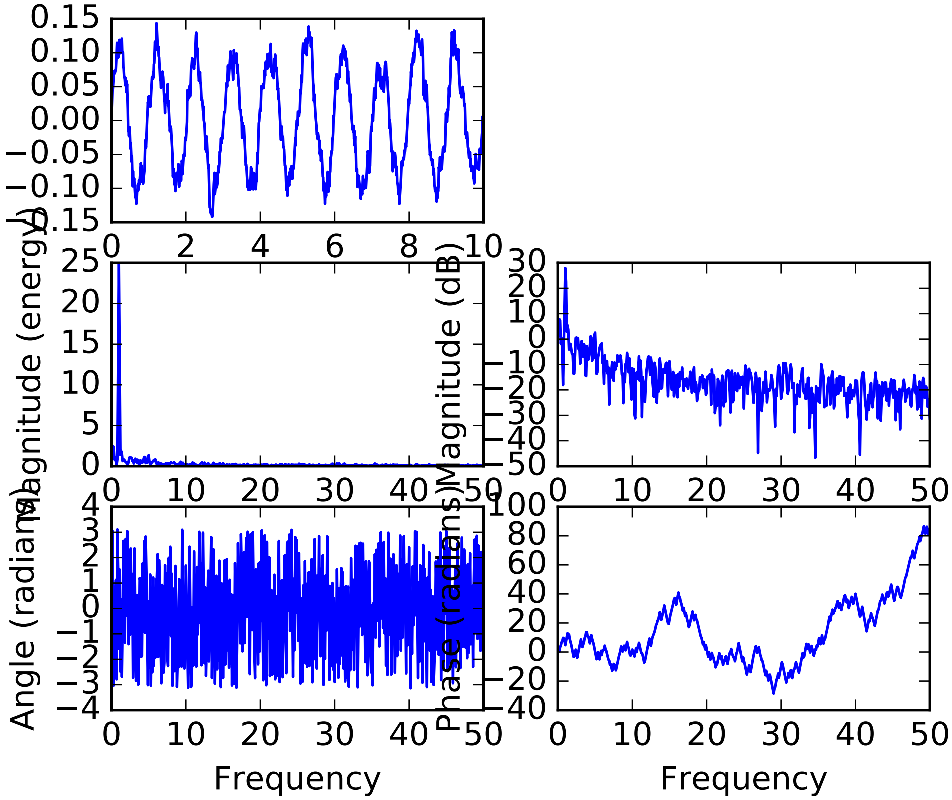
Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']

C

Table 68.1 – continued from previous page

Property	Description
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '
linewidth or lw	float value in points
marker	A valid marker style
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:



See also:

[`magnitude_spectrum()`] *`angle_spectrum()`* plots the magnitudes of the corresponding frequencies.

[`phase_spectrum()`] *`phase_spectrum()`* plots the unwrapped version of this function.

[`specgram()`] *`specgram()`* can plot the angle spectrum of segments within the signal in a colormap.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`annotate`**`(`**args*, ***kwargs*`)`

Annotate the point `xy` with text `s`.

Additional kwargs are passed to *`Text`*.

Parameters s : str

The text of the annotation

xy : iterable

Length 2 sequence specifying the *(x,y)* point to annotate

xytext : iterable, optional

Length 2 sequence specifying the *(x,y)* to place the text at. If None, defaults to `xy`.

xycoords : str, Artist, Transform, callable or tuple, optional
The coordinate system that `xy` is given in.

For a `str` the allowed values are:

Property	Description
'figure points'	points from the lower left of the figure
'figure pixels'	pixels from the lower left of the figure
'figure fraction'	fraction of figure from lower left
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	fraction of axes from lower left
'data'	use the coordinate system of the object being annotated (default)
'polar'	*(theta,r)* if not native 'data' coordinates

If a `Artist` object is passed in the units are fraction if it's bounding box.

If a `Transform` object is passed in use that to transform `xy` to screen coordinates

If a callable it must take a `RendererBase` object as input and return a `Transform` or `Bbox` object

If a `tuple` must be length 2 tuple of str, `Artist`, `Transform` or callable objects. The first transform is used for the *x* coordinate and the second for *y*.

See *Annotating Axes* for more details.

Defaults to `'data'`

textcoords : str, `Artist`, `Transform`, callable or tuple, optional
The coordinate system that `xytext` is given, which may be different than the coordinate system used for `xy`.

All `xycoords` values are valid as well as the following strings:

Property	Description
'offset points'	offset (in points) from the *xy* value
'offset pixels'	offset (in pixels) from the *xy* value

defaults to the input of `xycoords`

arrowprops : dict, optional
If not None, properties used to draw a `FancyArrowPatch` arrow between `xy` and `xytext`.

If `arrowprops` does not contain the key `'arrowstyle'` the allowed keys are:

Key	Description
width	the width of the arrow in points
headwidth	the width of the base of the arrow head in points
headlength	the length of the arrow head in points
shrink	fraction of total length to 'shrink' from both ends
?	any key to `matplotlib.patches.FancyArrowPatch`

If the `arrowprops` contains the key `'arrowstyle'` the above keys are forbidden. The allowed values of `'arrowstyle'` are:

Name	Attrs
`'-'`	None
`'->'`	head_length=0.4,head_width=0.2
`'-['`	widthB=1.0,lengthB=0.2,angleB=None
`'|-|'`	widthA=1.0,widthB=1.0
`'-|>'`	head_length=0.4,head_width=0.2
`'<-'`	head_length=0.4,head_width=0.2
`'<->'`	head_length=0.4,head_width=0.2
`'<|-'`	head_length=0.4,head_width=0.2
`'<|-|>'`	head_length=0.4,head_width=0.2
`'fancy'`	head_length=0.4,head_width=0.4,tail_width=0.4
`'simple'`	head_length=0.5,head_width=0.5,tail_width=0.2
`'wedge'`	tail_width=0.3,shrink_factor=0.5

Valid keys for `FancyArrowPatch` are:

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
mutation_aspect	default is 1.
?	any key for `matplotlib.patches.PathPatch`

Defaults to None

annotation_clip : bool, optional

Controls the visibility of the annotation when it goes outside the axes area.

If `True`, the annotation will only be drawn when the `xy` is inside the axes. If `False`, the annotation will always be drawn regardless of its position.

The default is `None`, which behave as `True` only if *xycoords* is "data".

Returns Annotation

`matplotlib.pyplot.arrow`(*x*, *y*, *dx*, *dy*, *hold=None*, ***kwargs*)

Add an arrow to the axes.

Call signature:

```
arrow(x, y, dx, dy, **kwargs)
```

Draws arrow on specified axis from (x, y) to ($x + dx$, $y + dy$). Uses FancyArrow patch to construct the arrow.

The resulting arrow is affected by the axes aspect ratio and limits. This may produce an arrow whose head is not square with its stem. To create an arrow whose head is square with its stem, use `annotate()` for example:

```
ax.annotate("", xy=(0.5, 0.5), xytext=(0, 0),
    arrowprops=dict(arrowstyle="->"))
```

Optional kwargs control the arrow construction and properties:

Constructor arguments

***width*: float (default: 0.001)** width of full arrow tail

***length_includes_head*: [True | False] (default: False)** True if head is to be counted in calculating the length.

***head_width*: float or None (default: 3*width)** total width of the full arrow head

***head_length*: float or None (default: 1.5 * head_width)** length of arrow head

***shape*: ['full', 'left', 'right'] (default: 'full')** draw the left-half, right-half, or full arrow

***overhang*: float (default: 0)** fraction that the arrow is swept back (0 overhang means triangular shape). Can be negative or greater than one.

***head_starts_at_zero*: [True | False] (default: False)** if True, the head starts being drawn at coordinate 0 instead of ending at coordinate 0.

Other valid kwargs (inherited from `Patch`) are:

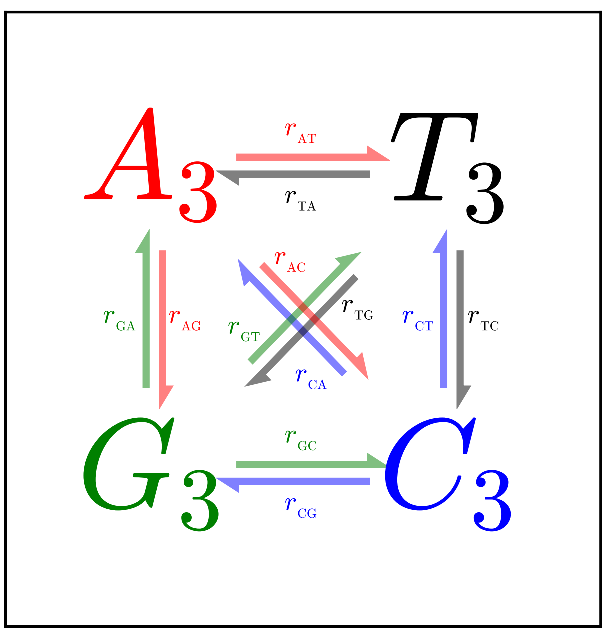
Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color

Continued on

Table 68.2 – continued from previous page

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>facecolor or fc</td><td>mpl color spec, or None for default, or ‘none’ for no color</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fill</td><td>[True | False]</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>hatch</td><td>[’/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]</td></tr>
<tr><td>joinstyle</td><td>[’miter’ | ‘round’ | ‘bevel’]</td></tr>
<tr><td>label</td><td>string or anything printable with ‘%s’ conversion.</td></tr>
<tr><td>linestyle or ls</td><td>[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> | <code>':'</code> | <code>'None'</code></td></tr>
<tr><td>linewidth or lw</td><td>float or None for default</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>transform</td><td>Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

Example:



Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.autoscale`(*enable=True*, *axis='both'*, *tight=None*)

Autoscale the axis view to the data (toggle).

Convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes.

***enable*: [True | False | None]** True (default) turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.

***axis*: ['x' | 'y' | 'both']** which axis to operate on; default is 'both'

***tight*: [True | False | None]** If True, set view limits to data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only artist is an image, otherwise treat *tight* as False. The *tight* setting is retained for future autoscaling until it is explicitly changed.

Returns None.

`matplotlib.pyplot.autumn`()

set the default colormap to autumn and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.axes`(**args*, ***kwargs*)

Add an axes to the figure.

The axes is added at position *rect* specified by:

- `axes()` by itself creates a default full `subplot(111)` window axis.
- `axes(rect, axisbg='w')` where *rect* = [left, bottom, width, height] in normalized (0, 1) units. *axisbg* is the background color for the axis, default white.
- `axes(h)` where *h* is an axes instance makes *h* the current axis. An `Axes` instance is returned.

kwarg	Accepts	Description
axisbg	color	the axes background color
frameon	[True|False]	display the frame?
sharex	otherax	current axes shares xaxis attribute with otherax
sharey	otherax	current axes shares yaxis attribute with otherax
polar	[True|False]	use a polar axes?
aspect	[str | num]	['equal', 'auto'] or a number. If a number the ratio of x-unit/y-unit in screen-space. Also see `set_aspect()`.

Examples:

- `examples/pylab_examples/axes_demo.py` places custom axes.
- `examples/pylab_examples/shared_axis_demo.py` uses *sharex* and *sharey*.

`matplotlib.pyplot.axhline`(*y=0*, *xmin=0*, *xmax=1*, *hold=None*, ***kwargs*)

Add a horizontal line across the axis.

Parameters y : scalar, optional, default: 0

y position in data coordinates of the horizontal line.

xmin : scalar, optional, default: 0

Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

xmax : scalar, optional, default: 1

Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.

Returns `Line2D`

See also:

axhspan for example plot and source code

`Additional`

Notes

kwargs are passed to *Line2D* and can be used to control the line properties.

Examples

- draw a thick red hline at 'y' = 0 that spans the xrange:

```
>>> axhline(linewidth=4, color='r')
```

- draw a default hline at 'y' = 1 that spans the xrange:

```
>>> axhline(y=1)
```

- draw a default hline at 'y' = .5 that spans the middle half of the xrange:

```
>>> axhline(y=.5, xmin=0.25, xmax=0.75)
```

Valid kwargs are *Line2D* properties, with the exception of 'transform':

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an *Axes* instance
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a *matplotlib.figure.Figure* instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
linewidth or lw	float value in points
marker	*A valid marker style*

C

Table 68.3 – continued from previous page

Property	Description
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a *matplotlib.transforms.Transform* instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

`matplotlib.pyplot.`**`axhspan`**(*ymin*, *ymax*, *xmin=0*, *xmax=1*, *hold=None*, ***kwargs*)

Add a horizontal span (rectangle) across the axis.

Call signature:

```
axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
```

y coords are in data units and *x* coords are in axes (relative 0-1) units.

Draw a horizontal span (rectangle) from *ymin* to *ymax*. With the default values of *xmin* = 0 and *xmax* = 1, this always spans the xrange, regardless of the xlim settings, even if you change them, e.g., with the `set_xlim()` command. That is, the horizontal extent is in axes coords: 0=left, 0.5=middle, 1.0=right but the *y* location is in data coordinates.

Return value is a *matplotlib.patches.Polygon* instance.

Examples:

- draw a gray rectangle from *y* = 0.25-0.75 that spans the horizontal extent of the axes:

```
>>> axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)
```

Valid kwargs are *Polygon* properties:

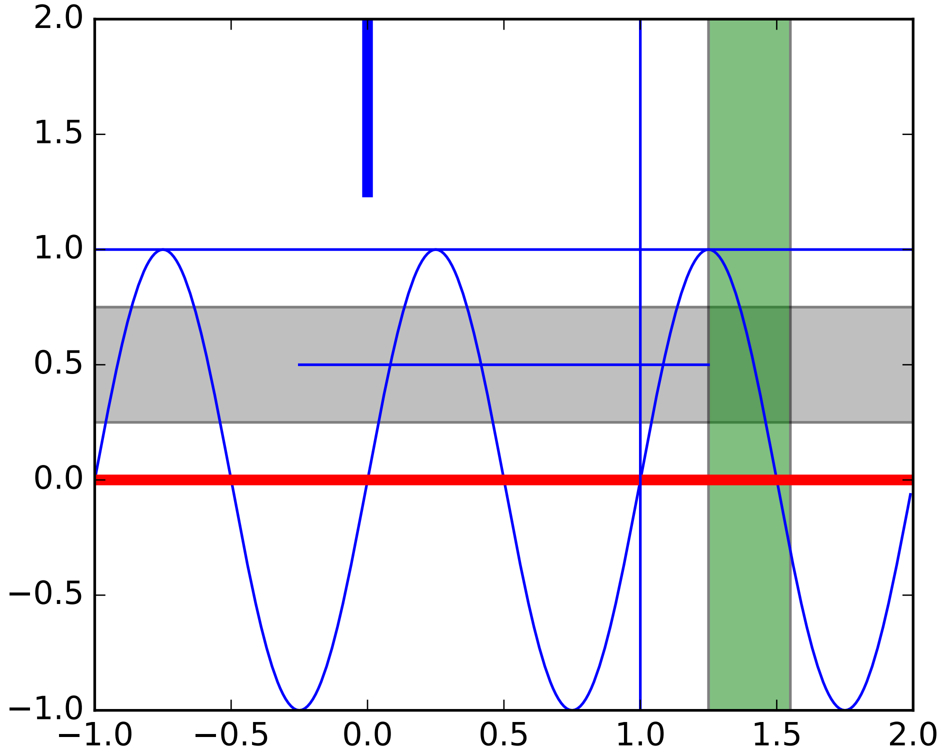
Property	Description
agg_filter	unknown

Continued on

Table 68.4 – continued from previous page

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>alpha</td><td>float or None</td></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or aa</td><td>[True | False] or None for default</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color</td><td>matplotlib color spec</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or ec</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td>facecolor or fc</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fill</td><td>[True | False]</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>hatch</td><td>['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']</td></tr>
<tr><td>joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'</td></tr>
<tr><td>linewidth or lw</td><td>float or None for default</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>transform</td><td>Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

Example:



Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.axis(*v, **kwargs)`

Convenience method to get or set axis properties.

Calling with no arguments:

```
>>> axis()
```

returns the current axes limits `[xmin, xmax, ymin, ymax]`.:

```
>>> axis(v)
```

sets the min and max of the x and y axes, with `v = [xmin, xmax, ymin, ymax]`.:

```
>>> axis('off')
```

turns off the axis lines and labels.:

```
>>> axis('equal')
```

changes limits of x or y axis so that equal increments of x and y have the same length; a circle is circular.:

```
>>> axis('scaled')
```

achieves the same result by changing the dimensions of the plot box instead of the axis data limits.:

```
>>> axis('tight')
```

changes *x* and *y* axis limits such that all data is shown. If all data is already shown, it will move it to the center of the figure without modifying (*xmax* - *xmin*) or (*ymax* - *ymin*). Note this is slightly different than in MATLAB.:

```
>>> axis('image')
```

is 'scaled' with the axis limits equal to the data limits.:

```
>>> axis('auto')
```

and:

```
>>> axis('normal')
```

are deprecated. They restore default behavior; axis limits are automatically scaled to make the data fit comfortably within the plot box.

if `len(*v)==0`, you can pass in *xmin*, *xmax*, *ymin*, *ymax* as kwargs selectively to alter just those limits without changing the others.

```
>>> axis('square')
```

changes the limit ranges (*xmax-xmin*) and (*ymax-ymin*) of the *x* and *y* axes to be the same, and have the same scaling, resulting in a square plot.

The xmin, xmax, ymin, ymax tuple is returned

See also:

`xlim()`, `ylim()` For setting the x- and y-limits individually.

`matplotlib.pyplot.`**`axvline`**(*x=0*, *ymin=0*, *ymax=1*, *hold=None*, ***kwargs*)

Add a vertical line across the axes.

Parameters x : scalar, optional, default: 0

x position in data coordinates of the vertical line.

ymin : scalar, optional, default: 0

Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

ymax : scalar, optional, default: 1

Should be between 0 and 1, 0 being the bottom of the plot, 1 the top of the plot.

Returns `Line2D`

See also:

`axhspan` for example plot and source code

`Additional`

Examples

- draw a thick red vline at $x = 0$ that spans the yrange:

```
>>> axvline(linewidth=4, color='r')
```

- draw a default vline at $x = 1$ that spans the yrange:

```
>>> axvline(x=1)
```

- draw a default vline at $x = .5$ that spans the middle half of the yrange:

```
>>> axvline(x=.5, ymin=0.25, ymax=0.75)
```

Valid kwargs are *Line2D* properties, with the exception of 'transform':

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>agg_filter</td><td>unknown</td></tr>
<tr><td>alpha</td><td>float (0.0 transparent through 1.0 opaque)</td></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or aa</td><td>[True | False]</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color or c</td><td>any matplotlib color</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>dash_capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>dash_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>dashes</td><td>sequence of on/off ink in points</td></tr>
<tr><td>drawstyle</td><td>['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> | <code>'</code></td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>marker</td><td>A valid marker style</td></tr>
<tr><td>markeredgecolor or mec</td><td>any matplotlib color</td></tr>
<tr><td>markeredgewidth or mew</td><td>float value in points</td></tr>
<tr><td>markerfacecolor or mfc</td><td>any matplotlib color</td></tr>
<tr><td>markerfacecoloralt or mfcalt</td><td>any matplotlib color</td></tr>
<tr><td>markersize or ms</td><td>float</td></tr>
<tr><td>markevery</td><td>[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>float distance in points or callable pick function <code>fn(artist, event)</code></td></tr>
<tr><td>pickradius</td><td>float distance in points</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td colspan="2">C</td></tr>
</table>

Table 68.5 – continued from previous page

Property	Description
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

matplotlib.pyplot.**axvspan**(*xmin*, *xmax*, *ymin=0*, *ymax=1*, *hold=None*, ***kwargs*)

Add a vertical span (rectangle) across the axes.

Call signature:

```
axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
```

x coords are in data units and *y* coords are in axes (relative 0-1) units.

Draw a vertical span (rectangle) from *xmin* to *xmax*. With the default values of *ymin* = 0 and *ymax* = 1, this always spans the yrange, regardless of the ylim settings, even if you change them, e.g., with the set_ylim() command. That is, the vertical extent is in axes coords: 0=bottom, 0.5=middle, 1.0=top but the *y* location is in data coordinates.

Return value is the matplotlib.patches.Polygon instance.

Examples:

- draw a vertical green translucent rectangle from x=1.25 to 1.55 that spans the yrange of the axes:

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

Valid kwargs are Polygon properties:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an Axes instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec

Continued on

Table 68.6 – continued from previous page

Property	Description
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

See also:

`axhspan()` for example plot and source code

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.bar`(*left*, *height*, *width=0.8*, *bottom=None*, *hold=None*, *data=None*, ***kwargs*)

Make a bar plot.

Make a bar plot with rectangles bounded by:

`left, left + width, bottom, bottom + height` (left, right, bottom and top edges)

Parameters left : sequence of scalars

the x coordinates of the left sides of the bars

height : sequence of scalars

the heights of the bars

width : scalar or array-like, optional

the width(s) of the bars default: 0.8

bottom : scalar or array-like, optional

the y coordinate(s) of the bars default: None

color : scalar or array-like, optional

the colors of the bar faces

edgecolor : scalar or array-like, optional

the colors of the bar edges

linewidth : scalar or array-like, optional
width of bar edge(s). If None, use default linewidth; If 0, don't draw edges. default: None

tick_label : string or array-like, optional
the tick labels of the bars default: None

xerr : scalar or array-like, optional
if not None, will be used to generate errorbar(s) on the bar chart default: None

yerr : scalar or array-like, optional
if not None, will be used to generate errorbar(s) on the bar chart default: None

ecolor : scalar or array-like, optional
specifies the color of errorbar(s) default: None

capsize : scalar, optional
determines the length in points of the error bar caps default: None, which will take the value from the `errorbar.capsize` *rcParam*.

error_kw : dict, optional
dictionary of kwargs to be passed to errorbar method. *ecolor* and *capsize* may be specified here rather than as independent kwargs.

align : {'edge', 'center'}, optional
If 'edge', aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If 'center', interpret the `left` argument as the coordinates of the centers of the bars. To align on the align bars on the right edge pass a negative `width`.

orientation : {'vertical', 'horizontal'}, optional
The orientation of the bars.

log : boolean, optional
If true, sets the axis to be log scale. default: False

Returns bars : matplotlib.container.BarContainer
Container with all of the bars + errorbars

See also:

barh Plot a horizontal bar plot.

Notes

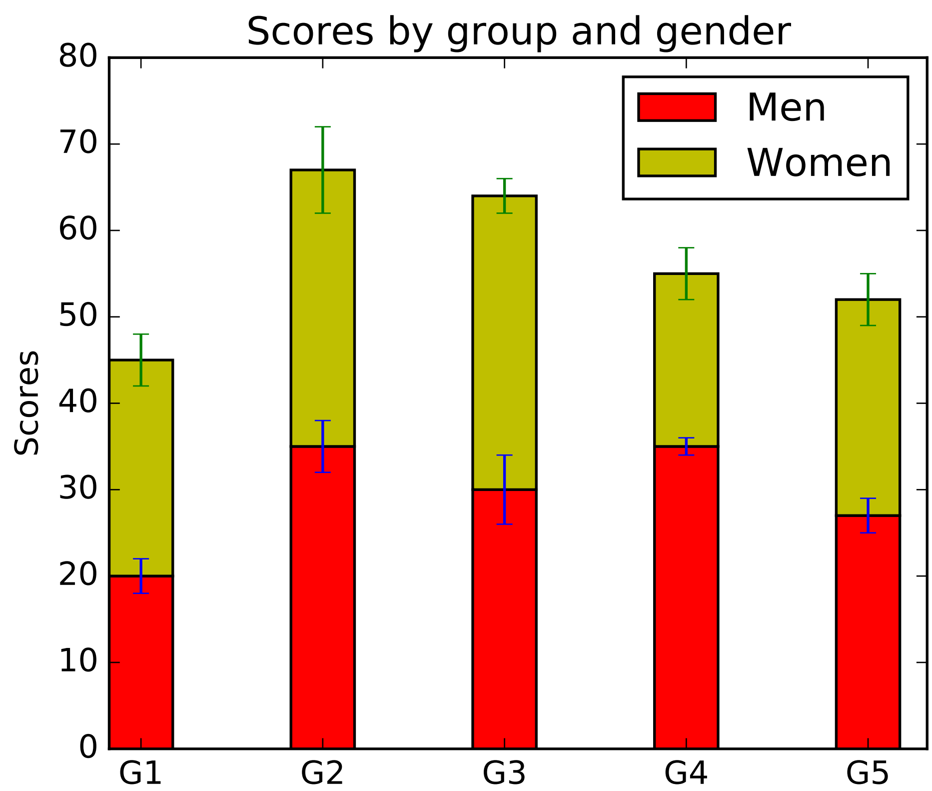
In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'height', 'yerr', 'tick_label', 'xerr', 'width', 'color', 'ecolor', 'linewidth', 'left', 'bottom', 'edgecolor'.

Additional kwargs: hold = [True|False] overrides default hold state

Examples

Example: A stacked bar chart.



matplotlib.pyplot.**barbs**(**args*, ***kw*)

Plot a 2-D field of barbs.

Call signatures:

```
barb(U, V, **kw)
barb(U, V, C, **kw)
barb(X, Y, U, V, **kw)
barb(X, Y, U, V, C, **kw)
```

Arguments:

***X*, *Y*:** The x and y coordinates of the barb locations (default is head of barb; see *pivot* kwarg)

***U*, *V*:** Give the x and y components of the barb shaft

***C*:** An optional array used to map colors to the barbs

All arguments may be 1-D or 2-D arrays or sequences. If *X* and *Y* are absent, they will be generated as a uniform grid. If *U* and *V* are 2-D arrays but *X* and *Y* are 1-D, and if `len(X)` and `len(Y)` match the column and row dimensions of *U*, then *X* and *Y* will be expanded with `numpy.meshgrid()`.

U, *V*, *C* may be masked arrays, but masked *X*, *Y* are not supported at present.

Keyword arguments:

***length*:** Length of the barb in points; the other parts of the barb are scaled against this. Default is 9

***pivot*: ['tip' | 'middle']** The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*. Default is 'tip'

***barbcolor*: [color | color sequence]** Specifies the color all parts of the barb except any flags. This parameter is analagous to the *edgecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor.

***flagcolor*: [color | color sequence]** Specifies the color of any flags on the barb. This parameter is analagous to the *facecolor* parameter for polygons, which can be used instead. However this parameter will override facecolor. If this is not set (and *C* has not either) then *flagcolor* will be set to match *barbcolor* so that the barb has a uniform color. If *C* has been set, *flagcolor* has no effect.

***sizes*:** A dictionary of coefficients specifying the ratio of a given feature to the length of the barb. Only those values one wishes to override need to be included. These features include:

- 'spacing' - space between features (flags, full/half barbs)
- 'height' - height (distance from shaft to top) of a flag or full barb
- 'width' - width of a flag, twice the width of a full barb
- 'emptybarb' - radius of the circle used for low magnitudes

***fill_empty*:** A flag on whether the empty barbs (circles) that are drawn should be filled with the flag color. If they are not filled, they will be drawn such that no color is applied to the center. Default is False

***rounding*:** A flag to indicate whether the vector magnitude should be rounded when allocating barb components. If True, the magnitude is rounded to the nearest multiple of the half-barb increment. If False, the magnitude is simply truncated to the next lowest multiple. Default is True

***barb_increments*:** A dictionary of increments specifying values to associate with different parts of the barb. Only those values one wishes to override need to be included.

- 'half' - half barbs (Default is 5)
- 'full' - full barbs (Default is 10)
- 'flag' - flags (default is 50)

***flip_barb*:** Either a single boolean flag or an array of booleans. Single boolean indicates whether the lines and flags should point opposite to normal for all barbs. An array (which should be the same size as the other data arrays) indicates whether to flip for each individual barb. Normal behavior is for the barbs and lines to point right (comes from wind barbs having these features point towards low pressure in the Northern Hemisphere.) Default is False

Barbs are traditionally used in meteorology as a way to plot the speed and direction of wind observations, but can technically be used to plot any two dimensional vector quantity. As opposed to arrows, which give vector magnitude by the length of the arrow, the barbs give more quantitative information about the vector magnitude by putting slanted lines or a triangle for various increments in magnitude, as show schematically below:

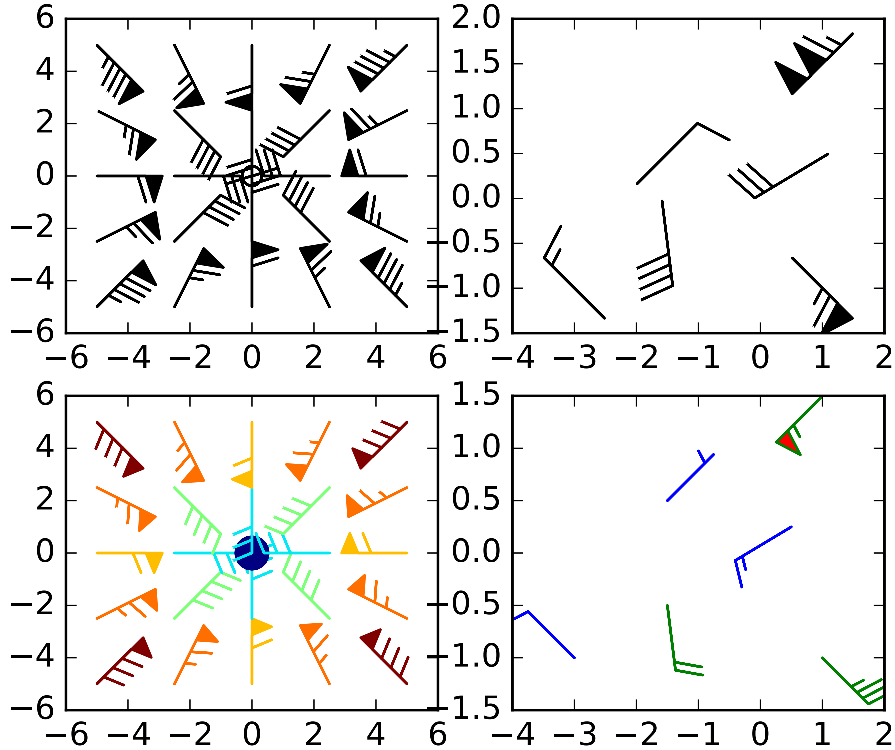
```
:     /\    \
:    /  \    \
:   /    \    \    \
:  /      \    \    \
: ------------------------------
```

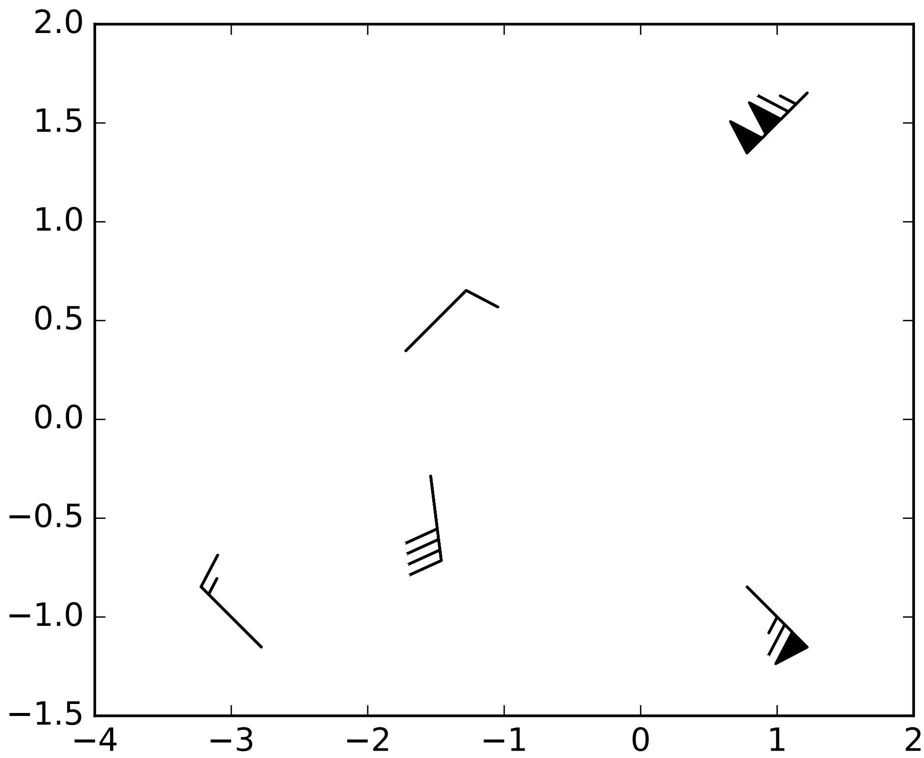
The largest increment is given by a triangle (or "flag"). After those come full lines (barbs). The smallest increment is a half line. There is only, of course, ever at most 1 half line. If the magnitude is small and only needs a single half-line and no full lines or triangles, the half-line is offset from the end of the barb so that it can be easily distinguished from barbs with a single full line. The magnitude for the barb shown above would nominally be 65, using the standard increments of 50, 10, and 5.

linewidths and edgecolors can be used to customize the barb. Additional `PolyCollection` keyword arguments:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color spec or sequence of specs
`facecolor` or facecolors	matplotlib color spec or sequence of specs
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`label`	string or anything printable with '%s' conversion.
`linestyle` or linestyles or dashes	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` |
`linewidth` or lw or linewidths	float or sequence of floats
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`pickradius`	unknown
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`urls`	unknown
`visible`	[True | False]
`zorder`	any number

Example:


6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4 −3 −2 −1 0 1 2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`barh`**(*bottom*, *width*, *height=0.8*, *left=None*, *hold=None*, ***kwargs*)

Make a horizontal bar plot.

Make a horizontal bar plot with rectangles bounded by:

`left, left + width, bottom, bottom + height` (left, right, bottom and top edges)

`bottom`, `width`, `height`, and `left` can be either scalars or sequences

Parameters bottom : scalar or array-like
the y coordinate(s) of the bars

width : scalar or array-like
the width(s) of the bars

height : sequence of scalars, optional, default: 0.8
the heights of the bars

left : sequence of scalars
the x coordinates of the left sides of the bars

Returns *`matplotlib.patches.Rectangle`* instances.

Other Parameters color : scalar or array-like, optional the colors of the bars

edgecolor : scalar or array-like, optional
    the colors of the bar edges

linewidth : scalar or array-like, optional, default: None
    width of bar edge(s). If None, use default linewidth; If 0, don't draw edges.

tick_label : string or array-like, optional, default: None
    the tick labels of the bars

xerr : scalar or array-like, optional, default: None
    if not None, will be used to generate errorbar(s) on the bar chart

yerr : scalar or array-like, optional, default: None
    if not None, will be used to generate errorbar(s) on the bar chart

ecolor : scalar or array-like, optional, default: None
    specifies the color of errorbar(s)

capsize : scalar, optional
    determines the length in points of the error bar caps default: None, which will take the value from the `errorbar.capsize` *rcParam*.

error_kw :
    dictionary of kwargs to be passed to errorbar method. `ecolor` and `capsize` may be specified here rather than as independent kwargs.

align : ['edge' | 'center'], optional, default: 'edge'
    If `edge`, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If `center`, interpret the `left` argument as the coordinates of the centers of the bars.

log : boolean, optional, default: False
    If true, sets the axis to be log scale

See also:

bar Plot a vertical bar plot.

`Additional`

Notes

The optional arguments `color`, `edgecolor`, `linewidth`, `xerr`, and `yerr` can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: `xerr` and `yerr` are passed directly to `errorbar()`, so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an *Axes* instance

Continued on

Table 68.8 – continued from previous page

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color</td><td>matplotlib color spec</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or ec</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td>facecolor or fc</td><td>mpl color spec, or None for default, or 'none' for no color</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fill</td><td>[True | False]</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>hatch</td><td>['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']</td></tr>
<tr><td>joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> | <code>':'</code> | <code>'None'</code></td></tr>
<tr><td>linewidth or lw</td><td>float or None for default</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>transform</td><td>Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

`matplotlib.pyplot.`**`bone`**`()`

set the default colormap to bone and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.`**`box`**`(`*on=None*`)`

Turn the axes box on or off. *on* may be a boolean or a string, 'on' or 'off'.

If *on* is *None*, toggle state.

`matplotlib.pyplot.`**`boxplot`**`(`*x, notch=None, sym=None, vert=None, whis=None, positions=None, widths=None, patch_artist=None, bootstrap=None, usermedians=None, conf_intervals=None, meanline=None, showmeans=None, showcaps=None, showbox=None, showfliers=None, boxprops=None, labels=None, flierprops=None, medianprops=None, meanprops=None, capprops=None, whiskerprops=None, manage_xticks=True, hold=None, data=None*`)`

Make a box and whisker plot.

Call signature:

```
boxplot(self, x, notch=None, sym=None, vert=None, whis=None,
        positions=None, widths=None, patch_artist=False,
        bootstrap=None, usermedians=None, conf_intervals=None,
        meanline=False, showmeans=False, showcaps=True,
        showbox=True, showfliers=True, boxprops=None,
        labels=None, flierprops=None, medianprops=None,
        meanprops=None, capprops=None, whiskerprops=None,
        manage_xticks=True, autorange=False):
```

Make a box and whisker plot for each column of `x` or each vector in sequence `x`. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

Parameters **x** : Array or a sequence of vectors.

The input data.

notch : bool, optional (False)

If `True`, will produce a notched box plot. Otherwise, a rectangular boxplot is produced.

sym : str, optional

The default symbol for flier points. Enter an empty string (‘’) if you don’t want to show fliers. If `None`, then the fliers default to ‘b+’ If you want more control use the flierprops kwarg.

vert : bool, optional (True)

If `True` (default), makes the boxes vertical. If `False`, everything is drawn horizontally.

whis : float, sequence, or string (default = 1.5)

As a float, determines the reach of the whiskers past the first and third quartiles (e.g., Q3 + whis*IQR, IQR = interquartile range, Q3-Q1). Beyond the whiskers, data are considered outliers and are plotted as individual points. Set this to an unreasonably high value to force the whiskers to show the min and max values. Alternatively, set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, `whis` can be the string `'range'` to force the whiskers to the min and max of the data.

bootstrap : int, optional

Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If `bootstrap` is None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine its 95% confidence intervals. Values between 1000 and 10000 are recommended.

usermedians : array-like, optional

An array or sequence whose first dimension (or length) is compatible with `x`. This overrides the medians computed by matplotlib for each element of `usermedians` that is not `None`. When an element of `usermedians` is None, the median will be computed by matplotlib as normal.

conf_intervals : array-like, optional

Array or sequence whose first dimension (or length) is compatible with `x` and whose second dimension is 2. When the an element of `conf_intervals` is not None, the notch locations computed by matplotlib are overridden (provided `notch` is `True`). When an element of `conf_intervals` is `None`, the notches are computed by the method specified by the other kwargs (e.g., `bootstrap`).

positions : array-like, optional
Sets the positions of the boxes. The ticks and limits are automatically set to match the positions. Defaults to `range(1, N+1)` where N is the number of boxes to be drawn.

widths : scalar or array-like
Sets the width of each box either with a scalar or a sequence. The default is 0.5, or `0.15*(distance between extreme positions)`, if that is smaller.

patch_artist : bool, optional (False)
If `False` produces boxes with the Line2D artist. Otherwise, boxes and drawn with Patch artists.

labels : sequence, optional
Labels for each dataset. Length must be compatible with dimensions of `x`.

manage_xticks : bool, optional (True)
If the function should adjust the xlim and xtick locations.

autorange : bool, optional (False)
When `True` and the data are distributed such that the 25th and 75th percentiles are equal, `whis` is set to `'range'` such that the whisker ends are at the minimum and maximum of the data.

meanline : bool, optional (False)
If `True` (and `showmeans` is `True`), will try to render the mean as a line spanning the full width of the box according to `meanprops` (see below). Not recommended if `shownotches` is also True. Otherwise, means will be shown as points.

Returns result : dict
A dictionary mapping each component of the boxplot to a list of the *`matplotlib.lines.Line2D`* instances created. That dictionary has the following keys (assuming vertical boxplots):

- `boxes`: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- `medians`: horizontal lines at the median of each box.
- `whiskers`: the vertical lines extending to the most extreme, non-outlier data points.
- `caps`: the horizontal lines at the ends of the whiskers.
- `fliers`: points representing data that extend beyond the whiskers (fliers).
- `means`: points or lines representing the means.

Other Parameters The following boolean options toggle the drawing of individual

components of the boxplots:

- showcaps: the caps on the ends of whiskers (default is True)

- showbox: the central box (default is True)
- showfliers: the outliers beyond the caps (default is True)
- showmeans: the arithmetic means (default is False)

The remaining options can accept dictionaries that specify the

style of the individual artists:

- capprops
- boxprops
- whiskerprops
- flierprops
- medianprops
- meanprops

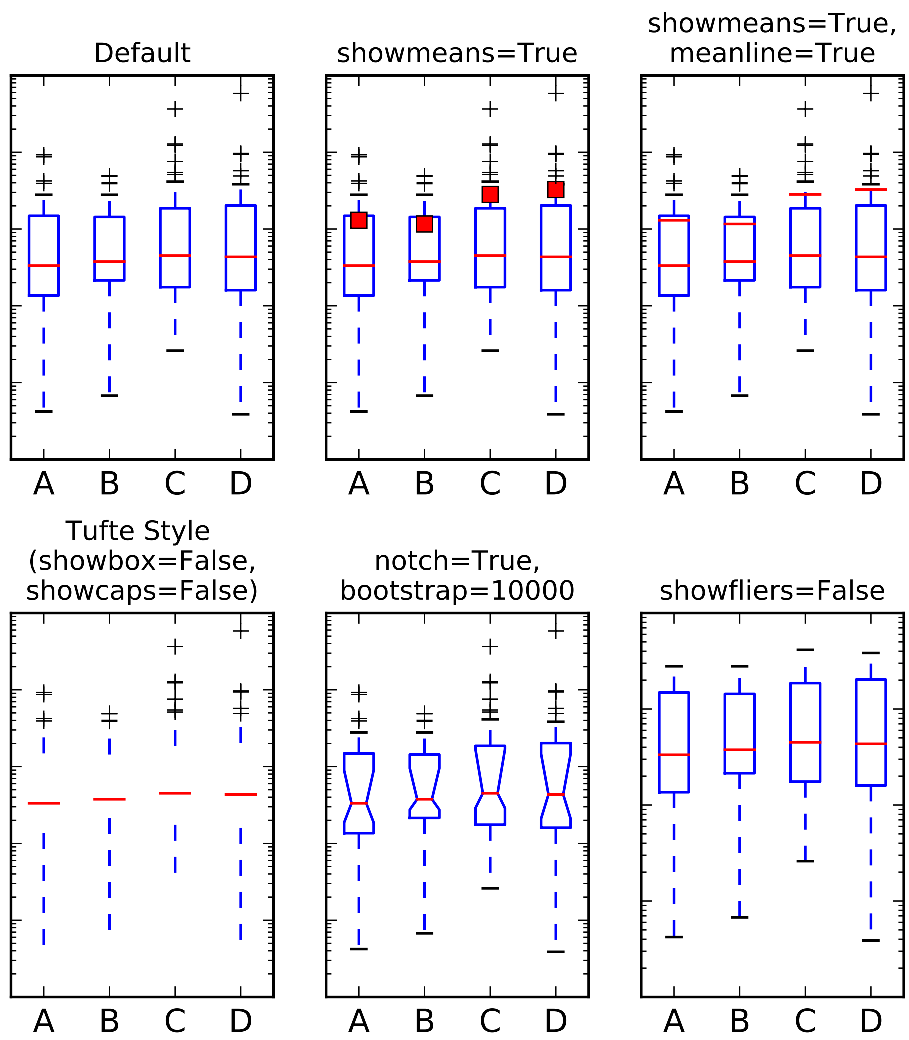
Notes

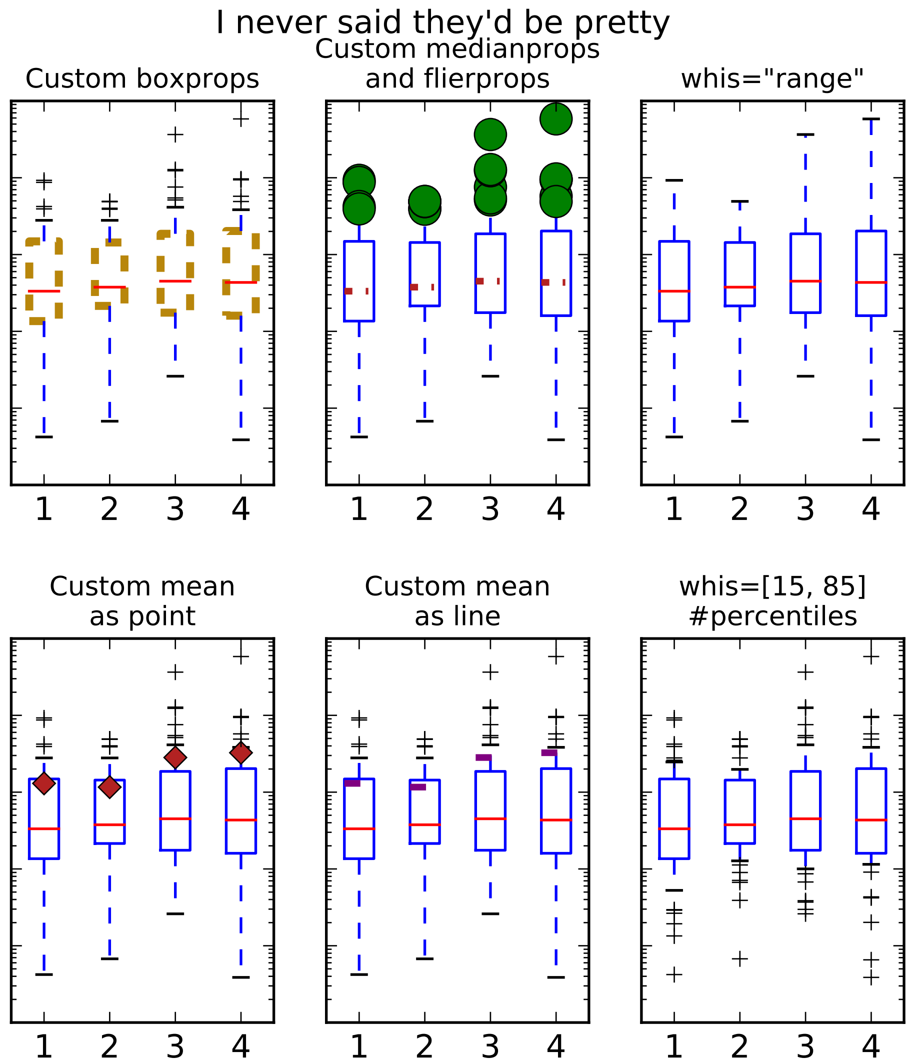
In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

Additional kwargs: hold = [True|False] overrides default hold state

Examples


Default
showmeans=True
showmeans=True,
meanline=True
Tufte Style
(showbox=False,
showcaps=False)
notch=True,
bootstrap=10000
showfliers=False
A B C D



`matplotlib.pyplot.`**`broken_barh`**(*xranges*, *yrange*, *hold=None*, *data=None*, ***kwargs*)

Plot horizontal bars.

Call signature:

```
broken_barh(self, xranges, yrange, **kwargs)
```

A collection of horizontal bars spanning *yrange* with a sequence of *xranges*.

Required arguments:

Argument	Description
xranges	sequence of (*xmin*, *xwidth*)
yrange	sequence of (*ymin*, *ywidth*)

kwargs are `matplotlib.collections.BrokenBarHCollection` properties:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>agg_filter</td><td>unknown</td></tr>
<tr><td>alpha</td><td>float or None</td></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or antialiaseds</td><td>Boolean or sequence of booleans</td></tr>
<tr><td>array</td><td>unknown</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>clim</td><td>a length 2 sequence of floats</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>cmap</td><td>a colormap or registered colormap name</td></tr>
<tr><td>color</td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or edgecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td>facecolor or facecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>hatch</td><td>[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]</td></tr>
<tr><td>label</td><td>string or anything printable with ‘%s’ conversion.</td></tr>
<tr><td>linestyle or linestyles or dashes</td><td>[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | '-' | '--' | '-.' |</td></tr>
<tr><td>linewidth or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td>norm</td><td>unknown</td></tr>
<tr><td>offset_position</td><td>unknown</td></tr>
<tr><td>offsets</td><td>float or sequence of floats</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>pickradius</td><td>unknown</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>transform</td><td>Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>urls</td><td>unknown</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

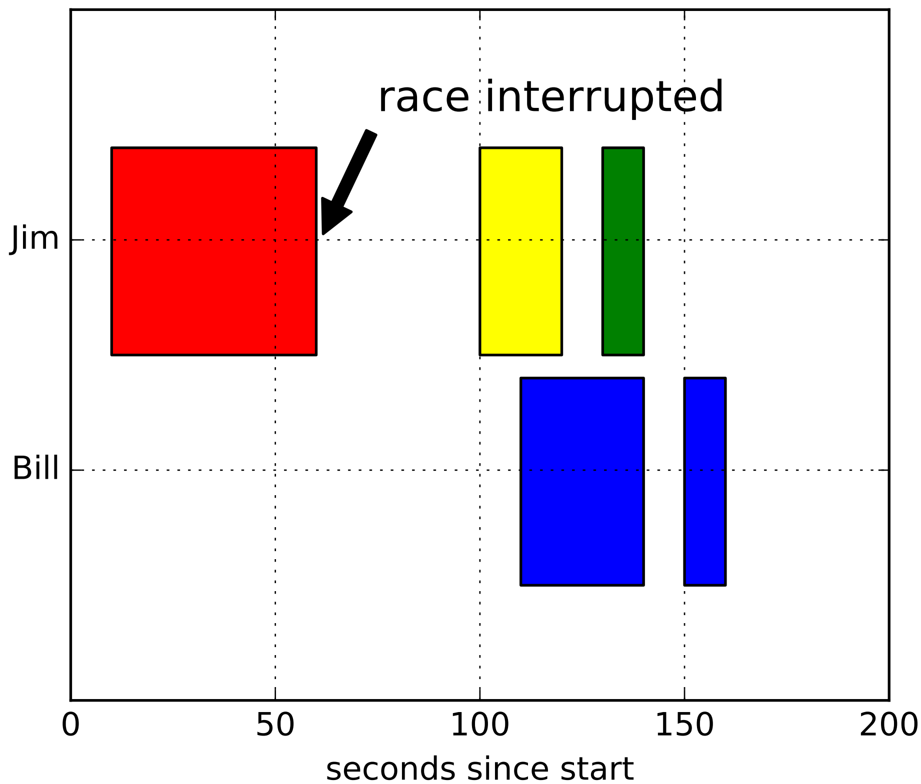
these can either be a single argument, i.e.,:

```
facecolors = 'black'
```

or a sequence of arguments for the various bars, i.e.,:

```
facecolors = ('black', 'red', 'green')
```

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.cla()`

Clear the current axes.

`matplotlib.pyplot.clabel(CS, *args, **kwargs)`

Label a contour plot.

Call signature:

```
clabel(cs, **kwargs)
```

Adds labels to line contours in *cs*, where *cs* is a `ContourSet` object returned by contour.

```
clabel(cs, v, **kwargs)
```

only labels contours listed in *v*.

Optional keyword arguments:

***fontsize*:** size in points or relative size e.g., 'smaller', 'x-large'

***colors*:**

- if *None*, the color of each label matches the color of the corresponding contour
- if one string color, e.g., *colors* = 'r' or *colors* = 'red', all labels will be plotted in this color
- if a tuple of matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified

***inline*:** controls whether the underlying contour is removed or not. Default is *True*.

***inline_spacing*:** space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

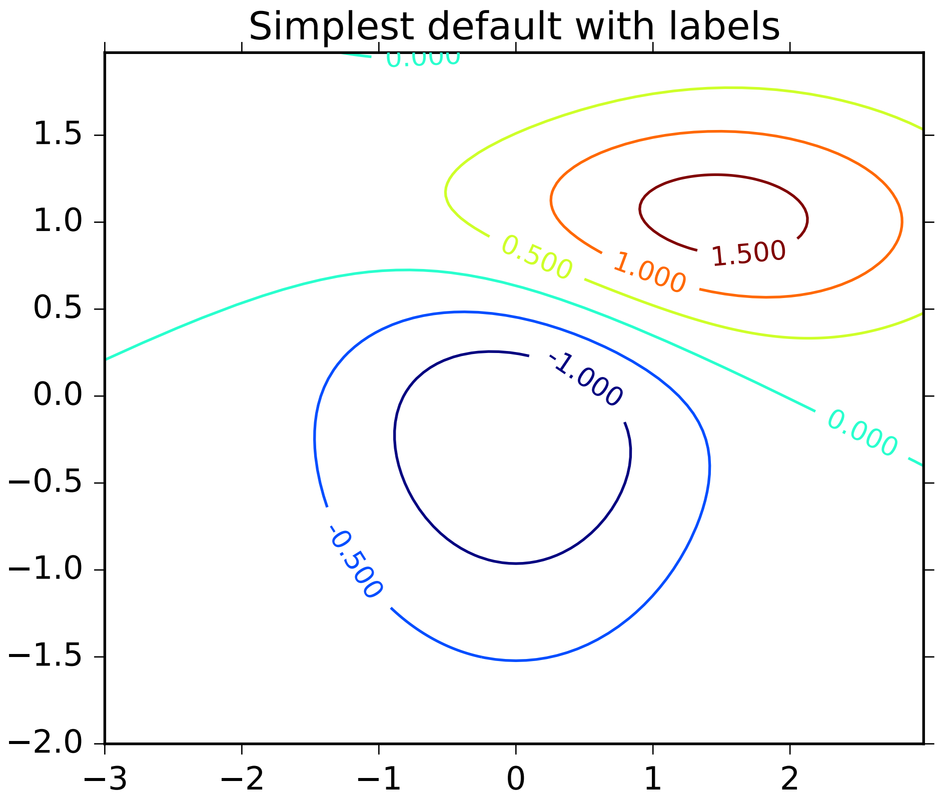
***fmt*:** a format string for the label. Default is '%1.3f' Alternatively, this can be a dictionary matching contour levels with arbitrary strings to use for each contour level (i.e., fmt[level]=string), or it can be any callable, such as a `Formatter` instance, that returns a string when called with a numeric contour level.

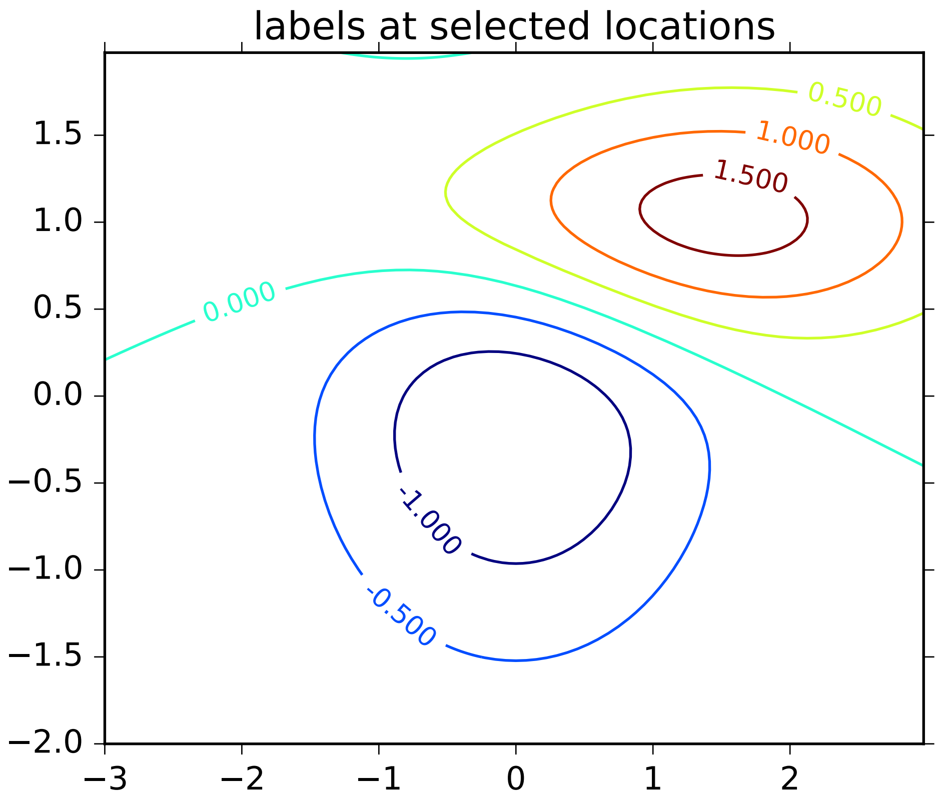
***manual*:** if *True*, contour labels will be placed manually using mouse clicks. Click the first button near a contour to add a label, click the second button (or potentially both mouse buttons at once) to finish adding labels. The third button can be used to remove the last label added, but only if labels are not inline. Alternatively, the keyboard can be used to select label locations (enter to end label placement, delete or backspace act like the third mouse button, and any other key will select a label location).

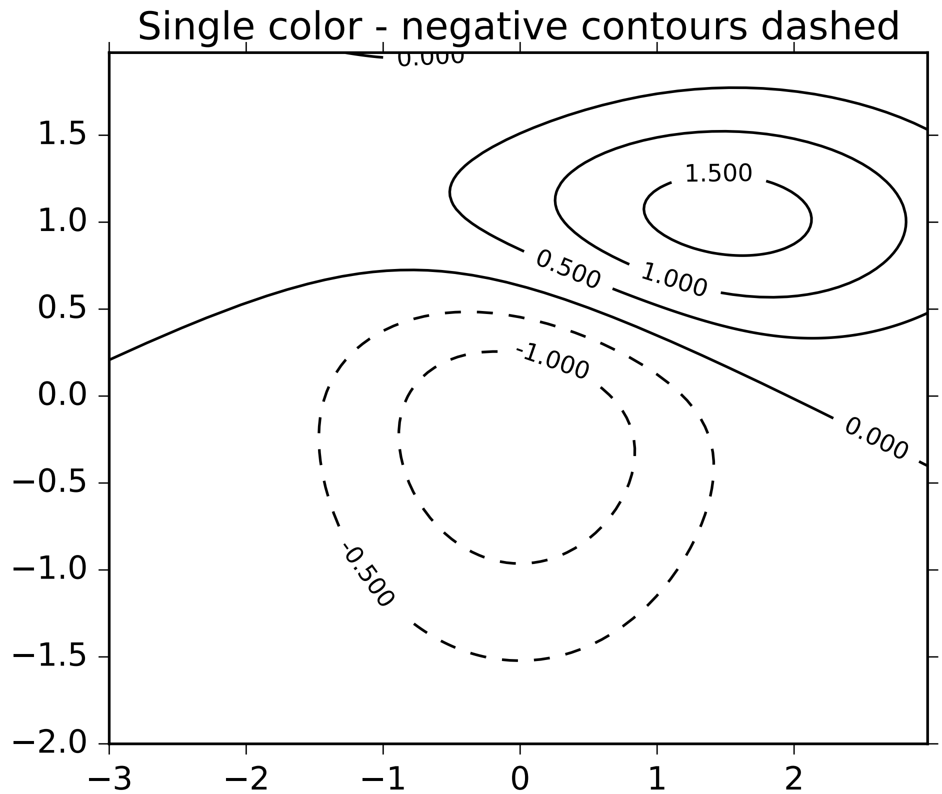
manual can be an iterable object of x,y tuples. Contour labels will be created as if mouse is clicked at each x,y positions.

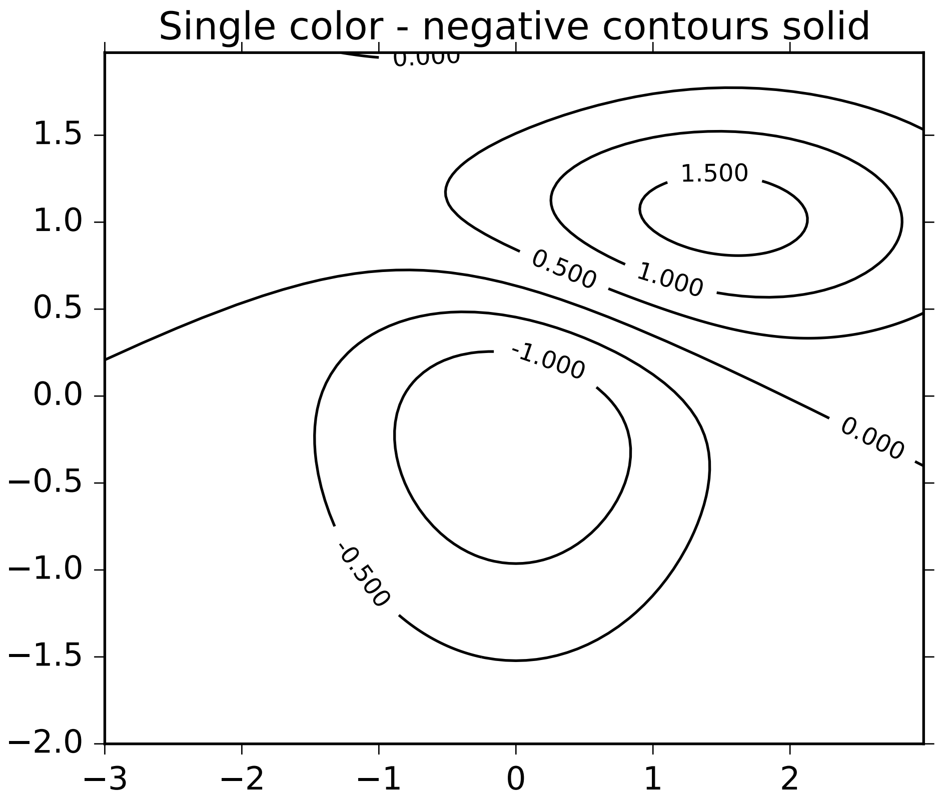
***rightside_up*:** if *True* (default), label rotations will always be plus or minus 90 degrees from level.

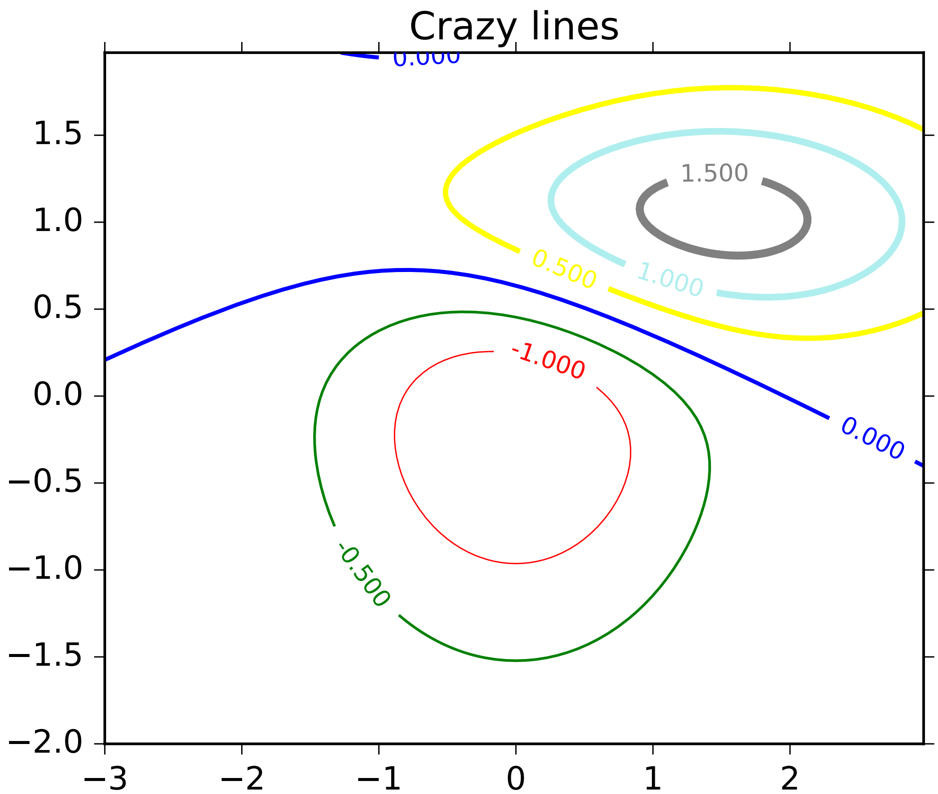
***use_clabeltext*:** if *True* (default is False), ClabelText class (instead of matplotlib.Text) is used to create labels. ClabelText recalculates rotation angles of texts during the drawing time, therefore this can be used if aspect of the axes changes.

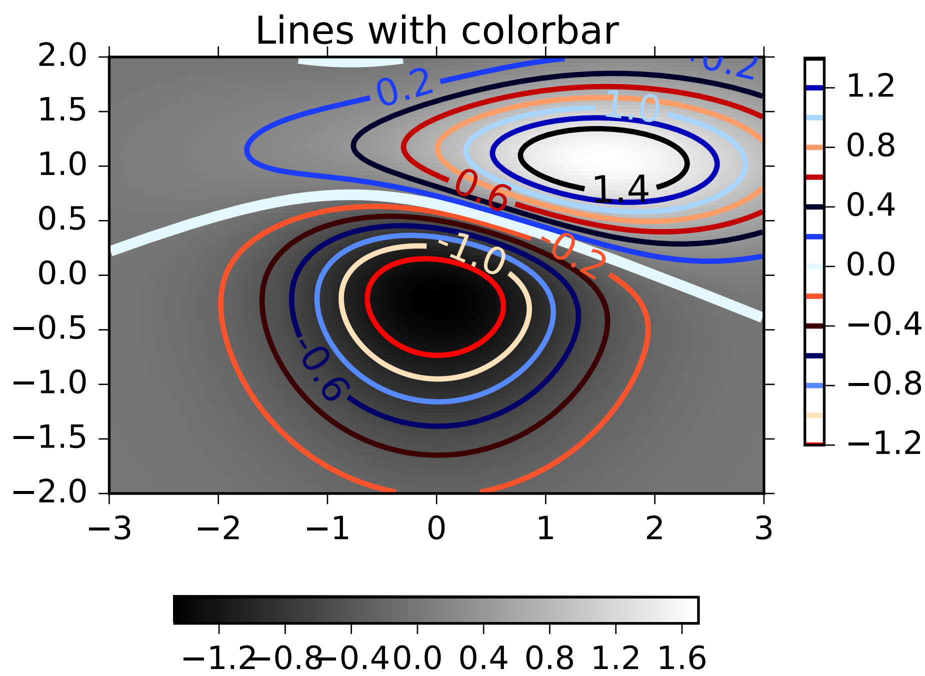

Simplest default with labels
0.000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
0.500
1.000
1.500
-1.000
0.000
-0.500


labels at selected locations
0.500
1.000
1.500
0.000
-1.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Single color - negative contours dashed
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Single color - negative contours solid
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Crazy lines
0.000
1.500
1.000
0.500
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2



Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.clf()`
Clear the current figure.

`matplotlib.pyplot.clim(`*vmin=None*, *vmax=None*`)`
Set the color limits of the current image.

To apply clim to all axes images do:

```
clim(0, 0.5)
```

If either *vmin* or *vmax* is None, the image min/max respectively will be used for color scaling.

If you want to set the clim of multiple images, use, for example:

```
for im in gca().get_images():
    im.set_clim(0, 0.05)
```

`matplotlib.pyplot.close(`**args*`)`
Close a figure window.

`close()` by itself closes the current figure

`close(h)` where *h* is a `Figure` instance, closes that figure

`close(num)` closes figure number *num*

`close(name)` where *name* is a string, closes figure with that label

`close('all')` closes all the figure windows

`matplotlib.pyplot.cohere`(*x*, *y*, *NFFT=256*, *Fs=2*, *Fc=0*, *detrend=<function detrend_none>*, *window=<function window_hanning>*, *noverlap=0*, *pad_to=None*, *sides='default'*, *scale_by_freq=None*, *hold=None*, *data=None*, ***kwargs*)

Plot the coherence between *x* and *y*.

Call signature:

```
cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend = mlab.detrend_none,
       window = mlab.window_hanning, noverlap=0, pad_to=None,
       sides='default', scale_by_freq=None, **kwargs)
```

Plot the coherence between *x* and *y*. Coherence is the normalized cross spectral density:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}} \tag{68.1}$$

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or** callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**
Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***noverlap*: integer** The number of points of overlap between blocks. The default value is 0 (no overlap).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

The return value is a tuple (Cxy, f), where f are the frequencies of the coherence vector.

kwargs are applied to the lines.

References:

- Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the `Line2D` properties of the coherence plot:

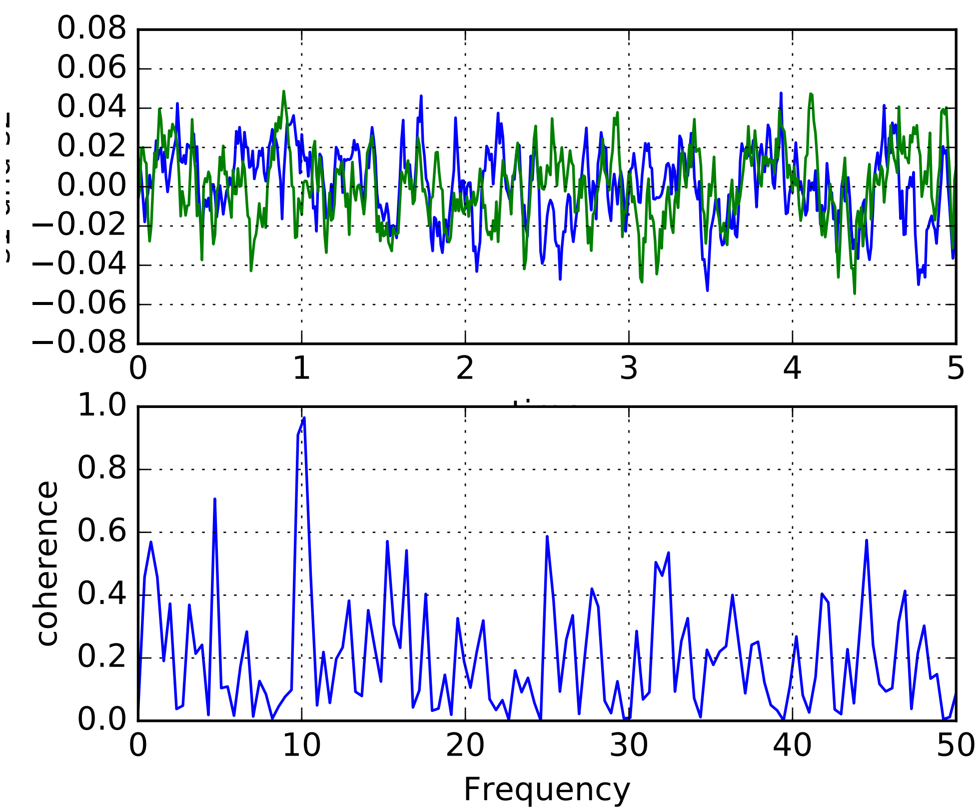
Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]

C

Table 68.10 – continued from previous page

Property	Description
`path_effects`	unknown
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`solid_capstyle`	[’butt’ | ‘round’ | ‘projecting’]
`solid_joinstyle`	[’miter’ | ‘round’ | ‘bevel’]
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
`ydata`	1D array
`zorder`	any number

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.colorbar`(*mappable=None*, *cax=None*, *ax=None*, ***kw*)

Add a colorbar to a plot.

Function signatures for the `pyplot` interface; all but the first are also method signatures for the `colorbar()` method:

```
colorbar(**kwargs)
colorbar(mappable, **kwargs)
colorbar(mappable, cax=cax, **kwargs)
colorbar(mappable, ax=ax, **kwargs)
```

arguments:

mappable the `Image`, `ContourSet`, etc. to which the colorbar applies; this argument is mandatory for the `colorbar()` method but optional for the `colorbar()` function, which sets the default to the current image.

keyword arguments:

cax None | axes object into which the colorbar will be drawn

ax None | parent axes object(s) from which space for a new colorbar axes will be stolen. If a list of axes is given they will all be resized to make room for the colorbar axes.

use_gridspec False | If *cax* is None, a new *cax* is created as an instance of Axes. If *ax* is an instance of Subplot and *use_gridspec* is True, *cax* is created as an instance of Subplot using the grid_spec module.

Additional keyword arguments are of two kinds:

axes properties:

Property	Description
orientation	vertical or horizontal
fraction	0.15; fraction of original axes to use for colorbar
pad	0.05 if vertical, 0.15 if horizontal; fraction of original axes between colorbar and new image axes
shrink	1.0; fraction by which to shrink the colorbar
aspect	20; ratio of long to short dimensions
anchor	(0.0, 0.5) if vertical; (0.5, 1.0) if horizontal; the anchor point of the colorbar axes
panchor	(1.0, 0.5) if vertical; (0.5, 0.0) if horizontal; the anchor point of the colorbar parent axes. If False, the parent axes' anchor will be unchanged

colorbar properties:

Property	Description
extend	['neither' | 'both' | 'min' | 'max'] If not 'neither', make pointed end(s) for out-of- range values. These are set for a given colormap using the colormap set_under and set_over methods.
extendfrac	[*None* | 'auto' | length | lengths] If set to *None*, both the minimum and maximum triangular colorbar extensions with have a length of 5% of the interior colorbar length (this is the default setting). If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when *spacing* is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when *spacing* is set to 'proportional'). If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.
extendrect	[*False* | *True*] If *False* the minimum and maximum colorbar extensions will be triangular (the default). If *True* the extensions will be rectangular.
spacing	['uniform' | 'proportional'] Uniform spacing gives each discrete color the same space; proportional makes the space proportional to the data interval.
ticks	[None | list of ticks | Locator object] If None, ticks are determined automatically from the input.
format	[None | format string | Formatter object] If None, the `ScalarFormatter` is used. If a format string is given, e.g., '%.3f', that is used. An alternative `Formatter` object may be given instead.
drawedges	[False | True] If true, draw lines at color boundaries.

The following will probably be useful only in the context of indexed colors (that is, when the mappable has norm=NoNorm()), or other unusual circumstances.

Property	Description
boundaries	None or a sequence
values	None or a sequence which must be of length 1 less than the sequence of *boundaries*. For each region delimited by adjacent entries in *boundaries*, the color mapped to the corresponding value in values will be used.

If *mappable* is a `ContourSet`, its *extend* kwarg is included automatically.

Note that the *shrink* kwarg provides a simple way to keep a vertical colorbar, for example, from being taller than the axes of the mappable to which the colorbar is attached; but it is a manual method requiring some trial and error. If the colorbar is too tall (or a horizontal colorbar is too wide) use a smaller value of *shrink*.

For more precise control, you can manually specify the positions of the axes objects in which the mappable and the colorbar are drawn. In this case, do not use any of the axes properties kwargs.

It is known that some vector graphics viewer (svg and pdf) renders white gaps between segments of the colorbar. This is due to bugs in the viewers not matplotlib. As a workaround the colorbar can be rendered with overlapping segments:

```
cbar = colorbar()
cbar.solids.set_edgecolor("face")
draw()
```

However this has negative consequences in other circumstances. Particularly with semi transparent images (alpha < 1) and colorbar extensions and is not enabled by default see (issue #1188).

returns: `Colorbar` instance; see also its base class, `ColorbarBase`. Call the `set_label()` method to label the colorbar.

`matplotlib.pyplot.colors()`

This is a do-nothing function to provide you with help on how matplotlib handles colors.

Commands which take color arguments can use several formats to specify the colors. For the basic built-in colors, you can use a single letter

Alias	Color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

For a greater range of colors, you have two options. You can specify the color using an html hex string, as in:

```
color = '#eeefff'
```

or you can pass an R,G,B tuple, where each of R,G,B are in the range [0,1].

You can also use any legal html name for a color, for example:

```
color = 'red'
color = 'burlywood'
color = 'chartreuse'
```

The example below creates a subplot with a dark slate gray background:

```
subplot(111, axisbg=(0.1843, 0.3098, 0.3098))
```

Here is an example that creates a pale turquoise title:

```
title('Is this the best color?', color='#afeeee')
```

`matplotlib.pyplot.`**`connect`**`(`*s*, *func*`)`

Connect event with string *s* to *func*. The signature of *func* is:

```
def func(event)
```

where event is a *`matplotlib.backend_bases.Event`*. The following events are recognized

- 'button_press_event'
- 'button_release_event'
- 'draw_event'
- 'key_press_event'
- 'key_release_event'
- 'motion_notify_event'
- 'pick_event'
- 'resize_event'
- 'scroll_event'
- 'figure_enter_event',
- 'figure_leave_event',
- 'axes_enter_event',
- 'axes_leave_event'
- 'close_event'

For the location events (button and key press/release), if the mouse is over the axes, the variable `event.inaxes` will be set to the *`Axes`* the event occurs is over, and additionally, the variables `event.xdata` and `event.ydata` will be defined. This is the mouse location in data coords. See *`KeyEvent`* and *`MouseEvent`* for more info.

Return value is a connection id that can be used with `mpl_disconnect()`.

Example usage:

```
def on_press(event):
    print('you pressed', event.button, event.xdata, event.ydata)

cid = canvas.mpl_connect('button_press_event', on_press)
```

`matplotlib.pyplot.`**`contour`**`(`**args*, ***kwargs*`)`

Plot contours.

`contour()` and *`contourf()`* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

`contourf()` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to *`contour()`*.

Call signatures:

```
contour(Z)
```

make a contour plot of an array *Z*. The level values are chosen automatically.

```
contour(X,Y,Z)
```

X, *Y* specify the (x, y) coordinates of the surface

```
contour(Z,N)
contour(X,Y,Z,N)
```

contour up to *N* automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence *V*, which must be in increasing order.

```
contourf(..., V)
```

fill the `len(V)-1` regions between the values in *V*, which must be in increasing order.

```
contour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

X and *Y* must both be 2-D with the same shape as *Z*, or they must both be 1-D such that `len(X)` is the number of columns in *Z* and `len(Y)` is the number of rows in *Z*.

`C = contour(...)` returns a `QuadContourSet` object.

Optional keyword arguments:

***corner_mask*: [*True* | *False* | 'legacy']** Enable/disable corner masking, which only has an effect if *Z* is a masked array. If *False*, any quad touching a masked point is masked out. If *True*, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual. If 'legacy', the old contouring algorithm is used, which is equivalent to *False* and is deprecated, only remaining whilst the new algorithm is tested fully.

If not specified, the default is taken from rcParams['contour.corner_mask'], which is True unless it has been modified.

***colors*: [*None* | string | (mpl_colors)]** If *None*, the colormap specified by cmap will be used.

If a string, like 'r' or 'red', all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

***alpha*: float** The alpha blending value

***cmap*: [*None* | Colormap]** A cm `Colormap` instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

***vmin*, *vmax*: [*None* | scalar]** If not *None*, either or both of these values will be supplied to the `matplotlib.colors.Normalize` instance, overriding the default color scaling based on *levels*.

***levels*: [level0, level1, ..., leveln]** A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

***origin*: [*None* | 'upper' | 'lower' | 'image']** If *None*, the first value of *Z* will correspond to the lower left corner, location (0,0). If 'image', the rc value for `image.origin` will be used.

This keyword is not active if *X* and *Y* are specified in the call to contour.

extent: [*None* | (x0,x1,y0,y1)]
If *origin* is not *None*, then *extent* is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0], and (*x1*, *y1*) is the position of Z[-1,-1].

This keyword is not active if *X* and *Y* are specified in the call to contour.

***locator*: [*None* | ticker.Locator subclass]** If *locator* is *None*, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

***extend*: ['neither' | 'both' | 'min' | 'max']** Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

***xunits*, *yunits*: [*None* | registered units]** Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

***antialiased*: [*True* | *False*]** enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from rcParams['lines.antialiased'].

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

contour-only keyword arguments:

***linewidths*: [*None* | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

***linestyles*: [*None* | 'solid' | 'dashed' | 'dashdot' | 'dotted']** If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc` `contour.negative_linestyle` setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

contourf-only keyword arguments:

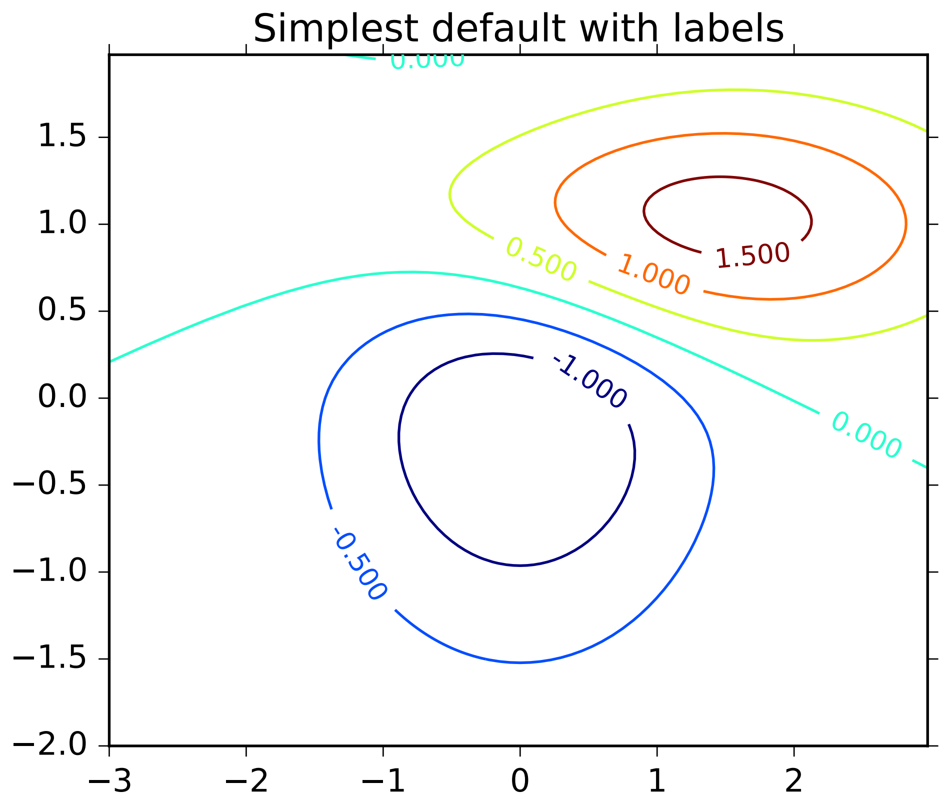
hatches: A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

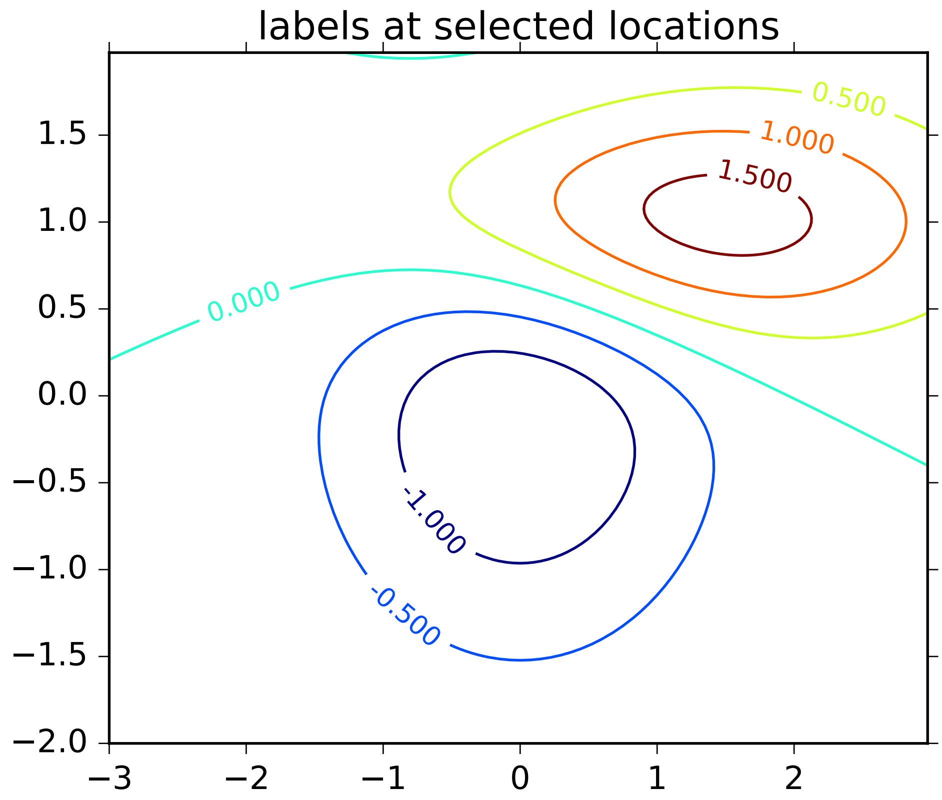
Note: contourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

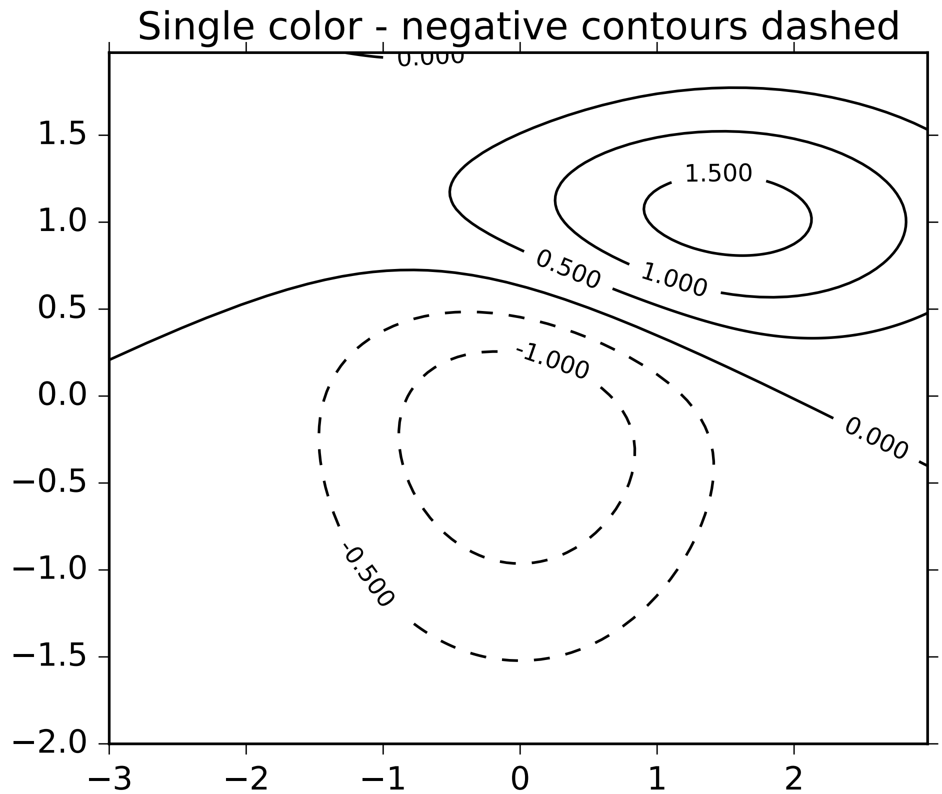
```
z1 < z <= z2
```

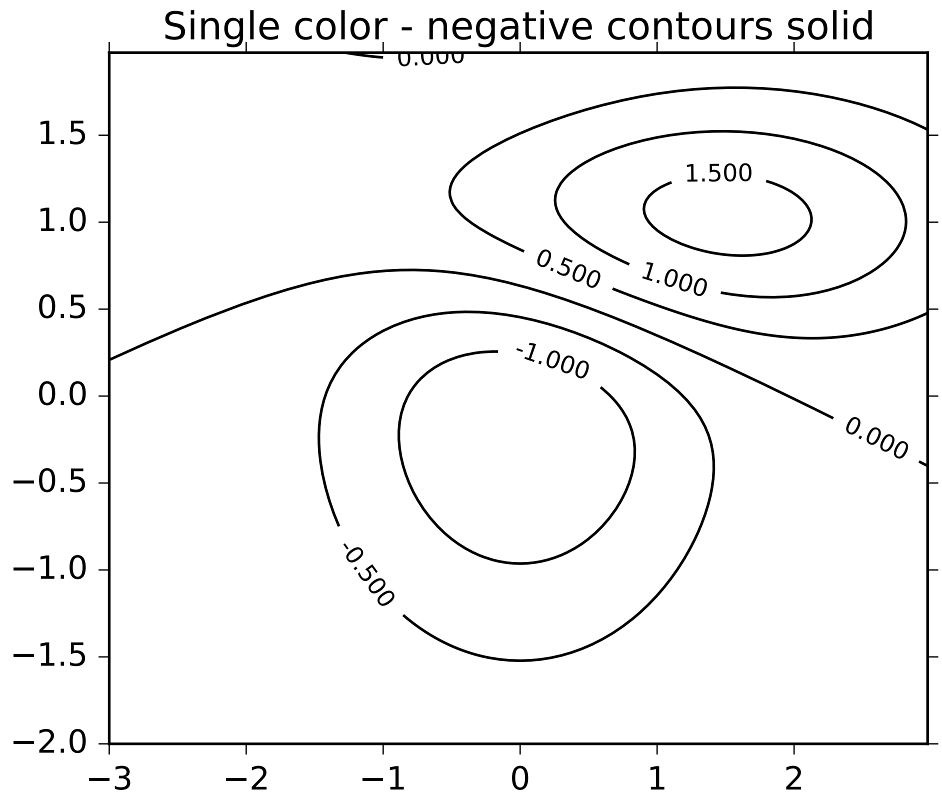
There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

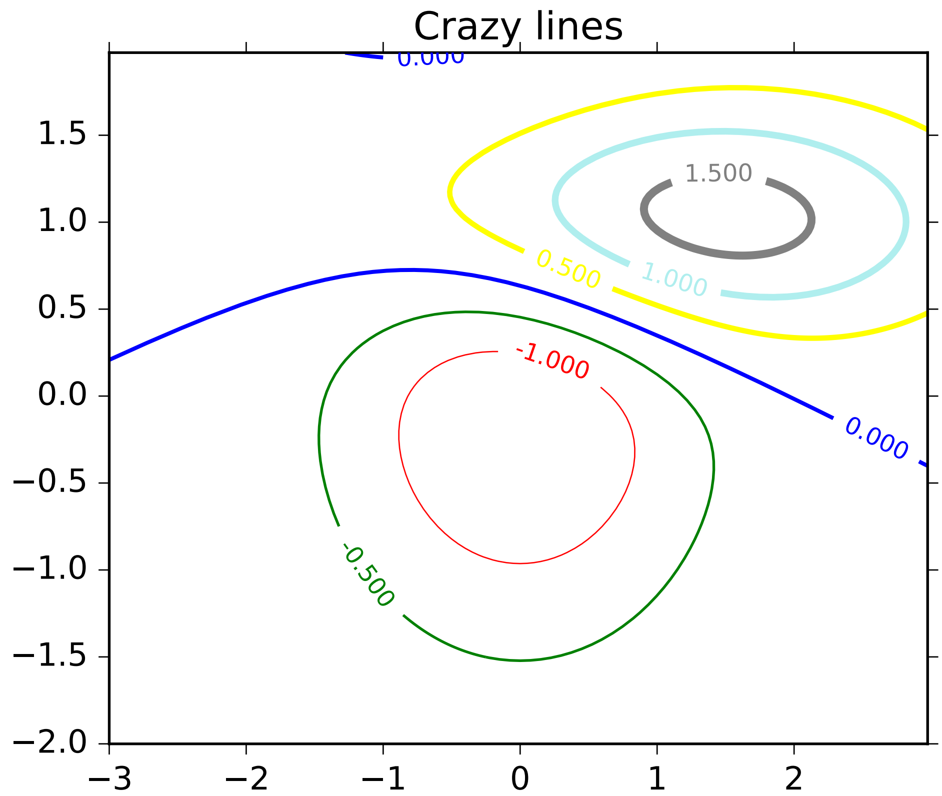
Examples:

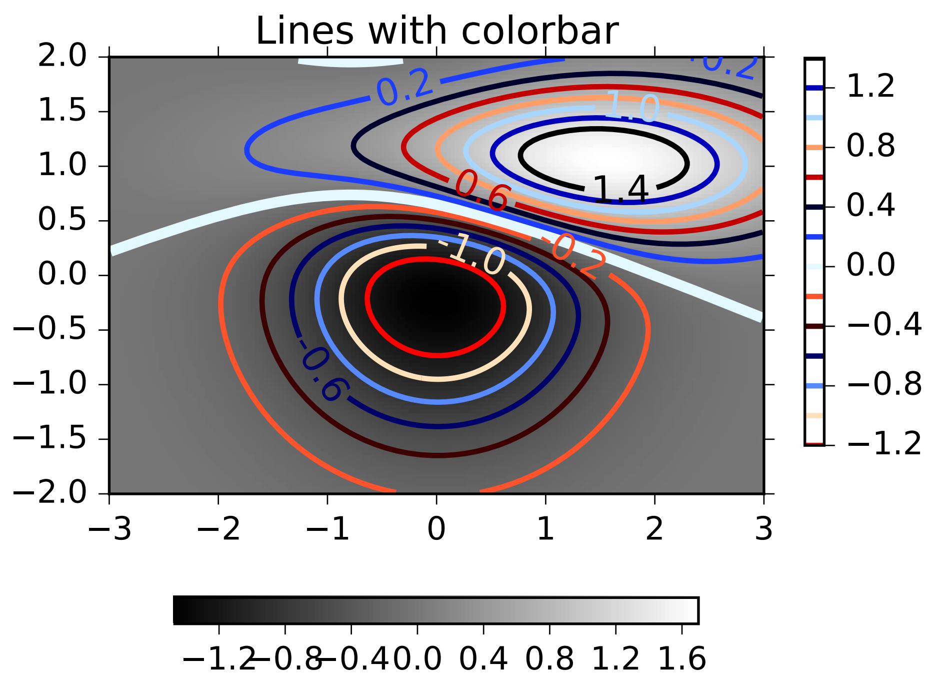


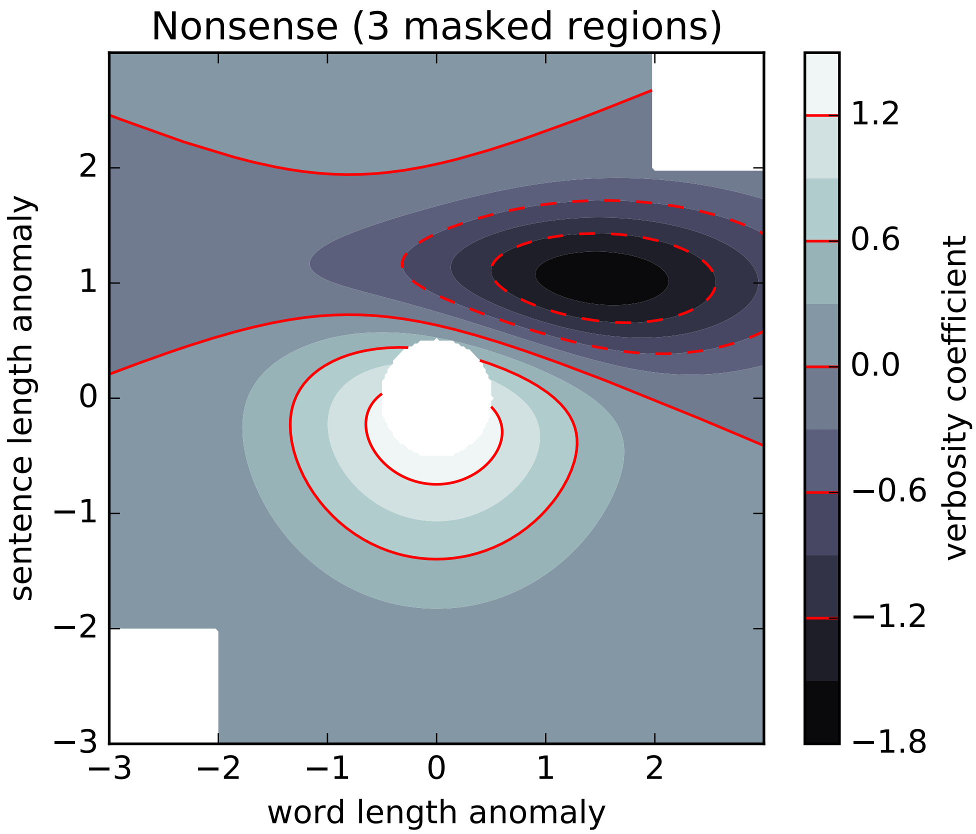

labels at selected locations
0.500
1.000
1.500
0.000
-1.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

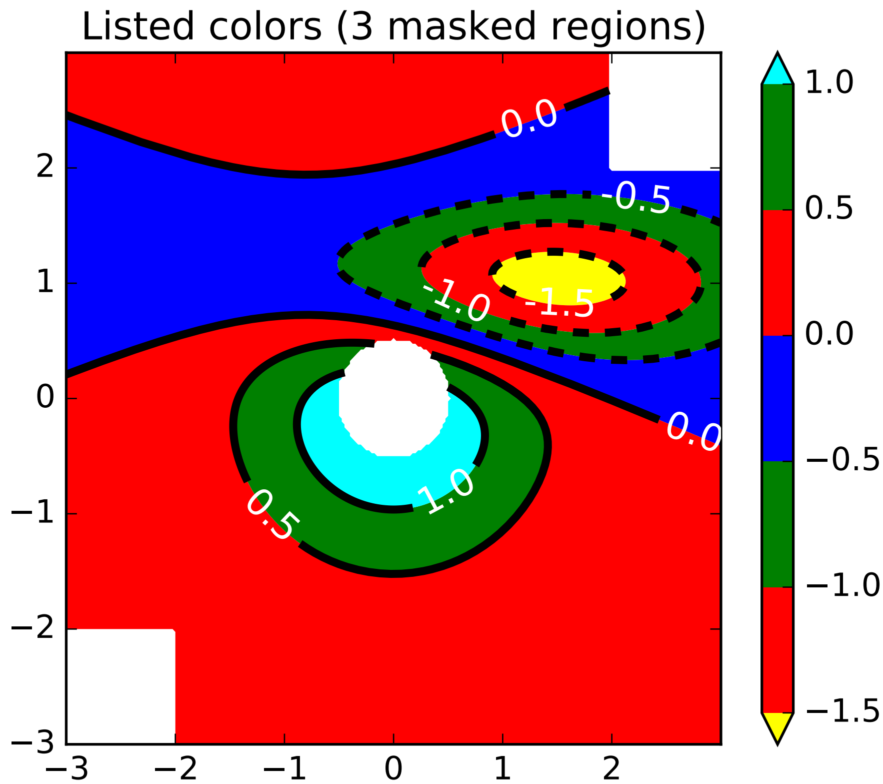

Single color - negative contours dashed
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

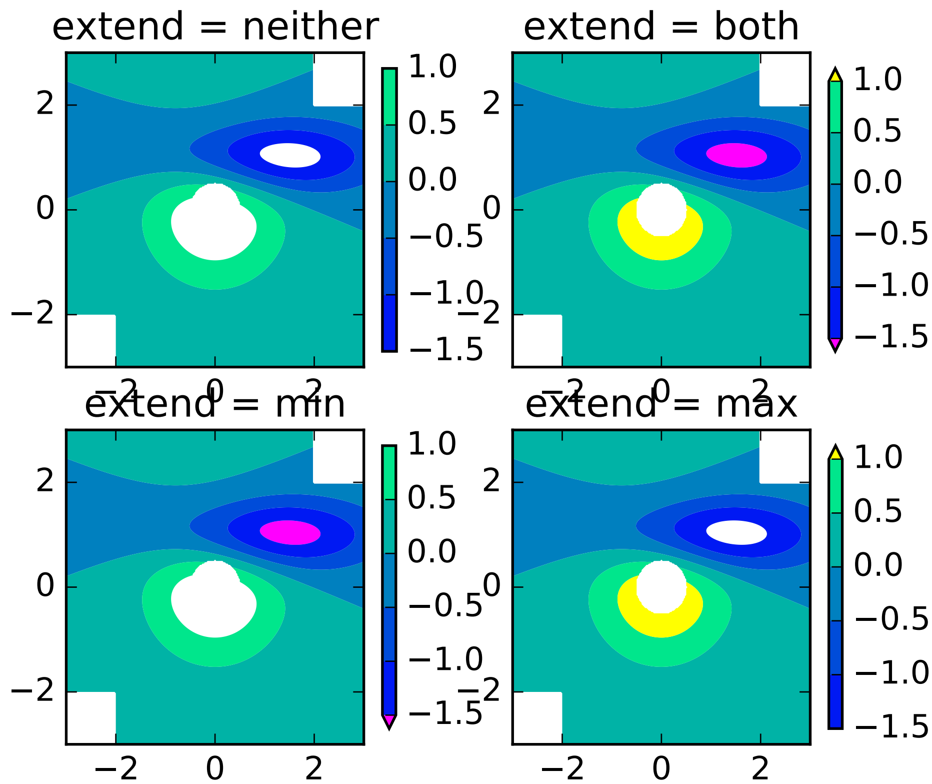

Single color - negative contours solid
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

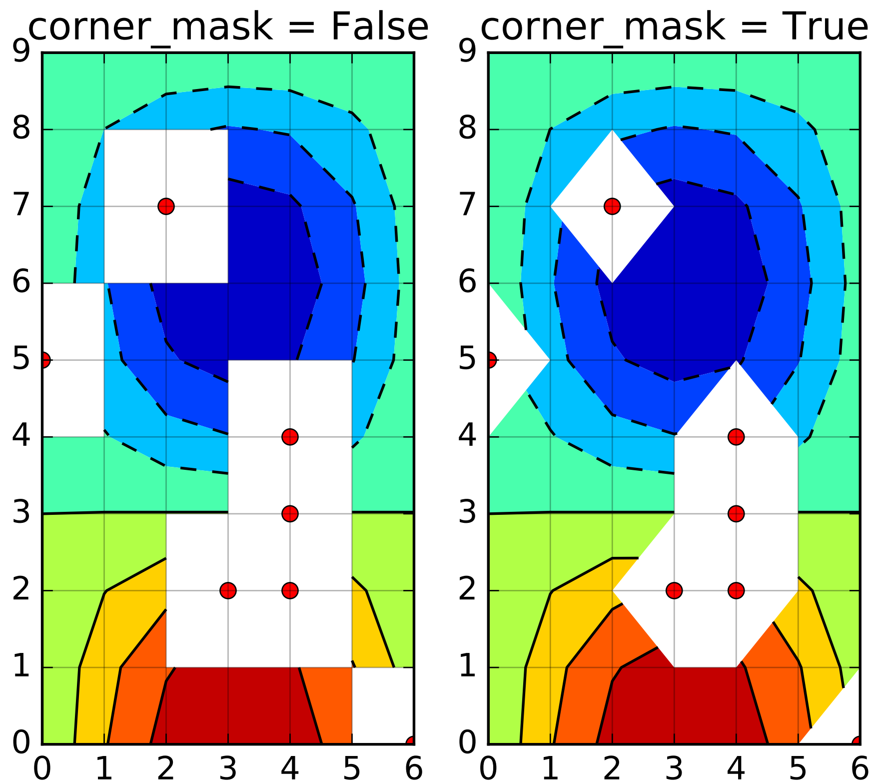

Crazy lines
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Lines with colorbar
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
3
0.2
1.0
1.4
0.6
-1.0
-0.2
-0.6
1.2
0.8
0.4
0.0
−0.4
−0.8
−1.2
−1.2
−0.8
−0.4
0.0
0.4
0.8
1.2
1.6


Nonsense (3 masked regions)
sentence length anomaly
word length anomaly
verbosity coefficient
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
1.2
0.6
0.0
−0.6
−1.2
−1.8


Listed colors (3 masked regions)
0.0
-0.5
-1.0
-1.5
0.0
0.5
1.0
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
1.0
0.5
0.0
−0.5
−1.0
−1.5


extend = neither
extend = both
extend = min
extend = max
1.0
0.5
0.0
−0.5
−1.0
−1.5
2
0
−2



Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.contourf(*args, **kwargs)`

Plot contours.

`contour()` and `contourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

`contourf()` differs from the MATLAB version in that it does not draw the polygon edges. To draw edges, add line contours with calls to `contour()`.

Call signatures:

```
contour(Z)
```

make a contour plot of an array *Z*. The level values are chosen automatically.

```
contour(X,Y,Z)
```

X, *Y* specify the (x, y) coordinates of the surface

```
contour(Z,N)
contour(X,Y,Z,N)
```

contour up to *N* automatically-chosen levels.

```
contour(Z,V)
contour(X,Y,Z,V)
```

draw contour lines at the values specified in sequence *V*, which must be in increasing order.

```
contourf(..., V)
```

fill the `len(V)-1` regions between the values in *V*, which must be in increasing order.

```
contour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

X and *Y* must both be 2-D with the same shape as *Z*, or they must both be 1-D such that `len(X)` is the number of columns in *Z* and `len(Y)` is the number of rows in *Z*.

`C = contour(...)` returns a `QuadContourSet` object.

Optional keyword arguments:

***corner_mask*: [*True* | *False* | 'legacy']** Enable/disable corner masking, which only has an effect if *Z* is a masked array. If *False*, any quad touching a masked point is masked out. If *True*, only the triangular corners of quads nearest those points are always masked out, other triangular corners comprising three unmasked points are contoured as usual. If 'legacy', the old contouring algorithm is used, which is equivalent to *False* and is deprecated, only remaining whilst the new algorithm is tested fully.

If not specified, the default is taken from rcParams['contour.corner_mask'], which is True unless it has been modified.

***colors*: [*None* | string | (mpl_colors)]** If *None*, the colormap specified by cmap will be used.

If a string, like 'r' or 'red', all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

***alpha*: float** The alpha blending value

***cmap*: [*None* | Colormap]** A cm `Colormap` instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

***vmin*, *vmax*: [*None* | scalar]** If not *None*, either or both of these values will be supplied to the `matplotlib.colors.Normalize` instance, overriding the default color scaling based on *levels*.

***levels*: [level0, level1, ..., leveln]** A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

***origin*: [*None* | 'upper' | 'lower' | 'image']** If *None*, the first value of *Z* will correspond to the lower left corner, location (0,0). If 'image', the rc value for `image.origin` will be used.

This keyword is not active if *X* and *Y* are specified in the call to contour.

extent: [*None* | (x0,x1,y0,y1)]
If *origin* is not *None*, then *extent* is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0], and (*x1*, *y1*) is the position of Z[-1,-1].

This keyword is not active if *X* and *Y* are specified in the call to contour.

***locator*: [*None* | ticker.Locator subclass]** If *locator* is *None*, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

***extend*: ['neither' | 'both' | 'min' | 'max']** Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

***xunits*, *yunits*: [*None* | registered units]** Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

***antialiased*: [*True* | *False*]** enable antialiasing, overriding the defaults. For filled contours, the default is *True*. For line contours, it is taken from rcParams['lines.antialiased'].

***nchunk*: [0 | integer]** If 0, no subdivision of the domain. Specify a positive integer to divide the domain into subdomains of *nchunk* by *nchunk* quads. Chunking reduces the maximum length of polygons generated by the contouring algorithm which reduces the rendering workload passed on to the backend and also requires slightly less RAM. It can however introduce rendering artifacts at chunk boundaries depending on the backend, the *antialiased* flag and value of *alpha*.

contour-only keyword arguments:

***linewidths*: [*None* | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified.

***linestyles*: [*None* | 'solid' | 'dashed' | 'dashdot' | 'dotted']** If *linestyles* is *None*, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from the `matplotlibrc` `contour.negative_linestyle` setting.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

contourf-only keyword arguments:

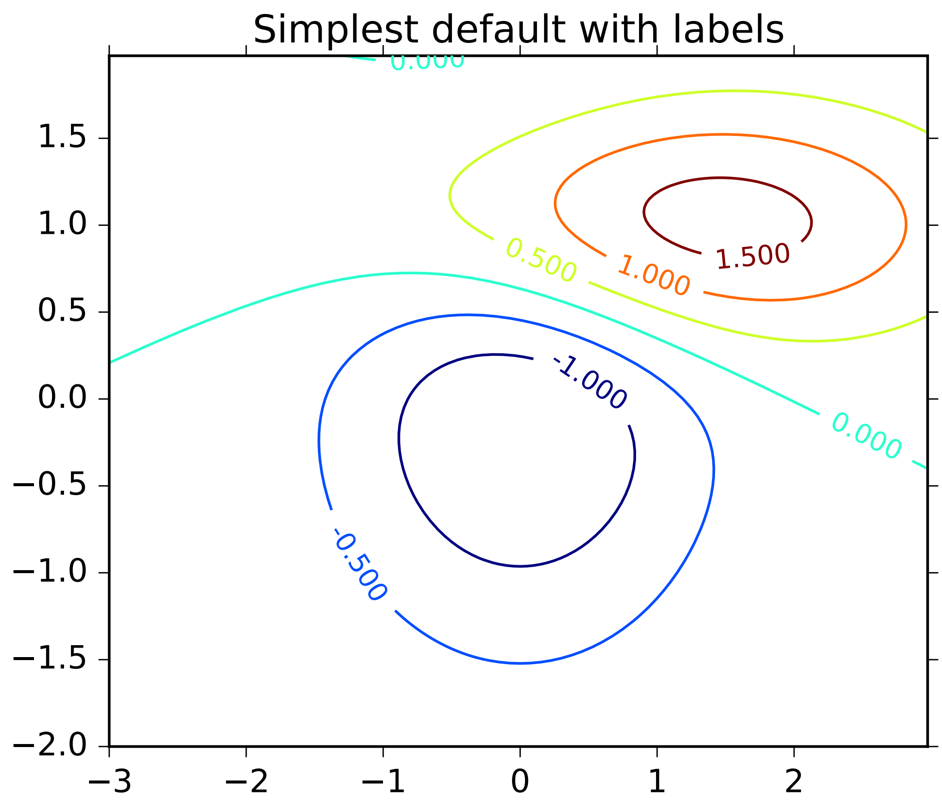
***hatches*:** A list of cross hatch patterns to use on the filled areas. If None, no hatching will be added to the contour. Hatching is supported in the PostScript, PDF, SVG and Agg backends only.

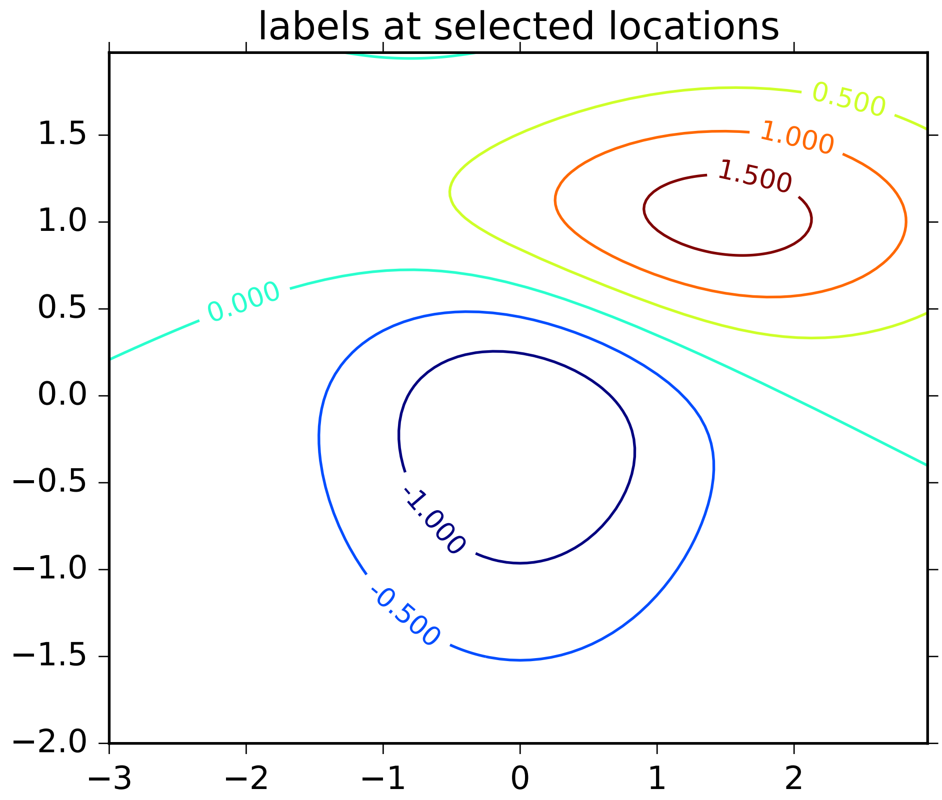
Note: contourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

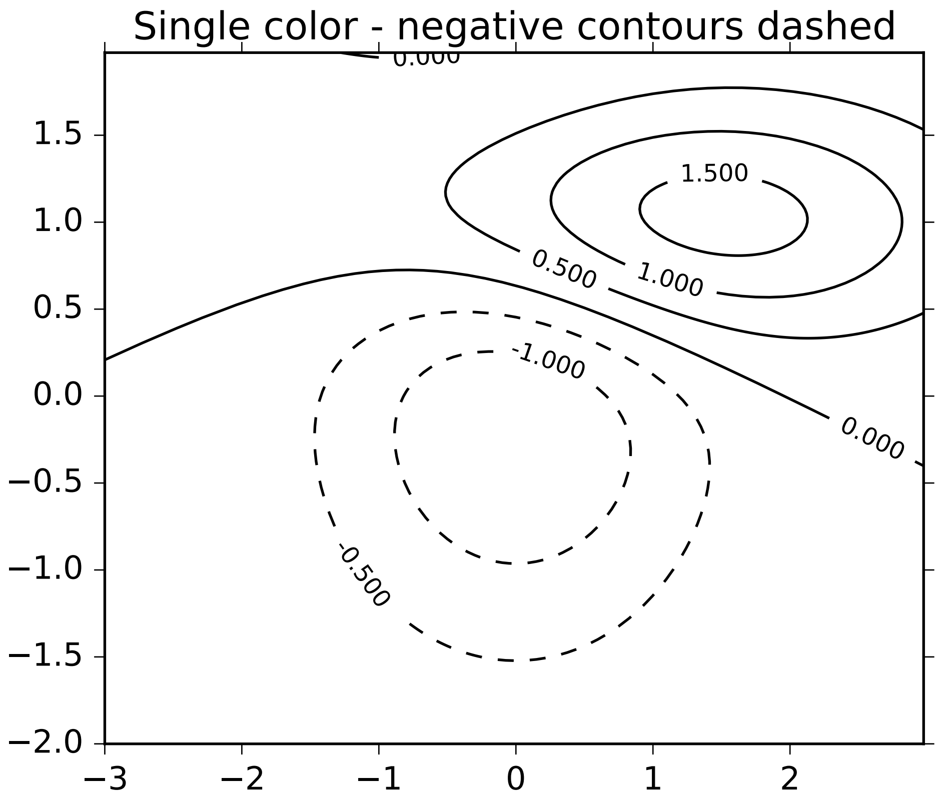
```
z1 < z <= z2
```

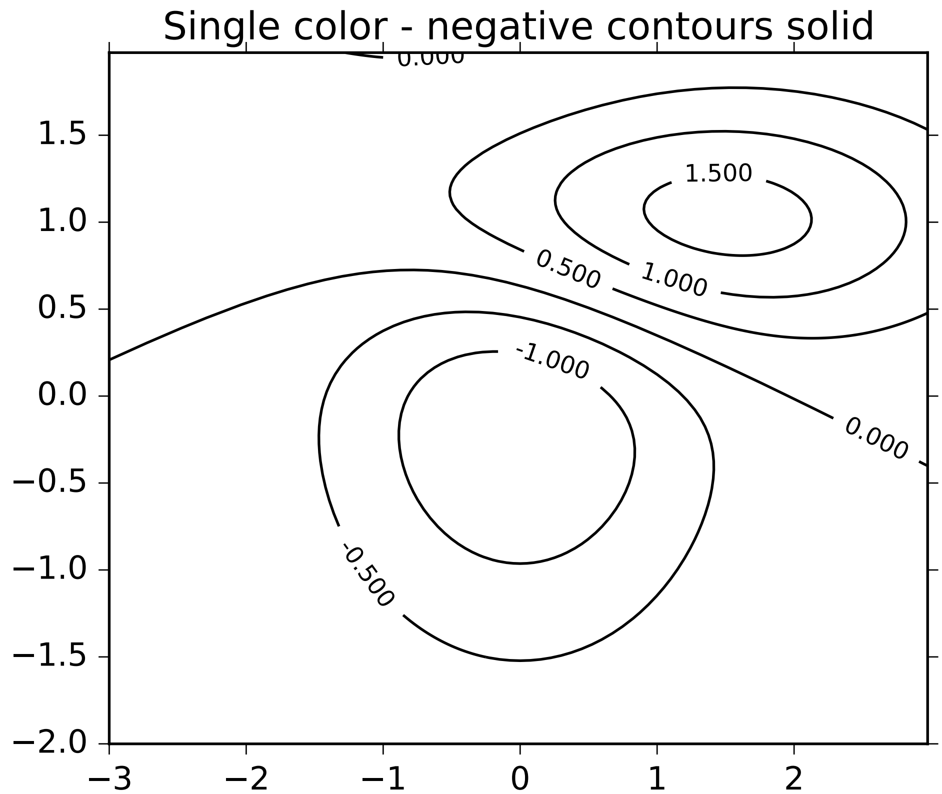
There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

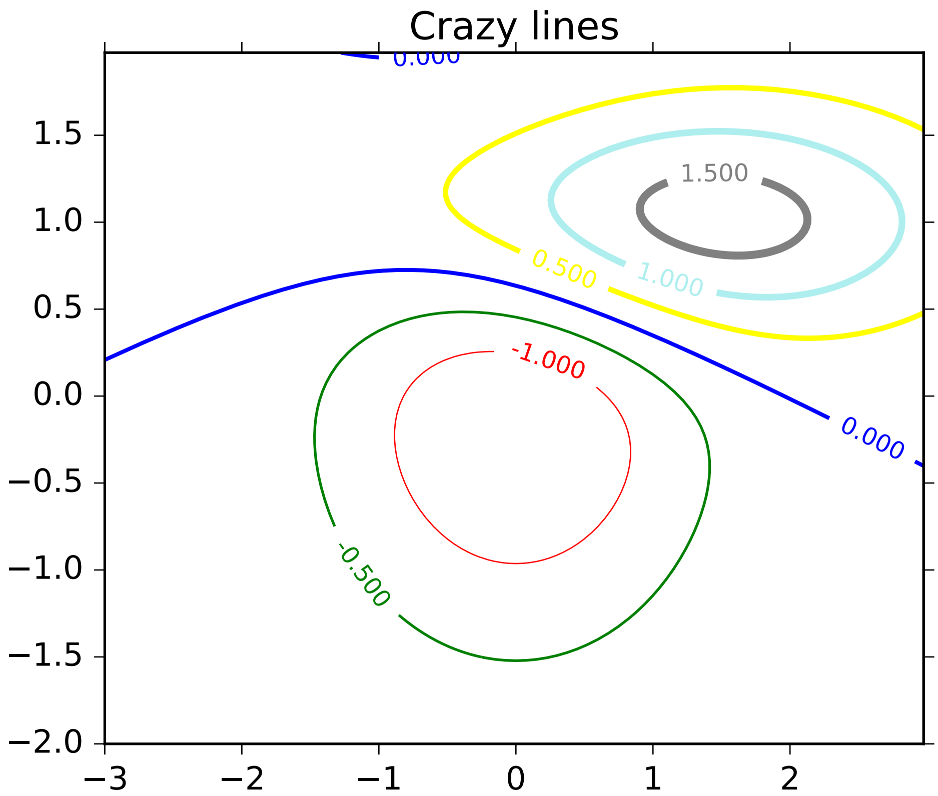
Examples:

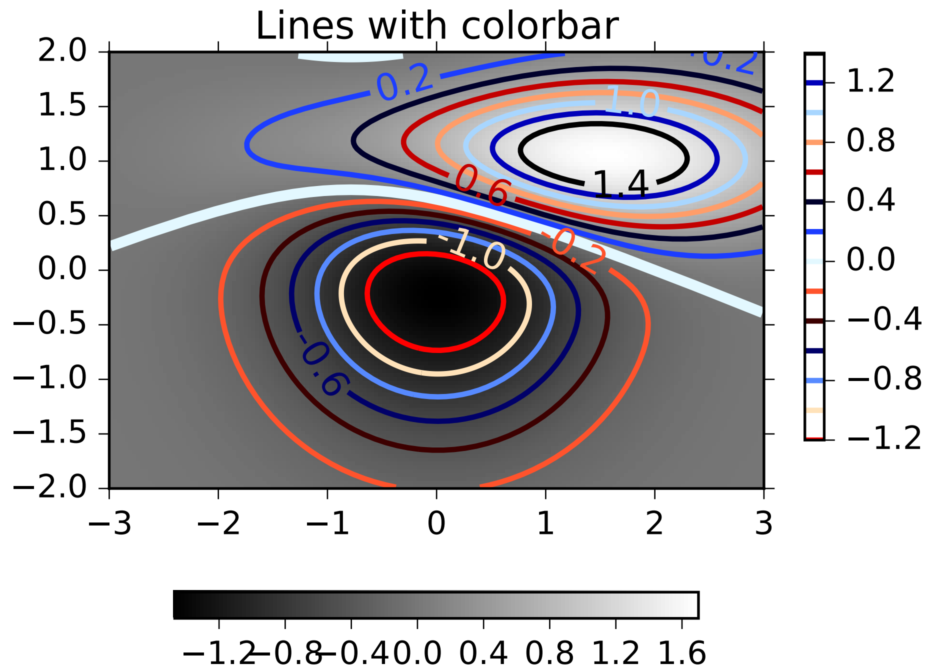


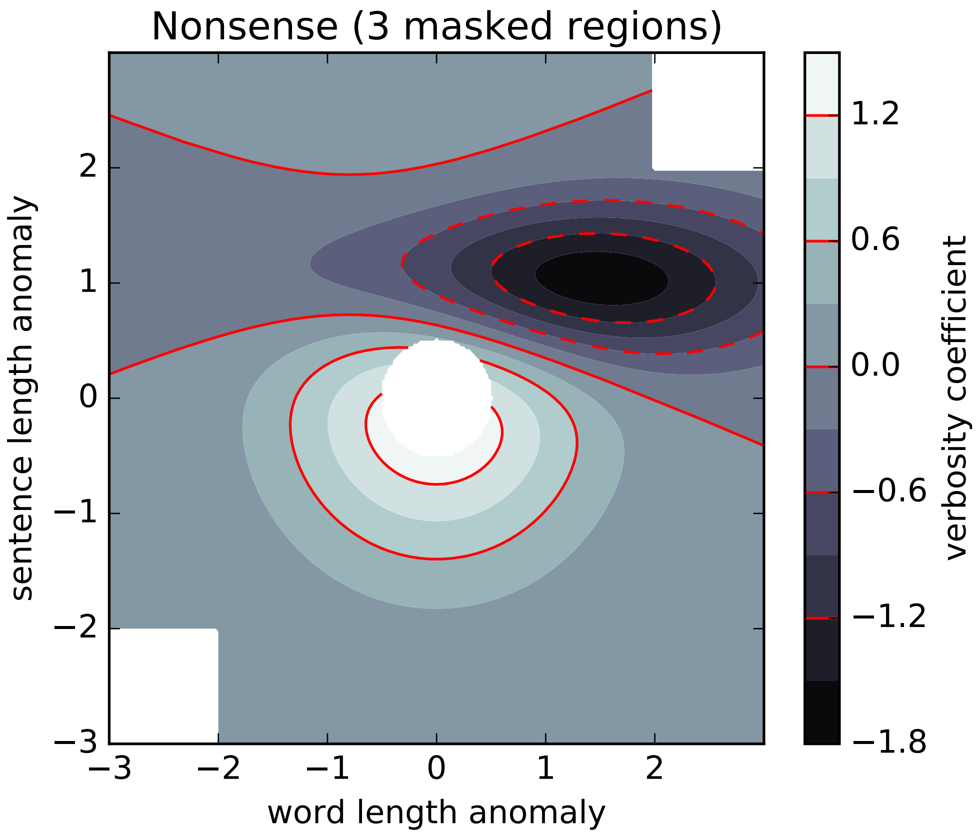

labels at selected locations
0.500
1.000
1.500
0.000
-1.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

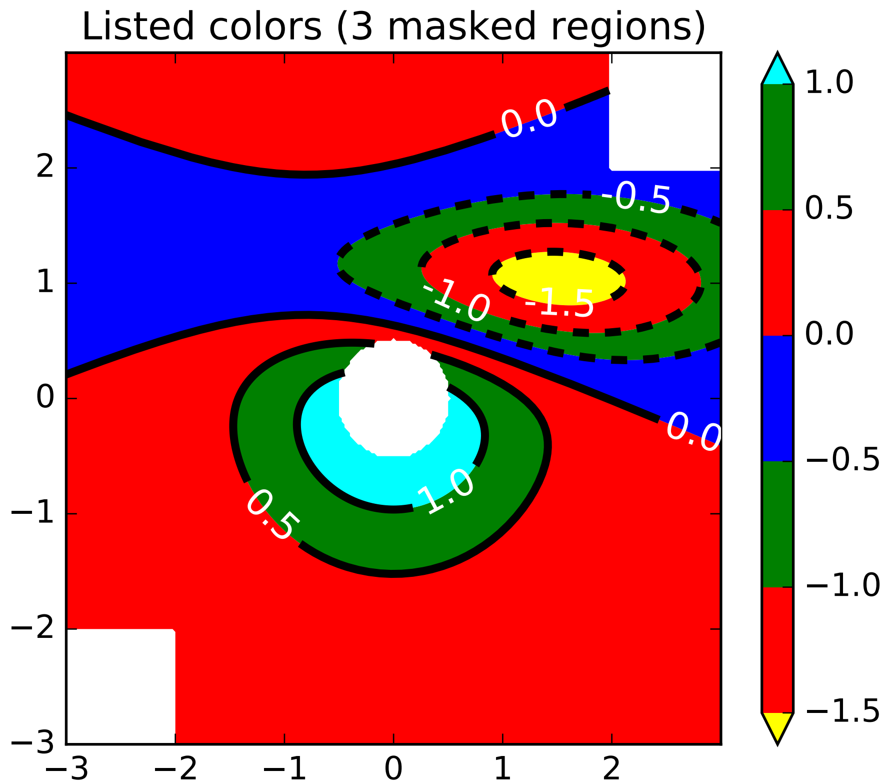

Single color - negative contours dashed
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

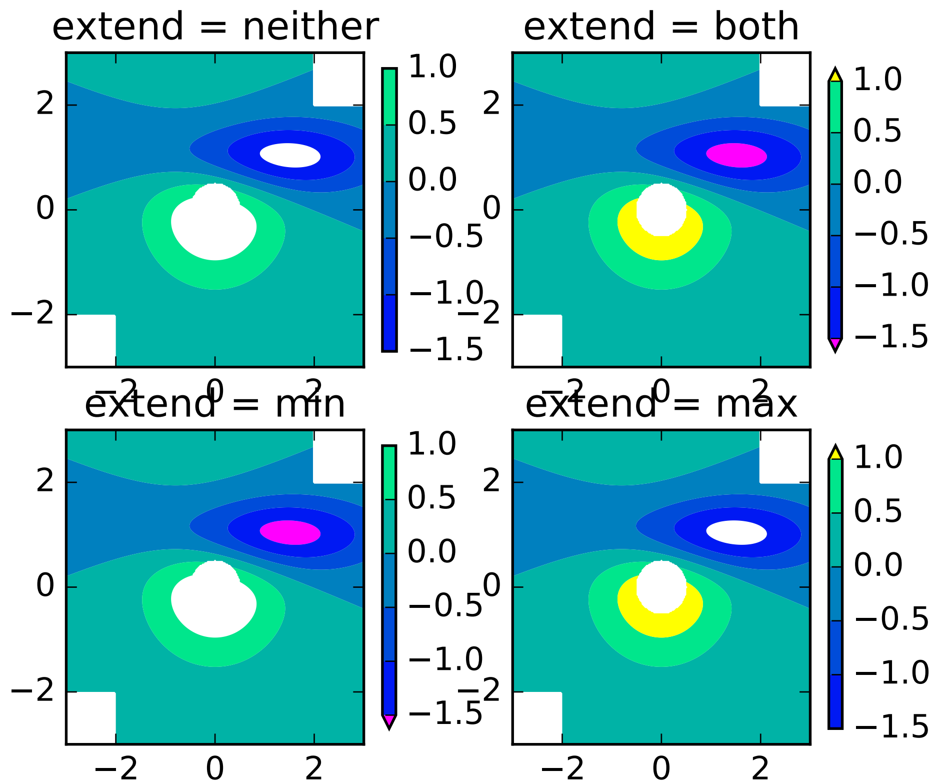

Single color - negative contours solid
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2

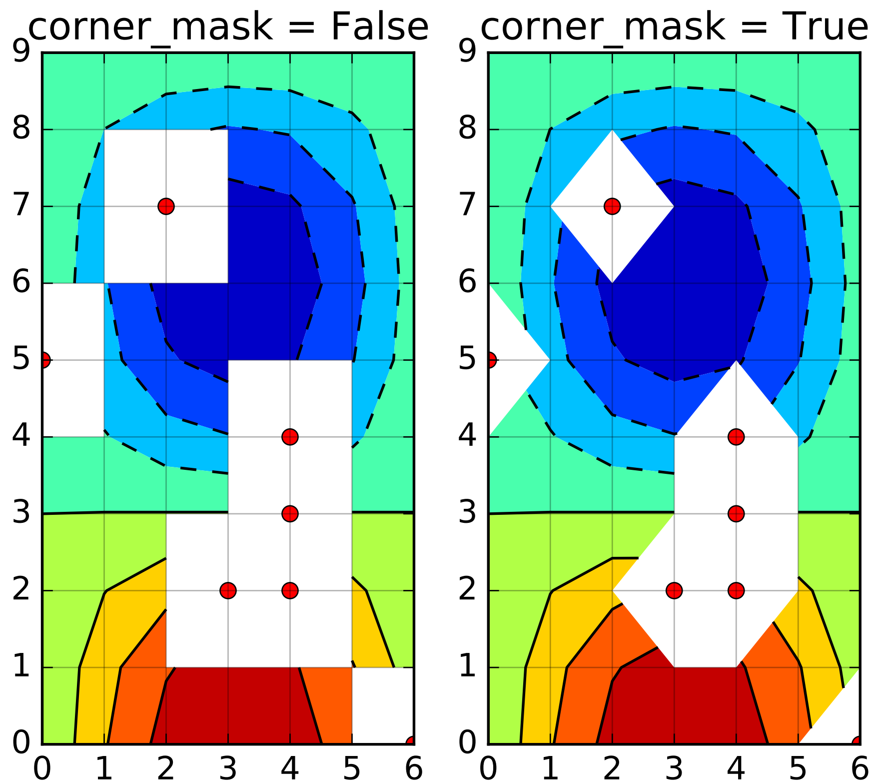

Crazy lines
0.000
1.500
1.000
0.500
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Lines with colorbar
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
3
0.2
-0.2
1.0
0.6
1.4
-1.0
-0.2
-0.6
1.2
0.8
0.4
0.0
−0.4
−0.8
−1.2
−1.2
−0.8
−0.4
0.0
0.4
0.8
1.2
1.6


Nonsense (3 masked regions)
sentence length anomaly
word length anomaly
verbosity coefficient
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
1.2
0.6
0.0
−0.6
−1.2
−1.8


Listed colors (3 masked regions)
0.0
-0.5
-1.0
-1.5
0.0
0.5
1.0
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
1.0
0.5
0.0
−0.5
−1.0
−1.5


extend = neither
extend = both
extend = min
extend = max
1.0
0.5
0.0
−0.5
−1.0
−1.5
2
0
−2
−2
0
2



Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.cool()`

set the default colormap to cool and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.copper()`

set the default colormap to copper and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.csd(`*x*, *y*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, *hold=None*, *data=None*, ***kwargs*`)`

Plot the cross-spectral density.

Call signature:

```
csd(x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None, return_line=None, **kwargs)
```

The cross spectral density P_{xy} by Welch's average periodogram method. The vectors *x* and *y* are divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The product of the direct FFTs of *x* and *y* are averaged over each segment to compute P_{xy}, with a scaling to correct for power loss due to windowing.

If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero padded to *NFFT*.

***x*, *y*: 1-D arrays or sequences** Arrays or sequences containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or** callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***noverlap*: integer** The number of points of overlap between segments. The default value is 0 (no overlap).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

***return_line*: bool** Whether to include the line object plotted in the returned values. Default is False.

If *return_line* is False, returns the tuple (*Pxy*, *freqs*). If *return_line* is True, returns the tuple (*Pxy*, *freqs*. *line*):

***Pxy*: 1-D array** The values for the cross spectrum `P_{xy}` before scaling (complex valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *Pxy*

***line*: a `Line2D` instance** The line created by this function. Only returend if *return_line* is True.

For plotting, the power is plotted as $10\log_{10}(P_{xy})$ for decibels, though `P_{xy}` itself is returned.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

kwargs control the Line2D properties:

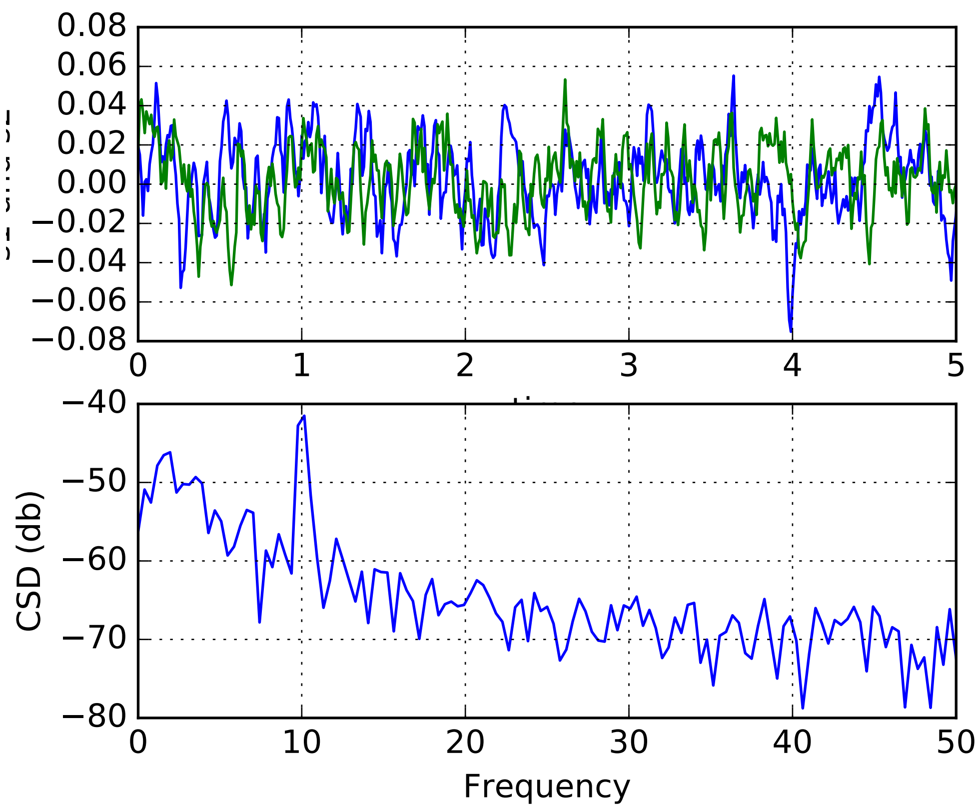
Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
`path_effects`	unknown
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string

C

Table 68.11 – continued from previous page

Property	Description
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:



See also:

psd() psd() is the equivalent to setting y=x.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.delaxes(*args)`

Remove an axes from the current figure. If *ax* doesn't exist, an error will be raised.

`delaxes()`: delete the current axes

`matplotlib.pyplot.disconnect(cid)`

Disconnect callback id cid

Example usage:

```
cid = canvas.mpl_connect('button_press_event', on_press)
#...later
canvas.mpl_disconnect(cid)
```

`matplotlib.pyplot.draw()`

Redraw the current figure.

This is used in interactive mode to update a figure that has been altered, but not automatically re-drawn. This should be only rarely needed, but there may be ways to modify the state of a figure with out marking it as `stale`. Please report these cases as bugs.

A more object-oriented alternative, given any *Figure* instance, `fig`, that was created using a *pyplot* function, is:

```
fig.canvas.draw_idle()
```

`matplotlib.pyplot.errorbar(`*x, y, yerr=None, xerr=None, fmt='', ecolor=None, elinewidth=None, capsize=None, barsabove=False, lolims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, capthick=None, hold=None, data=None, **kwargs*`)`

Plot an errorbar graph.

Call signature:

```
errorbar(x, y, yerr=None, xerr=None,
         fmt='', ecolor=None, elinewidth=None, capsize=None,
         barsabove=False, lolims=False, uplims=False,
         xlolims=False, xuplims=False, errorevery=1,
         capthick=None)
```

Plot *x* versus *y* with error deltas in *yerr* and *xerr*. Vertical errorbars are plotted if *yerr* is not *None*. Horizontal errorbars are plotted if *xerr* is not *None*.

x, *y*, *xerr*, and *yerr* can all be scalars, which plots a single error bar at *x*, *y*.

Optional keyword arguments:

***xerr*/*yerr*: [scalar | N, Nx1, or 2xN array-like]** If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn at +/-value relative to the data.

If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2 relative to the data.

***fmt*: ['' | 'none' | plot format string]** The plot format symbol. If *fmt* is 'none' (case-insensitive), only the errorbars are plotted. This is used for adding errorbars to a bar plot, for example. Default is '', an empty plot format string; properties are then identical to the defaults for *plot()*.

***ecolor*: [*None* | mpl color]** A matplotlib color arg which gives the color the errorbar lines; if *None*, use the color of the line connecting the markers.

***elinewidth*: scalar** The linewidth of the errorbar lines. If *None*, use the linewidth.

***capsize*: scalar** The length of the error bar caps in points; if *None*, it will take the value from `errorbar.capsize` *rcParam*.

***capthick*: scalar** An alias kwarg to *markeredgewidth* (a.k.a. - *mew*). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if *mew* or *markeredgewidth* are given, then they will over-ride *capthick*. This may change in future releases.

***barsabove*: [*True* | *False*]** if *True*, will plot the errorbars above the plot symbols. Default is below.

***lolims* / *uplims* / *xlolims* / *xuplims*: [*False* | *True*]** These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. lims-arguments may be of the same type as *xerr* and *yerr*. To use limits with inverted axes, `set_xlim()` or `set_ylim()` must be called before `errorbar()`.

***errorevery*: positive integer** subsamples the errorbars. e.g., if errorevery=5, errorbars for every 5-th datapoint will be plotted. The data plot itself still shows all data points.

All other keyword arguments are passed on to the plot command for the markers. For example, this code makes big red squares with thick green edges:

```
x,y,yerr = rand(3,10)
errorbar(x, y, yerr, marker='s',
         mfc='red', mec='green', ms=20, mew=4)
```

where *mfc*, *mec*, *ms* and *mew* are aliases for the longer property names, *markerfacecolor*, *markeredgecolor*, *markersize* and *markeredgewith*.

valid kwargs for the marker properties are

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']

C

Table 68.12 – continued from previous page

Property	Description
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '
linewidth or lw	float value in points
marker	A valid marker style
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

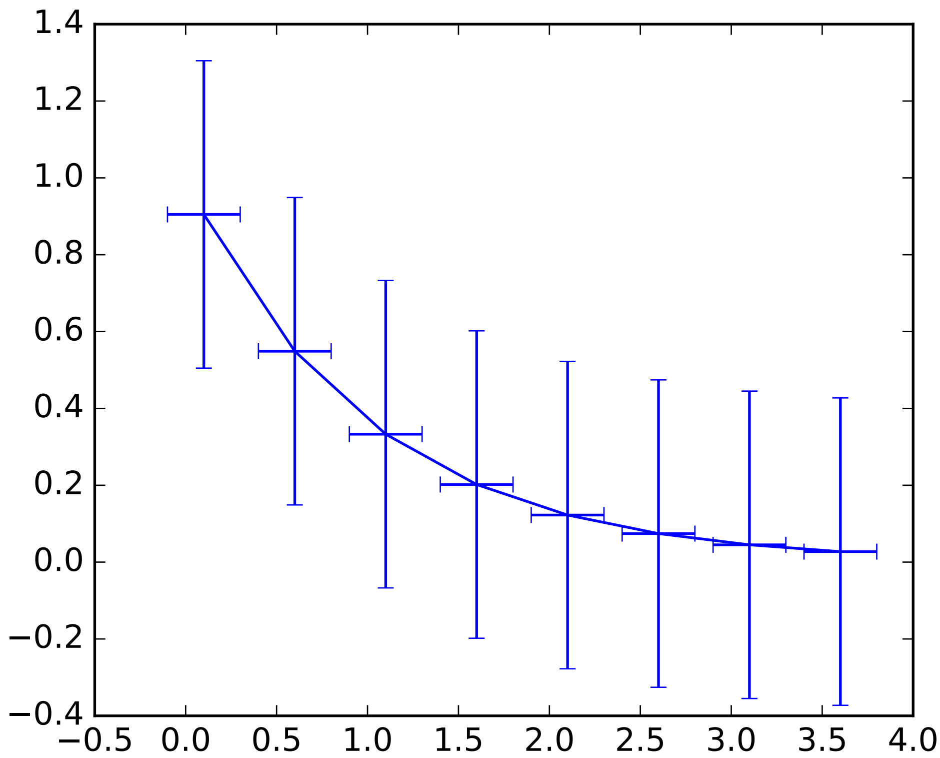
Returns (*plotline*, *caplines*, *barlinecols*):

***plotline*: Line2D instance** *x*, *y* plot markers and/or line

***caplines*: list of error bar cap** Line2D instances

***barlinecols*: list of** LineCollection instances for the horizontal and vertical error ranges.

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'yerr', 'y', 'xerr', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`eventplot`**`(`*positions*, *orientation='horizontal'*, *lineoffsets=1*, *linelengths=1*, *linewidths=None*, *colors=None*, *linestyles='solid'*, *hold=None*, *data=None*, ***kwargs*`)`

Plot identical parallel lines at specific positions.

Call signature:

```
eventplot(positions, orientation='horizontal', lineoffsets=0,
          linelengths=1, linewidths=None, color =None,
          linestyles='solid'
```

Plot parallel lines at the given positions. positions should be a 1D or 2D array-like object, with each row corresponding to a row or column of lines.

This type of plot is commonly used in neuroscience for representing neural events, where it is commonly called a spike raster, dot raster, or raster plot.

However, it is useful in any situation where you wish to show the timing or position of multiple sets of discrete events, such as the arrival times of people to a business on each day of the month or the date of hurricanes each year of the last century.

orientation [['horizonal' | 'vertical']] 'horizonal' : the lines will be vertical and arranged in rows "vertical' : lines will be horizontal and arranged in columns

lineoffsets **:** A float or array-like containing floats.

linelengths **:** A float or array-like containing floats.

linewidths **:** A float or array-like containing floats.

colors must be a sequence of RGBA tuples (e.g., arbitrary color strings, etc, not allowed) or a list of such sequences

linestyles **:** ['solid' | 'dashed' | 'dashdot' | 'dotted'] or an array of these values

For linelengths, linewidths, colors, and linestyles, if only a single value is given, that value is applied to all lines. If an array-like is given, it must have the same length as positions, and each value will be applied to the corresponding row or column in positions.

Returns a list of `matplotlib.collections.EventCollection` objects that were added.

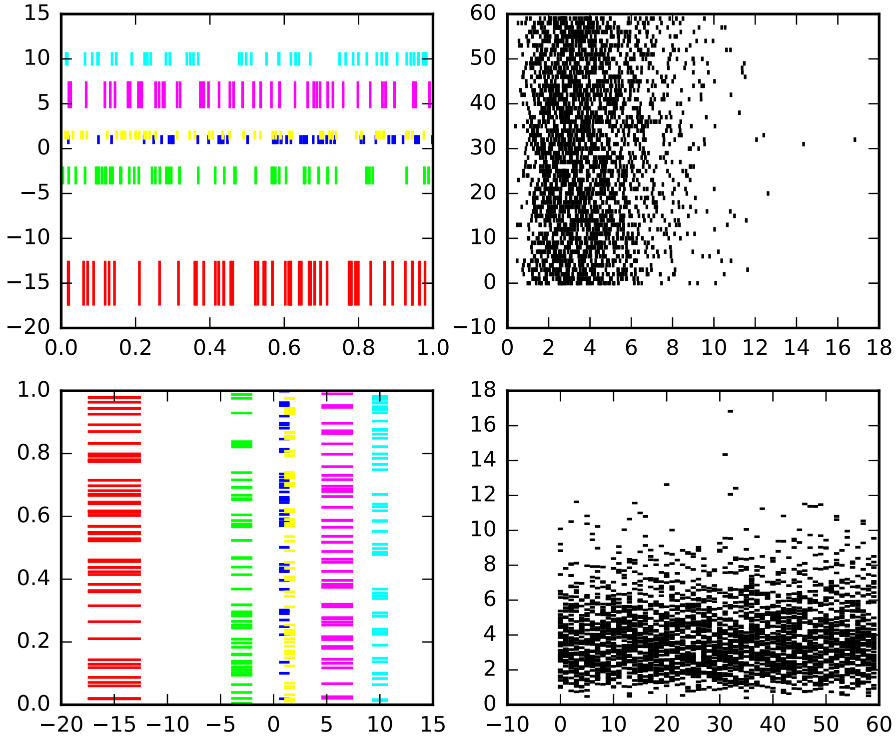
kwargs are `LineCollection` properties:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>agg_filter</td><td>unknown</td></tr>
<tr><td>alpha</td><td>float or None</td></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or antialiaseds</td><td>Boolean or sequence of booleans</td></tr>
<tr><td>array</td><td>unknown</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>clim</td><td>a length 2 sequence of floats</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>cmap</td><td>a colormap or registered colormap name</td></tr>
<tr><td>color</td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or edgecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td>facecolor or facecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>hatch</td><td>['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or linestyles or dashes</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' |</td></tr>
<tr><td>linewidth or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td>norm</td><td>unknown</td></tr>
<tr><td>offset_position</td><td>unknown</td></tr>
<tr><td>offsets</td><td>float or sequence of floats</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>paths</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
</table>

Table 68.13 – continued from previous page

Property	Description
pickradius	unknown
rasterized	[True | False | None]
segments	unknown
sketch_params	unknown
snap	unknown
transform	*Transform* instance
url	a url string
urls	unknown
verts	unknown
visible	[True | False]
zorder	any number

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'linelengths', 'linestyles', 'linewidths', 'positions', 'lineoffsets', 'colors'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.figimage(*args, **kwargs)`

Adds a non-resampled image to the figure.

call signatures:

```
figimage(X, **kwargs)
```

adds a non-resampled array *X* to the figure.

```
figimage(X, xo, yo)
```

with pixel offsets *xo*, *yo*,

X must be a float array:

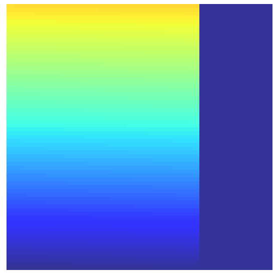
- If *X* is MxN, assume luminance (grayscale)
- If *X* is MxNx3, assume RGB
- If *X* is MxNx4, assume RGBA

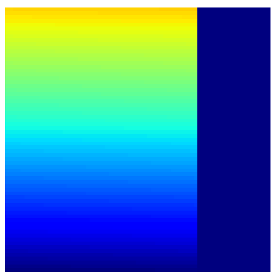
Optional keyword arguments:

Keyword	Description
resize	a boolean, True or False. If "True", then re-size the Figure to match the given image size.
xo or yo	An integer, the *x* and *y* image offset in pixels
cmap	a `matplotlib.colors.Colormap` instance, e.g., cm.jet. If *None*, default to the rc `image.cmap` value
norm	a `matplotlib.colors.Normalize` instance. The default is normalization(). This scales luminance -> 0-1
vmin|vmax	are used to scale a luminance image to 0-1. If either is *None*, the min and max of the luminance values will be used. Note if you pass a norm instance, the settings for *vmin* and *vmax* will be ignored.
alpha	the alpha blending value, default is *None*
origin	['upper' | 'lower'] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc image.origin value

figimage complements the axes image (`imshow()`) which will be resampled to fit the current axes. If you want a resampled image to fill the entire figure, you can define an `Axes` with size [0,1,0,1].

An `matplotlib.image.FigureImage` instance is returned.





Additional kwargs are Artist kwargs passed on to *FigureImage* Addition kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`figlegend`**(*handles*, *labels*, *loc*, ***kwargs*)

Place a legend in the figure.

labels a sequence of strings

handles a sequence of *Line2D* or *Patch* instances

loc can be a string or an integer specifying the legend location

A *matplotlib.legend.Legend* instance is returned.

Example:

```
figlegend( (line1, line2, line3),
           ('label1', 'label2', 'label3'),
           'upper right' )
```

See also:

legend()

`matplotlib.pyplot.`**`fignum_exists`**(*num*)

`matplotlib.pyplot.`**`figtext`**(**args*, ***kwargs*)

Add text to figure.

Call signature:

```
text(x, y, s, fontdict=None, **kwargs)
```

Add text to figure at location *x*, *y* (relative 0-1 coords). See `text()` for the meaning of the other arguments.

kwargs control the `Text` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`axes`	an `Axes` instance
`backgroundcolor`	any matplotlib color
`bbox`	FancyBboxPatch prop dict
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	any matplotlib color
`contains`	a callable function
`family` or fontfamily or name or fontname	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
`figure`	a `matplotlib.figure.Figure` instance
`fontproperties` or font_properties	a `matplotlib.font_manager.FontProperties` instance
`gid`	an id string
`horizontalalignment` or ha	['center' | 'right' | 'left']
`label`	string or anything printable with '%s' conversion.
`linespacing`	float (multiple of font size)
`multialignment`	['left' | 'right' | 'center']
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`position`	(x,y)
`rasterized`	[True | False | None]
`rotation`	[angle in degrees | 'vertical' | 'horizontal']
`rotation_mode`	unknown
`size` or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large'
`sketch_params`	unknown
`snap`	unknown
`stretch` or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'c
`style` or fontstyle	['normal' | 'italic' | 'oblique']
`text`	string or anything printable with '%s' conversion.
`transform`	`Transform` instance
`url`	a url string
`usetex`	unknown
`variant` or fontvariant	['normal' | 'small-caps']
`verticalalignment` or va or ma	['center' | 'top' | 'bottom' | 'baseline']
`visible`	[True | False]
`weight` or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' |

Table 68.14 – continued fro

Property	Description
wrap	unknown
x	float
y	float
zorder	any number

`matplotlib.pyplot.figure`(*num=None*, *figsize=None*, *dpi=None*, *facecolor=None*, *edgecolor=None*, *frameon=True*, *FigureClass=<class 'matplotlib.figure.Figure'>*, ***kwargs*)

Creates a new figure.

Parameters num : integer or string, optional, default: none

If not provided, a new figure will be created, and the figure number will be incremented. The figure objects holds this number in a `number` attribute. If num is provided, and a figure with this id already exists, make it active, and returns a reference to it. If this figure does not exists, create it and returns it. If num is a string, the window title will be set to this figure's `num`.

figsize : tuple of integers, optional, default: None

width, height in inches. If not provided, defaults to rc figure.figsize.

dpi : integer, optional, default: None

resolution of the figure. If not provided, defaults to rc figure.dpi.

facecolor :

the background color. If not provided, defaults to rc figure.facecolor

edgecolor :

the border color. If not provided, defaults to rc figure.edgecolor

Returns figure : Figure

The Figure instance returned will also be passed to new_figure_manager in the backends, which allows to hook custom Figure classes into the pylab interface. Additional kwargs will be passed to the figure init function.

Notes

If you are creating many figures, make sure you explicitly call "close" on the figures you are not using, because this will enable pylab to properly clean up the memory.

rcParams defines the default values, which can be modified in the matplotlibrc file

`matplotlib.pyplot.fill`(**args*, ***kwargs*)

Plot filled polygons.

Call signature:

```
fill(*args, **kwargs)
```

args is a variable length argument, allowing for multiple *x*, *y* pairs with an optional color format string; see `plot()` for details on the argument parsing. For example, to plot a polygon with vertices at *x*, *y* in blue.:

```
ax.fill(x,y, 'b' )
```

An arbitrary number of *x*, *y*, *color* groups can be specified:

```
ax.fill(x1, y1, 'g', x2, y2, 'r')
```

Return value is a list of `Patch` instances that were added.

The same color strings that `plot()` supports are supported by the fill format string.

If you would like to fill below a curve, e.g., shade a region between 0 and *y* along *x*, use `fill_between()`

The *closed* kwarg will close the polygon when *True* (default).

kwargs control the `Polygon` properties:

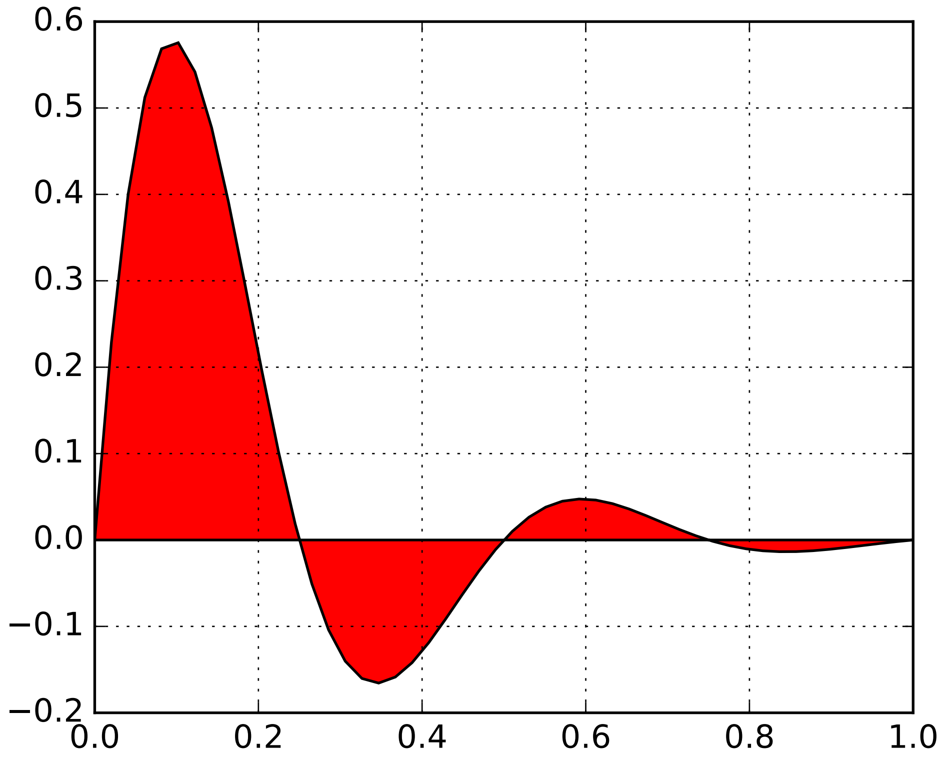
Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or aa	[True | False] or None for default
`axes`	an `Axes` instance
`capstyle`	['butt' | 'round' | 'projecting']
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color`	matplotlib color spec
`contains`	a callable function
`edgecolor` or ec	mpl color spec, or None for default, or 'none' for no color
`facecolor` or fc	mpl color spec, or None for default, or 'none' for no color
`figure`	a `matplotlib.figure.Figure` instance
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance

Continued on

Table 68.15 – continued from previous page

Property	Description
url	a url string
visible	[True | False]
zorder	any number

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.fill_between`(*x*, *y1*, *y2=0*, *where=None*, *interpolate=False*, *step=None*, *hold=None*, *data=None*, ***kwargs*)

Make filled polygons between two curves.

Create a *PolyCollection* filling the regions between *y1* and *y2* where `where==True`

Parameters x : array

An N-length array of the x data

y1 : array
    An N-length array (or scalar) of the y data

y2 : array
    An N-length array (or scalar) of the y data

where : array, optional
    If `None`, default to fill between everywhere. If not `None`, it is an N-length numpy boolean array and the fill will only happen over the regions where `where==True`.

interpolate : bool, optional
    If `True`, interpolate between the two lines to find the precise point of intersection. Otherwise, the start and end points of the filled region will only occur on explicit values in the *x* array.

step : {'pre', 'post', 'mid'}, optional
    If not None, fill with step logic.

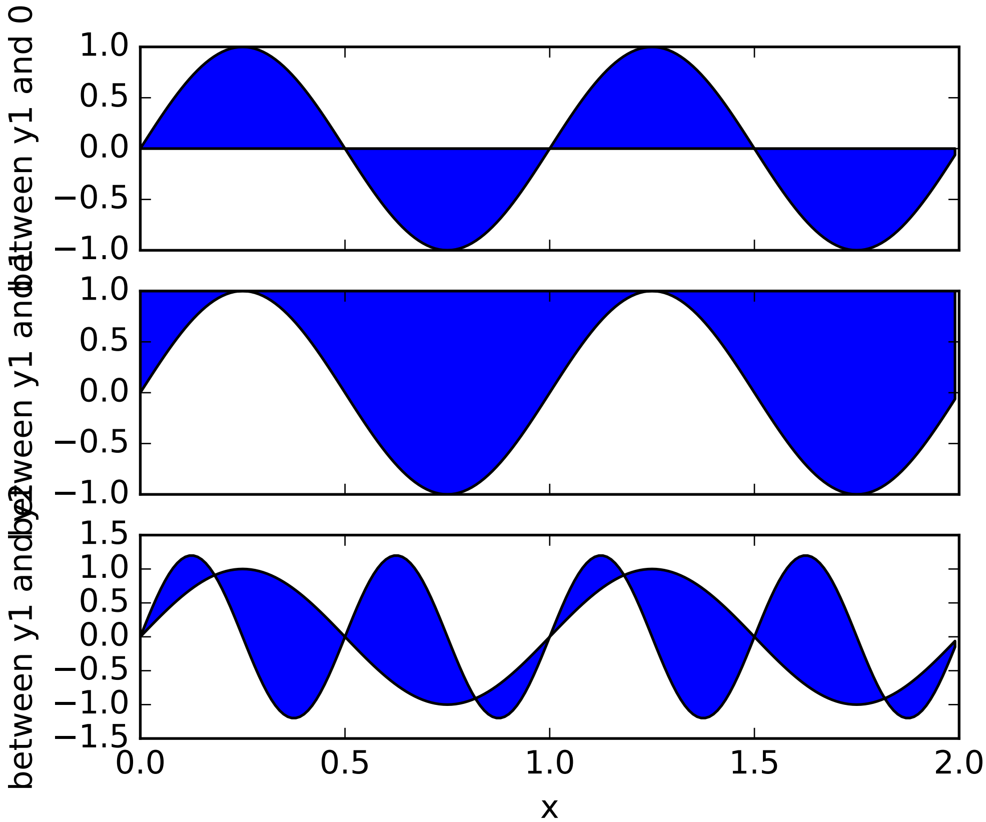
Notes

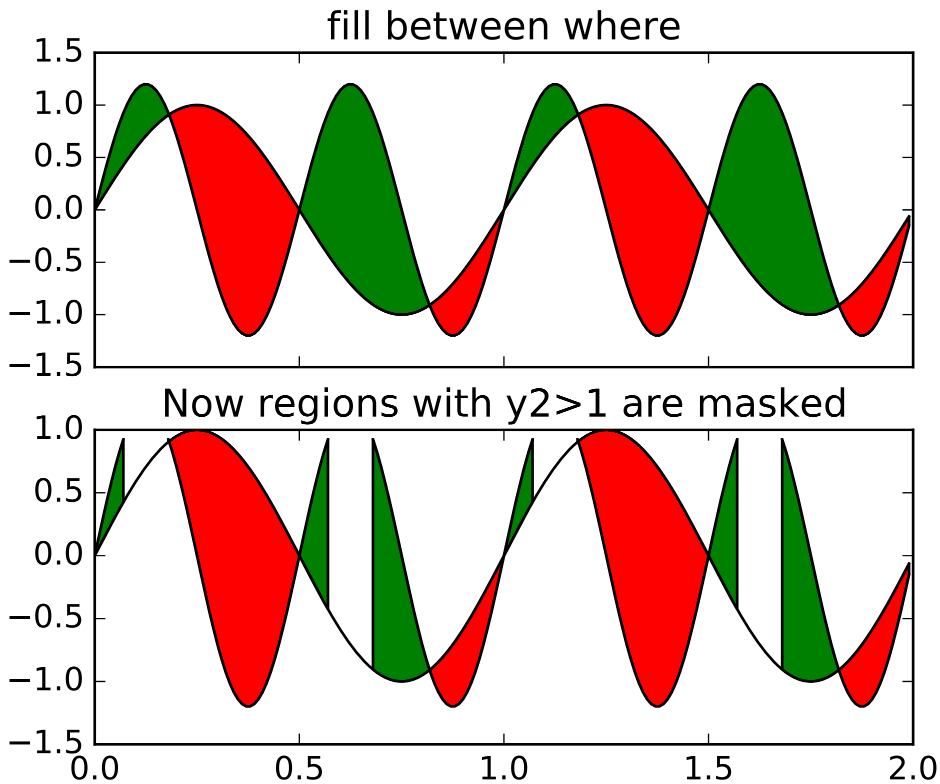
In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

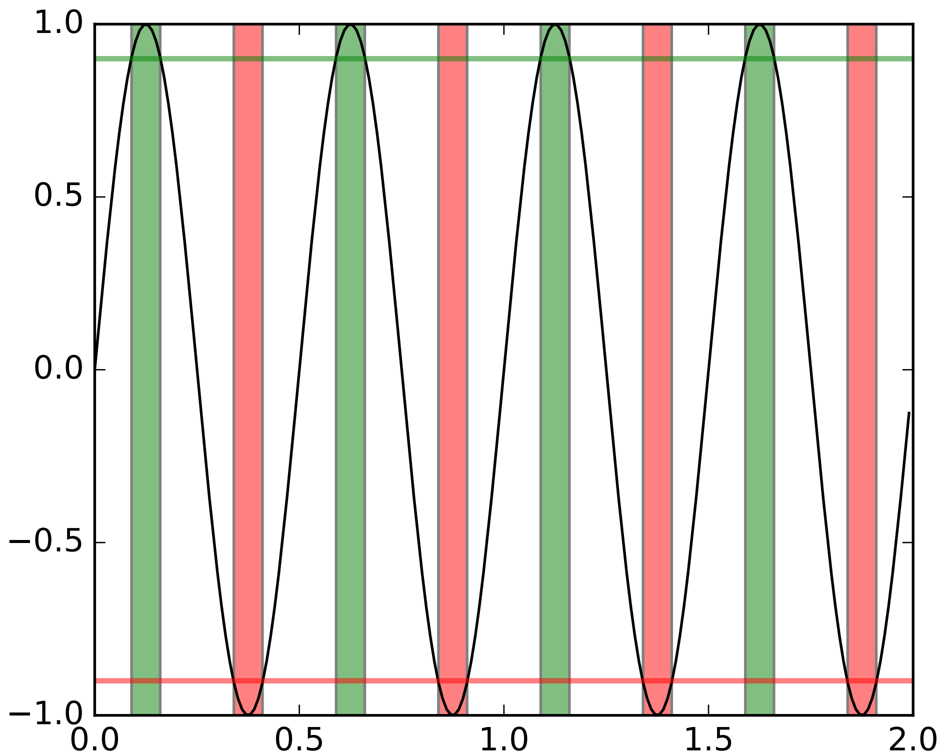
- All arguments with the following names: 'y1', 'y2', 'where', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

Examples


between y1 and 0
between y1 and 1
between y1 and y2
1.0
0.5
0.0
−0.5
−1.0
1.5
−1.5
0.0
0.5
1.0
1.5
2.0
x


fill between where
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Now regions with y2>1 are masked
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.5
1.0
1.5
2.0



`matplotlib.pyplot.fill_betweenx`(*y*, *x1*, *x2=0*, *where=None*, *step=None*, *hold=None*, *data=None*, ***kwargs*)

Make filled polygons between two horizontal curves.

Call signature:

```
fill_betweenx(y, x1, x2=0, where=None, **kwargs)
```

Create a *PolyCollection* filling the regions between *x1* and *x2* where `where==True`

Parameters y : array

An N-length array of the y data

x1 : array

An N-length array (or scalar) of the x data

x2 : array, optional

An N-length array (or scalar) of the x data

where : array, optional

If *None*, default to fill between everywhere. If not *None*, it is a N length numpy boolean array and the fill will only happen over the regions where `where==True`

step : {'pre', 'post', 'mid'}, optional

If not None, fill with step logic.

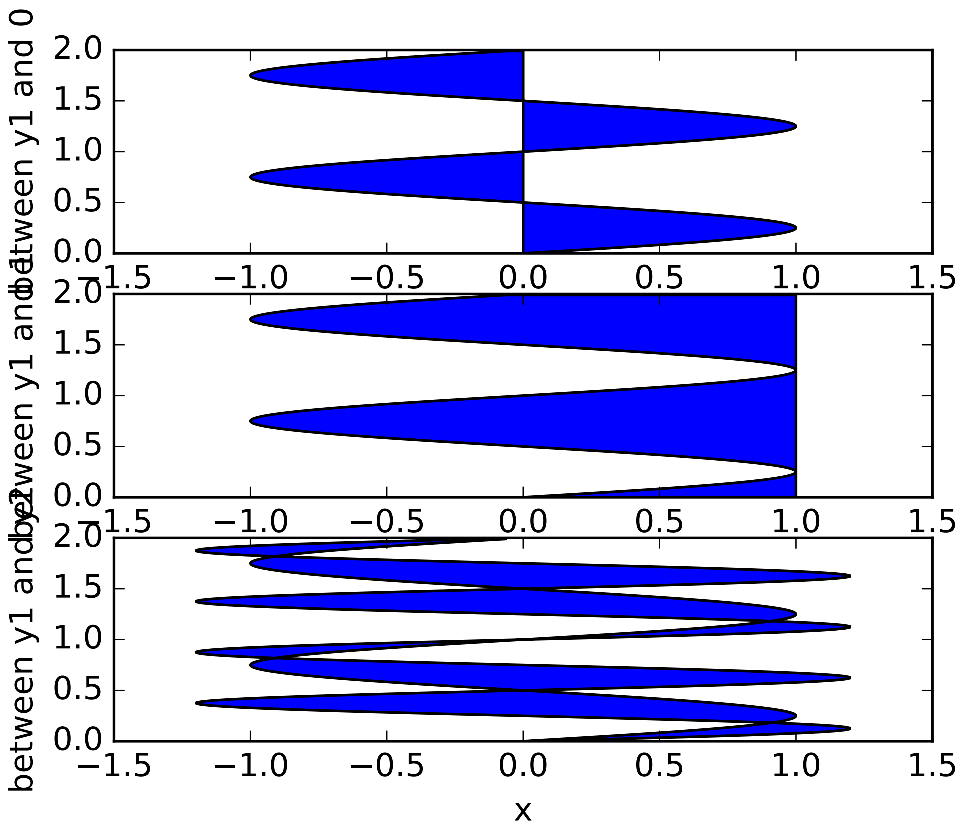
Notes

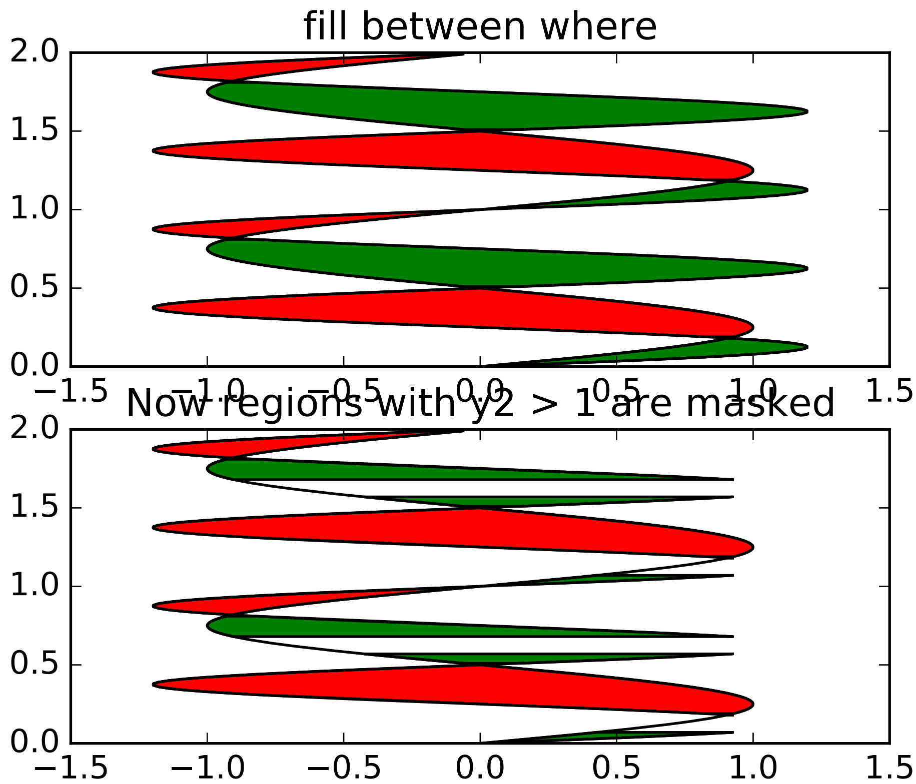
In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x1', 'where', 'y', 'x2'.

Additional kwargs: hold = [True|False] overrides default hold state

Examples





matplotlib.pyplot.**findobj**(*o=None*, *match=None*, *include_self=True*)

Find artist objects.

Recursively find all `Artist` instances contained in self.

match can be

- None: return all objects contained in artist.
- function with signature `boolean = match(artist)` used to filter matches
- class instance: e.g., Line2D. Only return artists of class type.

If *include_self* is True (default), include self in the list to be checked for a match.

matplotlib.pyplot.**flag**()

set the default colormap to flag and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.**gca**(***kwargs*)

Get the current `Axes` instance on the current figure matching the given keyword args, or create one.

See also:

`matplotlib.figure.Figure.gca` The figure's gca method.

Examples

To get the current polar axes on the current figure:

```
plt.gca(projection='polar')
```

If the current axes doesn't exist, or isn't a polar one, the appropriate axes will be created and then returned.

`matplotlib.pyplot.gcf()`
Get a reference to the current figure.

`matplotlib.pyplot.gci()`
Get the current colorable artist. Specifically, returns the current `ScalarMappable` instance (image or patch collection), or *None* if no images or patch collections have been defined. The commands `imshow()` and `figimage()` create `Image` instances, and the commands `pcolor()` and `scatter()` create `Collection` instances. The current image is an attribute of the current axes, or the nearest earlier axes in the current figure that contains an image.

`matplotlib.pyplot.get_current_fig_manager()`

`matplotlib.pyplot.get_figlabels()`
Return a list of existing figure labels.

`matplotlib.pyplot.get_fignums()`
Return a list of existing figure numbers.

`matplotlib.pyplot.get_plot_commands()`
Get a sorted list of all of the plotting commands.

`matplotlib.pyplot.ginput(*args, **kwargs)`
Call signature:

```
ginput(self, n=1, timeout=30, show_clicks=True,
       mouse_add=1, mouse_pop=3, mouse_stop=2)
```

Blocking call to interact with the figure.

This will wait for *n* clicks from the user and return a list of the coordinates of each click.

If *timeout* is zero or negative, does not timeout.

If *n* is zero or negative, accumulate clicks until a middle click (or potentially both mouse buttons at once) terminates the input.

Right clicking cancels last input.

The buttons used for the various actions (adding points, removing points, terminating the inputs) can be overriden via the arguments *mouse_add*, *mouse_pop* and *mouse_stop*, that give the associated mouse button: 1 for left, 2 for middle, 3 for right.

The keyboard can also be used to select points in case your mouse does not have one or more of the buttons. The delete and backspace keys act like right clicking (i.e., remove last point), the enter key terminates input and any other key (not already used by the window manager) selects a point.

`matplotlib.pyplot.gray()`
set the default colormap to gray and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.**grid**(*b=None*, *which='major'*, *axis='both'*, ***kwargs*)

Turn the axes grids on or off.

Call signature:

```
grid(self, b=None, which='major', axis='both', **kwargs)
```

Set the axes grids on or off; *b* is a boolean. (For MATLAB compatibility, *b* may also be a string, 'on' or 'off'.)

If *b* is *None* and `len(kwargs)==0`, toggle the grid state. If *kwargs* are supplied, it is assumed that you want a grid and *b* is thus set to *True*.

which can be 'major' (default), 'minor', or 'both' to control whether major tick grids, minor tick grids, or both are affected.

axis can be 'both' (default), 'x', or 'y' to control which set of gridlines are drawn.

kwargs are used to set the grid line properties, e.g.,:

```
ax.grid(color='r', linestyle='-', linewidth=2)
```

Valid `Line2D` kwargs are

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
	C

Table 68.16 – continued from previous page

Property	Description
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a *matplotlib.transforms.Transform* instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

matplotlib.pyplot.**hexbin**(*x, y, C=None, gridsize=100, bins=None, xscale='linear', yscale='linear', extent=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors='none', reduce_C_function=<function mean>, mincnt=None, marginals=False, hold=None, data=None, **kwargs*)

Make a hexagonal binning plot.

Call signature:

```
hexbin(x, y, C = None, gridsize = 100, bins = None,
       xscale = 'linear', yscale = 'linear',
       cmap=None, norm=None, vmin=None, vmax=None,
       alpha=None, linewidths=None, edgecolors='none'
       reduce_C_function = np.mean, mincnt=None, marginals=True
       **kwargs)
```

Make a hexagonal binning plot of *x* versus *y*, where *x*, *y* are 1-D sequences of the same length, *N*. If *C* is *None* (the default), this is a histogram of the number of occurences of the observations at (x[i],y[i]).

If *C* is specified, it specifies values at the coordinate (x[i],y[i]). These values are accumulated for each hexagonal bin and then reduced according to *reduce_C_function*, which defaults to numpy's mean function (np.mean). (If *C* is specified, it must also be a 1-D sequence of the same length as *x* and *y*.)

x, *y* and/or *C* may be masked arrays, in which case only unmasked points will be plotted.

Optional keyword arguments:

***gridsize*: [100 | integer]** The number of hexagons in the *x*-direction, default is 100. The corresponding number of hexagons in the *y*-direction is chosen such that the hexagons are approximately regular. Alternatively, gridsize can be a tuple with two elements specifying the number of hexagons in the *x*-direction and the *y*-direction.

***bins*: [*None* | 'log' | integer | sequence]** If *None*, no binning is applied; the color of each hexagon directly corresponds to its count value.

If 'log', use a logarithmic scale for the color map. Internally, $log_{10}(i+1)$ is used to determine the hexagon color.

If an integer, divide the counts in the specified number of bins, and color the hexagons accordingly.

If a sequence of values, the values of the lower bound of the bins to be used.

***xscale*: ['linear' | 'log']** Use a linear or log10 scale on the horizontal axis.

***scale*: ['linear' | 'log']** Use a linear or log10 scale on the vertical axis.

***mincnt*: [*None* | a positive integer]** If not *None*, only display cells with more than *mincnt* number of points in the cell

***marginals*: [*True* | *False*]** if marginals is *True*, plot the marginal density as colormapped rectagles along the bottom of the x-axis and left of the y-axis

***extent*: [*None* | scalars (left, right, bottom, top)]** The limits of the bins. The default assigns the limits based on gridsize, x, y, xscale and yscale.

Other keyword arguments controlling color mapping and normalization arguments:

***cmap*: [*None* | Colormap]** a `matplotlib.colors.Colormap` instance. If *None*, defaults to rc `image.cmap`.

***norm*: [*None* | Normalize]** `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1.

***vmin* / *vmax*: scalar** *vmin* and *vmax* are used in conjunction with *norm* to normalize luminance data. If either are *None*, the min and max of the color array *C* is used. Note if you pass a norm instance, your settings for *vmin* and *vmax* will be ignored.

alpha*: scalar between 0 and 1, or *None the alpha value for the patches

***linewidths*: [*None* | scalar]** If *None*, defaults to rc lines.linewidth. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by `RegularPolyCollection`.

Other keyword arguments controlling the Collection properties:

***edgecolors*: [*None* | `'none'` | mpl color | color sequence]** If `'none'`, draws the edges in the same color as the fill color. This is the default, as it avoids unsightly unpainted pixels between the hexagons.

If *None*, draws the outlines in the default color.

If a matplotlib color arg or sequence of rgba tuples, draws the outlines in the specified color.

Here are the standard descriptions of all the `Collection` kwargs:

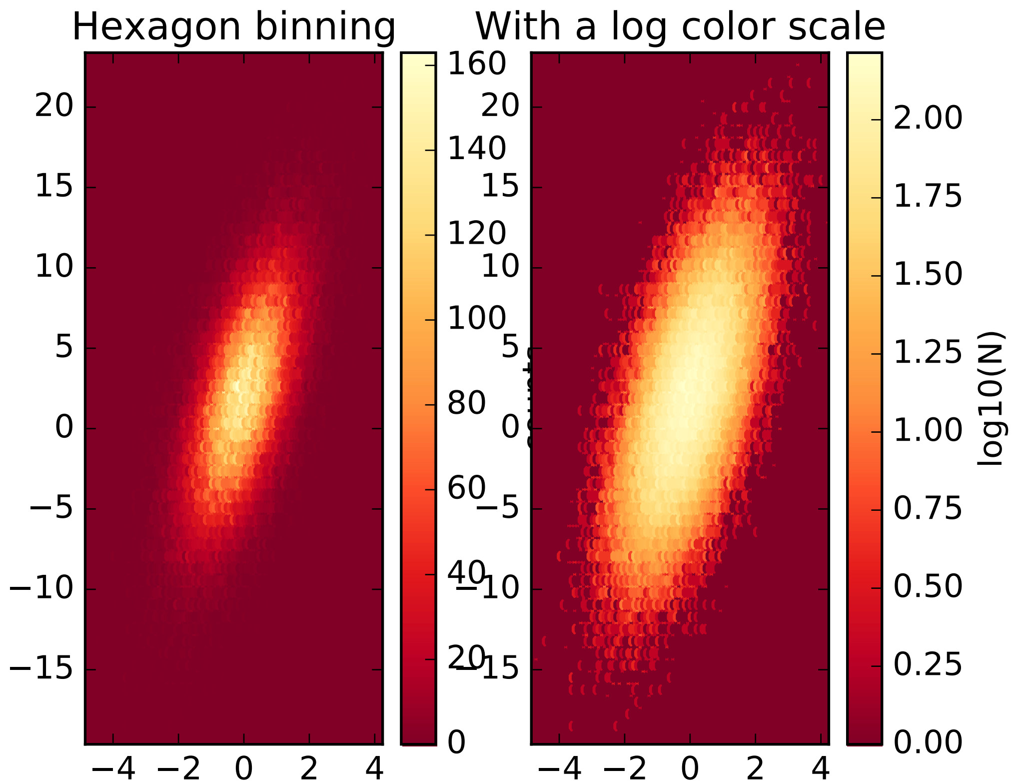
Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance

Table 68.17 – continued from previous page

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td><code>clim</code></td><td>a length 2 sequence of floats</td></tr>
<tr><td><code>clip_box</code></td><td>a <code>matplotlib.transforms.Bbox</code> instance</td></tr>
<tr><td><code>clip_on</code></td><td>[True | False]</td></tr>
<tr><td><code>clip_path</code></td><td>[(<code>Path</code>, <code>Transform</code>) | <code>Patch</code> | None]</td></tr>
<tr><td><code>cmap</code></td><td>a colormap or registered colormap name</td></tr>
<tr><td><code>color</code></td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td><code>contains</code></td><td>a callable function</td></tr>
<tr><td><code>edgecolor</code> or edgecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>facecolor</code> or facecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>figure</code></td><td>a <code>matplotlib.figure.Figure</code> instance</td></tr>
<tr><td><code>gid</code></td><td>an id string</td></tr>
<tr><td><code>hatch</code></td><td>[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]</td></tr>
<tr><td><code>label</code></td><td>string or anything printable with ‘%s’ conversion.</td></tr>
<tr><td><code>linestyle</code> or linestyles or dashes</td><td>[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> |</td></tr>
<tr><td><code>linewidth</code> or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td><code>norm</code></td><td>unknown</td></tr>
<tr><td><code>offset_position</code></td><td>unknown</td></tr>
<tr><td><code>offsets</code></td><td>float or sequence of floats</td></tr>
<tr><td><code>path_effects</code></td><td>unknown</td></tr>
<tr><td><code>picker</code></td><td>[None|float|boolean|callable]</td></tr>
<tr><td><code>pickradius</code></td><td>unknown</td></tr>
<tr><td><code>rasterized</code></td><td>[True | False | None]</td></tr>
<tr><td><code>sketch_params</code></td><td>unknown</td></tr>
<tr><td><code>snap</code></td><td>unknown</td></tr>
<tr><td><code>transform</code></td><td><code>Transform</code> instance</td></tr>
<tr><td><code>url</code></td><td>a url string</td></tr>
<tr><td><code>urls</code></td><td>unknown</td></tr>
<tr><td><code>visible</code></td><td>[True | False]</td></tr>
<tr><td><code>zorder</code></td><td>any number</td></tr>
</table>

The return value is a `PolyCollection` instance; use `get_array()` on this `PolyCollection` to get the counts in each hexagon. If *marginals* is *True*, horizontal bar and vertical bar (both PolyCollections) will be attached to the return collection as attributes *hbar* and *vbar*.

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.hist`(*x, bins=10, range=None, normed=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, hold=None, data=None, **kwargs*)

Plot a histogram.

Compute and draw the histogram of *x*. The return value is a tuple (*n*, *bins*, *patches*) or ([*n0*, *n1*, ...], *bins*, [*patches0*, *patches1*,...]) if the input contains multiple data.

Multiple data can be provided via *x* as a list of datasets of potentially different length ([*x0*, *x1*, ...]), or as a 2-D ndarray in which each column is a dataset. Note that the ndarray form is transposed relative to the list form.

Masked arrays are not supported at present.

Parameters x : (n,) array or sequence of (n,) arrays

Input values, this takes either a single array or a sequency of arrays which are not required to be of the same length

bins : integer or array_like, optional
If an integer is given, `bins + 1` bin edges are returned, consistently with `numpy.histogram()` for numpy version >= 1.3.

Unequally spaced bins are supported if `bins` is a sequence.

default is 10

range : tuple or None, optional
The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, `range` is (x.min(), x.max()). Range has no effect if `bins` is a sequence.

If `bins` is a sequence or `range` is specified, autoscaling is based on the specified bin range instead of the range of x.

Default is `None`

normed : boolean, optional
If `True`, the first element of the return tuple will be the counts normalized to form a probability density, i.e., `n/(len(x)‘dbin)`, i.e., the integral of the histogram will sum to 1. If *stacked* is also *True*, the sum of the histograms is normalized to 1.

Default is `False`

weights : (n,) array_like or None, optional
An array of weights, of the same shape as `x`. Each value in `x` only contributes its associated weight towards the bin count (instead of 1). If `normed` is True, the weights are normalized, so that the integral of the density over the range remains 1.

Default is `None`

cumulative : boolean, optional
If `True`, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints. If `normed` is also `True` then the histogram is normalized such that the last bin equals 1. If `cumulative` evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if `normed` is also `True`, then the histogram is normalized such that the first bin equals 1.

Default is `False`

bottom : array_like, scalar, or None
Location of the bottom baseline of each bin. If a scalar, the base line for each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

Default is `None`

histtype : {‘bar’, ‘barstacked’, ‘step’, ‘stepfilled’}, optional
The type of histogram to draw.

- ‘bar’ is a traditional bar-type histogram. If multiple data are given the bars are aranged side by side.

- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

Default is 'bar'

align : {'left', 'mid', 'right'}, optional

Controls how the histogram is plotted.

- 'left': bars are centered on the left bin edges.
- 'mid': bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

Default is 'mid'

orientation : {'horizontal', 'vertical'}, optional

If 'horizontal', `barh` will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

rwidth : scalar or None, optional

The relative width of the bars as a fraction of the bin width. If `None`, automatically compute the width.

Ignored if `histtype` is 'step' or 'stepfilled'.

Default is `None`

log : boolean, optional

If `True`, the histogram axis will be set to a log scale. If `log` is `True` and `x` is a 1D array, empty bins will be filtered out and only the non-empty (`n`, `bins`, `patches`) will be returned.

Default is `False`

color : color or array_like of colors or None, optional

Color spec or sequence of color specs, one per dataset. Default (`None`) uses the standard line color sequence.

Default is `None`

label : string or None, optional

String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that the legend command will work as expected.

default is `None`

stacked : boolean, optional

If `True`, multiple data are stacked on top of each other If `False` multiple data are aranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

Default is `False`

Returns n : array or list of arrays

The values of the histogram bins. See **normed** and **weights** for a description of the possible semantics. If input **x** is an array, then this is an array of length **nbins**. If input is a sequence arrays `[data1, data2,..]`, then this is a list of arrays with the values of the histograms for each of the arrays in the same order.

bins : array

The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

patches : list or list of lists
Silent list of individual patches used to create the histogram or list of such list if multiple input datasets.

Other Parameters kwargs : `Patch` properties

See also:

`hist2d` 2D histograms

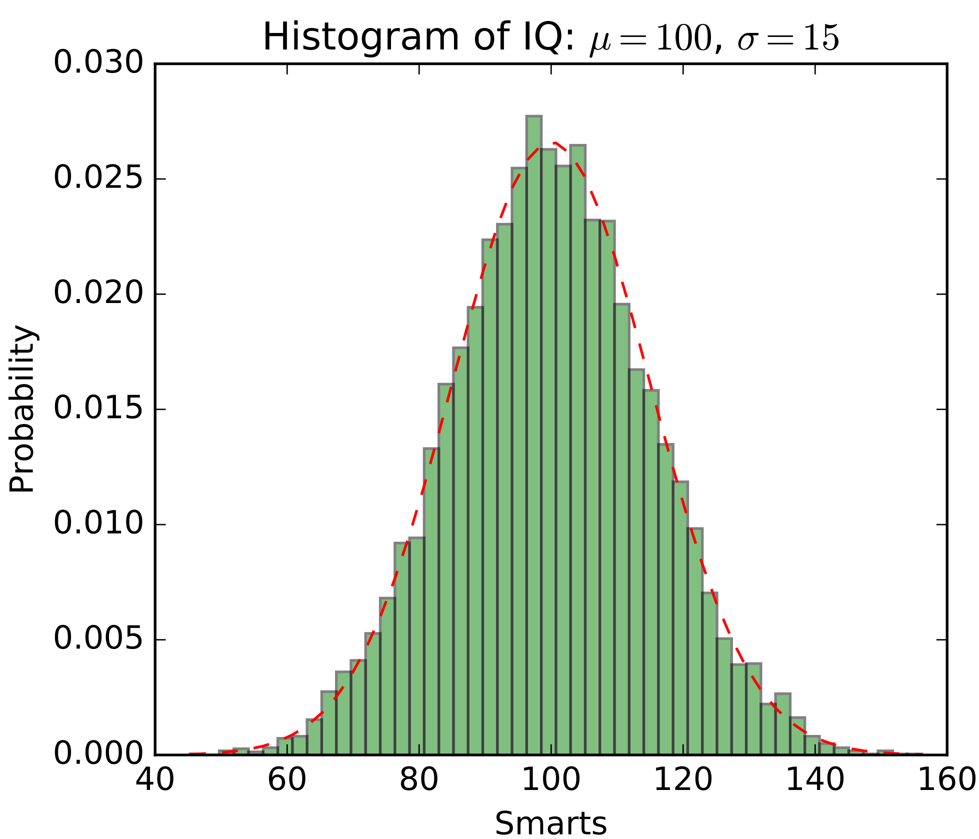
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'weights', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

Examples



`matplotlib.pyplot.`**`hist2d`**(*x*, *y*, *bins=10*, *range=None*, *normed=False*, *weights=None*, *cmin=None*, *cmax=None*, *hold=None*, *data=None*, ***kwargs*)

Make a 2D histogram plot.

Parameters x, y: array_like, shape (n,)

Input values

bins: [None | int | [int, int] | array_like | [array, array]]

The bin specification:

- If int, the number of bins for the two dimensions (nx=ny=bins).
- If [int, int], the number of bins in each dimension (nx, ny = bins).
- If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).
- If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).

The default value is 10.

range : array_like shape(2, 2), optional, default: None

The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): [[xmin, xmax], [ymin, ymax]]. All values outside of this range will be considered outliers and not tallied in the histogram.

normed : boolean, optional, default: False

Normalize histogram.

weights : array_like, shape (n,), optional, default: None

An array of values w_i weighing each sample (x_i, y_i).

cmin : scalar, optional, default: None

All bins that has count less than cmin will not be displayed and these count values in the return value count histogram will also be set to nan upon return

cmax : scalar, optional, default: None

All bins that has count more than cmax will not be displayed (set to none before passing to imshow) and these count values in the return value count histogram will also be set to nan upon return

Returns The return value is `(counts, xedges, yedges, Image)`.

Other Parameters kwargs : `pcolorfast()` properties.

See also:

`hist` 1D histogram

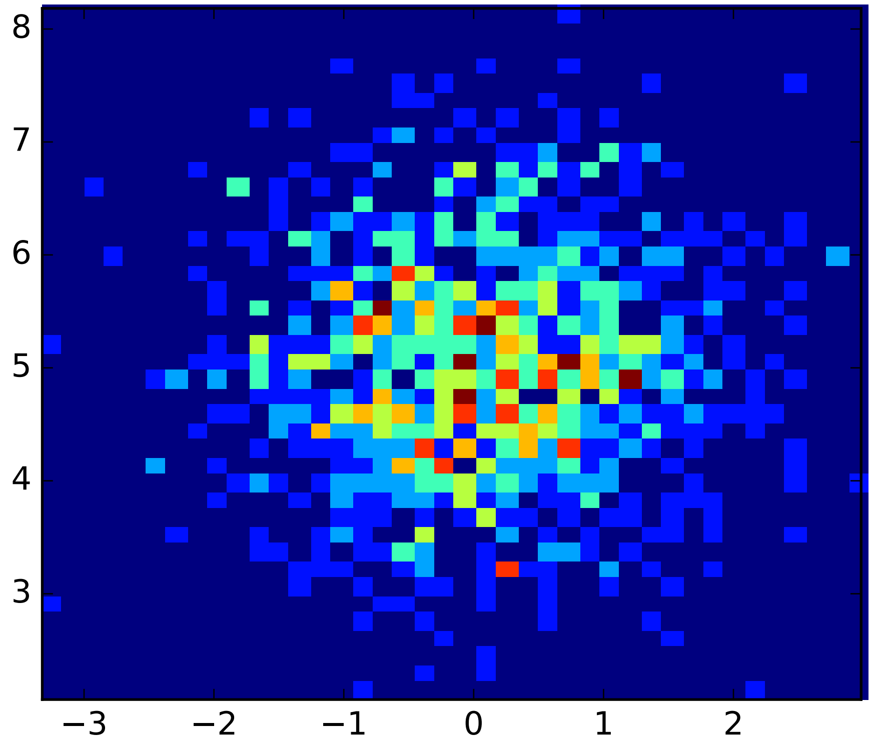
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: ‘y’, ‘weights’, ‘x’.

Additional kwargs: hold = [True|False] overrides default hold state

Examples



`matplotlib.pyplot.`**`hlines`**(*y*, *xmin*, *xmax*, *colors='k'*, *linestyles='solid'*, *label=''*, *hold=None*, *data=None*, ***kwargs*)

Plot horizontal lines at each `y` from `xmin` to `xmax`.

Parameters y : scalar or sequence of scalar

y-indexes where to plot the lines.

xmin, xmax : scalar or 1D array_like

Respective beginning and end of each line. If scalars are provided, all lines will have same length.

colors : array_like of colors, optional, default: 'k'

linestyles : ['solid' | 'dashed' | 'dashdot' | 'dotted'], optional

label : string, optional, default: ''

Returns lines : *LineCollection*

Other Parameters kwargs : *LineCollection* properties.

See also:

vlines vertical lines

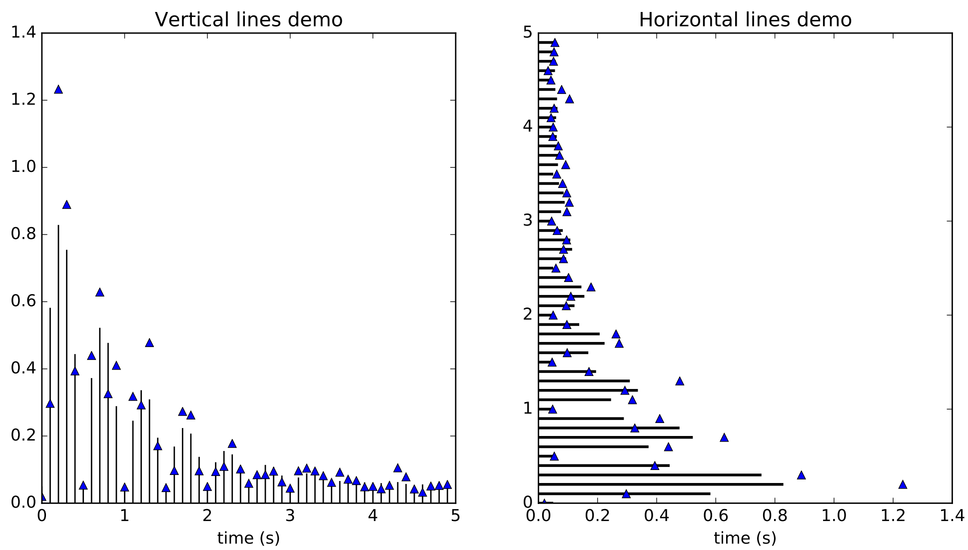
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'xmin', 'xmax', 'y'.

Additional kwargs: hold = [True|False] overrides default hold state

Examples



`matplotlib.pyplot.hold(b=None)`

Set the hold state. If *b* is None (default), toggle the hold state, else set the hold state to boolean value *b*:

```
hold()      # toggle hold
hold(True)  # hold is on
hold(False) # hold is off
```

When *hold* is *True*, subsequent plot commands will be added to the current axes. When *hold* is *False*, the current axes and figure will be cleared on the next plot command.

`matplotlib.pyplot.hot()`

set the default colormap to hot and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.hsv()`

set the default colormap to hsv and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.imread(*args, **kwargs)`

Read an image from a file into an array.

fname may be a string path, a valid URL, or a Python file-like object. If using a file object, it must be opened in binary mode.

If *format* is provided, will try to read file of that type, otherwise the format is deduced from the filename. If nothing can be deduced, PNG is tried.

Return value is a `numpy.array`. For grayscale images, the return array is MxN. For RGB images, the return value is MxNx3. For RGBA images the return value is MxNx4.

matplotlib can only read PNGs natively, but if PIL is installed, it will use it to load the image and return an array (if possible) which can be used with `imshow()`. Note, URL strings may not be compatible with PIL. Check the PIL documentation for more information.

`matplotlib.pyplot.imsave(*args, **kwargs)`

Save an array as in image file.

The output formats available depend on the backend being used.

Arguments:

***fname*:** A string containing a path to a filename, or a Python file-like object. If *format* is *None* and *fname* is a string, the output format is deduced from the extension of the filename.

***arr*:** An MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA) array.

Keyword arguments:

***vmin/vmax*: [None | scalar]** *vmin* and *vmax* set the color scaling for the image by fixing the values that map to the colormap color limits. If either *vmin* or *vmax* is None, that limit is determined from the *arr* min/max value.

***cmap*:** cmap is a colors.Colormap instance, e.g., cm.jet. If None, default to the rc image.cmap value.

***format*:** One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

origin ['upper' | 'lower'] Indicates where the [0,0] index of the array is in the upper left or lower left corner of the axes. Defaults to the rc image.origin value.

dpi The DPI to store in the metadata of the file. This does not affect the resolution of the output image.

`matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None, resample=None, url=None, hold=None, data=None, **kwargs)`

Display an image on the axes.

Parameters X : array_like, shape (n, m) or (n, m, 3) or (n, m, 4)

Display the image in `X` to current axes. `X` may be a float array, a uint8 array or a PIL image. If `X` is an array, it can have the following shapes:

- MxN – luminance (grayscale, float array only)
- MxNx3 – RGB (float or uint8 array)
- MxNx4 – RGBA (float or uint8 array)

The value for each component of MxNx3 and MxNx4 float arrays should be in the range 0.0 to 1.0; MxN float arrays may be normalised.

cmap : `Colormap`, optional, default: None

If None, default to rc `image.cmap` value. `cmap` is ignored when `X` has RGB(A) information

aspect : ['auto' | 'equal' | scalar], optional, default: None

If 'auto', changes the image aspect ratio to match that of the axes.

If ‘equal’, and `extent` is None, changes the axes aspect ratio to match that of the image. If `extent` is not `None`, the axes aspect ratio is changed to match that of the extent.

If None, default to rc `image.aspect` value.

interpolation : string, optional, default: None
Acceptable values are ‘none’, ‘nearest’, ‘bilinear’, ‘bicubic’, ‘spline16’, ‘spline36’, ‘hanning’, ‘hamming’, ‘hermite’, ‘kaiser’, ‘quadric’, ‘catrom’, ‘gaussian’, ‘bessel’, ‘mitchell’, ‘sinc’, ‘lanczos’

If `interpolation` is None, default to rc `image.interpolation`. See also the `filternorm` and `filterrad` parameters. If `interpolation` is ‘none’, then no interpolation is performed on the Agg, ps and pdf backends. Other backends will fall back to ‘nearest’.

norm : *Normalize*, optional, default: None
A *Normalize* instance is used to scale luminance data to 0, 1. If `None`, use the default func:`normalize`. `norm` is only used if `X` is an array of floats.

vmin, vmax : scalar, optional, default: None
`vmin` and `vmax` are used in conjunction with norm to normalize luminance data. Note if you pass a `norm` instance, your settings for `vmin` and `vmax` will be ignored.

alpha : scalar, optional, default: None
The alpha blending value, between 0 (transparent) and 1 (opaque)

origin : [’upper’ | ‘lower’], optional, default: None
Place the [0,0] index of the array in the upper left or lower left corner of the axes. If None, default to rc `image.origin`.

extent : scalars (left, right, bottom, top), optional, default: None
The location, in data-coordinates, of the lower-left and upper-right corners. If `None`, the image is positioned such that the pixel centers fall on zero-based (row, column) indices.

shape : scalars (columns, rows), optional, default: None
For raw buffer images

filternorm : scalar, optional, default: 1
A parameter for the antigrain image resize filter. From the antigrain documentation, if `filternorm` = 1, the filter normalizes integer values and corrects the rounding errors. It doesn’t do anything with the source floating point values, it corrects only integers according to the rule of 1.0 which means that any sum of pixel weights must be equal to 1.0. So, the filter function must produce a graph of the proper shape.

filterrad : scalar, optional, default: 4.0
The filter radius for filters that have a radius parameter, i.e. when interpolation is one of: ‘sinc’, ‘lanczos’ or ‘blackman’

Returns image : *AxesImage*

Other Parameters kwargs : *Artist* properties.

See also:

matshow Plot a matrix or an array as an image.

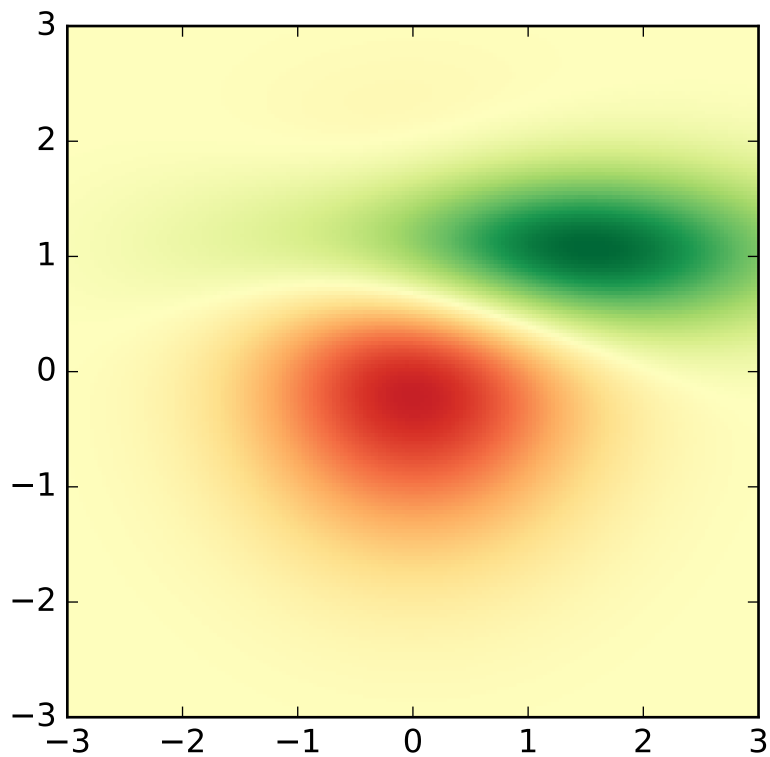
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

Additional kwargs: hold = [True|False] overrides default hold state

Examples



`matplotlib.pyplot.inferno()`
set the default colormap to inferno and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.install_repl_displayhook()`
Install a repl display hook so that any stale figure are automatically redrawn when control is returned to the repl.

This works with IPython terminals and kernels, as well as vanilla python shells.

`matplotlib.pyplot.ioff()`
Turn interactive mode off.

`matplotlib.pyplot.ion()`
Turn interactive mode on.

`matplotlib.pyplot.ishold()`
Return the hold status of the current axes.

`matplotlib.pyplot.isinteractive()`
Return status of interactive mode.

`matplotlib.pyplot.jet()`
set the default colormap to jet and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.legend(*args, **kwargs)`
Places a legend on the axes.

To make a legend for lines which already exist on the axes (via plot for instance), simply call this function with an iterable of strings, one for each legend item. For example:

```
ax.plot([1, 2, 3])
ax.legend(['A simple line'])
```

However, in order to keep the "label" and the legend element instance together, it is preferable to specify the label either at artist creation, or by calling the `set_label()` method on the artist:

```
line, = ax.plot([1, 2, 3], label='Inline label')
# Overwrite the label by calling the method.
line.set_label('Label via method')
ax.legend()
```

Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore. This is default for all artists, so calling `legend()` without any arguments and without setting the labels manually will result in no legend being drawn.

For full control of which artists have a legend entry, it is possible to pass an iterable of legend artists followed by an iterable of legend labels respectively:

```
legend((line1, line2, line3), ('label1', 'label2', 'label3'))
```

Parameters loc : int or string or pair of floats, default: 'upper right'

The location of the legend. Possible codes are:

Location String	Location Code
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

Alternatively can be a 2-tuple giving `x, y` of the lower-left corner of the legend in axes coordinates (in which case `bbox_to_anchor` will be ignored).

bbox_to_anchor : *matplotlib.transforms.BboxBase* instance or tuple of floats

Specify any arbitrary location for the legend in `bbox_transform` coordinates (default Axes coordinates).

For example, to put the legend's upper right hand corner in the center of the axes the following keywords can be used:

```
loc='upper right', bbox_to_anchor=(0.5, 0.5)
```

ncol : integer

The number of columns that the legend has. Default is 1.

prop : None or *matplotlib.font_manager.FontProperties* or dict

The font properties of the legend. If None (default), the current *matplotlib.rcParams* will be used.

fontsize : int or float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}

Controls the font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if `prop` is not specified.

numpoints : None or int

The number of marker points in the legend when creating a legend entry for a line/*matplotlib.lines.Line2D*. Default is `None` which will take the value from the `legend.numpoints` *rcParam*.

scatterpoints : None or int

The number of marker points in the legend when creating a legend entry for a scatter plot/ *matplotlib.collections.PathCollection*. Default is `None` which will take the value from the `legend.scatterpoints` *rcParam*.

scatteryoffsets : iterable of floats

The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to `[0.5]`. Default `[0.375, 0.5, 0.3125]`.

markerscale : None or int or float

The relative size of legend markers compared with the originally drawn ones. Default is `None` which will take the value from the `legend.markerscale` *rcParam*.

***markerfirst*: [*True* | *False*]**

if *True*, legend marker is placed to the left of the legend label if *False*, legend marker is placed to the right of the legend label

frameon : None or bool

Control whether a frame should be drawn around the legend. Default is `None` which will take the value from the `legend.frameon` *rcParam*.

fancybox : None or bool

Control whether round edges should be enabled around the *FancyBboxPatch* which makes up the legend's background. Default is None which will take the value from the legend.fancybox *rcParam*.

shadow : None or bool
Control whether to draw a shadow behind the legend. Default is None which will take the value from the legend.shadow *rcParam*.

framealpha : None or float
Control the alpha transparency of the legend's frame. Default is None which will take the value from the legend.framealpha *rcParam*.

mode : {"expand", None}
If mode is set to "expand" the legend will be horizontally expanded to fill the axes area (or bbox_to_anchor if defines the legend's size).

bbox_transform : None or *matplotlib.transforms.Transform*
The transform for the bounding box (bbox_to_anchor). For a value of None (default) the Axes' transAxes transform will be used.

title : str or None
The legend's title. Default is no title (None).

borderpad : float or None
The fractional whitespace inside the legend border. Measured in font-size units. Default is None which will take the value from the legend.borderpad *rcParam*.

labelspacing : float or None
The vertical space between the legend entries. Measured in font-size units. Default is None which will take the value from the legend.labelspacing *rcParam*.

handlelength : float or None
The length of the legend handles. Measured in font-size units. Default is None which will take the value from the legend.handlelength *rcParam*.

handletextpad : float or None
The pad between the legend handle and text. Measured in font-size units. Default is None which will take the value from the legend.handletextpad *rcParam*.

borderaxespad : float or None
The pad between the axes and legend border. Measured in font-size units. Default is None which will take the value from the legend.borderaxespad *rcParam*.

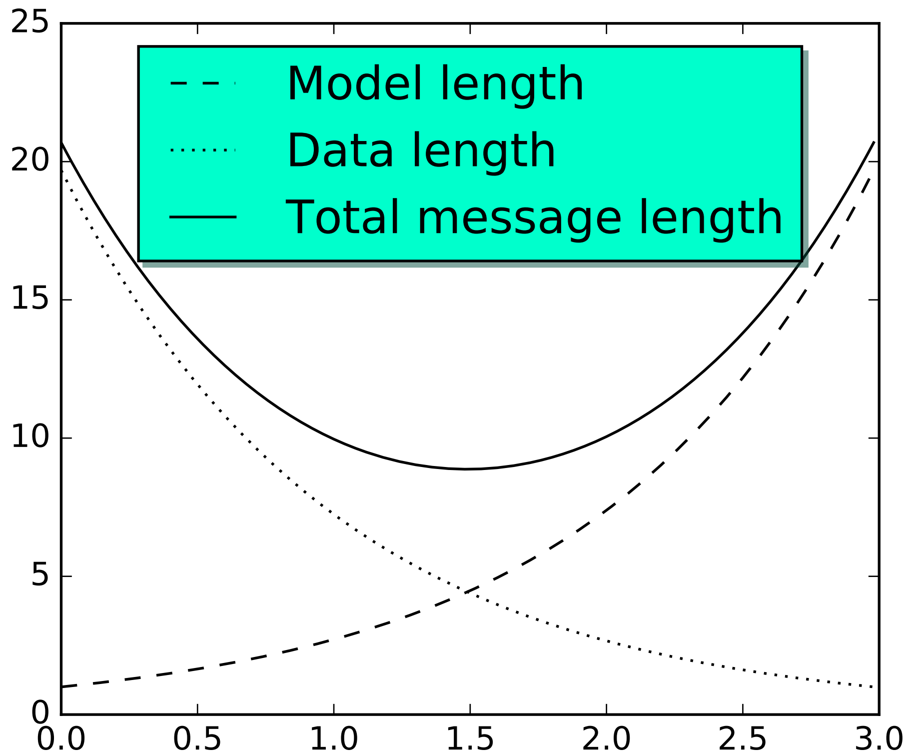
columnspacing : float or None
The spacing between columns. Measured in font-size units. Default is None which will take the value from the legend.columnspacing *rcParam*.

handler_map : dict or None
The custom dictionary mapping instances or types to a legend handler. This handler_map updates the default handler map found at *matplotlib.legend.Legend.get_legend_handler_map()*.

Notes

Not all kinds of artist are supported by the legend command. See *Legend guide* for details.

Examples



`matplotlib.pyplot.`**`locator_params`**(*axis='both'*, *tight=None*, ***kwargs*)

Control behavior of tick locators.

Keyword arguments:
axis ['x' | 'y' | 'both'] Axis on which to operate; default is 'both'.
tight [True | False | None] Parameter passed to `autoscale_view()`. Default is None, for no change.
Remaining keyword arguments are passed to directly to the *`set_params()`* method.

Typically one might want to reduce the maximum number of ticks and use tight bounds when plotting small subplots, for example:

```
ax.locator_params(tight=True, nbins=4)
```

Because the locator is involved in autoscaling, `autoscale_view()` is called automatically after the parameters are changed.

This presently works only for the *`MaxNLocator`* used by default on linear axes, but it may be generalized.

`matplotlib.pyplot.loglog(*args, **kwargs)`

Make a plot with log scaling on both the *x* and *y* axis.

Call signature:

```
loglog(*args, **kwargs)
```

`loglog()` supports all the keyword arguments of `plot()` and `matplotlib.axes.Axes.set_xscale()` / `matplotlib.axes.Axes.set_yscale()`.

Notable keyword arguments:

***basex/basey*: scalar > 1** Base of the *x*/*y* logarithm

***subsx/subsy*: [*None* | sequence]** The location of the minor *x*/*y* ticks; *None* defaults to autosubs, which depend on the number of decades in the plot; see `matplotlib.axes.Axes.set_xscale()` / `matplotlib.axes.Axes.set_yscale()` for details

***nonposx/nonposy*: ['mask' | 'clip']** Non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number

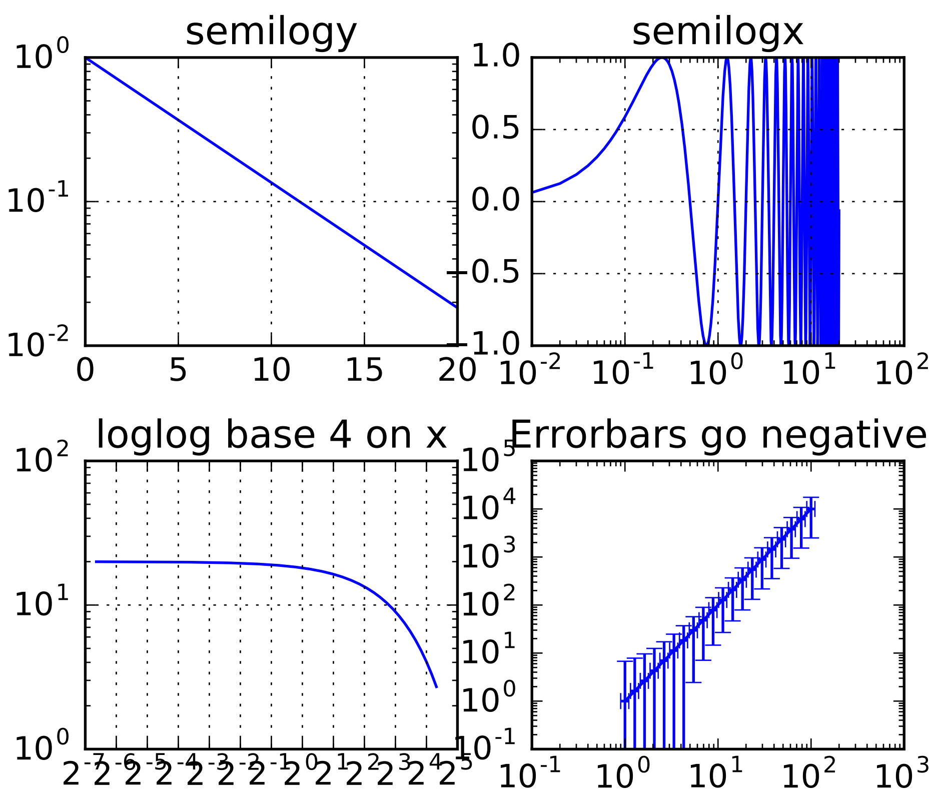
The remaining valid kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
	C

Table 68.18 – continued from previous page

Property	Description
`markersize` or ms	float
`markevery`	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
`path_effects`	unknown
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
`ydata`	1D array
`zorder`	any number

Example:



Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.magma()`

set the default colormap to magma and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.magnitude_spectrum(`*x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, scale=None, hold=None, data=None, **kwargs*`)`

Plot the magnitude spectrum.

Call signature:

```
magnitude_spectrum(x, Fs=2, Fc=0,  window=mlab.window_hanning,
                   pad_to=None, sides='default', **kwargs)
```

Compute the magnitude spectrum of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer**

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

***scale*: ['default' | 'linear' | 'dB']** The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale. When *mode* is 'density', this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). 'default' is 'linear'.

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns the tuple (*spectrum*, *freqs*, *line*):

***spectrum*: 1-D array** The values for the magnitude spectrum before scaling (real valued)

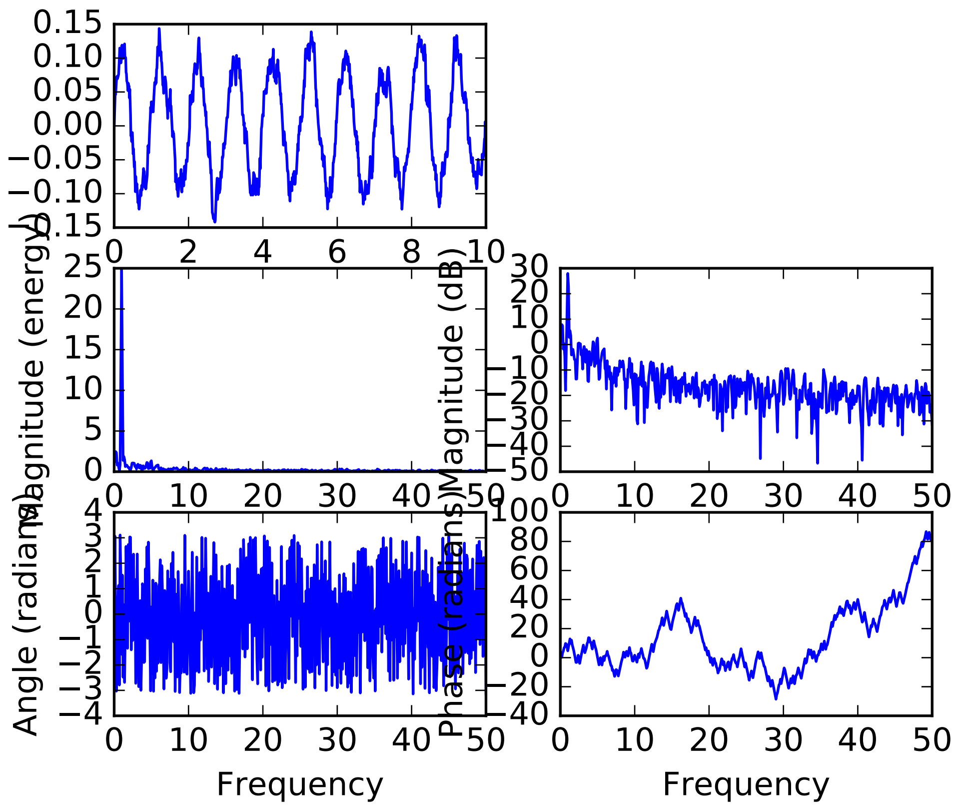
***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

***line*: a `Line2D` instance** The line created by this function

kwargs control the `Line2D` properties:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>agg_filter</td><td>unknown</td></tr>
<tr><td>alpha</td><td>float (0.0 transparent through 1.0 opaque)</td></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or aa</td><td>[True | False]</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color or c</td><td>any matplotlib color</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>dash_capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>dash_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>dashes</td><td>sequence of on/off ink in points</td></tr>
<tr><td>drawstyle</td><td>['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> | '</td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>marker</td><td>A valid marker style</td></tr>
<tr><td>markeredgecolor or mec</td><td>any matplotlib color</td></tr>
<tr><td>markeredgewidth or mew</td><td>float value in points</td></tr>
<tr><td>markerfacecolor or mfc</td><td>any matplotlib color</td></tr>
<tr><td>markerfacecoloralt or mfcalt</td><td>any matplotlib color</td></tr>
<tr><td>markersize or ms</td><td>float</td></tr>
<tr><td>markevery</td><td>[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>float distance in points or callable pick function <code>fn(artist, event)</code></td></tr>
<tr><td>pickradius</td><td>float distance in points</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>solid_capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>solid_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>transform</td><td>a matplotlib.transforms.Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>xdata</td><td>1D array</td></tr>
<tr><td>ydata</td><td>1D array</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

Example:



See also:

psd() *psd()* plots the power spectral density.'.

angle_spectrum() *angle_spectrum()* plots the angles of the corresponding frequencies.

phase_spectrum() *phase_spectrum()* plots the phase (unwrapped angle) of the corresponding frequencies.

specgram() *specgram()* can plot the magnitude spectrum of segments within the signal in a colormap.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`margins`**(**args*, ***kw*)

Set or retrieve autoscaling margins.

signatures:

```
margins()
```

returns xmargin, ymargin

```
margins(margin)

margins(xmargin, ymargin)

margins(x=xmargin, y=ymargin)

margins(..., tight=False)
```

All three forms above set the xmargin and ymargin parameters. All keyword parameters are optional. A single argument specifies both xmargin and ymargin. The *tight* parameter is passed to `autoscale_view()`, which is executed after a margin is changed; the default here is *True*, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting *tight* to *None* will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if *xmargin* is not None, then *xmargin* times the X data interval will be added to each end of that interval before it is used in autoscaling.

`matplotlib.pyplot.`**`matshow`**(*A*, *fignum=None*, ***kw*)
Display an array as a matrix in a new figure window.

The origin is set at the upper left hand corner and rows (first dimension of the array) are displayed horizontally. The aspect ratio of the figure window is that of the array, unless this would make an excessively short or narrow figure.

Tick labels for the xaxis are placed on top.

With the exception of *fignum*, keyword arguments are passed to `imshow()`. You may set the *origin* kwarg to "lower" if you want the first row in the array to be at the bottom instead of the top.

***fignum*: [None | integer | False]** By default, `matshow()` creates a new figure window with automatic numbering. If *fignum* is given as an integer, the created figure will use this figure number. Because of how `matshow()` tries to set the figure aspect ratio to be the one of the array, if you provide the number of an already existing figure, strange things may happen.

If *fignum* is *False* or 0, a new figure window will **NOT** be created.

`matplotlib.pyplot.`**`minorticks_off`**()
Remove minor ticks from the current plot.

`matplotlib.pyplot.`**`minorticks_on`**()
Display minor ticks on the current plot.

Displaying minor ticks reduces performance; turn them off using minorticks_off() if drawing speed is a problem.

`matplotlib.pyplot.`**`over`**(*func*, **args*, ***kwargs*)
Call a function with hold(True).

Calls:

```
func(*args, **kwargs)
```

with `hold(True)` and then restores the hold state.

`matplotlib.pyplot.pause(interval)`

Pause for *interval* seconds.

If there is an active figure it will be updated and displayed, and the GUI event loop will run during the pause.

If there is no active figure, or if a non-interactive backend is in use, this executes time.sleep(interval).

This can be used for crude animation. For more complex animation, see `matplotlib.animation`.

This function is experimental; its behavior may be changed or extended in a future release.

`matplotlib.pyplot.pcolor(*args, **kwargs)`

Create a pseudocolor plot of a 2-D array.

Note: pcolor can be very slow for large arrays; consider using the similar but much faster `pcolormesh()` instead.

Call signatures:

```
pcolor(C, **kwargs)
pcolor(X, Y, C, **kwargs)
```

C is the array of color values.

X and *Y*, if given, specify the (*x*, *y*) coordinates of the colored quadrilaterals; the quadrilateral for C[i,j] has corners at:

```
(X[i,   j],   Y[i,   j]),
(X[i,   j+1], Y[i,   j+1]),
(X[i+1, j],   Y[i+1, j]),
(X[i+1, j+1], Y[i+1, j+1]).
```

Ideally the dimensions of *X* and *Y* should be one greater than those of *C*; if the dimensions are the same, then the last row and column of *C* will be ignored.

Note that the column index corresponds to the *x*-coordinate, and the row index corresponds to *y*; for details, see the *Grid Orientation* section below.

If either or both of *X* and *Y* are 1-D arrays or column vectors, they will be expanded as needed into the appropriate 2-D arrays, making a rectangular grid.

X, *Y* and *C* may be masked arrays. If either C[i, j], or one of the vertices surrounding C[i,j] (*X* or *Y* at [i, j], [i+1, j], [i, j+1],[i+1, j+1]) is masked, nothing is plotted.

Keyword arguments:

***cmap*: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance. If *None*, use rc settings.

***norm*: [*None* | Normalize]** An `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If *None*, defaults to `normalize()`.

***vmin/vmax*: [*None* | scalar]** *vmin* and *vmax* are used in conjunction with *norm* to normalize luminance data. If either is *None*, it is autoscaled to the respective min or max of the color array *C*. If not *None*, *vmin* or *vmax* passed in here override any pre-existing values supplied in the *norm* instance.

***shading*: ['flat' | 'faceted']** If 'faceted', a black grid is drawn around each rectangle; if 'flat', edges are not drawn. Default is 'flat', contrary to MATLAB.

This kwarg is deprecated; please use 'edgecolors' instead:

- shading='flat' – edgecolors='none'
- shading='faceted – edgecolors='k'

***edgecolors*: [*None* | `'none'` | color | color sequence]** If *None*, the rc setting is used by default.

If `'none'`, edges will not be visible.

An mpl color or sequence of colors will set the edge color

alpha*: `0 <= scalar <= 1` or *None the alpha blending value

***snap*: bool** Whether to snap the mesh to pixel boundaries.

Return value is a `matplotlib.collections.Collection` instance. The grid orientation follows the MATLAB convention: an array *C* with shape (*nrows*, *ncolumns*) is plotted with the column number as *X* and the row number as *Y*, increasing up; hence it is plotted the way the array would be printed, except that the *Y* axis is reversed. That is, *C* is taken as *C*(*y, x*).

Similarly for `meshgrid()`:

```
x = np.arange(5)
y = np.arange(3)
X, Y = np.meshgrid(x, y)
```

is equivalent to:

```
X = array([[0, 1, 2, 3, 4],
           [0, 1, 2, 3, 4],
           [0, 1, 2, 3, 4]])

Y = array([[0, 0, 0, 0, 0],
           [1, 1, 1, 1, 1],
           [2, 2, 2, 2, 2]])
```

so if you have:

```
C = rand(len(x), len(y))
```

then you need to transpose C:

```
pcolor(X, Y, C.T)
```

or:

```
pcolor(C.T)
```

MATLAB `pcolor()` always discards the last row and column of *C*, but matplotlib displays the last row and column if *X* and *Y* are not specified, or if *X* and *Y* have one more row and column than *C*.

kwargs can be used to control the `PolyCollection` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float or None
`animated`	[True | False]
`antialiased` or antialiaseds	Boolean or sequence of booleans
`array`	unknown
`axes`	an `Axes` instance
`clim`	a length 2 sequence of floats
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`cmap`	a colormap or registered colormap name
`color`	matplotlib color arg or sequence of rgba tuples
`contains`	a callable function
`edgecolor` or edgecolors	matplotlib color spec or sequence of specs
`facecolor` or facecolors	matplotlib color spec or sequence of specs
`figure`	a `matplotlib.figure.Figure` instance
`gid`	an id string
`hatch`	[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]
`label`	string or anything printable with ‘%s’ conversion.
`linestyle` or linestyles or dashes	[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` |
`linewidth` or lw or linewidths	float or sequence of floats
`norm`	unknown
`offset_position`	unknown
`offsets`	float or sequence of floats
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`pickradius`	unknown
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`urls`	unknown
`visible`	[True | False]
`zorder`	any number

Note: The default *antialiaseds* is False if the default *edgecolors*=”none” is used. This eliminates artificial lines at patch boundaries, and works regardless of the value of alpha. If *edgecolors* is not “none”, then the default *antialiaseds* is taken from rcParams[’patch.antialiased’], which defaults to *True*. Stroking the edges may be preferred if *alpha* is 1, but will cause artifacts otherwise.

See also:

`pcolormesh()` For an explanation of the differences between pcolor and pcolormesh.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`pcolormesh`**(**args*, ***kwargs*)

Plot a quadrilateral mesh.

Call signatures:

```
pcolormesh(C)
pcolormesh(X, Y, C)
pcolormesh(C, **kwargs)
```

Create a pseudocolor plot of a 2-D array.

pcolormesh is similar to `pcolor()`, but uses a different mechanism and returns a different object; pcolor returns a `PolyCollection` but pcolormesh returns a `QuadMesh`. It is much faster, so it is almost always preferred for large arrays.

C may be a masked array, but *X* and *Y* may not. Masked array support is implemented via *cmap* and *norm*; in contrast, `pcolor()` simply does not draw quadrilaterals with masked colors or vertices.

Keyword arguments:

***cmap*: [*None* | Colormap]** A `matplotlib.colors.Colormap` instance. If *None*, use rc settings.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance is used to scale luminance data to 0,1. If *None*, defaults to `normalize()`.

***vmin/vmax*: [*None* | scalar]** *vmin* and *vmax* are used in conjunction with *norm* to normalize luminance data. If either is *None*, it is autoscaled to the respective min or max of the color array *C*. If not *None*, *vmin* or *vmax* passed in here override any pre-existing values supplied in the *norm* instance.

***shading*: ['flat' | 'gouraud']** 'flat' indicates a solid color for each quad. When 'gouraud', each quad will be Gouraud shaded. When gouraud shading, edgecolors is ignored.

***edgecolors*: [*None* | `'None'` | `'face'` | color | color sequence]**

If *None*, the rc setting is used by default.

If `'None'`, edges will not be visible.

If `'face'`, edges will have the same color as the faces.

An mpl color or sequence of colors will set the edge color

alpha*: `0 <= scalar <= 1` or *None the alpha blending value

Return value is a `matplotlib.collections.QuadMesh` object.

kwargs can be used to control the `matplotlib.collections.QuadMesh` properties:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td><code>agg_filter</code></td><td>unknown</td></tr>
<tr><td><code>alpha</code></td><td>float or None</td></tr>
<tr><td><code>animated</code></td><td>[True | False]</td></tr>
<tr><td><code>antialiased</code> or antialiaseds</td><td>Boolean or sequence of booleans</td></tr>
<tr><td><code>array</code></td><td>unknown</td></tr>
<tr><td><code>axes</code></td><td>an <code>Axes</code> instance</td></tr>
<tr><td><code>clim</code></td><td>a length 2 sequence of floats</td></tr>
<tr><td><code>clip_box</code></td><td>a <code>matplotlib.transforms.Bbox</code> instance</td></tr>
<tr><td><code>clip_on</code></td><td>[True | False]</td></tr>
<tr><td><code>clip_path</code></td><td>[(<code>Path</code>, <code>Transform</code>) | <code>Patch</code> | None]</td></tr>
<tr><td><code>cmap</code></td><td>a colormap or registered colormap name</td></tr>
<tr><td><code>color</code></td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td><code>contains</code></td><td>a callable function</td></tr>
<tr><td><code>edgecolor</code> or edgecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>facecolor</code> or facecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>figure</code></td><td>a <code>matplotlib.figure.Figure</code> instance</td></tr>
<tr><td><code>gid</code></td><td>an id string</td></tr>
<tr><td><code>hatch</code></td><td>[‘/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]</td></tr>
<tr><td><code>label</code></td><td>string or anything printable with ‘%s’ conversion.</td></tr>
<tr><td><code>linestyle</code> or linestyles or dashes</td><td>[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> |</td></tr>
<tr><td><code>linewidth</code> or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td><code>norm</code></td><td>unknown</td></tr>
<tr><td><code>offset_position</code></td><td>unknown</td></tr>
<tr><td><code>offsets</code></td><td>float or sequence of floats</td></tr>
<tr><td><code>path_effects</code></td><td>unknown</td></tr>
<tr><td><code>picker</code></td><td>[None|float|boolean|callable]</td></tr>
<tr><td><code>pickradius</code></td><td>unknown</td></tr>
<tr><td><code>rasterized</code></td><td>[True | False | None]</td></tr>
<tr><td><code>sketch_params</code></td><td>unknown</td></tr>
<tr><td><code>snap</code></td><td>unknown</td></tr>
<tr><td><code>transform</code></td><td><code>Transform</code> instance</td></tr>
<tr><td><code>url</code></td><td>a url string</td></tr>
<tr><td><code>urls</code></td><td>unknown</td></tr>
<tr><td><code>visible</code></td><td>[True | False]</td></tr>
<tr><td><code>zorder</code></td><td>any number</td></tr>
</table>

See also:

`pcolor()` For an explanation of the grid orientation and the expansion of 1-D *X* and/or *Y* to 2-D arrays.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.phase_spectrum`(*x*, *Fs=None*, *Fc=None*, *window=None*, *pad_to=None*, *sides=None*, *hold=None*, *data=None*, ***kwargs*)

Plot the phase spectrum.

Call signature:

```
phase_spectrum(x, Fs=2, Fc=0,  window=mlab.window_hanning,
               pad_to=None, sides='default', **kwargs)
```

Compute the phase spectrum (unwrapped angle spectrum) of *x*. Data is padded to a length of *pad_to* and the windowing function *window* is applied to the signal.

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer**

The number of points to which the data segment is padded when performing the FFT. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to the length of the input signal (i.e. no padding).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

Returns the tuple (*spectrum*, *freqs*, *line*):

***spectrum*: 1-D array** The values for the phase spectrum in radians (real valued)

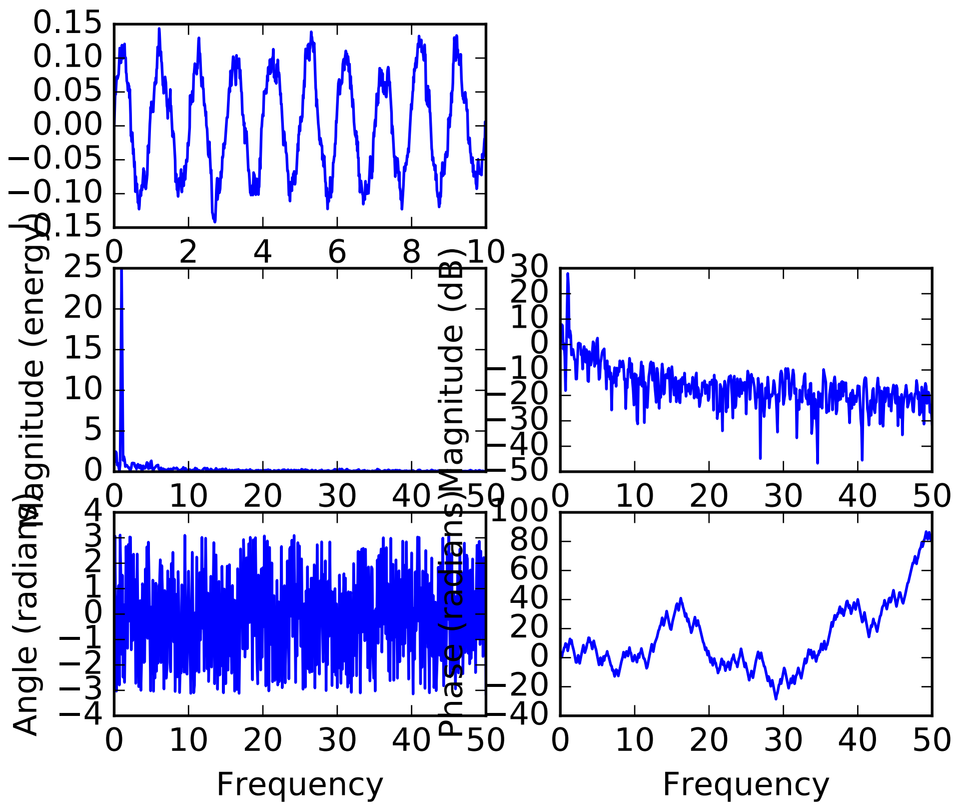
***freqs*: 1-D array** The frequencies corresponding to the elements in *spectrum*

***line*: a `Line2D` instance** The line created by this function

kwargs control the `Line2D` properties:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>agg_filter</td><td>unknown</td></tr>
<tr><td>alpha</td><td>float (0.0 transparent through 1.0 opaque)</td></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or aa</td><td>[True | False]</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color or c</td><td>any matplotlib color</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>dash_capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>dash_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>dashes</td><td>sequence of on/off ink in points</td></tr>
<tr><td>drawstyle</td><td>['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fillstyle</td><td>['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>label</td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td>linestyle or ls</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '</td></tr>
<tr><td>linewidth or lw</td><td>float value in points</td></tr>
<tr><td>marker</td><td>A valid marker style</td></tr>
<tr><td>markeredgecolor or mec</td><td>any matplotlib color</td></tr>
<tr><td>markeredgewidth or mew</td><td>float value in points</td></tr>
<tr><td>markerfacecolor or mfc</td><td>any matplotlib color</td></tr>
<tr><td>markerfacecoloralt or mfcalt</td><td>any matplotlib color</td></tr>
<tr><td>markersize or ms</td><td>float</td></tr>
<tr><td>markevery</td><td>[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>float distance in points or callable pick function fn(artist, event)</td></tr>
<tr><td>pickradius</td><td>float distance in points</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>solid_capstyle</td><td>['butt' | 'round' | 'projecting']</td></tr>
<tr><td>solid_joinstyle</td><td>['miter' | 'round' | 'bevel']</td></tr>
<tr><td>transform</td><td>a matplotlib.transforms.Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>xdata</td><td>1D array</td></tr>
<tr><td>ydata</td><td>1D array</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

Example:



See also:

[`magnitude_spectrum()`] *magnitude_spectrum()* plots the magnitudes of the corresponding frequencies.

[`angle_spectrum()`] *angle_spectrum()* plots the wrapped version of this function.

[`specgram()`] *specgram()* can plot the phase spectrum of segments within the signal in a colormap.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`pie`**(*x*, *explode=None*, *labels=None*, *colors=None*, *autopct=None*, *pctdistance=0.6*, *shadow=False*, *labeldistance=1.1*, *startangle=None*, *radius=None*, *counterclock=True*, *wedgeprops=None*, *textprops=None*, *center=(0, 0)*, *frame=False*, *hold=None*, *data=None*)

Plot a pie chart.

Call signature:

```
pie(x, explode=None, labels=None,
    colors=('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'),
```

```
    autopct=None, pctdistance=0.6, shadow=False,
    labeldistance=1.1, startangle=None, radius=None,
    counterclock=True, wedgeprops=None, textprops=None,
    center = (0, 0), frame = False )
```

Make a pie chart of array *x*. The fractional area of each wedge is given by x/sum(x). If sum(x) <= 1, then the values of x give the fractional area directly and the array will not be normalized. The wedges are plotted counterclockwise, by default starting from the x-axis.

Keyword arguments:

***explode*: [*None* | len(x) sequence]** If not *None*, is a `len(x)` array which specifies the fraction of the radius with which to offset each wedge.

***colors*: [*None* | color sequence]** A sequence of matplotlib color args through which the pie chart will cycle.

***labels*: [*None* | len(x) sequence of strings]** A sequence of strings providing the labels for each wedge

***autopct*: [*None* | format string | format function]** If not *None*, is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be `fmt%pct`. If it is a function, it will be called.

***pctdistance*: scalar** The ratio between the center of each pie slice and the start of the text generated by *autopct*. Ignored if *autopct* is *None*; default is 0.6.

***labeldistance*: scalar** The radial distance at which the pie labels are drawn

***shadow*: [*False* | *True*]** Draw a shadow beneath the pie.

***startangle*: [*None* | Offset angle]** If not *None*, rotates the start of the pie chart by *angle* degrees counterclockwise from the x-axis.

radius: [*None* | scalar] The radius of the pie, if *radius* is *None* it will be set to 1.

***counterclock*: [*False* | *True*]** Specify fractions direction, clockwise or counterclockwise.

***wedgeprops*: [*None* | dict of key value pairs]** Dict of arguments passed to the wedge objects making the pie. For example, you can pass in wedgeprops = { 'linewidth' : 3 } to set the width of the wedge border lines equal to 3. For more details, look at the doc/arguments of the wedge object. By default `clip_on=False`.

***textprops*: [*None* | dict of key value pairs]** Dict of arguments to pass to the text objects.

center: [(0,0) | sequence of 2 scalars] Center position of the chart.

***frame*: [*False* | *True*]** Plot axes frame with the chart.

The pie chart will probably look best if the figure and axes are square, or the Axes aspect is equal. e.g.:

```
figure(figsize=(8,8))
ax = axes([0.1, 0.1, 0.8, 0.8])
```

or:

```
axes(aspect=1)
```

Return value: If *autopct* is *None*, return the tuple (*patches*, *texts*):

- *patches* is a sequence of `matplotlib.patches.Wedge` instances
- *texts* is a list of the label `matplotlib.text.Text` instances.

If *autopct* is not *None*, return the tuple (*patches*, *texts*, *autotexts*), where *patches* and *texts* are as above, and *autotexts* is a list of `Text` instances for the numeric labels.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'colors', 'labels', 'x', 'explode'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.pink()`

set the default colormap to pink and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.plasma()`

set the default colormap to plasma and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.plot(*args, **kwargs)`

Plot lines and/or markers to the `Axes`. *args* is a variable length argument, allowing for multiple *x*, *y* pairs with an optional format string. For example, each of the following is legal:

```
plot(x, y)        # plot x and y using default line style and color
plot(x, y, 'bo')  # plot x and y using blue circle markers
plot(y)           # plot y using x as index array 0..N-1
plot(y, 'r+')     # ditto, but with red plusses
```

If *x* and/or *y* is 2-dimensional, then the corresponding columns will be plotted.

If used with labeled data, make sure that the color spec is not included as an element in data, as otherwise the last case `plot("v","r", data={"v":..., "r":...)` can be interpreted as the first case which would do `plot(v, r)` using the default line style and color.

If not used with labeled data (i.e., without a data argument), an arbitrary number of *x*, *y*, *fmt* groups can be specified, as in:

```
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

Return value is a list of lines that were added.

By default, each line is assigned a different style specified by a 'style cycle'. To change this behavior, you can edit the axes.prop_cycle rcParam.

The following format string characters are accepted to control the line style or marker:

character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
'1'	tri_down marker
'2'	tri_up marker
'3'	tri_left marker
'4'	tri_right marker
's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

The following color abbreviations are supported:

character	color
‘b’	blue
‘g’	green
‘r’	red
‘c’	cyan
‘m’	magenta
‘y’	yellow
‘k’	black
‘w’	white

In addition, you can specify colors in many weird and wonderful ways, including full names (`'green'`), hex strings (`'#008000'`), RGB or RGBA tuples (`(0,1,0,1)`) or grayscale intensities as a string (`'0.8'`). Of these, the string specifications can be used in place of a `fmt` group, but the tuple forms can be used only as `kwargs`.

Line styles and colors are combined in a single format string, as in `'bo'` for blue circles.

The *kwargs* can be used to set line properties (any property that has a `set_*` method). You can use this to set a line label (for auto legends), linewidth, anitialising, marker face color, etc. Here is an example:

```
plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plot([1,2,3], [1,4,9], 'rs',  label='line 2')
axis([0, 4, 0, 10])
legend()
```

If you make multiple lines with one plot command, the kwargs apply to all those lines, e.g.:

```
plot(x1, y1, x2, y2, antialiased=False)
```

Neither line will be antialiased.

You do not need to use format strings, which are just abbreviations. All of the line properties can be controlled by keyword arguments. For example, you can set the color, marker, linestyle, and markercolor with:

```
plot(x, y, color='green', linestyle='dashed', marker='o',
     markerfacecolor='blue', markersize=12).
```

See *Line2D* for details.

The kwargs are *Line2D* properties:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an *Axes* instance
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a *matplotlib.figure.Figure* instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
linewidth or lw	float value in points
marker	*A valid marker style*
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color

C

Table 68.23 – continued from previous page

Property	Description
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
`path_effects`	unknown
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
`ydata`	1D array
`zorder`	any number

kwargs *scalex* and *scaley*, if defined, are passed on to `autoscale_view()` to determine whether the *x* and *y* axes are autoscaled; the default is *True*.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**plot_date**(*x*, *y*, *fmt='o'*, *tz=None*, *xdate=True*, *ydate=False*, *hold=None*, *data=None*, ***kwargs*)

Plot with data with dates.

Call signature:

```
plot_date(x, y, fmt='bo', tz=None, xdate=True,
          ydate=False, **kwargs)
```

Similar to the `plot()` command, except the *x* or *y* (or both) data is considered to be dates, and the axis is labeled accordingly.

x and/or *y* can be a sequence of dates represented as float days since 0001-01-01 UTC.

Keyword arguments:

***fmt*: string** The plot format string.

***tz*: [*None* | timezone string | `tzinfo` instance]** The time zone to use in labeling dates. If *None*, defaults to rc value.

> ***xdate*: [*True* | *False*]** If *True*, the *x*-axis will be labeled with dates.
>
> ***ydate*: [*False* | *True*]** If *True*, the *y*-axis will be labeled with dates.

Note if you are using custom date tickers and formatters, it may be necessary to set the formatters/locators after the call to `plot_date()` since `plot_date()` will set the default tick locator to `matplotlib.dates.AutoDateLocator` (if the tick locator is not already set to a `matplotlib.dates.DateLocator` instance) and the default tick formatter to `matplotlib.dates.AutoDateFormatter` (if the tick formatter is not already set to a `matplotlib.dates.DateFormatter` instance).

Valid kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float
`markevery`	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
`path_effects`	unknown
`picker`	float distance in points or callable pick function `fn(artist, event)`
`pickradius`	float distance in points
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`solid_capstyle`	['butt' | 'round' | 'projecting']
`solid_joinstyle`	['miter' | 'round' | 'bevel']

C

Table 68.24 – continued from previous page

Property	Description
`transform`	a `matplotlib.transforms.Transform` instance
`url`	a url string
`visible`	[True | False]
`xdata`	1D array
`ydata`	1D array
`zorder`	any number

See also:

`dates` for helper functions

`date2num()`, `num2date()` and `drange()` for help on creating the required floating point dates.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.plotfile`(*fname*, *cols=(0,)*, *plotfuncs=None*, *comments='#'*, *skiprows=0*, *checkrows=5*, *delimiter=', '*, *names=None*, *subplots=True*, *newfig=True*, ***kwargs*)

Plot the data in in a file.

cols is a sequence of column identifiers to plot. An identifier is either an int or a string. If it is an int, it indicates the column number. If it is a string, it indicates the column header. matplotlib will make column headers lower case, replace spaces with underscores, and remove all illegal characters; so `'Adj Close*'` will have name `'adj_close'`.

- If len(*cols*) == 1, only that column will be plotted on the *y* axis.
- If len(*cols*) > 1, the first element will be an identifier for data for the *x* axis and the remaining elements will be the column indexes for multiple subplots if *subplots* is *True* (the default), or for lines in a single subplot if *subplots* is *False*.

plotfuncs, if not *None*, is a dictionary mapping identifier to an `Axes` plotting function as a string. Default is 'plot', other choices are 'semilogy', 'fill', 'bar', etc. You must use the same type of identifier in the *cols* vector as you use in the *plotfuncs* dictionary, e.g., integer column numbers in both or column names in both. If *subplots* is *False*, then including any function such as 'semilogy' that changes the axis scaling will set the scaling for all columns.

comments, *skiprows*, *checkrows*, *delimiter*, and *names* are all passed on to `matplotlib.pylab.csv2rec()` to load the data into a record array.

If *newfig* is *True*, the plot always will be made in a new figure; if *False*, it will be made in the current figure if one exists, else in a new figure.

kwargs are passed on to plotting functions.

Example usage:

```
# plot the 2nd and 4th column against the 1st in two subplots
plotfile(fname, (0,1,3))

# plot using column names; specify an alternate plot type for volume
plotfile(fname, ('date', 'volume', 'adj_close'),
                              plotfuncs={'volume': 'semilogy'})
```

Note: plotfile is intended as a convenience for quickly plotting data from flat files; it is not intended as an alternative interface to general plotting with pyplot or matplotlib.

`matplotlib.pyplot.polar`(**args*, ***kwargs*)

Make a polar plot.

call signature:

```
polar(theta, r, **kwargs)
```

Multiple *theta*, *r* arguments are supported, with format strings, as in `plot()`.

`matplotlib.pyplot.prism`()

set the default colormap to prism and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.psd`(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *return_line=None*, *hold=None*, *data=None*, ***kwargs*)

Plot the power spectral density.

Call signature:

```
psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=mlab.window_hanning, noverlap=0, pad_to=None,
    sides='default', scale_by_freq=None, return_line=None, **kwargs)
```

The power spectral density P_{xx} by Welch's average periodogram method. The vector x is divided into *NFFT* length segments. Each segment is detrended by function *detrend* and windowed by function *window*. *noverlap* gives the length of the overlap between segments. The $|\mathrm{fft}(i)|^2$ of each segment i are averaged to compute P_{xx}, with a scaling to correct for power loss due to windowing.

If len(x) < *NFFT*, it will be zero padded to *NFFT*.

x: 1-D array or sequence Array or sequence containing the data

Keyword arguments:

Fs: scalar The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

window: callable or ndarray A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or** callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***noverlap*: integer** The number of points of overlap between segments. The default value is 0 (no overlap).

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

***return_line*: bool** Whether to include the line object plotted in the returned values. Default is False.

If *return_line* is False, returns the tuple (*Pxx*, *freqs*). If *return_line* is True, returns the tuple (*Pxx*, *freqs*. *line*):

***Pxx*: 1-D array** The values for the power spectrum `P_{xx}` before scaling (real valued)

***freqs*: 1-D array** The frequencies corresponding to the elements in *Pxx*

***line*: a `Line2D` instance** The line created by this function. Only returend if *return_line* is True.

For plotting, the power is plotted as $10\log_{10}(P_{xx})$ for decibels, though *Pxx* itself is returned.

References: Bendat & Piersol – Random Data: Analysis and Measurement Procedures, John Wiley & Sons (1986)

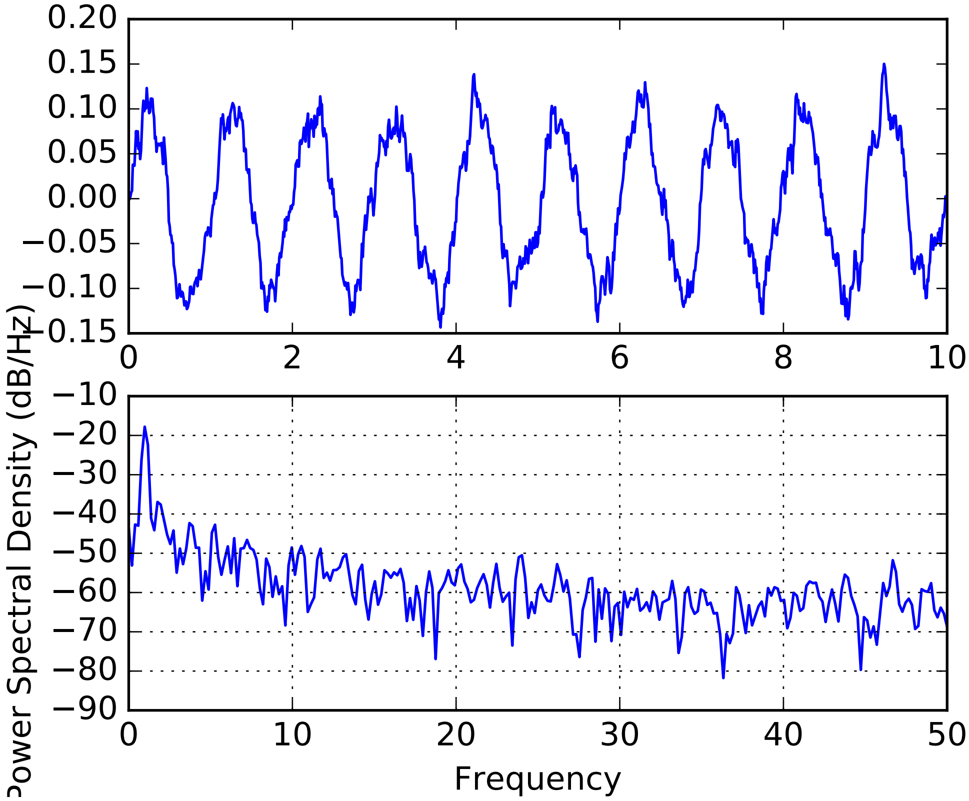
kwargs control the `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
	C

Table 68.25 – continued from previous page

Property	Description
antialiased or aa	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '
linewidth or lw	float value in points
marker	A valid marker style
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

Example:



See also:

specgram() *specgram()* differs in the default overlap; in not returning the mean of the segment periodograms; in returning the times of the segments; and in plotting a colormap instead of a line.

magnitude_spectrum() *magnitude_spectrum()* plots the magnitude spectrum.

csd() *csd()* plots the spectral density between two signals.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`quiver`**`(*args, **kw)`

Plot a 2-D field of arrows.

call signatures:

```
quiver(U, V, **kw)
quiver(U, V, C, **kw)
quiver(X, Y, U, V, **kw)
quiver(X, Y, U, V, C, **kw)
```

Arguments:

***X*, *Y*:** The x and y coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg)

***U*, *V*:** Give the x and y components of the arrow vectors

***C*:** An optional array used to map colors to the arrows

All arguments may be 1-D or 2-D arrays or sequences. If *X* and *Y* are absent, they will be generated as a uniform grid. If *U* and *V* are 2-D arrays but *X* and *Y* are 1-D, and if `len(X)` and `len(Y)` match the column and row dimensions of *U*, then *X* and *Y* will be expanded with `numpy.meshgrid()`.

U, *V*, *C* may be masked arrays, but masked *X*, *Y* are not supported at present.

Keyword arguments:

***units*: ['width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'xy']** Arrow units; the arrow dimensions *except for length* are in multiples of this unit.

- 'width' or 'height': the width or height of the axes
- 'dots' or 'inches': pixels or inches, based on the figure dpi
- 'x', 'y', or 'xy': *X*, *Y*, or sqrt(X^2+Y^2) data units

The arrows scale differently depending on the units. For 'x' or 'y', the arrows get larger as one zooms in; for other units, the arrow size is independent of the zoom state. For 'width or 'height', the arrow size increases with the width and height of the axes, respectively, when the window is resized; for 'dots' or 'inches', resizing does not change the arrows.

***angles*: ['uv' | 'xy' | array]** With the default 'uv', the arrow axis aspect ratio is 1, so that if *U*==*V* the orientation of the arrow on the plot is 45 degrees CCW from the horizontal axis (positive to the right). With 'xy', the arrow points from (x,y) to (x+u, y+v). Use this for plotting a gradient field, for example. Alternatively, arbitrary angles may be specified as an array of values in degrees, CCW from the horizontal axis. Note: inverting a data axis will correspondingly invert the arrows *only* with `angles='xy'`.

***scale*: [*None* | float]** Data units per arrow length unit, e.g., m/s per plot width; a smaller scale parameter makes the arrow longer. If *None*, a simple autoscaling algorithm is used, based on the average vector length and the number of vectors. The arrow length unit is given by the *scale_units* parameter

***scale_units*: *None*, or any of the *units* options.** For example, if *scale_units* is 'inches', *scale* is 2.0, and `(u,v) = (1,0)`, then the vector will be 0.5 inches long. If *scale_units* is 'width', then the vector will be half the width of the axes.

If *scale_units* is 'x' then the vector will be 0.5 x-axis units. To plot vectors in the x-y plane, with u and v having the same units as x and y, use "angles='xy', scale_units='xy', scale=1".

***width*:** Shaft width in arrow units; default depends on choice of units, above, and number of vectors; a typical starting value is about 0.005 times the width of the plot.

***headwidth*: scalar** Head width as multiple of shaft width, default is 3

***headlength*: scalar** Head length as multiple of shaft width, default is 5

***headaxislength*: scalar** Head length at shaft intersection, default is 4.5

***minshaft*: scalar** Length below which arrow scales, in units of head length. Do not set this to less than 1, or small arrows will look terrible! Default is 1

***minlength*: scalar** Minimum length as a multiple of shaft width; if an arrow length is less than this, plot a dot (hexagon) of this diameter instead. Default is 1.

***pivot*: ['tail' | 'mid' | 'middle' | 'tip']** The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*.

***color*: [color | color sequence]** This is a synonym for the `PolyCollection` facecolor kwarg. If *C* has been set, *color* has no effect.

The defaults give a slightly swept-back arrow; to make the head a triangle, make *headaxislength* the same as *headlength*. To make the arrow more pointed, reduce *headwidth* or increase *headlength* and *headaxislength*. To make the head smaller relative to the shaft, scale down all the head parameters. You will probably do best to leave minshaft alone.

linewidths and edgecolors can be used to customize the arrow outlines. Additional `PolyCollection` keyword arguments:

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td><code>agg_filter</code></td><td>unknown</td></tr>
<tr><td><code>alpha</code></td><td>float or None</td></tr>
<tr><td><code>animated</code></td><td>[True | False]</td></tr>
<tr><td><code>antialiased</code> or antialiaseds</td><td>Boolean or sequence of booleans</td></tr>
<tr><td><code>array</code></td><td>unknown</td></tr>
<tr><td><code>axes</code></td><td>an <code>Axes</code> instance</td></tr>
<tr><td><code>clim</code></td><td>a length 2 sequence of floats</td></tr>
<tr><td><code>clip_box</code></td><td>a <code>matplotlib.transforms.Bbox</code> instance</td></tr>
<tr><td><code>clip_on</code></td><td>[True | False]</td></tr>
<tr><td><code>clip_path</code></td><td>[(<code>Path</code>, <code>Transform</code>) | <code>Patch</code> | None]</td></tr>
<tr><td><code>cmap</code></td><td>a colormap or registered colormap name</td></tr>
<tr><td><code>color</code></td><td>matplotlib color arg or sequence of rgba tuples</td></tr>
<tr><td><code>contains</code></td><td>a callable function</td></tr>
<tr><td><code>edgecolor</code> or edgecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>facecolor</code> or facecolors</td><td>matplotlib color spec or sequence of specs</td></tr>
<tr><td><code>figure</code></td><td>a <code>matplotlib.figure.Figure</code> instance</td></tr>
<tr><td><code>gid</code></td><td>an id string</td></tr>
<tr><td><code>hatch</code></td><td>['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']</td></tr>
<tr><td><code>label</code></td><td>string or anything printable with '%s' conversion.</td></tr>
<tr><td><code>linestyle</code> or linestyles or dashes</td><td>['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | <code>'-'</code> | <code>'--'</code> | <code>'-.'</code> |</td></tr>
<tr><td><code>linewidth</code> or lw or linewidths</td><td>float or sequence of floats</td></tr>
<tr><td><code>norm</code></td><td>unknown</td></tr>
<tr><td><code>offset_position</code></td><td>unknown</td></tr>
<tr><td><code>offsets</code></td><td>float or sequence of floats</td></tr>
<tr><td><code>path_effects</code></td><td>unknown</td></tr>
<tr><td><code>picker</code></td><td>[None|float|boolean|callable]</td></tr>
<tr><td><code>pickradius</code></td><td>unknown</td></tr>
<tr><td><code>rasterized</code></td><td>[True | False | None]</td></tr>
<tr><td><code>sketch_params</code></td><td>unknown</td></tr>
<tr><td><code>snap</code></td><td>unknown</td></tr>
<tr><td><code>transform</code></td><td><code>Transform</code> instance</td></tr>
<tr><td><code>url</code></td><td>a url string</td></tr>
<tr><td><code>urls</code></td><td>unknown</td></tr>
<tr><td><code>visible</code></td><td>[True | False]</td></tr>
</table>

Table 68.26 – continued from previous page

Property	Description
zorder	any number

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.quiverkey(*args, **kw)`

Add a key to a quiver plot.

Call signature:

```
quiverkey(Q, X, Y, U, label, **kw)
```

Arguments:

***Q*:** The Quiver instance returned by a call to quiver.

***X, Y*:** The location of the key; additional explanation follows.

***U*:** The length of the key

***label*:** A string with the length and units of the key

Keyword arguments:

***coordinates* = ['axes' | 'figure' | 'data' | 'inches']** Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are normalized coordinate systems with 0,0 in the lower left and 1,1 in the upper right; 'data' are the axes data coordinates (used for the locations of the vectors in the quiver plot itself); 'inches' is position in the figure in inches, with 0,0 at the lower left corner.

***color*:** overrides face and edge colors from *Q*.

***labelpos* = ['N' | 'S' | 'E' | 'W']** Position the label above, below, to the right, to the left of the arrow, respectively.

***labelsep*:** Distance in inches between the arrow and the label. Default is 0.1

***labelcolor*:** defaults to default `Text` color.

***fontproperties*:** A dictionary with keyword arguments accepted by the `FontProperties` initializer: *family*, *style*, *variant*, *size*, *weight*

Any additional keyword arguments are used to override vector properties taken from *Q*.

The positioning of the key depends on *X*, *Y*, *coordinates*, and *labelpos*. If *labelpos* is 'N' or 'S', *X*, *Y* give the position of the middle of the key arrow. If *labelpos* is 'E', *X*, *Y* positions the head, and if *labelpos* is 'W', *X*, *Y* positions the tail; in either of these two cases, *X*, *Y* is somewhere in the middle of the arrow+label key object.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.rc(*args, **kwargs)`

Set the current rc params. Group is the grouping for the rc, e.g., for `lines.linewidth` the group is `lines`, for `axes.facecolor`, the group is `axes`, and so on. Group may also be a list or tuple of group names, e.g., (*xtick*, *ytick*). *kwargs* is a dictionary attribute name/value pairs, e.g.,:

```
rc('lines', linewidth=2, color='r')
```

sets the current rc params and is equivalent to:

```
rcParams['lines.linewidth'] = 2
rcParams['lines.color'] = 'r'
```

The following aliases are available to save typing for interactive users:

Alias	Property
'lw'	'linewidth'
'ls'	'linestyle'
'c'	'color'
'fc'	'facecolor'
'ec'	'edgecolor'
'mew'	'markeredgewidth'
'aa'	'antialiased'

Thus you could abbreviate the above rc command as:

```
rc('lines', lw=2, c='r')
```

Note you can use python's kwargs dictionary facility to store dictionaries of default parameters. e.g., you can customize the font rc as follows:

```
font = {'family' : 'monospace',
        'weight' : 'bold',
        'size'   : 'larger'}

rc('font', **font)  # pass in the font dict as kwargs
```

This enables you to easily switch between several configurations. Use `rcdefaults()` to restore the default rc params after changes.

`matplotlib.pyplot.rc_context(rc=None, fname=None)`

Return a context manager for managing rc settings.

This allows one to do:

```
with mpl.rc_context(fname='screen.rc'):
    plt.plot(x, a)
    with mpl.rc_context(fname='print.rc'):
        plt.plot(x, b)
    plt.plot(x, c)
```

The 'a' vs 'x' and 'c' vs 'x' plots would have settings from 'screen.rc', while the 'b' vs 'x' plot would have settings from 'print.rc'.

A dictionary can also be passed to the context manager:

```
with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
    plt.plot(x, a)
```

The 'rc' dictionary takes precedence over the settings loaded from 'fname'. Passing a dictionary only is also valid.

`matplotlib.pyplot.rcdefaults()`

Restore the default rc params. These are not the params loaded by the rc file, but mpl's internal params. See rc_file_defaults for reloading the default params from the rc file

`matplotlib.pyplot.rgrids(*args, **kwargs)`

Get or set the radial gridlines on a polar plot.

call signatures:

```
lines, labels = rgrids()
lines, labels = rgrids(radii, labels=None, angle=22.5, **kwargs)
```

When called with no arguments, `rgrid()` simply returns the tuple (*lines*, *labels*), where *lines* is an array of radial gridlines (`Line2D` instances) and *labels* is an array of tick labels (`Text` instances). When called with arguments, the labels will appear at the specified radial distances and angles.

labels, if not *None*, is a len(*radii*) list of strings of the labels to use at each angle.

If *labels* is None, the rformatter will be used

Examples:

```
# set the locations of the radial gridlines and labels
lines, labels = rgrids( (0.25, 0.5, 1.0) )

# set the locations and labels of the radial gridlines and labels
lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry' )
```

`matplotlib.pyplot.savefig(*args, **kwargs)`

Save the current figure.

Call signature:

```
savefig(fname, dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False, bbox_inches=None, pad_inches=0.1,
        frameon=None)
```

The output formats available depend on the backend being used.

Arguments:

***fname*:** A string containing a path to a filename, or a Python file-like object, or possibly some backend-dependent object such as `PdfPages`.

If *format* is *None* and *fname* is a string, the output format is deduced from the extension of the filename. If the filename has no extension, the value of the rc parameter `savefig.format` is used.

If *fname* is not a string, remember to specify *format* to ensure that the correct backend is used.

Keyword arguments:

***dpi*:** **[*None* | `scalar > 0` | 'figure']** The resolution in dots per inch. If *None* it will default to the value `savefig.dpi` in the matplotlibrc file. If 'figure' it will set the dpi to be the value of the figure.

***facecolor*, *edgecolor*:** the colors of the figure rectangle

***orientation*: ['landscape' | 'portrait']** not supported on all backends; currently only on postscript output

***papertype*:** One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

***format*:** One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

***transparent*:** If *True*, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

***frameon*:** If *True*, the figure patch will be colored, if *False*, the figure background will be transparent. If not provided, the rcParam 'savefig.frameon' will be used.

***bbox_inches*:** Bbox in inches. Only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

***pad_inches*:** Amount of padding around the figure when bbox_inches is 'tight'.

***bbox_extra_artists*:** A list of extra artists that will be considered when the tight bbox is calculated.

`matplotlib.pyplot.sca(ax)`

Set the current Axes instance to *ax*.

The current Figure is updated to the parent of *ax*.

`matplotlib.pyplot.scatter(x, y, s=20, c=None, marker='o', cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, edgecolors=None, hold=None, data=None, **kwargs)`

Make a scatter plot of x vs y, where x and y are sequence like objects of the same length.

Parameters x, y : array_like, shape (n,)

Input data

s : scalar or array_like, shape (n,), optional, default: 20

size in points^2.

c : color, sequence, or sequence of color, optional, default: 'b'

`c` can be a single color format string, or a sequence of color specifications of length `N`, or a sequence of `N` numbers to be mapped to colors using the `cmap` and `norm` specified via kwargs (see below). Note that `c` should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. `c` can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.

marker : `MarkerStyle`, optional, default: 'o'

See `markers` for more information on the different styles of markers scatter supports. `marker` can be either an instance of the class or the text shorthand for a particular marker.

cmap : `Colormap`, optional, default: None

A `Colormap` instance or registered name. `cmap` is only used if `c` is an array of floats. If None, defaults to rc `image.cmap`.

norm : *Normalize*, optional, default: None
: A *Normalize* instance is used to scale luminance data to 0, 1. `norm` is only used if `c` is an array of floats. If `None`, use the default `normalize()`.

vmin, vmax : scalar, optional, default: None
: `vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. If either are `None`, the min and max of the color array is used. Note if you pass a `norm` instance, your settings for `vmin` and `vmax` will be ignored.

alpha : scalar, optional, default: None
: The alpha blending value, between 0 (transparent) and 1 (opaque)

linewidths : scalar or array_like, optional, default: None
: If None, defaults to (lines.linewidth,).

edgecolors : color or sequence of color, optional, default: None
: If None, defaults to (patch.edgecolor). If 'face', the edge color will always be the same as the face color. If it is 'none', the patch boundary will not be drawn. For non-filled markers, the `edgecolors` kwarg is ignored; color is determined by `c`.

Returns paths : *PathCollection*

Other Parameters kwargs : *Collection* properties

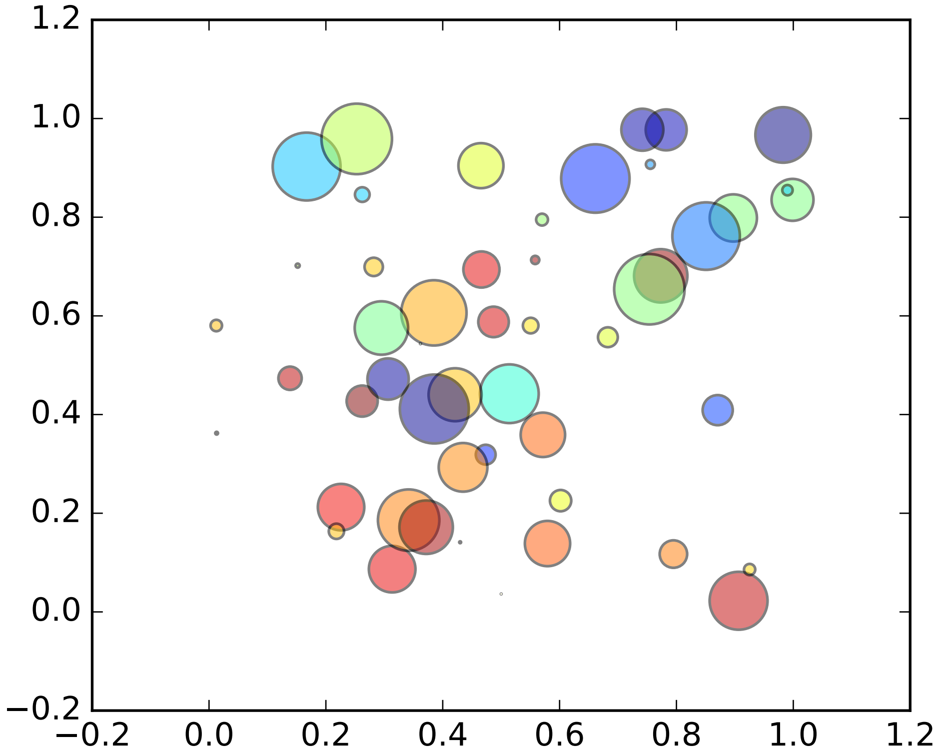
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 's', 'y', 'linewidths', 'color', 'c', 'facecolors', 'x', 'facecolor', 'edgecolors'.

Additional kwargs: hold = [True|False] overrides default hold state

Examples



`matplotlib.pyplot.sci(im)`

Set the current image. This image will be the target of colormap commands like `jet()`, `hot()` or `clim()`). The current image is an attribute of the current axes.

`matplotlib.pyplot.semilogx(*args, **kwargs)`

Make a plot with log scaling on the *x* axis.

Call signature:

```
semilogx(*args, **kwargs)
```

`semilogx()` supports all the keyword arguments of `plot()` and `matplotlib.axes.Axes.set_xscale()`.

Notable keyword arguments:

***basex*: scalar > 1** Base of the *x* logarithm

***subsx*: [*None* | sequence]** The location of the minor xticks; *None* defaults to autosubs, which depend on the number of decades in the plot; see `set_xscale()` for details.

***nonposx*: ['mask' | 'clip']** Non-positive values in *x* can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are `Line2D` properties:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
antialiased or aa	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a matplotlib.figure.Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
gid	an id string
label	string or anything printable with '%s' conversion.
linestyle or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '--' | '-.' | '
linewidth or lw	float value in points
marker	A valid marker style
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function `fn(artist, event)`
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See also:

`loglog()` For example code and figure

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.semilogy(*args, **kwargs)`

Make a plot with log scaling on the *y* axis.

call signature:

```
semilogy(*args, **kwargs)
```

`semilogy()` supports all the keyword arguments of `plot()` and `matplotlib.axes.Axes.set_yscale()`.

Notable keyword arguments:

***basey*: scalar > 1** Base of the *y* logarithm

***subsy*: [*None* | sequence]** The location of the minor yticks; *None* defaults to autosubs, which depend on the number of decades in the plot; see `set_yscale()` for details.

***nonposy*: ['mask' | 'clip']** Non-positive values in *y* can be masked as invalid, or clipped to a very small positive number

The remaining valid kwargs are `Line2D` properties:

Property	Description
`agg_filter`	unknown
`alpha`	float (0.0 transparent through 1.0 opaque)
`animated`	[True | False]
`antialiased` or aa	[True | False]
`axes`	an `Axes` instance
`clip_box`	a `matplotlib.transforms.Bbox` instance
`clip_on`	[True | False]
`clip_path`	[(`Path`, `Transform`) | `Patch` | None]
`color` or c	any matplotlib color
`contains`	a callable function
`dash_capstyle`	['butt' | 'round' | 'projecting']
`dash_joinstyle`	['miter' | 'round' | 'bevel']
`dashes`	sequence of on/off ink in points
`drawstyle`	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
`figure`	a `matplotlib.figure.Figure` instance
`fillstyle`	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
`gid`	an id string
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | '
`linewidth` or lw	float value in points
`marker`	`A valid marker style`
`markeredgecolor` or mec	any matplotlib color
`markeredgewidth` or mew	float value in points
`markerfacecolor` or mfc	any matplotlib color
`markerfacecoloralt` or mfcalt	any matplotlib color
`markersize` or ms	float

C

Table 68.28 – continued from previous page

Property	Description
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]
path_effects	unknown
picker	float distance in points or callable pick function fn(artist, event)
pickradius	float distance in points
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	[True | False]
xdata	1D array
ydata	1D array
zorder	any number

See also:

loglog() For example code and figure

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**set_cmap**(*cmap*)

Set the default colormap. Applies to the current image if any. See help(colormaps) for more information.

cmap must be a Colormap instance, or the name of a registered colormap.

See matplotlib.cm.register_cmap() and matplotlib.cm.get_cmap().

matplotlib.pyplot.**setp**(**args*, ***kwargs*)

Set a property on an artist object.

matplotlib supports the use of setp() ("set property") and getp() to set and get object properties, as well as to do introspection on the object. For example, to set the linestyle of a line to be dashed, you can do:

```
>>> line, = plot([1,2,3])
>>> setp(line, linestyle='--')
```

If you want to know the valid types of arguments, you can provide the name of the property you want to set without a value:

```
>>> setp(line, 'linestyle')
    linestyle: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]
```

If you want to see all the properties that can be set, and their possible values, you can do:

```
>>> setp(line)
    ... long output listing omitted
```

`setp()` operates on a single instance or a list of instances. If you are in query mode introspecting the possible values, only the first instance in the sequence is used. When actually setting values, all the instances will be set. e.g., suppose you have a list of two lines, the following will make both lines thicker and red:

```
>>> x = arange(0,1.0,0.01)
>>> y1 = sin(2*pi*x)
>>> y2 = sin(4*pi*x)
>>> lines = plot(x, y1, x, y2)
>>> setp(lines, linewidth=2, color='r')
```

`setp()` works with the MATLAB style string/value pairs or with python kwargs. For example, the following are equivalent:

```
>>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
>>> setp(lines, linewidth=2, color='r')        # python style
```

`matplotlib.pyplot.`**`show`**(**args*, ***kw*)

Display a figure. When running in ipython with its pylab mode, display all figures and return to the ipython prompt.

In non-interactive mode, display all figures and block until the figures have been closed; in interactive mode it has no effect unless figures were created prior to a change from non-interactive to interactive mode (not recommended). In that case it displays the figures but does not block.

A single experimental keyword argument, *block*, may be set to True or False to override the blocking behavior described above.

`matplotlib.pyplot.`**`specgram`**(*x*, *NFFT=None*, *Fs=None*, *Fc=None*, *detrend=None*, *window=None*, *noverlap=None*, *cmap=None*, *xextent=None*, *pad_to=None*, *sides=None*, *scale_by_freq=None*, *mode=None*, *scale=None*, *vmin=None*, *vmax=None*, *hold=None*, *data=None*, ***kwargs*)

Plot a spectrogram.

Call signature:

```
specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
         window=mlab.window_hanning, noverlap=128,
         cmap=None, xextent=None, pad_to=None, sides='default',
         scale_by_freq=None, mode='default', scale='default',
         **kwargs)
```

Compute and plot a spectrogram of data in *x*. Data are split into *NFFT* length segments and the spectrum of each section is computed. The windowing function *window* is applied to each segment, and the amount of overlap of each segment is specified with *noverlap*. The spectrogram is plotted as a colormap (using imshow).

***x*: 1-D array or sequence** Array or sequence containing the data

Keyword arguments:

***Fs*: scalar** The sampling frequency (samples per time unit). It is used to calculate the Fourier frequencies, freqs, in cycles per time unit. The default value is 2.

***window*: callable or ndarray** A function or a vector of length *NFFT*. To create window vectors see `window_hanning()`, `window_none()`, `numpy.blackman()`, `numpy.hamming()`, `numpy.bartlett()`, `scipy.signal()`, `scipy.signal.get_window()`, etc. The default is `window_hanning()`. If a function is passed as the argument, it must take a data segment as an argument and return the windowed version of the segment.

***sides*: ['default' | 'onesided' | 'twosided']** Specifies which sides of the spectrum to return. Default gives the default behavior, which returns one-sided for real data and both for complex data. 'onesided' forces the return of a one-sided spectrum, while 'twosided' forces two-sided.

***pad_to*: integer** The number of points to which the data segment is padded when performing the FFT. This can be different from *NFFT*, which specifies the number of data points used. While not increasing the actual resolution of the spectrum (the minimum distance between resolvable peaks), this can give more points in the plot, allowing for more detail. This corresponds to the *n* parameter in the call to fft(). The default is None, which sets *pad_to* equal to *NFFT*

***NFFT*: integer** The number of data points used in each block for the FFT. A power 2 is most efficient. The default value is 256. This should *NOT* be used to get zero padding, or the scaling of the result will be incorrect. Use *pad_to* for this instead.

***detrend*: ['default' | 'constant' | 'mean' | 'linear' | 'none'] or** callable

The function applied to each segment before fft-ing, designed to remove the mean or linear trend. Unlike in MATLAB, where the *detrend* parameter is a vector, in matplotlib is it a function. The `pylab` module defines `detrend_none()`, `detrend_mean()`, and `detrend_linear()`, but you can use a custom function as well. You can also use a string to choose one of the functions. 'default', 'constant', and 'mean' call `detrend_mean()`. 'linear' calls `detrend_linear()`. 'none' calls `detrend_none()`.

***scale_by_freq*: boolean**

Specifies whether the resulting density values should be scaled by the scaling frequency, which gives density in units of Hz^-1. This allows for integration over the returned frequency values. The default is True for MATLAB compatibility.

***mode*: ['default' | 'psd' | 'magnitude' | 'angle' | 'phase']** What sort of spectrum to use. Default is 'psd'. which takes the power spectral density. 'complex' returns the complex-valued frequency spectrum. 'magnitude' returns the magnitude spectrum. 'angle' returns the phase spectrum without unwrapping. 'phase' returns the phase spectrum with unwrapping.

***noverlap*: integer** The number of points of overlap between blocks. The default value is 128.

***scale*: ['default' | 'linear' | 'dB']** The scaling of the values in the *spec*. 'linear' is no scaling. 'dB' returns the values in dB scale. When *mode* is 'psd', this is dB power (10 * log10). Otherwise this is dB amplitude (20 * log10). 'default' is 'dB' if *mode* is 'psd' or 'magnitude' and 'linear' otherwise. This must be 'linear' if *mode* is 'angle' or 'phase'.

***Fc*: integer** The center frequency of *x* (defaults to 0), which offsets the x extents of the plot to reflect the frequency range used when a signal is acquired and then filtered and downsampled to baseband.

***cmap*:** A `matplotlib.colors.Colormap` instance; if *None*, use default determined by rc

***xextent*:** The image extent along the x-axis. xextent = (xmin,xmax) The default is

(0,max(bins)), where bins is the return value from `specgram()`

kwargs: Additional kwargs are passed on to imshow which makes the specgram image

---

Note: *detrend* and *scale_by_freq* only apply when *mode* is set to 'psd'

Returns the tuple (*spectrum*, *freqs*, *t*, *im*):

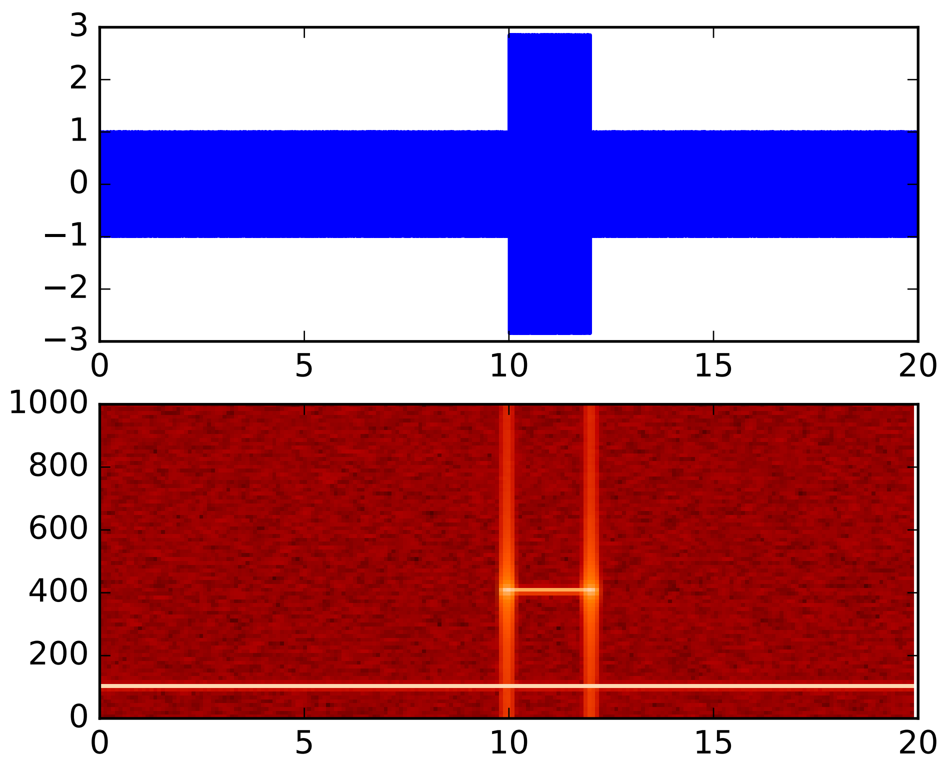
spectrum: **2-D array** columns are the periodograms of successive segments

freqs: **1-D array** The frequencies corresponding to the rows in *spectrum*

t: **1-D array** The times corresponding to midpoints of segments (i.e the columns in *spectrum*)

im: **instance of class** `AxesImage` The image created by imshow containing the spectrogram

Example:



See also:

`psd()` `psd()` differs in the default overlap; in returning the mean of the segment periodograms; in not returning times; and in generating a line plot instead of colormap.

`magnitude_spectrum()` A single spectrum, similar to having a single segment when *mode* is 'magnitude'. Plots a line instead of a colormap.

`angle_spectrum()` A single spectrum, similar to having a single segment when *mode* is 'angle'. Plots a line instead of a colormap.

`phase_spectrum()` A single spectrum, similar to having a single segment when *mode* is 'phase'.

Plots a line instead of a colormap.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.spectral()`

set the default colormap to spectral and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.spring()`

set the default colormap to spring and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.spy(`*Z*, *precision=0*, *marker=None*, *markersize=None*, *aspect='equal'*, *hold=None*, ***kwargs*`)`

Plot the sparsity pattern on a 2-D array.

`spy(Z)` plots the sparsity pattern of the 2-D array *Z*.

Parameters Z : sparse array (n, m)

The array to be plotted.

precision : float, optional, default: 0

If *precision* is 0, any non-zero value will be plotted; else, values of $|Z| > precision$ will be plotted.

For `scipy.sparse.spmatrix` instances, there is a special case: if *precision* is 'present', any value present in the array will be plotted, even if it is identically zero.

origin : ["upper", "lower"], optional, default: "upper"

Place the [0,0] index of the array in the upper left or lower left corner of the axes.

aspect : ['auto' | 'equal' | scalar], optional, default: "equal"

If 'equal', and `extent` is None, changes the axes aspect ratio to match that of the image. If `extent` is not `None`, the axes aspect ratio is changed to match that of the extent.

If 'auto', changes the image aspect ratio to match that of the axes.

If None, default to rc `image.aspect` value.

Two plotting styles are available: image or marker. Both

are available for full arrays, but only the marker style

works for :class:'scipy.sparse.spmatrix' instances.

If *marker* and *markersize* are *None*, an image will be

returned and any remaining kwargs are passed to

:func:'~matplotlib.pyplot.imshow'; else, a

:class:'~matplotlib.lines.Line2D' object will be returned with

the value of marker determining the marker type, and any

remaining kwargs passed to the

:meth:'~matplotlib.axes.Axes.plot' method.

If *marker* and *markersize* are *None*, useful kwargs include:

*** *cmap***

*** *alpha***

See also:

`imshow` for image options.
`plot` for plotting options

`Additional`

`matplotlib.pyplot.stackplot`(*x*, **args*, ***kwargs*)

Draws a stacked area plot.

x : 1d array of dimension N

y [2d array of dimension MxN, OR any number 1d arrays each of dimension] 1xN. The data is assumed to be unstacked. Each of the following calls is legal:

```
stackplot(x, y)               # where y is MxN
stackplot(x, y1, y2, y3, y4)  # where y1, y2, y3, y4, are all 1xNm
```

Keyword arguments:

baseline [['zero', 'sym', 'wiggle', 'weighted_wiggle']] Method used to calculate the baseline. 'zero' is just a simple stacked plot. 'sym' is symmetric around zero and is sometimes called `ThemeRiver`. 'wiggle' minimizes the sum of the squared slopes. 'weighted_wiggle' does the same but weights to account for size of each layer. It is also called `Streamgraph`-layout. More details can be found at http://www.leebyron.com/else/streamgraph/.

labels : A list or tuple of labels to assign to each data series.

colors [A list or tuple of colors. These will be cycled through and] used to colour the stacked areas. All other keyword arguments are passed to `fill_between()`

Returns *r* : A list of `PolyCollection`, one for each element in the stacked area plot.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.stem`(**args*, ***kwargs*)

Create a stem plot.

Call signatures:

```
stem(y, linefmt='b-', markerfmt='bo', basefmt='r-')
stem(x, y, linefmt='b-', markerfmt='bo', basefmt='r-')
```

A stem plot plots vertical lines (using *linefmt*) at each *x* location from the baseline to *y*, and places a marker there using *markerfmt*. A horizontal line at 0 is is plotted using *basefmt*.

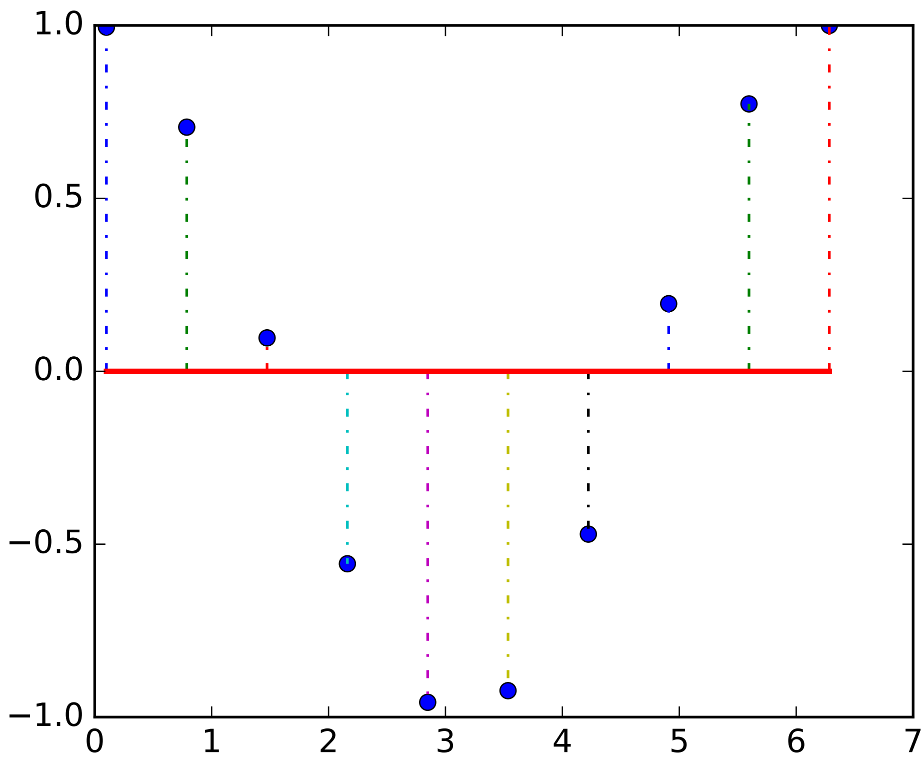
If no *x* values are provided, the default is (0, 1, ..., len(y) - 1)

Return value is a tuple (*markerline*, *stemlines*, *baseline*).

See also:

This document for details.

Example:



Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All positional and all keyword arguments.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.step(x, y, *args, **kwargs)`

Make a step plot.

Call signature:

```
step(x, y, *args, **kwargs)
```

Additional keyword args to `step()` are the same as those for `plot()`.

x and *y* must be 1-D sequences, and it is assumed, but not checked, that *x* is uniformly increasing.

Keyword arguments:

***where*: ['pre' | 'post' | 'mid']** If 'pre' (the default), the interval from x[i] to x[i+1] has level y[i+1].

If 'post', that interval has level y[i].

If 'mid', the jumps in *y* occur half-way between the *x*-values.

Return value is a list of lines that were added.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'y', 'x'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.streamplot`(*x*, *y*, *u*, *v*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='-|>'*, *minlength=0.1*, *transform=None*, *zorder=1*, *start_points=None*, *hold=None*, *data=None*)

Draws streamlines of a vector flow.

x*, *y [1d arrays] an *evenly spaced* grid.

u*, *v [2d arrays] x and y-velocities. Number of rows should match length of y, and the number of columns should match x.

density [float or 2-tuple] Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid—*density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use [density_x, density_y].

linewidth [numeric or 2d array] vary linewidth when given a 2d array with the same shape as velocities.

color [matplotlib color code, or 2d array] Streamline color. When given an array with the same shape as velocities, *color* values are converted to colors using *cmap*.

cmap [`Colormap`] Colormap used to plot streamlines and arrows. Only necessary when using an array input for *color*.

norm [`Normalize`] Normalize object used to scale luminance data to 0, 1. If None, stretch (min, max) to (0, 1). Only necessary when *color* is an array.

arrowsize [float] Factor scale arrow size.

arrowstyle [str] Arrow style specification. See `FancyArrowPatch`.

minlength [float] Minimum length of streamline in axes coordinates.

***start_points*: Nx2 array** Coordinates of starting points for the streamlines. In data coordinates, the same as the `x` and `y` arrays.

zorder [int] any number

Returns:

stream_container [StreamplotSet] Container object with attributes

- lines: `matplotlib.collections.LineCollection` of streamlines
- arrows: collection of `matplotlib.patches.FancyArrowPatch` objects representing arrows half-way along stream lines.

This container will probably change in the future to allow changes to the colormap, alpha, etc. for both lines and arrows, but these changes should be backward compatible.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.subplot(*args, **kwargs)`

Return a subplot axes positioned by the given grid definition.

Typical call signature:

```
subplot(nrows, ncols, plot_number)
```

Where *nrows* and *ncols* are used to notionally split the figure into `nrows * ncols` sub-axes, and *plot_number* is used to identify the particular subplot that this function is to create within the notional grid. *plot_number* starts at 1, increments across rows first and has a maximum of `nrows * ncols`.

In the case when *nrows*, *ncols* and *plot_number* are all less than 10, a convenience exists, such that the a 3 digit number can be given instead, where the hundreds represent *nrows*, the tens represent *ncols* and the units represent *plot_number*. For instance:

```
subplot(211)
```

produces a subaxes in a figure which represents the top plot (i.e. the first) in a 2 row by 1 column notional grid (no grid actually exists, but conceptually this is how the returned subplot has been positioned).

Note: Creating a new subplot with a position which is entirely inside a pre-existing axes will trigger the larger axes to be deleted:

```
import matplotlib.pyplot as plt
# plot a line, implicitly creating a subplot(111)
plt.plot([1,2,3])
# now create a subplot which represents the top plot of a grid
# with 2 rows and 1 column. Since this subplot will overlap the
# first, the plot (and its axes) previously created, will be removed
plt.subplot(211)
plt.plot(range(12))
plt.subplot(212, axisbg='y') # creates 2nd subplot with yellow background
```

If you do not want this behavior, use the `add_subplot()` method or the `axes()` function instead.

Keyword arguments:

axisbg: The background color of the subplot, which can be any valid color specifier. See `matplotlib.colors` for more information.

polar: A boolean flag indicating whether the subplot plot should be a polar projection. Defaults to *False*.

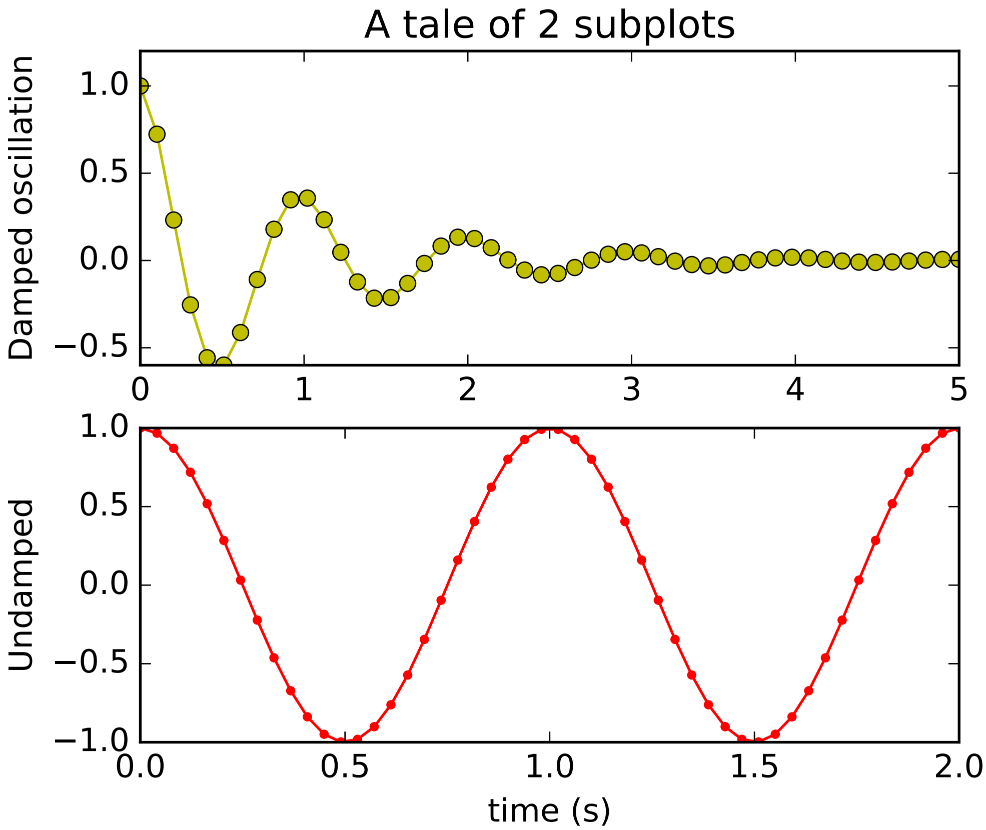
projection: A string giving the name of a custom projection to be used for the subplot. This projection must have been previously registered. See `matplotlib.projections`.

See also:

`axes()` For additional information on `axes()` and `subplot()` keyword arguments.

examples/pie_and_polar_charts/polar_scatter_demo.py For an example

Example:



`matplotlib.pyplot.`**`subplot2grid`**(*shape*, *loc*, *rowspan=1*, *colspan=1*, ***kwargs*)

Create a subplot in a grid. The grid is specified by *shape*, at location of *loc*, spanning *rowspan*, *colspan* cells in each direction. The index for loc is 0-based.

```
subplot2grid(shape, loc, rowspan=1, colspan=1)
```

is identical to

```
gridspec=GridSpec(shape[0], shape[1])
subplotspec=gridspec.new_subplotspec(loc, rowspan, colspan)
subplot(subplotspec)
```

`matplotlib.pyplot.`**`subplot_tool`**(*targetfig=None*)

Launch a subplot tool window for a figure.

A `matplotlib.widgets.SubplotTool` instance is returned.

`matplotlib.pyplot.`**`subplots`**(*nrows=1*, *ncols=1*, *sharex=False*, *sharey=False*, *squeeze=True*, *subplot_kw=None*, *gridspec_kw=None*, ***fig_kw*)

Create a figure with a set of subplots already made.

This utility wrapper makes it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

Keyword arguments:

nrows [int] Number of rows of the subplot grid. Defaults to 1.

ncols [int] Number of columns of the subplot grid. Defaults to 1.

sharex [string or bool] If *True*, the X axis will be shared amongst all subplots. If *True* and you have multiple rows, the x tick labels on all but the last row of plots will have visible set to *False* If a string must be one of "row", "col", "all", or "none". "all" has the same effect as *True*, "none" has the same effect as *False*. If "row", each subplot row will share a X axis. If "col", each subplot column will share a X axis and the x tick labels on all but the last row will have visible set to *False*.

sharey [string or bool] If *True*, the Y axis will be shared amongst all subplots. If *True* and you have multiple columns, the y tick labels on all but the first column of plots will have visible set to *False* If a string must be one of "row", "col", "all", or "none". "all" has the same effect as *True*, "none" has the same effect as *False*. If "row", each subplot row will share a Y axis and the y tick labels on all but the first column will have visible set to *False*. If "col", each subplot column will share a Y axis.

squeeze [bool] If *True*, extra dimensions are squeezed out from the returned axis object:

- if only one subplot is constructed (nrows=ncols=1), the resulting single Axis object is returned as a scalar.
- for Nx1 or 1xN subplots, the returned object is a 1-d numpy object array of Axis objects are returned as numpy 1-d arrays.
- for NxM subplots with N>1 and M>1 are returned as a 2d array.

If *False*, no squeezing at all is done: the returned axis object is always a 2-d array containing Axis instances, even if it ends up being 1x1.

subplot_kw [dict] Dict with keywords passed to the `add_subplot()` call used to create each subplots.

gridspec_kw [dict] Dict with keywords passed to the `GridSpec` constructor used to create the grid the subplots are placed on.

fig_kw [dict] Dict with keywords passed to the `figure()` call. Note that all keywords not recognized above will be automatically included here.

Returns:

fig, ax : tuple

- *fig* is the `matplotlib.figure.Figure` object
- *ax* can be either a single axis object or an array of axis objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the squeeze keyword, see above.

Examples:

```
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Just a figure and one subplot
f, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Two subplots, unpack the output array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)
```

```
# Four polar axes
plt.subplots(2, 2, subplot_kw=dict(polar=True))

# Share a X axis with each column of subplots
plt.subplots(2, 2, sharex='col')

# Share a Y axis with each row of subplots
plt.subplots(2, 2, sharey='row')

# Share a X and Y axis with all subplots
plt.subplots(2, 2, sharex='all', sharey='all')
# same as
plt.subplots(2, 2, sharex=True, sharey=True)
```

`matplotlib.pyplot.`**`subplots_adjust`**`(*args, **kwargs)`

Tune the subplot layout.

call signature:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

The parameter meanings (and suggested defaults) are:

```
left  = 0.125  # the left side of the subplots of the figure
right = 0.9    # the right side of the subplots of the figure
bottom = 0.1   # the bottom of the subplots of the figure
top = 0.9      # the top of the subplots of the figure
wspace = 0.2   # the amount of width reserved for blank space between subplots
hspace = 0.2   # the amount of height reserved for white space between subplots
```

The actual defaults are controlled by the rc file

`matplotlib.pyplot.`**`summer`**`()`

set the default colormap to summer and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.`**`suptitle`**`(*args, **kwargs)`

Add a centered title to the figure.

kwargs are `matplotlib.text.Text` properties. Using figure coordinates, the defaults are:

x [0.5] The x location of the text in figure coords

y [0.98] The y location of the text in figure coords

horizontalalignment ['center'] The horizontal alignment of the text

verticalalignment ['top'] The vertical alignment of the text

A `matplotlib.text.Text` instance is returned.

Example:

```
fig.suptitle('this is the figure title', fontsize=12)
```

`matplotlib.pyplot.`**`switch_backend`**`(newbackend)`

Switch the default backend. This feature is **experimental**, and is only expected to work switching to an image backend. e.g., if you have a bunch of PostScript scripts that you want to run from an interactive ipython session, you may want to switch to the PS backend before running them to avoid having a bunch of GUI windows popup. If you try to interactively switch from one GUI backend to another, you will explode.

Calling this command will close all open windows.

`matplotlib.pyplot.table(**kwargs)`

Add a table to the current axes.

Call signature:

```
table(cellText=None, cellColours=None,
      cellLoc='right', colWidths=None,
      rowLabels=None, rowColours=None, rowLoc='left',
      colLabels=None, colColours=None, colLoc='center',
      loc='bottom', bbox=None):
```

Returns a `matplotlib.table.Table` instance. For finer grained control over tables, use the `Table` class and add it to the axes with `add_table()`.

Thanks to John Gill for providing the class and table.

kwargs control the `Table` properties:

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
axes	an Axes instance
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
contains	a callable function
figure	a matplotlib.figure.Figure instance
fontsize	a float in points
gid	an id string
label	string or anything printable with '%s' conversion.
path_effects	unknown
picker	[None|float|boolean|callable]
rasterized	[True | False | None]
sketch_params	unknown
snap	unknown
transform	Transform instance
url	a url string
visible	[True | False]
zorder	any number

`matplotlib.pyplot.text(x, y, s, fontdict=None, withdash=False, **kwargs)`

Add text to the axes.

Add text in string s to axis at location x, y, data coordinates.

Parameters x, y : scalars
  data coordinates

s : string
  text

fontdict : dictionary, optional, default: None
  A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.

withdash : boolean, optional, default: False
  Creates a *TextWithDash* instance instead of a *Text* instance.

Other Parameters kwargs : *Text* properties.
  Other miscellaneous text parameters.

Examples

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the axes:

```
>>> text(0.5, 0.5,'matplotlib', horizontalalignment='center',
...      verticalalignment='center',
...      transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword bbox. bbox is a dictionary of *Rectangle* properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

matplotlib.pyplot.**thetagrids**(**args*, ***kwargs*)

Get or set the theta locations of the gridlines in a polar plot.

If no arguments are passed, return a tuple (*lines*, *labels*) where *lines* is an array of radial gridlines (*Line2D* instances) and *labels* is an array of tick labels (*Text* instances):

```
lines, labels = thetagrids()
```

Otherwise the syntax is:

```
lines, labels = thetagrids(angles, labels=None, fmt='%d', frac = 1.1)
```

set the angles at which to place the theta grids (these gridlines are equal along the theta dimension).

angles is in degrees.

labels, if not *None*, is a len(angles) list of strings of the labels to use at each angle.

If *labels* is *None*, the labels will be fmt%angle.

frac is the fraction of the polar axes radius at which to place the label (1 is the edge). e.g., 1.05 is outside the axes and 0.95 is inside the axes.

Return value is a list of tuples (*lines*, *labels*):

- *lines* are `Line2D` instances
- *labels* are `Text` instances.

Note that on input, the *labels* argument is a list of strings, and on output it is a list of `Text` instances.

Examples:

```
# set the locations of the radial gridlines and labels
lines, labels = thetagrids( range(45,360,90) )

# set the locations and labels of the radial gridlines and labels
lines, labels = thetagrids( range(45,360,90), ('NE', 'NW', 'SW','SE') )
```

`matplotlib.pyplot.`**`tick_params`**(*axis='both'*, ***kwargs*)

Change the appearance of ticks and tick labels.

Keyword arguments:

axis [['x' | 'y' | 'both']] Axis on which to operate; default is 'both'.

reset [[True | False]] If *True*, set all parameters to defaults before processing other keyword arguments. Default is *False*.

which [['major' | 'minor' | 'both']] Default is 'major'; apply arguments to *which* ticks.

direction [['in' | 'out' | 'inout']] Puts ticks inside the axes, outside the axes, or both.

length Tick length in points.

width Tick width in points.

color Tick color; accepts any mpl color spec.

pad Distance in points between tick and label.

labelsize Tick label font size in points or as a string (e.g., 'large').

labelcolor Tick label color; mpl color spec.

colors Changes the tick color and the label color to the same value: mpl color spec.

zorder Tick and label zorder.

bottom*, *top*, *left*, *right [[bool | 'on' | 'off']] controls whether to draw the respective ticks.

labelbottom*, *labeltop*, *labelleft*, *labelright Boolean or ['on' | 'off'], controls whether to draw the respective tick labels.

Example:

```
ax.tick_params(direction='out', length=6, width=2, colors='r')
```

This will make all major ticks be red, pointing out of the box, and with dimensions 6 points by 2 points. Tick labels will also be red.

`matplotlib.pyplot.`**`ticklabel_format`**(***kwargs*)

Change the `ScalarFormatter` used by default for linear axes.

Optional keyword arguments:

Keyword	Description
style	['sci' (or 'scientific') | 'plain'] plain turns off scientific notation
scilimits	(m, n), pair of integers; if *style* is 'sci', scientific notation will be used for numbers outside the range 10'm':sup: to 10'n':sup:. Use (0,0) to include all numbers.
useOffset	[True | False | offset]; if True, the offset will be calculated as needed; if False, no offset will be used; if a numeric offset is specified, it will be used.
axis	['x' | 'y' | 'both']
useLocale	If True, format the number according to the current locale. This affects things such as the character used for the decimal separator. If False, use C-style (English) formatting. The default setting is controlled by the axes.formatter.use_locale rcparam.

Only the major ticks are affected. If the method is called when the `ScalarFormatter` is not the `Formatter` being used, an `AttributeError` will be raised.

`matplotlib.pyplot.tight_layout`(*pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Automatically adjust subplot parameters to give specified padding.

Parameters:

pad [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.

h_pad, w_pad [float] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.

rect [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

`matplotlib.pyplot.title`(*s*, **args*, ***kwargs*)

Set a title of the current axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

See also:

See `text()` for adding text to the current axes

Parameters label : str

Text to use for the title

fontdict : dict

A dictionary controlling the appearance of the title text, the default `fontdict` is:

{'fontsize': rcParams['axes.titlesize'], 'fontweight' : rcParams['axes.titleweight'], 'verticalalignment': 'baseline', 'horizontalalignment': loc}

loc : {'center', 'left', 'right'}, str, optional

Which title to set, defaults to 'center'

Returns text : `Text`

The matplotlib text instance representing the title

Other Parameters kwargs : text properties

Other keyword arguments are text properties, see *Text* for a list of valid text properties.

`matplotlib.pyplot.tricontour(*args, **kwargs)`

Draw contours on an unstructured triangular grid. *tricontour()* and *tricontourf()* draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

```
tricontour(triangulation, ...)
```

where triangulation is a *matplotlib.tri.Triangulation* object, or

```
tricontour(x, y, ...)
tricontour(x, y, triangles, ...)
tricontour(x, y, triangles=triangles, ...)
tricontour(x, y, mask=mask, ...)
tricontour(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

The remaining arguments may be:

```
tricontour(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontour(..., Z, N)
```

contour *N* automatically-chosen levels.

```
tricontour(..., Z, V)
```

draw contour lines at the values specified in sequence *V*, which must be in increasing order.

```
tricontourf(..., Z, V)
```

fill the (len(*V*)-1) regions between the values in *V*, which must be in increasing order.

```
tricontour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

`C = tricontour(...)` returns a `TriContourSet` object.

Optional keyword arguments:

***colors*: [*None* | string | (mpl_colors)]** If *None*, the colormap specified by cmap will be used.

If a string, like 'r' or 'red', all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

***alpha*: float** The alpha blending value

***cmap*: [*None* | Colormap]** A cm `Colormap` instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

***levels* [level0, level1, ..., leveln]** A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

***origin*: [*None* | 'upper' | 'lower' | 'image']** If *None*, the first value of *Z* will correspond to the lower left corner, location (0,0). If 'image', the rc value for `image.origin` will be used.

This keyword is not active if *X* and *Y* are specified in the call to contour.

extent: [*None* | (x0,x1,y0,y1)]

If *origin* is not *None*, then *extent* is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0], and (*x1*, *y1*) is the position of Z[-1,-1].

This keyword is not active if *X* and *Y* are specified in the call to contour.

***locator*: [*None* | ticker.Locator subclass]** If *locator* is None, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

***extend*: ['neither' | 'both' | 'min' | 'max']** Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

***xunits*, *yunits*: [*None* | registered units]** Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

tricontour-only keyword arguments:

***linewidths*: [*None* | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified

***linestyles*: [*None* | 'solid' | 'dashed' | 'dashdot' | 'dotted']** If *linestyles* is *None*, the 'solid' is used.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in `contour.negative_linestyle` in `matplotlibrc` will be used.

tricontourf-only keyword arguments:

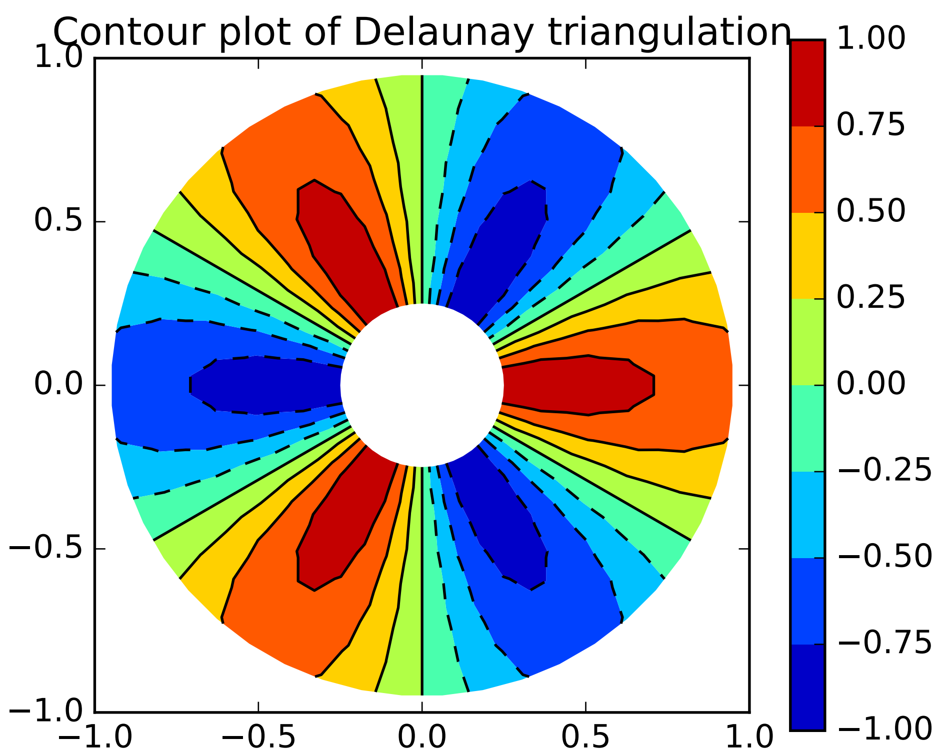
***antialiased*: [*True* | *False*]** enable antialiasing

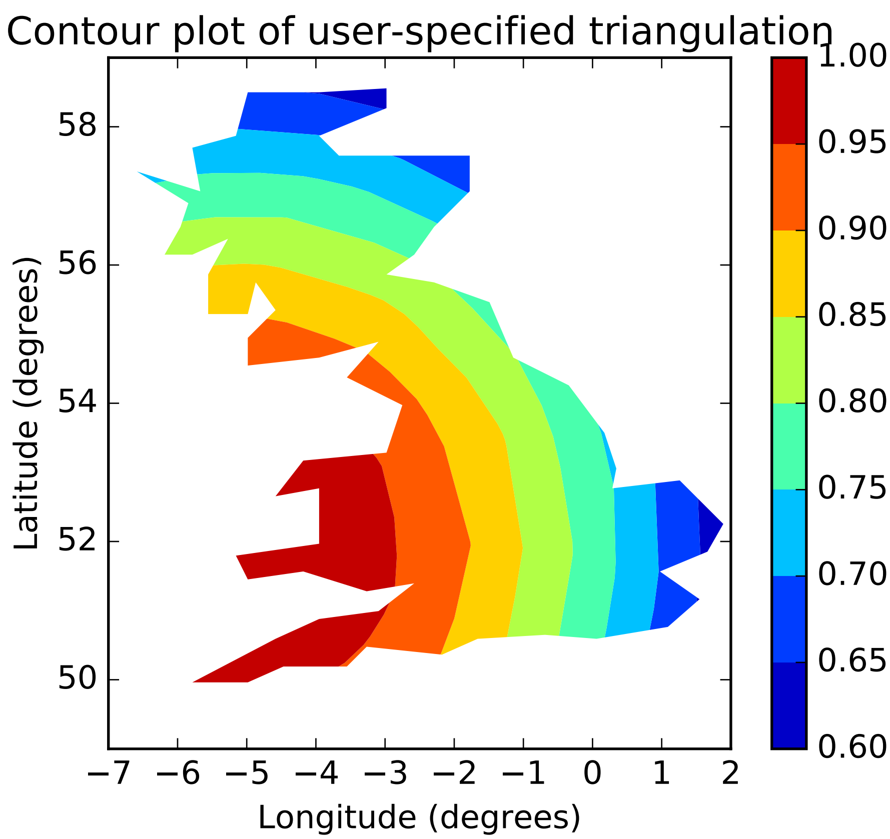
Note: tricontourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < z <= z2
```

There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

Examples:





Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`tricontourf`**(**args*, ***kwargs*)

Draw contours on an unstructured triangular grid. `tricontour()` and `tricontourf()` draw contour lines and filled contours, respectively. Except as noted, function signatures and return values are the same for both versions.

The triangulation can be specified in one of two ways; either:

```
tricontour(triangulation, ...)
```

where triangulation is a `matplotlib.tri.Triangulation` object, or

```
tricontour(x, y, ...)
tricontour(x, y, triangles, ...)
tricontour(x, y, triangles=triangles, ...)
tricontour(x, y, mask=mask, ...)
tricontour(x, y, triangles, mask=mask, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The remaining arguments may be:

```
tricontour(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation. The level values are chosen automatically.

```
tricontour(..., Z, N)
```

contour *N* automatically-chosen levels.

```
tricontour(..., Z, V)
```

draw contour lines at the values specified in sequence *V*, which must be in increasing order.

```
tricontourf(..., Z, V)
```

fill the (len(*V*)-1) regions between the values in *V*, which must be in increasing order.

```
tricontour(Z, **kwargs)
```

Use keyword args to control colors, linewidth, origin, cmap ... see below for more details.

`C = tricontour(...)` returns a `TriContourSet` object.

Optional keyword arguments:

***colors*: [*None* | string | (mpl_colors)]** If *None*, the colormap specified by cmap will be used.

If a string, like 'r' or 'red', all levels will be plotted in this color.

If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

***alpha*: float** The alpha blending value

***cmap*: [*None* | Colormap]** A cm `Colormap` instance or *None*. If *cmap* is *None* and *colors* is *None*, a default Colormap is used.

***norm*: [*None* | Normalize]** A `matplotlib.colors.Normalize` instance for scaling data values to colors. If *norm* is *None* and *colors* is *None*, the default linear scaling is used.

***levels* [level0, level1, ..., leveln]** A list of floating point numbers indicating the level curves to draw, in increasing order; e.g., to draw just the zero contour pass `levels=[0]`

***origin*: [*None* | 'upper' | 'lower' | 'image']** If *None*, the first value of *Z* will correspond to the lower left corner, location (0,0). If 'image', the rc value for `image.origin` will be used.

This keyword is not active if *X* and *Y* are specified in the call to contour.

extent: [*None* | (x0,x1,y0,y1)]

If *origin* is not *None*, then *extent* is interpreted as in `matplotlib.pyplot.imshow()`: it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0], and (*x1*, *y1*) is the position of Z[-1,-1].

This keyword is not active if *X* and *Y* are specified in the call to contour.

***locator*: [*None* | ticker.Locator subclass]** If *locator* is None, the default `MaxNLocator` is used. The locator is used to determine the contour levels if they are not given explicitly via the *V* argument.

***extend*: ['neither' | 'both' | 'min' | 'max']** Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range, but can be set via `matplotlib.colors.Colormap.set_under()` and `matplotlib.colors.Colormap.set_over()` methods.

***xunits*, *yunits*: [*None* | registered units]** Override axis units by specifying an instance of a `matplotlib.units.ConversionInterface`.

tricontour-only keyword arguments:

***linewidths*: [*None* | number | tuple of numbers]** If *linewidths* is *None*, the default width in `lines.linewidth` in `matplotlibrc` is used.

If a number, all levels will be plotted with this linewidth.

If a tuple, different levels will be plotted with different linewidths in the order specified

***linestyles*: [*None* | 'solid' | 'dashed' | 'dashdot' | 'dotted']** If *linestyles* is *None*, the 'solid' is used.

linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

If contour is using a monochrome colormap and the contour level is less than 0, then the linestyle specified in `contour.negative_linestyle` in `matplotlibrc` will be used.

tricontourf-only keyword arguments:

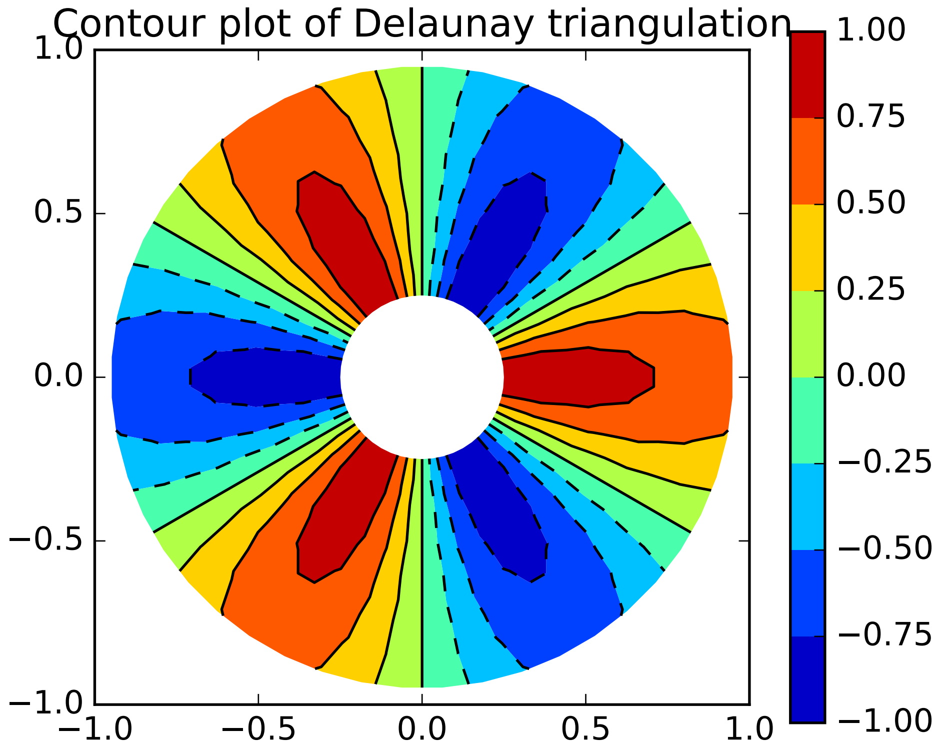
***antialiased*: [*True* | *False*]** enable antialiasing

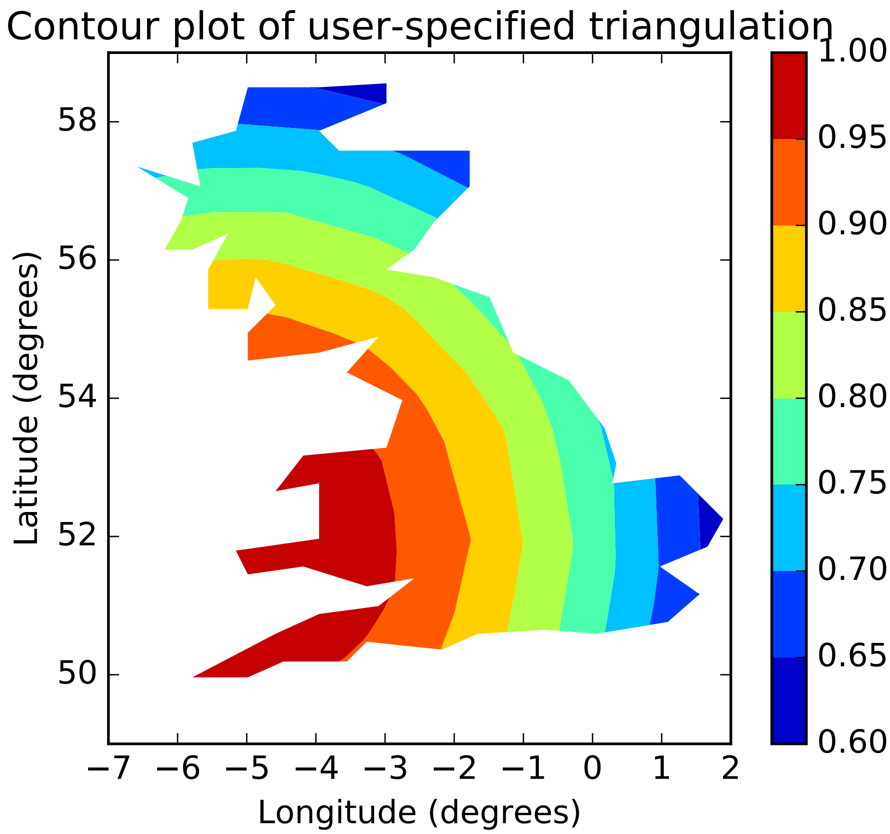
Note: tricontourf fills intervals that are closed at the top; that is, for boundaries *z1* and *z2*, the filled region is:

```
z1 < z <= z2
```

There is one exception: if the lowest boundary coincides with the minimum value of the *z* array, then that minimum value will be included in the lowest interval.

Examples:


Contour plot of Delaunay triangulation
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**tripcolor**(**args*, ***kwargs*)

Create a pseudocolor plot of an unstructured triangular grid.

The triangulation can be specified in one of two ways; either:

```
tripcolor(triangulation, ...)
```

where triangulation is a *matplotlib.tri.Triangulation* object, or

```
tripcolor(x, y, ...)
tripcolor(x, y, triangles, ...)
tripcolor(x, y, triangles=triangles, ...)
tripcolor(x, y, mask=mask, ...)
tripcolor(x, y, triangles, mask=mask, ...)
```

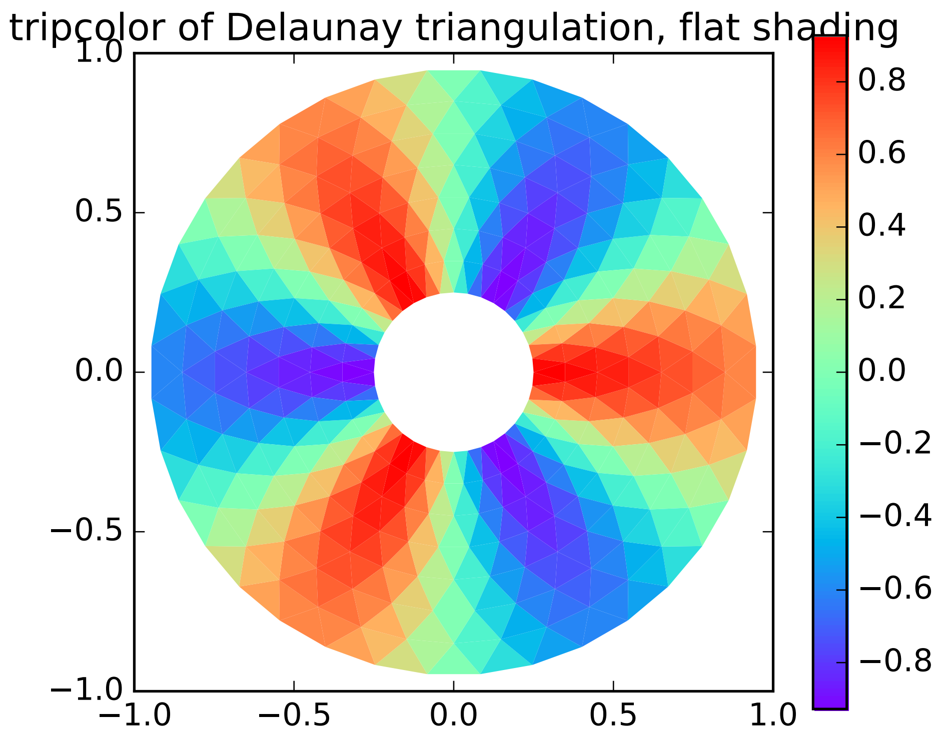
in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

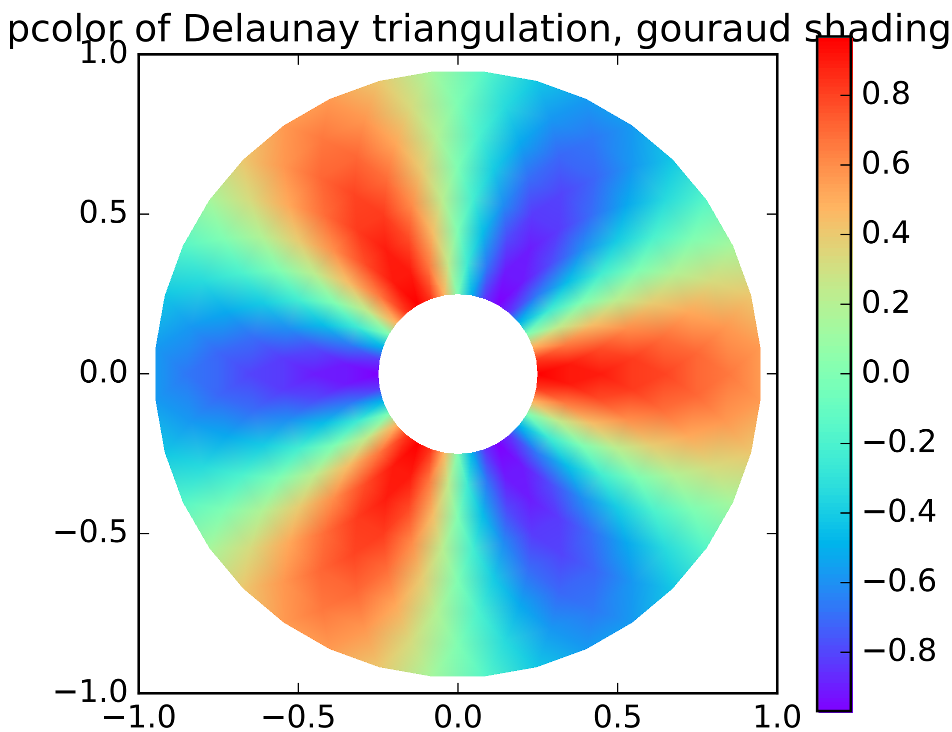
The next argument must be *C*, the array of color values, either one per point in the triangulation if color values are defined at points, or one per triangle in the triangulation if color values are defined at triangles. If there are the same number of points and triangles in the triangulation it is assumed that color values are defined at points; to force the use of color values at triangles use the kwarg *facecolors*=C instead of just *C*.

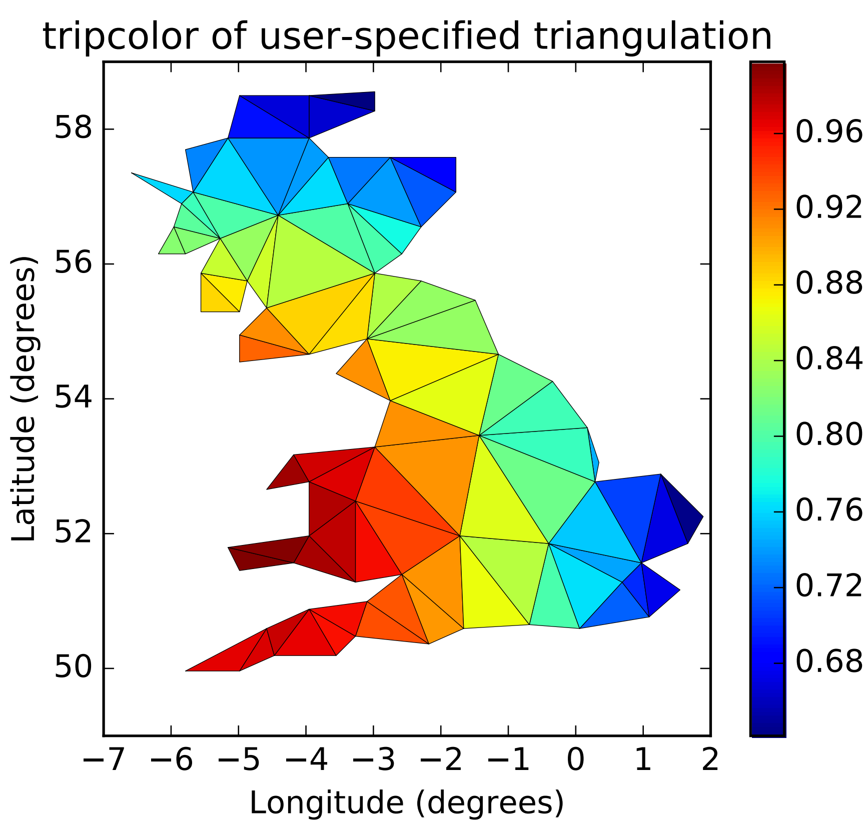
shading may be 'flat' (the default) or 'gouraud'. If *shading* is 'flat' and C values are defined at points, the color values used for each triangle are from the mean C of the triangle's three points. If *shading* is 'gouraud' then color values must be defined at points.

The remaining kwargs are the same as for `pcolor()`.

Example:




pcolor of Delaunay triangulation, gouraud shading
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8


tripcolor of user-specified triangulation
Latitude (degrees)
58
56
54
52
50
−7 −6 −5 −4 −3 −2 −1 0 1 2
Longitude (degrees)
0.96
0.92
0.88
0.84
0.80
0.76
0.72
0.68

Additional kwargs: hold = [True|False] overrides default hold state

matplotlib.pyplot.**triplot**(**args*, ***kwargs*)

Draw a unstructured triangular grid as lines and/or markers.

The triangulation to plot can be specified in one of two ways; either:

```
triplot(triangulation, ...)
```

where triangulation is a *matplotlib.tri.Triangulation* object, or

```
triplot(x, y, ...)
triplot(x, y, triangles, ...)
triplot(x, y, triangles=triangles, ...)
triplot(x, y, mask=mask, ...)
triplot(x, y, triangles, mask=mask, ...)
```

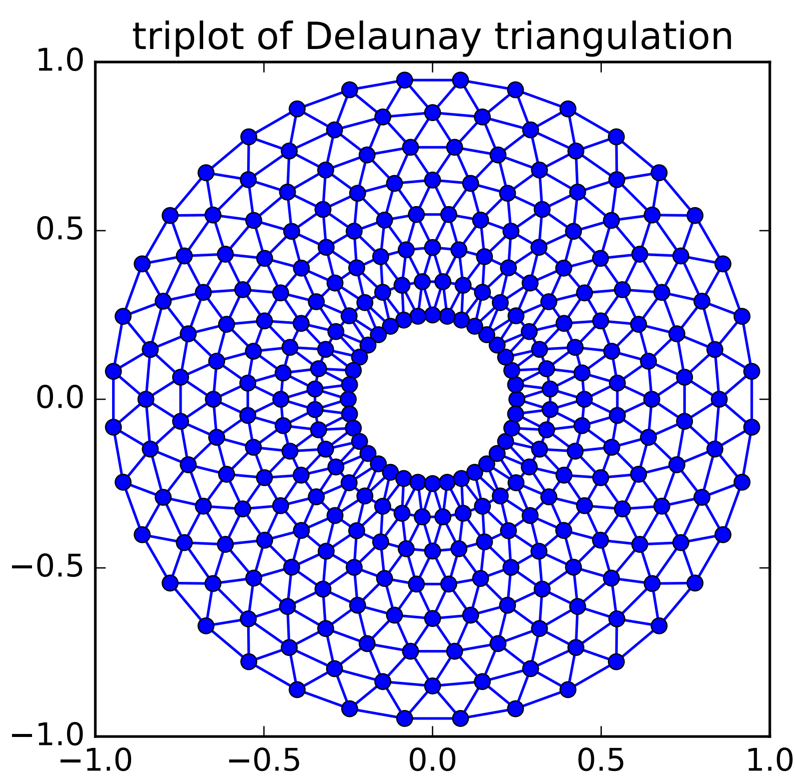
in which case a Triangulation object will be created. See *Triangulation* for a explanation of these possibilities.

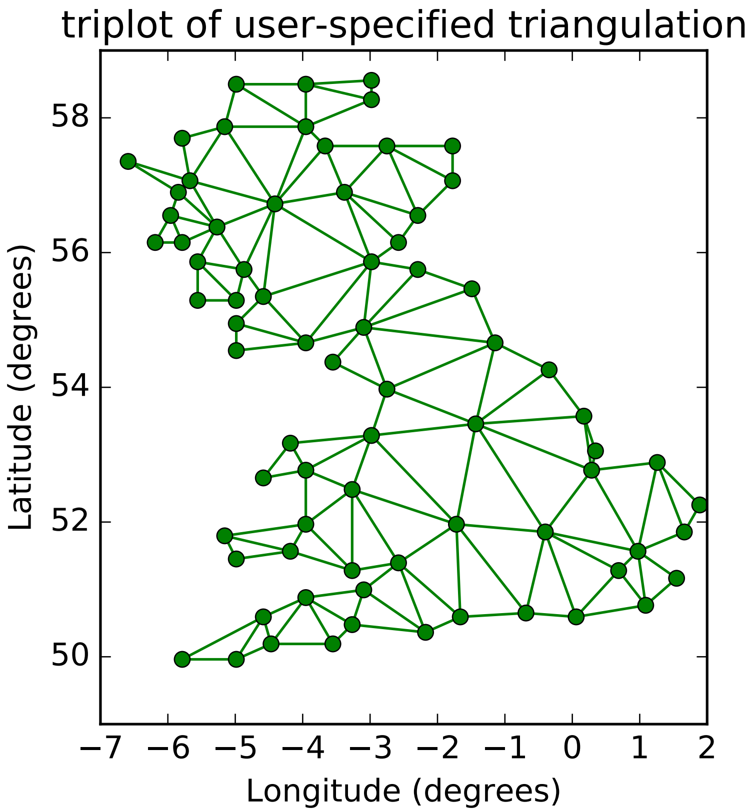
The remaining args and kwargs are the same as for *plot()*.

Return a list of 2 *Line2D* containing respectively:

- the lines plotted for triangles edges
- the markers plotted for triangles nodes

Example:





Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.`**`twinx`**(*ax=None*)

Make a second axes that shares the *x*-axis. The new axes will overlay *ax* (or the current axes if *ax* is *None*). The ticks for *ax2* will be placed on the right, and the *ax2* instance is returned.

See also:

`examples/api_examples/two_scales.py` For an example

`matplotlib.pyplot.`**`twiny`**(*ax=None*)

Make a second axes that shares the *y*-axis. The new axis will overlay *ax* (or the current axes if *ax* is *None*). The ticks for *ax2* will be placed on the top, and the *ax2* instance is returned.

`matplotlib.pyplot.`**`uninstall_repl_displayhook`**()

Uninstalls the matplotlib display hook.

`matplotlib.pyplot.`**`violinplot`**(*dataset*, *positions=None*, *vert=True*, *widths=0.5*, *showmeans=False*, *showextrema=True*, *showmedians=False*, *points=100*, *bw_method=None*, *hold=None*, *data=None*)

Make a violin plot.

Call signature:

```
violinplot(dataset, positions=None, vert=True, widths=0.5,
           showmeans=False, showextrema=True, showmedians=False,
           points=100, bw_method=None):
```

Make a violin plot for each column of *dataset* or each vector in sequence *dataset*. Each filled area extends to represent the entire data range, with optional lines at the mean, the median, the minimum, and the maximum.

Parameters dataset : Array or a sequence of vectors.

The input data.

positions [array-like, default = [1, 2, ..., n]] Sets the positions of the violins. The ticks and limits are automatically set to match the positions.

vert [bool, default = True.] If true, creates a vertical violin plot. Otherwise, creates a horizontal violin plot.

widths [array-like, default = 0.5] Either a scalar or a vector that sets the maximal width of each violin. The default is 0.5, which uses about half of the available horizontal space.

showmeans [bool, default = False] If `True`, will toggle rendering of the means.

showextrema [bool, default = True] If `True`, will toggle rendering of the extrema.

showmedians [bool, default = False] If `True`, will toggle rendering of the medians.

points [scalar, default = 100] Defines the number of points to evaluate each of the gaussian kernel density estimations at.

bw_method [str, scalar or callable, optional] The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If a scalar, this will be used directly as `kde.factor`. If a callable, it should take a `GaussianKDE` instance as its only parameter and return a scalar. If None (default), 'scott' is used.

Returns result : dict

A dictionary mapping each component of the violinplot to a list of the corresponding collection instances created. The dictionary has the following keys:

- `bodies`: A list of the *matplotlib.collections.PolyCollection* instances containing the filled area of each violin.
- `cmeans`: A *matplotlib.collections.LineCollection* instance created to identify the mean values of each of the violin's distribution.
- `cmins`: A *matplotlib.collections.LineCollection* instance created to identify the bottom of each violin's distribution.
- `cmaxes`: A *matplotlib.collections.LineCollection* instance created to identify the top of each violin's distribution.
- `cbars`: A *matplotlib.collections.LineCollection* instance created to identify the centers of each violin's distribution.
- `cmedians`: A *matplotlib.collections.LineCollection* instance created to identify the median values of each of the violin's distribution.

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'dataset'.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.viridis()`

set the default colormap to viridis and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.vlines(`*x*, *ymin*, *ymax*, *colors='k'*, *linestyles='solid'*, *label=''*, *hold=None*, *data=None*, ***kwargs*`)`

Plot vertical lines.

Plot vertical lines at each `x` from `ymin` to `ymax`.

Parameters x : scalar or 1D array_like

x-indexes where to plot the lines.

ymin, ymax : scalar or 1D array_like

Respective beginning and end of each line. If scalars are provided, all lines will have same length.

colors : array_like of colors, optional, default: 'k'

linestyles : ['solid' | 'dashed' | 'dashdot' | 'dotted'], optional

label : string, optional, default: ''

Returns lines : `LineCollection`

Other Parameters kwargs : `LineCollection` properties.

See also:

`hlines` horizontal lines

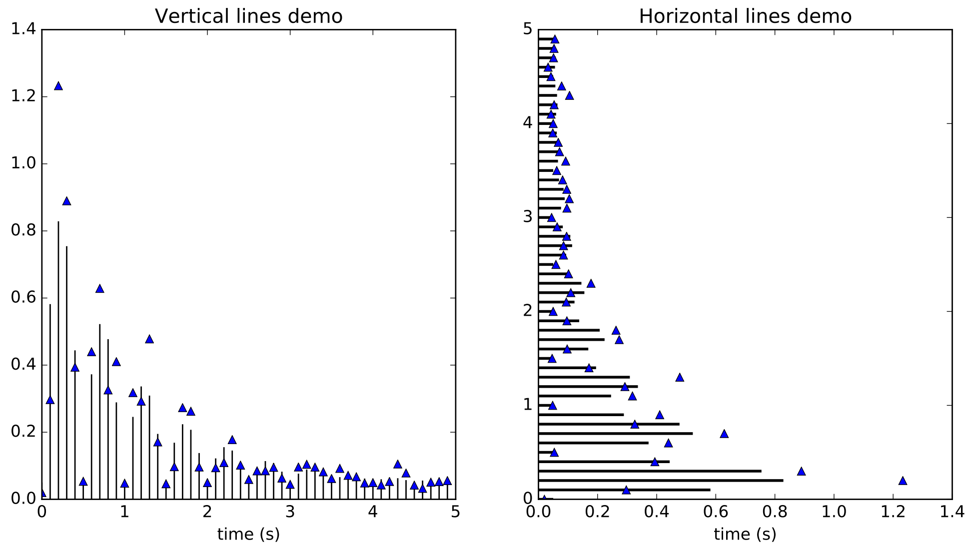
Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: 'ymax', 'x', 'colors', 'ymin'.

Additional kwargs: hold = [True|False] overrides default hold state

Examples



matplotlib.pyplot.**waitforbuttonpress**(**args*, ***kwargs*)

Call signature:

```
waitforbuttonpress(self, timeout=-1)
```

Blocking call to interact with the figure.

This will return True is a key was pressed, False if a mouse button was pressed and None if *timeout* was reached without either being pressed.

If *timeout* is negative, does not timeout.

matplotlib.pyplot.**winter**()

set the default colormap to winter and apply to current image if any. See help(colormaps) for more information

matplotlib.pyplot.**xcorr**(*x*, *y*, *normed=True*, *detrend=<function detrend_none>*, *usevlines=True*, *maxlags=10*, *hold=None*, *data=None*, ***kwargs*)

Plot the cross correlation between *x* and *y*.

Parameters x : sequence of scalars of length n

y : sequence of scalars of length n

hold : boolean, optional, default: True

detrend : callable, optional, default: `mlab.detrend_none`

x is detrended by the `detrend` callable. Default is no normalization.

normed : boolean, optional, default: True

if True, normalize the data by the autocorrelation at the 0-th lag.

usevlines : boolean, optional, default: True

if True, Axes.vlines is used to plot the vertical lines from the origin to the acorr. Otherwise, Axes.plot is used.

maxlags : integer, optional, default: 10

number of lags to show. If None, will return all 2 * len(x) - 1 lags.

Returns (lags, c, line, b) : where:

- `lags` are a length 2‘maxlags+1 lag vector.
- `c` is the 2‘maxlags+1 auto correlation vectorI
- `line` is a `Line2D` instance returned by `plot`.
- `b` is the x-axis (none, if plot is used).

Other Parameters linestyle : `Line2D` prop, optional, default: None

Only used if usevlines is False.

marker : string, optional, default: ‘o’

Notes

In addition to the above described arguments, this function can take a **data** keyword argument. If such a **data** argument is given, the following arguments are replaced by **data[<arg>]**:

- All arguments with the following names: ‘y’, ‘x’.

Additional kwargs: hold = [True|False] overrides default hold state

`matplotlib.pyplot.xkcd(scale=1, length=100, randomness=2)`

Turns on xkcd sketch-style drawing mode. This will only have effect on things drawn after this function is called.

For best results, the “Humor Sans” font should be installed: it is not included with matplotlib.

Parameters scale : float, optional

The amplitude of the wiggle perpendicular to the source line.

length : float, optional

The length of the wiggle along the line.

randomness : float, optional

The scale factor by which the length is shrunken or expanded.

Notes

This function works by a number of rcParams, so it will probably override others you have set before.

If you want the effects of this function to be temporary, it can be used as a context manager, for example:

```
with plt.xkcd():
    # This figure will be in XKCD-style
    fig1 = plt.figure()
    # ...

# This figure will be in regular style
fig2 = plt.figure()
```

`matplotlib.pyplot.xlabel(s, *args, **kwargs)`

Set the x axis label of the current axis.

Default override is:

```
override = {
    'fontsize'            : 'small',
    'verticalalignment'   : 'top',
    'horizontalalignment' : 'center'
    }
```

See also:

`text()` For information on how override and the optional args work

matplotlib.pyplot.**xlim**(**args*, ***kwargs*)
Get or set the *x* limits of the current axes.

```
xmin, xmax = xlim()   # return the current xlim
xlim( (xmin, xmax) )  # set the xlim to xmin, xmax
xlim( xmin, xmax )    # set the xlim to xmin, xmax
```

If you do not specify args, you can pass the xmin and xmax as kwargs, e.g.:

```
xlim(xmax=3) # adjust the max leaving min unchanged
xlim(xmin=1) # adjust the min leaving max unchanged
```

Setting limits turns autoscaling off for the x-axis.

The new axis limits are returned as a length 2 tuple.

matplotlib.pyplot.**xscale**(**args*, ***kwargs*)
Set the scaling of the *x*-axis.

call signature:

```
xscale(scale, **kwargs)
```

The available scales are: 'linear' | 'log' | 'logit' | 'symlog'

Different keywords may be accepted, depending on the scale:

'linear'

'log'

basex*/*basey: The base of the logarithm

nonposx*/*nonposy: **['mask' | 'clip']** non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number

subsx*/*subsy: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

'logit'

nonpos: **['mask' | 'clip']** values beyond]0, 1[can be masked as invalid, or clipped to a number very close to 0 or 1

'symlog'

basex*/*basey: The base of the logarithm

linthreshx*/*linthreshy: The range (-*x*, *x*) within which the plot is linear (to avoid having the plot go to infinity around zero).

subsx*/*subsy: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

linscalex*/*linscaley: This allows the linear range (-*linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

`matplotlib.pyplot.xticks(*args, **kwargs)`

Get or set the *x*-limits of the current tick locations and labels.

```
# return locs, labels where locs is an array of tick locations and
# labels is an array of tick labels.
locs, labels = xticks()

# set the locations of the xticks
xticks( arange(6) )

# set the locations and labels of the xticks
xticks( arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue') )
```

The keyword args, if any, are `Text` properties. For example, to rotate long labels:

```
xticks( arange(12), calendar.month_name[1:13], rotation=17 )
```

`matplotlib.pyplot.ylabel(s, *args, **kwargs)`

Set the *y* axis label of the current axis.

Defaults override is:

```
override = {
   'fontsize'            : 'small',
   'verticalalignment'   : 'center',
   'horizontalalignment' : 'right',
   'rotation'='vertical' : }
```

See also:

`text()` For information on how override and the optional args work.

`matplotlib.pyplot.ylim(*args, **kwargs)`

Get or set the *y*-limits of the current axes.

```
ymin, ymax = ylim()   # return the current ylim
ylim( (ymin, ymax) )  # set the ylim to ymin, ymax
ylim( ymin, ymax )    # set the ylim to ymin, ymax
```

If you do not specify args, you can pass the *ymin* and *ymax* as kwargs, e.g.:

```
ylim(ymax=3) # adjust the max leaving min unchanged
ylim(ymin=1) # adjust the min leaving max unchanged
```

Setting limits turns autoscaling off for the y-axis.

The new axis limits are returned as a length 2 tuple.

`matplotlib.pyplot.yscale`(**args*, ***kwargs*)

Set the scaling of the *y*-axis.

call signature:

```
yscale(scale, **kwargs)
```

The available scales are: 'linear' | 'log' | 'logit' | 'symlog'

Different keywords may be accepted, depending on the scale:

'linear'

'log'

basex/basey: The base of the logarithm

nonposx/nonposy: **['mask' | 'clip']** non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number

subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

'logit'

nonpos: **['mask' | 'clip']** values beyond]0, 1[can be masked as invalid, or clipped to a number very close to 0 or 1

'symlog'

basex/basey: The base of the logarithm

linthreshx/linthreshy: The range (-*x*, *x*) within which the plot is linear (to avoid having the plot go to infinity around zero).

subsx/subsy: Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

linscalex/linscaley: This allows the linear range (-*linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

matplotlib.pyplot.**yticks**(**args*, ***kwargs*)

Get or set the *y*-limits of the current tick locations and labels.

```
# return locs, labels where locs is an array of tick locations and
# labels is an array of tick labels.
locs, labels = yticks()

# set the locations of the yticks
yticks( arange(6) )

# set the locations and labels of the yticks
yticks( arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue') )
```

The keyword args, if any, are `Text` properties. For example, to rotate long labels:

```
yticks( arange(12), calendar.month_name[1:13], rotation=45 )
```

CHAPTER

SIXTYNINE

SANKEY

69.1 matplotlib.sankey

Module for creating Sankey diagrams using matplotlib

class `matplotlib.sankey.`**`Sankey`**(*ax=None*, *scale=1.0*, *unit=''*, *format='%G'*, *gap=0.25*, *radius=0.1*, *shoulder=0.03*, *offset=0.15*, *head_angle=100*, *margin=0.4*, *tolerance=1e-06*, ***kwargs*)

Bases: `object`

Sankey diagram in matplotlib

Sankey diagrams are a specific type of flow diagram, in which the width of the arrows is shown proportionally to the flow quantity. They are typically used to visualize energy or material or cost transfers between processes. Wikipedia (6/1/2011)

Create a new Sankey instance.

Optional keyword arguments:

Field	Description
ax	axes onto which the data should be plotted If *ax* isn't provided, new axes will be created.
scale	scaling factor for the flows *scale* sizes the width of the paths in order to maintain proper layout. The same scale is applied to all subdiagrams. The value should be chosen such that the product of the scale and the sum of the inputs is approximately 1.0 (and the product of the scale and the sum of the outputs is approximately -1.0).
unit	string representing the physical unit associated with the flow quantities If *unit* is None, then none of the quantities are labeled.
format	a Python number formatting string to be used in labeling the flow as a quantity (i.e., a number times a unit, where the unit is given)
gap	space between paths that break in/break away to/from the top or bottom
radius	inner radius of the vertical paths
shoulder	size of the shoulders of output arrowS
offset	text offset (from the dip or tip of the arrow)
head_angle	angle of the arrow heads (and negative of the angle of the tails) [deg]
margin	minimum space between Sankey outlines and the edge of the plot area
tolerance	acceptable maximum of the magnitude of the sum of flows The magnitude of the sum of connected flows cannot be greater than *tolerance*.

The optional arguments listed above are applied to all subdiagrams so that there is consistent alignment and formatting.

If `Sankey` is instantiated with any keyword arguments other than those explicitly listed above (`**kwargs`), they will be passed to `add()`, which will create the first subdiagram.

In order to draw a complex Sankey diagram, create an instance of `Sankey` by calling it without any kwargs:

```
sankey = Sankey()
```

Then add simple Sankey sub-diagrams:

```
sankey.add() # 1
sankey.add() # 2
#...
sankey.add() # n
```

Finally, create the full diagram:

```
sankey.finish()
```

Or, instead, simply daisy-chain those calls:

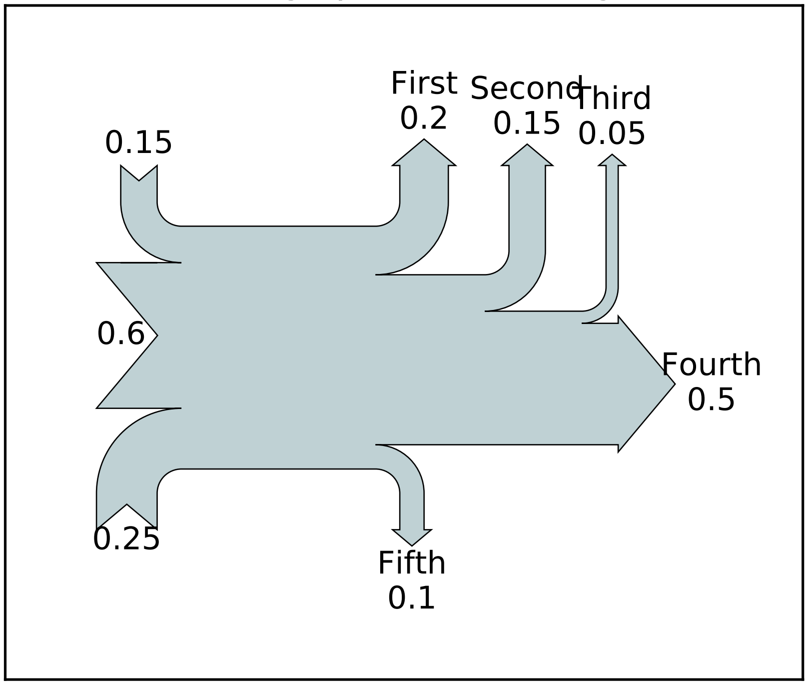
```
Sankey().add().add...  .add().finish()
```

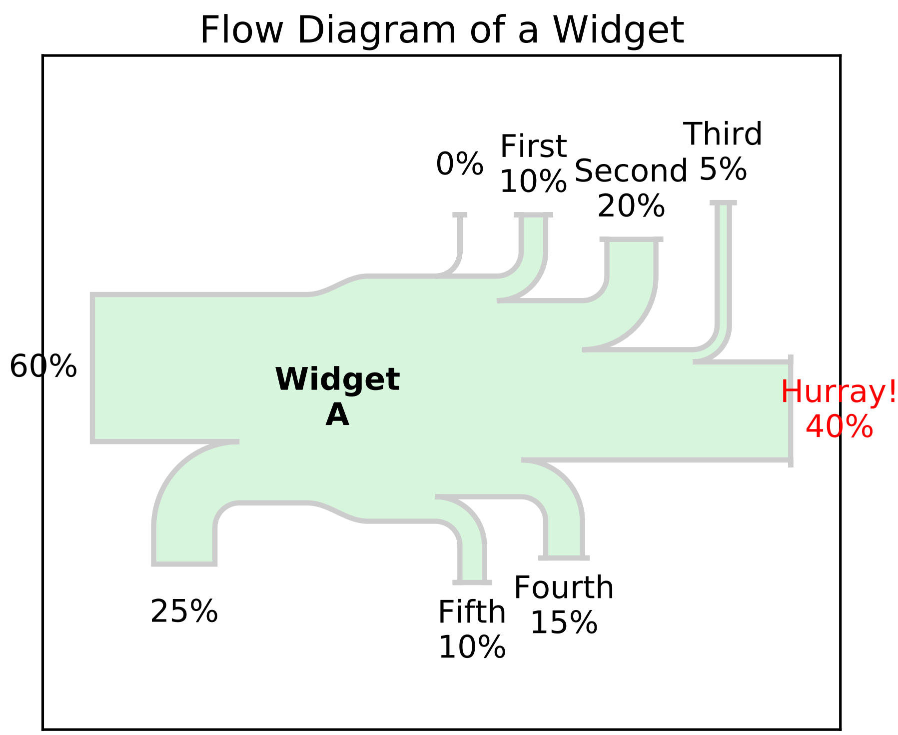
See also:

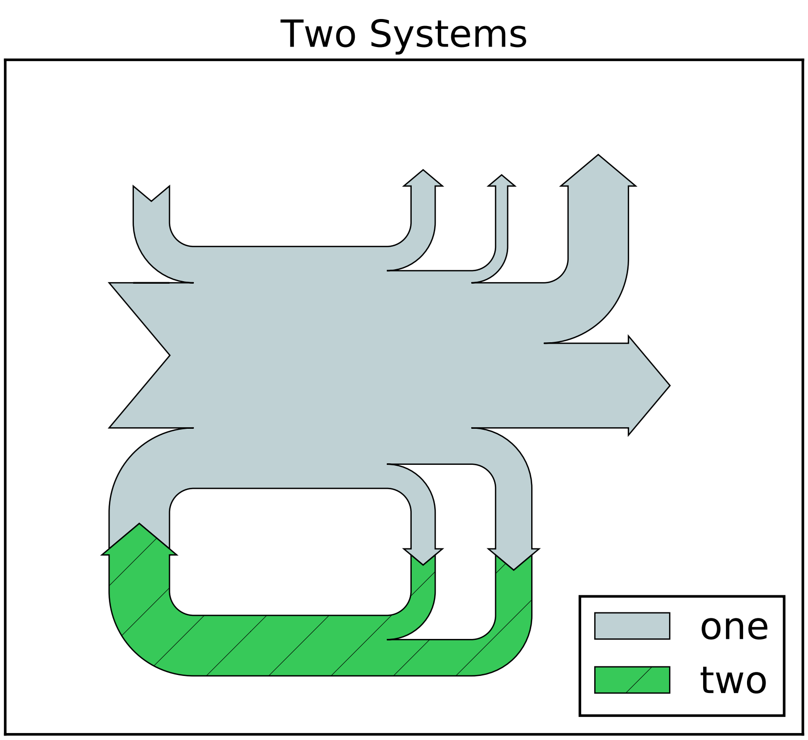
add() finish()

Examples:

The default settings produce a diagram like this.




Flow Diagram of a Widget
0%
First
10%
Second
20%
Third
5%
60%
Widget
A
Hurray!
40%
25%
Fifth
10%
Fourth
15%


Two Systems
one
two

add(*patchlabel=''*, *flows=None*, *orientations=None*, *labels=''*, *trunklength=1.0*, *pathlengths=0.25*, *prior=None*, *connect=(0, 0)*, *rotation=0*, ***kwargs*)

Add a simple Sankey diagram with flows at the same hierarchical level.

Return value is the instance of `Sankey`.

Optional keyword arguments:

Keyword	Description
patchlabel	label to be placed at the center of the diagram Note: *label* (not *patchlabel*) will be passed to the patch through `**kwargs` and can be used to create an entry in the legend.
flows	array of flow values By convention, inputs are positive and outputs are negative.
orientations	list of orientations of the paths Valid values are 1 (from/to the top), 0 (from/to the left or right), or -1 (from/to the bottom). If *orientations* == 0, inputs will break in from the left and outputs will break away to the right.
labels	list of specifications of the labels for the flows Each value may be *None* (no labels), '' (just label the quantities), or a labeling string. If a single value is provided, it will be applied to all flows. If an entry is a non-empty string, then the quantity for the corresponding flow will be shown below the string. However, if the *unit* of the main diagram is None, then quantities are never shown, regardless of the value of this argument.
trunklength	length between the bases of the input and output groups
pathlengths	list of lengths of the arrows before break-in or after break-away If a single value is given, then it will be applied to the first (inside) paths on the top and bottom, and the length of all other arrows will be justified accordingly. The *pathlengths* are not applied to the horizontal inputs and outputs.
prior	index of the prior diagram to which this diagram should be connected
connect	a (prior, this) tuple indexing the flow of the prior diagram and the flow of this diagram which should be connected If this is the first diagram or *prior* is *None*, *connect* will be ignored.
rotation	angle of rotation of the diagram [deg] *rotation* is ignored if this diagram is connected to an existing one (using *prior* and *connect*). The interpretation of the *orientations* argument will be rotated accordingly (e.g., if *rotation* == 90, an *orientations* entry of 1 means to/from the left).

Valid kwargs are `matplotlib.patches.PathPatch()` arguments:

Property	Description
`agg_filter`	unknown
`alpha`	float or None

Continued on

Table 69.1 – continued from previous page

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>animated</td><td>[True | False]</td></tr>
<tr><td>antialiased or aa</td><td>[True | False] or None for default</td></tr>
<tr><td>axes</td><td>an Axes instance</td></tr>
<tr><td>capstyle</td><td>[’butt’ | ‘round’ | ‘projecting’]</td></tr>
<tr><td>clip_box</td><td>a matplotlib.transforms.Bbox instance</td></tr>
<tr><td>clip_on</td><td>[True | False]</td></tr>
<tr><td>clip_path</td><td>[(Path, Transform) | Patch | None]</td></tr>
<tr><td>color</td><td>matplotlib color spec</td></tr>
<tr><td>contains</td><td>a callable function</td></tr>
<tr><td>edgecolor or ec</td><td>mpl color spec, or None for default, or ‘none’ for no color</td></tr>
<tr><td>facecolor or fc</td><td>mpl color spec, or None for default, or ‘none’ for no color</td></tr>
<tr><td>figure</td><td>a matplotlib.figure.Figure instance</td></tr>
<tr><td>fill</td><td>[True | False]</td></tr>
<tr><td>gid</td><td>an id string</td></tr>
<tr><td>hatch</td><td>[’/’ | ‘\’ | ‘|’ | ‘-‘ | ‘+’ | ‘x’ | ‘o’ | ‘O’ | ‘.’ | ‘*’]</td></tr>
<tr><td>joinstyle</td><td>[’miter’ | ‘round’ | ‘bevel’]</td></tr>
<tr><td>label</td><td>string or anything printable with ‘%s’ conversion.</td></tr>
<tr><td>linestyle or ls</td><td>[’solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None'</td></tr>
<tr><td>linewidth or lw</td><td>float or None for default</td></tr>
<tr><td>path_effects</td><td>unknown</td></tr>
<tr><td>picker</td><td>[None|float|boolean|callable]</td></tr>
<tr><td>rasterized</td><td>[True | False | None]</td></tr>
<tr><td>sketch_params</td><td>unknown</td></tr>
<tr><td>snap</td><td>unknown</td></tr>
<tr><td>transform</td><td>Transform instance</td></tr>
<tr><td>url</td><td>a url string</td></tr>
<tr><td>visible</td><td>[True | False]</td></tr>
<tr><td>zorder</td><td>any number</td></tr>
</table>

As examples, `fill=False` and `label='A legend entry'`. By default, `facecolor='#bfd1d4'` (light blue) and `linewidth=0.5`.

The indexing parameters (*prior* and *connect*) are zero-based.

The flows are placed along the top of the diagram from the inside out in order of their index within the *flows* list or array. They are placed along the sides of the diagram from the top down and along the bottom from the outside in.

If the sum of the inputs and outputs is nonzero, the discrepancy will appear as a cubic Bezier curve along the top and bottom edges of the trunk.

See also:

finish()

finish()

Adjust the axes and return a list of information about the Sankey subdiagram(s).

Return value is a list of subdiagrams represented with the following fields:

Field	Description
patch	Sankey outline (an instance of `PathPatch`)
flows	values of the flows (positive for input, negative for output)
angles	list of angles of the arrows [deg/90] For example, if the diagram has not been rotated, an input to the top side will have an angle of 3 (DOWN), and an output from the top side will have an angle of 1 (UP). If a flow has been skipped (because its magnitude is less than *tolerance*), then its angle will be *None*.
tips	array in which each row is an [x, y] pair indicating the positions of the tips (or "dips") of the flow paths If the magnitude of a flow is less the *tolerance* for the instance of `Sankey`, the flow is skipped and its tip will be at the center of the diagram.
text	`Text` instance for the label of the diagram
texts	list of `Text` instances for the labels of flows

See also:

`add()`

CHAPTER

SEVENTY

SCALE

70.1 matplotlib.scale

class matplotlib.scale.**InvertedLog10Transform**(*shorthand_name=None*)

Bases: *matplotlib.transforms.Transform*

Creates a new TransformNode.

shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

base = 10.0

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class matplotlib.scale.**InvertedLog2Transform**(*shorthand_name=None*)

Bases: *matplotlib.transforms.Transform*

Creates a new TransformNode.

shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

base = 2.0

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class matplotlib.scale.**InvertedLogTransform**(*base*)
Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class matplotlib.scale.**InvertedNaturalLogTransform**(*shorthand_name=None*)
Bases: *matplotlib.transforms.Transform*

Creates a new TransformNode.

shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of str(transform) when DEBUG=True.

base = 2.718281828459045

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class matplotlib.scale.**InvertedSymmetricalLogTransform**(*base*, *linthresh*, *linscale*)
Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class matplotlib.scale.**LinearScale**(*axis*, ***kwargs*)
Bases: *matplotlib.scale.ScaleBase*

The default linear scale.

get_transform()
The transform for linear scaling is just the *IdentityTransform*.

name = 'linear'

set_default_locators_and_formatters(*axis*)
Set the locators and formatters to reasonable defaults for linear scaling.

class matplotlib.scale.**Log10Transform**(*nonpos*)
Bases: *matplotlib.scale.LogTransformBase*

base = 10.0

inverted()

transform_non_affine(*a*)

class matplotlib.scale.**Log2Transform**(*nonpos*)
Bases: *matplotlib.scale.LogTransformBase*

base = 2.0

inverted()

transform_non_affine(*a*)

class matplotlib.scale.**LogScale**(*axis*, ***kwargs*)
Bases: `matplotlib.scale.ScaleBase`

A standard logarithmic scale. Care is taken so non-positive values are not plotted.

For computational efficiency (to push as much as possible to Numpy C code in the common cases), this scale provides different transforms depending on the base of the logarithm:

- base 10 (`Log10Transform`)
- base 2 (`Log2Transform`)
- base e (`NaturalLogTransform`)
- arbitrary base (`LogTransform`)

***basex/basey*: The base of the logarithm**

***nonposx/nonposy*: ['mask' | 'clip']** non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

class **InvertedLog10Transform**(*shorthand_name=None*)
Bases: `matplotlib.transforms.Transform`

Creates a new `TransformNode`.

shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

base = 10.0

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class LogScale.**InvertedLog2Transform**(*shorthand_name=None*)

Bases: `matplotlib.transforms.Transform`

Creates a new `TransformNode`.

shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

base = 2.0

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class LogScale.**InvertedLogTransform**(*base*)

Bases: `matplotlib.transforms.Transform`

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class LogScale.**InvertedNaturalLogTransform**(*shorthand_name=None*)

Bases: `matplotlib.transforms.Transform`

Creates a new `TransformNode`.

shorthand_name - a string representing the "name" of this transform. The name carries no significance other than to improve the readability of `str(transform)` when DEBUG=True.

base = 2.718281828459045

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class LogScale.**Log10Transform**(*nonpos*)

Bases: `matplotlib.scale.LogTransformBase`

base = 10.0

inverted()

transform_non_affine(*a*)

class LogScale.**Log2Transform**(*nonpos*)

Bases: `matplotlib.scale.LogTransformBase`

base = 2.0

inverted()

transform_non_affine(*a*)

class LogScale.**LogTransform**(*base*, *nonpos*)

Bases: `matplotlib.transforms.Transform`

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class LogScale.**LogTransformBase**(*nonpos*)
Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

is_separable = True

output_dims = 1

class LogScale.**NaturalLogTransform**(*nonpos*)
Bases: *matplotlib.scale.LogTransformBase*

base = 2.718281828459045

inverted()

transform_non_affine(*a*)

LogScale.**get_transform**()
Return a *Transform* instance appropriate for the given logarithm base.

LogScale.**limit_range_for_scale**(*vmin*, *vmax*, *minpos*)
Limit the domain to positive values.

LogScale.**name = 'log'**

LogScale.**set_default_locators_and_formatters**(*axis*)
Set the locators and formatters to specialized versions for log scaling.

class matplotlib.scale.**LogTransform**(*base*, *nonpos*)
Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class matplotlib.scale.**LogTransformBase**(*nonpos*)
Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

is_separable = True

output_dims = 1

class matplotlib.scale.**LogisticTransform**(*nonpos='mask'*)
Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)
logistic transform (base 10)

class matplotlib.scale.**LogitScale**(*axis*, *nonpos='mask'*)
Bases: *matplotlib.scale.ScaleBase*

Logit scale for data between zero and one, both excluded.

This scale is similar to a log scale close to zero and to one, and almost linear around 0.5. It maps the interval]0, 1[onto]-infty, +infty[.

***nonpos*: ['mask' | 'clip']** values beyond]0, 1[can be masked as invalid, or clipped to a number very close to 0 or 1

get_transform()
Return a *LogitTransform* instance.

limit_range_for_scale(*vmin*, *vmax*, *minpos*)
Limit the domain to values between 0 and 1 (excluded).

name = 'logit'

set_default_locators_and_formatters(*axis*)

class matplotlib.scale.**LogitTransform**(*nonpos*)
Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)
logit transform (base 10), masked or clipped

class matplotlib.scale.**NaturalLogTransform**(*nonpos*)
Bases: *matplotlib.scale.LogTransformBase*

base = 2.718281828459045

inverted()

transform_non_affine(*a*)

class matplotlib.scale.**ScaleBase**
Bases: object

The base class for all scales.

Scales are separable transformations, working on a single dimension.

Any subclasses will want to override:

- name
- *get_transform()*
- *set_default_locators_and_formatters()*

And optionally:

- *limit_range_for_scale()*

get_transform()

Return the *Transform* object associated with this scale.

limit_range_for_scale(*vmin*, *vmax*, *minpos*)

Returns the range *vmin*, *vmax*, possibly limited to the domain supported by this scale.

***minpos* should be the minimum positive value in the data.** This is used by log scales to determine a minimum value.

set_default_locators_and_formatters(*axis*)

Set the *Locator* and *Formatter* objects on the given axis to match this scale.

class matplotlib.scale.**SymmetricalLogScale**(*axis*, ***kwargs*)

Bases: *matplotlib.scale.ScaleBase*

The symmetrical logarithmic scale is logarithmic in both the positive and negative directions from the origin.

Since the values close to zero tend toward infinity, there is a need to have a range around zero that is linear. The parameter *linthresh* allows the user to specify the size of this range (-*linthresh*, *linthresh*).

***basex/basey*:** The base of the logarithm

***linthreshx/linthreshy*:** The range (-*x*, *x*) within which the plot is linear (to avoid having the plot go to infinity around zero).

***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

***linscalex/linscaley*:** This allows the linear range (-*linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

class InvertedSymmetricalLogTransform(*base*, *linthresh*, *linscale*)

Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

class SymmetricalLogScale.**SymmetricalLogTransform**(*base*, *linthresh*, *linscale*)
Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

SymmetricalLogScale.**get_transform**()
Return a *SymmetricalLogTransform* instance.

SymmetricalLogScale.**name = 'symlog'**

SymmetricalLogScale.**set_default_locators_and_formatters**(*axis*)
Set the locators and formatters to specialized versions for symmetrical log scaling.

class matplotlib.scale.**SymmetricalLogTransform**(*base*, *linthresh*, *linscale*)
Bases: *matplotlib.transforms.Transform*

has_inverse = True

input_dims = 1

inverted()

is_separable = True

output_dims = 1

transform_non_affine(*a*)

matplotlib.scale.**get_scale_docs**()
Helper function for generating docstrings related to scales.

`matplotlib.scale.get_scale_names()`

`matplotlib.scale.register_scale(scale_class)`

Register a new kind of scale.

scale_class must be a subclass of `ScaleBase`.

`matplotlib.scale.scale_factory(scale, axis, **kwargs)`

Return a scale class by name.

ACCEPTS: [linear | log | logit | symlog]

CHAPTER

SEVENTYONE

SPINES

71.1 matplotlib.spines

class matplotlib.spines.**Spine**(*axes*, *spine_type*, *path*, ***kwargs*)

Bases: matplotlib.patches.Patch

an axis spine – the line noting the data area boundaries

Spines are the lines connecting the axis tick marks and noting the boundaries of the data area. They can be placed at arbitrary positions. See function:set_position for more information.

The default position is ('outward',0).

Spines are subclasses of class:Patch, and inherit much of their behavior.

Spines draw a line or a circle, depending if function:set_patch_line or function:set_patch_circle has been called. Line-like is the default.

- *axes* : the Axes instance containing the spine
- *spine_type* : a string specifying the spine type
- *path* : the path instance used to draw the spine

Valid kwargs are:

Property	Description
agg_filter	unknown
alpha	float or None
animated	[True | False]
antialiased or aa	[True | False] or None for default
axes	an Axes instance
capstyle	['butt' | 'round' | 'projecting']
clip_box	a matplotlib.transforms.Bbox instance
clip_on	[True | False]
clip_path	[(Path, Transform) | Patch | None]
color	matplotlib color spec
contains	a callable function
edgecolor or ec	mpl color spec, or None for default, or 'none' for no color
facecolor or fc	mpl color spec, or None for default, or 'none' for no color
figure	a matplotlib.figure.Figure instance

Continued on

Table 71.1 – continued from previous page

Property	Description
`fill`	[True | False]
`gid`	an id string
`hatch`	['/' | '\' | '|' | '-' | '+' | 'x' | 'o' | 'O' | '.' | '*']
`joinstyle`	['miter' | 'round' | 'bevel']
`label`	string or anything printable with '%s' conversion.
`linestyle` or ls	['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | `'-'` | `'--'` | `'-.'` | `':'` | `'None'`
`linewidth` or lw	float or None for default
`path_effects`	unknown
`picker`	[None|float|boolean|callable]
`rasterized`	[True | False | None]
`sketch_params`	unknown
`snap`	unknown
`transform`	`Transform` instance
`url`	a url string
`visible`	[True | False]
`zorder`	any number

classmethod **`circular_spine`**(*axes*, *center*, *radius*, ***kwargs*)
(staticmethod) Returns a circular `Spine`.

`cla`()
Clear the current spine

`draw`(*artist*, *renderer*, **args*, ***kwargs*)

`get_bounds`()
Get the bounds of the spine.

`get_patch_transform`()

`get_path`()

`get_position`()
get the spine position

`get_smart_bounds`()
get whether the spine has smart bounds

`get_spine_transform`()
get the spine transform

`is_frame_like`()
return True if directly on axes frame

This is useful for determining if a spine is the edge of an old style MPL plot. If so, this function will return True.

classmethod linear_spine(*axes*, *spine_type*, ***kwargs*)
(staticmethod) Returns a linear Spine.

register_axis(*axis*)
register an axis

An axis should be registered with its corresponding spine from the Axes instance. This allows the spine to clear any axis properties when needed.

set_bounds(*low*, *high*)
Set the bounds of the spine.

set_color(*c*)
Set the edgecolor.

ACCEPTS: matplotlib color arg or sequence of rgba tuples

See also:

set_facecolor(), set_edgecolor() For setting the edge or face color individually.

set_patch_circle(*center*, *radius*)
set the spine to be circular

set_patch_line()
set the spine to be linear

set_position(*position*)
set the position of the spine

Spine position is specified by a 2 tuple of (position type, amount). The position types are:

- 'outward' : place the spine out from the data area by the specified number of points. (Negative values specify placing the spine inward.)
- 'axes' : place the spine at the specified Axes coordinate (from 0.0-1.0).
- 'data' : place the spine at the specified data coordinate.

Additionally, shorthand notations define a special positions:

- 'center' -> ('axes',0.5)
- 'zero' -> ('data', 0.0)

set_smart_bounds(*value*)
set the spine and associated axis to have smart bounds

CHAPTER

SEVENTYTWO

STYLE

72.1 matplotlib.style

`matplotlib.style.context(style, after_reset=False)`

Context manager for using style settings temporarily.

Parameters style : str, dict, or list

A style specification. Valid options are:

str	The name of a style or a path/URL to a style file. For a list of available style names, see `style.available`.
dict	Dictionary with valid key/value pairs for *matplotlib.rcParams*.
list	A list of style specifiers (str or dict) applied from first to last in the list.

after_reset : bool

If True, apply style after resetting settings to their defaults; otherwise, apply style on top of the current settings.

`matplotlib.style.reload_library()`

Reload style library.

`matplotlib.style.use(style)`

Use matplotlib style settings from a style specification.

The style name of 'default' is reserved for reverting back to the default style settings.

Parameters style : str, dict, or list

A style specification. Valid options are:

str	The name of a style or a path/URL to a style file. For a list of available style names, see `style.available`.
dict	Dictionary with valid key/value pairs for *matplotlib.rcParams*.
list	A list of style specifiers (str or dict) applied from first to last in the list.

`matplotlib.style.library`

Dictionary of available styles

`matplotlib.style.available`

List of available styles

CHAPTER

SEVENTYTHREE

TEXT

73.1 matplotlib.text

Classes for including text in a figure.

class `matplotlib.text.`**`Annotation`**(*s*, *xy*, *xytext=None*, *xycoords='data'*, *textcoords=None*, *arrowprops=None*, *annotation_clip=None*, ***kwargs*)

Bases: *`matplotlib.text.Text`*, `matplotlib.text._AnnotationBase`

Annotate the point `xy` with text `s`.

Additional kwargs are passed to *`Text`*.

Parameters s : str

The text of the annotation

xy : iterable

Length 2 sequence specifying the *(x,y)* point to annotate

xytext : iterable, optional

Length 2 sequence specifying the *(x,y)* to place the text at. If None, defaults to `xy`.

xycoords : str, Artist, Transform, callable or tuple, optional

The coordinate system that `xy` is given in.

For a `str` the allowed values are:

Property	Description
'figure points'	points from the lower left of the figure
'figure pixels'	pixels from the lower left of the figure
'figure fraction'	fraction of figure from lower left
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	fraction of axes from lower left
'data'	use the coordinate system of the object being annotated (default)
'polar'	*(theta,r)* if not native 'data' coordinates

If a *Artist* object is passed in the units are fraction if it's bounding box.

If a *Transform* object is passed in use that to transform `xy` to screen coordinates

If a callable it must take a *RendererBase* object as input and return a *Transform* or *Bbox* object

If a `tuple` must be length 2 tuple of str, `Artist`, `Transform` or callable objects. The first transform is used for the x coordinate and the second for y.

See *Annotating Axes* for more details.

Defaults to `'data'`

textcoords : str, `Artist`, `Transform`, callable or tuple, optional

The coordinate system that `xytext` is given, which may be different than the coordinate system used for `xy`.

All `xycoords` values are valid as well as the following strings:

Property	Description
'offset points'	offset (in points) from the xy value
'offset pixels'	offset (in pixels) from the xy value

defaults to the input of `xycoords`

arrowprops : dict, optional

If not None, properties used to draw a *FancyArrowPatch* arrow between `xy` and `xytext`.

If `arrowprops` does not contain the key `'arrowstyle'` the allowed keys are:

Key	Description
width	the width of the arrow in points
headwidth	the width of the base of the arrow head in points
headlength	the length of the arrow head in points
shrink	fraction of total length to 'shrink' from both ends
?	any key to *matplotlib.patches.FancyArrowPatch*

If the `arrowprops` contains the key `'arrowstyle'` the above keys are forbidden. The allowed values of `'arrowstyle'` are:

Name	Attrs
`'-'`	None
`'->'`	head_length=0.4,head_width=0.2
`'-['`	widthB=1.0,lengthB=0.2,angleB=None
`'|-|'`	widthA=1.0,widthB=1.0
`'-|>'`	head_length=0.4,head_width=0.2
`'<-'`	head_length=0.4,head_width=0.2
`'<->'`	head_length=0.4,head_width=0.2
`'<|-'`	head_length=0.4,head_width=0.2
`'<|-|>'`	head_length=0.4,head_width=0.2
`'fancy'`	head_length=0.4,head_width=0.4,tail_width=0.4
`'simple'`	head_length=0.5,head_width=0.5,tail_width=0.2
`'wedge'`	tail_width=0.3,shrink_factor=0.5

Valid keys for *FancyArrowPatch* are:

Key	Description
arrowstyle	the arrow style
connectionstyle	the connection style
relpos	default is (0.5, 0.5)
patchA	default is bounding box of the text
patchB	default is None
shrinkA	default is 2 points
shrinkB	default is 2 points
mutation_scale	default is text size (in points)
muta-tion_aspect	default is 1.
?	any key for *matplotlib.patches.PathPatch*

Defaults to None

annotation_clip : bool, optional

Controls the visibility of the annotation when it goes outside the axes area.

If `True`, the annotation will only be drawn when the `xy` is inside the axes. If `False`, the annotation will always be drawn regardless of its position.

The default is `None`, which behave as `True` only if *xycoords* is "data".

Returns Annotation

anncoords

contains(*event*)

draw(*artist*, *renderer*, **args*, ***kwargs*)

Draw the *Annotation* object to the given *renderer*.

get_window_extent(*renderer=None*)

Return a *Bbox* object bounding the text and arrow annotation, in display units.

renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of *draw()*, so you must use this kwarg if you want to call *get_window_extent()* prior to the first *draw()*. For getting web page regions, it is simpler to call the method after saving the figure. The *dpi* used defaults to self.figure.dpi; the renderer dpi is irrelevant.

set_figure(*fig*)

update_positions(*renderer*)
"Update the pixel positions of the annotated point and the text.

xyann

class matplotlib.text.**OffsetFrom**(*artist*, *ref_coord*, *unit='points'*)
Bases: object

Callable helper class for working with *Annotation*

Parameters artist : Artist, BboxBase, or Transform
The object to compute the offset from.

ref_coord : length 2 sequence
If artist is an Artist or BboxBase, this values is the location to of the offset origin in fractions of the artist bounding box.

If artist is a transform, the offset origin is the transform applied to this value.

unit : {'points, 'pixels'}
The screen units to use (pixels or points) for the offset input.

get_unit()
The unit for input to the transform used by __call__

set_unit(*unit*)
The unit for input to the transform used by __call__

Parameters unit : {'points', 'pixels'}

class matplotlib.text.**Text**(*x=0*, *y=0*, *text='‘*, *color=None*, *verticalalignment='baseline'*, *horizontalalignment='left'*, *multialignment=None*, *fontproperties=None*, *rotation=None*, *linespacing=None*, *rotation_mode=None*, *usetex=None*, *wrap=False*, ***kwargs*)
Bases: *matplotlib.artist.Artist*

Handle storing and drawing of text in window or data coordinates.

Create a *Text* instance at *x*, *y* with string *text*.

Valid kwargs are

Property	Description
agg_filter	unknown
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]

Table 73.1 – continued fror

Property	Description
axes	an *Axes* instance
backgroundcolor	any matplotlib color
bbox	FancyBboxPatch prop dict
clip_box	a *matplotlib.transforms.Bbox* instance
clip_on	[True | False]
clip_path	[(*Path*, *Transform*) | *Patch* | None]
color	any matplotlib color
contains	a callable function
family or name or fontname or fontfamily	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
figure	a *matplotlib.figure.Figure* instance
fontproperties or font_properties	a *matplotlib.font_manager.FontProperties* instance
gid	an id string
horizontalalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
multialignment	['left' | 'right' | 'center']
path_effects	unknown
picker	[None|float|boolean|callable]
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
rotation_mode	unknown
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large'
sketch_params	unknown
snap	unknown
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'c
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
transform	*Transform* instance
url	a url string
usetex	unknown
variant or fontvariant	['normal' | 'small-caps']
verticalalignment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' |
wrap	unknown
x	float
y	float
zorder	any number

contains(*mouseevent*)

Test whether the mouse event occurred in the patch.

In the case of text, a hit is true anywhere in the axis-aligned bounding-box containing the text.

Returns True or False.

draw(*artist*, *renderer*, **args*, ***kwargs*)
Draws the `Text` object to the given *renderer*.

get_bbox_patch()
Return the bbox Patch object. Returns None if the FancyBboxPatch is not made.

get_color()
Return the color of the text

get_family()
Return the list of font families used for font lookup

get_font_properties()
alias for get_fontproperties

get_fontfamily()
alias for get_family

get_fontname()
alias for get_name

get_fontproperties()
Return the `FontProperties` object

get_fontsize()
alias for get_size

get_fontstretch()
alias for get_stretch

get_fontstyle()
alias for get_style

get_fontvariant()
alias for get_variant

get_fontweight()
alias for get_weight

get_ha()
alias for get_horizontalalignment

get_horizontalalignment()
Return the horizontal alignment as string. Will be one of 'left', 'center' or 'right'.

get_name()
Return the font name as string

get_position()
Return the position of the text as a tuple (x, y)

get_prop_tup()
Return a hashable tuple of properties.

Not intended to be human readable, but useful for backends who want to cache derived information about text (e.g., layouts) and need to know if the text has changed.

get_rotation()
return the text angle as float in degrees

get_rotation_mode()
get text rotation mode

get_size()
Return the font size as integer

get_stretch()
Get the font stretch as a string or number

get_style()
Return the font style as string

get_text()
Get the text as string

get_unitless_position()
Return the unitless position of the text as a tuple (*x*, *y*)

get_usetex()
Return whether this *Text* object will render using TeX.

If the user has not manually set this value, it will default to the value of `rcParams['text.usetex']`

get_va()
alias for `getverticalalignment()`

get_variant()
Return the font variant as a string

get_verticalalignment()
Return the vertical alignment as string. Will be one of 'top', 'center', 'bottom' or 'baseline'.

get_weight()
Get the font weight as string or number

get_window_extent(*renderer=None*, *dpi=None*)
Return a *Bbox* object bounding the text, in display units.

In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of *draw()*, so you must use this kwarg if you want to call *get_window_extent()* prior to the first *draw()*. For getting web page regions, it is simpler to call the method after saving the figure.

dpi defaults to self.figure.dpi; the renderer dpi is irrelevant. For the web application, if figure.dpi is not the value used when saving the figure, then the value that was used must be specified as the *dpi* argument.

get_wrap()
Returns the wrapping state for the text.

static **is_math_text**(*s*)
Returns a cleaned string and a boolean flag. The flag indicates if the given string *s* contains any mathtext, determined by counting unescaped dollar signs. If no mathtext is present, the cleaned string has its dollar signs unescaped. If usetex is on, the flag always has the value "TeX".

set_backgroundcolor(*color*)
Set the background color of the text by updating the bbox.

See also:

`set_bbox()` To change the position of the bounding box.

ACCEPTS: any matplotlib color

set_bbox(*rectprops*)
Draw a bounding box around self. rectprops are any settable properties for a FancyBboxPatch, e.g., facecolor='red', alpha=0.5.

t.set_bbox(dict(facecolor='red', alpha=0.5))

The default boxstyle is 'square'. The mutation scale of the FancyBboxPatch is set to the fontsize.

ACCEPTS: FancyBboxPatch prop dict

set_clip_box(*clipbox*)
Set the artist's clip `Bbox`.

ACCEPTS: a `matplotlib.transforms.Bbox` instance

set_clip_on(*b*)
Set whether artist uses clipping.

When False artists will be visible out side of the axes which can lead to unexpected results.

ACCEPTS: [True | False]

set_clip_path(*path*, *transform=None*)
Set the artist's clip path, which may be:

- a `Patch` (or subclass) instance
- **a `Path` instance, in which case** an optional `Transform` instance may be provided, which will be applied to the path before using it for clipping.
- *None*, to remove the clipping path

For efficiency, if the path happens to be an axis-aligned rectangle, this method will set the clipping box to the corresponding rectangle and set the clipping path to *None*.

ACCEPTS: [(`Path`, `Transform`) | `Patch` | None]

set_color(*color*)
Set the foreground color of the text

ACCEPTS: any matplotlib color

set_family(*fontname*)
Set the font family. May be either a single string, or a list of strings in decreasing priority. Each

string may be either a real font name or a generic font class name. If the latter, the specific font names will be looked up in the `matplotlibrc` file.

ACCEPTS: [FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']

set_font_properties(*fp*)
alias for set_fontproperties

set_fontname(*fontname*)
alias for set_family

set_fontproperties(*fp*)
Set the font properties that control the text. *fp* must be a `matplotlib.font_manager.FontProperties` object.

ACCEPTS: a `matplotlib.font_manager.FontProperties` instance

set_fontsize(*fontsize*)
alias for set_size

set_fontstretch(*stretch*)
alias for set_stretch

set_fontstyle(*fontstyle*)
alias for set_style

set_fontvariant(*variant*)
alias for set_variant

set_fontweight(*weight*)
alias for set_weight

set_ha(*align*)
alias for set_horizontalalignment

set_horizontalalignment(*align*)
Set the horizontal alignment to one of

ACCEPTS: ['center' | 'right' | 'left']

set_linespacing(*spacing*)
Set the line spacing as a multiple of the font size. Default is 1.2.

ACCEPTS: float (multiple of font size)

set_ma(*align*)
alias for set_verticalalignment

set_multialignment(*align*)
Set the alignment for multiple lines layout. The layout of the bounding box of all the lines is determined bu the horizontalalignment and verticalalignment properties, but the multiline text within that box can be

ACCEPTS: ['left' | 'right' | 'center']

set_name(*fontname*)
alias for set_family

set_position(*xy*)
Set the (*x*, *y*) position of the text

ACCEPTS: (x,y)

set_rotation(*s*)
Set the rotation of the text

ACCEPTS: [angle in degrees | 'vertical' | 'horizontal']

set_rotation_mode(*m*)
set text rotation mode. If "anchor", the un-rotated text will first aligned according to their *ha* and *va*, and then will be rotated with the alignement reference point as a origin. If None (default), the text will be rotated first then will be aligned.

set_size(*fontsize*)
Set the font size. May be either a size string, relative to the default font size, or an absolute font size in points.

ACCEPTS: [size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large' | 'xx-large']

set_stretch(*stretch*)
Set the font stretch (horizontal condensation or expansion).

ACCEPTS: [a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'condensed' | 'semi-condensed' | 'normal' | 'semi-expanded' | 'expanded' | 'extra-expanded' | 'ultra-expanded']

set_style(*fontstyle*)
Set the font style.

ACCEPTS: ['normal' | 'italic' | 'oblique']

set_text(*s*)
Set the text string *s*

It may contain newlines (`\n`) or math in LaTeX syntax.

ACCEPTS: string or anything printable with '%s' conversion.

set_usetex(*usetex*)
Set this `Text` object to render using TeX (or not).

If `None` is given, the option will be reset to use the value of `rcParams['text.usetex']`

set_va(*align*)
alias for set_verticalalignment

set_variant(*variant*)
Set the font variant, either 'normal' or 'small-caps'.

ACCEPTS: ['normal' | 'small-caps']

set_verticalalignment(*align*)
Set the vertical alignment

ACCEPTS: ['center' | 'top' | 'bottom' | 'baseline']

set_weight(*weight*)
Set the font weight.

ACCEPTS: [a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' | 'book' | 'medium' | 'roman' | 'semibold' | 'demibold' | 'demi' | 'bold' | 'heavy' | 'extra bold' | 'black']

set_wrap(*wrap*)
Sets the wrapping state for the text.

set_x(*x*)
Set the *x* position of the text

ACCEPTS: float

set_y(*y*)
Set the *y* position of the text

ACCEPTS: float

update(*kwargs*)
Update properties from a dictionary.

update_bbox_position_size(*renderer*)
Update the location and the size of the bbox. This method should be used when the position and size of the bbox needs to be updated before actually drawing the bbox.

update_from(*other*)
Copy properties from other to self

zorder = 3

class matplotlib.text.**TextWithDash**(*x=0*, *y=0*, *text=''*, *color=None*, *verticalalignment='center'*, *horizontalalignment='center'*, *multialignment=None*, *fontproperties=None*, *rotation=None*, *linespacing=None*, *dashlength=0.0*, *dashdirection=0*, *dashrotation=None*, *dashpad=3*, *dashpush=0*)

Bases: `matplotlib.text.Text`

This is basically a `Text` with a dash (drawn with a `Line2D`) before/after it. It is intended to be a drop-in replacement for `Text`, and should behave identically to it when *dashlength* = 0.0.

The dash always comes between the point specified by `set_position()` and the text. When a dash exists, the text alignment arguments (*horizontalalignment*, *verticalalignment*) are ignored.

dashlength is the length of the dash in canvas units. (default = 0.0).

dashdirection is one of 0 or 1, where 0 draws the dash after the text and 1 before. (default = 0).

dashrotation specifies the rotation of the dash, and should generally stay *None*. In this case `get_dashrotation()` returns `get_rotation()`. (i.e., the dash takes its rotation from the text's rotation). Because the text center is projected onto the dash, major deviations in the rotation cause what may be considered visually unappealing results. (default = *None*)

dashpad is a padding length to add (or subtract) space between the text and the dash, in canvas units. (default = 3) *dashpush* "pushes" the dash and text away from the point specified by `set_position()` by the amount in canvas units. (default = 0)

Note: The alignment of the two objects is based on the bounding box of the `Text`, as obtained by `get_window_extent()`. This, in turn, appears to depend on the font metrics as given by the rendering backend. Hence the quality of the "centering" of the label text with respect to the dash varies depending on the backend used.

Note: I'm not sure that I got the `get_window_extent()` right, or whether that's sufficient for providing the object bounding box.

---

draw(*renderer*)
Draw the `TextWithDash` object to the given *renderer*.

get_dashdirection()
Get the direction dash. 1 is before the text and 0 is after.

get_dashlength()
Get the length of the dash.

get_dashpad()
Get the extra spacing between the dash and the text, in canvas units.

get_dashpush()
Get the extra spacing between the dash and the specified text position, in canvas units.

get_dashrotation()
Get the rotation of the dash in degrees.

get_figure()
return the figure instance the artist belongs to

get_position()
Return the position of the text as a tuple (*x*, *y*)

get_prop_tup()
Return a hashable tuple of properties.

Not intended to be human readable, but useful for backends who want to cache derived information about text (e.g., layouts) and need to know if the text has changed.

get_unitless_position()
Return the unitless position of the text as a tuple (*x*, *y*)

get_window_extent(*renderer=None*)
Return a `Bbox` object bounding the text, in display units.

In addition to being used internally, this is useful for specifying clickable regions in a png file on a web page.

renderer defaults to the _renderer attribute of the text object. This is not assigned until the first execution of `draw()`, so you must use this kwarg if you want to call `get_window_extent()`

prior to the first `draw()`. For getting web page regions, it is simpler to call the method after saving the figure.

set_dashdirection(*dd*)
Set the direction of the dash following the text. 1 is before the text and 0 is after. The default is 0, which is what you'd want for the typical case of ticks below and on the left of the figure.

ACCEPTS: int (1 is before, 0 is after)

set_dashlength(*dl*)
Set the length of the dash.

ACCEPTS: float (canvas units)

set_dashpad(*dp*)
Set the "pad" of the TextWithDash, which is the extra spacing between the dash and the text, in canvas units.

ACCEPTS: float (canvas units)

set_dashpush(*dp*)
Set the "push" of the TextWithDash, which is the extra spacing between the beginning of the dash and the specified position.

ACCEPTS: float (canvas units)

set_dashrotation(*dr*)
Set the rotation of the dash, in degrees

ACCEPTS: float (degrees)

set_figure(*fig*)
Set the figure instance the artist belong to.

ACCEPTS: a `matplotlib.figure.Figure` instance

set_position(*xy*)
Set the (*x*, *y*) position of the `TextWithDash`.

ACCEPTS: (x, y)

set_transform(*t*)
Set the `matplotlib.transforms.Transform` instance used by this artist.

ACCEPTS: a `matplotlib.transforms.Transform` instance

set_x(*x*)
Set the *x* position of the `TextWithDash`.

ACCEPTS: float

set_y(*y*)
Set the *y* position of the `TextWithDash`.

ACCEPTS: float

update_coords(*renderer*)

Computes the actual *x*, *y* coordinates for text based on the input *x*, *y* and the *dashlength*. Since the rotation is with respect to the actual canvas's coordinates we need to map back and forth.

matplotlib.text.**get_rotation**(*rotation*)

Return the text angle as float. The returned angle is between 0 and 360 deg.

rotation may be 'horizontal', 'vertical', or a numeric value in degrees.

CHAPTER SEVENTYFOUR

TICKER

74.1 matplotlib.ticker

74.1.1 Tick locating and formatting

This module contains classes to support completely configurable tick locating and formatting. Although the locators know nothing about major or minor ticks, they are used by the Axis class to support major and minor tick locating and formatting. Generic tick locators and formatters are provided, as well as domain specific custom ones..

Default Formatter

The default formatter identifies when the x-data being plotted is a small range on top of a large off set. To reduce the chances that the ticklabels overlap the ticks are labeled as deltas from a fixed offset. For example:

```
ax.plot(np.arange(2000, 2010), range(10))
```

will have tick of 0-9 with an offset of +2e3. If this is not desired turn off the use of the offset on the default formatter:

```
ax.get_xaxis().get_major_formatter().set_useOffset(False)
```

set the rcParam `axes.formatter.useoffset=False` to turn it off globally, or set a different formatter.

Tick locating

The Locator class is the base class for all tick locators. The locators handle autoscaling of the view limits based on the data limits, and the choosing of tick locations. A useful semi-automatic tick locator is MultipleLocator. You initialize this with a base, e.g., 10, and it picks axis limits and ticks that are multiples of your base.

The Locator subclasses defined here are

NullLocator No ticks

FixedLocator Tick locations are fixed

`IndexLocator` locator for index plots (e.g., where x = range(len(y)))

`LinearLocator` evenly spaced ticks from min to max

`LogLocator` logarithmically ticks from min to max

`SymmetricalLogLocator` locator for use with with the symlog norm, works like the `LogLocator` for the part outside of the threshold and add 0 if inside the limits

`MultipleLocator`

ticks and range are a multiple of base; either integer or float

`OldAutoLocator` choose a MultipleLocator and dyamically reassign it for intelligent ticking during navigation

`MaxNLocator` finds up to a max number of ticks at nice locations

`AutoLocator` `MaxNLocator` with simple defaults. This is the default tick locator for most plotting.

`AutoMinorLocator` locator for minor ticks when the axis is linear and the major ticks are uniformly spaced. It subdivides the major tick interval into a specified number of minor intervals, defaulting to 4 or 5 depending on the major interval.

There are a number of locators specialized for date locations - see the dates module

You can define your own locator by deriving from Locator. You must override the __call__ method, which returns a sequence of locations, and you will probably want to override the autoscale method to set the view limits from the data limits.

If you want to override the default locator, use one of the above or a custom locator and pass it to the x or y axis instance. The relevant methods are:

```
ax.xaxis.set_major_locator( xmajorLocator )
ax.xaxis.set_minor_locator( xminorLocator )
ax.yaxis.set_major_locator( ymajorLocator )
ax.yaxis.set_minor_locator( yminorLocator )
```

The default minor locator is the NullLocator, e.g., no minor ticks on by default.

Tick formatting

Tick formatting is controlled by classes derived from Formatter. The formatter operates on a single tick value and returns a string to the axis.

`NullFormatter` no labels on the ticks

`IndexFormatter` set the strings from a list of labels

`FixedFormatter` set the strings manually for the labels

`FuncFormatter` user defined function sets the labels

`StrMethodFormatter` Use string `format` method

`FormatStrFormatter` use a sprintf format string

`ScalarFormatter` default formatter for scalars; autopick the fmt string

`LogFormatter` formatter for log axes

You can derive your own formatter from the Formatter base class by simply overriding the `__call__` method. The formatter class has access to the axis view and data limits.

To control the major and minor tick label formats, use one of the following methods:

```
ax.xaxis.set_major_formatter( xmajorFormatter )
ax.xaxis.set_minor_formatter( xminorFormatter )
ax.yaxis.set_major_formatter( ymajorFormatter )
ax.yaxis.set_minor_formatter( yminorFormatter )
```

See *pylab_examples example code: major_minor_demo1.py* for an example of setting major and minor ticks. See the `matplotlib.dates` module for more information and examples of using date locators and formatters.

class `matplotlib.ticker.`**`TickHelper`**

Bases: `object`

`axis` **= None**

`create_dummy_axis`(***kwargs*)

`set_axis`(*axis*)

`set_bounds`(*vmin*, *vmax*)

`set_data_interval`(*vmin*, *vmax*)

`set_view_interval`(*vmin*, *vmax*)

class `matplotlib.ticker.`**`Formatter`**

Bases: `matplotlib.ticker.TickHelper`

Convert the tick location to a string

`fix_minus`(*s*)

Some classes may want to replace a hyphen for minus with the proper unicode symbol (U+2212) for typographical correctness. The default is to not replace it.

Note, if you use this method, e.g., in `format_data()` or call, you probably don't want to use it for `format_data_short()` since the toolbar uses this for interactive coord reporting and I doubt we can expect GUIs across platforms will handle the unicode correctly. So for now the classes that override `fix_minus()` should have an explicit `format_data_short()` method

`format_data`(*value*)

format_data_short(*value*)
return a short string version

get_offset()

locs = []

set_locs(*locs*)

class matplotlib.ticker.**FixedFormatter**(*seq*)
Bases: *matplotlib.ticker.Formatter*

Return fixed strings for tick labels

seq is a sequence of strings. For positions `i < len(seq)` return *seq[i]* regardless of *x*. Otherwise return ''

get_offset()

set_offset_string(*ofs*)

class matplotlib.ticker.**NullFormatter**
Bases: *matplotlib.ticker.Formatter*

Always return the empty string

class matplotlib.ticker.**FuncFormatter**(*func*)
Bases: *matplotlib.ticker.Formatter*

User defined function for formatting

The function should take in two inputs (tick value *x* and position *pos*) and return a string

class matplotlib.ticker.**FormatStrFormatter**(*fmt*)
Bases: *matplotlib.ticker.Formatter*

Use an old-style ('%' operator) format string to format the tick

class matplotlib.ticker.**StrMethodFormatter**(*fmt*)
Bases: *matplotlib.ticker.Formatter*

Use a new-style format string (as used by `str.format()`) to format the tick. The field formatting must be labeled `x`.

class matplotlib.ticker.**ScalarFormatter**(*useOffset=None*, *useMathText=None*, *useLocale=None*)
Bases: *matplotlib.ticker.Formatter*

Tick location is a plain old number. If useOffset==True and the data range is much smaller than the data average, then an offset will be determined such that the tick labels are meaningful. Scientific notation is used for data < 10^-n or data >= 10^m, where n and m are the power limits set using set_powerlimits((n,m)). The defaults for these are controlled by the axes.formatter.limits rc parameter.

fix_minus(*s*)
use a unicode minus rather than hyphen

format_data(*value*)
return a formatted string representation of a number

format_data_short(*value*)
return a short formatted string representation of a number

get_offset()
Return scientific notation, plus offset

get_useLocale()

get_useOffset()

pprint_val(*x*)

set_locs(*locs*)
set the locations of the ticks

set_powerlimits(*lims*)
Sets size thresholds for scientific notation.

e.g., `formatter.set_powerlimits((-3, 4))` sets the pre-2007 default in which scientific notation is used for numbers less than 1e-3 or greater than 1e4. See also `set_scientific()`.

set_scientific(*b*)
True or False to turn scientific notation on or off see also `set_powerlimits()`

set_useLocale(*val*)

set_useOffset(*val*)

useLocale

useOffset

class matplotlib.ticker.**LogFormatter**(*base=10.0*, *labelOnlyBase=True*)
Bases: `matplotlib.ticker.Formatter`

Format values for log axis;

base is used to locate the decade tick, which will be the only one to be labeled if *labelOnlyBase* is `False`

base(*base*)
change the *base* for labeling - warning: should always match the base used for `LogLocator`

format_data(*value*)

format_data_short(*value*)
return a short formatted string representation of a number

label_minor(*labelOnlyBase*)
switch on/off minor ticks labeling

pprint_val(*x*, *d*)

class `matplotlib.ticker.`**LogFormatterExponent**(*base=10.0*, *labelOnlyBase=True*)
Bases: `matplotlib.ticker.LogFormatter`

Format values for log axis; using `exponent = log_base(value)`

base is used to locate the decade tick, which will be the only one to be labeled if *labelOnlyBase* is `False`

class `matplotlib.ticker.`**LogFormatterMathtext**(*base=10.0*, *labelOnlyBase=True*)
Bases: `matplotlib.ticker.LogFormatter`

Format values for log axis; using `exponent = log_base(value)`

base is used to locate the decade tick, which will be the only one to be labeled if *labelOnlyBase* is `False`

class `matplotlib.ticker.`**Locator**
Bases: `matplotlib.ticker.TickHelper`

Determine the tick locations;

Note, you should not use the same locator between different `Axis` because the locator stores references to the Axis data and view limits

MAXTICKS = 1000

autoscale()
autoscale the view limits

pan(*numsteps*)
Pan numticks (can be positive or negative)

raise_if_exceeds(*locs*)
raise a RuntimeError if Locator attempts to create more than MAXTICKS locs

refresh()
refresh internal information based on current lim

set_params(***kwargs*)
Do nothing, and rase a warning. Any locator class not supporting the set_params() function will call this.

tick_values(*vmin*, *vmax*)
Return the values of the located ticks given **vmin** and **vmax**.

Note: To get tick locations with the vmin and vmax values defined automatically for the associated `axis` simply call the Locator instance:

```
>>> print((type(loc)))
<type 'Locator'>
>>> print((loc()))
[1, 2, 3, 4]
```

view_limits(*vmin*, *vmax*)
select a scale for the range from vmin to vmax

Normally this method is overridden by subclasses to change locator behaviour.

zoom(*direction*)
Zoom in/out on axis; if direction is >0 zoom in, else zoom out

class matplotlib.ticker.**IndexLocator**(*base*, *offset*)
Bases: *matplotlib.ticker.Locator*

Place a tick on every multiple of some base number of points plotted, e.g., on every 5th point. It is assumed that you are doing index plotting; i.e., the axis is 0, len(data). This is mainly useful for x ticks.

place ticks on the i-th data points where (i-offset)%base==0

set_params(*base=None*, *offset=None*)
Set parameters within this locator

tick_values(*vmin*, *vmax*)

class matplotlib.ticker.**FixedLocator**(*locs*, *nbins=None*)
Bases: *matplotlib.ticker.Locator*

Tick locations are fixed. If nbins is not None, the array of possible positions will be subsampled to keep the number of ticks <= nbins +1. The subsampling will be done so as to include the smallest absolute value; for example, if zero is included in the array of possibilities, then it is guaranteed to be one of the chosen ticks.

set_params(*nbins=None*)
Set parameters within this locator.

tick_values(*vmin*, *vmax*)
" Return the locations of the ticks.

Note: Because the values are fixed, vmin and vmax are not used in this method.

class `matplotlib.ticker.NullLocator`

Bases: `matplotlib.ticker.Locator`

No ticks

`tick_values`(*vmin*, *vmax*)

” Return the locations of the ticks.

Note: Because the values are Null, vmin and vmax are not used in this method.

class `matplotlib.ticker.LinearLocator`(*numticks=None*, *presets=None*)

Bases: `matplotlib.ticker.Locator`

Determine the tick locations

The first time this function is called it will try to set the number of ticks to make a nice tick partitioning. Thereafter the number of ticks will be fixed so that interactive navigation will be nice

Use presets to set locs based on lom. A dict mapping vmin, vmax->locs

`set_params`(*numticks=None*, *presets=None*)

Set parameters within this locator.

`tick_values`(*vmin*, *vmax*)

`view_limits`(*vmin*, *vmax*)

Try to choose the view limits intelligently

class `matplotlib.ticker.LogLocator`(*base=10.0, subs=[1.0], numdecs=4, numticks=15*)

Bases: `matplotlib.ticker.Locator`

Determine the tick locations for log axes

place ticks on the location= base**i*subs[j]

`base`(*base*)

set the base of the log scaling (major tick every base**i, i integer)

`set_params`(*base=None*, *subs=None*, *numdecs=None*, *numticks=None*)

Set parameters within this locator.

`subs`(*subs*)

set the minor ticks the log scaling every base**i*subs[j]

`tick_values`(*vmin*, *vmax*)

`view_limits`(*vmin*, *vmax*)

Try to choose the view limits intelligently

class `matplotlib.ticker.AutoLocator`

Bases: `matplotlib.ticker.MaxNLocator`

class `matplotlib.ticker.MultipleLocator`(*base=1.0*)

Bases: `matplotlib.ticker.Locator`

Set a tick on every integer that is multiple of base in the view interval

set_params(*base*)
Set parameters within this locator.

tick_values(*vmin*, *vmax*)

view_limits(*dmin*, *dmax*)
Set the view limits to the nearest multiples of base that contain the data

class matplotlib.ticker.**MaxNLocator**(**args*, ***kwargs*)
Bases: `matplotlib.ticker.Locator`

Select no more than N intervals at nice locations.

Keyword args:
nbins Maximum number of intervals; one less than max number of ticks.
steps Sequence of nice numbers starting with 1 and ending with 10; e.g., [1, 2, 4, 5, 10]
integer If True, ticks will take only integer values.
symmetric If True, autoscaling will result in a range symmetric about zero.
prune ['lower' | 'upper' | 'both' | None] Remove edge ticks – useful for stacked or ganged plots where the upper tick of one axes overlaps with the lower tick of the axes above it. If prune=='lower', the smallest tick will be removed. If prune=='upper', the largest tick will be removed. If prune=='both', the largest and smallest ticks will be removed. If prune==None, no ticks will be removed.

bin_boundaries(*vmin*, *vmax*)

default_params = {'steps': None, 'trim': True, 'symmetric': False, 'nbins': 10, 'integer': False, 'prune': Non

set_params(***kwargs*)
Set parameters within this locator.

tick_values(*vmin*, *vmax*)

view_limits(*dmin*, *dmax*)

class matplotlib.ticker.**AutoMinorLocator**(*n=None*)
Bases: `matplotlib.ticker.Locator`

Dynamically find minor tick positions based on the positions of major ticks. Assumes the scale is linear and major ticks are evenly spaced.

n is the number of subdivisions of the interval between major ticks; e.g., n=2 will place a single minor tick midway between major ticks.

If *n* is omitted or None, it will be set to 5 or 4.

tick_values(*vmin*, *vmax*)

class `matplotlib.ticker.SymmetricalLogLocator`(*transform*, *subs=None*)

Bases: `matplotlib.ticker.Locator`

Determine the tick locations for log axes

place ticks on the location= base**i*subs[j]

set_params(*subs=None*, *numticks=None*)

Set parameters within this locator.

tick_values(*vmin*, *vmax*)

view_limits(*vmin*, *vmax*)

Try to choose the view limits intelligently

CHAPTER SEVENTYFIVE

TIGHT_LAYOUT

75.1 matplotlib.tight_layout

This module provides routines to adjust subplot params so that subplots are nicely fit in the figure. In doing so, only axis labels, tick labels, axes titles and offsetboxes that are anchored to axes are currently considered.

Internally, it assumes that the margins (left_margin, etc.) which are differences between ax.get_tightbbox and ax.bbox are independent of axes position. This may fail if Axes.adjustable is datalim. Also, This will fail for some cases (for example, left or right margin is affected by xlabel).

`matplotlib.tight_layout.`**`auto_adjust_subplotpars`**(*fig*, *renderer*, *nrows_ncols*, *num1num2_list*, *subplot_list*, *ax_bbox_list=None*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Return a dictionary of subplot parameters so that spacing between subplots are adjusted. Note that this function ignore geometry information of subplot itself, but uses what is given by *nrows_ncols* and *num1num2_list* parameteres. Also, the results could be incorrect if some subplots have `adjustable=datalim`.

Parameters:

nrows_ncols number of rows and number of columns of the grid.
num1num2_list list of numbers specifying the area occupied by the subplot
subplot_list list of subplots that will be used to calcuate optimal subplot_params.
pad [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.
h_pad, w_pad [float]
padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.
rect [left, bottom, right, top] in normalized (0, 1) figure coordinates.

`matplotlib.tight_layout.`**`get_renderer`**(*fig*)

`matplotlib.tight_layout.`**`get_subplotspec_list`**(*axes_list*, *grid_spec=None*)

Return a list of subplotspec from the given list of axes. For an instance of axes that does not support subplotspec, None is inserted in the list.

If grid_spec is given, None is inserted for those not from the given grid_spec.

`matplotlib.tight_layout.get_tight_layout_figure`(*fig*, *axes_list*, *subplotspec_list*, *renderer*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)

Return subplot parameters for tight-layouted-figure with specified padding.

Parameters:

fig : figure instance

axes_list : a list of axes

subplotspec_list [a list of subplotspec associated with each] axes in axes_list

renderer : renderer instance

pad [float] padding between the figure edge and the edges of subplots, as a fraction of the font-size.

h_pad*, *w_pad [float] padding (height/width) between edges of adjacent subplots. Defaults to `pad_inches`.

rect [if rect is given, it is interpreted as a rectangle] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into. Default is (0, 0, 1, 1).

CHAPTER

SEVENTYSIX

TRIANGULAR GRIDS

76.1 matplotlib.tri

Unstructured triangular grid functions.

class matplotlib.tri.**Triangulation**(*x*, *y*, *triangles=None*, *mask=None*)

An unstructured triangular grid consisting of npoints points and ntri triangles. The triangles can either be specified by the user or automatically generated using a Delaunay triangulation.

Parameters x, y : array_like of shape (npoints)

Coordinates of grid points.

triangles : integer array_like of shape (ntri, 3), optional

For each triangle, the indices of the three points that make up the triangle, ordered in an anticlockwise manner. If not specified, the Delaunay triangulation is calculated.

mask : boolean array_like of shape (ntri), optional

Which triangles are masked out.

Notes

For a Triangulation to be valid it must not have duplicate points, triangles formed from colinear points, or overlapping triangles.

Attributes

edges	
neighbors	
is_delaunay	(bool) Whether the Triangulation is a calculated Delaunay triangulation (where `triangles` was not specified) or not.

calculate_plane_coefficients(*z*)

Calculate plane equation coefficients for all unmasked triangles from the point (x,y) coordinates and specified z-array of shape (npoints). Returned array has shape (npoints,3) and allows z-value at (x,y) position in triangle tri to be calculated using z = array[tri,0]*x + array[tri,1]*y + array[tri,2].

edges

Return integer array of shape (nedges,2) containing all edges of non-masked triangles.

Each edge is the start point index and end point index. Each edge (start,end and end,start) appears only once.

static **get_from_args_and_kwargs**(**args*, ***kwargs*)

Return a Triangulation object from the args and kwargs, and the remaining args and kwargs with the consumed values removed.

There are two alternatives: either the first argument is a Triangulation object, in which case it is returned, or the args and kwargs are sufficient to create a new Triangulation to return. In the latter case, see Triangulation.__init__ for the possible args and kwargs.

get_masked_triangles()

Return an array of triangles that are not masked.

get_trifinder()

Return the default `matplotlib.tri.TriFinder` of this triangulation, creating it if necessary. This allows the same TriFinder object to be easily shared.

neighbors

Return integer array of shape (ntri,3) containing neighbor triangles.

For each triangle, the indices of the three triangles that share the same edges, or -1 if there is no such neighboring triangle. neighbors[i,j] is the triangle that is the neighbor to the edge from point index triangles[i,j] to point index triangles[i,(j+1)%3].

set_mask(*mask*)

Set or clear the mask array. This is either None, or a boolean array of shape (ntri).

class matplotlib.tri.**TriFinder**(*triangulation*)

Abstract base class for classes used to find the triangles of a Triangulation in which (x,y) points lie.

Rather than instantiate an object of a class derived from TriFinder, it is usually better to use the function `matplotlib.tri.Triangulation.get_trifinder()`.

Derived classes implement __call__(x,y) where x,y are array_like point coordinates of the same shape.

class matplotlib.tri.**TrapezoidMapTriFinder**(*triangulation*)

Bases: `matplotlib.tri.trifinder.TriFinder`

`TriFinder` class implemented using the trapezoid map algorithm from the book "Computational Geometry, Algorithms and Applications", second edition, by M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf.

The triangulation must be valid, i.e. it must not have duplicate points, triangles formed from colinear points, or overlapping triangles. The algorithm has some tolerance to triangles formed from colinear points, but this should not be relied upon.

__call__(*x*, *y*)

Return an array containing the indices of the triangles in which the specified x,y points lie, or -1 for points that do not lie within a triangle.

x, *y* are array_like x and y coordinates of the same shape and any number of dimensions.

Returns integer array with the same shape and *x* and *y*.

class `matplotlib.tri.TriInterpolator`(*triangulation*, *z*, *trifinder=None*)

Abstract base class for classes used to perform interpolation on triangular grids.

Derived classes implement the following methods:

- `__call__(x, y)` , where x, y are array_like point coordinates of the same shape, and that returns a masked array of the same shape containing the interpolated z-values.
- `gradient(x, y)` , where x, y are array_like point coordinates of the same shape, and that returns a list of 2 masked arrays of the same shape containing the 2 derivatives of the interpolator (derivatives of interpolated z values with respect to x and y).

class `matplotlib.tri.LinearTriInterpolator`(*triangulation*, *z*, *trifinder=None*)

Bases: `matplotlib.tri.triinterpolate.TriInterpolator`

A LinearTriInterpolator performs linear interpolation on a triangular grid.

Each triangle is represented by a plane so that an interpolated value at point (x,y) lies on the plane of the triangle containing (x,y). Interpolated values are therefore continuous across the triangulation, but their first derivatives are discontinuous at edges between triangles.

Parameters triangulation : `Triangulation` object

The triangulation to interpolate over.

z : array_like of shape (npoints,)

Array of values, defined at grid points, to interpolate between.

trifinder : `TriFinder` object, optional

If this is not specified, the Triangulation's default TriFinder will be used by calling `matplotlib.tri.Triangulation.get_trifinder()`.

Methods

`__call__` (x, y)	(Returns interpolated values at x,y points)
`gradient` (x, y)	(Returns interpolated derivatives at x,y points)

`__call__`(*x*, *y*)

Returns a masked array containing interpolated values at the specified x,y points.

Parameters x, y : array-like

x and y coordinates of the same shape and any number of dimensions.

Returns z : np.ma.array

Masked array of the same shape as *x* and *y* ; values corresponding to (*x*, *y*) points outside of the triangulation are masked out.

`gradient`(*x*, *y*)

Returns a list of 2 masked arrays containing interpolated derivatives at the specified x,y points.

Parameters x, y : array-like

x and y coordinates of the same shape and any number of dimensions.

Returns dzdx, dzdy : np.ma.array

2 masked arrays of the same shape as x and y ; values corresponding to (x,y) points outside of the triangulation are masked out. The first returned array contains the values of $\frac{\partial z}{\partial x}$ and the second those of $\frac{\partial z}{\partial y}$.

class `matplotlib.tri.`**`CubicTriInterpolator`**(*triangulation*, *z*, *kind='min_E'*, *trifinder=None*, *dz=None*)

Bases: `matplotlib.tri.triinterpolate.TriInterpolator`

A CubicTriInterpolator performs cubic interpolation on triangular grids.

In one-dimension - on a segment - a cubic interpolating function is defined by the values of the function and its derivative at both ends. This is almost the same in 2-d inside a triangle, except that the values of the function and its 2 derivatives have to be defined at each triangle node.

The CubicTriInterpolator takes the value of the function at each node - provided by the user - and internally computes the value of the derivatives, resulting in a smooth interpolation. (As a special feature, the user can also impose the value of the derivatives at each node, but this is not supposed to be the common usage.)

Parameters triangulation : `Triangulation` object

The triangulation to interpolate over.

z : array_like of shape (npoints,)

Array of values, defined at grid points, to interpolate between.

kind : {'min_E', 'geom', 'user'}, optional

Choice of the smoothing algorithm, in order to compute the interpolant derivatives (defaults to 'min_E'):

- if 'min_E': (default) The derivatives at each node is computed to minimize a bending energy.
- if 'geom': The derivatives at each node is computed as a weighted average of relevant triangle normals. To be used for speed optimization (large grids).
- if 'user': The user provides the argument `dz`, no computation is hence needed.

trifinder : `TriFinder` object, optional

If not specified, the Triangulation's default TriFinder will be used by calling `matplotlib.tri.Triangulation.get_trifinder()`.

dz : tuple of array_likes (dzdx, dzdy), optional

Used only if *kind* ='user'. In this case *dz* must be provided as (dzdx, dzdy) where dzdx, dzdy are arrays of the same shape as *z* and are the interpolant first derivatives at the *triangulation* points.

Notes

This note is a bit technical and details the way a `CubicTriInterpolator` computes a cubic interpolation.

The interpolation is based on a Clough-Tocher subdivision scheme of the *triangulation* mesh (to make it clearer, each triangle of the grid will be divided in 3 child-triangles, and on each child triangle the interpolated function is a cubic polynomial of the 2 coordinates). This technique originates from

FEM (Finite Element Method) analysis; the element used is a reduced Hsieh-Clough-Tocher (HCT) element. Its shape functions are described in [R1]. The assembled function is guaranteed to be C1-smooth, i.e. it is continuous and its first derivatives are also continuous (this is easy to show inside the triangles but is also true when crossing the edges).

In the default case (*kind* ='min_E'), the interpolant minimizes a curvature energy on the functional space generated by the HCT element shape functions - with imposed values but arbitrary derivatives at each node. The minimized functional is the integral of the so-called total curvature (implementation based on an algorithm from [R2] - PCG sparse solver):

$$E(z) = \frac{1}{2} \int_{\Omega} \left(\left(\frac{\partial^2 z}{\partial x^2} \right)^2 + \left(\frac{\partial^2 z}{\partial y^2} \right)^2 + 2 \left(\frac{\partial^2 z}{\partial y \partial x} \right)^2 \right) dx\, dy \tag{76.1}$$

If the case *kind* ='geom' is chosen by the user, a simple geometric approximation is used (weighted average of the triangle normal vectors), which could improve speed on very large grids.

References

[R1], [R2]

Methods

`__call__` (x, y)	(Returns interpolated values at x,y points)
`gradient` (x, y)	(Returns interpolated derivatives at x,y points)

`__call__`(*x*, *y*)

Returns a masked array containing interpolated values at the specified x,y points.

Parameters x, y : array-like

x and y coordinates of the same shape and any number of dimensions.

Returns z : np.ma.array

Masked array of the same shape as *x* and *y* ; values corresponding to (*x*, *y*) points outside of the triangulation are masked out.

`gradient`(*x*, *y*)

Returns a list of 2 masked arrays containing interpolated derivatives at the specified x,y points.

Parameters x, y : array-like

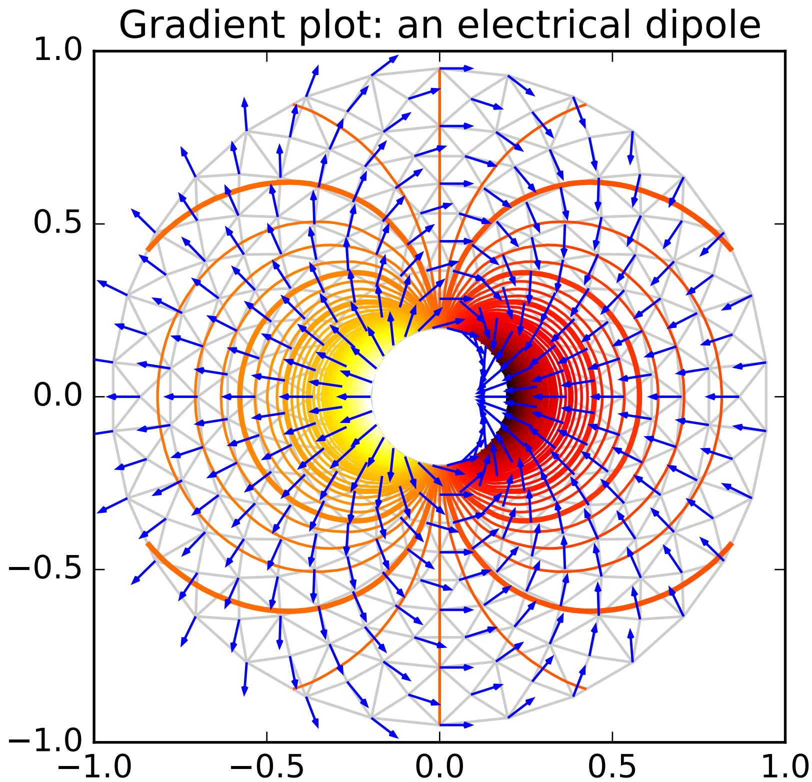
x and y coordinates of the same shape and any number of dimensions.

Returns dzdx, dzdy : np.ma.array

2 masked arrays of the same shape as *x* and *y* ; values corresponding to (x,y) points outside of the triangulation are masked out. The first returned array contains the values of $\frac{\partial z}{\partial x}$ and the second those of $\frac{\partial z}{\partial y}$.

Examples

An example of effective application is shown below (plot of the direction of the vector field derivated from a known potential field):



class `matplotlib.tri.`**`TriRefiner`**(*triangulation*)

Abstract base class for classes implementing mesh refinement.

A TriRefiner encapsulates a Triangulation object and provides tools for mesh refinement and interpolation.

Derived classes must implements:

- `refine_triangulation(return_tri_index=False, **kwargs)` , where the optional keyword arguments *kwargs* are defined in each TriRefiner concrete implementation, and which returns :
  - a refined triangulation
  - optionally (depending on *return_tri_index*), for each point of the refined triangulation: the index of the initial triangulation triangle to which it belongs.
- `refine_field(z, triinterpolator=None, **kwargs)` , where:
  - *z* array of field values (to refine) defined at the base triangulation nodes
  - *triinterpolator* is a `TriInterpolator` (optional)
  - the other optional keyword arguments *kwargs* are defined in each TriRefiner concrete implementation

  and which returns (as a tuple) a refined triangular mesh and the interpolated values of the field at the refined triangulation nodes.

class `matplotlib.tri.UniformTriRefiner`(*triangulation*)

Bases: `matplotlib.tri.trirefine.TriRefiner`

Uniform mesh refinement by recursive subdivisions.

Parameters triangulation : `Triangulation`

The encapsulated triangulation (to be refined)

`refine_field`(*z*, *triinterpolator=None*, *subdiv=3*)

Refines a field defined on the encapsulated triangulation.

Returns *refi_tri* (refined triangulation), *refi_z* (interpolated values of the field at the node of the refined triangulation).

Parameters z : 1d-array-like of length `n_points`

Values of the field to refine, defined at the nodes of the encapsulated triangulation. (`n_points` is the number of points in the initial triangulation)

triinterpolator : `TriInterpolator`, optional

Interpolator used for field interpolation. If not specified, a `CubicTriInterpolator` will be used.

subdiv : integer, optional

Recursion level for the subdivision. Defaults to 3. Each triangle will be divided into `4**subdiv` child triangles.

Returns refi_tri : `Triangulation` object

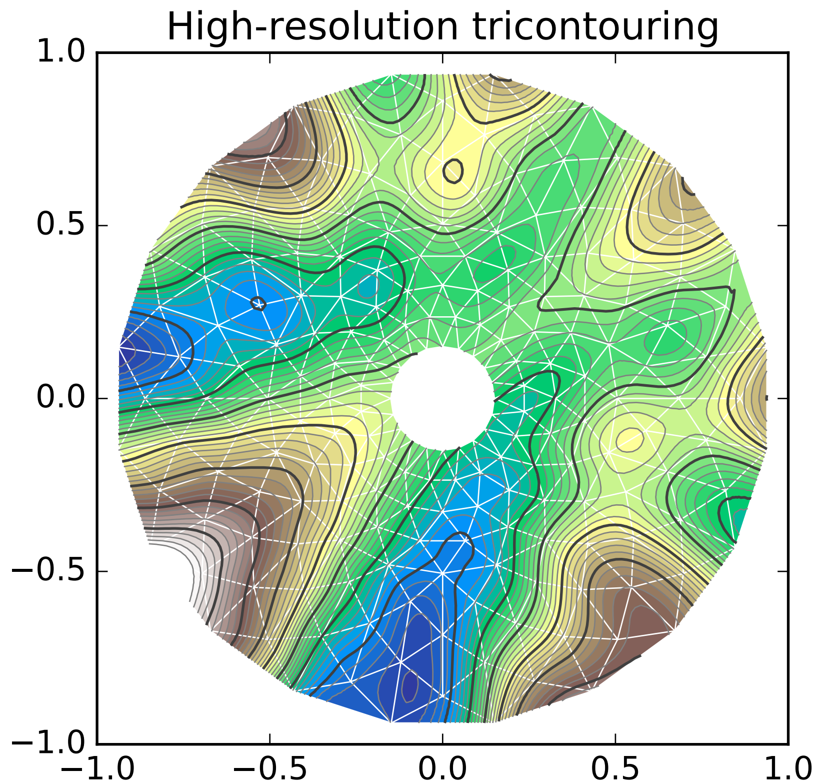
The returned refined triangulation

refi_z : 1d array of length: *refi_tri* node count.

The returned interpolated field (at *refi_tri* nodes)

Examples

The main application of this method is to plot high-quality iso-contours on a coarse triangular grid (e.g., triangulation built from relatively sparse test data):



refine_triangulation(*return_tri_index=False*, *subdiv=3*)

Computes an uniformly refined triangulation *refi_triangulation* of the encapsulated `triangulation`.

This function refines the encapsulated triangulation by splitting each father triangle into 4 child sub-triangles built on the edges midside nodes, recursively (level of recursion *subdiv*). In the end, each triangle is hence divided into `4**subdiv` child triangles. The default value for *subdiv* is 3 resulting in 64 refined subtriangles for each triangle of the initial triangulation.

Parameters return_tri_index : boolean, optional

Boolean indicating whether an index table indicating the father triangle index of each point will be returned. Default value False.

subdiv : integer, optional

Recursion level for the subdivision. Defaults value 3. Each triangle will be divided into `4**subdiv` child triangles.

Returns refi_triangulation : `Triangulation`

The returned refined triangulation

found_index : array-like of integers

Index of the initial triangulation containing triangle, for each point of *refi_triangulation*. Returned only if *return_tri_index* is set to True.

class `matplotlib.tri.`**`TriAnalyzer`**(*triangulation*)

Define basic tools for triangular mesh analysis and improvement.

A TriAnalizer encapsulates a `Triangulation` object and provides basic tools for mesh analysis and mesh improvement.

Parameters triangulation : `Triangulation` object
The encapsulated triangulation to analyze.

Attributes

scale_factors	

circle_ratios(*rescale=True*)
Returns a measure of the triangulation triangles flatness.

The ratio of the incircle radius over the circumcircle radius is a widely used indicator of a triangle flatness. It is always <= `0.5` and == `0.5` only for equilateral triangles. Circle ratios below 0.01 denote very flat triangles.

To avoid unduly low values due to a difference of scale between the 2 axis, the triangular mesh can first be rescaled to fit inside a unit square with *scale_factors* (Only if *rescale* is True, which is its default value).

Parameters rescale : boolean, optional
If True, a rescaling will be internally performed (based on *scale_factors*, so that the (unmasked) triangles fit exactly inside a unit square mesh. Default is True.

Returns circle_ratios : masked array
Ratio of the incircle radius over the circumcircle radius, for each 'rescaled' triangle of the encapsulated triangulation. Values corresponding to masked triangles are masked out.

get_flat_tri_mask(*min_circle_ratio=0.01*, *rescale=True*)
Eliminates excessively flat border triangles from the triangulation.

Returns a mask *new_mask* which allows to clean the encapsulated triangulation from its border-located flat triangles (according to their `circle_ratios()`). This mask is meant to be subsequently applied to the triangulation using `matplotlib.tri.Triangulation.set_mask()` . *new_mask* is an extension of the initial triangulation mask in the sense that an initially masked triangle will remain masked.

The *new_mask* array is computed recursively ; at each step flat triangles are removed only if they share a side with the current mesh border. Thus no new holes in the triangulated domain will be created.

Parameters min_circle_ratio : float, optional
Border triangles with incircle/circumcircle radii ratio r/R will be removed if r/R < *min_circle_ratio*. Default value: 0.01

rescale : boolean, optional
If True, a rescaling will first be internally performed (based on *scale_factors*), so that the (unmasked) triangles fit exactly inside a unit square mesh. This rescaling accounts for the difference of scale which might exist between the 2 axis. Default (and recommended) value is True.

Returns new_mask : array-like of booleans

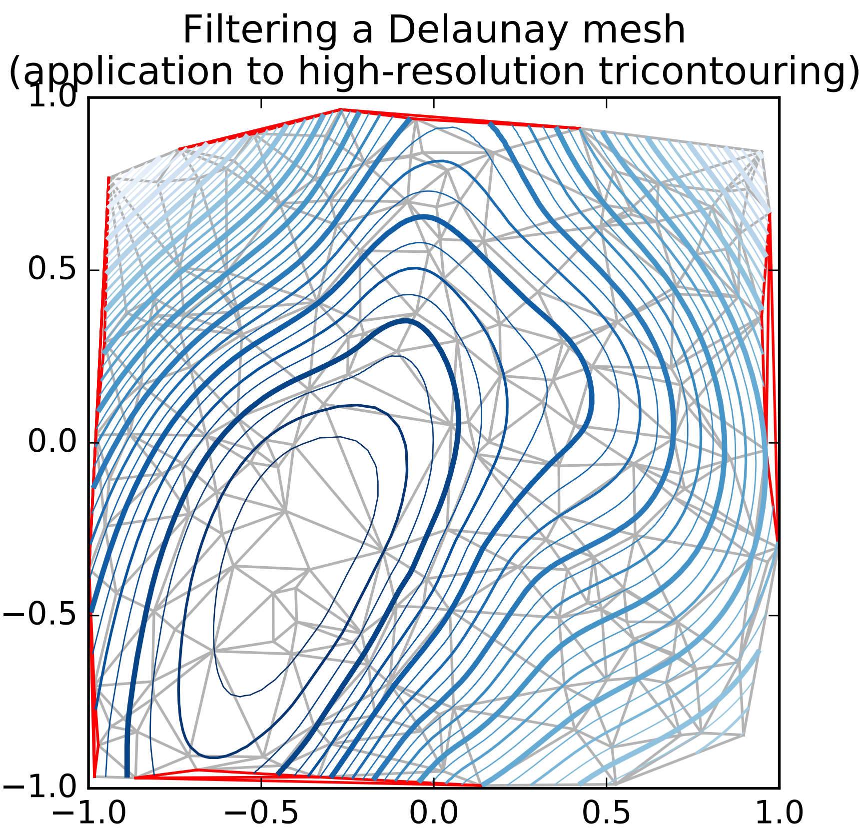
Mask to apply to encapsulated triangulation. All the initially masked triangles remain masked in the *new_mask*.

Notes

The rationale behind this function is that a Delaunay triangulation - of an unstructured set of points - sometimes contains almost flat triangles at its border, leading to artifacts in plots (especially for high-resolution contouring). Masked with computed *new_mask*, the encapsulated triangulation would contain no more unmasked border triangles with a circle ratio below *min_circle_ratio*, thus improving the mesh quality for subsequent plots or interpolation.

Examples

Please refer to the following illustrating example:



scale_factors

Factors to rescale the triangulation into a unit square.

Returns *k*, tuple of 2 scale factors.

Returns k : tuple of 2 floats (kx, ky)

Tuple of floats that would rescale the triangulation : `[triangulation.x * kx, triangulation.y * ky]` fits exactly inside a unit square.

CHAPTER SEVENTYSEVEN

TYPE1FONT

77.1 matplotlib.type1font

This module contains a class representing a Type 1 font.

This version reads pfa and pfb files and splits them for embedding in pdf files. It also supports SlantFont and ExtendFont transformations, similarly to pdfTeX and friends. There is no support yet for subsetting.

Usage:

```
>>> font = Type1Font(filename)
>>> clear_part, encrypted_part, finale = font.parts
>>> slanted_font = font.transform({'slant': 0.167})
>>> extended_font = font.transform({'extend': 1.2})
```

Sources:

- Adobe Technical Note #5040, Supporting Downloadable PostScript Language Fonts.
- Adobe Type 1 Font Format, Adobe Systems Incorporated, third printing, v1.1, 1993. ISBN 0-201-57044-0.

class `matplotlib.type1font.`**`Type1Font`**`(`*input*`)`

Bases: `object`

A class representing a Type-1 font, for use by backends.

`parts`
A 3-tuple of the cleartext part, the encrypted part, and the finale of zeros.

`prop`
A dictionary of font properties.

Initialize a Type-1 font. *input* can be either the file name of a pfb file or a 3-tuple of already-decoded Type-1 font parts.

`parts`

`prop`

transform(*effects*)

Transform the font by slanting or extending. *effects* should be a dict where `effects['slant']` is the tangent of the angle that the font is to be slanted to the right (so negative values slant to the left) and `effects['extend']` is the multiplier by which the font is to be extended (so values less than 1.0 condense). Returns a new *Type1Font* object.

`matplotlib.type1font.`**ord**(*x*)

CHAPTER SEVENTYEIGHT

UNITS

78.1 matplotlib.units

The classes here provide support for using custom classes with matplotlib, e.g., those that do not expose the array interface but know how to converter themselves to arrays. It also supoprts classes with units and units conversion. Use cases include converters for custom objects, e.g., a list of datetime objects, as well as for objects that are unit aware. We don't assume any particular units implementation, rather a units implementation must provide a ConversionInterface, and the register with the Registry converter dictionary. For example, here is a complete implementation which supports plotting with native datetime objects:

```
import matplotlib.units as units
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import datetime

class DateConverter(units.ConversionInterface):

    @staticmethod
    def convert(value, unit, axis):
        'convert value to a scalar or array'
        return dates.date2num(value)

    @staticmethod
    def axisinfo(unit, axis):
        'return major and minor tick locators and formatters'
        if unit!='date': return None
        majloc = dates.AutoDateLocator()
        majfmt = dates.AutoDateFormatter(majloc)
        return AxisInfo(majloc=majloc,
                        majfmt=majfmt,
                        label='date')

    @staticmethod
    def default_units(x, axis):
        'return the default unit for x or None'
        return 'date'

# finally we register our object type with a converter
units.registry[datetime.date] = DateConverter()
```

class `matplotlib.units.`**`AxisInfo`**(*majloc=None*, *minloc=None*, *majfmt=None*, *minfmt=None*, *label=None*, *default_limits=None*)

Bases: `object`

information to support default axis labeling and tick labeling, and default limits

majloc and minloc: TickLocators for the major and minor ticks majfmt and minfmt: TickFormatters for the major and minor ticks label: the default axis label default_limits: the default min, max of the axis if no data is present If any of the above are None, the axis will simply use the default

class `matplotlib.units.`**`ConversionInterface`**

Bases: `object`

The minimal interface for a converter to take custom instances (or sequences) and convert them to values mpl can use

static **`axisinfo`**(*unit*, *axis*)

return an units.AxisInfo instance for axis with the specified units

static **`convert`**(*obj*, *unit*, *axis*)

convert obj using unit for the specified axis. If obj is a sequence, return the converted sequence. The ouput must be a sequence of scalars that can be used by the numpy array layer

static **`default_units`**(*x*, *axis*)

return the default unit for x or None for the given axis

static **`is_numlike`**(*x*)

The matplotlib datalim, autoscaling, locators etc work with scalars which are the units converted to floats given the current unit. The converter may be passed these floats, or arrays of them, even when units are set. Derived conversion interfaces may opt to pass plain-ol unitless numbers through the conversion interface and this is a helper function for them.

class `matplotlib.units.`**`Registry`**

Bases: `dict`

register types with conversion interface

`get_converter`(*x*)

get the converter interface instance for x, or None

CHAPTER

SEVENTYNINE

WIDGETS

79.1 matplotlib.widgets

79.1.1 GUI neutral widgets

Widgets that are designed to work for any of the GUI backends. All of these widgets require you to predefine a `matplotlib.axes.Axes` instance and pass that as the first arg. matplotlib doesn't try to be too smart with respect to layout – you will have to figure out how wide and tall you want your Axes to be to accommodate your widget.

class `matplotlib.widgets.`**`AxesWidget`**(*ax*)

Bases: `matplotlib.widgets.Widget`

Widget that is connected to a single `Axes`.

To guarantee that the widget remains responsive and not garbage-collected, a reference to the object should be maintained by the user.

This is necessary because the callback registry maintains only weak-refs to the functions, which are member functions of the widget. If there are no references to the widget object it may be garbage collected which will disconnect the callbacks.

Attributes:

ax [`Axes`] The parent axes for the widget

canvas [`FigureCanvasBase` subclass] The parent figure canvas for the widget.

active [bool] If False, the widget does not respond to events.

`connect_event`(*event*, *callback*)

Connect callback with an event.

This should be used in lieu of `figure.canvas.mpl_connect` since this function stores callback ids for later clean up.

`disconnect_events`()

Disconnect all events created by this widget.

class `matplotlib.widgets.`**`Button`**(*ax*, *label*, *image=None*, *color='0.85'*, *hovercolor='0.95'*)

Bases: `matplotlib.widgets.AxesWidget`

A GUI neutral button.

For the button to remain responsive you must keep a reference to it.

The following attributes are accessible

ax The `matplotlib.axes.Axes` the button renders into.

label A `matplotlib.text.Text` instance.

color The color of the button when not hovering.

hovercolor The color of the button when hovering.

Call `on_clicked()` to connect to the button

Parameters ax : matplotlib.axes.Axes

The `matplotlib.axes.Axes` instance the button will be placed into.

label : str

The button text. Accepts string.

image : array, mpl image, Pillow Image

The image to place in the button, if not *None*. Can be any legal arg to imshow (numpy array, matplotlib Image instance, or Pillow Image).

color : color

The color of the button when not activated

hovercolor : color

The color of the button when the mouse is over it

disconnect(*cid*)

remove the observer with connection id *cid*

on_clicked(*func*)

When the button is clicked, call this *func* with event.

A connection id is returned. It can be used to disconnect the button from its callback.

class matplotlib.widgets.**CheckButtons**(*ax*, *labels*, *actives*)

Bases: `matplotlib.widgets.AxesWidget`

A GUI neutral radio button.

For the check buttons to remain responsive you must keep a reference to this object.

The following attributes are exposed

ax The `matplotlib.axes.Axes` instance the buttons are located in

labels List of `matplotlib.text.Text` instances

lines List of (line1, line2) tuples for the x's in the check boxes. These lines exist for each box, but have `set_visible(False)` when its box is not checked.

rectangles List of `matplotlib.patches.Rectangle` instances

Connect to the CheckButtons with the `on_clicked()` method

Add check buttons to `matplotlib.axes.Axes` instance *ax*

labels A len(buttons) list of labels as strings

actives

A len(buttons) list of booleans indicating whether the button is active

disconnect(*cid*)

remove the observer with connection id *cid*

on_clicked(*func*)

When the button is clicked, call *func* with button label

A connection id is returned which can be used to disconnect

set_active(*index*)
Directly (de)activate a check button by index.

***index* is an index into the original label list** that this object was constructed with. Raises ValueError if *index* is invalid.

Callbacks will be triggered if `eventson` is True.

class `matplotlib.widgets.`**Cursor**(*ax*, *horizOn=True*, *vertOn=True*, *useblit=False*, ***lineprops*)
Bases: `matplotlib.widgets.AxesWidget`

A horizontal and vertical line that spans the axes and moves with the pointer. You can turn off the hline or vline respectively with the following attributes:

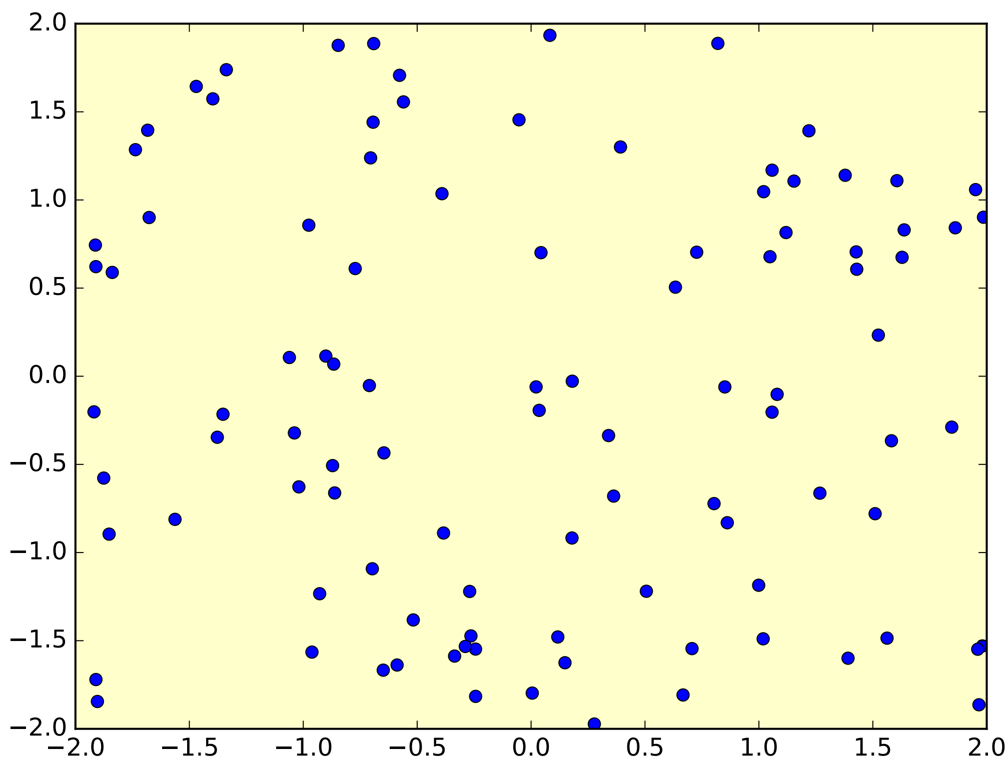
horizOn Controls the visibility of the horizontal line

vertOn Controls the visibility of the horizontal line

and the visibility of the cursor itself with the *visible* attribute.

For the cursor to remain responsive you must keep a reference to it.

Add a cursor to *ax*. If `useblit=True`, use the backend- dependent blitting features for faster updates (GTKAgg only for now). *lineprops* is a dictionary of line properties.



clear(*event*)
clear the cursor

onmove(*event*)
on mouse motion draw the cursor if visible

class `matplotlib.widgets.EllipseSelector`(*ax*, *onselect*, *drawtype='box'*, *minspanx=None*, *minspany=None*, *useblit=False*, *lineprops=None*, *rectprops=None*, *spancoords='data'*, *button=None*, *maxdist=10*, *marker_props=None*, *interactive=False*, *state_modifier_keys=None*)

Bases: `matplotlib.widgets.RectangleSelector`

Select an elliptical region of an axes.

For the cursor to remain responsive you must keep a reference to it.

Example usage:

```
from matplotlib.widgets import  EllipseSelector
from pylab import *

def onselect(eclick, erelease):
  'eclick and erelease are matplotlib events at press and release'
  print(' startposition : (%f, %f)' % (eclick.xdata, eclick.ydata))
  print(' endposition   : (%f, %f)' % (erelease.xdata, erelease.ydata))
  print(' used button   : ', eclick.button)

def toggle_selector(event):
    print(' Key pressed.')
    if event.key in ['Q', 'q'] and toggle_selector.ES.active:
        print(' EllipseSelector deactivated.')
        toggle_selector.RS.set_active(False)
    if event.key in ['A', 'a'] and not toggle_selector.ES.active:
        print(' EllipseSelector activated.')
        toggle_selector.ES.set_active(True)

x = arange(100)/(99.0)
y = sin(x)
fig = figure
ax = subplot(111)
ax.plot(x,y)

toggle_selector.ES = EllipseSelector(ax, onselect, drawtype='line')
connect('key_press_event', toggle_selector)
show()
```

Create a selector in *ax*. When a selection is made, clear the span and call onselect with:

```
onselect(pos_1, pos_2)
```

and clear the drawn box/line. The `pos_1` and `pos_2` are arrays of length 2 containing the x- and y-coordinate.

If *minspanx* is not *None* then events smaller than *minspanx* in x direction are ignored (it's the same for y).

The rectangle is drawn with *rectprops*; default:

```
rectprops = dict(facecolor='red', edgecolor = 'black',
                 alpha=0.2, fill=True)
```

The line is drawn with *lineprops*; default:

```
lineprops = dict(color='black', linestyle='-',
                 linewidth = 2, alpha=0.5)
```

Use *drawtype* if you want the mouse to draw a line, a box or nothing between click and actual position by setting

`drawtype = 'line'`, `drawtype='box'` or `drawtype = 'none'`.

spancoords is one of 'data' or 'pixels'. If 'data', *minspanx* and *minspanx* will be interpreted in the same coordinates as the x and y axis. If 'pixels', they are in pixels.

button is a list of integers indicating which mouse buttons should be used for rectangle selection. You can also specify a single integer if only a single button is desired. Default is *None*, which does not limit which button can be used.

Note, typically: 1 = left mouse button 2 = center mouse button (scroll wheel) 3 = right mouse button

interactive will draw a set of handles and allow you interact with the widget after it is drawn.

state_modifier_keys are keyboard modifiers that affect the behavior of the widget.

The defaults are: dict(move=' ', clear='escape', square='shift', center='ctrl')

Keyboard modifiers, which: 'move': Move the existing shape. 'clear': Clear the current shape. 'square': Makes the shape square. 'center': Make the initial point the center of the shape. 'square' and 'center' can be combined.

draw_shape(*extents*)

class matplotlib.widgets.**Lasso**(*ax*, *xy*, *callback=None*, *useblit=True*)

Bases: `matplotlib.widgets.AxesWidget`

Selection curve of an arbitrary shape.

The selected path can be used in conjunction with `contains_point()` to select data points from an image.

Unlike `LassoSelector`, this must be initialized with a starting point `xy`, and the `Lasso` events are destroyed upon release.

Parameters:

ax [`Axes`] The parent axes for the widget.

xy [array] Coordinates of the start of the lasso.

callback [function] Whenever the lasso is released, the `callback` function is called and passed the vertices of the selected path.

onmove(*event*)

onrelease(*event*)

class `matplotlib.widgets.LassoSelector`(*ax*, *onselect=None*, *useblit=True*, *lineprops=None*, *button=None*)

Bases: `matplotlib.widgets._SelectorWidget`

Selection curve of an arbitrary shape.

For the selector to remain responsive you must keep a reference to it.

The selected path can be used in conjunction with `contains_point()` to select data points from an image.

In contrast to `Lasso`, `LassoSelector` is written with an interface similar to `RectangleSelector` and `SpanSelector` and will continue to interact with the axes until disconnected.

Parameters:

ax [`Axes`] The parent axes for the widget.

onselect [function] Whenever the lasso is released, the `onselect` function is called and passed the vertices of the selected path.

Example usage:

```
    ax = subplot(111)
    ax.plot(x,y)

    def onselect(verts):
        print(verts)
    lasso = LassoSelector(ax, onselect)

*button* is a list of integers indicating which mouse buttons should
    be used for rectangle selection.  You can also specify a single
    integer if only a single button is desired.  Default is *None*,
    which does not limit which button can be used.

    Note, typically:
     1 = left mouse button
     2 = center mouse button (scroll wheel)
     3 = right mouse button
```

`onpress`(*event*)

`onrelease`(*event*)

class `matplotlib.widgets.LockDraw`

Bases: `object`

Some widgets, like the cursor, draw onto the canvas, and this is not desirable under all circumstances, like when the toolbar is in zoom-to-rect mode and drawing a rectangle. The module level "lock" allows someone to grab the lock and prevent other widgets from drawing. Use `matplotlib.widgets.lock(someobj)` to prevent other widgets from drawing while you're interacting with the canvas.

`available`(*o*)

drawing is available to *o*

isowner(*o*)
Return True if *o* owns this lock

locked()
Return True if the lock is currently held by an owner

release(*o*)
release the lock

class matplotlib.widgets.**MultiCursor**(*canvas*, *axes*, *useblit=True*, *horizOn=False*, *vertOn=True*, ***lineprops*)
Bases: *matplotlib.widgets.Widget*

Provide a vertical (default) and/or horizontal line cursor shared between multiple axes.

For the cursor to remain responsive you must keep a reference to it.

Example usage:

```
from matplotlib.widgets import MultiCursor
from pylab import figure, show, np

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(4*np.pi*t)
fig = figure()
ax1 = fig.add_subplot(211)
ax1.plot(t, s1)

ax2 = fig.add_subplot(212, sharex=ax1)
ax2.plot(t, s2)

multi = MultiCursor(fig.canvas, (ax1, ax2), color='r', lw=1,
                    horizOn=False, vertOn=True)
show()
```

clear(*event*)
clear the cursor

connect()
connect events

disconnect()
disconnect events

onmove(*event*)

class matplotlib.widgets.**RadioButtons**(*ax*, *labels*, *active=0*, *activecolor='blue'*)
Bases: *matplotlib.widgets.AxesWidget*

A GUI neutral radio button.

For the buttons to remain responsive you must keep a reference to this object.

The following attributes are exposed:

ax The `matplotlib.axes.Axes` instance the buttons are in
activecolor The color of the button when clicked
labels A list of `matplotlib.text.Text` instances
circles A list of `matplotlib.patches.Circle` instances
value_selected A string listing the current value selected

Connect to the RadioButtons with the `on_clicked()` method

Add radio buttons to `matplotlib.axes.Axes` instance *ax*
labels A len(buttons) list of labels as strings
active The index into labels for the button that is active
activecolor The color of the button when clicked

disconnect(*cid*)
remove the observer with connection id *cid*

on_clicked(*func*)
When the button is clicked, call *func* with button label

A connection id is returned which can be used to disconnect

set_active(*index*)
Trigger which radio button to make active.
***index* is an index into the original label list** that this object was constructed with. Raise ValueError if the index is invalid.
Callbacks will be triggered if `eventson` is True.

class matplotlib.widgets.**RectangleSelector**(*ax*, *onselect*, *drawtype='box'*, *minspanx=None*, *minspany=None*, *useblit=False*, *lineprops=None*, *rectprops=None*, *spancoords='data'*, *button=None*, *maxdist=10*, *marker_props=None*, *interactive=False*, *state_modifier_keys=None*)

Bases: `matplotlib.widgets._SelectorWidget`

Select a rectangular region of an axes.

For the cursor to remain responsive you must keep a reference to it.

Example usage:

```
from matplotlib.widgets import  RectangleSelector
from pylab import *

def onselect(eclick, erelease):
  'eclick and erelease are matplotlib events at press and release'
  print(' startposition : (%f, %f)' % (eclick.xdata, eclick.ydata))
  print(' endposition   : (%f, %f)' % (erelease.xdata, erelease.ydata))
  print(' used button   : ', eclick.button)

def toggle_selector(event):
    print(' Key pressed.')
    if event.key in ['Q', 'q'] and toggle_selector.RS.active:
        print(' RectangleSelector deactivated.')
        toggle_selector.RS.set_active(False)
    if event.key in ['A', 'a'] and not toggle_selector.RS.active:
```

```
        print(' RectangleSelector activated.')
        toggle_selector.RS.set_active(True)

x = arange(100)/(99.0)
y = sin(x)
fig = figure
ax = subplot(111)
ax.plot(x,y)

toggle_selector.RS = RectangleSelector(ax, onselect, drawtype='line')
connect('key_press_event', toggle_selector)
show()
```

Create a selector in *ax*. When a selection is made, clear the span and call onselect with:

```
onselect(pos_1, pos_2)
```

and clear the drawn box/line. The `pos_1` and `pos_2` are arrays of length 2 containing the x- and y-coordinate.

If *minspanx* is not *None* then events smaller than *minspanx* in x direction are ignored (it's the same for y).

The rectangle is drawn with *rectprops*; default:

```
rectprops = dict(facecolor='red', edgecolor = 'black',
                 alpha=0.2, fill=True)
```

The line is drawn with *lineprops*; default:

```
lineprops = dict(color='black', linestyle='-',
                 linewidth = 2, alpha=0.5)
```

Use *drawtype* if you want the mouse to draw a line, a box or nothing between click and actual position by setting

`drawtype = 'line'`, `drawtype='box'` or `drawtype = 'none'`.

spancoords is one of 'data' or 'pixels'. If 'data', *minspanx* and *minspanx* will be interpreted in the same coordinates as the x and y axis. If 'pixels', they are in pixels.

button is a list of integers indicating which mouse buttons should be used for rectangle selection. You can also specify a single integer if only a single button is desired. Default is *None*, which does not limit which button can be used.
Note, typically: 1 = left mouse button 2 = center mouse button (scroll wheel) 3 = right mouse button
interactive will draw a set of handles and allow you interact with the widget after it is drawn.

state_modifier_keys are keyboard modifiers that affect the behavior of the widget.

The defaults are: dict(move=' ', clear='escape', square='shift', center='ctrl')

Keyboard modifiers, which: 'move': Move the existing shape. 'clear': Clear the current shape. 'square': Makes the shape square. 'center': Make the initial point the center of the shape. 'square' and 'center' can be combined.

center
Center of rectangle

corners
Corners of rectangle from lower left, moving clockwise.

draw_shape(*extents*)

edge_centers
Midpoint of rectangle edges from left, moving clockwise.

extents
Return (xmin, xmax, ymin, ymax).

geometry

class matplotlib.widgets.**Slider**(*ax*, *label*, *valmin*, *valmax*, *valinit=0.5*, *valfmt='%1.2f'*, *closedmin=True*, *closedmax=True*, *slidermin=None*, *slidermax=None*, *dragging=True*, ***kwargs*)

Bases: `matplotlib.widgets.AxesWidget`

A slider representing a floating point range.

For the slider to remain responsive you must maintain a reference to it.

The following attributes are defined *ax* : the slider `matplotlib.axes.Axes` instance

val : the current slider value
vline [a `matplotlib.lines.Line2D` instance] representing the initial value of the slider
poly [A `matplotlib.patches.Polygon` instance] which is the slider knob
valfmt : the format string for formatting the slider text
label [a `matplotlib.text.Text` instance] for the slider label
closedmin : whether the slider is closed on the minimum

closedmax : whether the slider is closed on the maximum
slidermin [another slider - if not *None*, this slider must be] greater than *slidermin*
slidermax [another slider - if not *None*, this slider must be] less than *slidermax*
dragging : allow for mouse dragging on slider

Call `on_changed()` to connect to the slider event

Create a slider from *valmin* to *valmax* in axes *ax*.

Additional kwargs are passed on to `self.poly` which is the `matplotlib.patches.Rectangle` that draws the slider knob. See the `matplotlib.patches.Rectangle` documentation for valid property names (e.g., *facecolor*, *edgecolor*, *alpha*, ...).

Parameters ax : Axes
The Axes to put the slider in

label : str
Slider label

valmin : float
The minimum value of the slider

valmax : float
The maximum value of the slider

valinit : float
The slider initial position

label : str
The slider label

valfmt : str
Used to format the slider value, fprint format string

closedmin : bool
Indicate whether the slider interval is closed on the bottom

closedmax : bool
Indicate whether the slider interval is closed on the top

slidermin : Slider or None
Do not allow the current slider to have a value less than `slidermin`

slidermax : Slider or None
Do not allow the current slider to have a value greater than `slidermax`

dragging : bool
if the slider can be dragged by the mouse

disconnect(*cid*)
remove the observer with connection id *cid*

on_changed(*func*)
When the slider value is changed, call *func* with the new slider position

A connection id is returned which can be used to disconnect

reset()
reset the slider to the initial value if needed

set_val(*val*)

class `matplotlib.widgets.`**SpanSelector**(*ax*, *onselect*, *direction*, *minspan=None*, *useblit=False*, *rectprops=None*, *onmove_callback=None*, *span_stays=False*, *button=None*)

Bases: `matplotlib.widgets._SelectorWidget`

Select a min/max range of the x or y axes for a matplotlib Axes.

For the selector to remain responsive you must keep a reference to it.

Example usage:

```
ax = subplot(111)
ax.plot(x,y)

def onselect(vmin, vmax):
    print(vmin, vmax)
span = SpanSelector(ax, onselect, 'horizontal')
```

onmove_callback is an optional callback that is called on mouse move within the span range

Create a span selector in *ax*. When a selection is made, clear the span and call *onselect* with:

```
onselect(vmin, vmax)
```

and clear the span.

direction must be 'horizontal' or 'vertical'

If *minspan* is not *None*, ignore events smaller than *minspan*

The span rectangle is drawn with *rectprops*; default:

```
rectprops = dict(facecolor='red', alpha=0.5)
```

Set the visible attribute to *False* if you want to turn off the functionality of the span selector

If *span_stays* is True, the span stays visble after making a valid selection.

button is a list of integers indicating which mouse buttons should be used for selection. You can also specify a single integer if only a single button is desired. Default is *None*, which does not limit which button can be used.

Note, typically: 1 = left mouse button 2 = center mouse button (scroll wheel) 3 = right mouse button

ignore(*event*)
    return *True* if *event* should be ignored

new_axes(*ax*)

class matplotlib.widgets.**SubplotTool**(*targetfig*, *toolfig*)
    Bases: `matplotlib.widgets.Widget`

A tool to adjust the subplot params of a `matplotlib.figure.Figure`.

targetfig The figure instance to adjust.

toolfig The figure instance to embed the subplot tool into. If *None*, a default figure will be created. If you are using this from the GUI

funcbottom(*val*)

funchspace(*val*)

funcleft(*val*)

funcright(*val*)

functop(*val*)

funcwspace(*val*)

class matplotlib.widgets.**ToolHandles**(*ax*, *x*, *y*, *marker='o'*, *marker_props=None*, *useblit=True*)
    Bases: `object`

Control handles for canvas tools.

Parameters ax : `matplotlib.axes.Axes`

Matplotlib axes where tool handles are displayed.

x, y : 1D arrays

Coordinates of control handles.

marker : str

Shape of marker used to display handle. See `matplotlib.pyplot.plot`.

marker_props : dict

Additional marker properties. See `matplotlib.lines.Line2D`.

closest(*x*, *y*)

Return index and pixel distance to closest index.

set_animated(*val*)

set_data(*pts*, *y=None*)

Set x and y positions of handles

set_visible(*val*)

x

y

class `matplotlib.widgets.`**Widget**

Bases: `object`

Abstract base class for GUI neutral widgets

active

Is the widget active?

drawon = True

eventson = True

get_active()

Get whether the widget is active.

ignore(*event*)

Return True if event should be ignored.

This method (or a version of it) should be called at the beginning of any event callback.

set_active(*active*)

Set whether the widget is active.

Part X

Matplotlib Examples

CHAPTER

EIGHTY

ANIMATION EXAMPLES

80.1 animation example code: animate_decay.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation


def data_gen(t=0):
    cnt = 0
    while cnt < 1000:
        cnt += 1
        t += 0.1
        yield t, np.sin(2*np.pi*t) * np.exp(-t/10.)


def init():
    ax.set_ylim(-1.1, 1.1)
    ax.set_xlim(0, 10)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
    return line,

fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.grid()
xdata, ydata = [], []


def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()

    if t >= xmax:
```

```
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)

    return line,

ani = animation.FuncAnimation(fig, run, data_gen, blit=False, interval=10,
                              repeat=False, init_func=init)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.2 animation example code: basic_example.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation


def update_line(num, data, line):
    line.set_data(data[..., :num])
    return line,

fig1 = plt.figure()

data = np.random.rand(2, 25)
l, = plt.plot([], [], 'r-')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel('x')
plt.title('test')
line_ani = animation.FuncAnimation(fig1, update_line, 25, fargs=(data, l),
                                   interval=50, blit=True)
#line_ani.save('lines.mp4')

fig2 = plt.figure()

x = np.arange(-9, 10)
y = np.arange(-9, 10).reshape(-1, 1)
base = np.hypot(x, y)
ims = []
for add in np.arange(15):
    ims.append((plt.pcolor(x, y, base + add, norm=plt.Normalize(0, 30)),))

im_ani = animation.ArtistAnimation(fig2, ims, interval=50, repeat_delay=3000,
                                   blit=True)
#im_ani.save('im.mp4', metadata={'artist':'Guido'})

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.3 animation example code: basic_example_writer.py

[source code]

```
# Same as basic_example, but writes files using a single MovieWriter instance
# without putting on screen
# -*- noplot -*-
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import matplotlib.animation as animation


def update_line(num, data, line):
    line.set_data(data[..., :num])
    return line,

# Set up formatting for the movie files
Writer = animation.writers['ffmpeg']
writer = Writer(fps=15, metadata=dict(artist='Me'), bitrate=1800)


fig1 = plt.figure()

data = np.random.rand(2, 25)
l, = plt.plot([], [], 'r-')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel('x')
plt.title('test')
line_ani = animation.FuncAnimation(fig1, update_line, 25, fargs=(data, l),
                                   interval=50, blit=True)
line_ani.save('lines.mp4', writer=writer)

fig2 = plt.figure()

x = np.arange(-9, 10)
y = np.arange(-9, 10).reshape(-1, 1)
base = np.hypot(x, y)
ims = []
for add in np.arange(15):
    ims.append((plt.pcolor(x, y, base + add, norm=plt.Normalize(0, 30)),))

im_ani = animation.ArtistAnimation(fig2, ims, interval=50, repeat_delay=3000,
                                   blit=True)
im_ani.save('im.mp4', writer=writer)
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.4 animation example code: bayes_update.py

[source code]

```
# update a distribution based on new data.
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as ss
from matplotlib.animation import FuncAnimation


class UpdateDist(object):
    def __init__(self, ax, prob=0.5):
        self.success = 0
        self.prob = prob
        self.line, = ax.plot([], [], 'k-')
        self.x = np.linspace(0, 1, 200)
        self.ax = ax

        # Set up plot parameters
        self.ax.set_xlim(0, 1)
        self.ax.set_ylim(0, 15)
        self.ax.grid(True)

        # This vertical line represents the theoretical value, to
        # which the plotted distribution should converge.
        self.ax.axvline(prob, linestyle='--', color='black')

    def init(self):
        self.success = 0
        self.line.set_data([], [])
        return self.line,

    def __call__(self, i):
        # This way the plot can continuously run and we just keep
        # watching new realizations of the process
        if i == 0:
            return self.init()

        # Choose success based on exceed a threshold with a uniform pick
        if np.random.rand(1,) < self.prob:
            self.success += 1
        y = ss.beta.pdf(self.x, self.success + 1, (i - self.success) + 1)
        self.line.set_data(self.x, y)
        return self.line,

fig, ax = plt.subplots()
ud = UpdateDist(ax, prob=0.7)
anim = FuncAnimation(fig, ud, frames=np.arange(100), init_func=ud.init,
                     interval=100, blit=True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.5 animation example code: double_pendulum_animated.py

[source code]

```
# Double pendulum formula translated from the C code at
# http://www.physics.usyd.edu.au/~wheat/dpend_html/solve_dpend.c

from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation

G = 9.8  # acceleration due to gravity, in m/s^2
L1 = 1.0  # length of pendulum 1 in m
L2 = 1.0  # length of pendulum 2 in m
M1 = 1.0  # mass of pendulum 1 in kg
M2 = 1.0  # mass of pendulum 2 in kg


def derivs(state, t):

    dydx = np.zeros_like(state)
    dydx[0] = state[1]

    del_ = state[2] - state[0]
    den1 = (M1 + M2)*L1 - M2*L1*cos(del_)*cos(del_)
    dydx[1] = (M2*L1*state[1]*state[1]*sin(del_)*cos(del_) +
               M2*G*sin(state[2])*cos(del_) +
               M2*L2*state[3]*state[3]*sin(del_) -
               (M1 + M2)*G*sin(state[0]))/den1

    dydx[2] = state[3]

    den2 = (L2/L1)*den1
    dydx[3] = (-M2*L2*state[3]*state[3]*sin(del_)*cos(del_) +
               (M1 + M2)*G*sin(state[0])*cos(del_) -
               (M1 + M2)*L1*state[1]*state[1]*sin(del_) -
               (M1 + M2)*G*sin(state[2]))/den2

    return dydx

# create a time array from 0..100 sampled at 0.05 second steps
dt = 0.05
t = np.arange(0.0, 20, dt)

# th1 and th2 are the initial angles (degrees)
# w10 and w20 are the initial angular velocities (degrees per second)
th1 = 120.0
w1 = 0.0
th2 = -10.0
w2 = 0.0
```

```
# initial state
state = np.radians([th1, w1, th2, w2])

# integrate your ODE using scipy.integrate.
y = integrate.odeint(derivs, state, t)

x1 = L1*sin(y[:, 0])
y1 = -L1*cos(y[:, 0])

x2 = L2*sin(y[:, 2]) + x1
y2 = -L2*cos(y[:, 2]) + y1

fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 2))
ax.grid()

line, = ax.plot([], [], 'o-', lw=2)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)


def init():
    line.set_data([], [])
    time_text.set_text('')
    return line, time_text


def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    line.set_data(thisx, thisy)
    time_text.set_text(time_template % (i*dt))
    return line, time_text

ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)),
                              interval=25, blit=True, init_func=init)

#ani.save('double_pendulum.mp4', fps=15)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.6 animation example code: dynamic_image.py

[source code]

```
#!/usr/bin/env python
"""
An animated image
"""
```

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()


def f(x, y):
    return np.sin(x) + np.cos(y)

x = np.linspace(0, 2 * np.pi, 120)
y = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)

im = plt.imshow(f(x, y), cmap=plt.get_cmap('viridis'), animated=True)


def updatefig(*args):
    global x, y
    x += np.pi / 15.
    y += np.pi / 20.
    im.set_array(f(x, y))
    return im,

ani = animation.FuncAnimation(fig, updatefig, interval=50, blit=True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.7 animation example code: dynamic_image2.py

[source code]

```
#!/usr/bin/env python
"""
An animated image
"""
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()


def f(x, y):
    return np.sin(x) + np.cos(y)

x = np.linspace(0, 2 * np.pi, 120)
y = np.linspace(0, 2 * np.pi, 100).reshape(-1, 1)
# ims is a list of lists, each row is a list of artists to draw in the
# current frame; here we are just animating one artist, the image, in
# each frame
```

```
ims = []
for i in range(60):
    x += np.pi / 15.
    y += np.pi / 20.
    im = plt.imshow(f(x, y), cmap='viridis', animated=True)
    ims.append([im])

ani = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                                repeat_delay=1000)

#ani.save('dynamic_images.mp4')


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.8 animation example code: histogram.py

[source code]

```
"""
This example shows how to use a path patch to draw a bunch of
rectangles for an animated histogram
"""
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.path as path
import matplotlib.animation as animation

fig, ax = plt.subplots()

# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 100)

# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n
nrects = len(left)

# here comes the tricky part -- we have to set up the vertex and path
# codes arrays using moveto, lineto and closepoly

# for each rect: 1 for the MOVETO, 3 for the LINETO, 1 for the
# CLOSEPOLY; the vert for the closepoly is ignored but we still need
# it to keep the codes aligned with the vertices
```

```
nverts = nrects*(1 + 3 + 1)
verts = np.zeros((nverts, 2))
codes = np.ones(nverts, int) * path.Path.LINETO
codes[0::5] = path.Path.MOVETO
codes[4::5] = path.Path.CLOSEPOLY
verts[0::5, 0] = left
verts[0::5, 1] = bottom
verts[1::5, 0] = left
verts[1::5, 1] = top
verts[2::5, 0] = right
verts[2::5, 1] = top
verts[3::5, 0] = right
verts[3::5, 1] = bottom

barpath = path.Path(verts, codes)
patch = patches.PathPatch(
    barpath, facecolor='green', edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)

ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())


def animate(i):
    # simulate new data coming in
    data = np.random.randn(1000)
    n, bins = np.histogram(data, 100)
    top = bottom + n
    verts[1::5, 1] = top
    verts[2::5, 1] = top
    return [patch, ]

ani = animation.FuncAnimation(fig, animate, 100, repeat=False, blit=True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.9 animation example code: moviewriter.py

[source code]

```
# This example uses a MovieWriter directly to grab individual frames and
# write them to a file. This avoids any event loop integration, but has
# the advantage of working with even the Agg backend. This is not recommended
# for use in an interactive setting.
# -*- noplot -*-

import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
```

```
import matplotlib.animation as manimation

FFMpegWriter = manimation.writers['ffmpeg']
metadata = dict(title='Movie Test', artist='Matplotlib',
                comment='Movie support!')
writer = FFMpegWriter(fps=15, metadata=metadata)

fig = plt.figure()
l, = plt.plot([], [], 'k-o')

plt.xlim(-5, 5)
plt.ylim(-5, 5)

x0, y0 = 0, 0

with writer.saving(fig, "writer_test.mp4", 100):
    for i in range(100):
        x0 += 0.1 * np.random.randn()
        y0 += 0.1 * np.random.randn()
        l.set_data(x0, y0)
        writer.grab_frame()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.10 animation example code: rain.py

[source code]

```
"""
Rain simulation

Simulates rain drops on a surface by animating the scale and opacity
of 50 scatter points.

Author: Nicolas P. Rougier
"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation


# Create new Figure and an Axes which fills it.
fig = plt.figure(figsize=(7, 7))
ax = fig.add_axes([0, 0, 1, 1], frameon=False)
ax.set_xlim(0, 1), ax.set_xticks([])
ax.set_ylim(0, 1), ax.set_yticks([])

# Create rain data
n_drops = 50
rain_drops = np.zeros(n_drops, dtype=[('position', float, 2),
                                      ('size',     float, 1),
```

```
                                        ('growth',   float, 1),
                                        ('color',    float, 4)])

# Initialize the raindrops in random positions and with
# random growth rates.
rain_drops['position'] = np.random.uniform(0, 1, (n_drops, 2))
rain_drops['growth'] = np.random.uniform(50, 200, n_drops)

# Construct the scatter which we will update during animation
# as the raindrops develop.
scat = ax.scatter(rain_drops['position'][:, 0], rain_drops['position'][:, 1],
                  s=rain_drops['size'], lw=0.5, edgecolors=rain_drops['color'],
                  facecolors='none')


def update(frame_number):
    # Get an index which we can use to re-spawn the oldest raindrop.
    current_index = frame_number % n_drops

    # Make all colors more transparent as time progresses.
    rain_drops['color'][:, 3] -= 1.0/len(rain_drops)
    rain_drops['color'][:, 3] = np.clip(rain_drops['color'][:, 3], 0, 1)

    # Make all circles bigger.
    rain_drops['size'] += rain_drops['growth']

    # Pick a new position for oldest rain drop, resetting its size,
    # color and growth factor.
    rain_drops['position'][current_index] = np.random.uniform(0, 1, 2)
    rain_drops['size'][current_index] = 5
    rain_drops['color'][current_index] = (0, 0, 0, 1)
    rain_drops['growth'][current_index] = np.random.uniform(50, 200)

    # Update the scatter collection, with the new colors, sizes and positions.
    scat.set_edgecolors(rain_drops['color'])
    scat.set_sizes(rain_drops['size'])
    scat.set_offsets(rain_drops['position'])


# Construct the animation, using the update function as the animation
# director.
animation = FuncAnimation(fig, update, interval=10)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.11 animation example code: random_data.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig, ax = plt.subplots()
line, = ax.plot(np.random.rand(10))
ax.set_ylim(0, 1)


def update(data):
    line.set_ydata(data)
    return line,


def data_gen():
    while True:
        yield np.random.rand(10)

ani = animation.FuncAnimation(fig, update, data_gen, interval=100)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.12 animation example code: simple_3danim.py

[source code]

```
"""
A simple example of an animated plot... In 3D!
"""
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation


def Gen_RandLine(length, dims=2):
    """
    Create a line using a random walk algorithm

    length is the number of points for the line.
    dims is the number of dimensions the line has.
    """
    lineData = np.empty((dims, length))
    lineData[:, 0] = np.random.rand(dims)
    for index in range(1, length):
        # scaling the random numbers by 0.1 so
        # movement is small compared to position.
        # subtraction by 0.5 is to change the range to [-0.5, 0.5]
        # to allow a line to move backwards.
        step = ((np.random.rand(dims) - 0.5) * 0.1)
```

```
        lineData[:, index] = lineData[:, index - 1] + step

    return lineData


def update_lines(num, dataLines, lines):
    for line, data in zip(lines, dataLines):
        # NOTE: there is no .set_data() for 3 dim data...
        line.set_data(data[0:2, :num])
        line.set_3d_properties(data[2, :num])
    return lines

# Attaching 3D axis to the figure
fig = plt.figure()
ax = p3.Axes3D(fig)

# Fifty lines of random 3-D lines
data = [Gen_RandLine(25, 3) for index in range(50)]

# Creating fifty line objects.
# NOTE: Can't pass empty arrays into 3d version of plot()
lines = [ax.plot(dat[0, 0:1], dat[1, 0:1], dat[2, 0:1])[0] for dat in data]

# Setting the axes properties
ax.set_xlim3d([0.0, 1.0])
ax.set_xlabel('X')

ax.set_ylim3d([0.0, 1.0])
ax.set_ylabel('Y')

ax.set_zlim3d([0.0, 1.0])
ax.set_zlabel('Z')

ax.set_title('3D Test')

# Creating the Animation object
line_ani = animation.FuncAnimation(fig, update_lines, 25, fargs=(data, lines),
                                   interval=50, blit=False)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.13 animation example code: simple_anim.py

[source code]

```
"""
A simple example of an animated plot
"""
import numpy as np
```

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig, ax = plt.subplots()

x = np.arange(0, 2*np.pi, 0.01)
line, = ax.plot(x, np.sin(x))


def animate(i):
    line.set_ydata(np.sin(x + i/10.0))  # update the data
    return line,


# Init only required for blitting to give a clean slate.
def init():
    line.set_ydata(np.ma.array(x, mask=True))
    return line,

ani = animation.FuncAnimation(fig, animate, np.arange(1, 200), init_func=init,
                              interval=25, blit=True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.14 animation example code: strip_chart_demo.py

[source code]

```
"""
Emulate an oscilloscope.  Requires the animation API introduced in
matplotlib 1.0 SVN.
"""
import numpy as np
from matplotlib.lines import Line2D
import matplotlib.pyplot as plt
import matplotlib.animation as animation


class Scope(object):
    def __init__(self, ax, maxt=2, dt=0.02):
        self.ax = ax
        self.dt = dt
        self.maxt = maxt
        self.tdata = [0]
        self.ydata = [0]
        self.line = Line2D(self.tdata, self.ydata)
        self.ax.add_line(self.line)
        self.ax.set_ylim(-.1, 1.1)
        self.ax.set_xlim(0, self.maxt)
```

```
    def update(self, y):
        lastt = self.tdata[-1]
        if lastt > self.tdata[0] + self.maxt:  # reset the arrays
            self.tdata = [self.tdata[-1]]
            self.ydata = [self.ydata[-1]]
            self.ax.set_xlim(self.tdata[0], self.tdata[0] + self.maxt)
            self.ax.figure.canvas.draw()

        t = self.tdata[-1] + self.dt
        self.tdata.append(t)
        self.ydata.append(y)
        self.line.set_data(self.tdata, self.ydata)
        return self.line,


def emitter(p=0.03):
    'return a random value with probability p, else 0'
    while True:
        v = np.random.rand(1)
        if v > p:
            yield 0.
        else:
            yield np.random.rand(1)

fig, ax = plt.subplots()
scope = Scope(ax)

# pass a generator in "emitter" to produce data for the update func
ani = animation.FuncAnimation(fig, scope.update, emitter, interval=10,
                              blit=True)


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.15 animation example code: subplots.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
import matplotlib.animation as animation


# This example uses subclassing, but there is no reason that the proper
# function couldn't be set up and then use FuncAnimation. The code is long, but
# not really complex. The length is due solely to the fact that there are a
# total of 9 lines that need to be changed for the animation as well as 3
# subplots that need initial set up.
```

```
class SubplotAnimation(animation.TimedAnimation):
    def __init__(self):
        fig = plt.figure()
        ax1 = fig.add_subplot(1, 2, 1)
        ax2 = fig.add_subplot(2, 2, 2)
        ax3 = fig.add_subplot(2, 2, 4)

        self.t = np.linspace(0, 80, 400)
        self.x = np.cos(2 * np.pi * self.t / 10.)
        self.y = np.sin(2 * np.pi * self.t / 10.)
        self.z = 10 * self.t

        ax1.set_xlabel('x')
        ax1.set_ylabel('y')
        self.line1 = Line2D([], [], color='black')
        self.line1a = Line2D([], [], color='red', linewidth=2)
        self.line1e = Line2D(
            [], [], color='red', marker='o', markeredgecolor='r')
        ax1.add_line(self.line1)
        ax1.add_line(self.line1a)
        ax1.add_line(self.line1e)
        ax1.set_xlim(-1, 1)
        ax1.set_ylim(-2, 2)
        ax1.set_aspect('equal', 'datalim')

        ax2.set_xlabel('y')
        ax2.set_ylabel('z')
        self.line2 = Line2D([], [], color='black')
        self.line2a = Line2D([], [], color='red', linewidth=2)
        self.line2e = Line2D(
            [], [], color='red', marker='o', markeredgecolor='r')
        ax2.add_line(self.line2)
        ax2.add_line(self.line2a)
        ax2.add_line(self.line2e)
        ax2.set_xlim(-1, 1)
        ax2.set_ylim(0, 800)

        ax3.set_xlabel('x')
        ax3.set_ylabel('z')
        self.line3 = Line2D([], [], color='black')
        self.line3a = Line2D([], [], color='red', linewidth=2)
        self.line3e = Line2D(
            [], [], color='red', marker='o', markeredgecolor='r')
        ax3.add_line(self.line3)
        ax3.add_line(self.line3a)
        ax3.add_line(self.line3e)
        ax3.set_xlim(-1, 1)
        ax3.set_ylim(0, 800)

        animation.TimedAnimation.__init__(self, fig, interval=50, blit=True)

    def _draw_frame(self, framedata):
        i = framedata
```

```
        head = i - 1
        head_len = 10
        head_slice = (self.t > self.t[i] - 1.0) & (self.t < self.t[i])

        self.line1.set_data(self.x[:i], self.y[:i])
        self.line1a.set_data(self.x[head_slice], self.y[head_slice])
        self.line1e.set_data(self.x[head], self.y[head])

        self.line2.set_data(self.y[:i], self.z[:i])
        self.line2a.set_data(self.y[head_slice], self.z[head_slice])
        self.line2e.set_data(self.y[head], self.z[head])

        self.line3.set_data(self.x[:i], self.z[:i])
        self.line3a.set_data(self.x[head_slice], self.z[head_slice])
        self.line3e.set_data(self.x[head], self.z[head])

        self._drawn_artists = [self.line1, self.line1a, self.line1e,
                               self.line2, self.line2a, self.line2e,
                               self.line3, self.line3a, self.line3e]

    def new_frame_seq(self):
        return iter(range(self.t.size))

    def _init_draw(self):
        lines = [self.line1, self.line1a, self.line1e,
                 self.line2, self.line2a, self.line2e,
                 self.line3, self.line3a, self.line3e]
        for l in lines:
            l.set_data([], [])

ani = SubplotAnimation()
#ani.save('test_sub.mp4')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

80.16 animation example code: unchained.py

[source code]

```
"""
Comparative path demonstration of frequency from a fake signal of a pulsar.
(mostly known because of the cover for Joy Division's Unknown Pleasures)

Author: Nicolas P. Rougier
"""
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Create new Figure with black background
```

```
fig = plt.figure(figsize=(8, 8), facecolor='black')

# Add a subplot with no frame
ax = plt.subplot(111, frameon=False)

# Generate random data
data = np.random.uniform(0, 1, (64, 75))
X = np.linspace(-1, 1, data.shape[-1])
G = 1.5 * np.exp(-4 * X * X)

# Generate line plots
lines = []
for i in range(len(data)):
    # Small reduction of the X extents to get a cheap perspective effect
    xscale = 1 - i / 200.
    # Same for linewidth (thicker strokes on bottom)
    lw = 1.5 - i / 100.0
    line, = ax.plot(xscale * X, i + G * data[i], color="w", lw=lw)
    lines.append(line)

# Set y limit (or first line is cropped because of thickness)
ax.set_ylim(-1, 70)

# No ticks
ax.set_xticks([])
ax.set_yticks([])

# 2 part titles to get different font weights
ax.text(0.5, 1.0, "MATPLOTLIB ", transform=ax.transAxes,
        ha="right", va="bottom", color="w",
        family="sans-serif", fontweight="light", fontsize=16)
ax.text(0.5, 1.0, "UNCHAINED", transform=ax.transAxes,
        ha="left", va="bottom", color="w",
        family="sans-serif", fontweight="bold", fontsize=16)


def update(*args):
    # Shift all data to the right
    data[:, 1:] = data[:, :-1]

    # Fill-in new values
    data[:, 0] = np.random.uniform(0, 1, len(data))

    # Update data
    for i in range(len(data)):
        lines[i].set_ydata(i + G * data[i])

    # Return modified artists
    return lines

# Construct the animation, using the update function as the animation
# director.
anim = animation.FuncAnimation(fig, update, interval=10)
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER EIGHTYONE

API EXAMPLES

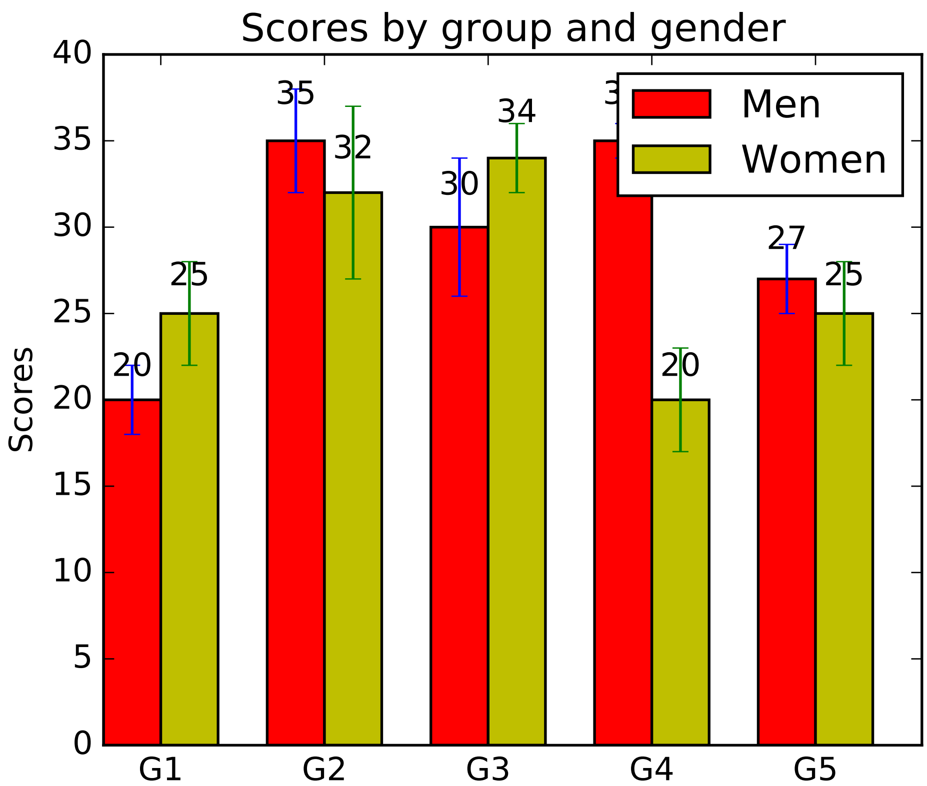
81.1 api example code: agg_oo.py

[source code]

```
#!/usr/bin/env python
# -*- noplot -*-
"""
A pure OO (look Ma, no pylab!) example using the agg backend
"""
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.plot([1, 2, 3])
ax.set_title('hi mom')
ax.grid(True)
ax.set_xlabel('time')
ax.set_ylabel('volts')
canvas.print_figure('test')
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.2 api example code: barchart_demo.py

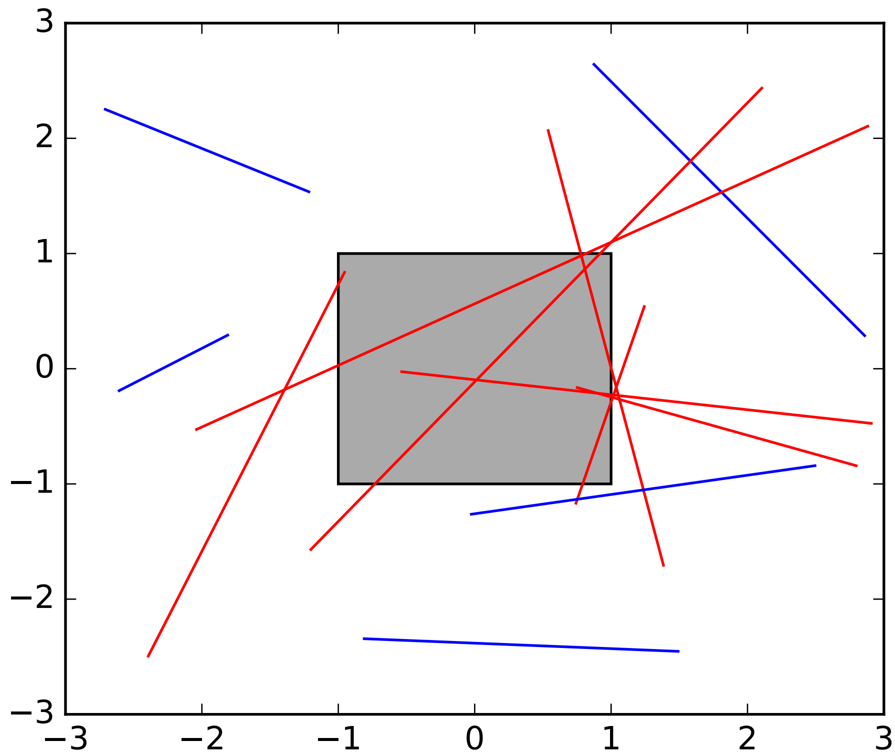



```
#!/usr/bin/env python
# a bar plot with errorbars
import numpy as np
import matplotlib.pyplot as plt

N = 5
menMeans = (20, 35, 30, 35, 27)
menStd = (2, 3, 4, 1, 2)

ind = np.arange(N)  # the x locations for the groups
width = 0.35       # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(ind, menMeans, width, color='r', yerr=menStd)

womenMeans = (25, 32, 34, 20, 25)
womenStd = (3, 5, 2, 3, 3)
rects2 = ax.bar(ind + width, womenMeans, width, color='y', yerr=womenStd)

# add some text for labels, title and axes ticks
ax.set_ylabel('Scores')
ax.set_title('Scores by group and gender')
```

```
ax.set_xticks(ind + width)
ax.set_xticklabels(('G1', 'G2', 'G3', 'G4', 'G5'))

ax.legend((rects1[0], rects2[0]), ('Men', 'Women'))


def autolabel(rects):
    # attach some text labels
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width()/2., 1.05*height,
                '%d' % int(height),
                ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

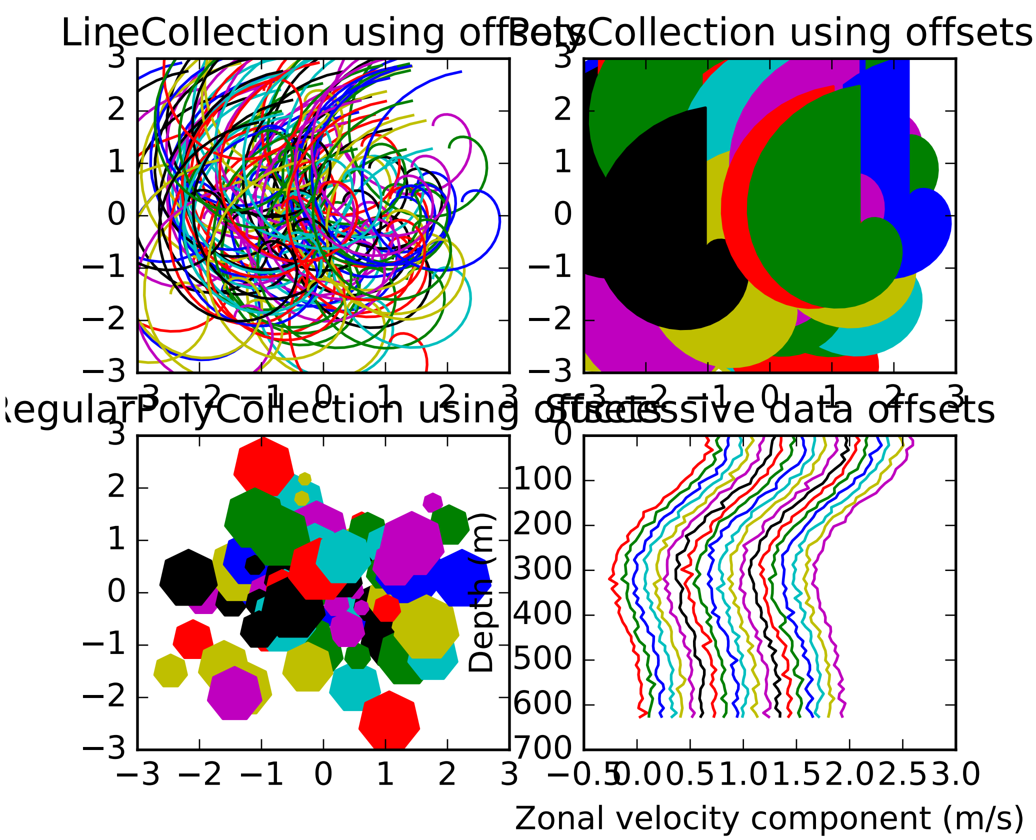
81.3 api example code: bbox_intersect.py



```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.transforms import Bbox
from matplotlib.path import Path

rect = plt.Rectangle((-1, -1), 2, 2, facecolor="#aaaaaa")
plt.gca().add_patch(rect)
bbox = Bbox.from_bounds(-1, -1, 2, 2)

for i in range(12):
    vertices = (np.random.random((2, 2)) - 0.5) * 6.0
    path = Path(vertices)
    if path.intersects_bbox(bbox):
        color = 'r'
    else:
        color = 'b'
    plt.plot(vertices[:, 0], vertices[:, 1], color=color)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.4 api example code: collections_demo.py



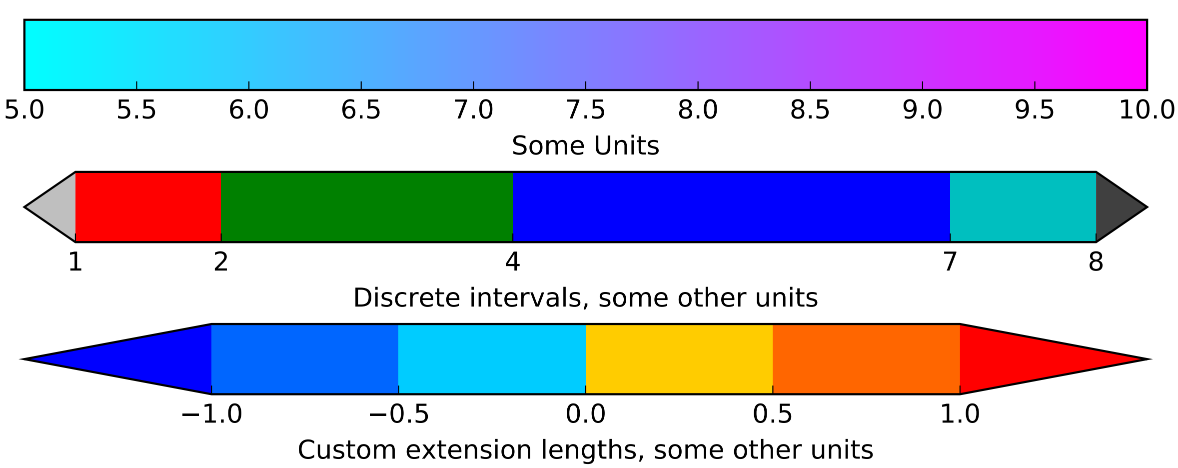
```
#!/usr/bin/env python
'''Demonstration of LineCollection, PolyCollection, and
RegularPolyCollection with autoscaling.

For the first two subplots, we will use spirals.  Their
size will be set in plot units, not data units.  Their positions
will be set in data units by using the "offsets" and "transOffset"
kwargs of the LineCollection and PolyCollection.

The third subplot will make regular polygons, with the same
type of scaling and positioning as in the first two.

The last subplot illustrates the use of "offsets=(xo,yo)",
that is, a single tuple instead of a list of tuples, to generate
successively offset curves, with the offset given in data
units.  This behavior is available only for the LineCollection.

'''

import matplotlib.pyplot as plt
from matplotlib import collections, transforms
from matplotlib.colors import colorConverter
import numpy as np

nverts = 50
npts = 100

# Make some spirals
r = np.arange(nverts)
theta = np.linspace(0, 2*np.pi, nverts)
xx = r * np.sin(theta)
yy = r * np.cos(theta)
spiral = list(zip(xx, yy))

# Make some offsets
rs = np.random.RandomState([12345678])
xo = rs.randn(npts)
yo = rs.randn(npts)
xyo = list(zip(xo, yo))

# Make a list of colors cycling through the rgbcmyk series.
colors = [colorConverter.to_rgba(c)
          for c in ('r', 'g', 'b', 'c', 'y', 'm', 'k')]

fig, axes = plt.subplots(2, 2)
((ax1, ax2), (ax3, ax4)) = axes  # unpack the axes


col = collections.LineCollection([spiral], offsets=xyo,
                                 transOffset=ax1.transData)
trans = fig.dpi_scale_trans + transforms.Affine2D().scale(1.0/72.0)
col.set_transform(trans)  # the points to pixels transform
# Note: the first argument to the collection initializer
```

```
# must be a list of sequences of x,y tuples; we have only
# one sequence, but we still have to put it in a list.
ax1.add_collection(col, autolim=True)
# autolim=True enables autoscaling.  For collections with
# offsets like this, it is neither efficient nor accurate,
# but it is good enough to generate a plot that you can use
# as a starting point.  If you know beforehand the range of
# x and y that you want to show, it is better to set them
# explicitly, leave out the autolim kwarg (or set it to False),
# and omit the 'ax1.autoscale_view()' call below.

# Make a transform for the line segments such that their size is
# given in points:
col.set_color(colors)

ax1.autoscale_view()  # See comment above, after ax1.add_collection.
ax1.set_title('LineCollection using offsets')


# The same data as above, but fill the curves.
col = collections.PolyCollection([spiral], offsets=xyo,
                                 transOffset=ax2.transData)
trans = transforms.Affine2D().scale(fig.dpi/72.0)
col.set_transform(trans)  # the points to pixels transform
ax2.add_collection(col, autolim=True)
col.set_color(colors)


ax2.autoscale_view()
ax2.set_title('PolyCollection using offsets')

# 7-sided regular polygons

col = collections.RegularPolyCollection(7,
                                        sizes=np.fabs(xx) * 10.0, offsets=xyo,
                                        transOffset=ax3.transData)
trans = transforms.Affine2D().scale(fig.dpi / 72.0)
col.set_transform(trans)  # the points to pixels transform
ax3.add_collection(col, autolim=True)
col.set_color(colors)
ax3.autoscale_view()
ax3.set_title('RegularPolyCollection using offsets')


# Simulate a series of ocean current profiles, successively
# offset by 0.1 m/s so that they form what is sometimes called
# a "waterfall" plot or a "stagger" plot.

nverts = 60
ncurves = 20
offs = (0.1, 0.0)

yy = np.linspace(0, 2*np.pi, nverts)
```

```
ym = np.amax(yy)
xx = (0.2 + (ym - yy)/ym)**2 * np.cos(yy - 0.4)*0.5
segs = []
for i in range(ncurves):
    xxx = xx + 0.02*rs.randn(nverts)
    curve = list(zip(xxx, yy*100))
    segs.append(curve)

col = collections.LineCollection(segs, offsets=offs)
ax4.add_collection(col, autolim=True)
col.set_color(colors)
ax4.autoscale_view()
ax4.set_title('Successive data offsets')
ax4.set_xlabel('Zonal velocity component (m/s)')
ax4.set_ylabel('Depth (m)')
# Reverse the y-axis so depth increases downward
ax4.set_ylim(ax4.get_ylim()[::-1])


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.5 api example code: colorbar_only.py



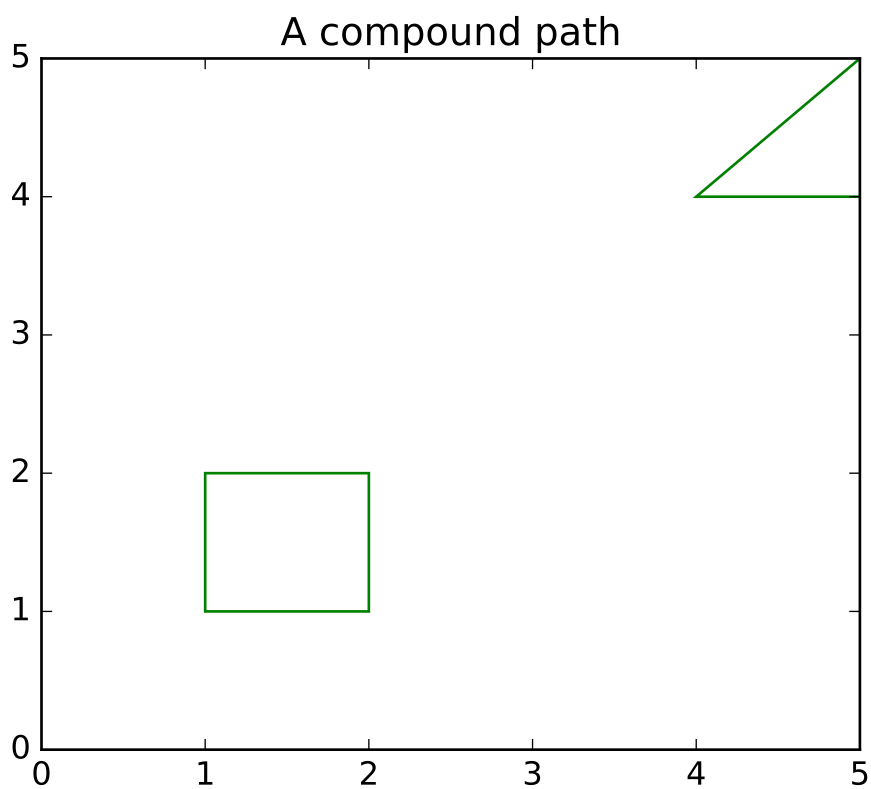

```
'''
Make a colorbar as a separate figure.
'''

from matplotlib import pyplot
import matplotlib as mpl

# Make a figure and axes with dimensions as desired.
fig = pyplot.figure(figsize=(8, 3))
ax1 = fig.add_axes([0.05, 0.80, 0.9, 0.15])
```

```
ax2 = fig.add_axes([0.05, 0.475, 0.9, 0.15])
ax3 = fig.add_axes([0.05, 0.15, 0.9, 0.15])

# Set the colormap and norm to correspond to the data for which
# the colorbar will be used.
cmap = mpl.cm.cool
norm = mpl.colors.Normalize(vmin=5, vmax=10)

# ColorbarBase derives from ScalarMappable and puts a colorbar
# in a specified axes, so it has everything needed for a
# standalone colorbar.  There are many more kwargs, but the
# following gives a basic continuous colorbar with ticks
# and labels.
cb1 = mpl.colorbar.ColorbarBase(ax1, cmap=cmap,
                                norm=norm,
                                orientation='horizontal')
cb1.set_label('Some Units')

# The second example illustrates the use of a ListedColormap, a
# BoundaryNorm, and extended ends to show the "over" and "under"
# value colors.
cmap = mpl.colors.ListedColormap(['r', 'g', 'b', 'c'])
cmap.set_over('0.25')
cmap.set_under('0.75')

# If a ListedColormap is used, the length of the bounds array must be
# one greater than the length of the color list.  The bounds must be
# monotonically increasing.
bounds = [1, 2, 4, 7, 8]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
cb2 = mpl.colorbar.ColorbarBase(ax2, cmap=cmap,
                                norm=norm,
                                # to use 'extend', you must
                                # specify two extra boundaries:
                                boundaries=[0] + bounds + [13],
                                extend='both',
                                ticks=bounds,  # optional
                                spacing='proportional',
                                orientation='horizontal')
cb2.set_label('Discrete intervals, some other units')

# The third example illustrates the use of custom length colorbar
# extensions, used on a colorbar with discrete intervals.
cmap = mpl.colors.ListedColormap([[0., .4, 1.], [0., .8, 1.],
                                  [1., .8, 0.], [1., .4, 0.]])
cmap.set_over((1., 0., 0.))
cmap.set_under((0., 0., 1.))

bounds = [-1., -.5, 0., .5, 1.]
norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
cb3 = mpl.colorbar.ColorbarBase(ax3, cmap=cmap,
                                norm=norm,
                                boundaries=[-10] + bounds + [10],
```

```
                                    extend='both',
                                    # Make the length of each extension
                                    # the same as the length of the
                                    # interior colors:
                                    extendfrac='auto',
                                    ticks=bounds,
                                    spacing='uniform',
                                    orientation='horizontal')
cb3.set_label('Custom extension lengths, some other units')

pyplot.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.6 api example code: compound_path.py

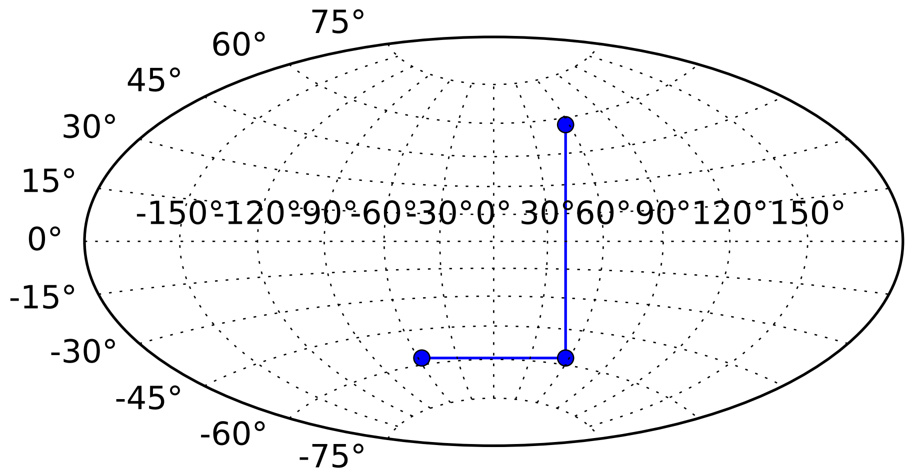



```
"""
Make a compund path -- in this case two simple polygons, a rectangle
and a triangle.  Use CLOSEOPOLY and MOVETO for the different parts of
the compound path
"""
import numpy as np
```

```
from matplotlib.path import Path
from matplotlib.patches import PathPatch
import matplotlib.pyplot as plt


vertices = []
codes = []

codes = [Path.MOVETO] + [Path.LINETO]*3 + [Path.CLOSEPOLY]
vertices = [(1, 1), (1, 2), (2, 2), (2, 1), (0, 0)]

codes += [Path.MOVETO] + [Path.LINETO]*2 + [Path.CLOSEPOLY]
vertices += [(4, 4), (5, 5), (5, 4), (0, 0)]

vertices = np.array(vertices, float)
path = Path(vertices, codes)

pathpatch = PathPatch(path, facecolor='None', edgecolor='green')

fig, ax = plt.subplots()
ax.add_patch(pathpatch)
ax.set_title('A compound path')

ax.dataLim.update_from_data_xy(vertices)
ax.autoscale_view()


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.7 api example code: custom_projection_example.py




```
from __future__ import unicode_literals

import matplotlib
from matplotlib.axes import Axes
from matplotlib.patches import Circle
from matplotlib.path import Path
from matplotlib.ticker import NullLocator, Formatter, FixedLocator
from matplotlib.transforms import Affine2D, BboxTransformTo, Transform
from matplotlib.projections import register_projection
import matplotlib.spines as mspines
import matplotlib.axis as maxis

import numpy as np

# This example projection class is rather long, but it is designed to
# illustrate many features, not all of which will be used every time.
# It is also common to factor out a lot of these methods into common
# code used by a number of projections with similar characteristics
# (see geo.py).


class HammerAxes(Axes):
```

```
    """
    A custom class for the Aitoff-Hammer projection, an equal-area map
    projection.

    http://en.wikipedia.org/wiki/Hammer_projection
    """
    # The projection must specify a name.  This will be used be the
    # user to select the projection, i.e. ``subplot(111,
    # projection='custom_hammer')``.
    name = 'custom_hammer'

    def __init__(self, *args, **kwargs):
        Axes.__init__(self, *args, **kwargs)
        self.set_aspect(0.5, adjustable='box', anchor='C')
        self.cla()

    def _init_axis(self):
        self.xaxis = maxis.XAxis(self)
        self.yaxis = maxis.YAxis(self)
        # Do not register xaxis or yaxis with spines -- as done in
        # Axes._init_axis() -- until HammerAxes.xaxis.cla() works.
        #self.spines['hammer'].register_axis(self.yaxis)
        self._update_transScale()

    def cla(self):
        """
        Override to set up some reasonable defaults.
        """
        # Don't forget to call the base class
        Axes.cla(self)

        # Set up a default grid spacing
        self.set_longitude_grid(30)
        self.set_latitude_grid(15)
        self.set_longitude_grid_ends(75)

        # Turn off minor ticking altogether
        self.xaxis.set_minor_locator(NullLocator())
        self.yaxis.set_minor_locator(NullLocator())

        # Do not display ticks -- we only want gridlines and text
        self.xaxis.set_ticks_position('none')
        self.yaxis.set_ticks_position('none')

        # The limits on this projection are fixed -- they are not to
        # be changed by the user.  This makes the math in the
        # transformation itself easier, and since this is a toy
        # example, the easier, the better.
        Axes.set_xlim(self, -np.pi, np.pi)
        Axes.set_ylim(self, -np.pi / 2.0, np.pi / 2.0)

    def _set_lim_and_transforms(self):
        """
```

```
        This is called once when the plot is created to set up all the
        transforms for the data, text and grids.
        """
        # There are three important coordinate spaces going on here:
        #
        #    1. Data space: The space of the data itself
        #
        #    2. Axes space: The unit rectangle (0, 0) to (1, 1)
        #       covering the entire plot area.
        #
        #    3. Display space: The coordinates of the resulting image,
        #       often in pixels or dpi/inch.

        # This function makes heavy use of the Transform classes in
        # ``lib/matplotlib/transforms.py.`` For more information, see
        # the inline documentation there.

        # The goal of the first two transformations is to get from the
        # data space (in this case longitude and latitude) to axes
        # space.  It is separated into a non-affine and affine part so
        # that the non-affine part does not have to be recomputed when
        # a simple affine change to the figure has been made (such as
        # resizing the window or changing the dpi).

        # 1) The core transformation from data space into
        # rectilinear space defined in the HammerTransform class.
        self.transProjection = self.HammerTransform()

        # 2) The above has an output range that is not in the unit
        # rectangle, so scale and translate it so it fits correctly
        # within the axes.  The peculiar calculations of xscale and
        # yscale are specific to a Aitoff-Hammer projection, so don't
        # worry about them too much.
        xscale = 2.0 * np.sqrt(2.0) * np.sin(0.5 * np.pi)
        yscale = np.sqrt(2.0) * np.sin(0.5 * np.pi)
        self.transAffine = Affine2D() \
            .scale(0.5 / xscale, 0.5 / yscale) \
            .translate(0.5, 0.5)

        # 3) This is the transformation from axes space to display
        # space.
        self.transAxes = BboxTransformTo(self.bbox)

        # Now put these 3 transforms together -- from data all the way
        # to display coordinates.  Using the '+' operator, these
        # transforms will be applied "in order".  The transforms are
        # automatically simplified, if possible, by the underlying
        # transformation framework.
        self.transData = \
            self.transProjection + \
            self.transAffine + \
            self.transAxes
```

```
        # The main data transformation is set up.  Now deal with
        # gridlines and tick labels.

        # Longitude gridlines and ticklabels.  The input to these
        # transforms are in display space in x and axes space in y.
        # Therefore, the input values will be in range (-xmin, 0),
        # (xmax, 1).  The goal of these transforms is to go from that
        # space to display space.  The tick labels will be offset 4
        # pixels from the equator.
        self._xaxis_pretransform = \
            Affine2D() \
            .scale(1.0, np.pi) \
            .translate(0.0, -np.pi)
        self._xaxis_transform = \
            self._xaxis_pretransform + \
            self.transData
        self._xaxis_text1_transform = \
            Affine2D().scale(1.0, 0.0) + \
            self.transData + \
            Affine2D().translate(0.0, 4.0)
        self._xaxis_text2_transform = \
            Affine2D().scale(1.0, 0.0) + \
            self.transData + \
            Affine2D().translate(0.0, -4.0)

        # Now set up the transforms for the latitude ticks.  The input to
        # these transforms are in axes space in x and display space in
        # y.  Therefore, the input values will be in range (0, -ymin),
        # (1, ymax).  The goal of these transforms is to go from that
        # space to display space.  The tick labels will be offset 4
        # pixels from the edge of the axes ellipse.
        yaxis_stretch = Affine2D().scale(2*np.pi, 1.0).translate(-np.pi, 0.0)
        yaxis_space = Affine2D().scale(1.0, 1.1)
        self._yaxis_transform = \
            yaxis_stretch + \
            self.transData
        yaxis_text_base = \
            yaxis_stretch + \
            self.transProjection + \
            (yaxis_space +
             self.transAffine +
             self.transAxes)
        self._yaxis_text1_transform = \
            yaxis_text_base + \
            Affine2D().translate(-8.0, 0.0)
        self._yaxis_text2_transform = \
            yaxis_text_base + \
            Affine2D().translate(8.0, 0.0)

    def get_xaxis_transform(self, which='grid'):
        """
        Override this method to provide a transformation for the
        x-axis grid and ticks.
```

```
        """
        assert which in ['tick1', 'tick2', 'grid']
        return self._xaxis_transform

    def get_xaxis_text1_transform(self, pixelPad):
        """
        Override this method to provide a transformation for the
        x-axis tick labels.

        Returns a tuple of the form (transform, valign, halign)
        """
        return self._xaxis_text1_transform, 'bottom', 'center'

    def get_xaxis_text2_transform(self, pixelPad):
        """
        Override this method to provide a transformation for the
        secondary x-axis tick labels.

        Returns a tuple of the form (transform, valign, halign)
        """
        return self._xaxis_text2_transform, 'top', 'center'

    def get_yaxis_transform(self, which='grid'):
        """
        Override this method to provide a transformation for the
        y-axis grid and ticks.
        """
        assert which in ['tick1', 'tick2', 'grid']
        return self._yaxis_transform

    def get_yaxis_text1_transform(self, pixelPad):
        """
        Override this method to provide a transformation for the
        y-axis tick labels.

        Returns a tuple of the form (transform, valign, halign)
        """
        return self._yaxis_text1_transform, 'center', 'right'

    def get_yaxis_text2_transform(self, pixelPad):
        """
        Override this method to provide a transformation for the
        secondary y-axis tick labels.

        Returns a tuple of the form (transform, valign, halign)
        """
        return self._yaxis_text2_transform, 'center', 'left'

    def _gen_axes_patch(self):
        """
        Override this method to define the shape that is used for the
        background of the plot.  It should be a subclass of Patch.
```

```
        In this case, it is a Circle (that may be warped by the axes
        transform into an ellipse).  Any data and gridlines will be
        clipped to this shape.
        """
        return Circle((0.5, 0.5), 0.5)

    def _gen_axes_spines(self):
        return {'custom_hammer': mspines.Spine.circular_spine(self,
                                                               (0.5, 0.5), 0.5)}

    # Prevent the user from applying scales to one or both of the
    # axes.  In this particular case, scaling the axes wouldn't make
    # sense, so we don't allow it.
    def set_xscale(self, *args, **kwargs):
        if args[0] != 'linear':
            raise NotImplementedError
        Axes.set_xscale(self, *args, **kwargs)

    def set_yscale(self, *args, **kwargs):
        if args[0] != 'linear':
            raise NotImplementedError
        Axes.set_yscale(self, *args, **kwargs)

    # Prevent the user from changing the axes limits.  In our case, we
    # want to display the whole sphere all the time, so we override
    # set_xlim and set_ylim to ignore any input.  This also applies to
    # interactive panning and zooming in the GUI interfaces.
    def set_xlim(self, *args, **kwargs):
        Axes.set_xlim(self, -np.pi, np.pi)
        Axes.set_ylim(self, -np.pi / 2.0, np.pi / 2.0)
    set_ylim = set_xlim

    def format_coord(self, lon, lat):
        """
        Override this method to change how the values are displayed in
        the status bar.

        In this case, we want them to be displayed in degrees N/S/E/W.
        """
        lon = np.degrees(lon)
        lat = np.degrees(lat)
        if lat >= 0.0:
            ns = 'N'
        else:
            ns = 'S'
        if lon >= 0.0:
            ew = 'E'
        else:
            ew = 'W'
        # \u00b0 : degree symbol
        return '%f\u00b0%s, %f\u00b0%s' % (abs(lat), ns, abs(lon), ew)

    class DegreeFormatter(Formatter):
```

```
        """
        This is a custom formatter that converts the native unit of
        radians into (truncated) degrees and adds a degree symbol.
        """

        def __init__(self, round_to=1.0):
            self._round_to = round_to

        def __call__(self, x, pos=None):
            degrees = round(np.degrees(x) / self._round_to) * self._round_to
            # \u00b0 : degree symbol
            return "%d\u00b0" % degrees

    def set_longitude_grid(self, degrees):
        """
        Set the number of degrees between each longitude grid.

        This is an example method that is specific to this projection
        class -- it provides a more convenient interface to set the
        ticking than set_xticks would.
        """
        # Set up a FixedLocator at each of the points, evenly spaced
        # by degrees.
        number = (360.0 / degrees) + 1
        self.xaxis.set_major_locator(
            plt.FixedLocator(
                np.linspace(-np.pi, np.pi, number, True)[1:-1]))
        # Set the formatter to display the tick labels in degrees,
        # rather than radians.
        self.xaxis.set_major_formatter(self.DegreeFormatter(degrees))

    def set_latitude_grid(self, degrees):
        """
        Set the number of degrees between each longitude grid.

        This is an example method that is specific to this projection
        class -- it provides a more convenient interface than
        set_yticks would.
        """
        # Set up a FixedLocator at each of the points, evenly spaced
        # by degrees.
        number = (180.0 / degrees) + 1
        self.yaxis.set_major_locator(
            FixedLocator(
                np.linspace(-np.pi / 2.0, np.pi / 2.0, number, True)[1:-1]))
        # Set the formatter to display the tick labels in degrees,
        # rather than radians.
        self.yaxis.set_major_formatter(self.DegreeFormatter(degrees))

    def set_longitude_grid_ends(self, degrees):
        """
        Set the latitude(s) at which to stop drawing the longitude grids.
```

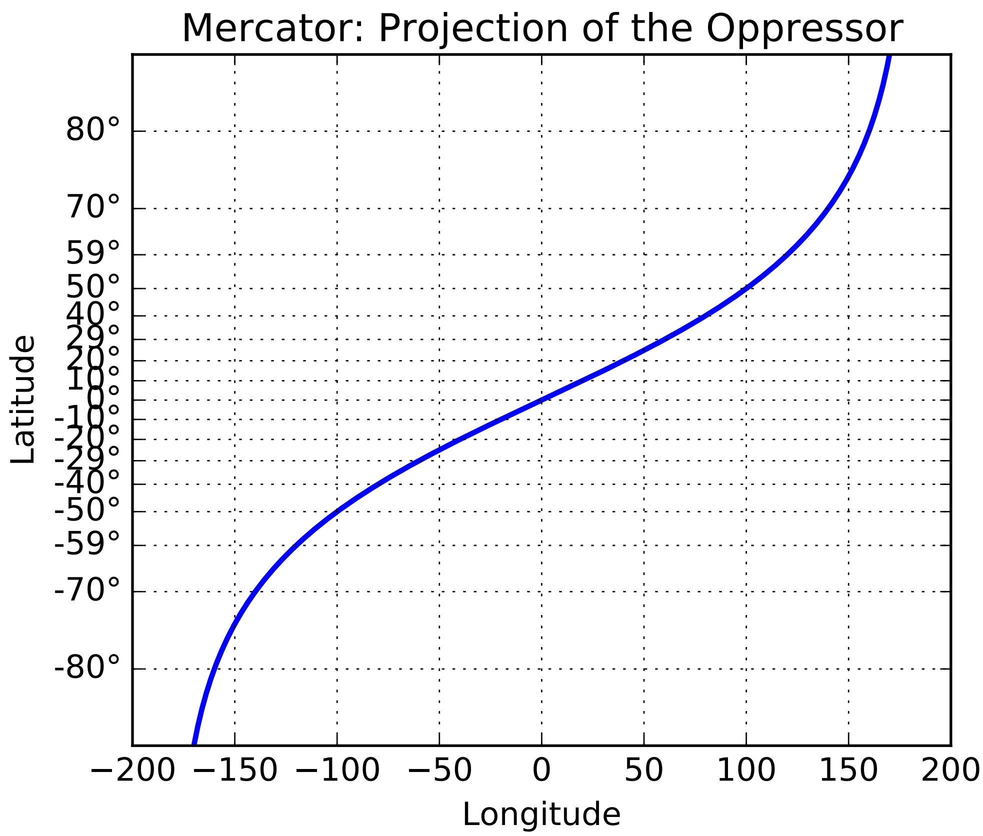
```
        Often, in geographic projections, you wouldn't want to draw
        longitude gridlines near the poles.  This allows the user to
        specify the degree at which to stop drawing longitude grids.

        This is an example method that is specific to this projection
        class -- it provides an interface to something that has no
        analogy in the base Axes class.
        """
        longitude_cap = np.radians(degrees)
        # Change the xaxis gridlines transform so that it draws from
        # -degrees to degrees, rather than -pi to pi.
        self._xaxis_pretransform \
            .clear() \
            .scale(1.0, longitude_cap * 2.0) \
            .translate(0.0, -longitude_cap)

    def get_data_ratio(self):
        """
        Return the aspect ratio of the data itself.

        This method should be overridden by any Axes that have a
        fixed data ratio.
        """
        return 1.0

    # Interactive panning and zooming is not supported with this projection,
    # so we override all of the following methods to disable it.
    def can_zoom(self):
        """
        Return True if this axes support the zoom box
        """
        return False

    def start_pan(self, x, y, button):
        pass

    def end_pan(self):
        pass

    def drag_pan(self, button, key, x, y):
        pass

    # Now, the transforms themselves.

    class HammerTransform(Transform):
        """
        The base Hammer transform.
        """
        input_dims = 2
        output_dims = 2
        is_separable = False

        def transform_non_affine(self, ll):
```

```
        """
        Override the transform_non_affine method to implement the custom
        transform.

        The input and output are Nx2 numpy arrays.
        """
        longitude = ll[:, 0:1]
        latitude = ll[:, 1:2]

        # Pre-compute some values
        half_long = longitude / 2.0
        cos_latitude = np.cos(latitude)
        sqrt2 = np.sqrt(2.0)

        alpha = 1.0 + cos_latitude * np.cos(half_long)
        x = (2.0 * sqrt2) * (cos_latitude * np.sin(half_long)) / alpha
        y = (sqrt2 * np.sin(latitude)) / alpha
        return np.concatenate((x, y), 1)

    # This is where things get interesting.  With this projection,
    # straight lines in data space become curves in display space.
    # This is done by interpolating new values between the input
    # values of the data.  Since ``transform`` must not return a
    # differently-sized array, any transform that requires
    # changing the length of the data array must happen within
    # ``transform_path``.
    def transform_path_non_affine(self, path):
        ipath = path.interpolated(path._interpolation_steps)
        return Path(self.transform(ipath.vertices), ipath.codes)
    transform_path_non_affine.__doc__ = \
        Transform.transform_path_non_affine.__doc__

    if matplotlib.__version__ < '1.2':
        # Note: For compatibility with matplotlib v1.1 and older, you'll
        # need to explicitly implement a ``transform`` method as well.
        # Otherwise a ``NotImplementedError`` will be raised. This isn't
        # necessary for v1.2 and newer, however.
        transform = transform_non_affine

        # Similarly, we need to explicitly override ``transform_path`` if
        # compatibility with older matplotlib versions is needed. With v1.2
        # and newer, only overriding the ``transform_path_non_affine``
        # method is sufficient.
        transform_path = transform_path_non_affine
        transform_path.__doc__ = Transform.transform_path.__doc__

    def inverted(self):
        return HammerAxes.InvertedHammerTransform()
    inverted.__doc__ = Transform.inverted.__doc__

class InvertedHammerTransform(Transform):
    input_dims = 2
    output_dims = 2
```

```
        is_separable = False

        def transform_non_affine(self, xy):
            x = xy[:, 0:1]
            y = xy[:, 1:2]

            quarter_x = 0.25 * x
            half_y = 0.5 * y
            z = np.sqrt(1.0 - quarter_x*quarter_x - half_y*half_y)
            longitude = 2*np.arctan((z*x)/(2.0*(2.0*z*z - 1.0)))
            latitude = np.arcsin(y*z)
            return np.concatenate((longitude, latitude), 1)
        transform_non_affine.__doc__ = Transform.transform_non_affine.__doc__

        # As before, we need to implement the "transform" method for
        # compatibility with matplotlib v1.1 and older.
        if matplotlib.__version__ < '1.2':
            transform = transform_non_affine

        def inverted(self):
            # The inverse of the inverse is the original transform... ;)
            return HammerAxes.HammerTransform()
        inverted.__doc__ = Transform.inverted.__doc__

# Now register the projection with matplotlib so the user can select
# it.
register_projection(HammerAxes)

if __name__ == '__main__':
    import matplotlib.pyplot as plt
    # Now make a simple example using the custom projection.
    plt.subplot(111, projection="custom_hammer")
    p = plt.plot([-1, 1, 1], [-1, -1, 1], "o-")
    plt.grid(True)

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.8 api example code: custom_scale_example.py




```
from __future__ import unicode_literals

import numpy as np
from numpy import ma
from matplotlib import scale as mscale
from matplotlib import transforms as mtransforms
from matplotlib.ticker import Formatter, FixedLocator


class MercatorLatitudeScale(mscale.ScaleBase):
    """
    Scales data in range -pi/2 to pi/2 (-90 to 90 degrees) using
    the system used to scale latitudes in a Mercator projection.

    The scale function:
      ln(tan(y) + sec(y))

    The inverse scale function:
      atan(sinh(y))

    Since the Mercator scale tends to infinity at +/- 90 degrees,
    there is user-defined threshold, above and below which nothing
```

```
    will be plotted.  This defaults to +/- 85 degrees.

    source:
    http://en.wikipedia.org/wiki/Mercator_projection
    """

    # The scale class must have a member ``name`` that defines the
    # string used to select the scale.  For example,
    # ``gca().set_yscale("mercator")`` would be used to select this
    # scale.
    name = 'mercator'

    def __init__(self, axis, **kwargs):
        """
        Any keyword arguments passed to ``set_xscale`` and
        ``set_yscale`` will be passed along to the scale's
        constructor.

        thresh: The degree above which to crop the data.
        """
        mscale.ScaleBase.__init__(self)
        thresh = kwargs.pop("thresh", np.radians(85))
        if thresh >= np.pi / 2.0:
            raise ValueError("thresh must be less than pi/2")
        self.thresh = thresh

    def get_transform(self):
        """
        Override this method to return a new instance that does the
        actual transformation of the data.

        The MercatorLatitudeTransform class is defined below as a
        nested class of this one.
        """
        return self.MercatorLatitudeTransform(self.thresh)

    def set_default_locators_and_formatters(self, axis):
        """
        Override to set up the locators and formatters to use with the
        scale.  This is only required if the scale requires custom
        locators and formatters.  Writing custom locators and
        formatters is rather outside the scope of this example, but
        there are many helpful examples in ``ticker.py``.

        In our case, the Mercator example uses a fixed locator from
        -90 to 90 degrees and a custom formatter class to put convert
        the radians to degrees and put a degree symbol after the
        value::
        """
        class DegreeFormatter(Formatter):
            def __call__(self, x, pos=None):
                # \u00b0 : degree symbol
                return "%d\u00b0" % (np.degrees(x))
```

```
        axis.set_major_locator(FixedLocator(
            np.radians(np.arange(-90, 90, 10))))
        axis.set_major_formatter(DegreeFormatter())
        axis.set_minor_formatter(DegreeFormatter())

    def limit_range_for_scale(self, vmin, vmax, minpos):
        """
        Override to limit the bounds of the axis to the domain of the
        transform.  In the case of Mercator, the bounds should be
        limited to the threshold that was passed in.  Unlike the
        autoscaling provided by the tick locators, this range limiting
        will always be adhered to, whether the axis range is set
        manually, determined automatically or changed through panning
        and zooming.
        """
        return max(vmin, -self.thresh), min(vmax, self.thresh)

    class MercatorLatitudeTransform(mtransforms.Transform):
        # There are two value members that must be defined.
        # ``input_dims`` and ``output_dims`` specify number of input
        # dimensions and output dimensions to the transformation.
        # These are used by the transformation framework to do some
        # error checking and prevent incompatible transformations from
        # being connected together.  When defining transforms for a
        # scale, which are, by definition, separable and have only one
        # dimension, these members should always be set to 1.
        input_dims = 1
        output_dims = 1
        is_separable = True

        def __init__(self, thresh):
            mtransforms.Transform.__init__(self)
            self.thresh = thresh

        def transform_non_affine(self, a):
            """
            This transform takes an Nx1 ``numpy`` array and returns a
            transformed copy.  Since the range of the Mercator scale
            is limited by the user-specified threshold, the input
            array must be masked to contain only valid values.
            ``matplotlib`` will handle masked arrays and remove the
            out-of-range data from the plot.  Importantly, the
            ``transform`` method *must* return an array that is the
            same shape as the input array, since these values need to
            remain synchronized with values in the other dimension.
            """
            masked = ma.masked_where((a < -self.thresh) | (a > self.thresh), a)
            if masked.mask.any():
                return ma.log(np.abs(ma.tan(masked) + 1.0 / ma.cos(masked)))
            else:
                return np.log(np.abs(np.tan(a) + 1.0 / np.cos(a)))
```

```
        def inverted(self):
            """
            Override this method so matplotlib knows how to get the
            inverse transform for this transform.
            """
            return MercatorLatitudeScale.InvertedMercatorLatitudeTransform(
                self.thresh)

    class InvertedMercatorLatitudeTransform(mtransforms.Transform):
        input_dims = 1
        output_dims = 1
        is_separable = True

        def __init__(self, thresh):
            mtransforms.Transform.__init__(self)
            self.thresh = thresh

        def transform_non_affine(self, a):
            return np.arctan(np.sinh(a))

        def inverted(self):
            return MercatorLatitudeScale.MercatorLatitudeTransform(self.thresh)

# Now that the Scale class has been defined, it must be registered so
# that ``matplotlib`` can find it.
mscale.register_scale(MercatorLatitudeScale)


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    t = np.arange(-180.0, 180.0, 0.1)
    s = np.radians(t)/2.

    plt.plot(t, s, '-', lw=2)
    plt.gca().set_yscale('mercator')

    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.title('Mercator: Projection of the Oppressor')
    plt.grid(True)

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

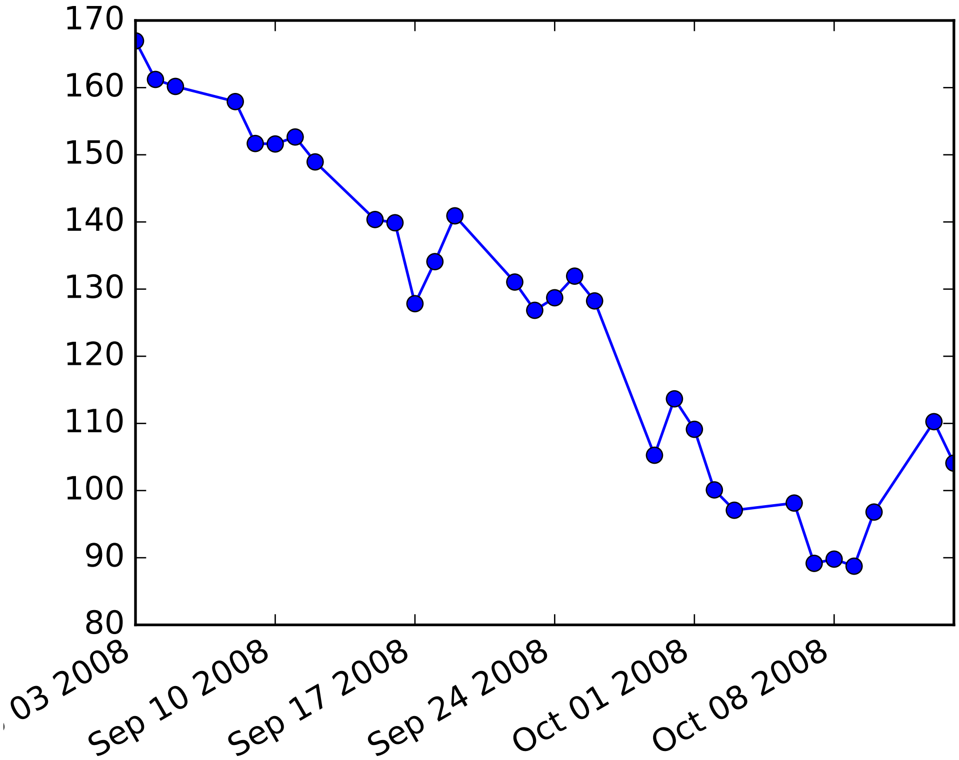
81.9 api example code: date_demo.py

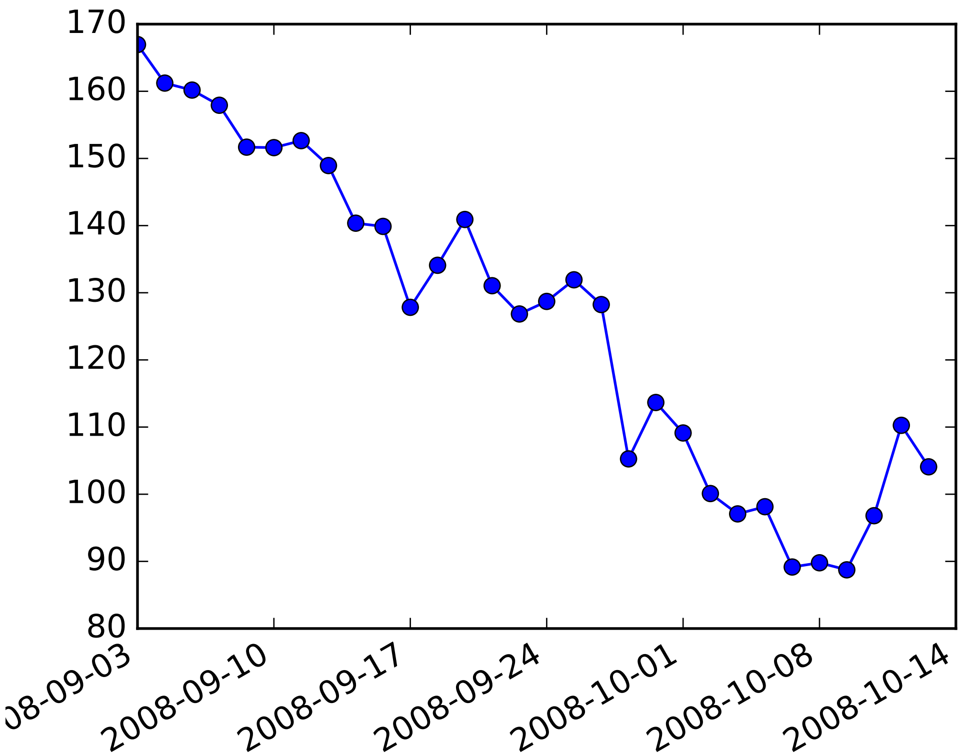

```
#!/usr/bin/env python
"""
Show how to make date plots in matplotlib using date tick locators and
formatters.  See major_minor_demo1.py for more information on
controlling major and minor ticks

All matplotlib date plotting is done by converting date instances into
days since the 0001-01-01 UTC.  The conversion, tick locating and
formatting is done behind the scenes so this is most transparent to
you.  The dates module provides several converter functions date2num
and num2date

"""
import datetime
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import matplotlib.cbook as cbook

years = mdates.YearLocator()   # every year
months = mdates.MonthLocator()  # every month
yearsFmt = mdates.DateFormatter('%Y')
```

```
# load a numpy record array from yahoo csv data with fields date,
# open, close, volume, adj_close from the mpl-data/example directory.
# The record array stores python datetime.date as an object array in
# the date column
datafile = cbook.get_sample_data('goog.npy')
try:
    # Python3 cannot load python2 .npy files with datetime(object) arrays
    # unless the encoding is set to bytes. Hovever this option was
    # not added until numpy 1.10 so this example will only work with
    # python 2 or with numpy 1.10 and later.
    r = np.load(datafile, encoding='bytes').view(np.recarray)
except TypeError:
    r = np.load(datafile).view(np.recarray)

fig, ax = plt.subplots()
ax.plot(r.date, r.adj_close)


# format the ticks
ax.xaxis.set_major_locator(years)
ax.xaxis.set_major_formatter(yearsFmt)
ax.xaxis.set_minor_locator(months)

datemin = datetime.date(r.date.min().year, 1, 1)
datemax = datetime.date(r.date.max().year + 1, 1, 1)
ax.set_xlim(datemin, datemax)


# format the coords message box
def price(x):
    return '$%1.2f' % x
ax.format_xdata = mdates.DateFormatter('%Y-%m-%d')
ax.format_ydata = price
ax.grid(True)

# rotates and right aligns the x labels, and moves the bottom of the
# axes up to make room for them
fig.autofmt_xdate()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.10 api example code: date_index_formatter.py



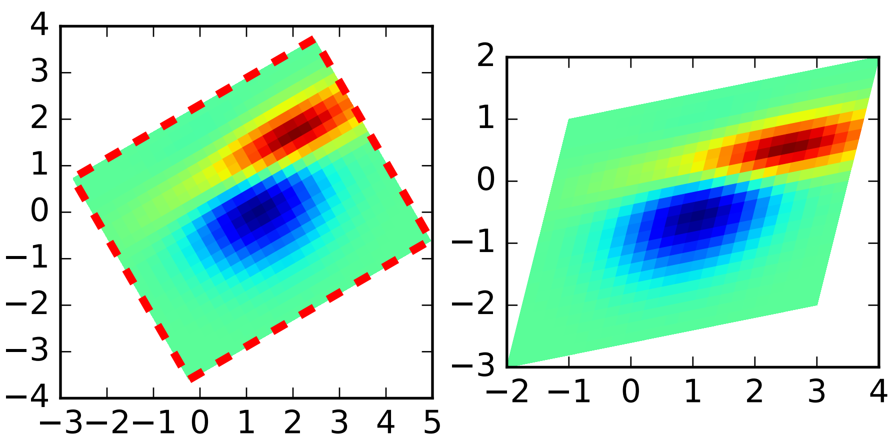



```
"""
When plotting time series, e.g., financial time series, one often wants
to leave out days on which there is no data, eh weekends.  The example
below shows how to use an 'index formatter' to achieve the desired plot
"""
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import matplotlib.ticker as ticker

datafile = cbook.get_sample_data('aapl.csv', asfileobj=False)
print('loading %s' % datafile)
r = mlab.csv2rec(datafile)

r.sort()
r = r[-30:]  # get the last 30 days


# first we'll do it the default way, with gaps on weekends
fig, ax = plt.subplots()
ax.plot(r.date, r.adj_close, 'o-')
fig.autofmt_xdate()

# next we'll write a custom formatter
```

```
N = len(r)
ind = np.arange(N)  # the evenly spaced plot indices


def format_date(x, pos=None):
    thisind = np.clip(int(x + 0.5), 0, N - 1)
    return r.date[thisind].strftime('%Y-%m-%d')

fig, ax = plt.subplots()
ax.plot(ind, r.adj_close, 'o-')
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_date))
fig.autofmt_xdate()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.11 api example code: demo_affine_image.py



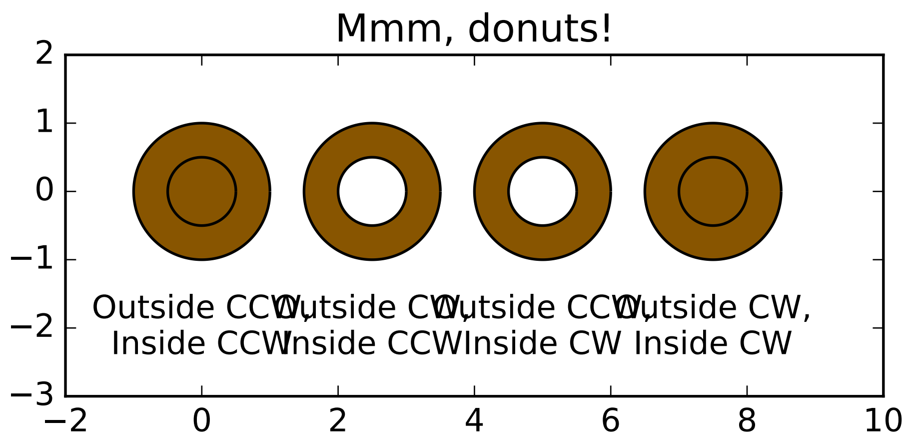

```
#!/usr/bin/env python
```

```python
"""
For the backends that supports draw_image with optional affine
transform (e.g., agg, ps backend), the image of the output should
have its boundary matches the red rectangles.
"""

import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms


def get_image():
    delta = 0.25
    x = y = np.arange(-3.0, 3.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
    Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
    Z = Z2 - Z1  # difference of Gaussians
    return Z


def imshow_affine(ax, z, *kl, **kwargs):
    im = ax.imshow(z, *kl, **kwargs)
    x1, x2, y1, y2 = im.get_extent()
    im._image_skew_coordinate = (x2, y1)
    return im


if 1:

    # image rotation

    fig, (ax1, ax2) = plt.subplots(1, 2)
    Z = get_image()
    im1 = imshow_affine(ax1, Z, interpolation='none', cmap=cm.jet,
                        origin='lower',
                        extent=[-2, 4, -3, 2], clip_on=True)

    trans_data2 = mtransforms.Affine2D().rotate_deg(30) + ax1.transData
    im1.set_transform(trans_data2)

    # display intended extent of the image
    x1, x2, y1, y2 = im1.get_extent()
    x3, y3 = x2, y1

    ax1.plot([x1, x2, x2, x1, x1], [y1, y1, y2, y2, y1], "r--", lw=3,
             transform=trans_data2)

    ax1.set_xlim(-3, 5)
    ax1.set_ylim(-4, 4)
```

```
    # image skew

    im2 = ax2.imshow(Z, interpolation='none', cmap=cm.jet,
                     origin='lower',
                     extent=[-2, 4, -3, 2], clip_on=True)
    im2._image_skew_coordinate = (3, -2)

    plt.show()
    #plt.savefig("demo_affine_image")
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.12 api example code: donut_demo.py

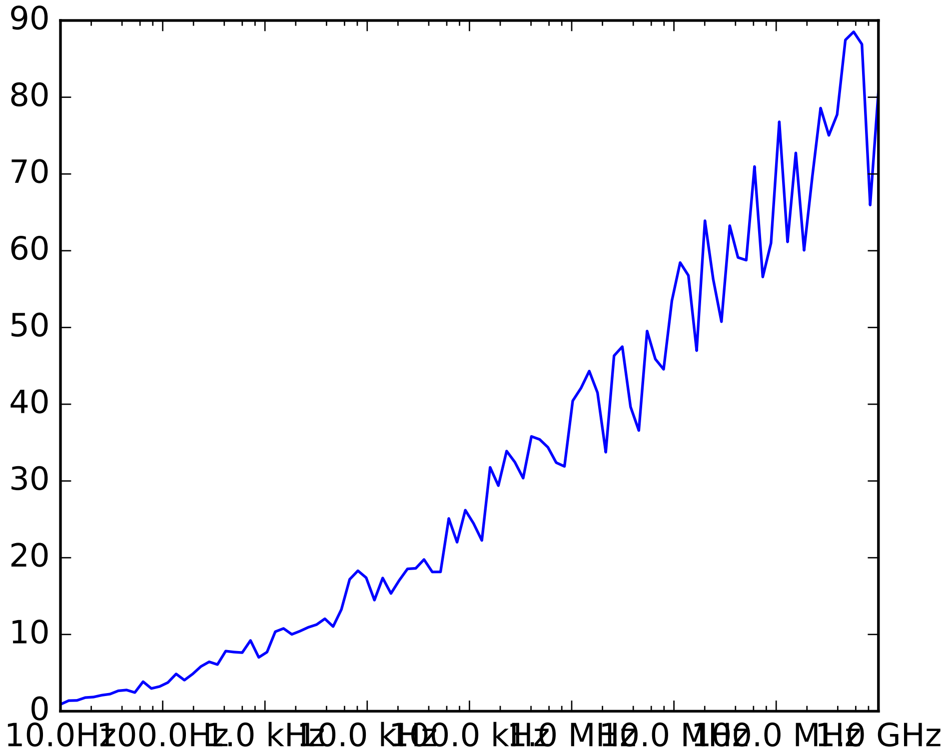



```
import numpy as np
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt


def wise(v):
    if v == 1:
```

```
        return "CCW"
    else:
        return "CW"


def make_circle(r):
    t = np.arange(0, np.pi * 2.0, 0.01)
    t = t.reshape((len(t), 1))
    x = r * np.cos(t)
    y = r * np.sin(t)
    return np.hstack((x, y))

Path = mpath.Path

fig, ax = plt.subplots()

inside_vertices = make_circle(0.5)
outside_vertices = make_circle(1.0)
codes = np.ones(
    len(inside_vertices), dtype=mpath.Path.code_type) * mpath.Path.LINETO
codes[0] = mpath.Path.MOVETO

for i, (inside, outside) in enumerate(((1, 1), (1, -1), (-1, 1), (-1, -1))):
    # Concatenate the inside and outside subpaths together, changing their
    # order as needed
    vertices = np.concatenate((outside_vertices[::outside],
                               inside_vertices[::inside]))
    # Shift the path
    vertices[:, 0] += i * 2.5
    # The codes will be all "LINETO" commands, except for "MOVETO"s at the
    # beginning of each subpath
    all_codes = np.concatenate((codes, codes))
    # Create the Path object
    path = mpath.Path(vertices, all_codes)
    # Add plot it
    patch = mpatches.PathPatch(path, facecolor='#885500', edgecolor='black')
    ax.add_patch(patch)

    ax.annotate("Outside %s,\nInside %s" % (wise(outside), wise(inside)),
                (i * 2.5, -1.5), va="top", ha="center")

ax.set_xlim(-2, 10)
ax.set_ylim(-3, 2)
ax.set_title('Mmm, donuts!')
ax.set_aspect(1.0)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

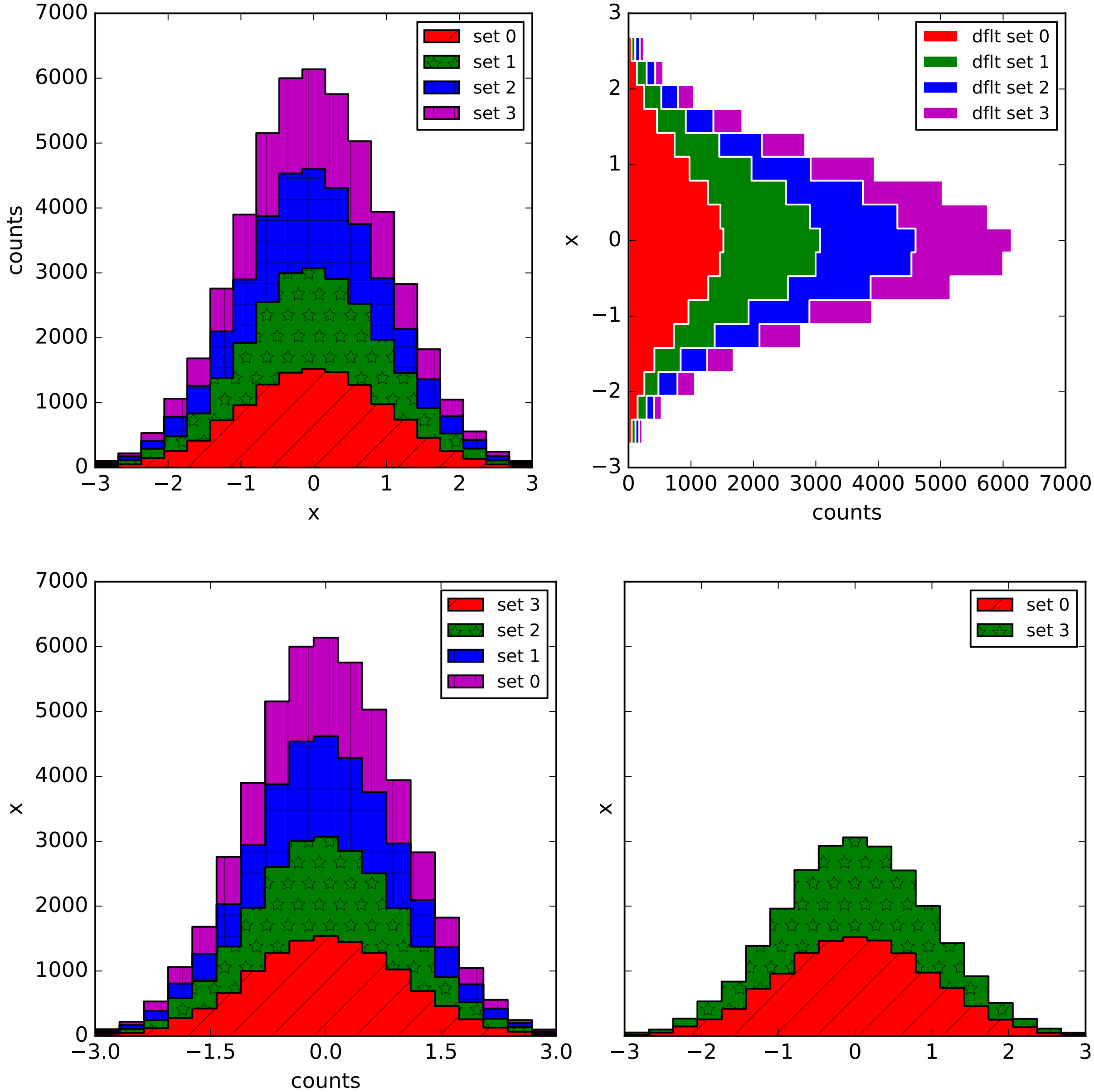
81.13 api example code: engineering_formatter.py




```
'''
Demo to show use of the engineering Formatter.
'''

import matplotlib.pyplot as plt
import numpy as np

from matplotlib.ticker import EngFormatter

fig, ax = plt.subplots()
ax.set_xscale('log')
formatter = EngFormatter(unit='Hz', places=1)
ax.xaxis.set_major_formatter(formatter)

xs = np.logspace(1, 9, 100)
ys = (0.8 + 0.4*np.random.uniform(size=100))*np.log10(xs)**2
ax.plot(xs, ys)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.14 api example code: filled_step.py




```
import itertools
from functools import partial

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from cycler import cycler
from six.moves import zip


def filled_hist(ax, edges, values, bottoms=None, orientation='v',
```

```
            **kwargs):
    """
    Draw a histogram as a stepped patch.

    Extra kwargs are passed through to `fill_between`

    Parameters
    ----------
    ax : Axes
        The axes to plot to

    edges : array
        A length n+1 array giving the left edges of each bin and the
        right edge of the last bin.

    values : array
        A length n array of bin counts or values

    bottoms : scalar or array, optional
        A length n array of the bottom of the bars.  If None, zero is used.

    orientation : {'v', 'h'}
       Orientation of the histogram.  'v' (default) has
       the bars increasing in the positive y-direction.

    Returns
    -------
    ret : PolyCollection
        Artist added to the Axes
    """
    print(orientation)
    if orientation not in set('hv'):
        raise ValueError("orientation must be in {{'h', 'v'}} "
                         "not {o}".format(o=orientation))

    kwargs.setdefault('step', 'post')
    edges = np.asarray(edges)
    values = np.asarray(values)
    if len(edges) - 1 != len(values):
        raise ValueError('Must provide one more bin edge than value not: '
                         'len(edges): {lb} len(values): {lv}'.format(
                             lb=len(edges), lv=len(values)))

    if bottoms is None:
        bottoms = np.zeros_like(values)
    if np.isscalar(bottoms):
        bottoms = np.ones_like(values) * bottoms

    values = np.r_[values, values[-1]]
    bottoms = np.r_[bottoms, bottoms[-1]]
    if orientation == 'h':
        return ax.fill_betweenx(edges, values, bottoms, **kwargs)
    elif orientation == 'v':
```

```
        return ax.fill_between(edges, values, bottoms, **kwargs)
    else:
        raise AssertionError("you should never be here")


def stack_hist(ax, stacked_data, sty_cycle, bottoms=None,
               hist_func=None, labels=None,
               plot_func=None, plot_kwargs=None):
    """
    ax : axes.Axes
        The axes to add artists too

    stacked_data : array or Mapping
        A (N, M) shaped array.  The first dimension will be iterated over to
        compute histograms row-wise

    sty_cycle : Cycler or operable of dict
        Style to apply to each set

    bottoms : array, optional
        The initial positions of the bottoms, defaults to 0

    hist_func : callable, optional
        Must have signature `bin_vals, bin_edges = f(data)`.
        `bin_edges` expected to be one longer than `bin_vals`

    labels : list of str, optional
        The label for each set.

        If not given and stacked data is an array defaults to 'default set {n}'

        If stacked_data is a mapping, and labels is None, default to the keys
        (which may come out in a random order).

        If stacked_data is a mapping and labels is given then only
        the columns listed by be plotted.

    plot_func : callable, optional
        Function to call to draw the histogram must have signature:

          ret = plot_func(ax, edges, top, bottoms=bottoms,
                          label=label, **kwargs)

    plot_kwargs : dict, optional
        Any extra kwargs to pass through to the plotting function.  This
        will be the same for all calls to the plotting function and will
        over-ride the values in cycle.

    Returns
    -------
    arts : dict
        Dictionary of artists keyed on their labels
    """
```

```
    # deal with default binning function
    if hist_func is None:
        hist_func = np.histogram

    # deal with default plotting function
    if plot_func is None:
        plot_func = filled_hist

    # deal with default
    if plot_kwargs is None:
        plot_kwargs = {}
    print(plot_kwargs)
    try:
        l_keys = stacked_data.keys()
        label_data = True
        if labels is None:
            labels = l_keys

    except AttributeError:
        label_data = False
        if labels is None:
            labels = itertools.repeat(None)

    if label_data:
        loop_iter = enumerate((stacked_data[lab], lab, s) for lab, s in
                              zip(labels, sty_cycle))
    else:
        loop_iter = enumerate(zip(stacked_data, labels, sty_cycle))

    arts = {}
    for j, (data, label, sty) in loop_iter:
        if label is None:
            label = 'dflt set {n}'.format(n=j)
        label = sty.pop('label', label)
        vals, edges = hist_func(data)
        if bottoms is None:
            bottoms = np.zeros_like(vals)
        top = bottoms + vals
        print(sty)
        sty.update(plot_kwargs)
        print(sty)
        ret = plot_func(ax, edges, top, bottoms=bottoms,
                        label=label, **sty)
        bottoms = top
        arts[label] = ret
    ax.legend(fontsize=10)
    return arts


# set up histogram function to fixed bins
edges = np.linspace(-3, 3, 20, endpoint=True)
hist_func = partial(np.histogram, bins=edges)
```

```
# set up style cycles
color_cycle = cycler('facecolor', 'rgbm')
label_cycle = cycler('label', ['set {n}'.format(n=n) for n in range(4)])
hatch_cycle = cycler('hatch', ['/', '*', '+', '|'])

# make some synthetic data
stack_data = np.random.randn(4, 12250)
dict_data = {lab: d for lab, d in zip(list(c['label'] for c in label_cycle),
                                      stack_data)}

# work with plain arrays
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4.5), tight_layout=True)
arts = stack_hist(ax1, stack_data, color_cycle + label_cycle + hatch_cycle,
                  hist_func=hist_func)

arts = stack_hist(ax2, stack_data, color_cycle,
                  hist_func=hist_func,
                  plot_kwargs=dict(edgecolor='w', orientation='h'))
ax1.set_ylabel('counts')
ax1.set_xlabel('x')
ax2.set_xlabel('counts')
ax2.set_ylabel('x')

# work with labeled data

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4.5),
                               tight_layout=True, sharey=True)

arts = stack_hist(ax1, dict_data, color_cycle + hatch_cycle,
                  hist_func=hist_func)

arts = stack_hist(ax2, dict_data, color_cycle + hatch_cycle,
                  hist_func=hist_func, labels=['set 0', 'set 3'])
ax1.xaxis.set_major_locator(mticker.MaxNLocator(5))
ax1.set_xlabel('counts')
ax1.set_ylabel('x')
ax2.set_ylabel('x')
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.15 api example code: font_family_rc.py

[source code]

```
"""
You can explicitly set which font family is picked up for a given font
style (e.g., 'serif', 'sans-serif', or 'monospace').

In the example below, we only allow one font family (Tahoma) for the
san-serif font style.  You the default family with the font.family rc
param, e.g.,::
```

```
  rcParams['font.family'] = 'sans-serif'

and for the font.family you set a list of font styles to try to find
in order::

  rcParams['font.sans-serif'] = ['Tahoma', 'Bitstream Vera Sans',
                                 'Lucida Grande', 'Verdana']

"""

# -*- noplot -*-

from matplotlib import rcParams
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Tahoma']
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot([1, 2, 3], label='test')

ax.legend()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.16 api example code: font_file.py

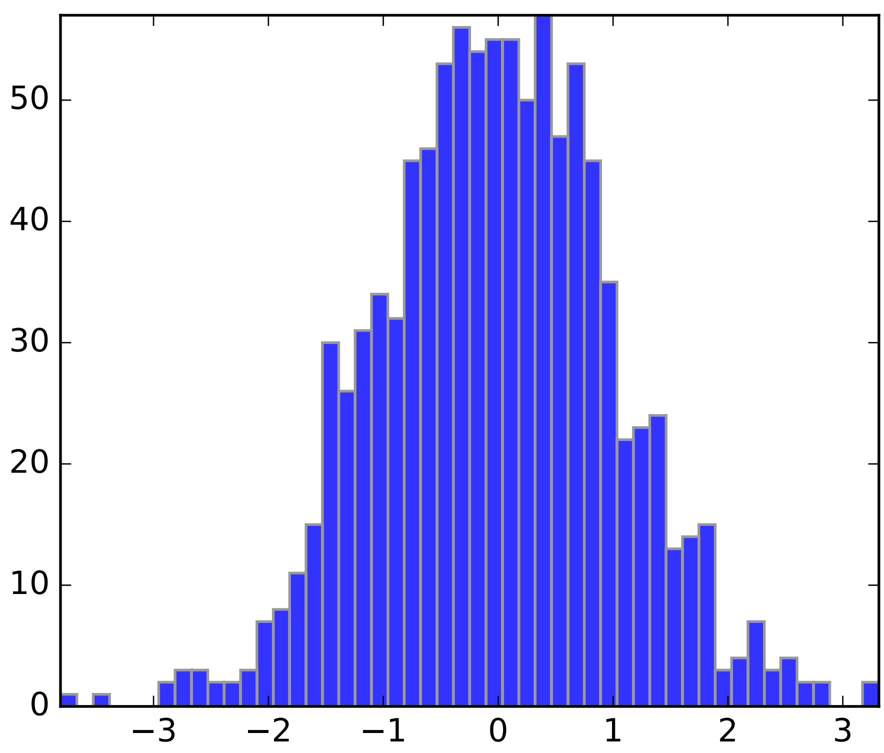
[source code]

```
# -*- noplot -*-
"""
Although it is usually not a good idea to explicitly point to a single
ttf file for a font instance, you can do so using the
font_manager.FontProperties fname argument (for a more flexible
solution, see the font_fmaily_rc.py and fonts_demo.py examples).
"""
import sys
import os
import matplotlib.font_manager as fm

import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot([1, 2, 3])

if sys.platform == 'win32':
    fpath = 'C:\\Windows\\Fonts\\Tahoma.ttf'
elif sys.platform.startswith('linux'):
    basedir = '/usr/share/fonts/truetype'
    fonts = ['freefont/FreeSansBoldOblique.ttf',
```

```
             'ttf-liberation/LiberationSans-BoldItalic.ttf',
             'msttcorefonts/Comic_Sans_MS.ttf']
    for fpath in fonts:
        if os.path.exists(os.path.join(basedir, fpath)):
            break
else:
    fpath = '/Library/Fonts/Tahoma.ttf'

if os.path.exists(fpath):
    prop = fm.FontProperties(fname=fpath)
    fname = os.path.split(fpath)[1]
    ax.set_title('this is a special font: %s' % fname, fontproperties=prop)
else:
    ax.set_title('Demo fails--cannot find a demo font')
ax.set_xlabel('This is the default font')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

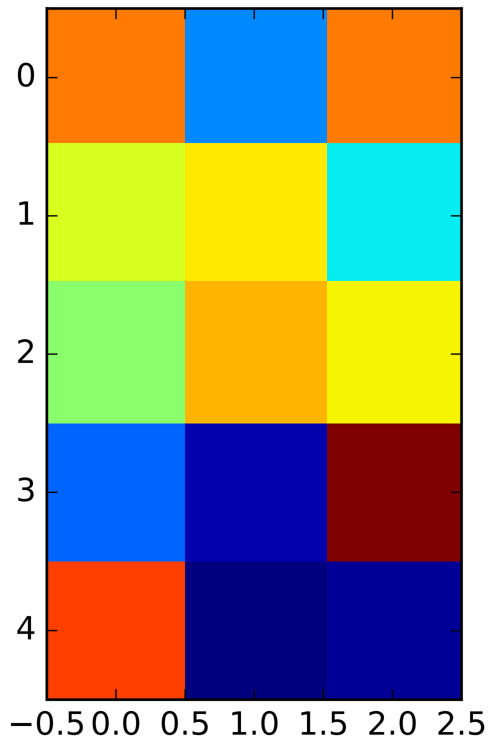
81.17 api example code: histogram_path_demo.py



```
"""
This example shows how to use a path patch to draw a bunch of
rectangles.  The technique of using lots of Rectangle instances, or
the faster method of using PolyCollections, were implemented before we
had proper paths with moveto/lineto, closepoly etc in mpl.  Now that
we have them, we can draw collections of regularly shaped objects with
homogeous properties more efficiently with a PathCollection.  This
example makes a histogram -- its more work to set up the vertex arrays
at the outset, but it should be much faster for large numbers of
objects
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.path as path

fig, ax = plt.subplots()

# histogram our data with numpy
data = np.random.randn(1000)
n, bins = np.histogram(data, 50)

# get the corners of the rectangles for the histogram
left = np.array(bins[:-1])
right = np.array(bins[1:])
bottom = np.zeros(len(left))
top = bottom + n


# we need a (numrects x numsides x 2) numpy array for the path helper
# function to build a compound path
XY = np.array([[left, left, right, right], [bottom, top, top, bottom]]).T

# get the Path object
barpath = path.Path.make_compound_path_from_polys(XY)

# make a patch out of it
patch = patches.PathPatch(
    barpath, facecolor='blue', edgecolor='gray', alpha=0.8)
ax.add_patch(patch)

# update the view limits
ax.set_xlim(left[0], right[-1])
ax.set_ylim(bottom.min(), top.max())

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.18 api example code: image_zcoord.py

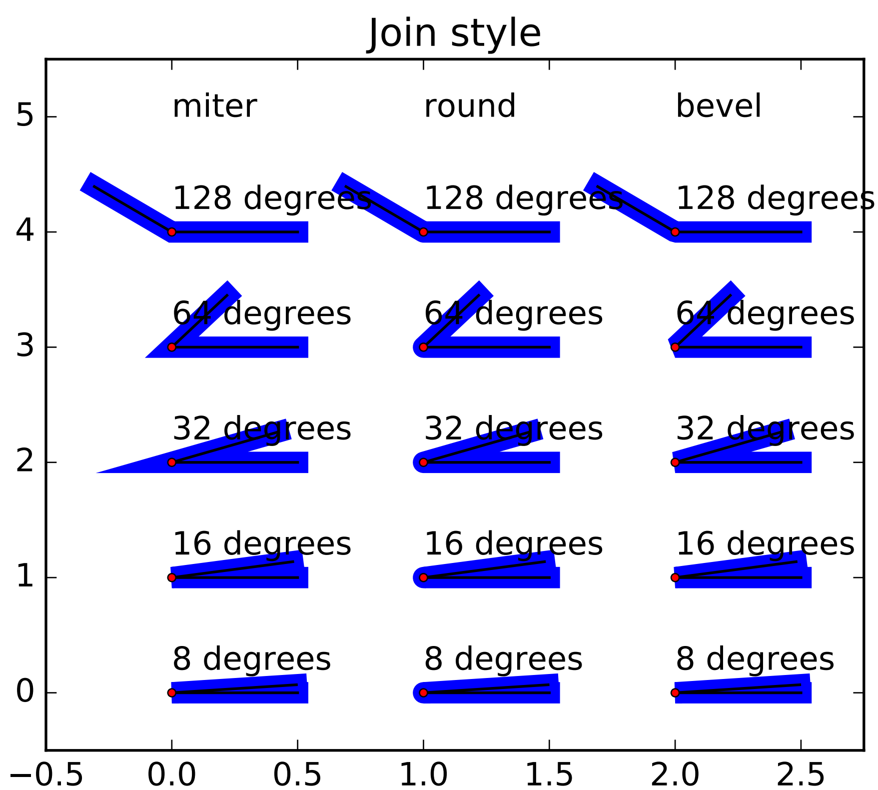



```
"""
Show how to modify the coordinate formatter to report the image "z"
value of the nearest pixel given x and y
"""
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

X = 10*np.random.rand(5, 3)

fig, ax = plt.subplots()
ax.imshow(X, cmap=cm.jet, interpolation='nearest')

numrows, numcols = X.shape


def format_coord(x, y):
    col = int(x + 0.5)
    row = int(y + 0.5)
    if col >= 0 and col < numcols and row >= 0 and row < numrows:
        z = X[row, col]
        return 'x=%1.4f, y=%1.4f, z=%1.4f' % (x, y, z)
```

```
    else:
        return 'x=%1.4f, y=%1.4f' % (x, y)

ax.format_coord = format_coord
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.19 api example code: joinstyle.py

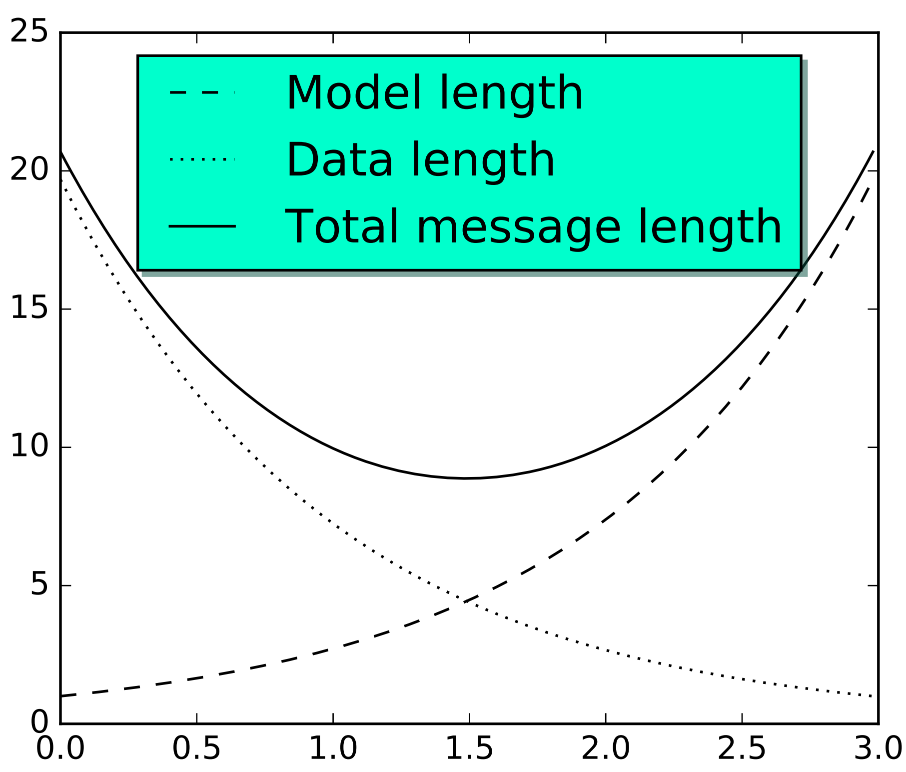



```
#!/usr/bin/env python
"""
Illustrate the three different join styles
"""

import numpy as np
import matplotlib.pyplot as plt


def plot_angle(ax, x, y, angle, style):
    phi = np.radians(angle)
    xx = [x + .5, x, x + .5*np.cos(phi)]
```

```
    yy = [y, y, y + .5*np.sin(phi)]
    ax.plot(xx, yy, lw=8, color='blue', solid_joinstyle=style)
    ax.plot(xx[1:], yy[1:], lw=1, color='black')
    ax.plot(xx[1::-1], yy[1::-1], lw=1, color='black')
    ax.plot(xx[1:2], yy[1:2], 'o', color='red', markersize=3)
    ax.text(x, y + .2, '%.0f degrees' % angle)

fig, ax = plt.subplots()
ax.set_title('Join style')

for x, style in enumerate((('miter', 'round', 'bevel'))):
    ax.text(x, 5, style)
    for i in range(5):
        plot_angle(ax, x, i, pow(2.0, 3 + i), style)

ax.set_xlim(-.5, 2.75)
ax.set_ylim(-.5, 5.5)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

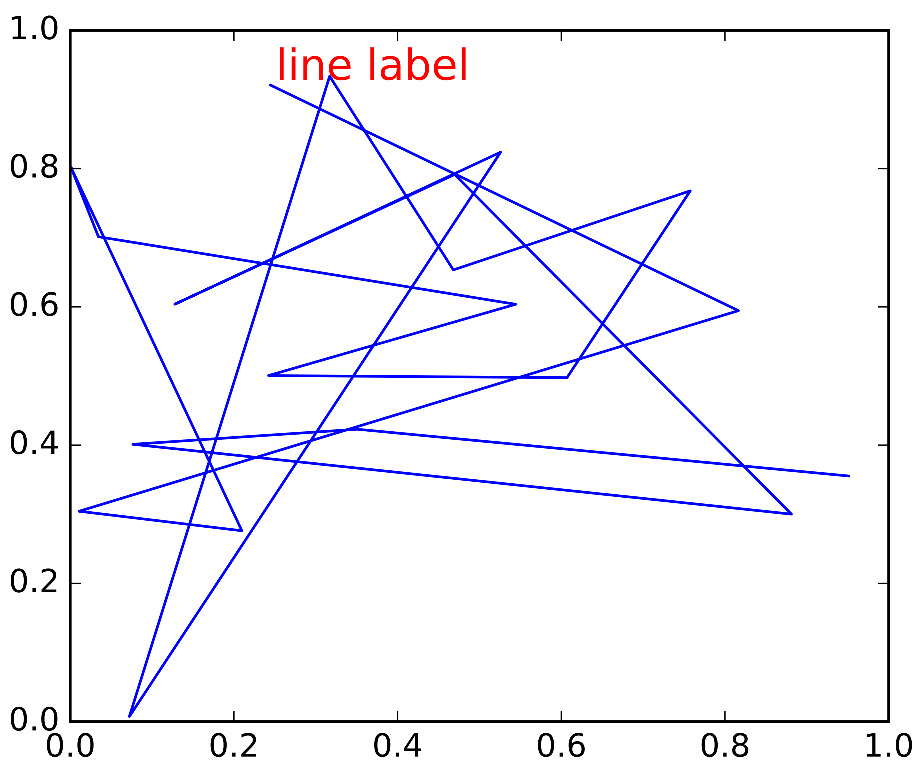
81.20 api example code: legend_demo.py



```
import numpy as np
import matplotlib.pyplot as plt

# Make some fake data.
a = b = np.arange(0, 3, .02)
c = np.exp(a)
d = c[::-1]

# Create plots with pre-defined labels.
plt.plot(a, c, 'k--', label='Model length')
plt.plot(a, d, 'k:', label='Data length')
plt.plot(a, c + d, 'k', label='Total message length')

legend = plt.legend(loc='upper center', shadow=True, fontsize='x-large')

# Put a nicer background color on the legend.
legend.get_frame().set_facecolor('#00FFCC')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.21 api example code: line_with_text.py

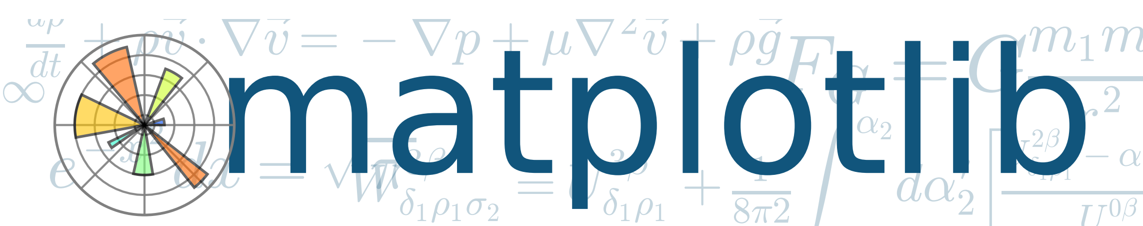


```
"""
Show how to override basic methods so an artist can contain another
artist.  In this case, the line contains a Text instance to label it.
"""
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines
import matplotlib.transforms as mtransforms
import matplotlib.text as mtext


class MyLine(lines.Line2D):
    def __init__(self, *args, **kwargs):
        # we'll update the position when the line data is set
        self.text = mtext.Text(0, 0, '')
        lines.Line2D.__init__(self, *args, **kwargs)

        # we can't access the label attr until *after* the line is
        # inited
        self.text.set_text(self.get_label())

    def set_figure(self, figure):
        self.text.set_figure(figure)
        lines.Line2D.set_figure(self, figure)

    def set_axes(self, axes):
        self.text.set_axes(axes)
        lines.Line2D.set_axes(self, axes)

    def set_transform(self, transform):
        # 2 pixel offset
        texttrans = transform + mtransforms.Affine2D().translate(2, 2)
        self.text.set_transform(texttrans)
        lines.Line2D.set_transform(self, transform)

    def set_data(self, x, y):
        if len(x):
            self.text.set_position((x[-1], y[-1]))

        lines.Line2D.set_data(self, x, y)

    def draw(self, renderer):
        # draw my label at the end of the line with 2 pixel offset
        lines.Line2D.draw(self, renderer)
        self.text.draw(renderer)


fig, ax = plt.subplots()
x, y = np.random.rand(2, 20)
line = MyLine(x, y, mfc='red', ms=12, label='line label')
#line.text.set_text('line label')
line.text.set_color('red')
line.text.set_fontsize(16)
```

```
ax.add_line(line)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.22 api example code: logo2.py




```
"""
Thanks to Tony Yu <tsyu80@gmail.com> for the logo design
"""

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.cm as cm

mpl.rcParams['xtick.labelsize'] = 10
mpl.rcParams['ytick.labelsize'] = 12
mpl.rcParams['axes.edgecolor'] = 'gray'


axalpha = 0.05
#figcolor = '#EFEFEF'
figcolor = 'white'
dpi = 80
fig = plt.figure(figsize=(6, 1.1), dpi=dpi)
fig.figurePatch.set_edgecolor(figcolor)
fig.figurePatch.set_facecolor(figcolor)


def add_math_background():
    ax = fig.add_axes([0., 0., 1., 1.])

    text = []
    text.append(
        (r"$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = "
         r"U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8 \pi 2}"
         r"\int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 "
         r"\left[\frac{ U^{2\beta}_{\delta_1 \rho_1} - "
```

```
        r"\alpha^\prime_2U^{1\beta}_{\rho_1 \sigma_2} "
        r"}{U^{0\beta}_{\rho_1 \sigma_2}}\right]$", (0.7, 0.2), 20))
    text.append((r"$\frac{d\rho}{d t} + \rho \vec{v}\cdot\nabla\vec{v} "
                r"= -\nabla p + \mu\nabla^2 \vec{v} + \rho \vec{g}$",
                (0.35, 0.9), 20))
    text.append((r"$\int_{-\infty}^\infty e^{-x^2}dx=\sqrt{\pi}$",
                (0.15, 0.3), 25))
    #text.append((r"$E = mc^2 = \sqrt{{m_0}^2c^4 + p^2c^2}$",
    #             (0.7, 0.42), 30))
    text.append((r"$F_G = G\frac{m_1m_2}{r^2}$",
                (0.85, 0.7), 30))
    for eq, (x, y), size in text:
        ax.text(x, y, eq, ha='center', va='center', color="#11557c",
                alpha=0.25, transform=ax.transAxes, fontsize=size)
    ax.set_axis_off()
    return ax


def add_matplotlib_text(ax):
    ax.text(0.95, 0.5, 'matplotlib', color='#11557c', fontsize=65,
            ha='right', va='center', alpha=1.0, transform=ax.transAxes)


def add_polar_bar():
    ax = fig.add_axes([0.025, 0.075, 0.2, 0.85], projection='polar')

    ax.axesPatch.set_alpha(axalpha)
    ax.set_axisbelow(True)
    N = 7
    arc = 2. * np.pi
    theta = np.arange(0.0, arc, arc/N)
    radii = 10 * np.array([0.2, 0.6, 0.8, 0.7, 0.4, 0.5, 0.8])
    width = np.pi / 4 * np.array([0.4, 0.4, 0.6, 0.8, 0.2, 0.5, 0.3])
    bars = ax.bar(theta, radii, width=width, bottom=0.0)
    for r, bar in zip(radii, bars):
        bar.set_facecolor(cm.jet(r/10.))
        bar.set_alpha(0.6)

    for label in ax.get_xticklabels() + ax.get_yticklabels():
        label.set_visible(False)

    for line in ax.get_ygridlines() + ax.get_xgridlines():
        line.set_lw(0.8)
        line.set_alpha(0.9)
        line.set_ls('-')
        line.set_color('0.5')

    ax.set_yticks(np.arange(1, 9, 2))
    ax.set_rmax(9)

if __name__ == '__main__':
    main_axes = add_math_background()
    add_polar_bar()
```

```
    add_matplotlib_text(main_axes)
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

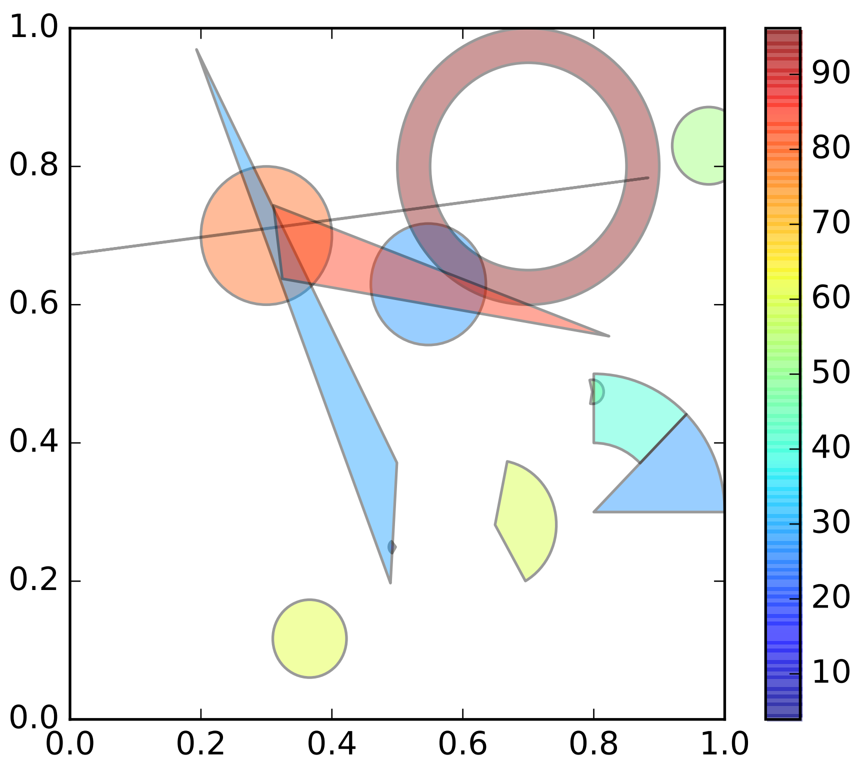
81.23 api example code: mathtext_asarray.py


```
"""
Load a mathtext image as numpy array
"""

import matplotlib.mathtext as mathtext
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rc('image', origin='upper')

parser = mathtext.MathTextParser("Bitmap")
parser.to_png('test2.png',
              r'$\left[\left\lfloor\frac{5}{\frac{\left(3\right)}{4}} '
              r'y\right)\right]$', color='green', fontsize=14, dpi=100)

rgba1, depth1 = parser.to_rgba(
```

```
    r'IQ: $\sigma_i=15$', color='blue', fontsize=20, dpi=200)
rgba2, depth2 = parser.to_rgba(
    r'some other string', color='red', fontsize=20, dpi=200)

fig = plt.figure()
fig.figimage(rgba1.astype(float)/255., 100, 100)
fig.figimage(rgba2.astype(float)/255., 100, 300)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.24 api example code: patch_collection.py

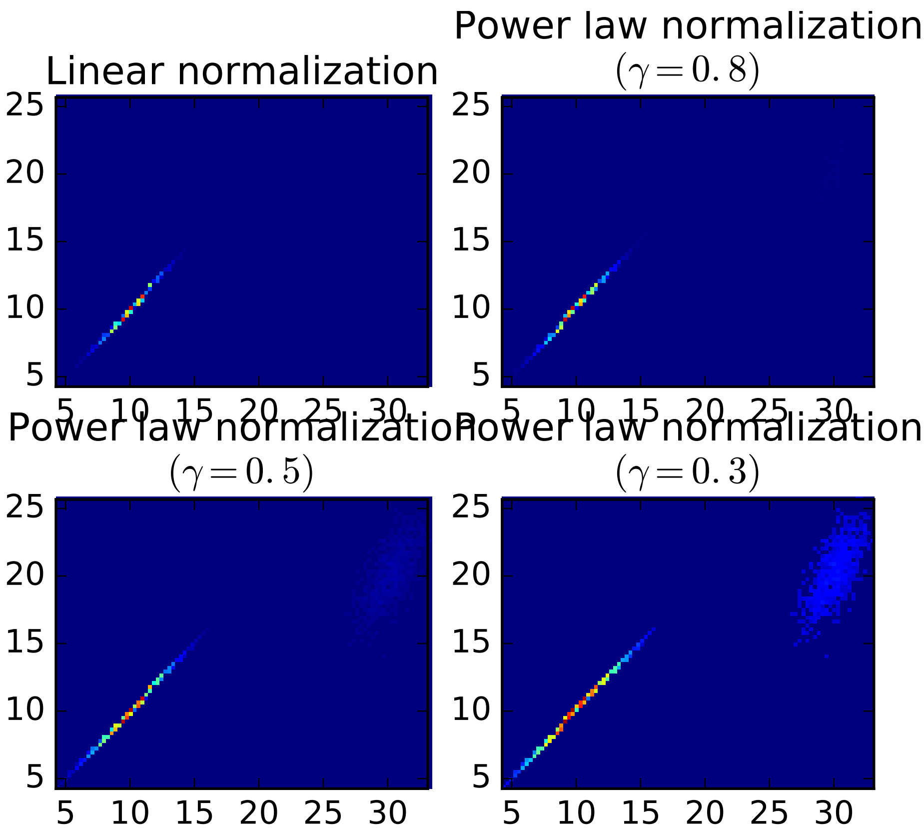



```
import numpy as np
import matplotlib
from matplotlib.patches import Circle, Wedge, Polygon
from matplotlib.collections import PatchCollection
import matplotlib.pyplot as plt


fig, ax = plt.subplots()
```

```
resolution = 50  # the number of vertices
N = 3
x = np.random.rand(N)
y = np.random.rand(N)
radii = 0.1*np.random.rand(N)
patches = []
for x1, y1, r in zip(x, y, radii):
    circle = Circle((x1, y1), r)
    patches.append(circle)

x = np.random.rand(N)
y = np.random.rand(N)
radii = 0.1*np.random.rand(N)
theta1 = 360.0*np.random.rand(N)
theta2 = 360.0*np.random.rand(N)
for x1, y1, r, t1, t2 in zip(x, y, radii, theta1, theta2):
    wedge = Wedge((x1, y1), r, t1, t2)
    patches.append(wedge)

# Some limiting conditions on Wedge
patches += [
    Wedge((.3, .7), .1, 0, 360),             # Full circle
    Wedge((.7, .8), .2, 0, 360, width=0.05),  # Full ring
    Wedge((.8, .3), .2, 0, 45),              # Full sector
    Wedge((.8, .3), .2, 45, 90, width=0.10),  # Ring sector
]

for i in range(N):
    polygon = Polygon(np.random.rand(N, 2), True)
    patches.append(polygon)

colors = 100*np.random.rand(len(patches))
p = PatchCollection(patches, cmap=matplotlib.cm.jet, alpha=0.4)
p.set_array(np.array(colors))
ax.add_collection(p)
plt.colorbar(p)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.25 api example code: power_norm_demo.py

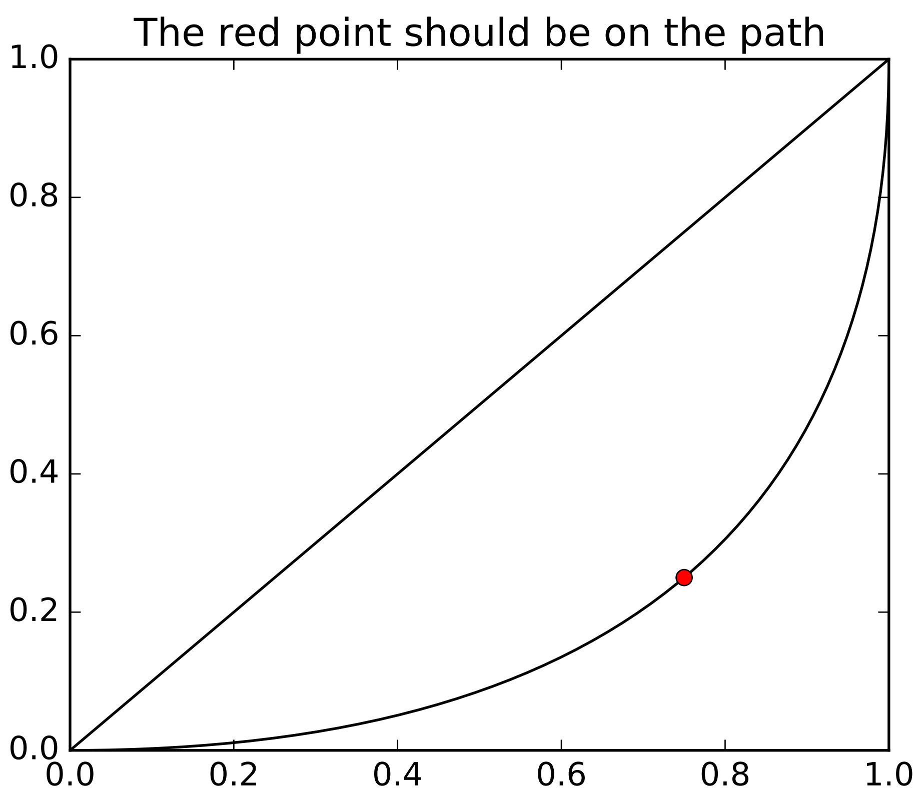



```
#!/usr/bin/python

from matplotlib import pyplot as plt
import matplotlib.colors as mcolors
import numpy as np
from numpy.random import multivariate_normal

data = np.vstack([multivariate_normal([10, 10], [[2, 2], [2, 2]], size=100000),
                  multivariate_normal([30, 20], [[2, 3], [1, 3]], size=1000)
                  ])

gammas = [0.8, 0.5, 0.3]
xgrid = np.floor((len(gammas) + 1.) / 2)
ygrid = np.ceil((len(gammas) + 1.) / 2)

plt.subplot(xgrid, ygrid, 1)
plt.title('Linear normalization')
plt.hist2d(data[:, 0], data[:, 1], bins=100)

for i, gamma in enumerate(gammas):
    plt.subplot(xgrid, ygrid, i + 2)
    plt.title('Power law normalization\n$(\gamma=%1.1f)$' % gamma)
```

```
    plt.hist2d(data[:, 0], data[:, 1],
               bins=100, norm=mcolors.PowerNorm(gamma))

plt.subplots_adjust(hspace=0.39)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.26 api example code: quad_bezier.py

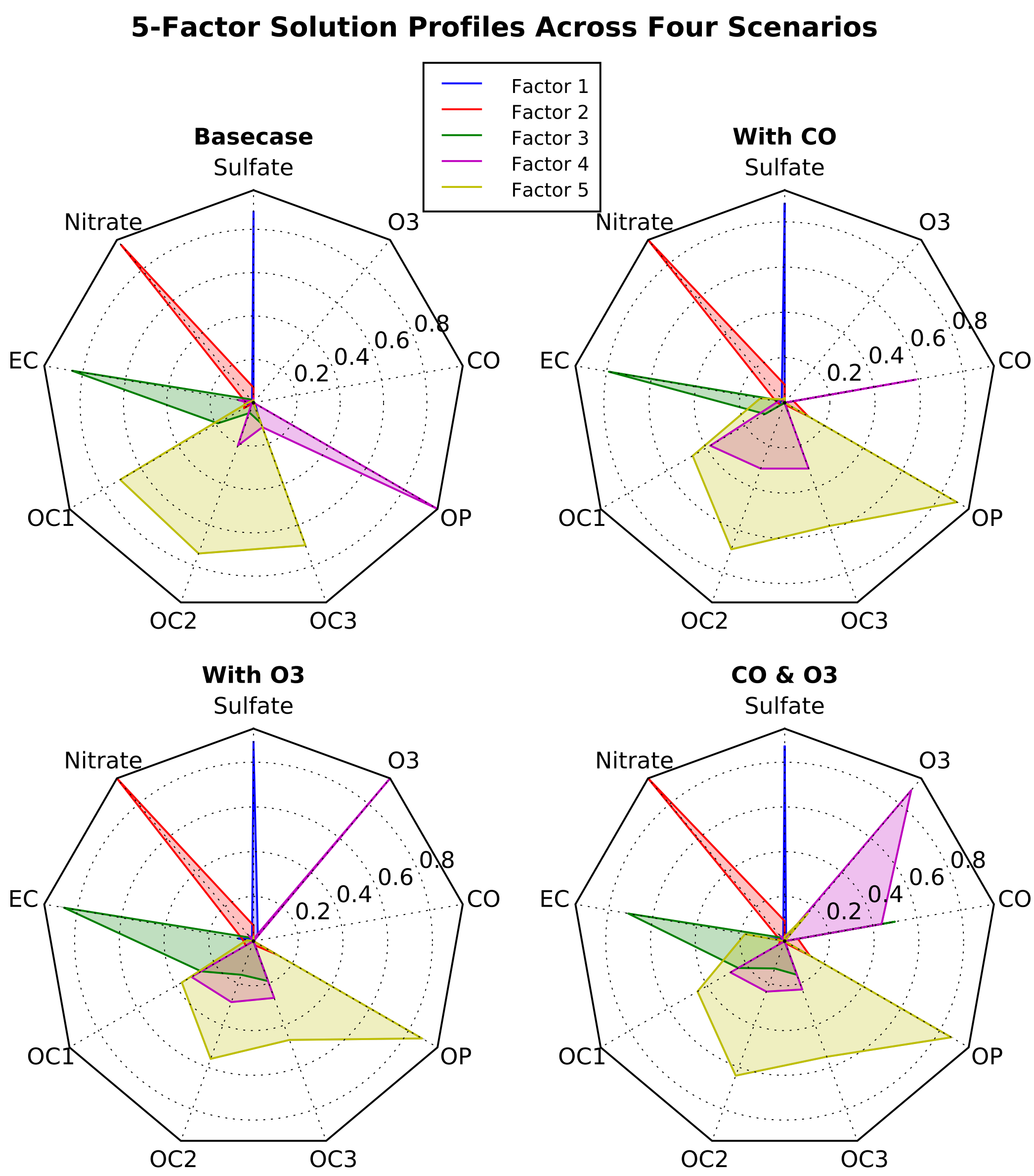



```
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

Path = mpath.Path

fig, ax = plt.subplots()
pp1 = mpatches.PathPatch(
    Path([(0, 0), (1, 0), (1, 1), (0, 0)],
         [Path.MOVETO, Path.CURVE3, Path.CURVE3, Path.CLOSEPOLY]),
    fc="none", transform=ax.transData)

```

```
ax.add_patch(pp1)
ax.plot([0.75], [0.25], "ro")
ax.set_title('The red point should be on the path')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.27 api example code: radar_chart.py



```
"""
Example of creating a radar chart (a.k.a. a spider or star chart) [1]_.

Although this example allows a frame of either 'circle' or 'polygon', polygon
frames don't have proper gridlines (the lines are circles instead of polygons).
It's possible to get a polygon grid by setting GRIDLINE_INTERPOLATION_STEPS in
matplotlib.axis to the desired number of vertices, but the orientation of the
polygon is not aligned with the radial axes.

.. [1] http://en.wikipedia.org/wiki/Radar_chart
"""
import numpy as np

import matplotlib.pyplot as plt
from matplotlib.path import Path
from matplotlib.spines import Spine
from matplotlib.projections.polar import PolarAxes
from matplotlib.projections import register_projection


def radar_factory(num_vars, frame='circle'):
    """Create a radar chart with `num_vars` axes.

    This function creates a RadarAxes projection and registers it.

    Parameters
    ----------
    num_vars : int
        Number of variables for radar chart.
    frame : {'circle' | 'polygon'}
        Shape of frame surrounding axes.

    """
    # calculate evenly-spaced axis angles
    theta = np.linspace(0, 2*np.pi, num_vars, endpoint=False)
    # rotate theta such that the first axis is at the top
    theta += np.pi/2

    def draw_poly_patch(self):
        verts = unit_poly_verts(theta)
        return plt.Polygon(verts, closed=True, edgecolor='k')

    def draw_circle_patch(self):
        # unit circle centered on (0.5, 0.5)
        return plt.Circle((0.5, 0.5), 0.5)

    patch_dict = {'polygon': draw_poly_patch, 'circle': draw_circle_patch}
    if frame not in patch_dict:
        raise ValueError('unknown value for `frame`: %s' % frame)

    class RadarAxes(PolarAxes):

        name = 'radar'
```

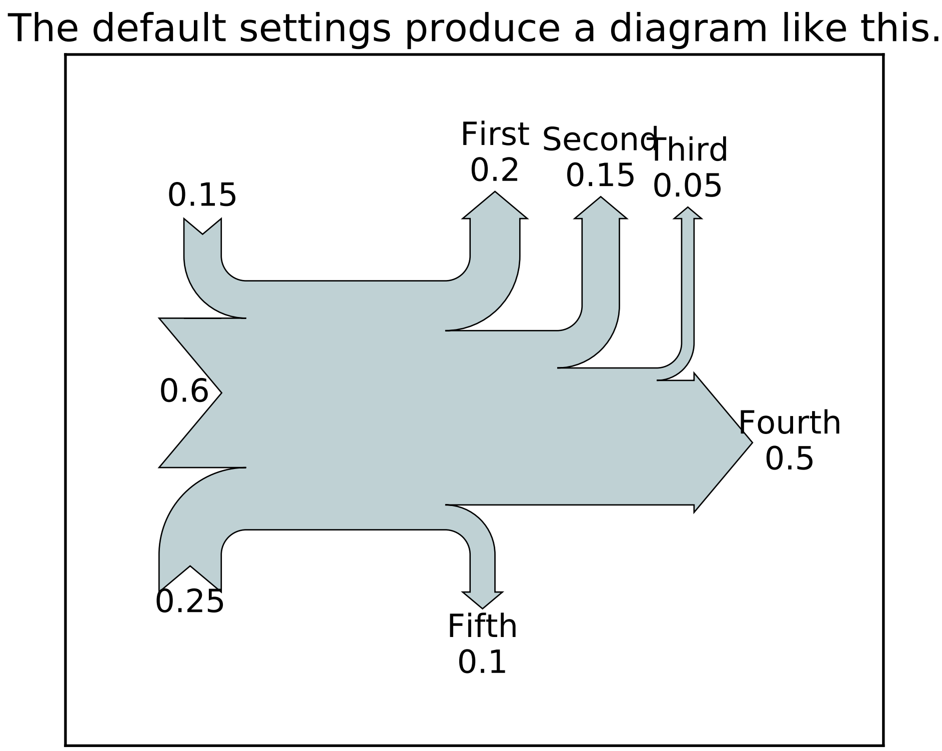
```python
        # use 1 line segment to connect specified points
        RESOLUTION = 1
        # define draw_frame method
        draw_patch = patch_dict[frame]

        def fill(self, *args, **kwargs):
            """Override fill so that line is closed by default"""
            closed = kwargs.pop('closed', True)
            return super(RadarAxes, self).fill(closed=closed, *args, **kwargs)

        def plot(self, *args, **kwargs):
            """Override plot so that line is closed by default"""
            lines = super(RadarAxes, self).plot(*args, **kwargs)
            for line in lines:
                self._close_line(line)

        def _close_line(self, line):
            x, y = line.get_data()
            # FIXME: markers at x[0], y[0] get doubled-up
            if x[0] != x[-1]:
                x = np.concatenate((x, [x[0]]))
                y = np.concatenate((y, [y[0]]))
                line.set_data(x, y)

        def set_varlabels(self, labels):
            self.set_thetagrids(np.degrees(theta), labels)

        def _gen_axes_patch(self):
            return self.draw_patch()

        def _gen_axes_spines(self):
            if frame == 'circle':
                return PolarAxes._gen_axes_spines(self)
            # The following is a hack to get the spines (i.e. the axes frame)
            # to draw correctly for a polygon frame.

            # spine_type must be 'left', 'right', 'top', 'bottom', or `circle`.
            spine_type = 'circle'
            verts = unit_poly_verts(theta)
            # close off polygon by repeating first vertex
            verts.append(verts[0])
            path = Path(verts)

            spine = Spine(self, spine_type, path)
            spine.set_transform(self.transAxes)
            return {'polar': spine}

    register_projection(RadarAxes)
    return theta


def unit_poly_verts(theta):
    """Return vertices of polygon for subplot axes.
```

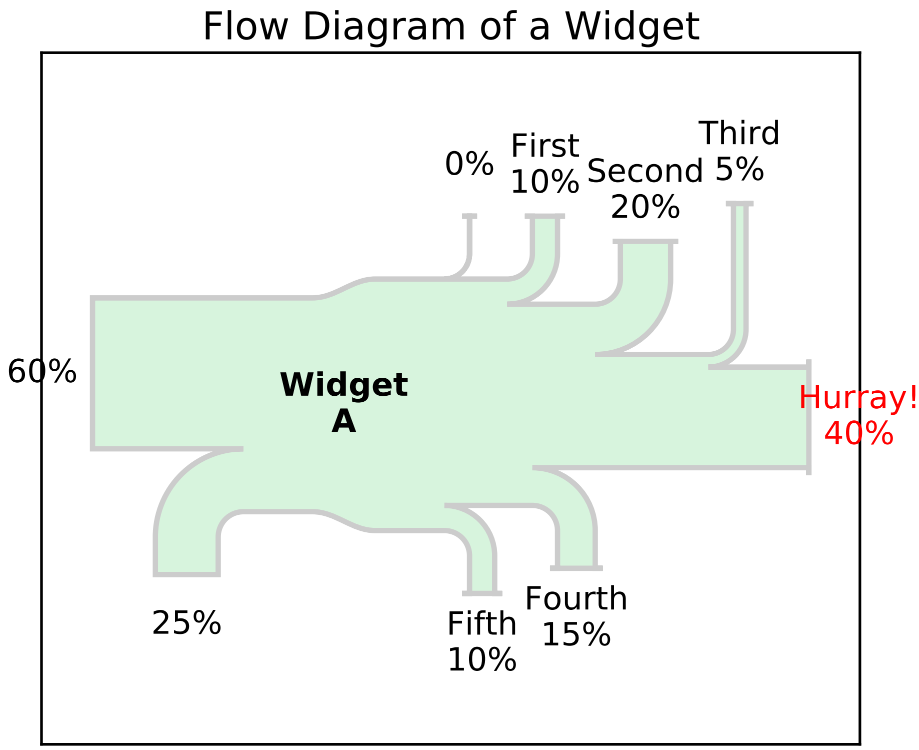
```
    This polygon is circumscribed by a unit circle centered at (0.5, 0.5)
    """
    x0, y0, r = [0.5] * 3
    verts = [(r*np.cos(t) + x0, r*np.sin(t) + y0) for t in theta]
    return verts


def example_data():
    # The following data is from the Denver Aerosol Sources and Health study.
    # See  doi:10.1016/j.atmosenv.2008.12.017
    #
    # The data are pollution source profile estimates for five modeled
    # pollution sources (e.g., cars, wood-burning, etc) that emit 7-9 chemical
    # species. The radar charts are experimented with here to see if we can
    # nicely visualize how the modeled source profiles change across four
    # scenarios:
    #  1) No gas-phase species present, just seven particulate counts on
    #     Sulfate
    #     Nitrate
    #     Elemental Carbon (EC)
    #     Organic Carbon fraction 1 (OC)
    #     Organic Carbon fraction 2 (OC2)
    #     Organic Carbon fraction 3 (OC3)
    #     Pyrolized Organic Carbon (OP)
    #  2)Inclusion of gas-phase specie carbon monoxide (CO)
    #  3)Inclusion of gas-phase specie ozone (O3).
    #  4)Inclusion of both gas-phase speciesis present...
    data = [
        ['Sulfate', 'Nitrate', 'EC', 'OC1', 'OC2', 'OC3', 'OP', 'CO', 'O3'],
        ('Basecase', [
            [0.88, 0.01, 0.03, 0.03, 0.00, 0.06, 0.01, 0.00, 0.00],
            [0.07, 0.95, 0.04, 0.05, 0.00, 0.02, 0.01, 0.00, 0.00],
            [0.01, 0.02, 0.85, 0.19, 0.05, 0.10, 0.00, 0.00, 0.00],
            [0.02, 0.01, 0.07, 0.01, 0.21, 0.12, 0.98, 0.00, 0.00],
            [0.01, 0.01, 0.02, 0.71, 0.74, 0.70, 0.00, 0.00, 0.00]]),
        ('With CO', [
            [0.88, 0.02, 0.02, 0.02, 0.00, 0.05, 0.00, 0.05, 0.00],
            [0.08, 0.94, 0.04, 0.02, 0.00, 0.01, 0.12, 0.04, 0.00],
            [0.01, 0.01, 0.79, 0.10, 0.00, 0.05, 0.00, 0.31, 0.00],
            [0.00, 0.02, 0.03, 0.38, 0.31, 0.31, 0.00, 0.59, 0.00],
            [0.02, 0.02, 0.11, 0.47, 0.69, 0.58, 0.88, 0.00, 0.00]]),
        ('With O3', [
            [0.89, 0.01, 0.07, 0.00, 0.00, 0.05, 0.00, 0.00, 0.03],
            [0.07, 0.95, 0.05, 0.04, 0.00, 0.02, 0.12, 0.00, 0.00],
            [0.01, 0.02, 0.86, 0.27, 0.16, 0.19, 0.00, 0.00, 0.00],
            [0.01, 0.03, 0.00, 0.32, 0.29, 0.27, 0.00, 0.00, 0.95],
            [0.02, 0.00, 0.03, 0.37, 0.56, 0.47, 0.87, 0.00, 0.00]]),
        ('CO & O3', [
            [0.87, 0.01, 0.08, 0.00, 0.00, 0.04, 0.00, 0.00, 0.01],
            [0.09, 0.95, 0.02, 0.03, 0.00, 0.01, 0.13, 0.06, 0.00],
            [0.01, 0.02, 0.71, 0.24, 0.13, 0.16, 0.00, 0.50, 0.00],
            [0.01, 0.03, 0.00, 0.28, 0.24, 0.23, 0.00, 0.44, 0.88],
```

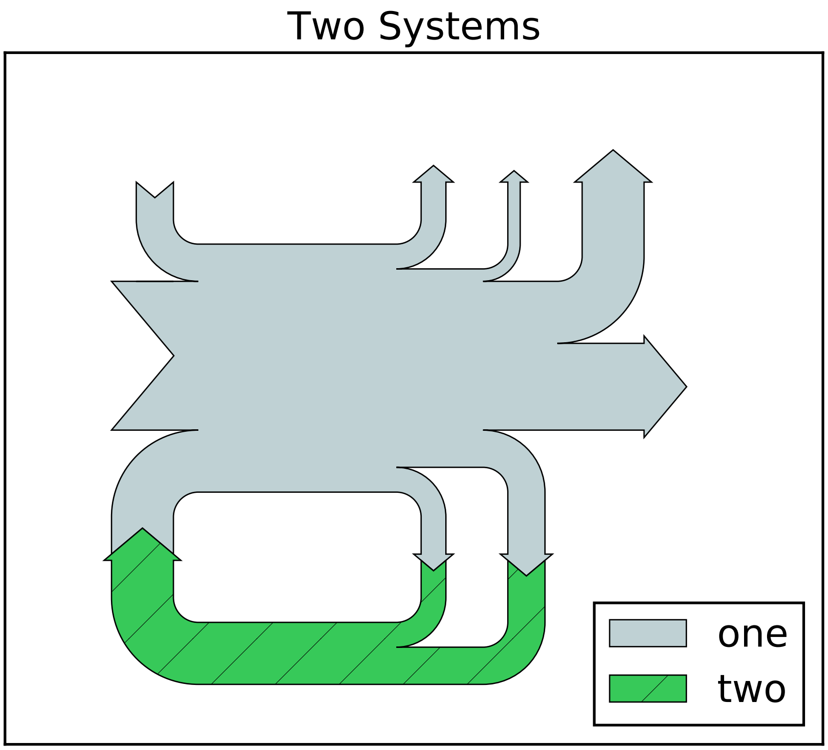
```
            [0.02, 0.00, 0.18, 0.45, 0.64, 0.55, 0.86, 0.00, 0.16]])
    ]
    return data


if __name__ == '__main__':
    N = 9
    theta = radar_factory(N, frame='polygon')

    data = example_data()
    spoke_labels = data.pop(0)

    fig = plt.figure(figsize=(9, 9))
    fig.subplots_adjust(wspace=0.25, hspace=0.20, top=0.85, bottom=0.05)

    colors = ['b', 'r', 'g', 'm', 'y']
    # Plot the four cases from the example data on separate axes
    for n, (title, case_data) in enumerate(data):
        ax = fig.add_subplot(2, 2, n + 1, projection='radar')
        plt.rgrids([0.2, 0.4, 0.6, 0.8])
        ax.set_title(title, weight='bold', size='medium', position=(0.5, 1.1),
                     horizontalalignment='center', verticalalignment='center')
        for d, color in zip(case_data, colors):
            ax.plot(theta, d, color=color)
            ax.fill(theta, d, facecolor=color, alpha=0.25)
        ax.set_varlabels(spoke_labels)

    # add legend relative to top-left plot
    plt.subplot(2, 2, 1)
    labels = ('Factor 1', 'Factor 2', 'Factor 3', 'Factor 4', 'Factor 5')
    legend = plt.legend(labels, loc=(0.9, .95), labelspacing=0.1)
    plt.setp(legend.get_texts(), fontsize='small')

    plt.figtext(0.5, 0.965, '5-Factor Solution Profiles Across Four Scenarios',
                ha='center', color='black', weight='bold', size='large')
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.28 api example code: sankey_demo_basics.py




Flow Diagram of a Widget
0%
First
10%
Second
20%
Third
5%
60%
Widget
A
Hurray!
40%
25%
Fifth
10%
Fourth
15%

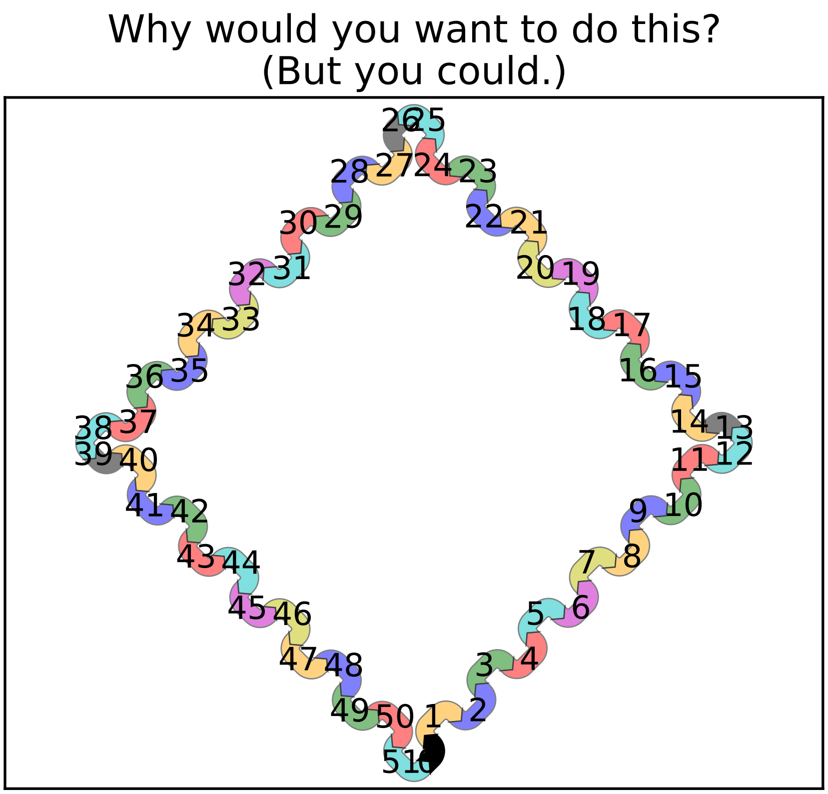



```
"""Demonstrate the Sankey class by producing three basic diagrams.
"""
import numpy as np
import matplotlib.pyplot as plt

from matplotlib.sankey import Sankey


# Example 1 -- Mostly defaults
# This demonstrates how to create a simple diagram by implicitly calling the
# Sankey.add() method and by appending finish() to the call to the class.
Sankey(flows=[0.25, 0.15, 0.60, -0.20, -0.15, -0.05, -0.50, -0.10],
       labels=['', '', '', 'First', 'Second', 'Third', 'Fourth', 'Fifth'],
       orientations=[-1, 1, 0, 1, 1, 1, 0, -1]).finish()
plt.title("The default settings produce a diagram like this.")
# Notice:
#   1. Axes weren't provided when Sankey() was instantiated, so they were
#      created automatically.
#   2. The scale argument wasn't necessary since the data was already
#      normalized.
#   3. By default, the lengths of the paths are justified.

# Example 2
# This demonstrates:
#   1. Setting one path longer than the others
#   2. Placing a label in the middle of the diagram
```

```
#   3. Using the scale argument to normalize the flows
#   4. Implicitly passing keyword arguments to PathPatch()
#   5. Changing the angle of the arrow heads
#   6. Changing the offset between the tips of the paths and their labels
#   7. Formatting the numbers in the path labels and the associated unit
#   8. Changing the appearance of the patch and the labels after the figure is
#      created
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
                     title="Flow Diagram of a Widget")
sankey = Sankey(ax=ax, scale=0.01, offset=0.2, head_angle=180,
                format='%.0f', unit='%')
sankey.add(flows=[25, 0, 60, -10, -20, -5, -15, -10, -40],
           labels=['', '', '', 'First', 'Second', 'Third', 'Fourth',
                   'Fifth', 'Hurray!'],
           orientations=[-1, 1, 0, 1, 1, 1, -1, -1, 0],
           pathlengths=[0.25, 0.25, 0.25, 0.25, 0.25, 0.6, 0.25, 0.25,
                        0.25],
           patchlabel="Widget\nA",
           alpha=0.2, lw=2.0)  # Arguments to matplotlib.patches.PathPatch()
diagrams = sankey.finish()
diagrams[0].patch.set_facecolor('#37c959')
diagrams[0].texts[-1].set_color('r')
diagrams[0].text.set_fontweight('bold')
# Notice:
#   1. Since the sum of the flows is nonzero, the width of the trunk isn't
#      uniform.  If verbose.level is helpful (in matplotlibrc), a message is
#      given in the terminal window.
#   2. The second flow doesn't appear because its value is zero.  Again, if
#      verbose.level is helpful, a message is given in the terminal window.

# Example 3
# This demonstrates:
#   1. Connecting two systems
#   2. Turning off the labels of the quantities
#   3. Adding a legend
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[], title="Two Systems")
flows = [0.25, 0.15, 0.60, -0.10, -0.05, -0.25, -0.15, -0.10, -0.35]
sankey = Sankey(ax=ax, unit=None)
sankey.add(flows=flows, label='one',
           orientations=[-1, 1, 0, 1, 1, 1, -1, -1, 0])
sankey.add(flows=[-0.25, 0.15, 0.1], fc='#37c959', label='two',
           orientations=[-1, -1, -1], prior=0, connect=(0, 0))
diagrams = sankey.finish()
diagrams[-1].patch.set_hatch('/')
plt.legend(loc='best')
# Notice that only one connection is specified, but the systems form a
# circuit since: (1) the lengths of the paths are justified and (2) the
# orientation and ordering of the flows is mirrored.

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.29 api example code: sankey_demo_links.py

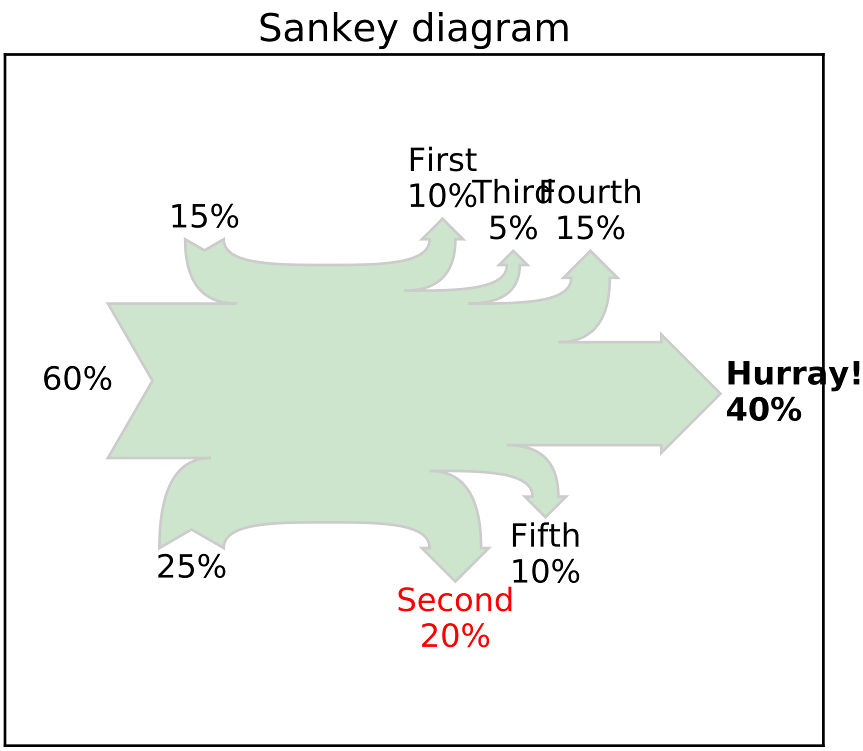



```
"""Demonstrate/test the Sankey class by producing a long chain of connections.
"""

from itertools import cycle

import matplotlib.pyplot as plt
from matplotlib.sankey import Sankey

links_per_side = 6


def side(sankey, n=1):
    """Generate a side chain."""
    prior = len(sankey.diagrams)
    colors = cycle(['orange', 'b', 'g', 'r', 'c', 'm', 'y'])
    for i in range(0, 2*n, 2):
        sankey.add(flows=[1, -1], orientations=[-1, -1],
                   patchlabel=str(prior + i), facecolor=next(colors),
                   prior=prior + i - 1, connect=(1, 0), alpha=0.5)
```

```
        sankey.add(flows=[1, -1], orientations=[1, 1],
                   patchlabel=str(prior + i + 1), facecolor=next(colors),
                   prior=prior + i, connect=(1, 0), alpha=0.5)


def corner(sankey):
    """Generate a corner link."""
    prior = len(sankey.diagrams)
    sankey.add(flows=[1, -1], orientations=[0, 1],
               patchlabel=str(prior), facecolor='k',
               prior=prior - 1, connect=(1, 0), alpha=0.5)


fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
                     title="Why would you want to do this?\n(But you could.)")
sankey = Sankey(ax=ax, unit=None)
sankey.add(flows=[1, -1], orientations=[0, 1],
           patchlabel="0", facecolor='k',
           rotation=45)
side(sankey, n=links_per_side)
corner(sankey)
side(sankey, n=links_per_side)
corner(sankey)
side(sankey, n=links_per_side)
corner(sankey)
side(sankey, n=links_per_side)
sankey.finish()
# Notice:
# 1. The alignment doesn't drift significantly (if at all; with 16007
#    subdiagrams there is still closure).
# 2. The first diagram is rotated 45 deg, so all other diagrams are rotated
#    accordingly.

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.30 api example code: sankey_demo_old.py




```
#!/usr/bin/env python

from __future__ import print_function

__author__ = "Yannick Copin <ycopin@ipnl.in2p3.fr>"
__version__ = "Time-stamp: <10/02/2010 16:49 ycopin@lyopc548.in2p3.fr>"

import numpy as np


def sankey(ax,
           outputs=[100.], outlabels=None,
           inputs=[100.], inlabels='',
           dx=40, dy=10, outangle=45, w=3, inangle=30, offset=2, **kwargs):
    """Draw a Sankey diagram.

    outputs: array of outputs, should sum up to 100%
    outlabels: output labels (same length as outputs),
    or None (use default labels) or '' (no labels)
    inputs and inlabels: similar for inputs
    dx: horizontal elongation
    dy: vertical elongation
```

```
    outangle: output arrow angle [deg]
    w: output arrow shoulder
    inangle: input dip angle
    offset: text offset
    **kwargs: propagated to Patch (e.g., fill=False)

    Return (patch,[intexts,outtexts]).
    """
    import matplotlib.patches as mpatches
    from matplotlib.path import Path

    outs = np.absolute(outputs)
    outsigns = np.sign(outputs)
    outsigns[-1] = 0  # Last output

    ins = np.absolute(inputs)
    insigns = np.sign(inputs)
    insigns[0] = 0  # First input

    assert sum(outs) == 100, "Outputs don't sum up to 100%"
    assert sum(ins) == 100, "Inputs don't sum up to 100%"

    def add_output(path, loss, sign=1):
        # Arrow tip height
        h = (loss/2 + w) * np.tan(np.radians(outangle))
        move, (x, y) = path[-1]  # Use last point as reference
        if sign == 0:  # Final loss (horizontal)
            path.extend([(Path.LINETO, [x + dx, y]),
                         (Path.LINETO, [x + dx, y + w]),
                         (Path.LINETO, [x + dx + h, y - loss/2]),  # Tip
                         (Path.LINETO, [x + dx, y - loss - w]),
                         (Path.LINETO, [x + dx, y - loss])])
            outtips.append((sign, path[-3][1]))
        else:  # Intermediate loss (vertical)
            path.extend([(Path.CURVE4, [x + dx/2, y]),
                         (Path.CURVE4, [x + dx, y]),
                         (Path.CURVE4, [x + dx, y + sign*dy]),
                         (Path.LINETO, [x + dx - w, y + sign*dy]),
                         # Tip
                         (Path.LINETO, [
                          x + dx + loss/2, y + sign*(dy + h)]),
                         (Path.LINETO, [x + dx + loss + w, y + sign*dy]),
                         (Path.LINETO, [x + dx + loss, y + sign*dy]),
                         (Path.CURVE3, [x + dx + loss, y - sign*loss]),
                         (Path.CURVE3, [x + dx/2 + loss, y - sign*loss])])
            outtips.append((sign, path[-5][1]))

    def add_input(path, gain, sign=1):
        h = (gain / 2) * np.tan(np.radians(inangle))  # Dip depth
        move, (x, y) = path[-1]  # Use last point as reference
        if sign == 0:  # First gain (horizontal)
            path.extend([(Path.LINETO, [x - dx, y]),
                         (Path.LINETO, [x - dx + h, y + gain/2]),  # Dip
```

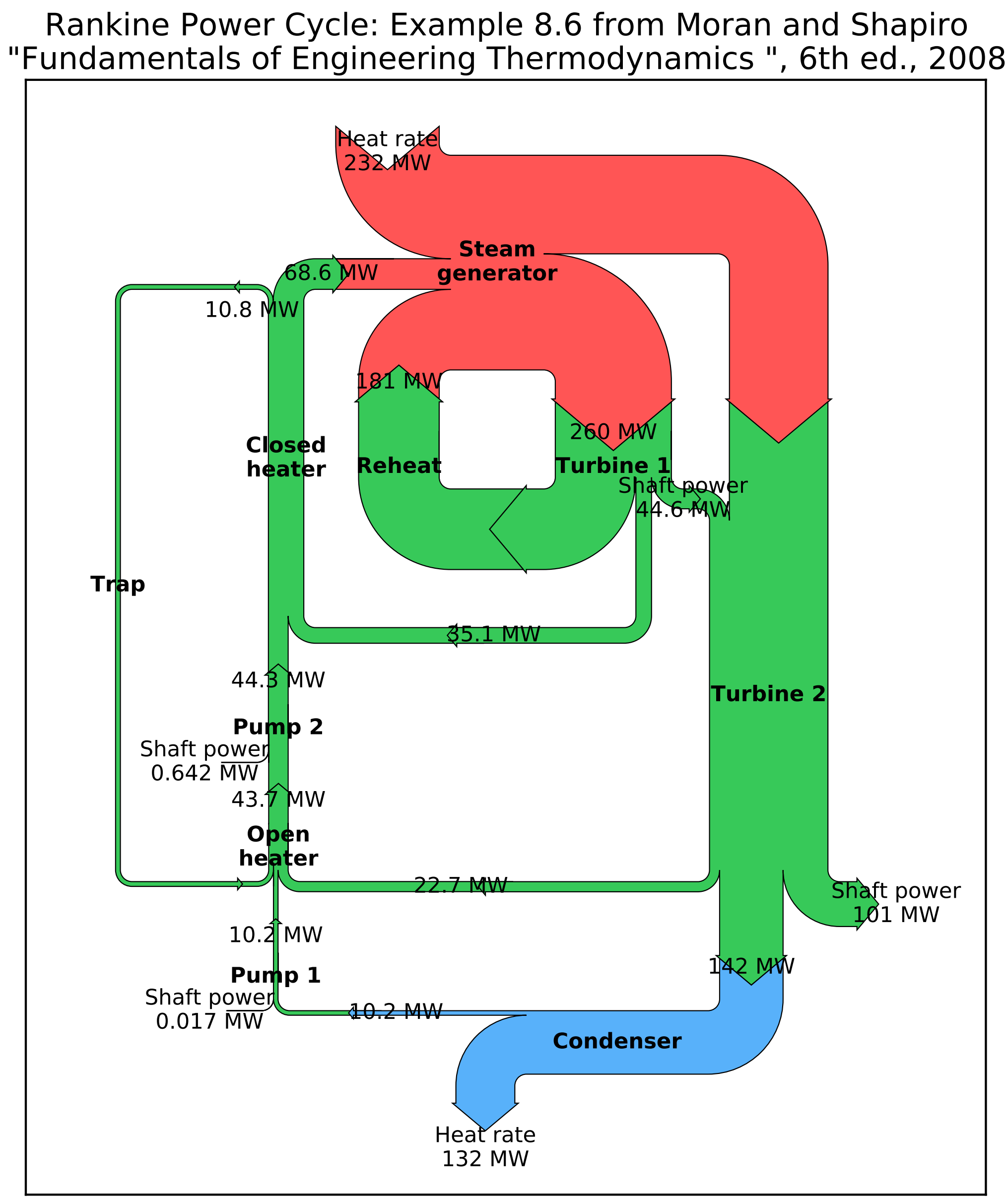
```
                         (Path.LINETO, [x - dx, y + gain])])
            xd, yd = path[-2][1]  # Dip position
            indips.append((sign, [xd - h, yd]))
        else:  # Intermediate gain (vertical)
            path.extend([(Path.CURVE4, [x - dx/2, y]),
                         (Path.CURVE4, [x - dx, y]),
                         (Path.CURVE4, [x - dx, y + sign*dy]),
                         # Dip
                         (Path.LINETO, [
                          x - dx - gain / 2, y + sign*(dy - h)]),
                         (Path.LINETO, [x - dx - gain, y + sign*dy]),
                         (Path.CURVE3, [x - dx - gain, y - sign*gain]),
                         (Path.CURVE3, [x - dx/2 - gain, y - sign*gain])])
            xd, yd = path[-4][1]  # Dip position
            indips.append((sign, [xd, yd + sign*h]))

    outtips = []  # Output arrow tip dir. and positions
    urpath = [(Path.MOVETO, [0, 100])]  # 1st point of upper right path
    lrpath = [(Path.LINETO, [0, 0])]  # 1st point of lower right path
    for loss, sign in zip(outs, outsigns):
        add_output(sign >= 0 and urpath or lrpath, loss, sign=sign)

    indips = []  # Input arrow tip dir. and positions
    llpath = [(Path.LINETO, [0, 0])]  # 1st point of lower left path
    ulpath = [(Path.MOVETO, [0, 100])]  # 1st point of upper left path
    for gain, sign in reversed(list(zip(ins, insigns))):
        add_input(sign <= 0 and llpath or ulpath, gain, sign=sign)

    def revert(path):
        """A path is not just revertable by path[::-1] because of Bezier
        curves."""
        rpath = []
        nextmove = Path.LINETO
        for move, pos in path[::-1]:
            rpath.append((nextmove, pos))
            nextmove = move
        return rpath

    # Concatenate subpathes in correct order
    path = urpath + revert(lrpath) + llpath + revert(ulpath)

    codes, verts = zip(*path)
    verts = np.array(verts)

    # Path patch
    path = Path(verts, codes)
    patch = mpatches.PathPatch(path, **kwargs)
    ax.add_patch(patch)

    if False:  # DEBUG
        print("urpath", urpath)
        print("lrpath", revert(lrpath))
        print("llpath", llpath)
```

```
        print("ulpath", revert(ulpath))
        xs, ys = zip(*verts)
        ax.plot(xs, ys, 'go-')

    # Labels

    def set_labels(labels, values):
        """Set or check labels according to values."""
        if labels == '':  # No labels
            return labels
        elif labels is None:  # Default labels
            return ['%2d%%' % val for val in values]
        else:
            assert len(labels) == len(values)
            return labels

    def put_labels(labels, positions, output=True):
        """Put labels to positions."""
        texts = []
        lbls = output and labels or labels[::-1]
        for i, label in enumerate(lbls):
            s, (x, y) = positions[i]  # Label direction and position
            if s == 0:
                t = ax.text(x + offset, y, label,
                            ha=output and 'left' or 'right', va='center')
            elif s > 0:
                t = ax.text(x, y + offset, label, ha='center', va='bottom')
            else:
                t = ax.text(x, y - offset, label, ha='center', va='top')
            texts.append(t)
        return texts

    outlabels = set_labels(outlabels, outs)
    outtexts = put_labels(outlabels, outtips, output=True)

    inlabels = set_labels(inlabels, ins)
    intexts = put_labels(inlabels, indips, output=False)

    # Axes management
    ax.set_xlim(verts[:, 0].min() - dx, verts[:, 0].max() + dx)
    ax.set_ylim(verts[:, 1].min() - dy, verts[:, 1].max() + dy)
    ax.set_aspect('equal', adjustable='datalim')

    return patch, [intexts, outtexts]


if __name__ == '__main__':

    import matplotlib.pyplot as plt

    outputs = [10., -20., 5., 15., -10., 40.]
    outlabels = ['First', 'Second', 'Third', 'Fourth', 'Fifth', 'Hurray!']
    outlabels = [s + '\n%d%%' % abs(l) for l, s in zip(outputs, outlabels)]
```

```
inputs = [60., -25., 15.]

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[], title="Sankey diagram")

patch, (intexts, outtexts) = sankey(ax, outputs=outputs,
                                    outlabels=outlabels, inputs=inputs,
                                    inlabels=None, fc='g', alpha=0.2)
outtexts[1].set_color('r')
outtexts[-1].set_fontweight('bold')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.31 api example code: sankey_demo_rankine.py



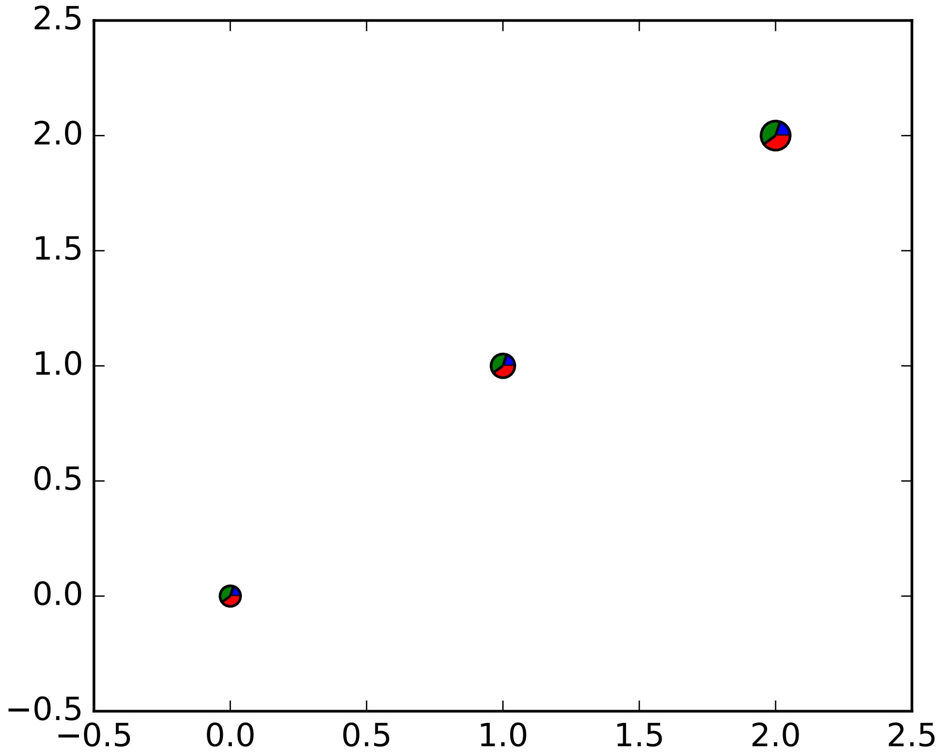

```
"""Demonstrate the Sankey class with a practicle example of a Rankine power
cycle.

"""
import matplotlib.pyplot as plt
```

```python
from matplotlib.sankey import Sankey

fig = plt.figure(figsize=(8, 9))
ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[],
                     title="Rankine Power Cycle: Example 8.6 from Moran and "
                     "Shapiro\n\x22Fundamentals of Engineering Thermodynamics "
                     "\x22, 6th ed., 2008")
Hdot = [260.431, 35.078, 180.794, 221.115, 22.700,
        142.361, 10.193, 10.210, 43.670, 44.312,
        68.631, 10.758, 10.758, 0.017, 0.642,
        232.121, 44.559, 100.613, 132.168]  # MW
sankey = Sankey(ax=ax, format='%.3G', unit=' MW', gap=0.5, scale=1.0/Hdot[0])
sankey.add(patchlabel='\n\nPump 1', rotation=90, facecolor='#37c959',
           flows=[Hdot[13], Hdot[6], -Hdot[7]],
           labels=['Shaft power', '', None],
           pathlengths=[0.4, 0.883, 0.25],
           orientations=[1, -1, 0])
sankey.add(patchlabel='\n\nOpen\nheater', facecolor='#37c959',
           flows=[Hdot[11], Hdot[7], Hdot[4], -Hdot[8]],
           labels=[None, '', None, None],
           pathlengths=[0.25, 0.25, 1.93, 0.25],
           orientations=[1, 0, -1, 0], prior=0, connect=(2, 1))
sankey.add(patchlabel='\n\nPump 2', facecolor='#37c959',
           flows=[Hdot[14], Hdot[8], -Hdot[9]],
           labels=['Shaft power', '', None],
           pathlengths=[0.4, 0.25, 0.25],
           orientations=[1, 0, 0], prior=1, connect=(3, 1))
sankey.add(patchlabel='Closed\nheater', trunklength=2.914, fc='#37c959',
           flows=[Hdot[9], Hdot[1], -Hdot[11], -Hdot[10]],
           pathlengths=[0.25, 1.543, 0.25, 0.25],
           labels=['', '', None, None],
           orientations=[0, -1, 1, -1], prior=2, connect=(2, 0))
sankey.add(patchlabel='Trap', facecolor='#37c959', trunklength=5.102,
           flows=[Hdot[11], -Hdot[12]],
           labels=['\n', None],
           pathlengths=[1.0, 1.01],
           orientations=[1, 1], prior=3, connect=(2, 0))
sankey.add(patchlabel='Steam\ngenerator', facecolor='#ff5555',
           flows=[Hdot[15], Hdot[10], Hdot[2], -Hdot[3], -Hdot[0]],
           labels=['Heat rate', '', '', None, None],
           pathlengths=0.25,
           orientations=[1, 0, -1, -1, -1], prior=3, connect=(3, 1))
sankey.add(patchlabel='\n\n\nTurbine 1', facecolor='#37c959',
           flows=[Hdot[0], -Hdot[16], -Hdot[1], -Hdot[2]],
           labels=['', None, None, None],
           pathlengths=[0.25, 0.153, 1.543, 0.25],
           orientations=[0, 1, -1, -1], prior=5, connect=(4, 0))
sankey.add(patchlabel='\n\n\nReheat', facecolor='#37c959',
           flows=[Hdot[2], -Hdot[2]],
           labels=[None, None],
           pathlengths=[0.725, 0.25],
           orientations=[-1, 0], prior=6, connect=(3, 0))
sankey.add(patchlabel='Turbine 2', trunklength=3.212, facecolor='#37c959',
```

```
           flows=[Hdot[3], Hdot[16], -Hdot[5], -Hdot[4], -Hdot[17]],
           labels=[None, 'Shaft power', None, '', 'Shaft power'],
           pathlengths=[0.751, 0.15, 0.25, 1.93, 0.25],
           orientations=[0, -1, 0, -1, 1], prior=6, connect=(1, 1))
sankey.add(patchlabel='Condenser', facecolor='#58b1fa', trunklength=1.764,
           flows=[Hdot[5], -Hdot[18], -Hdot[6]],
           labels=['', 'Heat rate', None],
           pathlengths=[0.45, 0.25, 0.883],
           orientations=[-1, 1, 0], prior=8, connect=(2, 0))
diagrams = sankey.finish()
for diagram in diagrams:
    diagram.text.set_fontweight('bold')
    diagram.text.set_fontsize('10')
    for text in diagram.texts:
        text.set_fontsize('10')
# Notice that the explicit connections are handled automatically, but the
# implicit ones currently are not.  The lengths of the paths and the trunks
# must be adjusted manually, and that is a bit tricky.

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.32 api example code: scatter_piecharts.py

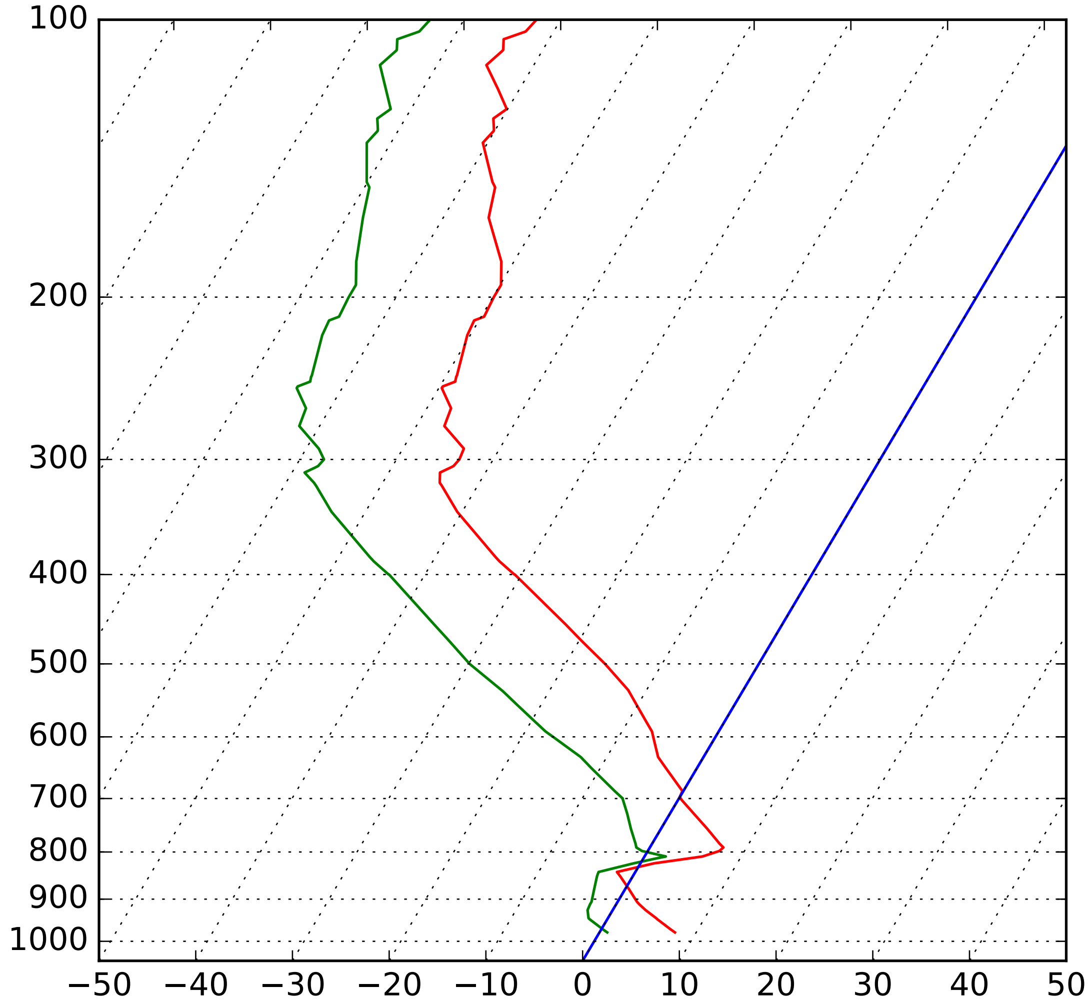



```
"""
This example makes custom 'pie charts' as the markers for a scatter plotqu

Thanks to Manuel Metz for the example
"""
import math
import numpy as np
import matplotlib.pyplot as plt

# first define the ratios
r1 = 0.2       # 20%
r2 = r1 + 0.4  # 40%

# define some sizes of the scatter marker
sizes = [60, 80, 120]

# calculate the points of the first pie marker
#
# these are just the origin (0,0) +
# some points on a circle cos,sin
x = [0] + np.cos(np.linspace(0, 2*math.pi*r1, 10)).tolist()
y = [0] + np.sin(np.linspace(0, 2*math.pi*r1, 10)).tolist()
```

```
xy1 = list(zip(x, y))
s1 = max(max(x), max(y))

# ...
x = [0] + np.cos(np.linspace(2*math.pi*r1, 2*math.pi*r2, 10)).tolist()
y = [0] + np.sin(np.linspace(2*math.pi*r1, 2*math.pi*r2, 10)).tolist()
xy2 = list(zip(x, y))
s2 = max(max(x), max(y))

x = [0] + np.cos(np.linspace(2*math.pi*r2, 2*math.pi, 10)).tolist()
y = [0] + np.sin(np.linspace(2*math.pi*r2, 2*math.pi, 10)).tolist()
xy3 = list(zip(x, y))
s3 = max(max(x), max(y))

fig, ax = plt.subplots()
ax.scatter(np.arange(3), np.arange(3), marker=(xy1, 0),
           s=[s1*s1*_ for _ in sizes], facecolor='blue')
ax.scatter(np.arange(3), np.arange(3), marker=(xy2, 0),
           s=[s2*s2*_ for _ in sizes], facecolor='green')
ax.scatter(np.arange(3), np.arange(3), marker=(xy3, 0),
           s=[s3*s3*_ for _ in sizes], facecolor='red')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.33 api example code: skewt.py




```python
# This serves as an intensive exercise of matplotlib's transforms
# and custom projection API. This example produces a so-called
# SkewT-logP diagram, which is a common plot in meteorology for
# displaying vertical profiles of temperature. As far as matplotlib is
# concerned, the complexity comes from having X and Y axes that are
# not orthogonal. This is handled by including a skew component to the
# basic Axes transforms. Additional complexity comes in handling the
# fact that the upper and lower X-axes have different data ranges, which
# necessitates a bunch of custom classes for ticks,spines, and the axis
# to handle this.

from matplotlib.axes import Axes
import matplotlib.transforms as transforms
```

```python
import matplotlib.axis as maxis
import matplotlib.spines as mspines
import matplotlib.path as mpath
from matplotlib.projections import register_projection

# The sole purpose of this class is to look at the upper, lower, or total
# interval as appropriate and see what parts of the tick to draw, if any.


class SkewXTick(maxis.XTick):
    def draw(self, renderer):
        if not self.get_visible():
            return
        renderer.open_group(self.__name__)

        lower_interval = self.axes.xaxis.lower_interval
        upper_interval = self.axes.xaxis.upper_interval

        if self.gridOn and transforms.interval_contains(
                self.axes.xaxis.get_view_interval(), self.get_loc()):
            self.gridline.draw(renderer)

        if transforms.interval_contains(lower_interval, self.get_loc()):
            if self.tick1On:
                self.tick1line.draw(renderer)
            if self.label1On:
                self.label1.draw(renderer)

        if transforms.interval_contains(upper_interval, self.get_loc()):
            if self.tick2On:
                self.tick2line.draw(renderer)
            if self.label2On:
                self.label2.draw(renderer)

        renderer.close_group(self.__name__)


# This class exists to provide two separate sets of intervals to the tick,
# as well as create instances of the custom tick
class SkewXAxis(maxis.XAxis):
    def __init__(self, *args, **kwargs):
        maxis.XAxis.__init__(self, *args, **kwargs)
        self.upper_interval = 0.0, 1.0

    def _get_tick(self, major):
        return SkewXTick(self.axes, 0, '', major=major)

    @property
    def lower_interval(self):
        return self.axes.viewLim.intervalx

    def get_view_interval(self):
        return self.upper_interval[0], self.axes.viewLim.intervalx[1]
```

```python
# This class exists to calculate the separate data range of the
# upper X-axis and draw the spine there. It also provides this range
# to the X-axis artist for ticking and gridlines
class SkewSpine(mspines.Spine):
    def _adjust_location(self):
        trans = self.axes.transDataToAxes.inverted()
        if self.spine_type == 'top':
            yloc = 1.0
        else:
            yloc = 0.0
        left = trans.transform_point((0.0, yloc))[0]
        right = trans.transform_point((1.0, yloc))[0]

        pts = self._path.vertices
        pts[0, 0] = left
        pts[1, 0] = right
        self.axis.upper_interval = (left, right)


# This class handles registration of the skew-xaxes as a projection as well
# as setting up the appropriate transformations. It also overrides standard
# spines and axes instances as appropriate.
class SkewXAxes(Axes):
    # The projection must specify a name.  This will be used be the
    # user to select the projection, i.e. ``subplot(111,
    # projection='skewx')``.
    name = 'skewx'

    def _init_axis(self):
        # Taken from Axes and modified to use our modified X-axis
        self.xaxis = SkewXAxis(self)
        self.spines['top'].register_axis(self.xaxis)
        self.spines['bottom'].register_axis(self.xaxis)
        self.yaxis = maxis.YAxis(self)
        self.spines['left'].register_axis(self.yaxis)
        self.spines['right'].register_axis(self.yaxis)

    def _gen_axes_spines(self):
        spines = {'top': SkewSpine.linear_spine(self, 'top'),
                  'bottom': mspines.Spine.linear_spine(self, 'bottom'),
                  'left': mspines.Spine.linear_spine(self, 'left'),
                  'right': mspines.Spine.linear_spine(self, 'right')}
        return spines

    def _set_lim_and_transforms(self):
        """
        This is called once when the plot is created to set up all the
        transforms for the data, text and grids.
        """
        rot = 30
```

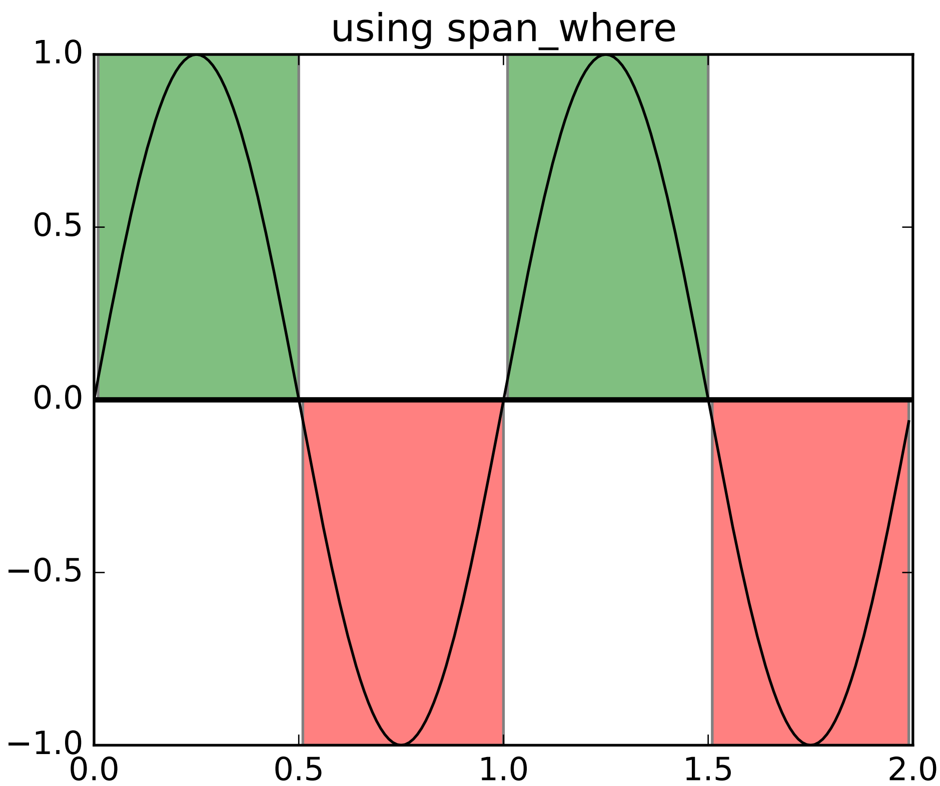
```
        # Get the standard transform setup from the Axes base class
        Axes._set_lim_and_transforms(self)

        # Need to put the skew in the middle, after the scale and limits,
        # but before the transAxes. This way, the skew is done in Axes
        # coordinates thus performing the transform around the proper origin
        # We keep the pre-transAxes transform around for other users, like the
        # spines for finding bounds
        self.transDataToAxes = self.transScale + \
            self.transLimits + transforms.Affine2D().skew_deg(rot, 0)

        # Create the full transform from Data to Pixels
        self.transData = self.transDataToAxes + self.transAxes

        # Blended transforms like this need to have the skewing applied using
        # both axes, in axes coords like before.
        self._xaxis_transform = (transforms.blended_transform_factory(
            self.transScale + self.transLimits,
            transforms.IdentityTransform()) +
            transforms.Affine2D().skew_deg(rot, 0)) + self.transAxes

# Now register the projection with matplotlib so the user can select
# it.
register_projection(SkewXAxes)

if __name__ == '__main__':
    # Now make a simple example using the custom projection.
    from matplotlib.ticker import ScalarFormatter, MultipleLocator
    import matplotlib.pyplot as plt
    from six import StringIO
    import numpy as np

    # Some examples data
    data_txt = '''
  978.0    345    7.8    0.8     61   4.16    325     14  282.7  294.6  283.4
  971.0    404    7.2    0.2     61   4.01    327     17  282.7  294.2  283.4
  946.7    610    5.2   -1.8     61   3.56    335     26  282.8  293.0  283.4
  944.0    634    5.0   -2.0     61   3.51    336     27  282.8  292.9  283.4
  925.0    798    3.4   -2.6     65   3.43    340     32  282.8  292.7  283.4
  911.8    914    2.4   -2.7     69   3.46    345     37  282.9  292.9  283.5
  906.0    966    2.0   -2.7     71   3.47    348     39  283.0  293.0  283.6
  877.9   1219    0.4   -3.2     77   3.46      0     48  283.9  293.9  284.5
  850.0   1478   -1.3   -3.7     84   3.44      0     47  284.8  294.8  285.4
  841.0   1563   -1.9   -3.8     87   3.45    358     45  285.0  295.0  285.6
  823.0   1736    1.4   -0.7     86   4.44    353     42  290.3  303.3  291.0
  813.6   1829    4.5    1.2     80   5.17    350     40  294.5  309.8  295.4
  809.0   1875    6.0    2.2     77   5.57    347     39  296.6  313.2  297.6
  798.0   1988    7.4   -0.6     57   4.61    340     35  299.2  313.3  300.1
  791.0   2061    7.6   -1.4     53   4.39    335     33  300.2  313.6  301.0
  783.9   2134    7.0   -1.7     54   4.32    330     31  300.4  313.6  301.2
  755.1   2438    4.8   -3.1     57   4.06    300     24  301.2  313.7  301.9
  727.3   2743    2.5   -4.4     60   3.81    285     29  301.9  313.8  302.6
  700.5   3048    0.2   -5.8     64   3.57    275     31  302.7  313.8  303.3
```

```
  700.0   3054    0.2   -5.8     64   3.56    280     31  302.7  313.8  303.3
  698.0   3077    0.0   -6.0     64   3.52    280     31  302.7  313.7  303.4
  687.0   3204   -0.1   -7.1     59   3.28    281     31  304.0  314.3  304.6
  648.9   3658   -3.2  -10.9     55   2.59    285     30  305.5  313.8  305.9
  631.0   3881   -4.7  -12.7     54   2.29    289     33  306.2  313.6  306.6
  600.7   4267   -6.4  -16.7     44   1.73    295     39  308.6  314.3  308.9
  592.0   4381   -6.9  -17.9     41   1.59    297     41  309.3  314.6  309.6
  577.6   4572   -8.1  -19.6     39   1.41    300     44  310.1  314.9  310.3
  555.3   4877  -10.0  -22.3     36   1.16    295     39  311.3  315.3  311.5
  536.0   5151  -11.7  -24.7     33   0.97    304     39  312.4  315.8  312.6
  533.8   5182  -11.9  -25.0     33   0.95    305     39  312.5  315.8  312.7
  500.0   5680  -15.9  -29.9     29   0.64    290     44  313.6  315.9  313.7
  472.3   6096  -19.7  -33.4     28   0.49    285     46  314.1  315.8  314.1
  453.0   6401  -22.4  -36.0     28   0.39    300     50  314.4  315.8  314.4
  400.0   7310  -30.7  -43.7     27   0.20    285     44  315.0  315.8  315.0
  399.7   7315  -30.8  -43.8     27   0.20    285     44  315.0  315.8  315.0
  387.0   7543  -33.1  -46.1     26   0.16    281     47  314.9  315.5  314.9
  382.7   7620  -33.8  -46.8     26   0.15    280     48  315.0  315.6  315.0
  342.0   8398  -40.5  -53.5     23   0.08    293     52  316.1  316.4  316.1
  320.4   8839  -43.7  -56.7     22   0.06    300     54  317.6  317.8  317.6
  318.0   8890  -44.1  -57.1     22   0.05    301     55  317.8  318.0  317.8
  310.0   9060  -44.7  -58.7     19   0.04    304     61  319.2  319.4  319.2
  306.1   9144  -43.9  -57.9     20   0.05    305     63  321.5  321.7  321.5
  305.0   9169  -43.7  -57.7     20   0.05    303     63  322.1  322.4  322.1
  300.0   9280  -43.5  -57.5     20   0.05    295     64  323.9  324.2  323.9
  292.0   9462  -43.7  -58.7     17   0.05    293     67  326.2  326.4  326.2
  276.0   9838  -47.1  -62.1     16   0.03    290     74  326.6  326.7  326.6
  264.0  10132  -47.5  -62.5     16   0.03    288     79  330.1  330.3  330.1
  251.0  10464  -49.7  -64.7     16   0.03    285     85  331.7  331.8  331.7
  250.0  10490  -49.7  -64.7     16   0.03    285     85  332.1  332.2  332.1
  247.0  10569  -48.7  -63.7     16   0.03    283     88  334.7  334.8  334.7
  244.0  10649  -48.9  -63.9     16   0.03    280     91  335.6  335.7  335.6
  243.3  10668  -48.9  -63.9     16   0.03    280     91  335.8  335.9  335.8
  220.0  11327  -50.3  -65.3     15   0.03    280     85  343.5  343.6  343.5
  212.0  11569  -50.5  -65.5     15   0.03    280     83  346.8  346.9  346.8
  210.0  11631  -49.7  -64.7     16   0.03    280     83  349.0  349.1  349.0
  200.0  11950  -49.9  -64.9     15   0.03    280     80  353.6  353.7  353.6
  194.0  12149  -49.9  -64.9     15   0.03    279     78  356.7  356.8  356.7
  183.0  12529  -51.3  -66.3     15   0.03    278     75  360.4  360.5  360.4
  164.0  13233  -55.3  -68.3     18   0.02    277     69  365.2  365.3  365.2
  152.0  13716  -56.5  -69.5     18   0.02    275     65  371.1  371.2  371.1
  150.0  13800  -57.1  -70.1     18   0.02    275     64  371.5  371.6  371.5
  136.0  14414  -60.5  -72.5     19   0.02    268     54  376.0  376.1  376.0
  132.0  14600  -60.1  -72.1     19   0.02    265     51  380.0  380.1  380.0
  131.4  14630  -60.2  -72.2     19   0.02    265     51  380.3  380.4  380.3
  128.0  14792  -60.9  -72.9     19   0.02    266     50  381.9  382.0  381.9
  125.0  14939  -60.1  -72.1     19   0.02    268     49  385.9  386.0  385.9
  119.0  15240  -62.2  -73.8     20   0.01    270     48  387.4  387.5  387.4
  112.0  15616  -64.9  -75.9     21   0.01    265     53  389.3  389.3  389.3
  108.0  15838  -64.1  -75.1     21   0.01    265     58  394.8  394.9  394.8
  107.8  15850  -64.1  -75.1     21   0.01    265     58  395.0  395.1  395.0
  105.0  16010  -64.7  -75.7     21   0.01    272     50  396.9  396.9  396.9
  103.0  16128  -62.9  -73.9     21   0.02    277     45  402.5  402.6  402.5
```

```
 100.0  16310  -62.5  -73.5     21   0.02    285     36  406.7  406.7  406.7
    '''

    # Parse the data
    sound_data = StringIO(data_txt)
    p, h, T, Td = np.loadtxt(sound_data, usecols=range(0, 4), unpack=True)

    # Create a new figure. The dimensions here give a good aspect ratio
    fig = plt.figure(figsize=(6.5875, 6.2125))
    ax = fig.add_subplot(111, projection='skewx')

    plt.grid(True)

    # Plot the data using normal plotting functions, in this case using
    # log scaling in Y, as dicatated by the typical meteorological plot
    ax.semilogy(T, p, 'r')
    ax.semilogy(Td, p, 'g')

    # An example of a slanted line at constant X
    l = ax.axvline(0, color='b')

    # Disables the log-formatting that comes with semilogy
    ax.yaxis.set_major_formatter(ScalarFormatter())
    ax.set_yticks(np.linspace(100, 1000, 10))
    ax.set_ylim(1050, 100)

    ax.xaxis.set_major_locator(MultipleLocator(10))
    ax.set_xlim(-50, 50)

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.34 api example code: span_regions.py

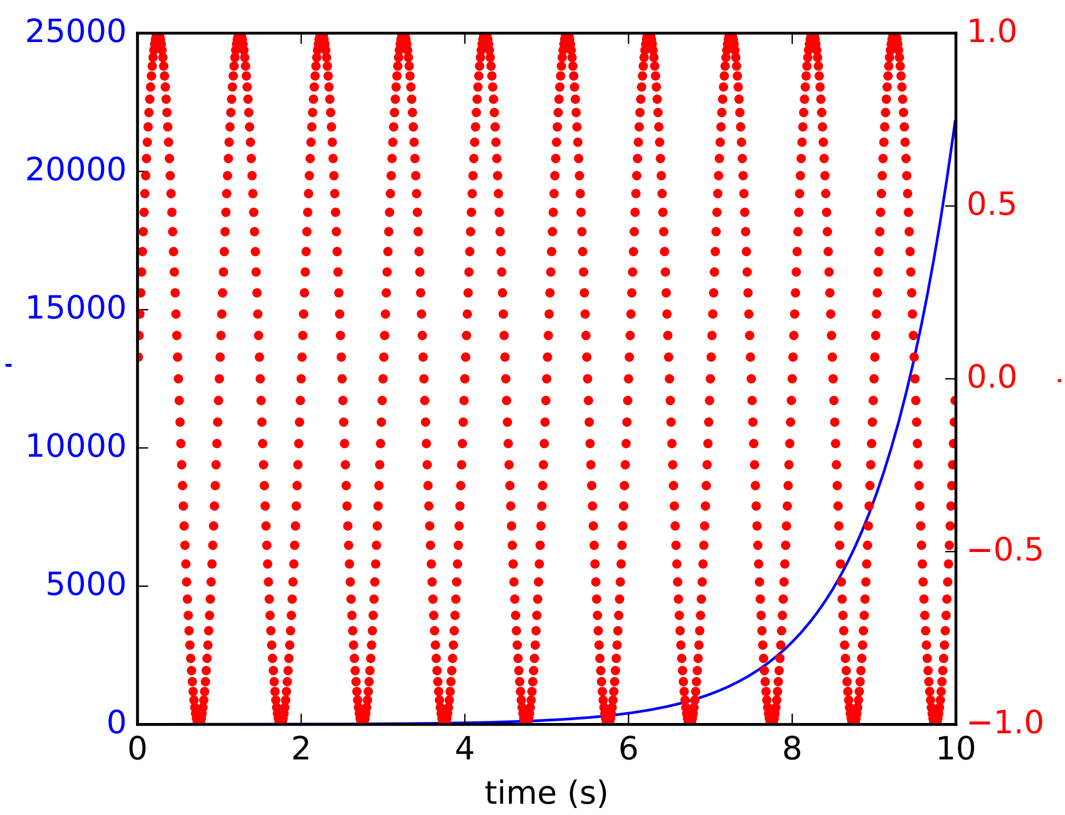



```
"""
Illustrate some helper functions for shading regions where a logical
mask is True

See :meth:`matplotlib.collections.BrokenBarHCollection.span_where`
"""
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.collections as collections


t = np.arange(0.0, 2, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = 1.2*np.sin(4*np.pi*t)


fig, ax = plt.subplots()
ax.set_title('using span_where')
ax.plot(t, s1, color='black')
ax.axhline(0, color='black', lw=2)

collection = collections.BrokenBarHCollection.span_where(
```

```
    t, ymin=0, ymax=1, where=s1 > 0, facecolor='green', alpha=0.5)
ax.add_collection(collection)

collection = collections.BrokenBarHCollection.span_where(
    t, ymin=-1, ymax=0, where=s1 < 0, facecolor='red', alpha=0.5)
ax.add_collection(collection)


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.35 api example code: two_scales.py

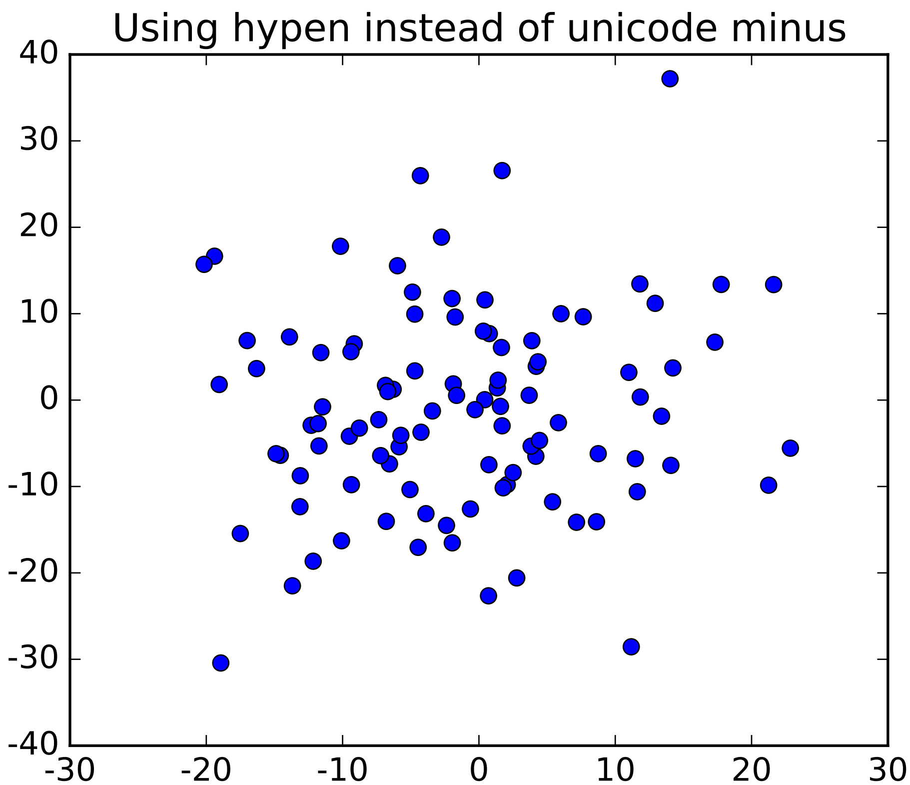



```
#!/usr/bin/env python
"""

Demonstrate how to do two plots on the same axes with different left
right scales.


The trick is to use *2 different axes*.  Turn the axes rectangular
```

```
frame off on the 2nd axes to keep it from obscuring the first.
Manually set the tick locs and labels as desired.  You can use
separate matplotlib.ticker formatters and locators as desired since
the two axes are independent.

This is achieved in the following example by calling the Axes.twinx()
method, which performs this work. See the source of twinx() in
axes.py for an example of how to do it for different x scales. (Hint:
use the xaxis instance and call tick_bottom and tick_top in place of
tick_left and tick_right.)

The twinx and twiny methods are also exposed as pyplot functions.

"""

import numpy as np
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots()
t = np.arange(0.01, 10.0, 0.01)
s1 = np.exp(t)
ax1.plot(t, s1, 'b-')
ax1.set_xlabel('time (s)')
# Make the y-axis label and tick labels match the line color.
ax1.set_ylabel('exp', color='b')
for tl in ax1.get_yticklabels():
    tl.set_color('b')


ax2 = ax1.twinx()
s2 = np.sin(2*np.pi*t)
ax2.plot(t, s2, 'r.')
ax2.set_ylabel('sin', color='r')
for tl in ax2.get_yticklabels():
    tl.set_color('r')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

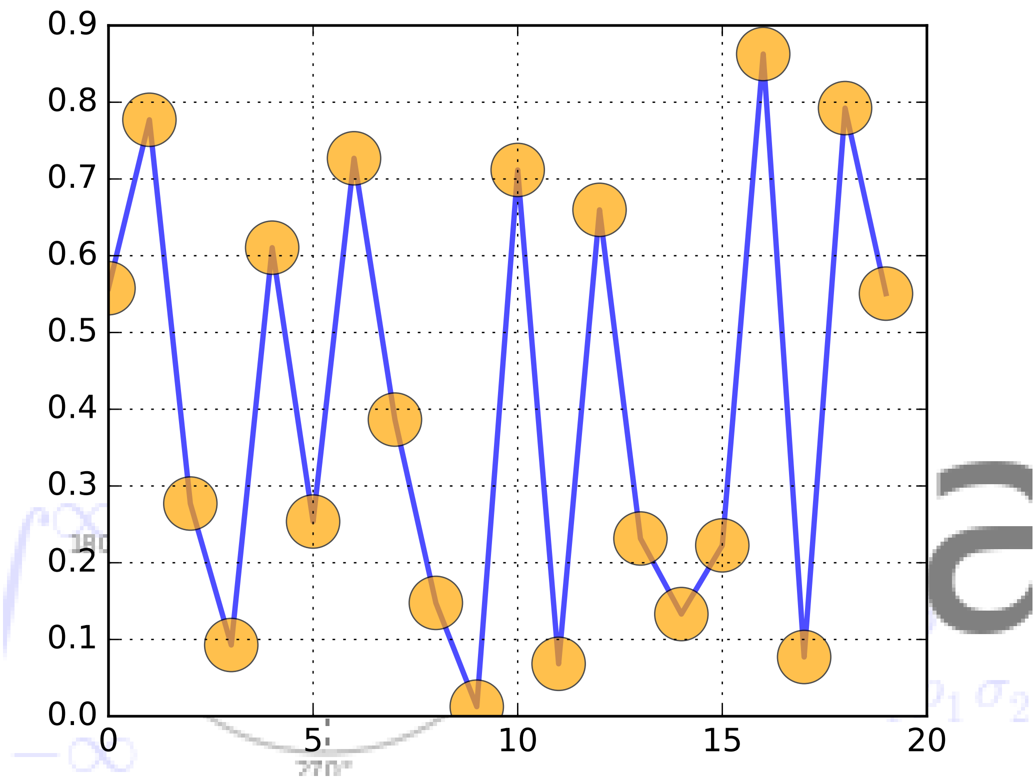
81.36 api example code: unicode_minus.py




```
"""
You can use the proper typesetting unicode minus (see
http://en.wikipedia.org/wiki/Plus_sign#Plus_sign) or the ASCII hypen
for minus, which some people prefer.  The matplotlibrc param
axes.unicode_minus controls the default behavior.

The default is to use the unicode minus
"""
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

matplotlib.rcParams['axes.unicode_minus'] = False
fig, ax = plt.subplots()
ax.plot(10*np.random.randn(100), 10*np.random.randn(100), 'o')
ax.set_title('Using hypen instead of unicode minus')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

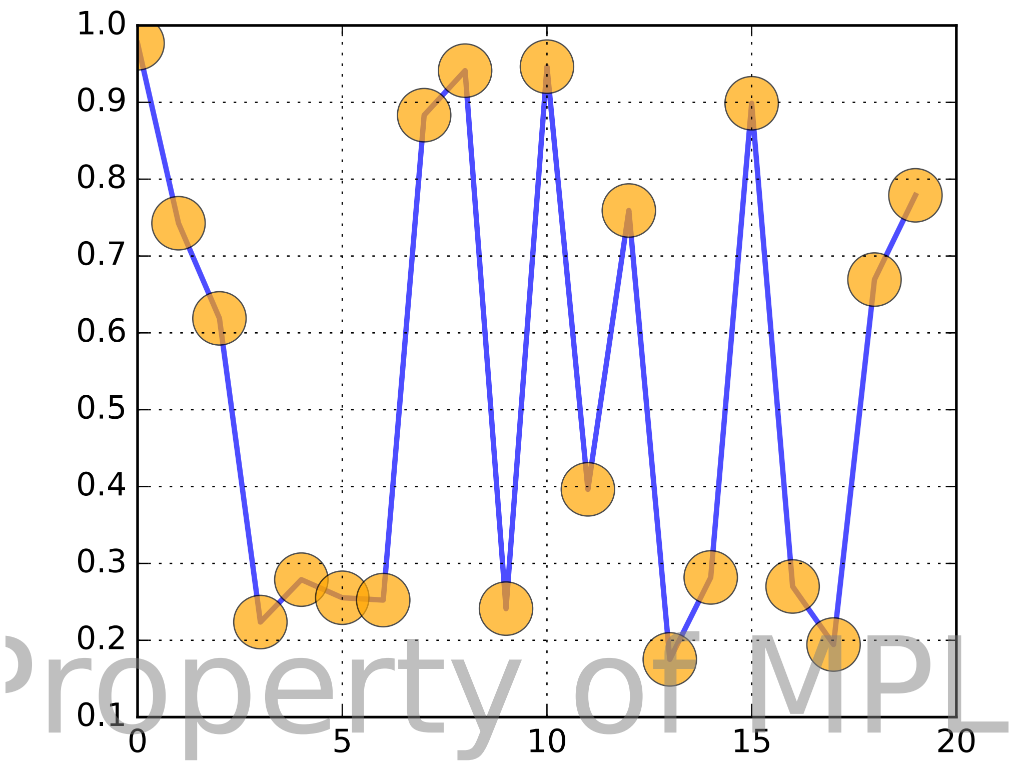
81.37 api example code: watermark_image.py




```
"""
Use a PNG file as a watermark
"""
from __future__ import print_function
import numpy as np
import matplotlib.cbook as cbook
import matplotlib.image as image
import matplotlib.pyplot as plt

datafile = cbook.get_sample_data('logo2.png', asfileobj=False)
print('loading %s' % datafile)
im = image.imread(datafile)
im[:, :, -1] = 0.5  # set the alpha channel

fig, ax = plt.subplots()

ax.plot(np.random.rand(20), '-o', ms=20, lw=2, alpha=0.7, mfc='orange')
ax.grid()
fig.figimage(im, 10, 10)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

81.38 api example code: watermark_text.py




```
"""
Use a Text as a watermark
"""
import numpy as np
#import matplotlib
#matplotlib.use('Agg')

import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(np.random.rand(20), '-o', ms=20, lw=2, alpha=0.7, mfc='orange')
ax.grid()

# position bottom right
fig.text(0.95, 0.05, 'Property of MPL',
         fontsize=50, color='gray',
         ha='right', va='bottom', alpha=0.5)

plt.show()
```

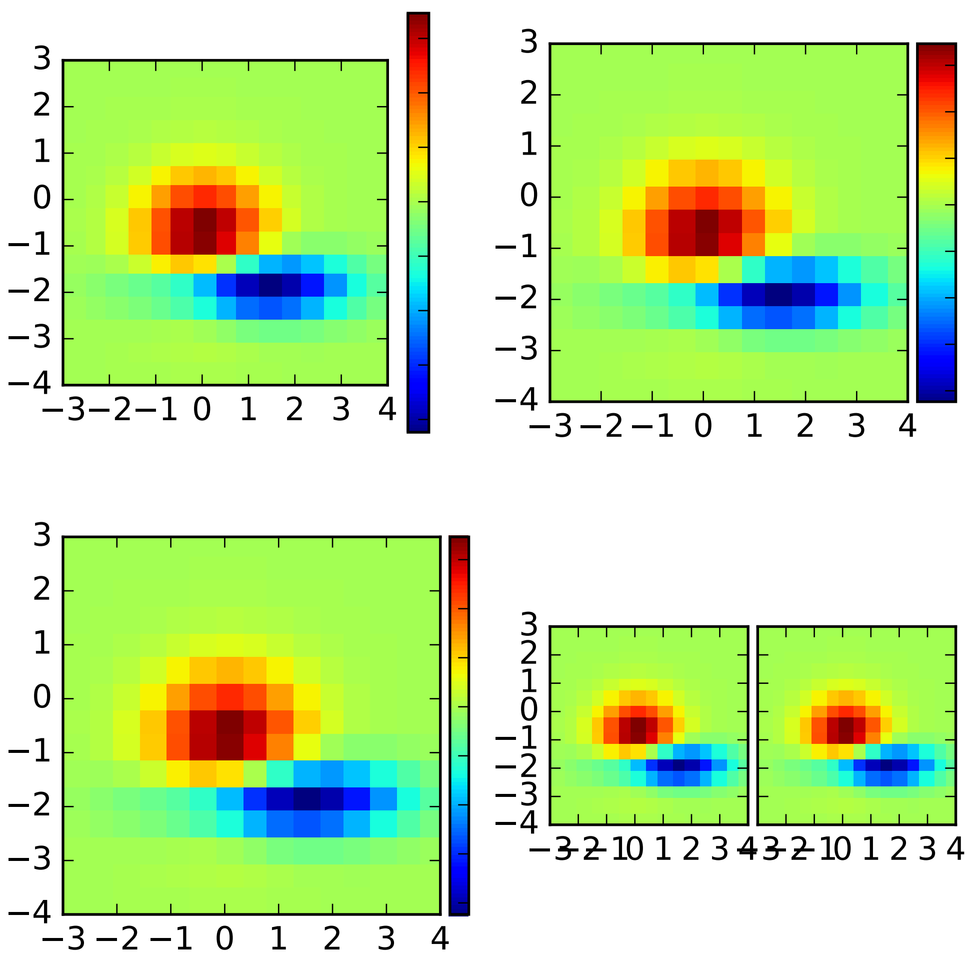
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

EIGHTYTWO

AXES_GRID EXAMPLES

82.1 axes_grid example code: demo_axes_divider.py



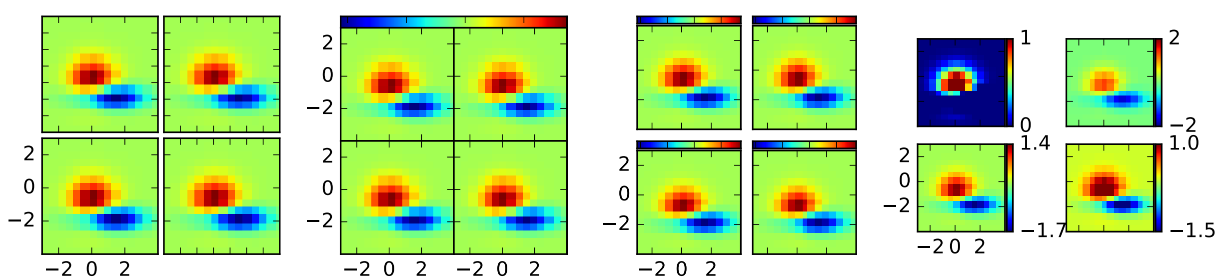
```
import matplotlib.pyplot as plt


def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)


def demo_simple_image(ax):
    Z, extent = get_demo_image()

    im = ax.imshow(Z, extent=extent, interpolation="nearest")
    cb = plt.colorbar(im)
    plt.setp(cb.ax.get_yticklabels(), visible=False)


def demo_locatable_axes_hard(fig1):

    from mpl_toolkits.axes_grid1 \
        import SubplotDivider, LocatableAxes, Size

    divider = SubplotDivider(fig1, 2, 2, 2, aspect=True)

    # axes for image
    ax = LocatableAxes(fig1, divider.get_position())

    # axes for colorbar
    ax_cb = LocatableAxes(fig1, divider.get_position())

    h = [Size.AxesX(ax),  # main axes
         Size.Fixed(0.05),  # padding, 0.1 inch
         Size.Fixed(0.2),  # colorbar, 0.3 inch
         ]

    v = [Size.AxesY(ax)]

    divider.set_horizontal(h)
    divider.set_vertical(v)

    ax.set_axes_locator(divider.new_locator(nx=0, ny=0))
    ax_cb.set_axes_locator(divider.new_locator(nx=2, ny=0))

    fig1.add_axes(ax)
    fig1.add_axes(ax_cb)

    ax_cb.axis["left"].toggle(all=False)
    ax_cb.axis["right"].toggle(ticks=True)

    Z, extent = get_demo_image()
```

```
    im = ax.imshow(Z, extent=extent, interpolation="nearest")
    plt.colorbar(im, cax=ax_cb)
    plt.setp(ax_cb.get_yticklabels(), visible=False)


def demo_locatable_axes_easy(ax):
    from mpl_toolkits.axes_grid1 import make_axes_locatable

    divider = make_axes_locatable(ax)

    ax_cb = divider.new_horizontal(size="5%", pad=0.05)
    fig1 = ax.get_figure()
    fig1.add_axes(ax_cb)

    Z, extent = get_demo_image()
    im = ax.imshow(Z, extent=extent, interpolation="nearest")

    plt.colorbar(im, cax=ax_cb)
    ax_cb.yaxis.tick_right()
    for tl in ax_cb.get_yticklabels():
        tl.set_visible(False)
    ax_cb.yaxis.tick_right()


def demo_images_side_by_side(ax):
    from mpl_toolkits.axes_grid1 import make_axes_locatable

    divider = make_axes_locatable(ax)

    Z, extent = get_demo_image()
    ax2 = divider.new_horizontal(size="100%", pad=0.05)
    fig1 = ax.get_figure()
    fig1.add_axes(ax2)

    ax.imshow(Z, extent=extent, interpolation="nearest")
    ax2.imshow(Z, extent=extent, interpolation="nearest")
    for tl in ax2.get_yticklabels():
        tl.set_visible(False)


def demo():

    fig1 = plt.figure(1, (6, 6))
    fig1.clf()

    # PLOT 1
    # simple image & colorbar
    ax = fig1.add_subplot(2, 2, 1)
    demo_simple_image(ax)

    # PLOT 2
    # image and colorbar whose location is adjusted in the drawing time.
```

```
    # a hard way

    demo_locatable_axes_hard(fig1)

    # PLOT 3
    # image and colorbar whose location is adjusted in the drawing time.
    # a easy way

    ax = fig1.add_subplot(2, 2, 3)
    demo_locatable_axes_easy(ax)

    # PLOT 4
    # two images side by side with fixed padding.

    ax = fig1.add_subplot(2, 2, 4)
    demo_images_side_by_side(ax)

    plt.draw()
    plt.show()


demo()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.2 axes_grid example code: demo_axes_grid.py




```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import AxesGrid


def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)


def demo_simple_grid(fig):
```

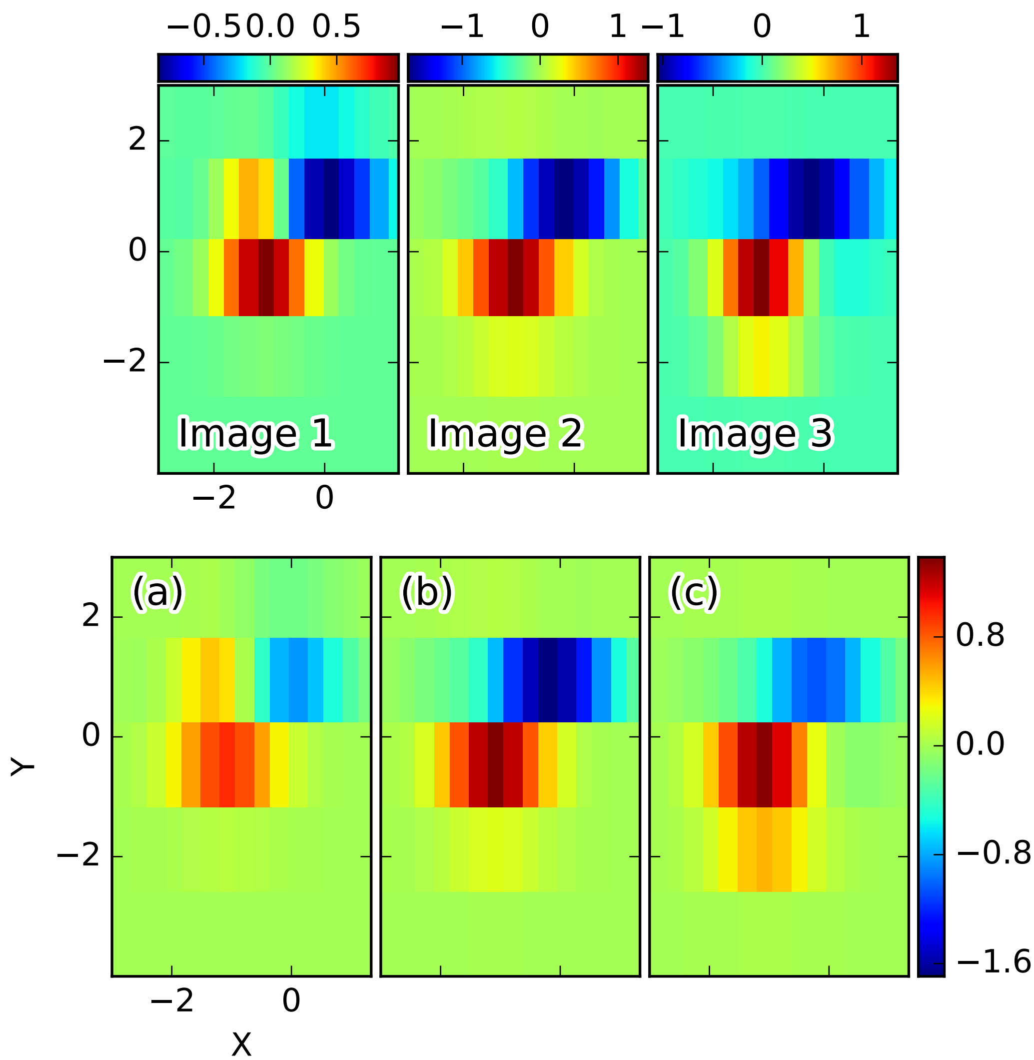
```
    """
    A grid of 2x2 images with 0.05 inch pad between images and only
    the lower-left axes is labeled.
    """
    grid = AxesGrid(fig, 141,  # similar to subplot(141)
                    nrows_ncols=(2, 2),
                    axes_pad=0.05,
                    label_mode="1",
                    )

    Z, extent = get_demo_image()
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest")

    # This only affects axes in first column and second row as share_all =
    # False.
    grid.axes_llc.set_xticks([-2, 0, 2])
    grid.axes_llc.set_yticks([-2, 0, 2])


def demo_grid_with_single_cbar(fig):
    """
    A grid of 2x2 images with a single colorbar
    """
    grid = AxesGrid(fig, 142,  # similar to subplot(142)
                    nrows_ncols=(2, 2),
                    axes_pad=0.0,
                    share_all=True,
                    label_mode="L",
                    cbar_location="top",
                    cbar_mode="single",
                    )

    Z, extent = get_demo_image()
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest")
    #plt.colorbar(im, cax = grid.cbar_axes[0])
    grid.cbar_axes[0].colorbar(im)

    for cax in grid.cbar_axes:
        cax.toggle_label(False)

    # This affects all axes as share_all = True.
    grid.axes_llc.set_xticks([-2, 0, 2])
    grid.axes_llc.set_yticks([-2, 0, 2])


def demo_grid_with_each_cbar(fig):
    """
    A grid of 2x2 images. Each image has its own colorbar.
    """

    grid = AxesGrid(fig, 143,  # similar to subplot(143)
```

```
                    nrows_ncols=(2, 2),
                    axes_pad=0.1,
                    label_mode="1",
                    share_all=True,
                    cbar_location="top",
                    cbar_mode="each",
                    cbar_size="7%",
                    cbar_pad="2%",
                    )
    Z, extent = get_demo_image()
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest")
        grid.cbar_axes[i].colorbar(im)

    for cax in grid.cbar_axes:
        cax.toggle_label(False)

    # This affects all axes because we set share_all = True.
    grid.axes_llc.set_xticks([-2, 0, 2])
    grid.axes_llc.set_yticks([-2, 0, 2])


def demo_grid_with_each_cbar_labelled(fig):
    """
    A grid of 2x2 images. Each image has its own colorbar.
    """

    grid = AxesGrid(fig, 144,  # similar to subplot(144)
                    nrows_ncols=(2, 2),
                    axes_pad=(0.45, 0.15),
                    label_mode="1",
                    share_all=True,
                    cbar_location="right",
                    cbar_mode="each",
                    cbar_size="7%",
                    cbar_pad="2%",
                    )
    Z, extent = get_demo_image()

    # Use a different colorbar range every time
    limits = ((0, 1), (-2, 2), (-1.7, 1.4), (-1.5, 1))
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest",
                            vmin=limits[i][0], vmax=limits[i][1])
        grid.cbar_axes[i].colorbar(im)

    for i, cax in enumerate(grid.cbar_axes):
        cax.set_yticks((limits[i][0], limits[i][1]))

    # This affects all axes because we set share_all = True.
    grid.axes_llc.set_xticks([-2, 0, 2])
    grid.axes_llc.set_yticks([-2, 0, 2])
```

```
if 1:
    F = plt.figure(1, (10.5, 2.5))

    F.subplots_adjust(left=0.05, right=0.95)

    demo_simple_grid(F)
    demo_grid_with_single_cbar(F)
    demo_grid_with_each_cbar(F)
    demo_grid_with_each_cbar_labelled(F)

    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.3 axes_grid example code: demo_axes_grid2.py



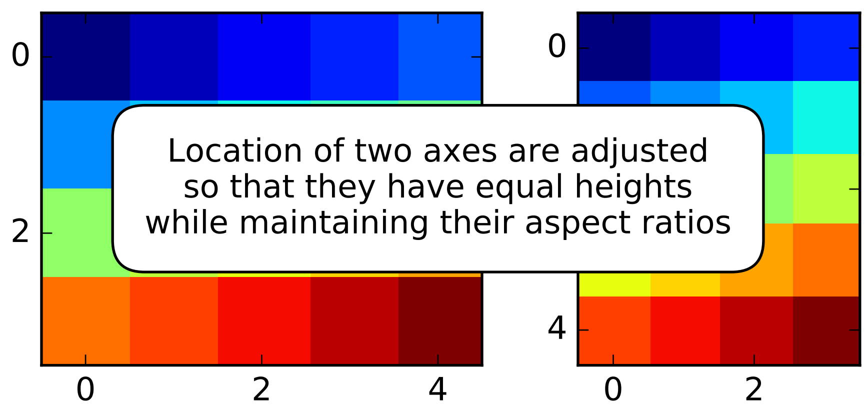

```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np


def get_demo_image():
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)
```

```python
def add_inner_title(ax, title, loc, size=None, **kwargs):
    from matplotlib.offsetbox import AnchoredText
    from matplotlib.patheffects import withStroke
    if size is None:
        size = dict(size=plt.rcParams['legend.fontsize'])
    at = AnchoredText(title, loc=loc, prop=size,
                      pad=0., borderpad=0.5,
                      frameon=False, **kwargs)
    ax.add_artist(at)
    at.txt._text.set_path_effects([withStroke(foreground="w", linewidth=3)])
    return at

if 1:
    F = plt.figure(1, (6, 6))
    F.clf()

    # prepare images
    Z, extent = get_demo_image()
    ZS = [Z[i::3, :] for i in range(3)]
    extent = extent[0], extent[1]/3., extent[2], extent[3]

    # demo 1 : colorbar at each axes

    grid = ImageGrid(F, 211,  # similar to subplot(111)
                     nrows_ncols=(1, 3),
                     direction="row",
                     axes_pad=0.05,
                     add_all=True,
                     label_mode="1",
                     share_all=True,
                     cbar_location="top",
                     cbar_mode="each",
                     cbar_size="7%",
                     cbar_pad="1%",
                     )

    for ax, z in zip(grid, ZS):
        im = ax.imshow(
            z, origin="lower", extent=extent, interpolation="nearest")
        ax.cax.colorbar(im)

    for ax, im_title in zip(grid, ["Image 1", "Image 2", "Image 3"]):
        t = add_inner_title(ax, im_title, loc=3)
        t.patch.set_alpha(0.5)

    for ax, z in zip(grid, ZS):
        ax.cax.toggle_label(True)
        #axis = ax.cax.axis[ax.cax.orientation]
        #axis.label.set_text("counts s$^{-1}$")
        #axis.label.set_size(10)
        #axis.major_ticklabels.set_size(6)

    # changing the colorbar ticks
```

```
    grid[1].cax.set_xticks([-1, 0, 1])
    grid[2].cax.set_xticks([-1, 0, 1])

    grid[0].set_xticks([-2, 0])
    grid[0].set_yticks([-2, 0, 2])

    # demo 2 : shared colorbar

    grid2 = ImageGrid(F, 212,
                      nrows_ncols=(1, 3),
                      direction="row",
                      axes_pad=0.05,
                      add_all=True,
                      label_mode="1",
                      share_all=True,
                      cbar_location="right",
                      cbar_mode="single",
                      cbar_size="10%",
                      cbar_pad=0.05,
                      )

    grid2[0].set_xlabel("X")
    grid2[0].set_ylabel("Y")

    vmax, vmin = np.max(ZS), np.min(ZS)
    import matplotlib.colors
    norm = matplotlib.colors.Normalize(vmax=vmax, vmin=vmin)

    for ax, z in zip(grid2, ZS):
        im = ax.imshow(z, norm=norm,
                       origin="lower", extent=extent,
                       interpolation="nearest")

    # With cbar_mode="single", cax attribute of all axes are identical.
    ax.cax.colorbar(im)
    ax.cax.toggle_label(True)

    for ax, im_title in zip(grid2, ["(a)", "(b)", "(c)"]):
        t = add_inner_title(ax, im_title, loc=2)
        t.patch.set_ec("none")
        t.patch.set_alpha(0.5)

    grid2[0].set_xticks([-2, 0])
    grid2[0].set_yticks([-2, 0, 2])

    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.4 axes_grid example code: demo_axes_hbox_divider.py

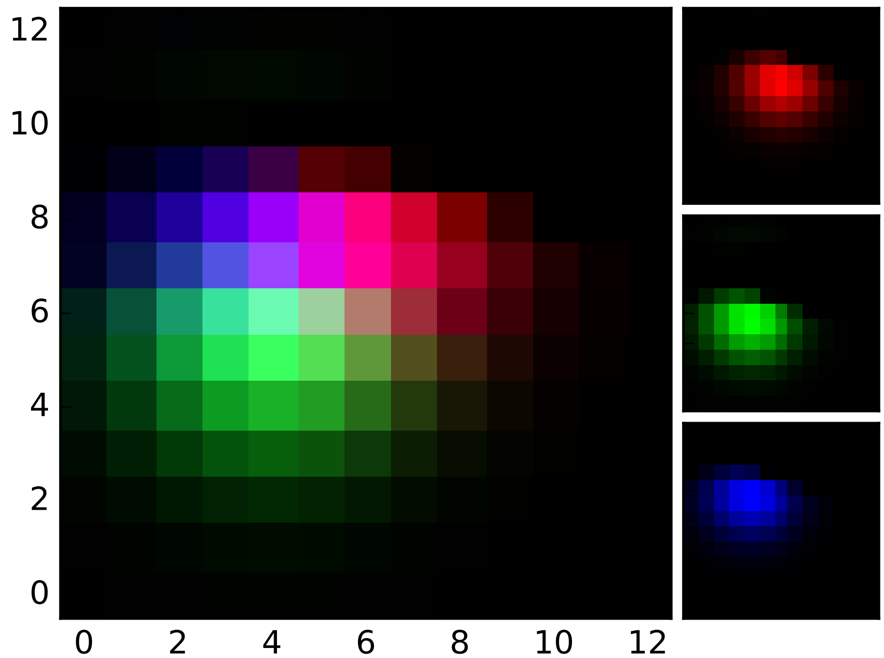


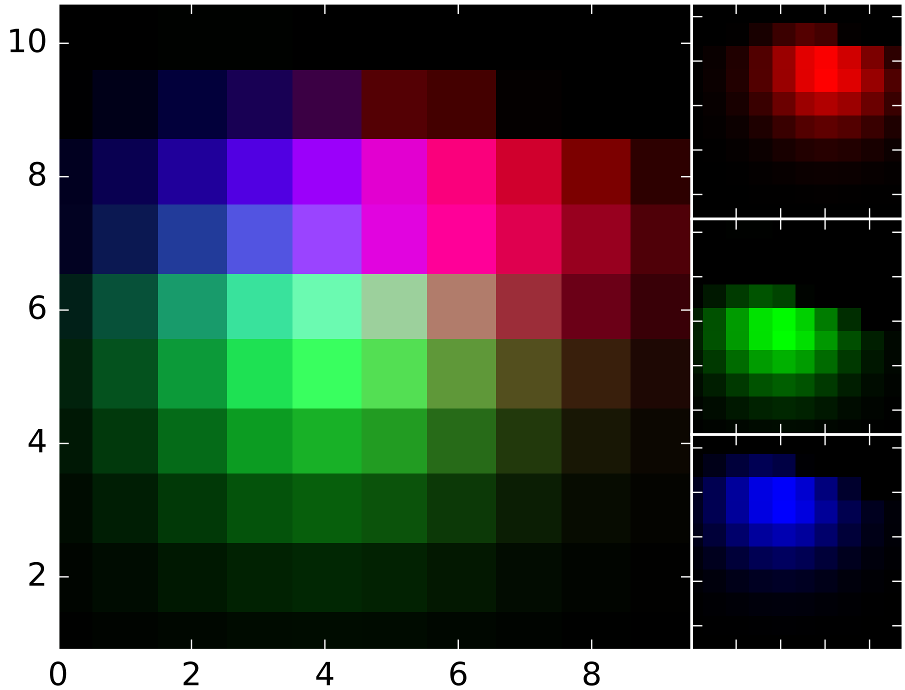

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.axes_divider import HBoxDivider
import mpl_toolkits.axes_grid1.axes_size as Size


def make_heights_equal(fig, rect, ax1, ax2, pad):
    # pad in inches

    h1, v1 = Size.AxesX(ax1), Size.AxesY(ax1)
    h2, v2 = Size.AxesX(ax2), Size.AxesY(ax2)

    pad_v = Size.Scaled(1)
    pad_h = Size.Fixed(pad)

    my_divider = HBoxDivider(fig, rect,
                             horizontal=[h1, pad_h, h2],
                             vertical=[v1, pad_v, v2])

    ax1.set_axes_locator(my_divider.new_locator(0))
    ax2.set_axes_locator(my_divider.new_locator(2))
```

```
if __name__ == "__main__":

    arr1 = np.arange(20).reshape((4, 5))
    arr2 = np.arange(20).reshape((5, 4))

    fig, (ax1, ax2) = plt.subplots(1, 2)
    ax1.imshow(arr1, interpolation="nearest")
    ax2.imshow(arr2, interpolation="nearest")

    rect = 111  # subplot param for combined axes
    make_heights_equal(fig, rect, ax1, ax2, pad=0.5)  # pad in inches

    for ax in [ax1, ax2]:
        ax.locator_params(nbins=4)

    # annotate
    ax3 = plt.axes([0.5, 0.5, 0.001, 0.001], frameon=False)
    ax3.xaxis.set_visible(False)
    ax3.yaxis.set_visible(False)
    ax3.annotate("Location of two axes are adjusted\n"
                 "so that they have equal heights\n"
                 "while maintaining their aspect ratios", (0.5, 0.5),
                 xycoords="axes fraction", va="center", ha="center",
                 bbox=dict(boxstyle="round, pad=1", fc="w"))

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.5 axes_grid example code: demo_axes_rgb.py





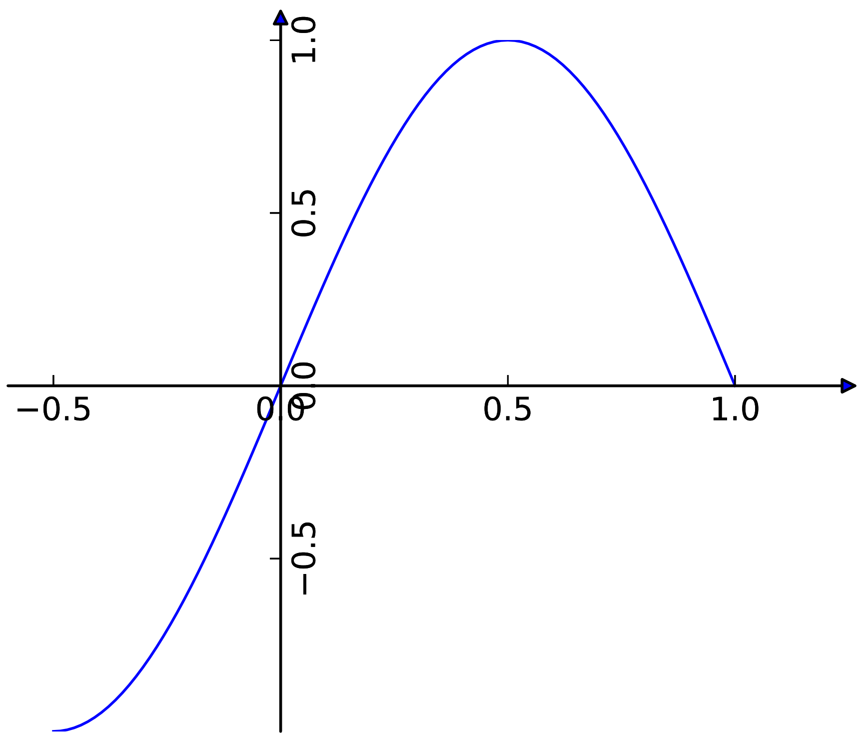

```
import numpy as np
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.axes_rgb import make_rgb_axes, RGBAxes


def get_demo_image():
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)


def get_rgb():
    Z, extent = get_demo_image()

    Z[Z < 0] = 0.
    Z = Z/Z.max()

    R = Z[:13, :13]
    G = Z[2:, 2:]
    B = Z[:13, 2:]

    return R, G, B
```

```
def make_cube(r, g, b):
    ny, nx = r.shape
    R = np.zeros([ny, nx, 3], dtype="d")
    R[:, :, 0] = r
    G = np.zeros_like(R)
    G[:, :, 1] = g
    B = np.zeros_like(R)
    B[:, :, 2] = b

    RGB = R + G + B

    return R, G, B, RGB


def demo_rgb():
    fig, ax = plt.subplots()
    ax_r, ax_g, ax_b = make_rgb_axes(ax, pad=0.02)
    #fig.add_axes(ax_r)
    #fig.add_axes(ax_g)
    #fig.add_axes(ax_b)

    r, g, b = get_rgb()
    im_r, im_g, im_b, im_rgb = make_cube(r, g, b)
    kwargs = dict(origin="lower", interpolation="nearest")
    ax.imshow(im_rgb, **kwargs)
    ax_r.imshow(im_r, **kwargs)
    ax_g.imshow(im_g, **kwargs)
    ax_b.imshow(im_b, **kwargs)


def demo_rgb2():
    fig = plt.figure(2)
    ax = RGBAxes(fig, [0.1, 0.1, 0.8, 0.8], pad=0.0)
    #fig.add_axes(ax)
    #ax.add_RGB_to_figure()

    r, g, b = get_rgb()
    kwargs = dict(origin="lower", interpolation="nearest")
    ax.imshow_rgb(r, g, b, **kwargs)

    ax.RGB.set_xlim(0., 9.5)
    ax.RGB.set_ylim(0.9, 10.6)

    for ax1 in [ax.RGB, ax.R, ax.G, ax.B]:
        for sp1 in ax1.spines.values():
            sp1.set_color("w")
        for tick in ax1.xaxis.get_major_ticks() + ax1.yaxis.get_major_ticks():
            tick.tick1line.set_mec("w")
            tick.tick2line.set_mec("w")

    return ax
```

```
demo_rgb()
ax = demo_rgb2()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.6 axes_grid example code: demo_axisline_style.py

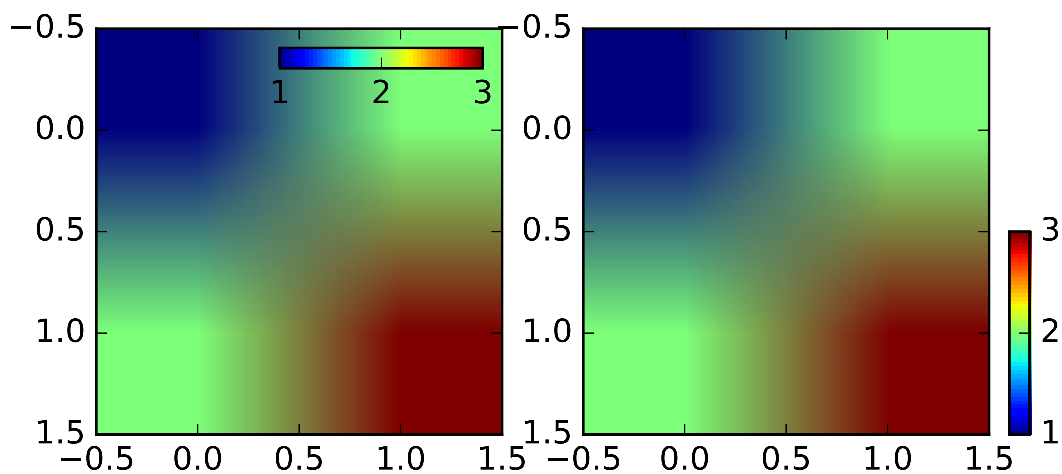



```
from mpl_toolkits.axes_grid.axislines import SubplotZero
import matplotlib.pyplot as plt
import numpy as np

if 1:
    fig = plt.figure(1)
    ax = SubplotZero(fig, 111)
    fig.add_subplot(ax)

    for direction in ["xzero", "yzero"]:
        ax.axis[direction].set_axisline_style("-|>")
        ax.axis[direction].set_visible(True)
```

```
    for direction in ["left", "right", "bottom", "top"]:
        ax.axis[direction].set_visible(False)

    x = np.linspace(-0.5, 1., 100)
    ax.plot(x, np.sin(x*np.pi))

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.7 axes_grid example code: demo_colorbar_with_inset_locator.py

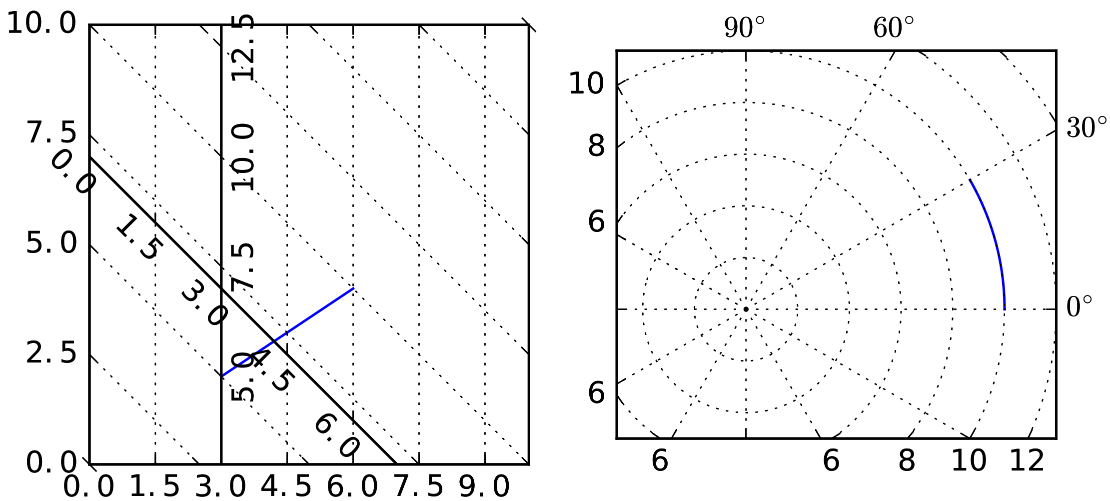



```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.inset_locator import inset_axes

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=[6, 3])

axins1 = inset_axes(ax1,
                    width="50%",  # width = 10% of parent_bbox width
                    height="5%",  # height : 50%
                    loc=1)

im1 = ax1.imshow([[1, 2], [2, 3]])
plt.colorbar(im1, cax=axins1, orientation="horizontal", ticks=[1, 2, 3])
axins1.xaxis.set_ticks_position("bottom")

axins = inset_axes(ax2,
                   width="5%",  # width = 10% of parent_bbox width
                   height="50%",  # height : 50%
```

```
                   loc=3,
                   bbox_to_anchor=(1.05, 0., 1, 1),
                   bbox_transform=ax2.transAxes,
                   borderpad=0,
                   )

# Controlling the placement of the inset axes is basically same as that
# of the legend.  you may want to play with the borderpad value and
# the bbox_to_anchor coordinate.

im = ax2.imshow([[1, 2], [2, 3]])
plt.colorbar(im, cax=axins, ticks=[1, 2, 3])

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.8 axes_grid example code: demo_curvelinear_grid.py




```
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

from mpl_toolkits.axisartist import Subplot
from mpl_toolkits.axisartist import SubplotHost, \
    ParasiteAxesAuxTrans
```

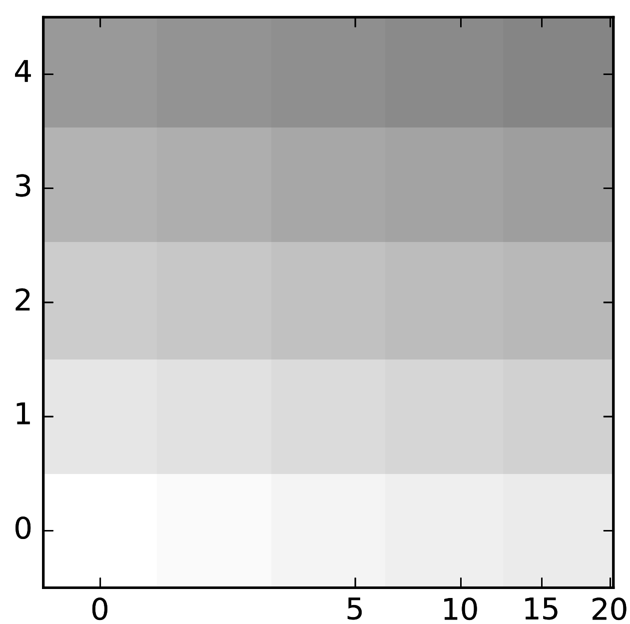
```
from mpl_toolkits.axisartist.grid_helper_curvelinear import \
    GridHelperCurveLinear


def curvelinear_test1(fig):
    """
    grid for custom transform.
    """

    def tr(x, y):
        x, y = np.asarray(x), np.asarray(y)
        return x, y - x

    def inv_tr(x, y):
        x, y = np.asarray(x), np.asarray(y)
        return x, y + x

    grid_helper = GridHelperCurveLinear((tr, inv_tr))

    ax1 = Subplot(fig, 1, 2, 1, grid_helper=grid_helper)
    # ax1 will have a ticks and gridlines defined by the given
    # transform (+ transData of the Axes). Note that the transform of
    # the Axes itself (i.e., transData) is not affected by the given
    # transform.

    fig.add_subplot(ax1)

    xx, yy = tr([3, 6], [5.0, 10.])
    ax1.plot(xx, yy)

    ax1.set_aspect(1.)
    ax1.set_xlim(0, 10.)
    ax1.set_ylim(0, 10.)

    ax1.axis["t"] = ax1.new_floating_axis(0, 3.)
    ax1.axis["t2"] = ax1.new_floating_axis(1, 7.)
    ax1.grid(True)


import mpl_toolkits.axisartist.angle_helper as angle_helper
from matplotlib.projections import PolarAxes
from matplotlib.transforms import Affine2D


def curvelinear_test2(fig):
    """
    polar projection, but in a rectangular box.
    """

    # PolarAxes.PolarTransform takes radian. However, we want our coordinate
    # system in degree
    tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()
```

```
    # polar projection, which involves cycle, and also has limits in
    # its coordinates, needs a special method to find the extremes
    # (min, max of the coordinate within the view).

    # 20, 20 : number of sampling points along x, y direction
    extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
                                                     lon_cycle=360,
                                                     lat_cycle=None,
                                                     lon_minmax=None,
                                                     lat_minmax=(0, np.inf),
                                                     )

    grid_locator1 = angle_helper.LocatorDMS(12)
    # Find a grid values appropriate for the coordinate (degree,
    # minute, second).

    tick_formatter1 = angle_helper.FormatterDMS()
    # And also uses an appropriate formatter.  Note that,the
    # acceptable Locator and Formatter class is a bit different than
    # that of mpl's, and you cannot directly use mpl's Locator and
    # Formatter here (but may be possible in the future).

    grid_helper = GridHelperCurveLinear(tr,
                                        extreme_finder=extreme_finder,
                                        grid_locator1=grid_locator1,
                                        tick_formatter1=tick_formatter1
                                        )

    ax1 = SubplotHost(fig, 1, 2, 2, grid_helper=grid_helper)

    # make ticklabels of right and top axis visible.
    ax1.axis["right"].major_ticklabels.set_visible(True)
    ax1.axis["top"].major_ticklabels.set_visible(True)

    # let right axis shows ticklabels for 1st coordinate (angle)
    ax1.axis["right"].get_helper().nth_coord_ticks = 0
    # let bottom axis shows ticklabels for 2nd coordinate (radius)
    ax1.axis["bottom"].get_helper().nth_coord_ticks = 1

    fig.add_subplot(ax1)

    # A parasite axes with given transform
    ax2 = ParasiteAxesAuxTrans(ax1, tr, "equal")
    # note that ax2.transData == tr + ax1.transData
    # Anthing you draw in ax2 will match the ticks and grids of ax1.
    ax1.parasites.append(ax2)
    intp = cbook.simple_linear_interpolation
    ax2.plot(intp(np.array([0, 30]), 50),
             intp(np.array([10., 10.]), 50))

    ax1.set_aspect(1.)
    ax1.set_xlim(-5, 12)
    ax1.set_ylim(-5, 10)
```

```
    ax1.grid(True)

if 1:
    fig = plt.figure(1, figsize=(7, 4))
    fig.clf()

    curvelinear_test1(fig)
    curvelinear_test2(fig)

    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.9 axes_grid example code: demo_curvelinear_grid2.py

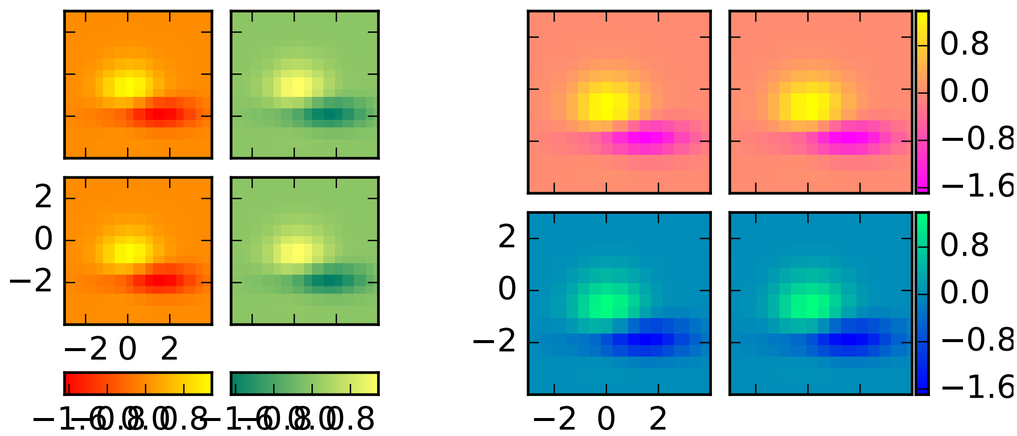



```
import numpy as np
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid.grid_helper_curvelinear import \
    GridHelperCurveLinear
from mpl_toolkits.axes_grid.axislines import Subplot

import mpl_toolkits.axes_grid.angle_helper as angle_helper


def curvelinear_test1(fig):
```

```
    """
    grid for custom transform.
    """

    def tr(x, y):
        sgn = np.sign(x)
        x, y = np.abs(np.asarray(x)), np.asarray(y)
        return sgn*x**.5, y

    def inv_tr(x, y):
        sgn = np.sign(x)
        x, y = np.asarray(x), np.asarray(y)
        return sgn*x**2, y

    extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
                                                     lon_cycle=None,
                                                     lat_cycle=None,
                                                     # (0, np.inf),
                                                     lon_minmax=None,
                                                     lat_minmax=None,
                                                     )

    grid_helper = GridHelperCurveLinear((tr, inv_tr),
                                        extreme_finder=extreme_finder)

    ax1 = Subplot(fig, 111, grid_helper=grid_helper)
    # ax1 will have a ticks and gridlines defined by the given
    # transform (+ transData of the Axes). Note that the transform of
    # the Axes itself (i.e., transData) is not affected by the given
    # transform.

    fig.add_subplot(ax1)

    ax1.imshow(np.arange(25).reshape(5, 5),
               vmax=50, cmap=plt.cm.gray_r,
               interpolation="nearest",
               origin="lower")

    # tick density
    grid_helper.grid_finder.grid_locator1._nbins = 6
    grid_helper.grid_finder.grid_locator2._nbins = 6


if 1:
    fig = plt.figure(1, figsize=(7, 4))
    fig.clf()

    curvelinear_test1(fig)
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.10 axes_grid example code: demo_edge_colorbar.py

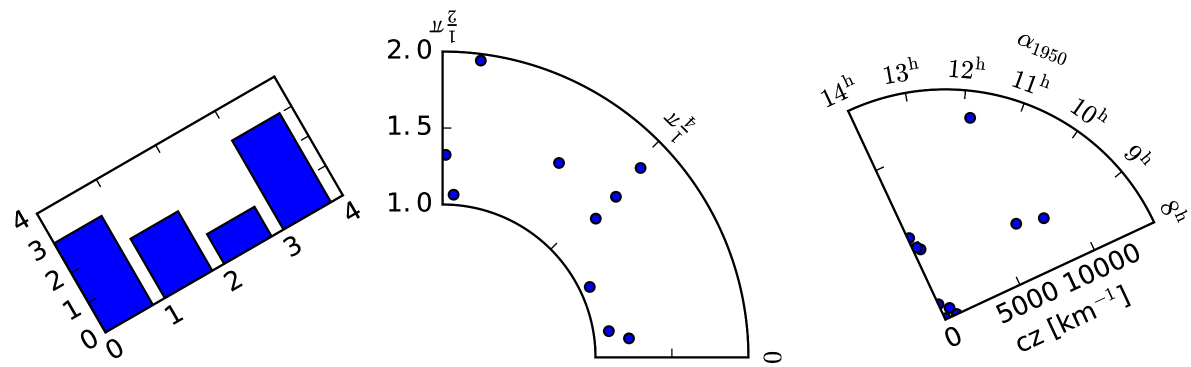



```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import AxesGrid


def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)


def demo_bottom_cbar(fig):
    """
    A grid of 2x2 images with a colorbar for each column.
    """
    grid = AxesGrid(fig, 121,  # similar to subplot(132)
                    nrows_ncols=(2, 2),
                    axes_pad=0.10,
                    share_all=True,
                    label_mode="1",
                    cbar_location="bottom",
                    cbar_mode="edge",
                    cbar_pad=0.25,
                    cbar_size="15%",
                    direction="column"
                    )

    Z, extent = get_demo_image()
    cmaps = [plt.get_cmap("autumn"), plt.get_cmap("summer")]
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest",
                            cmap=cmaps[i//2])
        if i % 2:
```

```
            cbar = grid.cbar_axes[i//2].colorbar(im)

    for cax in grid.cbar_axes:
        cax.toggle_label(True)
        cax.axis[cax.orientation].set_label("Bar")

    # This affects all axes as share_all = True.
    grid.axes_llc.set_xticks([-2, 0, 2])
    grid.axes_llc.set_yticks([-2, 0, 2])


def demo_right_cbar(fig):
    """
    A grid of 2x2 images. Each row has its own colorbar.
    """

    grid = AxesGrid(F, 122,  # similar to subplot(122)
                    nrows_ncols=(2, 2),
                    axes_pad=0.10,
                    label_mode="1",
                    share_all=True,
                    cbar_location="right",
                    cbar_mode="edge",
                    cbar_size="7%",
                    cbar_pad="2%",
                    )
    Z, extent = get_demo_image()
    cmaps = [plt.get_cmap("spring"), plt.get_cmap("winter")]
    for i in range(4):
        im = grid[i].imshow(Z, extent=extent, interpolation="nearest",
                            cmap=cmaps[i//2])
        if i % 2:
            grid.cbar_axes[i//2].colorbar(im)

    for cax in grid.cbar_axes:
        cax.toggle_label(True)
        cax.axis[cax.orientation].set_label('Foo')

    # This affects all axes because we set share_all = True.
    grid.axes_llc.set_xticks([-2, 0, 2])
    grid.axes_llc.set_yticks([-2, 0, 2])


if 1:
    F = plt.figure(1, (5.5, 2.5))

    F.subplots_adjust(left=0.05, right=0.93)

    demo_bottom_cbar(F)
    demo_right_cbar(F)

    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.11 axes_grid example code: demo_floating_axes.py




```
from matplotlib.transforms import Affine2D
import mpl_toolkits.axisartist.floating_axes as floating_axes
import numpy as np
import mpl_toolkits.axisartist.angle_helper as angle_helper
from matplotlib.projections import PolarAxes
from mpl_toolkits.axisartist.grid_finder import (FixedLocator, MaxNLocator,
                                                 DictFormatter)
import matplotlib.pyplot as plt


def setup_axes1(fig, rect):
    """
    A simple one.
    """
    tr = Affine2D().scale(2, 1).rotate_deg(30)

    grid_helper = floating_axes.GridHelperCurveLinear(
        tr, extremes=(0, 4, 0, 4))

    ax1 = floating_axes.FloatingSubplot(fig, rect, grid_helper=grid_helper)
    fig.add_subplot(ax1)

    aux_ax = ax1.get_aux_axes(tr)

    grid_helper.grid_finder.grid_locator1._nbins = 4
    grid_helper.grid_finder.grid_locator2._nbins = 4
```

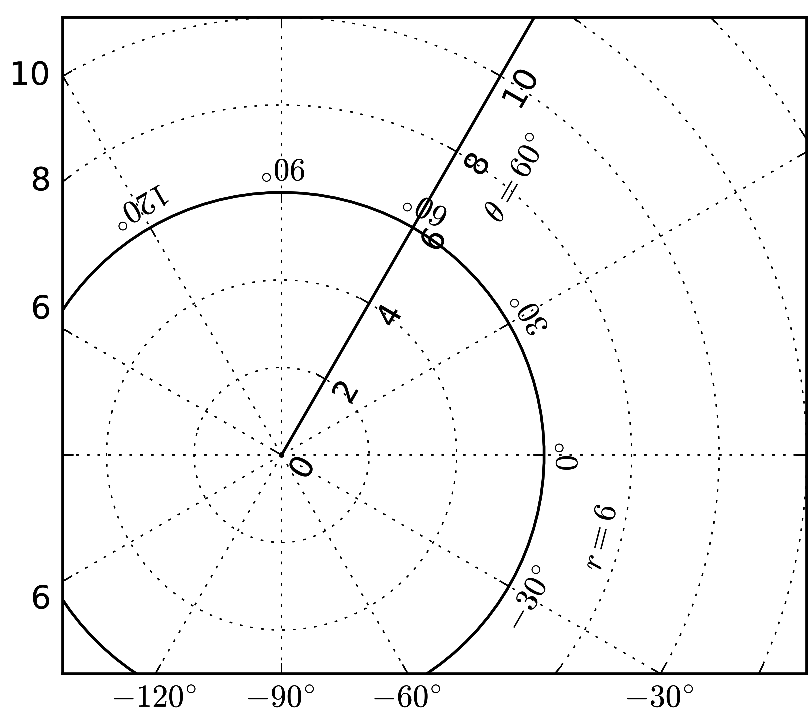
```
    return ax1, aux_ax


def setup_axes2(fig, rect):
    """
    With custom locator and formatter.
    Note that the extreme values are swapped.
    """
    tr = PolarAxes.PolarTransform()

    pi = np.pi
    angle_ticks = [(0, r"$0$"),
                   (.25*pi, r"$\frac{1}{4}\pi$"),
                   (.5*pi, r"$\frac{1}{2}\pi$")]
    grid_locator1 = FixedLocator([v for v, s in angle_ticks])
    tick_formatter1 = DictFormatter(dict(angle_ticks))

    grid_locator2 = MaxNLocator(2)

    grid_helper = floating_axes.GridHelperCurveLinear(
        tr, extremes=(.5*pi, 0, 2, 1),
        grid_locator1=grid_locator1,
        grid_locator2=grid_locator2,
        tick_formatter1=tick_formatter1,
        tick_formatter2=None)

    ax1 = floating_axes.FloatingSubplot(fig, rect, grid_helper=grid_helper)
    fig.add_subplot(ax1)

    # create a parasite axes whose transData in RA, cz
    aux_ax = ax1.get_aux_axes(tr)

    aux_ax.patch = ax1.patch  # for aux_ax to have a clip path as in ax
    ax1.patch.zorder = 0.9  # but this has a side effect that the patch is
    # drawn twice, and possibly over some other
    # artists. So, we decrease the zorder a bit to
    # prevent this.

    return ax1, aux_ax


def setup_axes3(fig, rect):
    """
    Sometimes, things like axis_direction need to be adjusted.
    """

    # rotate a bit for better orientation
    tr_rotate = Affine2D().translate(-95, 0)

    # scale degree to radians
    tr_scale = Affine2D().scale(np.pi/180., 1.)

    tr = tr_rotate + tr_scale + PolarAxes.PolarTransform()
```

```
    grid_locator1 = angle_helper.LocatorHMS(4)
    tick_formatter1 = angle_helper.FormatterHMS()

    grid_locator2 = MaxNLocator(3)

    ra0, ra1 = 8.*15, 14.*15
    cz0, cz1 = 0, 14000
    grid_helper = floating_axes.GridHelperCurveLinear(
        tr, extremes=(ra0, ra1, cz0, cz1),
        grid_locator1=grid_locator1,
        grid_locator2=grid_locator2,
        tick_formatter1=tick_formatter1,
        tick_formatter2=None)

    ax1 = floating_axes.FloatingSubplot(fig, rect, grid_helper=grid_helper)
    fig.add_subplot(ax1)

    # adjust axis
    ax1.axis["left"].set_axis_direction("bottom")
    ax1.axis["right"].set_axis_direction("top")

    ax1.axis["bottom"].set_visible(False)
    ax1.axis["top"].set_axis_direction("bottom")
    ax1.axis["top"].toggle(ticklabels=True, label=True)
    ax1.axis["top"].major_ticklabels.set_axis_direction("top")
    ax1.axis["top"].label.set_axis_direction("top")

    ax1.axis["left"].label.set_text(r"cz [km$^{-1}$]")
    ax1.axis["top"].label.set_text(r"$\alpha_{1950}$")

    # create a parasite axes whose transData in RA, cz
    aux_ax = ax1.get_aux_axes(tr)

    aux_ax.patch = ax1.patch  # for aux_ax to have a clip path as in ax
    ax1.patch.zorder = 0.9  # but this has a side effect that the patch is
    # drawn twice, and possibly over some other
    # artists. So, we decrease the zorder a bit to
    # prevent this.

    return ax1, aux_ax


##########################################################
fig = plt.figure(1, figsize=(8, 4))
fig.subplots_adjust(wspace=0.3, left=0.05, right=0.95)

ax1, aux_ax1 = setup_axes1(fig, 131)
aux_ax1.bar([0, 1, 2, 3], [3, 2, 1, 3])

ax2, aux_ax2 = setup_axes2(fig, 132)
theta = np.random.rand(10)*.5*np.pi
radius = np.random.rand(10) + 1.
```

```
aux_ax2.scatter(theta, radius)

ax3, aux_ax3 = setup_axes3(fig, 133)

theta = (8 + np.random.rand(10)*(14 - 8))*15.  # in degrees
radius = np.random.rand(10)*14000.
aux_ax3.scatter(theta, radius)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.12 axes_grid example code: demo_floating_axis.py



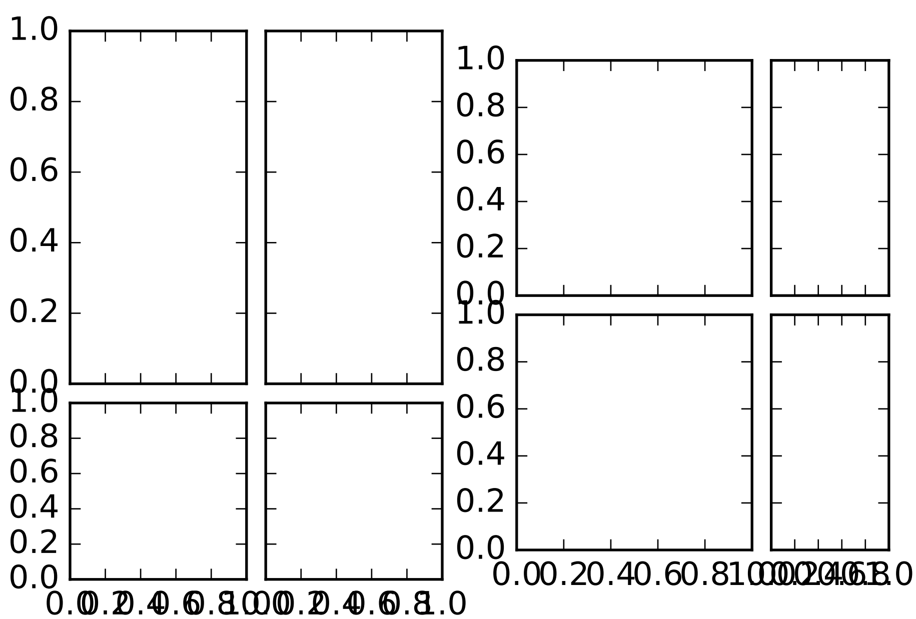

```
"""
An experimental support for curvilinear grid.
"""
```

```
def curvelinear_test2(fig):
    """
    polar projection, but in a rectangular box.
    """
    global ax1
    import numpy as np
    import mpl_toolkits.axisartist.angle_helper as angle_helper
    from matplotlib.projections import PolarAxes
    from matplotlib.transforms import Affine2D

    from mpl_toolkits.axisartist import SubplotHost

    from mpl_toolkits.axisartist import GridHelperCurveLinear

    # see demo_curvelinear_grid.py for details
    tr = Affine2D().scale(np.pi/180., 1.) + PolarAxes.PolarTransform()

    extreme_finder = angle_helper.ExtremeFinderCycle(20, 20,
                                                     lon_cycle=360,
                                                     lat_cycle=None,
                                                     lon_minmax=None,
                                                     lat_minmax=(0, np.inf),
                                                     )

    grid_locator1 = angle_helper.LocatorDMS(12)

    tick_formatter1 = angle_helper.FormatterDMS()

    grid_helper = GridHelperCurveLinear(tr,
                                        extreme_finder=extreme_finder,
                                        grid_locator1=grid_locator1,
                                        tick_formatter1=tick_formatter1
                                        )

    ax1 = SubplotHost(fig, 1, 1, 1, grid_helper=grid_helper)

    fig.add_subplot(ax1)

    # Now creates floating axis

    #grid_helper = ax1.get_grid_helper()
    # floating axis whose first coordinate (theta) is fixed at 60
    ax1.axis["lat"] = axis = ax1.new_floating_axis(0, 60)
    axis.label.set_text(r"$\theta = 60^{\circ}$")
    axis.label.set_visible(True)

    # floating axis whose second coordinate (r) is fixed at 6
    ax1.axis["lon"] = axis = ax1.new_floating_axis(1, 6)
    axis.label.set_text(r"$r = 6$")

    ax1.set_aspect(1.)
    ax1.set_xlim(-5, 12)
    ax1.set_ylim(-5, 10)
```

```
    ax1.grid(True)

import matplotlib.pyplot as plt
fig = plt.figure(1, figsize=(5, 5))
fig.clf()

curvelinear_test2(fig)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.13 axes_grid example code: demo_imagegrid_aspect.py

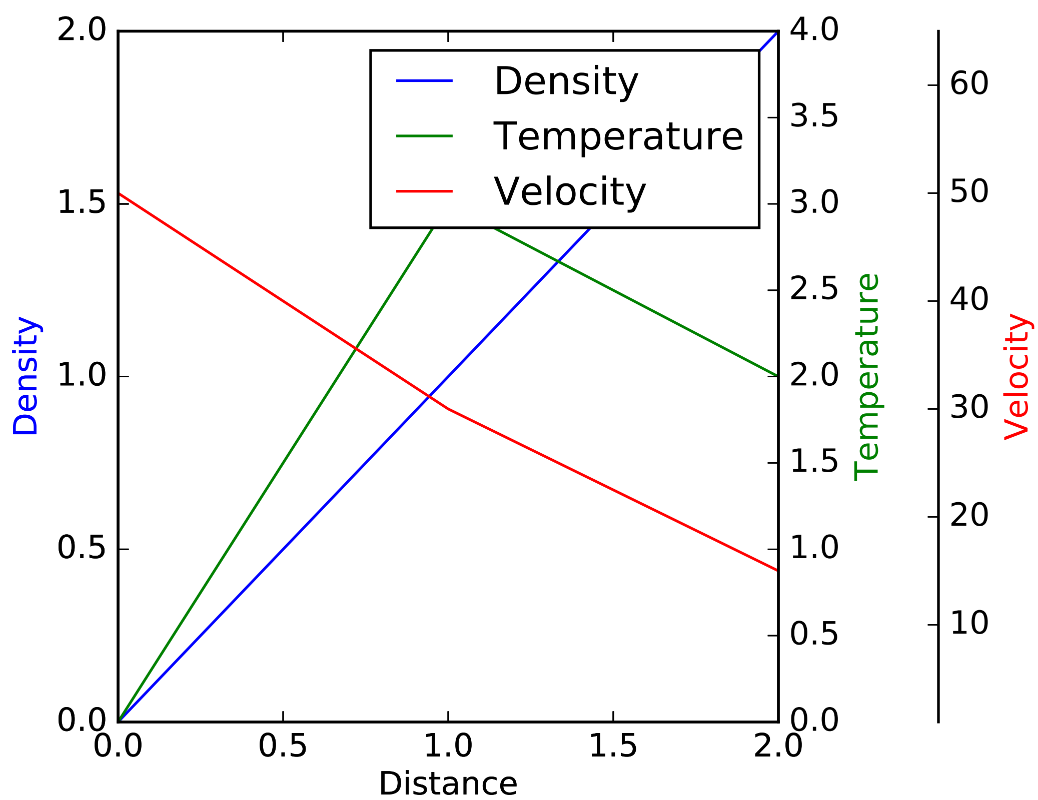


```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1 import ImageGrid
fig = plt.figure(1)

grid1 = ImageGrid(fig, 121, (2, 2), axes_pad=0.1,
                  aspect=True, share_all=True)
```

```
for i in [0, 1]:
    grid1[i].set_aspect(2)


grid2 = ImageGrid(fig, 122, (2, 2), axes_pad=0.1,
                  aspect=True, share_all=True)


for i in [1, 3]:
    grid2[i].set_aspect(2)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.14 axes_grid example code: demo_parasite_axes2.py

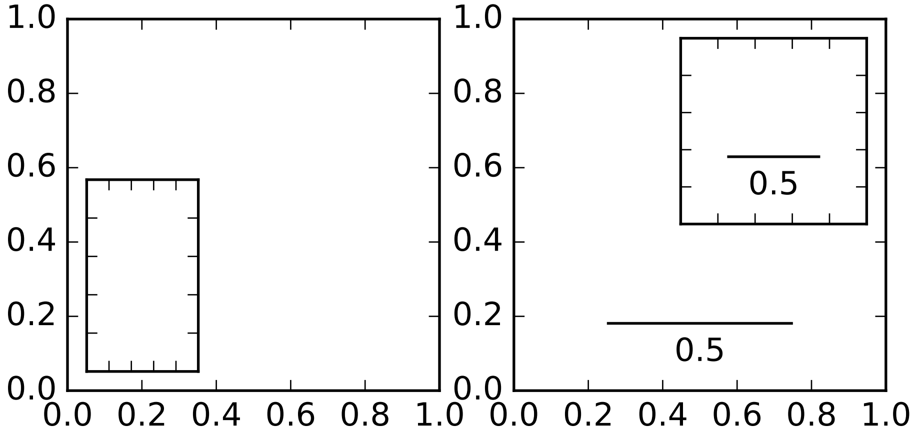



```
from mpl_toolkits.axes_grid1 import host_subplot
import mpl_toolkits.axisartist as AA
import matplotlib.pyplot as plt
```

```
if 1:

    host = host_subplot(111, axes_class=AA.Axes)
    plt.subplots_adjust(right=0.75)

    par1 = host.twinx()
    par2 = host.twinx()

    offset = 60
    new_fixed_axis = par2.get_grid_helper().new_fixed_axis
    par2.axis["right"] = new_fixed_axis(loc="right",
                                        axes=par2,
                                        offset=(offset, 0))

    par2.axis["right"].toggle(all=True)

    host.set_xlim(0, 2)
    host.set_ylim(0, 2)

    host.set_xlabel("Distance")
    host.set_ylabel("Density")
    par1.set_ylabel("Temperature")
    par2.set_ylabel("Velocity")

    p1, = host.plot([0, 1, 2], [0, 1, 2], label="Density")
    p2, = par1.plot([0, 1, 2], [0, 3, 2], label="Temperature")
    p3, = par2.plot([0, 1, 2], [50, 30, 15], label="Velocity")

    par1.set_ylim(0, 4)
    par2.set_ylim(1, 65)

    host.legend()

    host.axis["left"].label.set_color(p1.get_color())
    par1.axis["right"].label.set_color(p2.get_color())
    par2.axis["right"].label.set_color(p3.get_color())

    plt.draw()
    plt.show()

    #plt.savefig("Test")
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.15 axes_grid example code: inset_locator_demo.py

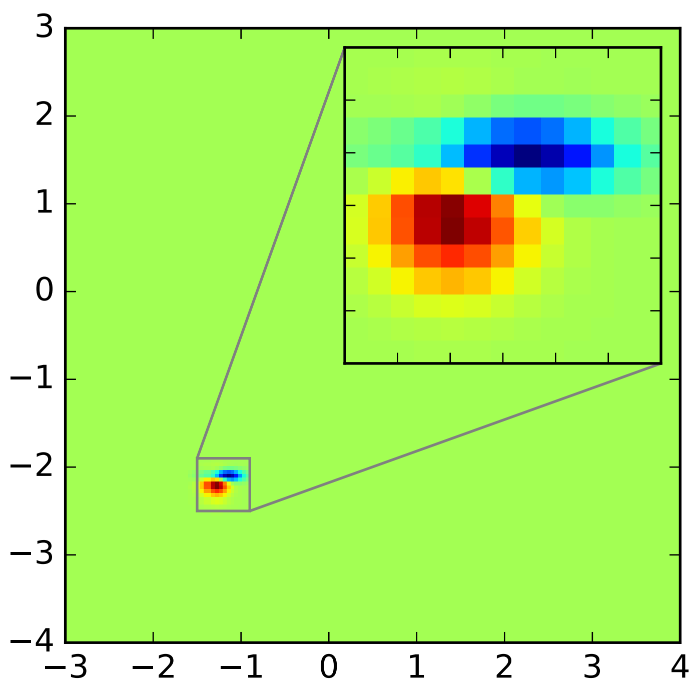



```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.inset_locator import inset_axes, zoomed_inset_axes
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar


def add_sizebar(ax, size):
    asb = AnchoredSizeBar(ax.transData,
                          size,
                          str(size),
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)
    ax.add_artist(asb)


fig, (ax, ax2) = plt.subplots(1, 2, figsize=[5.5, 3])

# first subplot
ax.set_aspect(1.)

axins = inset_axes(ax,
                   width="30%",  # width = 30% of parent_bbox
                   height=1.,  # height : 1 inch
                   loc=3)

plt.xticks(visible=False)
plt.yticks(visible=False)


# second subplot
```

```
ax2.set_aspect(1.)

axins = zoomed_inset_axes(ax2, 0.5, loc=1)  # zoom = 0.5

plt.xticks(visible=False)
plt.yticks(visible=False)

add_sizebar(ax2, 0.5)
add_sizebar(axins, 0.5)

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.16 axes_grid example code: inset_locator_demo2.py

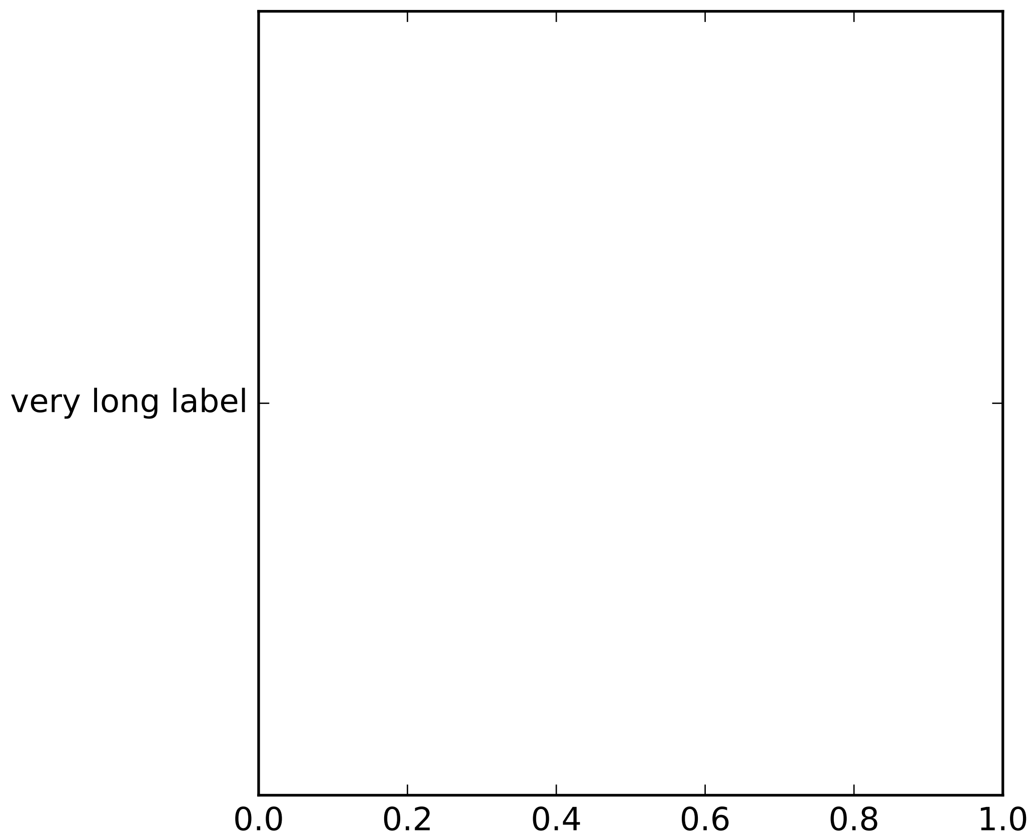


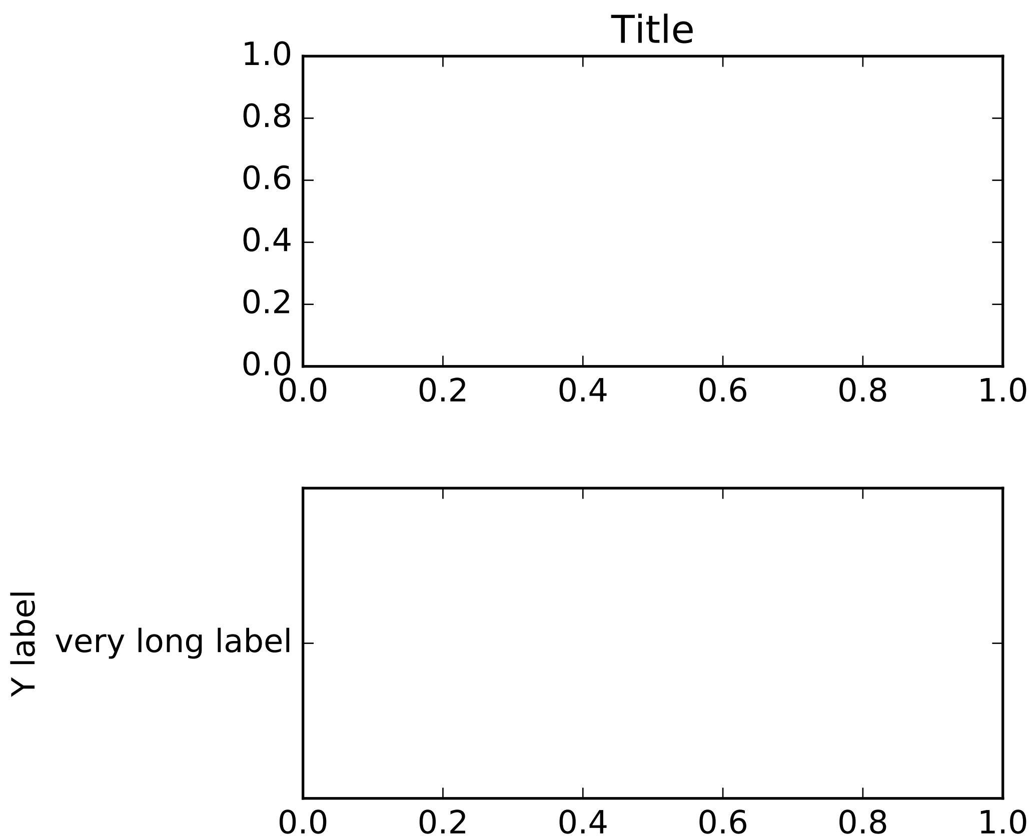

```
import matplotlib.pyplot as plt

from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset

import numpy as np
```

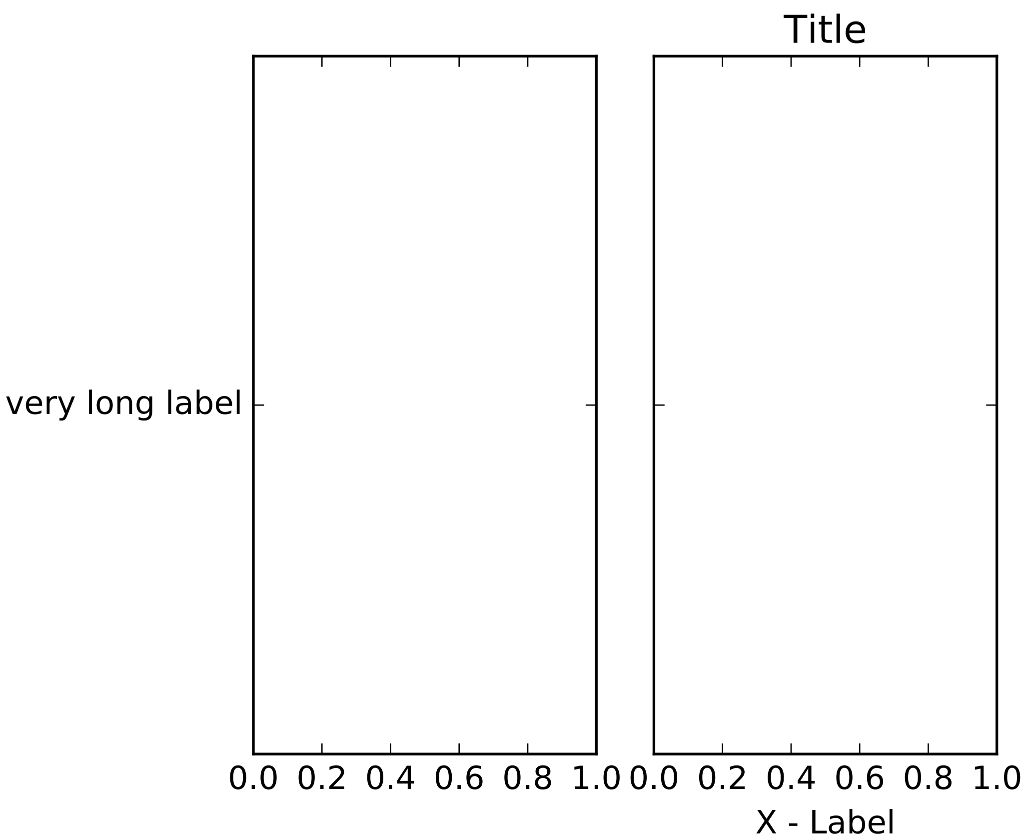
```
def get_demo_image():
    from matplotlib.cbook import get_sample_data
    import numpy as np
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

fig, ax = plt.subplots(figsize=[5, 4])

# prepare the demo image
Z, extent = get_demo_image()
Z2 = np.zeros([150, 150], dtype="d")
ny, nx = Z.shape
Z2[30:30 + ny, 30:30 + nx] = Z

# extent = [-3, 4, -4, 3]
ax.imshow(Z2, extent=extent, interpolation="nearest",
          origin="lower")

axins = zoomed_inset_axes(ax, 6, loc=1)  # zoom = 6
axins.imshow(Z2, extent=extent, interpolation="nearest",
             origin="lower")

# sub region of the original image
x1, x2, y1, y2 = -1.5, -0.9, -2.5, -1.9
axins.set_xlim(x1, x2)
axins.set_ylim(y1, y2)

plt.xticks(visible=False)
plt.yticks(visible=False)

# draw a bbox of the region of the inset axes in the parent axes and
# connecting lines between the bbox and the inset axes area
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.17 axes_grid example code: make_room_for_ylabel_using_axesgrid.py




Title
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.2 0.4 0.6 0.8 1.0
Y label
very long label
0.0 0.2 0.4 0.6 0.8 1.0

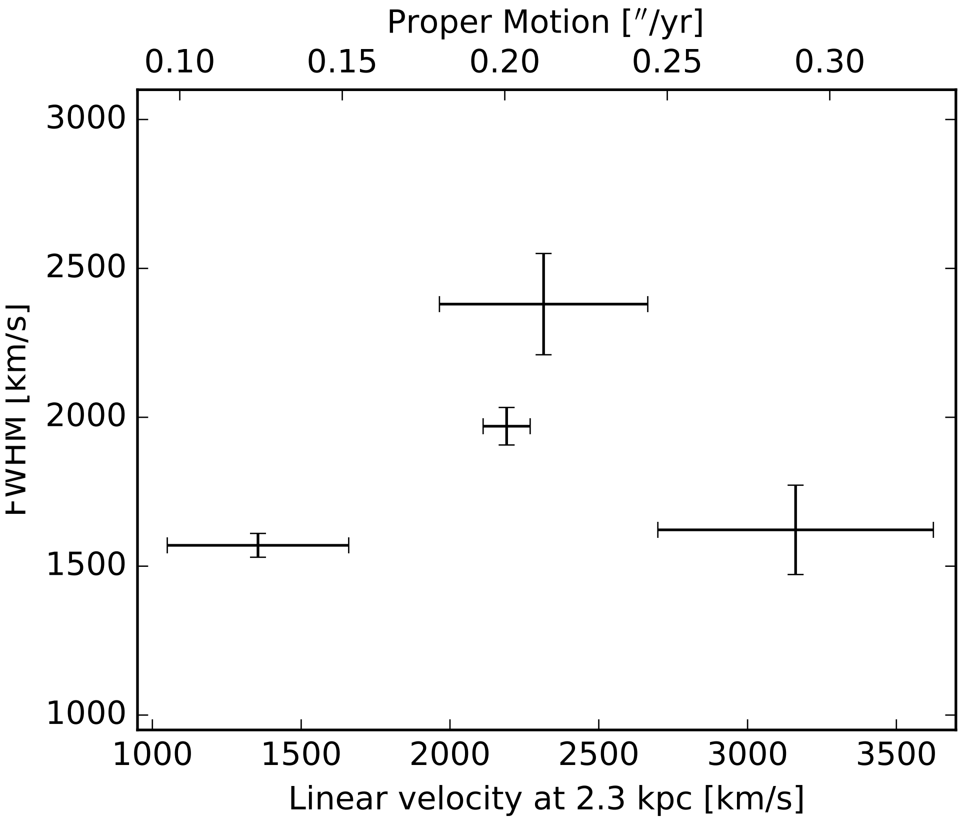



```
from mpl_toolkits.axes_grid1 import make_axes_locatable
from mpl_toolkits.axes_grid1.axes_divider import make_axes_area_auto_adjustable


if __name__ == "__main__":

    import matplotlib.pyplot as plt

    def ex1():
        plt.figure(1)
        ax = plt.axes([0, 0, 1, 1])
        #ax = plt.subplot(111)

        ax.set_yticks([0.5])
        ax.set_yticklabels(["very long label"])

        make_axes_area_auto_adjustable(ax)

    def ex2():

        plt.figure(2)
        ax1 = plt.axes([0, 0, 1, 0.5])
        ax2 = plt.axes([0, 0.5, 1, 0.5])

        ax1.set_yticks([0.5])
        ax1.set_yticklabels(["very long label"])
```

```
    ax1.set_ylabel("Y label")

    ax2.set_title("Title")

    make_axes_area_auto_adjustable(ax1, pad=0.1, use_axes=[ax1, ax2])
    make_axes_area_auto_adjustable(ax2, pad=0.1, use_axes=[ax1, ax2])

def ex3():

    fig = plt.figure(3)
    ax1 = plt.axes([0, 0, 1, 1])
    divider = make_axes_locatable(ax1)

    ax2 = divider.new_horizontal("100%", pad=0.3, sharey=ax1)
    ax2.tick_params(labelleft="off")
    fig.add_axes(ax2)

    divider.add_auto_adjustable_area(use_axes=[ax1], pad=0.1,
                                     adjust_dirs=["left"])
    divider.add_auto_adjustable_area(use_axes=[ax2], pad=0.1,
                                     adjust_dirs=["right"])
    divider.add_auto_adjustable_area(use_axes=[ax1, ax2], pad=0.1,
                                     adjust_dirs=["top", "bottom"])

    ax1.set_yticks([0.5])
    ax1.set_yticklabels(["very long label"])

    ax2.set_title("Title")
    ax2.set_xlabel("X - Label")

ex1()
ex2()
ex3()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.18 axes_grid example code: parasite_simple2.py

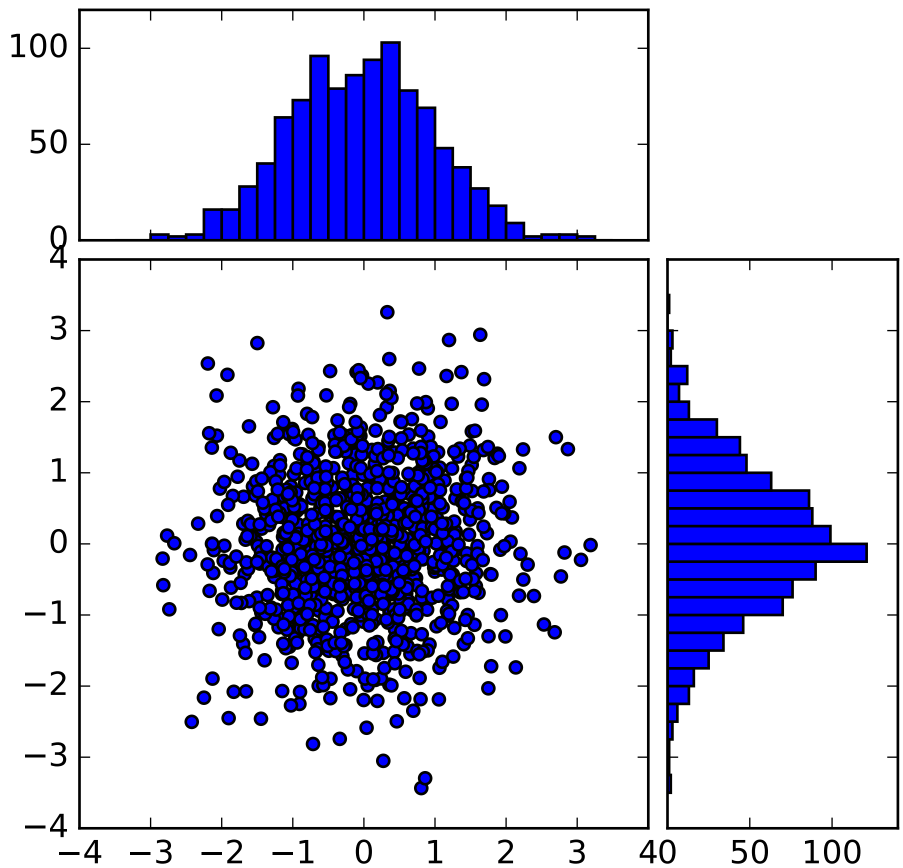



```
import matplotlib.transforms as mtransforms
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.parasite_axes import SubplotHost

obs = [["01_S1", 3.88, 0.14, 1970, 63],
       ["01_S4", 5.6, 0.82, 1622, 150],
       ["02_S1", 2.4, 0.54, 1570, 40],
       ["03_S1", 4.1, 0.62, 2380, 170]]


fig = plt.figure()

ax_kms = SubplotHost(fig, 1, 1, 1, aspect=1.)

# angular proper motion("/yr) to linear velocity(km/s) at distance=2.3kpc
pm_to_kms = 1./206265.*2300*3.085e18/3.15e7/1.e5

aux_trans = mtransforms.Affine2D().scale(pm_to_kms, 1.)
ax_pm = ax_kms.twin(aux_trans)
ax_pm.set_viewlim_mode("transform")

fig.add_subplot(ax_kms)
```

```
for n, ds, dse, w, we in obs:
    time = ((2007 + (10. + 4/30.)/12) - 1988.5)
    v = ds / time * pm_to_kms
    ve = dse / time * pm_to_kms
    ax_kms.errorbar([v], [w], xerr=[ve], yerr=[we], color="k")


ax_kms.axis["bottom"].set_label("Linear velocity at 2.3 kpc [km/s]")
ax_kms.axis["left"].set_label("FWHM [km/s]")
ax_pm.axis["top"].set_label(r"Proper Motion [$''$/yr]")
ax_pm.axis["top"].label.set_visible(True)
ax_pm.axis["right"].major_ticklabels.set_visible(False)

ax_kms.set_xlim(950, 3700)
ax_kms.set_ylim(950, 3100)
# xlim and ylim of ax_pms will be automatically adjusted.

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.19 axes_grid example code: scatter_hist.py

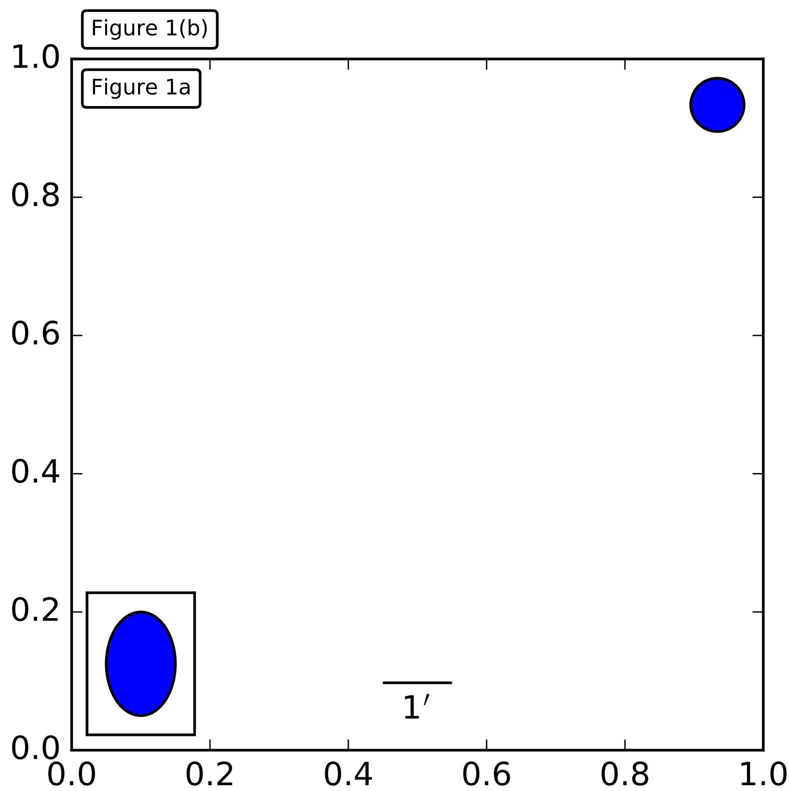



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)


fig, axScatter = plt.subplots(figsize=(5.5, 5.5))

# the scatter plot:
axScatter.scatter(x, y)
axScatter.set_aspect(1.)

# create new axes on the right and on the top of the current axes
```

```
# The first argument of the new_vertical(new_horizontal) method is
# the height (width) of the axes to be created in inches.
divider = make_axes_locatable(axScatter)
axHistx = divider.append_axes("top", 1.2, pad=0.1, sharex=axScatter)
axHisty = divider.append_axes("right", 1.2, pad=0.1, sharey=axScatter)

# make some labels invisible
plt.setp(axHistx.get_xticklabels() + axHisty.get_yticklabels(),
         visible=False)

# now determine nice limits by hand:
binwidth = 0.25
xymax = np.max([np.max(np.fabs(x)), np.max(np.fabs(y))])
lim = (int(xymax/binwidth) + 1)*binwidth

bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

# the xaxis of axHistx and yaxis of axHisty are shared with axScatter,
# thus there is no need to manually adjust the xlim and ylim of these
# axis.

#axHistx.axis["bottom"].major_ticklabels.set_visible(False)
for tl in axHistx.get_xticklabels():
    tl.set_visible(False)
axHistx.set_yticks([0, 50, 100])

#axHisty.axis["left"].major_ticklabels.set_visible(False)
for tl in axHisty.get_yticklabels():
    tl.set_visible(False)
axHisty.set_xticks([0, 50, 100])

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.20 axes_grid example code: simple_anchored_artists.py

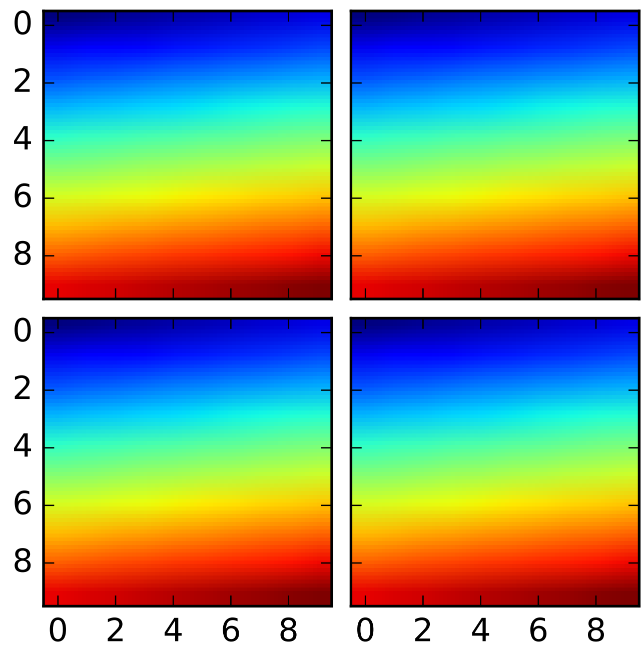



```
import matplotlib.pyplot as plt


def draw_text(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredText
    at = AnchoredText("Figure 1a",
                      loc=2, prop=dict(size=8), frameon=True,
                      )
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    at2 = AnchoredText("Figure 1(b)",
                       loc=3, prop=dict(size=8), frameon=True,
                       bbox_to_anchor=(0., 1.),
                       bbox_transform=ax.transAxes
                       )
    at2.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at2)


def draw_circle(ax):  # circle in the canvas coordinate
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDrawingArea
```

```
    from matplotlib.patches import Circle
    ada = AnchoredDrawingArea(20, 20, 0, 0,
                              loc=1, pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)


def draw_ellipse(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredEllipse
    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
                         loc=3, pad=0.5, borderpad=0.4, frameon=True)

    ax.add_artist(ae)


def draw_sizebar(ax):
    from mpl_toolkits.axes_grid1.anchored_artists import AnchoredSizeBar
    # draw a horizontal bar with length of 0.1 in Data coordinate
    # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData,
                          0.1,
                          r"1$^{\prime}$",
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)
    ax.add_artist(asb)


if 1:
    ax = plt.gca()
    ax.set_aspect(1.)

    draw_text(ax)
    draw_circle(ax)
    draw_ellipse(ax)
    draw_sizebar(ax)

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

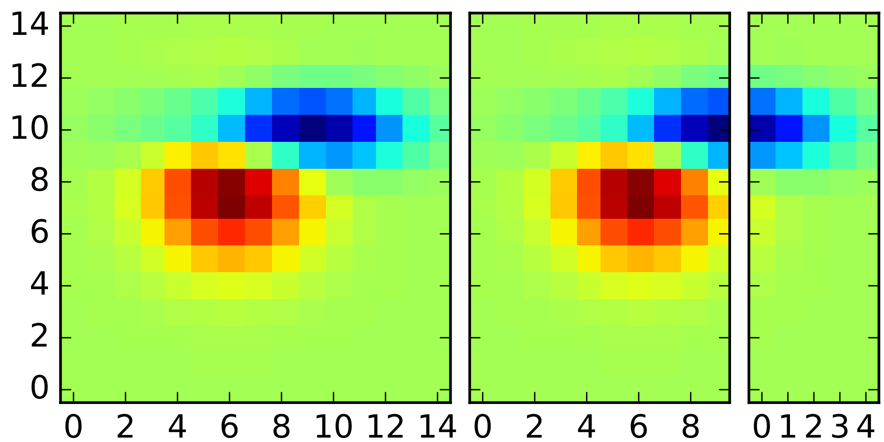
82.21 axes_grid example code: simple_axesgrid.py




```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
import numpy as np

im = np.arange(100)
im.shape = 10, 10

fig = plt.figure(1, (4., 4.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
                 nrows_ncols=(2, 2),  # creates 2x2 grid of axes
                 axes_pad=0.1,  # pad between axes in inch.
                 )

for i in range(4):
    grid[i].imshow(im)  # The AxesGrid object work as a list of axes.

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.22 axes_grid example code: simple_axesgrid2.py

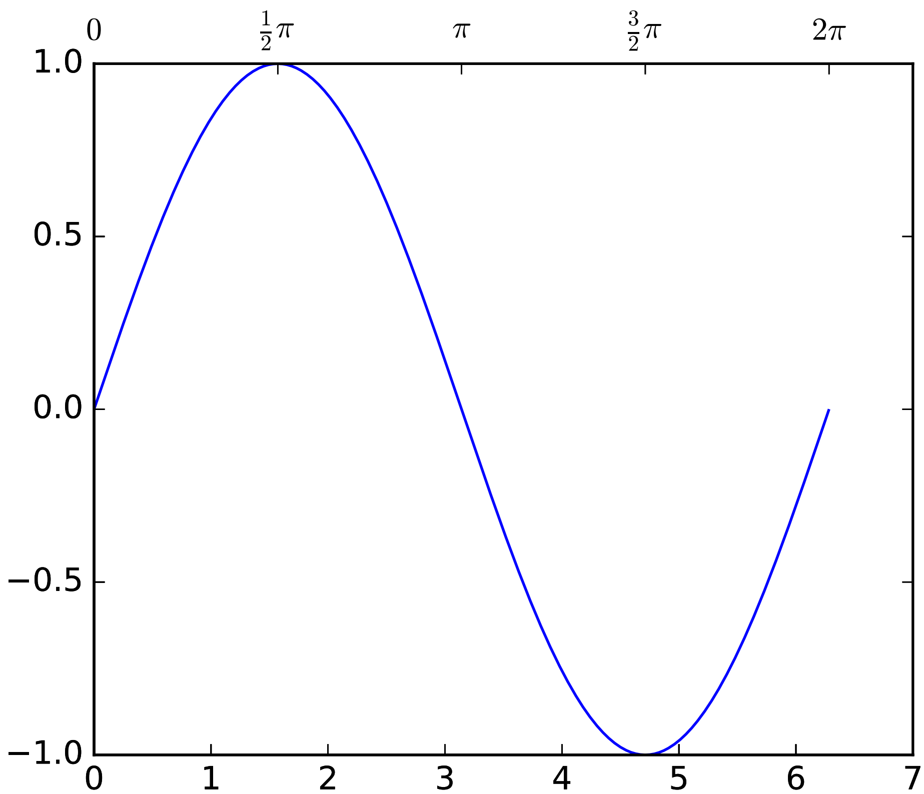



```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid


def get_demo_image():
    import numpy as np
    from matplotlib.cbook import get_sample_data
    f = get_sample_data("axes_grid/bivariate_normal.npy", asfileobj=False)
    z = np.load(f)
    # z is a numpy array of 15x15
    return z, (-3, 4, -4, 3)

F = plt.figure(1, (5.5, 3.5))
grid = ImageGrid(F, 111,  # similar to subplot(111)
                 nrows_ncols=(1, 3),
                 axes_pad=0.1,
                 add_all=True,
                 label_mode="L",
                 )

Z, extent = get_demo_image()  # demo image

im1 = Z
im2 = Z[:, :10]
im3 = Z[:, 10:]
vmin, vmax = Z.min(), Z.max()
for i, im in enumerate([im1, im2, im3]):
    ax = grid[i]
```

```
    ax.imshow(im, origin="lower", vmin=vmin,
              vmax=vmax, interpolation="nearest")

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

82.23 axes_grid example code: simple_axisline4.py




```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import host_subplot
import mpl_toolkits.axisartist as AA
import numpy as np

ax = host_subplot(111, axes_class=AA.Axes)
xx = np.arange(0, 2*np.pi, 0.01)
ax.plot(xx, np.sin(xx))

ax2 = ax.twin()  # ax2 is responsible for "top" axis and "right" axis
ax2.set_xticks([0., .5*np.pi, np.pi, 1.5*np.pi, 2*np.pi])
ax2.set_xticklabels(["$0$", r"$\frac{1}{2}\pi$",
```

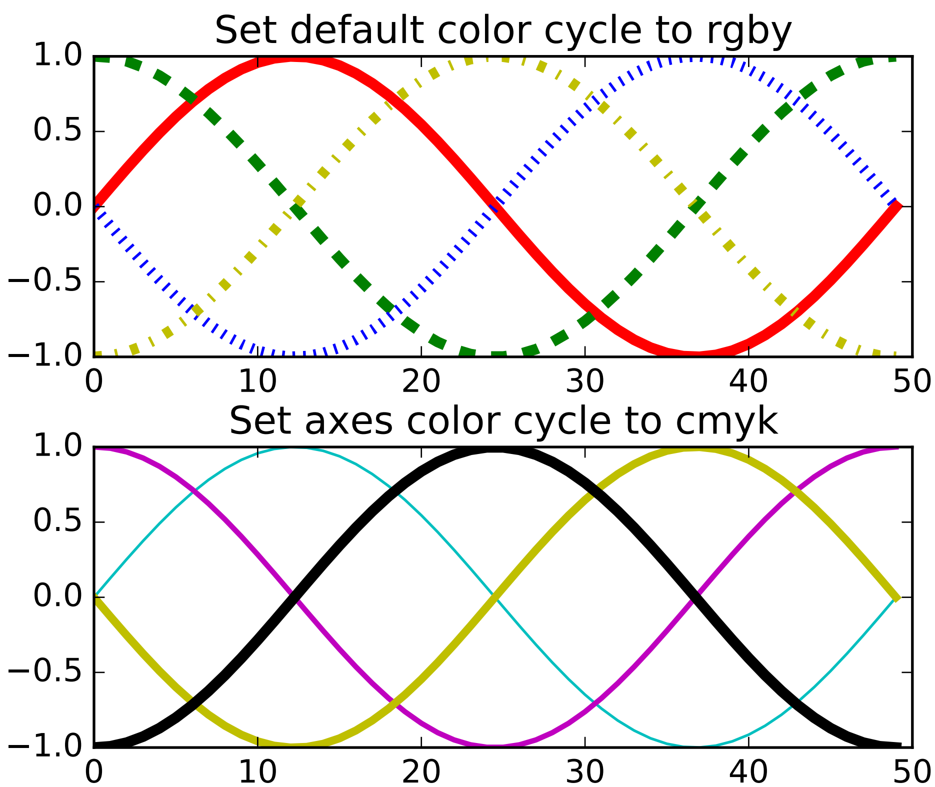
```
                     r"$\pi$", r"$\frac{3}{2}\pi$", r"$2\pi$"])

ax2.axis["right"].major_ticklabels.set_visible(False)

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER EIGHTYTHREE

COLOR EXAMPLES

83.1 color example code: color_cycle_demo.py

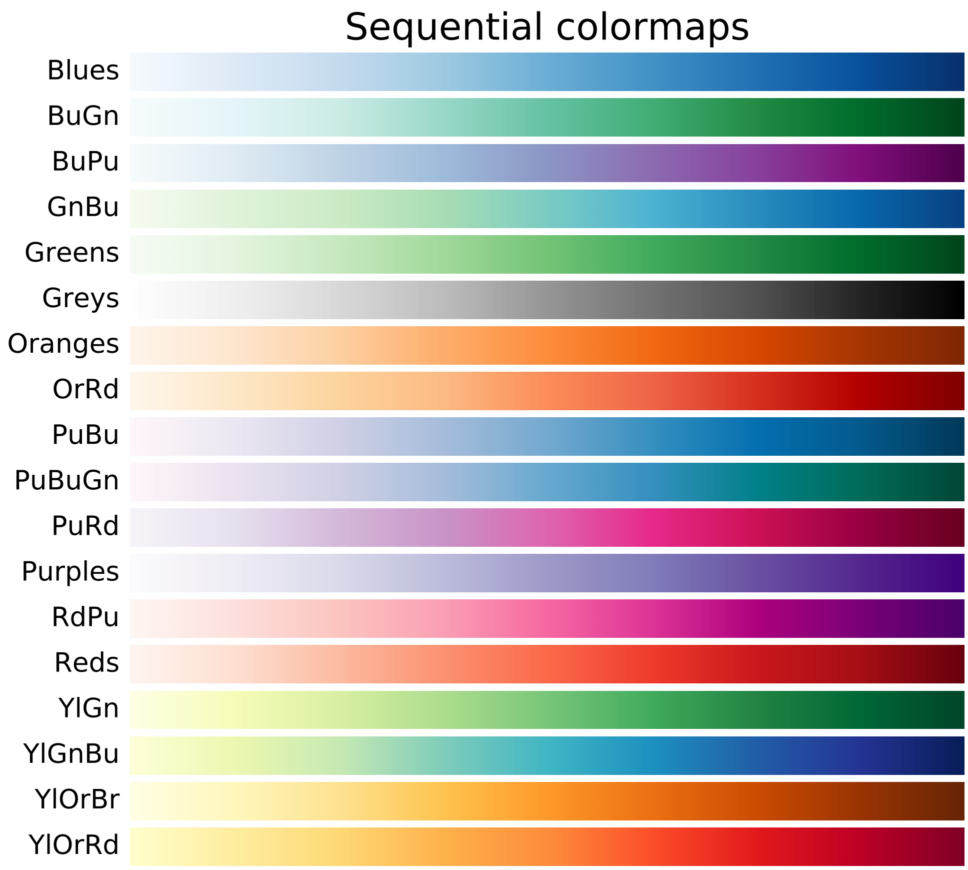


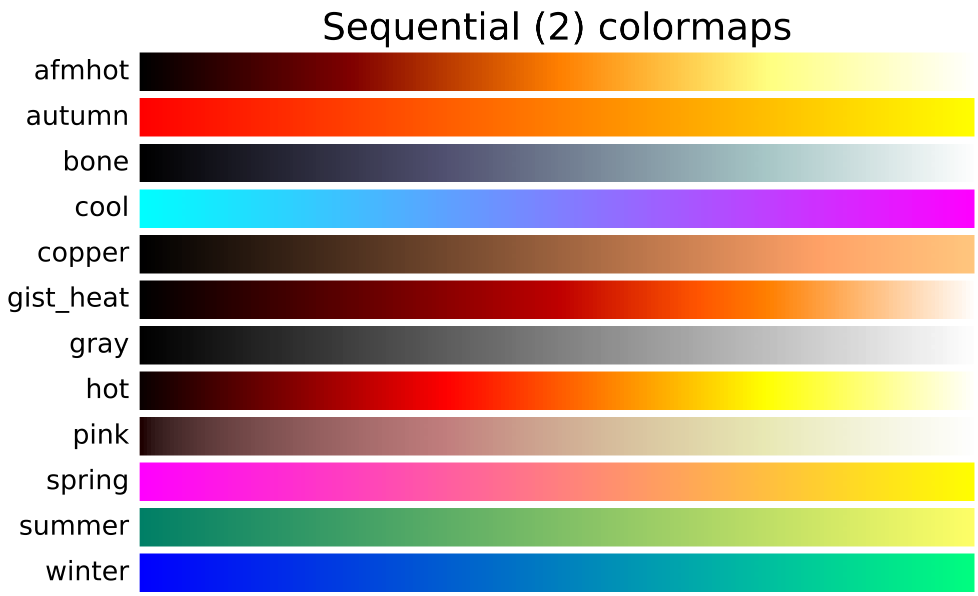

```
"""
Demo of custom property-cycle settings to control colors and such
for multi-line plots.

This example demonstrates two different APIs:

    1. Setting the default rc-parameter specifying the property cycle.
       This affects all subsequent axes (but not axes already created).
    2. Setting the property cycle for a specific axes. This only
```

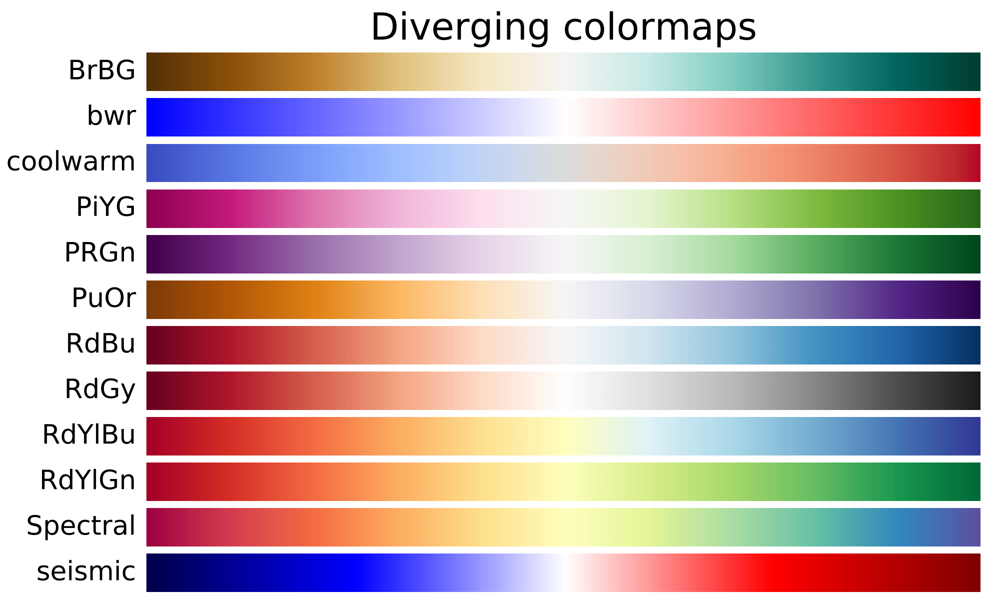
```
    affects a single axes.
"""
from cycler import cycler
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi)
offsets = np.linspace(0, 2*np.pi, 4, endpoint=False)
# Create array with shifted-sine curve along each column
yy = np.transpose([np.sin(x + phi) for phi in offsets])

plt.rc('lines', linewidth=4)
plt.rc('axes', prop_cycle=(cycler('color', ['r', 'g', 'b', 'y']) +
                           cycler('linestyle', ['-', '--', ':', '-.'])))
fig, (ax0, ax1) = plt.subplots(nrows=2)
ax0.plot(yy)
ax0.set_title('Set default color cycle to rgby')

ax1.set_prop_cycle(cycler('color', ['c', 'm', 'y', 'k']) +
                   cycler('lw', [1, 2, 3, 4]))
ax1.plot(yy)
ax1.set_title('Set axes color cycle to cmyk')

# Tweak spacing between subplots to prevent labels from overlapping
plt.subplots_adjust(hspace=0.3)
plt.show()
```

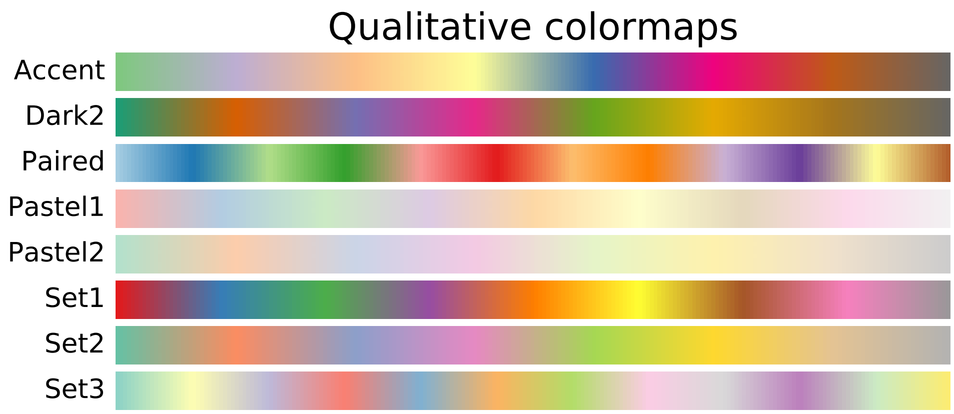
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

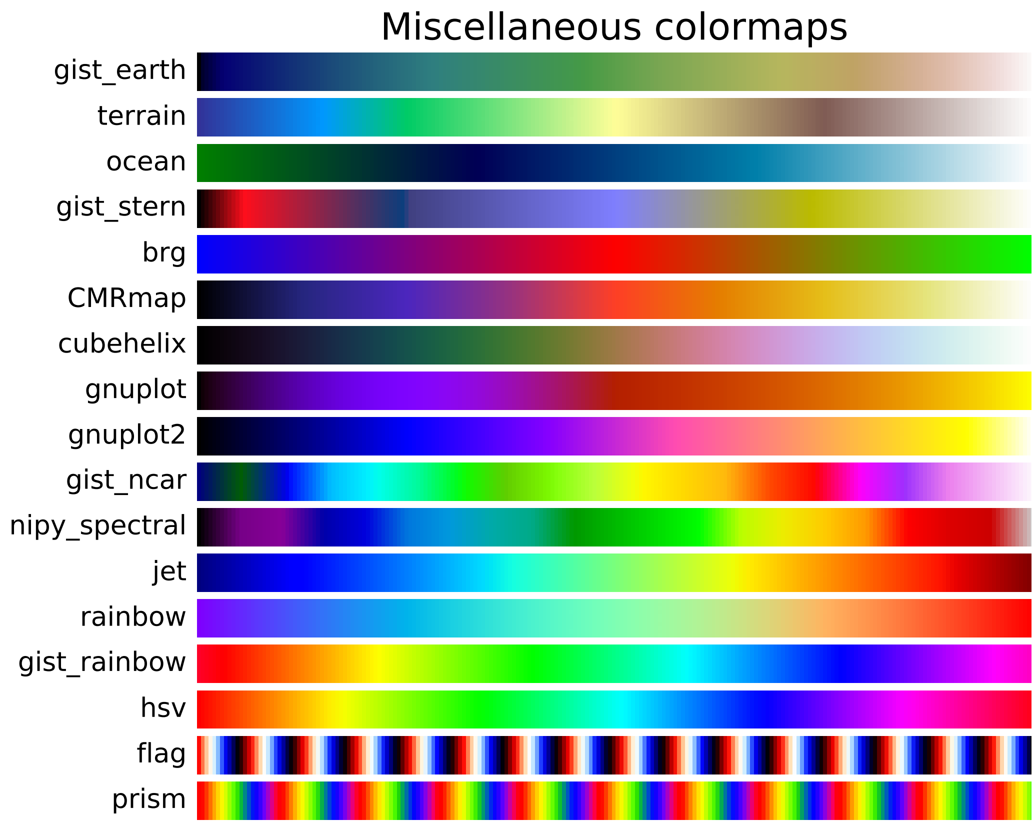
83.2 color example code: colormaps_reference.py


Sequential colormaps
Blues
BuGn
BuPu
GnBu
Greens
Greys
Oranges
OrRd
PuBu
PuBuGn
PuRd
Purples
RdPu
Reds
YlGn
YlGnBu
YlOrBr
YlOrRd


Sequential (2) colormaps
afmhot
autumn
bone
cool
copper
gist_heat
gray
hot
pink
spring
summer
winter


Diverging colormaps
BrBG
bwr
coolwarm
PiYG
PRGn
PuOr
RdBu
RdGy
RdYlBu
RdYlGn
Spectral
seismic


Qualitative colormaps
Accent
Dark2
Paired
Pastel1
Pastel2
Set1
Set2
Set3



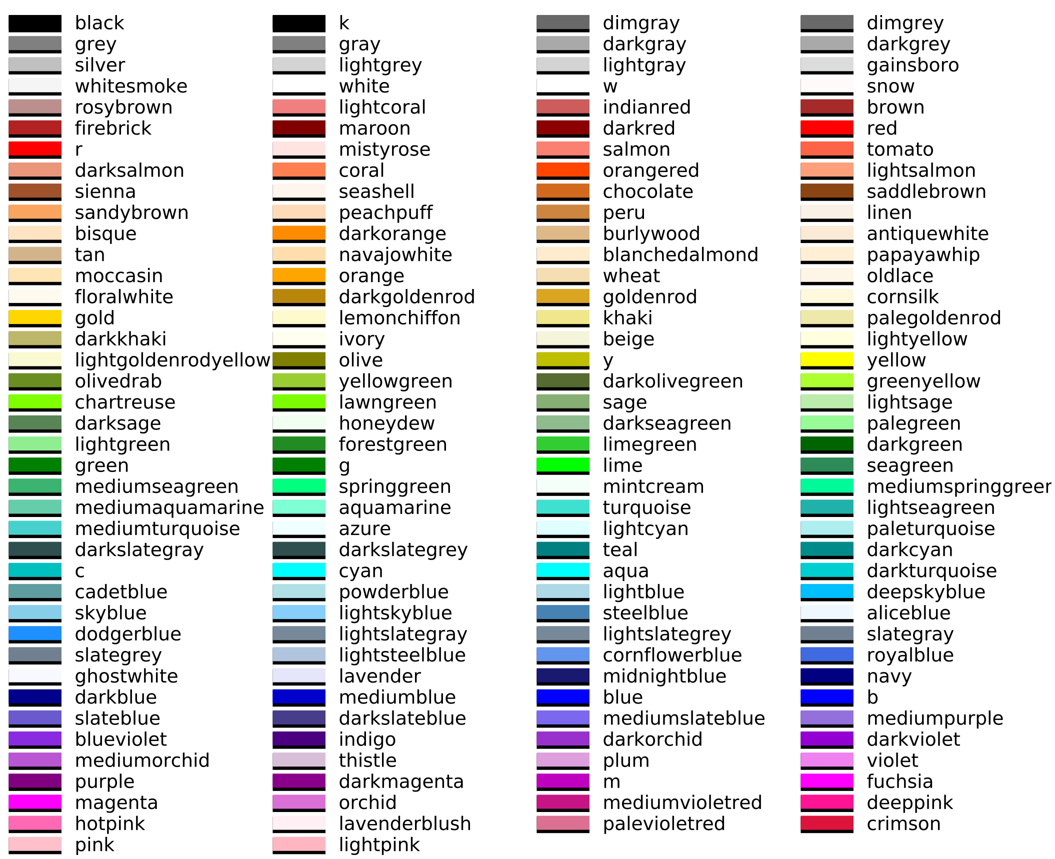

```
"""
Reference for colormaps included with Matplotlib.

This reference example shows all colormaps included with Matplotlib. Note that
any colormap listed here can be reversed by appending "_r" (e.g., "pink_r").
These colormaps are divided into the following categories:

Sequential:
    These colormaps are approximately monochromatic colormaps varying smoothly
    between two color tones---usually from low saturation (e.g. white) to high
    saturation (e.g. a bright blue). Sequential colormaps are ideal for
    representing most scientific data since they show a clear progression from
    low-to-high values.

Diverging:
    These colormaps have a median value (usually light in color) and vary
    smoothly to two different color tones at high and low values. Diverging
    colormaps are ideal when your data has a median value that is significant
    (e.g.  0, such that positive and negative values are represented by
    different colors of the colormap).

Qualitative:
    These colormaps vary rapidly in color. Qualitative colormaps are useful for
    choosing a set of discrete colors. For example::

        color_list = plt.cm.Set3(np.linspace(0, 1, 12))
```

```
    gives a list of RGB colors that are good for plotting a series of lines on
    a dark background.

Miscellaneous:
    Colormaps that don't fit into the categories above.

"""
import numpy as np
import matplotlib.pyplot as plt

# Have colormaps separated into categories:
# http://matplotlib.org/examples/color/colormaps_reference.html

cmaps = [('Perceptually Uniform Sequential',
                            ['viridis', 'inferno', 'plasma', 'magma']),
         ('Sequential',     ['Blues', 'BuGn', 'BuPu',
                             'GnBu', 'Greens', 'Greys', 'Oranges', 'OrRd',
                             'PuBu', 'PuBuGn', 'PuRd', 'Purples', 'RdPu',
                             'Reds', 'YlGn', 'YlGnBu', 'YlOrBr', 'YlOrRd']),
         ('Sequential (2)', ['afmhot', 'autumn', 'bone', 'cool',
                             'copper', 'gist_heat', 'gray', 'hot',
                             'pink', 'spring', 'summer', 'winter']),
         ('Diverging',      ['BrBG', 'bwr', 'coolwarm', 'PiYG', 'PRGn', 'PuOr',
                             'RdBu', 'RdGy', 'RdYlBu', 'RdYlGn', 'Spectral',
                             'seismic']),
         ('Qualitative',    ['Accent', 'Dark2', 'Paired', 'Pastel1',
                             'Pastel2', 'Set1', 'Set2', 'Set3']),
         ('Miscellaneous',  ['gist_earth', 'terrain', 'ocean', 'gist_stern',
                             'brg', 'CMRmap', 'cubehelix',
                             'gnuplot', 'gnuplot2', 'gist_ncar',
                             'nipy_spectral', 'jet', 'rainbow',
                             'gist_rainbow', 'hsv', 'flag', 'prism'])]


nrows = max(len(cmap_list) for cmap_category, cmap_list in cmaps)
gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))


def plot_color_gradients(cmap_category, cmap_list):
    fig, axes = plt.subplots(nrows=nrows)
    fig.subplots_adjust(top=0.95, bottom=0.01, left=0.2, right=0.99)
    axes[0].set_title(cmap_category + ' colormaps', fontsize=14)

    for ax, name in zip(axes, cmap_list):
        ax.imshow(gradient, aspect='auto', cmap=plt.get_cmap(name))
        pos = list(ax.get_position().bounds)
        x_text = pos[0] - 0.01
        y_text = pos[1] + pos[3]/2.
        fig.text(x_text, y_text, name, va='center', ha='right', fontsize=10)

    # Turn off *all* ticks & spines, not just the ones with colormaps.
```

```
    for ax in axes:
        ax.set_axis_off()

for cmap_category, cmap_list in cmaps:
    plot_color_gradients(cmap_category, cmap_list)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

83.3 color example code: named_colors.py




```
"""
Visualization of named colors.

Simple plot example with the named colors and its visual representation.
"""

from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

import six
```

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors


colors_ = list(six.iteritems(colors.cnames))

# Add the single letter colors.
for name, rgb in six.iteritems(colors.ColorConverter.colors):
    hex_ = colors.rgb2hex(rgb)
    colors_.append((name, hex_))

# Transform to hex color values.
hex_ = [color[1] for color in colors_]
# Get the rgb equivalent.
rgb = [colors.hex2color(color) for color in hex_]
# Get the hsv equivalent.
hsv = [colors.rgb_to_hsv(color) for color in rgb]

# Split the hsv values to sort.
hue = [color[0] for color in hsv]
sat = [color[1] for color in hsv]
val = [color[2] for color in hsv]

# Sort by hue, saturation and value.
ind = np.lexsort((val, sat, hue))
sorted_colors = [colors_[i] for i in ind]

n = len(sorted_colors)
ncols = 4
nrows = int(np.ceil(1. * n / ncols))

fig, ax = plt.subplots()

X, Y = fig.get_dpi() * fig.get_size_inches()

# row height
h = Y / (nrows + 1)
# col width
w = X / ncols

for i, (name, color) in enumerate(sorted_colors):
    col = i % ncols
    row = int(i / ncols)
    y = Y - (row * h) - h

    xi_line = w * (col + 0.05)
    xf_line = w * (col + 0.25)
    xi_text = w * (col + 0.3)

    ax.text(xi_text, y, name, fontsize=(h * 0.8),
            horizontalalignment='left',
```

```
            verticalalignment='center')

    # Add extra black line a little bit thicker to make
    # clear colors more visible.
    ax.hlines(y, xi_line, xf_line, color='black', linewidth=(h * 0.7))
    ax.hlines(y + h * 0.1, xi_line, xf_line, color=color, linewidth=(h * 0.6))

ax.set_xlim(0, X)
ax.set_ylim(0, Y)
ax.set_axis_off()

fig.subplots_adjust(left=0, right=1,
                    top=1, bottom=0,
                    hspace=0, wspace=0)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

EIGHTYFOUR

EVENT_HANDLING EXAMPLES

84.1 event_handling example code: close_event.py

[source code]

```
from __future__ import print_function
import matplotlib.pyplot as plt


def handle_close(evt):
    print('Closed Figure!')

fig = plt.figure()
fig.canvas.mpl_connect('close_event', handle_close)

plt.text(0.35, 0.5, 'Close Me!', dict(size=30))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.2 event_handling example code: data_browser.py

[source code]

```
import numpy as np


class PointBrowser(object):
    """
    Click on a point to select and highlight it -- the data that
    generated the point will be shown in the lower axes.  Use the 'n'
    and 'p' keys to browse through the next and previous points
    """

    def __init__(self):
        self.lastind = 0

        self.text = ax.text(0.05, 0.95, 'selected: none',
```

```
                                transform=ax.transAxes, va='top')
        self.selected, = ax.plot([xs[0]], [ys[0]], 'o', ms=12, alpha=0.4,
                                 color='yellow', visible=False)

    def onpress(self, event):
        if self.lastind is None:
            return
        if event.key not in ('n', 'p'):
            return
        if event.key == 'n':
            inc = 1
        else:
            inc = -1

        self.lastind += inc
        self.lastind = np.clip(self.lastind, 0, len(xs) - 1)
        self.update()

    def onpick(self, event):

        if event.artist != line:
            return True

        N = len(event.ind)
        if not N:
            return True

        # the click locations
        x = event.mouseevent.xdata
        y = event.mouseevent.ydata

        distances = np.hypot(x - xs[event.ind], y - ys[event.ind])
        indmin = distances.argmin()
        dataind = event.ind[indmin]

        self.lastind = dataind
        self.update()

    def update(self):
        if self.lastind is None:
            return

        dataind = self.lastind

        ax2.cla()
        ax2.plot(X[dataind])

        ax2.text(0.05, 0.9, 'mu=%1.3f\nsigma=%1.3f' % (xs[dataind], ys[dataind]),
                 transform=ax2.transAxes, va='top')
        ax2.set_ylim(-0.5, 1.5)
        self.selected.set_visible(True)
        self.selected.set_data(xs[dataind], ys[dataind])
```

```
        self.text.set_text('selected: %d' % dataind)
        fig.canvas.draw()


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    X = np.random.rand(100, 200)
    xs = np.mean(X, axis=1)
    ys = np.std(X, axis=1)

    fig, (ax, ax2) = plt.subplots(2, 1)
    ax.set_title('click on point to plot time series')
    line, = ax.plot(xs, ys, 'o', picker=5)  # 5 points tolerance

    browser = PointBrowser()

    fig.canvas.mpl_connect('pick_event', browser.onpick)
    fig.canvas.mpl_connect('key_press_event', browser.onpress)

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.3 event_handling example code: figure_axes_enter_leave.py

[source code]

```
"""
Illustrate the figure and axes enter and leave events by changing the
frame colors on enter and leave
"""
from __future__ import print_function
import matplotlib.pyplot as plt


def enter_axes(event):
    print('enter_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('yellow')
    event.canvas.draw()


def leave_axes(event):
    print('leave_axes', event.inaxes)
    event.inaxes.patch.set_facecolor('white')
    event.canvas.draw()


def enter_figure(event):
    print('enter_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('red')
```

```
    event.canvas.draw()


def leave_figure(event):
    print('leave_figure', event.canvas.figure)
    event.canvas.figure.patch.set_facecolor('grey')
    event.canvas.draw()

fig1, (ax, ax2) = plt.subplots(2, 1)
fig1.suptitle('mouse hover over figure or axes to trigger events')

fig1.canvas.mpl_connect('figure_enter_event', enter_figure)
fig1.canvas.mpl_connect('figure_leave_event', leave_figure)
fig1.canvas.mpl_connect('axes_enter_event', enter_axes)
fig1.canvas.mpl_connect('axes_leave_event', leave_axes)

fig2, (ax, ax2) = plt.subplots(2, 1)
fig2.suptitle('mouse hover over figure or axes to trigger events')

fig2.canvas.mpl_connect('figure_enter_event', enter_figure)
fig2.canvas.mpl_connect('figure_leave_event', leave_figure)
fig2.canvas.mpl_connect('axes_enter_event', enter_axes)
fig2.canvas.mpl_connect('axes_leave_event', leave_axes)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.4 event_handling example code: idle_and_timeout.py

[source code]

```
from __future__ import print_function
"""
Demonstrate/test the idle and timeout API

WARNING: idle_event is deprecated. Use the animations module instead.

This is only tested on gtk so far and is a prototype implementation
"""
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

t = np.arange(0.0, 2.0, 0.01)
y1 = np.sin(2*np.pi*t)
y2 = np.cos(2*np.pi*t)
line1, = ax.plot(y1)
line2, = ax.plot(y2)
```

```
N = 100


def on_idle(event):
    on_idle.count += 1
    print('idle', on_idle.count)
    line1.set_ydata(np.sin(2*np.pi*t*(N - on_idle.count)/float(N)))
    event.canvas.draw()
    # test boolean return removal
    if on_idle.count == N:
        return False
    return True
on_idle.cid = None
on_idle.count = 0

fig.canvas.mpl_connect('idle_event', on_idle)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.5 event_handling example code: keypress_demo.py

[source code]

```
#!/usr/bin/env python

"""
Show how to connect to keypress events
"""
from __future__ import print_function
import sys
import numpy as np
import matplotlib.pyplot as plt


def press(event):
    print('press', event.key)
    sys.stdout.flush()
    if event.key == 'x':
        visible = xl.get_visible()
        xl.set_visible(not visible)
        fig.canvas.draw()

fig, ax = plt.subplots()

fig.canvas.mpl_connect('key_press_event', press)

ax.plot(np.random.rand(12), np.random.rand(12), 'go')
xl = ax.set_xlabel('easy come, easy go')
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.6 event_handling example code: lasso_demo.py

[source code]

```
"""
Show how to use a lasso to select a set of points and get the indices
of the selected points.  A callback is used to change the color of the
selected points

This is currently a proof-of-concept implementation (though it is
usable as is).  There will be some refinement of the API.
"""
from matplotlib.widgets import Lasso
from matplotlib.colors import colorConverter
from matplotlib.collections import RegularPolyCollection
from matplotlib import path

import matplotlib.pyplot as plt
from numpy import nonzero
from numpy.random import rand


class Datum(object):
    colorin = colorConverter.to_rgba('red')
    colorout = colorConverter.to_rgba('blue')

    def __init__(self, x, y, include=False):
        self.x = x
        self.y = y
        if include:
            self.color = self.colorin
        else:
            self.color = self.colorout


class LassoManager(object):
    def __init__(self, ax, data):
        self.axes = ax
        self.canvas = ax.figure.canvas
        self.data = data

        self.Nxy = len(data)

        facecolors = [d.color for d in data]
        self.xys = [(d.x, d.y) for d in data]
        fig = ax.figure
        self.collection = RegularPolyCollection(
```

```
            fig.dpi, 6, sizes=(100,),
            facecolors=facecolors,
            offsets=self.xys,
            transOffset=ax.transData)

        ax.add_collection(self.collection)

        self.cid = self.canvas.mpl_connect('button_press_event', self.onpress)

    def callback(self, verts):
        facecolors = self.collection.get_facecolors()
        p = path.Path(verts)
        ind = p.contains_points(self.xys)
        for i in range(len(self.xys)):
            if ind[i]:
                facecolors[i] = Datum.colorin
            else:
                facecolors[i] = Datum.colorout

        self.canvas.draw_idle()
        self.canvas.widgetlock.release(self.lasso)
        del self.lasso

    def onpress(self, event):
        if self.canvas.widgetlock.locked():
            return
        if event.inaxes is None:
            return
        self.lasso = Lasso(event.inaxes,
                           (event.xdata, event.ydata),
                           self.callback)
        # acquire a lock on the widget drawing
        self.canvas.widgetlock(self.lasso)

if __name__ == '__main__':

    data = [Datum(*xy) for xy in rand(100, 2)]

    ax = plt.axes(xlim=(0, 1), ylim=(0, 1), autoscale_on=False)
    lman = LassoManager(ax, data)

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.7 event_handling example code: legend_picking.py

[source code]

```
"""
Enable picking on the legend to toggle the legended line on and off
```

```
"""
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0.0, 0.2, 0.1)
y1 = 2*np.sin(2*np.pi*t)
y2 = 4*np.sin(2*np.pi*2*t)

fig, ax = plt.subplots()
ax.set_title('Click on legend line to toggle line on/off')
line1, = ax.plot(t, y1, lw=2, color='red', label='1 HZ')
line2, = ax.plot(t, y2, lw=2, color='blue', label='2 HZ')
leg = ax.legend(loc='upper left', fancybox=True, shadow=True)
leg.get_frame().set_alpha(0.4)


# we will set up a dict mapping legend line to orig line, and enable
# picking on the legend line
lines = [line1, line2]
lined = dict()
for legline, origline in zip(leg.get_lines(), lines):
    legline.set_picker(5)  # 5 pts tolerance
    lined[legline] = origline


def onpick(event):
    # on the pick event, find the orig line corresponding to the
    # legend proxy line, and toggle the visibility
    legline = event.artist
    origline = lined[legline]
    vis = not origline.get_visible()
    origline.set_visible(vis)
    # Change the alpha on the line in the legend so we can see what lines
    # have been toggled
    if vis:
        legline.set_alpha(1.0)
    else:
        legline.set_alpha(0.2)
    fig.canvas.draw()

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.8 event_handling example code: looking_glass.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
x, y = np.random.rand(2, 200)

fig, ax = plt.subplots()
circ = patches.Circle((0.5, 0.5), 0.25, alpha=0.8, fc='yellow')
ax.add_patch(circ)


ax.plot(x, y, alpha=0.2)
line, = ax.plot(x, y, alpha=1.0, clip_path=circ)


class EventHandler(object):
    def __init__(self):
        fig.canvas.mpl_connect('button_press_event', self.onpress)
        fig.canvas.mpl_connect('button_release_event', self.onrelease)
        fig.canvas.mpl_connect('motion_notify_event', self.onmove)
        self.x0, self.y0 = circ.center
        self.pressevent = None

    def onpress(self, event):
        if event.inaxes != ax:
            return

        if not circ.contains(event)[0]:
            return

        self.pressevent = event

    def onrelease(self, event):
        self.pressevent = None
        self.x0, self.y0 = circ.center

    def onmove(self, event):
        if self.pressevent is None or event.inaxes != self.pressevent.inaxes:
            return

        dx = event.xdata - self.pressevent.xdata
        dy = event.ydata - self.pressevent.ydata
        circ.center = self.x0 + dx, self.y0 + dy
        line.set_clip_path(circ)
        fig.canvas.draw()

handler = EventHandler()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.9 event_handling example code: path_editor.py

[source code]

```
import numpy as np
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

Path = mpath.Path

fig, ax = plt.subplots()

pathdata = [
    (Path.MOVETO, (1.58, -2.57)),
    (Path.CURVE4, (0.35, -1.1)),
    (Path.CURVE4, (-1.75, 2.0)),
    (Path.CURVE4, (0.375, 2.0)),
    (Path.LINETO, (0.85, 1.15)),
    (Path.CURVE4, (2.2, 3.2)),
    (Path.CURVE4, (3, 0.05)),
    (Path.CURVE4, (2.0, -0.5)),
    (Path.CLOSEPOLY, (1.58, -2.57)),
    ]

codes, verts = zip(*pathdata)
path = mpath.Path(verts, codes)
patch = mpatches.PathPatch(path, facecolor='green', edgecolor='yellow', alpha=0.5)
ax.add_patch(patch)


class PathInteractor(object):
    """
    An path editor.

    Key-bindings

      't' toggle vertex markers on and off.  When vertex markers are on,
          you can move them, delete them


    """

    showverts = True
    epsilon = 5  # max pixel distance to count as a vertex hit

    def __init__(self, pathpatch):

        self.ax = pathpatch.axes
        canvas = self.ax.figure.canvas
        self.pathpatch = pathpatch
        self.pathpatch.set_animated(True)
```

```
        x, y = zip(*self.pathpatch.get_path().vertices)

        self.line, = ax.plot(x, y, marker='o', markerfacecolor='r', animated=True)

        self._ind = None  # the active vert

        canvas.mpl_connect('draw_event', self.draw_callback)
        canvas.mpl_connect('button_press_event', self.button_press_callback)
        canvas.mpl_connect('key_press_event', self.key_press_callback)
        canvas.mpl_connect('button_release_event', self.button_release_callback)
        canvas.mpl_connect('motion_notify_event', self.motion_notify_callback)
        self.canvas = canvas

    def draw_callback(self, event):
        self.background = self.canvas.copy_from_bbox(self.ax.bbox)
        self.ax.draw_artist(self.pathpatch)
        self.ax.draw_artist(self.line)
        self.canvas.blit(self.ax.bbox)

    def pathpatch_changed(self, pathpatch):
        'this method is called whenever the pathpatchgon object is called'
        # only copy the artist props to the line (except visibility)
        vis = self.line.get_visible()
        plt.Artist.update_from(self.line, pathpatch)
        self.line.set_visible(vis)  # don't use the pathpatch visibility state

    def get_ind_under_point(self, event):
        'get the index of the vertex under point if within epsilon tolerance'

        # display coords
        xy = np.asarray(self.pathpatch.get_path().vertices)
        xyt = self.pathpatch.get_transform().transform(xy)
        xt, yt = xyt[:, 0], xyt[:, 1]
        d = np.sqrt((xt - event.x)**2 + (yt - event.y)**2)
        ind = d.argmin()

        if d[ind] >= self.epsilon:
            ind = None

        return ind

    def button_press_callback(self, event):
        'whenever a mouse button is pressed'
        if not self.showverts:
            return
        if event.inaxes is None:
            return
        if event.button != 1:
            return
        self._ind = self.get_ind_under_point(event)

    def button_release_callback(self, event):
        'whenever a mouse button is released'
```

```
        if not self.showverts:
            return
        if event.button != 1:
            return
        self._ind = None

    def key_press_callback(self, event):
        'whenever a key is pressed'
        if not event.inaxes:
            return
        if event.key == 't':
            self.showverts = not self.showverts
            self.line.set_visible(self.showverts)
            if not self.showverts:
                self._ind = None

        self.canvas.draw()

    def motion_notify_callback(self, event):
        'on mouse movement'
        if not self.showverts:
            return
        if self._ind is None:
            return
        if event.inaxes is None:
            return
        if event.button != 1:
            return
        x, y = event.xdata, event.ydata

        vertices = self.pathpatch.get_path().vertices

        vertices[self._ind] = x, y
        self.line.set_data(zip(*vertices))

        self.canvas.restore_region(self.background)
        self.ax.draw_artist(self.pathpatch)
        self.ax.draw_artist(self.line)
        self.canvas.blit(self.ax.bbox)


interactor = PathInteractor(patch)
ax.set_title('drag vertices to update path')
ax.set_xlim(-3, 4)
ax.set_ylim(-3, 4)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.10 event_handling example code: pick_event_demo.py

[source code]

```
#!/usr/bin/env python

"""

You can enable picking by setting the "picker" property of an artist
(for example, a matplotlib Line2D, Text, Patch, Polygon, AxesImage,
etc...)

There are a variety of meanings of the picker property

    None -  picking is disabled for this artist (default)

    boolean - if True then picking will be enabled and the
      artist will fire a pick event if the mouse event is over
      the artist

    float - if picker is a number it is interpreted as an
      epsilon tolerance in points and the artist will fire
      off an event if it's data is within epsilon of the mouse
      event.  For some artists like lines and patch collections,
      the artist may provide additional data to the pick event
      that is generated, for example, the indices of the data within
      epsilon of the pick event

    function - if picker is callable, it is a user supplied
      function which determines whether the artist is hit by the
      mouse event.

         hit, props = picker(artist, mouseevent)

      to determine the hit test.  If the mouse event is over the
      artist, return hit=True and props is a dictionary of properties
      you want added to the PickEvent attributes


After you have enabled an artist for picking by setting the "picker"
property, you need to connect to the figure canvas pick_event to get
pick callbacks on mouse press events.  For example,

  def pick_handler(event):
      mouseevent = event.mouseevent
      artist = event.artist
      # now do something with this...


The pick event (matplotlib.backend_bases.PickEvent) which is passed to
your callback is always fired with two attributes:

  mouseevent - the mouse event that generate the pick event.  The
```

```
    mouse event in turn has attributes like x and y (the coordinates in
    display space, such as pixels from left, bottom) and xdata, ydata (the
    coords in data space).  Additionally, you can get information about
    which buttons were pressed, which keys were pressed, which Axes
    the mouse is over, etc.  See matplotlib.backend_bases.MouseEvent
    for details.

  artist - the matplotlib.artist that generated the pick event.

Additionally, certain artists like Line2D and PatchCollection may
attach additional meta data like the indices into the data that meet
the picker criteria (for example, all the points in the line that are within
the specified epsilon tolerance)

The examples below illustrate each of these methods.
"""

from __future__ import print_function
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.patches import Rectangle
from matplotlib.text import Text
from matplotlib.image import AxesImage
import numpy as np
from numpy.random import rand

if 1:  # simple picking, lines, rectangles and text
    fig, (ax1, ax2) = plt.subplots(2, 1)
    ax1.set_title('click on points, rectangles or text', picker=True)
    ax1.set_ylabel('ylabel', picker=True, bbox=dict(facecolor='red'))
    line, = ax1.plot(rand(100), 'o', picker=5)  # 5 points tolerance

    # pick the rectangle
    bars = ax2.bar(range(10), rand(10), picker=True)
    for label in ax2.get_xticklabels():  # make the xtick labels pickable
        label.set_picker(True)

    def onpick1(event):
        if isinstance(event.artist, Line2D):
            thisline = event.artist
            xdata = thisline.get_xdata()
            ydata = thisline.get_ydata()
            ind = event.ind
            print('onpick1 line:', zip(np.take(xdata, ind), np.take(ydata, ind)))
        elif isinstance(event.artist, Rectangle):
            patch = event.artist
            print('onpick1 patch:', patch.get_path())
        elif isinstance(event.artist, Text):
            text = event.artist
            print('onpick1 text:', text.get_text())

    fig.canvas.mpl_connect('pick_event', onpick1)
```

```
if 1:  # picking with a custom hit test function
    # you can define custom pickers by setting picker to a callable
    # function.  The function has the signature
    #
    #  hit, props = func(artist, mouseevent)
    #
    # to determine the hit test.  if the mouse event is over the artist,
    # return hit=True and props is a dictionary of
    # properties you want added to the PickEvent attributes

    def line_picker(line, mouseevent):
        """
        find the points within a certain distance from the mouseclick in
        data coords and attach some extra attributes, pickx and picky
        which are the data points that were picked
        """
        if mouseevent.xdata is None:
            return False, dict()
        xdata = line.get_xdata()
        ydata = line.get_ydata()
        maxd = 0.05
        d = np.sqrt((xdata - mouseevent.xdata)**2. + (ydata - mouseevent.ydata)**2.)

        ind = np.nonzero(np.less_equal(d, maxd))
        if len(ind):
            pickx = np.take(xdata, ind)
            picky = np.take(ydata, ind)
            props = dict(ind=ind, pickx=pickx, picky=picky)
            return True, props
        else:
            return False, dict()

    def onpick2(event):
        print('onpick2 line:', event.pickx, event.picky)

    fig, ax = plt.subplots()
    ax.set_title('custom picker for line data')
    line, = ax.plot(rand(100), rand(100), 'o', picker=line_picker)
    fig.canvas.mpl_connect('pick_event', onpick2)


if 1:  # picking on a scatter plot (matplotlib.collections.RegularPolyCollection)

    x, y, c, s = rand(4, 100)

    def onpick3(event):
        ind = event.ind
        print('onpick3 scatter:', ind, np.take(x, ind), np.take(y, ind))

    fig, ax = plt.subplots()
    col = ax.scatter(x, y, 100*s, c, picker=True)
    #fig.savefig('pscoll.eps')
    fig.canvas.mpl_connect('pick_event', onpick3)
```

```
if 1:  # picking images (matplotlib.image.AxesImage)
    fig, ax = plt.subplots()
    im1 = ax.imshow(rand(10, 5), extent=(1, 2, 1, 2), picker=True)
    im2 = ax.imshow(rand(5, 10), extent=(3, 4, 1, 2), picker=True)
    im3 = ax.imshow(rand(20, 25), extent=(1, 2, 3, 4), picker=True)
    im4 = ax.imshow(rand(30, 12), extent=(3, 4, 3, 4), picker=True)
    ax.axis([0, 5, 0, 5])

    def onpick4(event):
        artist = event.artist
        if isinstance(artist, AxesImage):
            im = artist
            A = im.get_array()
            print('onpick4 image', A.shape)

    fig.canvas.mpl_connect('pick_event', onpick4)


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.11 event_handling example code: pick_event_demo2.py

[source code]

```
"""
compute the mean and standard deviation (stddev) of 100 data sets and plot
mean vs stddev.  When you click on one of the mu, sigma points, plot the raw
data from the dataset that generated the mean and stddev.
"""
import numpy
import matplotlib.pyplot as plt


X = numpy.random.rand(100, 1000)
xs = numpy.mean(X, axis=1)
ys = numpy.std(X, axis=1)

fig, ax = plt.subplots()
ax.set_title('click on point to plot time series')
line, = ax.plot(xs, ys, 'o', picker=5)  # 5 points tolerance


def onpick(event):

    if event.artist != line:
        return True

    N = len(event.ind)
```

```
    if not N:
        return True

    figi = plt.figure()
    for subplotnum, dataind in enumerate(event.ind):
        ax = figi.add_subplot(N, 1, subplotnum + 1)
        ax.plot(X[dataind])
        ax.text(0.05, 0.9, 'mu=%1.3f\nsigma=%1.3f' % (xs[dataind], ys[dataind]),
                transform=ax.transAxes, va='top')
        ax.set_ylim(-0.5, 1.5)
    figi.show()
    return True

fig.canvas.mpl_connect('pick_event', onpick)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.12 event_handling example code: pipong.py

[source code]

```
#!/usr/bin/env python
# A matplotlib based game of Pong illustrating one way to write interactive
# animation which are easily ported to multiple backends
# pipong.py was written by Paul Ivanov <http://pirsquared.org>

from __future__ import print_function

import numpy as np
import matplotlib.pyplot as plt
from numpy.random import randn, randint

instructions = """
Player A:       Player B:
  'e'      up     'i'
  'd'     down    'k'

press 't' -- close these instructions
            (animation will be much faster)
press 'a' -- add a puck
press 'A' -- remove a puck
press '1' -- slow down all pucks
press '2' -- speed up all pucks
press '3' -- slow down distractors
press '4' -- speed up distractors
press ' ' -- reset the first puck
press 'n' -- toggle distractors on/off
press 'g' -- toggle the game on/off
```

```
"""


class Pad(object):
    def __init__(self, disp, x, y, type='l'):
        self.disp = disp
        self.x = x
        self.y = y
        self.w = .3
        self.score = 0
        self.xoffset = 0.3
        self.yoffset = 0.1
        if type == 'r':
            self.xoffset *= -1.0

        if type == 'l' or type == 'r':
            self.signx = -1.0
            self.signy = 1.0
        else:
            self.signx = 1.0
            self.signy = -1.0

    def contains(self, loc):
        return self.disp.get_bbox().contains(loc.x, loc.y)


class Puck(object):
    def __init__(self, disp, pad, field):
        self.vmax = .2
        self.disp = disp
        self.field = field
        self._reset(pad)

    def _reset(self, pad):
        self.x = pad.x + pad.xoffset
        if pad.y < 0:
            self.y = pad.y + pad.yoffset
        else:
            self.y = pad.y - pad.yoffset
        self.vx = pad.x - self.x
        self.vy = pad.y + pad.w/2 - self.y
        self._speedlimit()
        self._slower()
        self._slower()

    def update(self, pads):
        self.x += self.vx
        self.y += self.vy
        for pad in pads:
            if pad.contains(self):
                self.vx *= 1.2 * pad.signx
                self.vy *= 1.2 * pad.signy
        fudge = .001
```

```
        # probably cleaner with something like...
        #if not self.field.contains(self.x, self.y):
        if self.x < fudge:
            #print("player A loses")
            pads[1].score += 1
            self._reset(pads[0])
            return True
        if self.x > 7 - fudge:
            #print("player B loses")
            pads[0].score += 1
            self._reset(pads[1])
            return True
        if self.y < -1 + fudge or self.y > 1 - fudge:
            self.vy *= -1.0
            # add some randomness, just to make it interesting
            self.vy -= (randn()/300.0 + 1/300.0) * np.sign(self.vy)
        self._speedlimit()
        return False

    def _slower(self):
        self.vx /= 5.0
        self.vy /= 5.0

    def _faster(self):
        self.vx *= 5.0
        self.vy *= 5.0

    def _speedlimit(self):
        if self.vx > self.vmax:
            self.vx = self.vmax
        if self.vx < -self.vmax:
            self.vx = -self.vmax

        if self.vy > self.vmax:
            self.vy = self.vmax
        if self.vy < -self.vmax:
            self.vy = -self.vmax


class Game(object):
    def __init__(self, ax):
        # create the initial line
        self.ax = ax
        padAx = padBx = .50
        padAy = padBy = .30
        padBx += 6.3
        pA, = self.ax.barh(padAy, .2, height=.3, color='k', alpha=.5, edgecolor='b',␣
↪lw=2, label="Player B", animated=True)
        pB, = self.ax.barh(padBy, .2, height=.3, left=padBx, color='k', alpha=.5,␣
↪edgecolor='r', lw=2, label="Player A", animated=True)

        # distractors
        self.x = np.arange(0, 2.22*np.pi, 0.01)
```

```
        self.line, = self.ax.plot(self.x, np.sin(self.x), "r", animated=True, lw=4)
        self.line2, = self.ax.plot(self.x, np.cos(self.x), "g", animated=True, lw=4)
        self.line3, = self.ax.plot(self.x, np.cos(self.x), "g", animated=True, lw=4)
        self.line4, = self.ax.plot(self.x, np.cos(self.x), "r", animated=True, lw=4)
        self.centerline, = self.ax.plot([3.5, 3.5], [1, -1], 'k', alpha=.5,␣
↪animated=True, lw=8)
        self.puckdisp = self.ax.scatter([1], [1], label='_nolegend_', s=200, c='g',␣
↪alpha=.9, animated=True)

        self.canvas = self.ax.figure.canvas
        self.background = None
        self.cnt = 0
        self.distract = True
        self.res = 100.0
        self.on = False
        self.inst = True    # show instructions from the beginning
        self.background = None
        self.pads = []
        self.pads.append(Pad(pA, 0, padAy))
        self.pads.append(Pad(pB, padBx, padBy, 'r'))
        self.pucks = []
        self.i = self.ax.annotate(instructions, (.5, 0.5),
                                  name='monospace',
                                  verticalalignment='center',
                                  horizontalalignment='center',
                                  multialignment='left',
                                  textcoords='axes fraction', animated=True)
        self.canvas.mpl_connect('key_press_event', self.key_press)

    def draw(self, evt):
        draw_artist = self.ax.draw_artist
        if self.background is None:
            self.background = self.canvas.copy_from_bbox(self.ax.bbox)

        # restore the clean slate background
        self.canvas.restore_region(self.background)

        # show the distractors
        if self.distract:
            self.line.set_ydata(np.sin(self.x + self.cnt/self.res))
            self.line2.set_ydata(np.cos(self.x - self.cnt/self.res))
            self.line3.set_ydata(np.tan(self.x + self.cnt/self.res))
            self.line4.set_ydata(np.tan(self.x - self.cnt/self.res))
            draw_artist(self.line)
            draw_artist(self.line2)
            draw_artist(self.line3)
            draw_artist(self.line4)

        # show the instructions - this is very slow
        if self.inst:
            self.ax.draw_artist(self.i)

        # pucks and pads
```

```
        if self.on:
            self.ax.draw_artist(self.centerline)
            for pad in self.pads:
                pad.disp.set_y(pad.y)
                pad.disp.set_x(pad.x)
                self.ax.draw_artist(pad.disp)

            for puck in self.pucks:
                if puck.update(self.pads):
                    # we only get here if someone scored
                    self.pads[0].disp.set_label("   " + str(self.pads[0].score))
                    self.pads[1].disp.set_label("   " + str(self.pads[1].score))
                    self.ax.legend(loc='center')
                    self.leg = self.ax.get_legend()
                    #self.leg.draw_frame(False) #don't draw the legend border
                    self.leg.get_frame().set_alpha(.2)
                    plt.setp(self.leg.get_texts(), fontweight='bold', fontsize='xx-large
↪')
                    self.leg.get_frame().set_facecolor('0.2')
                    self.background = None
                    self.ax.figure.canvas.draw()
                    return True
                puck.disp.set_offsets([puck.x, puck.y])
                self.ax.draw_artist(puck.disp)

        # just redraw the axes rectangle
        self.canvas.blit(self.ax.bbox)

        if self.cnt == 50000:
            # just so we don't get carried away
            print("...and you've been playing for too long!!!")
            plt.close()

        self.cnt += 1
        return True

    def key_press(self, event):
        if event.key == '3':
            self.res *= 5.0
        if event.key == '4':
            self.res /= 5.0

        if event.key == 'e':
            self.pads[0].y += .1
            if self.pads[0].y > 1 - .3:
                self.pads[0].y = 1 - .3
        if event.key == 'd':
            self.pads[0].y -= .1
            if self.pads[0].y < -1:
                self.pads[0].y = -1

        if event.key == 'i':
            self.pads[1].y += .1
```

```
            if self.pads[1].y > 1 - .3:
                self.pads[1].y = 1 - .3
        if event.key == 'k':
            self.pads[1].y -= .1
            if self.pads[1].y < -1:
                self.pads[1].y = -1

        if event.key == 'a':
            self.pucks.append(Puck(self.puckdisp, self.pads[randint(2)], self.ax.bbox))
        if event.key == 'A' and len(self.pucks):
            self.pucks.pop()
        if event.key == ' ' and len(self.pucks):
            self.pucks[0]._reset(self.pads[randint(2)])
        if event.key == '1':
            for p in self.pucks:
                p._slower()
        if event.key == '2':
            for p in self.pucks:
                p._faster()

        if event.key == 'n':
            self.distract = not self.distract

        if event.key == 'g':
            #self.ax.clear()
            self.on = not self.on
        if event.key == 't':
            self.inst = not self.inst
            self.i.set_visible(self.i.get_visible())
        if event.key == 'q':
            plt.close()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.13 event_handling example code: poly_editor.py

[source code]

```
"""
This is an example to show how to build cross-GUI applications using
matplotlib event handling to interact with objects on the canvas

"""
import numpy as np
from matplotlib.lines import Line2D
from matplotlib.artist import Artist
from matplotlib.mlab import dist_point_to_segment


class PolygonInteractor(object):
    """
```

```
    An polygon editor.

    Key-bindings

      't' toggle vertex markers on and off.  When vertex markers are on,
          you can move them, delete them

      'd' delete the vertex under point

      'i' insert a vertex at point.  You must be within epsilon of the
          line connecting two existing vertices

    """

    showverts = True
    epsilon = 5  # max pixel distance to count as a vertex hit

    def __init__(self, ax, poly):
        if poly.figure is None:
            raise RuntimeError('You must first add the polygon to a figure or canvas␣
↪before defining the interactor')
        self.ax = ax
        canvas = poly.figure.canvas
        self.poly = poly

        x, y = zip(*self.poly.xy)
        self.line = Line2D(x, y, marker='o', markerfacecolor='r', animated=True)
        self.ax.add_line(self.line)
        #self._update_line(poly)

        cid = self.poly.add_callback(self.poly_changed)
        self._ind = None  # the active vert

        canvas.mpl_connect('draw_event', self.draw_callback)
        canvas.mpl_connect('button_press_event', self.button_press_callback)
        canvas.mpl_connect('key_press_event', self.key_press_callback)
        canvas.mpl_connect('button_release_event', self.button_release_callback)
        canvas.mpl_connect('motion_notify_event', self.motion_notify_callback)
        self.canvas = canvas

    def draw_callback(self, event):
        self.background = self.canvas.copy_from_bbox(self.ax.bbox)
        self.ax.draw_artist(self.poly)
        self.ax.draw_artist(self.line)
        self.canvas.blit(self.ax.bbox)

    def poly_changed(self, poly):
        'this method is called whenever the polygon object is called'
        # only copy the artist props to the line (except visibility)
        vis = self.line.get_visible()
        Artist.update_from(self.line, poly)
        self.line.set_visible(vis)  # don't use the poly visibility state
```

```
    def get_ind_under_point(self, event):
        'get the index of the vertex under point if within epsilon tolerance'

        # display coords
        xy = np.asarray(self.poly.xy)
        xyt = self.poly.get_transform().transform(xy)
        xt, yt = xyt[:, 0], xyt[:, 1]
        d = np.sqrt((xt - event.x)**2 + (yt - event.y)**2)
        indseq = np.nonzero(np.equal(d, np.amin(d)))[0]
        ind = indseq[0]

        if d[ind] >= self.epsilon:
            ind = None

        return ind

    def button_press_callback(self, event):
        'whenever a mouse button is pressed'
        if not self.showverts:
            return
        if event.inaxes is None:
            return
        if event.button != 1:
            return
        self._ind = self.get_ind_under_point(event)

    def button_release_callback(self, event):
        'whenever a mouse button is released'
        if not self.showverts:
            return
        if event.button != 1:
            return
        self._ind = None

    def key_press_callback(self, event):
        'whenever a key is pressed'
        if not event.inaxes:
            return
        if event.key == 't':
            self.showverts = not self.showverts
            self.line.set_visible(self.showverts)
            if not self.showverts:
                self._ind = None
        elif event.key == 'd':
            ind = self.get_ind_under_point(event)
            if ind is not None:
                self.poly.xy = [tup for i, tup in enumerate(self.poly.xy) if i != ind]
                self.line.set_data(zip(*self.poly.xy))
        elif event.key == 'i':
            xys = self.poly.get_transform().transform(self.poly.xy)
            p = event.x, event.y  # display coords
            for i in range(len(xys) - 1):
                s0 = xys[i]
```

```
                s1 = xys[i + 1]
                d = dist_point_to_segment(p, s0, s1)
                if d <= self.epsilon:
                    self.poly.xy = np.array(
                        list(self.poly.xy[:i]) +
                        [(event.xdata, event.ydata)] +
                        list(self.poly.xy[i:]))
                    self.line.set_data(zip(*self.poly.xy))
                    break

        self.canvas.draw()

    def motion_notify_callback(self, event):
        'on mouse movement'
        if not self.showverts:
            return
        if self._ind is None:
            return
        if event.inaxes is None:
            return
        if event.button != 1:
            return
        x, y = event.xdata, event.ydata

        self.poly.xy[self._ind] = x, y
        self.line.set_data(zip(*self.poly.xy))

        self.canvas.restore_region(self.background)
        self.ax.draw_artist(self.poly)
        self.ax.draw_artist(self.line)
        self.canvas.blit(self.ax.bbox)


if __name__ == '__main__':
    import matplotlib.pyplot as plt
    from matplotlib.patches import Polygon

    theta = np.arange(0, 2*np.pi, 0.1)
    r = 1.5

    xs = r*np.cos(theta)
    ys = r*np.sin(theta)

    poly = Polygon(list(zip(xs, ys)), animated=True)

    fig, ax = plt.subplots()
    ax.add_patch(poly)
    p = PolygonInteractor(ax, poly)

    #ax.add_line(p.line)
    ax.set_title('Click and drag a point to move it')
    ax.set_xlim((-2, 2))
    ax.set_ylim((-2, 2))
```

```
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.14 event_handling example code: pong_gtk.py

[source code]

```
#!/usr/bin/env python

from __future__ import print_function

# For detailed comments on animation and the techniques used here, see
# the wiki entry
# http://www.scipy.org/wikis/topical_software/MatplotlibAnimation
import time

import gobject

import matplotlib
matplotlib.use('GTKAgg')

import matplotlib.pyplot as plt
import pipong


fig, ax = plt.subplots()
canvas = ax.figure.canvas


def start_anim(event):
    # gobject.idle_add(animation.draw,animation)
    gobject.timeout_add(10, animation.draw, animation)
    canvas.mpl_disconnect(start_anim.cid)

animation = pipong.Game(ax)
start_anim.cid = canvas.mpl_connect('draw_event', start_anim)


tstart = time.time()
plt.grid()  # to ensure proper background restore
plt.show()
print('FPS: %f' % animation.cnt/(time.time() - tstart))
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.15 event_handling example code: resample.py

[source code]

```python
import numpy as np
import matplotlib.pyplot as plt
from scikits.audiolab import wavread


# A class that will downsample the data and recompute when zoomed.
class DataDisplayDownsampler(object):
    def __init__(self, xdata, ydata):
        self.origYData = ydata
        self.origXData = xdata
        self.numpts = 3000
        self.delta = xdata[-1] - xdata[0]

    def resample(self, xstart, xend):
        # Very simple downsampling that takes the points within the range
        # and picks every Nth point
        mask = (self.origXData > xstart) & (self.origXData < xend)
        xdata = self.origXData[mask]
        ratio = int(xdata.size / self.numpts) + 1
        xdata = xdata[::ratio]

        ydata = self.origYData[mask]
        ydata = ydata[::ratio]

        return xdata, ydata

    def update(self, ax):
        # Update the line
        lims = ax.viewLim
        if np.abs(lims.width - self.delta) > 1e-8:
            self.delta = lims.width
            xstart, xend = lims.intervalx
            self.line.set_data(*self.downsample(xstart, xend))
            ax.figure.canvas.draw_idle()

# Read data
data = wavread('/usr/share/sounds/purple/receive.wav')[0]
ydata = np.tile(data[:, 0], 100)
xdata = np.arange(ydata.size)

d = DataDisplayDownsampler(xdata, ydata)

fig, ax = plt.subplots()

# Hook up the line
xdata, ydata = d.downsample(xdata[0], xdata[-1])
d.line, = ax.plot(xdata, ydata)
ax.set_autoscale_on(False)  # Otherwise, infinite loop

# Connect for changing the view limits
ax.callbacks.connect('xlim_changed', d.update)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.16 event_handling example code: test_mouseclicks.py

[source code]

```
#!/usr/bin/env python
from __future__ import print_function

import matplotlib
#matplotlib.use("WxAgg")
#matplotlib.use("TkAgg")
#matplotlib.use("GTKAgg")
#matplotlib.use("Qt4Agg")
#matplotlib.use("CocoaAgg")
#matplotlib.use("MacOSX")
import matplotlib.pyplot as plt

#print("***** TESTING WITH BACKEND: %s"%matplotlib.get_backend() + " *****")


def OnClick(event):
    if event.dblclick:
        print("DBLCLICK", event)
    else:
        print("DOWN    ", event)


def OnRelease(event):
    print("UP      ", event)


fig = plt.gcf()
cid_up = fig.canvas.mpl_connect('button_press_event', OnClick)
cid_down = fig.canvas.mpl_connect('button_release_event', OnRelease)

plt.gca().text(0.5, 0.5, "Click on the canvas to test mouse events.",
               ha="center", va="center")

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.17 event_handling example code: timers.py

[source code]

```
# Simple example of using general timer objects. This is used to update
# the time placed in the title of the figure.
import matplotlib.pyplot as plt
```

```
import numpy as np
from datetime import datetime


def update_title(axes):
    axes.set_title(datetime.now())
    axes.figure.canvas.draw()

fig, ax = plt.subplots()

x = np.linspace(-3, 3)
ax.plot(x, x*x)

# Create a new timer object. Set the interval to 100 milliseconds
# (1000 is default) and tell the timer what function should be called.
timer = fig.canvas.new_timer(interval=100)
timer.add_callback(update_title, ax)
timer.start()

# Or could start the timer on first figure draw
#def start_timer(evt):
#    timer.start()
#    fig.canvas.mpl_disconnect(drawid)
#drawid = fig.canvas.mpl_connect('draw_event', start_timer)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.18 event_handling example code: trifinder_event_demo.py

[source code]

```
"""
Example showing the use of a TriFinder object.  As the mouse is moved over the
triangulation, the triangle under the cursor is highlighted and the index of
the triangle is displayed in the plot title.
"""
import matplotlib.pyplot as plt
from matplotlib.tri import Triangulation
from matplotlib.patches import Polygon
import numpy as np
import math


def update_polygon(tri):
    if tri == -1:
        points = [0, 0, 0]
    else:
        points = triangulation.triangles[tri]
    xs = triangulation.x[points]
```

```
    ys = triangulation.y[points]
    polygon.set_xy(zip(xs, ys))


def motion_notify(event):
    if event.inaxes is None:
        tri = -1
    else:
        tri = trifinder(event.xdata, event.ydata)
    update_polygon(tri)
    plt.title('In triangle %i' % tri)
    event.canvas.draw()


# Create a Triangulation.
n_angles = 16
n_radii = 5
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)
angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi / n_angles
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
triangulation = Triangulation(x, y)
xmid = x[triangulation.triangles].mean(axis=1)
ymid = y[triangulation.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triangulation.set_mask(mask)

# Use the triangulation's default TriFinder object.
trifinder = triangulation.get_trifinder()

# Setup plot and callbacks.
plt.subplot(111, aspect='equal')
plt.triplot(triangulation, 'bo-')
polygon = Polygon([[0, 0], [0, 0]], facecolor='y')  # dummy data for xs,ys
update_polygon(-1)
plt.gca().add_patch(polygon)
plt.gcf().canvas.mpl_connect('motion_notify_event', motion_notify)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.19 event_handling example code: viewlims.py

[source code]

```
# Creates two identical panels.  Zooming in on the right panel will show
# a rectangle in the first panel, denoting the zoomed region.
import numpy as np
```

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle


# We just subclass Rectangle so that it can be called with an Axes
# instance, causing the rectangle to update its shape to match the
# bounds of the Axes
class UpdatingRect(Rectangle):
    def __call__(self, ax):
        self.set_bounds(*ax.viewLim.bounds)
        ax.figure.canvas.draw_idle()


# A class that will regenerate a fractal set as we zoom in, so that you
# can actually see the increasing detail.  A box in the left panel will show
# the area to which we are zoomed.
class MandlebrotDisplay(object):
    def __init__(self, h=500, w=500, niter=50, radius=2., power=2):
        self.height = h
        self.width = w
        self.niter = niter
        self.radius = radius
        self.power = power

    def __call__(self, xstart, xend, ystart, yend):
        self.x = np.linspace(xstart, xend, self.width)
        self.y = np.linspace(ystart, yend, self.height).reshape(-1, 1)
        c = self.x + 1.0j * self.y
        threshold_time = np.zeros((self.height, self.width))
        z = np.zeros(threshold_time.shape, dtype=np.complex)
        mask = np.ones(threshold_time.shape, dtype=np.bool)
        for i in range(self.niter):
            z[mask] = z[mask]**self.power + c[mask]
            mask = (np.abs(z) < self.radius)
            threshold_time += mask
        return threshold_time

    def ax_update(self, ax):
        ax.set_autoscale_on(False)  # Otherwise, infinite loop

        # Get the number of points from the number of pixels in the window
        dims = ax.axesPatch.get_window_extent().bounds
        self.width = int(dims[2] + 0.5)
        self.height = int(dims[2] + 0.5)

        # Get the range for the new area
        xstart, ystart, xdelta, ydelta = ax.viewLim.bounds
        xend = xstart + xdelta
        yend = ystart + ydelta

        # Update the image object with our new data and extent
        im = ax.images[-1]
        im.set_data(self.__call__(xstart, xend, ystart, yend))
```

```
        im.set_extent((xstart, xend, ystart, yend))
        ax.figure.canvas.draw_idle()

md = MandlebrotDisplay()
Z = md(-2., 0.5, -1.25, 1.25)

fig1, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(Z, origin='lower', extent=(md.x.min(), md.x.max(), md.y.min(), md.y.max()))
ax2.imshow(Z, origin='lower', extent=(md.x.min(), md.x.max(), md.y.min(), md.y.max()))

rect = UpdatingRect([0, 0], 0, 0, facecolor='None', edgecolor='black')
rect.set_bounds(*ax2.viewLim.bounds)
ax1.add_patch(rect)

# Connect for changing the view limits
ax2.callbacks.connect('xlim_changed', rect)
ax2.callbacks.connect('ylim_changed', rect)

ax2.callbacks.connect('xlim_changed', md.ax_update)
ax2.callbacks.connect('ylim_changed', md.ax_update)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

84.20 event_handling example code: zoom_window.py

[source code]

```
"""
This example shows how to connect events in one window, for example, a mouse
press, to another figure window.

If you click on a point in the first window, the z and y limits of the
second will be adjusted so that the center of the zoom in the second
window will be the x,y coordinates of the clicked point.

Note the diameter of the circles in the scatter are defined in
points**2, so their size is independent of the zoom
"""
from matplotlib.pyplot import figure, show
import numpy
figsrc = figure()
figzoom = figure()

axsrc = figsrc.add_subplot(111, xlim=(0, 1), ylim=(0, 1), autoscale_on=False)
axzoom = figzoom.add_subplot(111, xlim=(0.45, 0.55), ylim=(0.4, .6),
                             autoscale_on=False)
axsrc.set_title('Click to zoom')
axzoom.set_title('zoom window')
x, y, s, c = numpy.random.rand(4, 200)
```

```
s *= 200


axsrc.scatter(x, y, s, c)
axzoom.scatter(x, y, s, c)


def onpress(event):
    if event.button != 1:
        return
    x, y = event.xdata, event.ydata
    axzoom.set_xlim(x - 0.1, x + 0.1)
    axzoom.set_ylim(y - 0.1, y + 0.1)
    figzoom.canvas.draw()

figsrc.canvas.mpl_connect('button_press_event', onpress)
show()
```

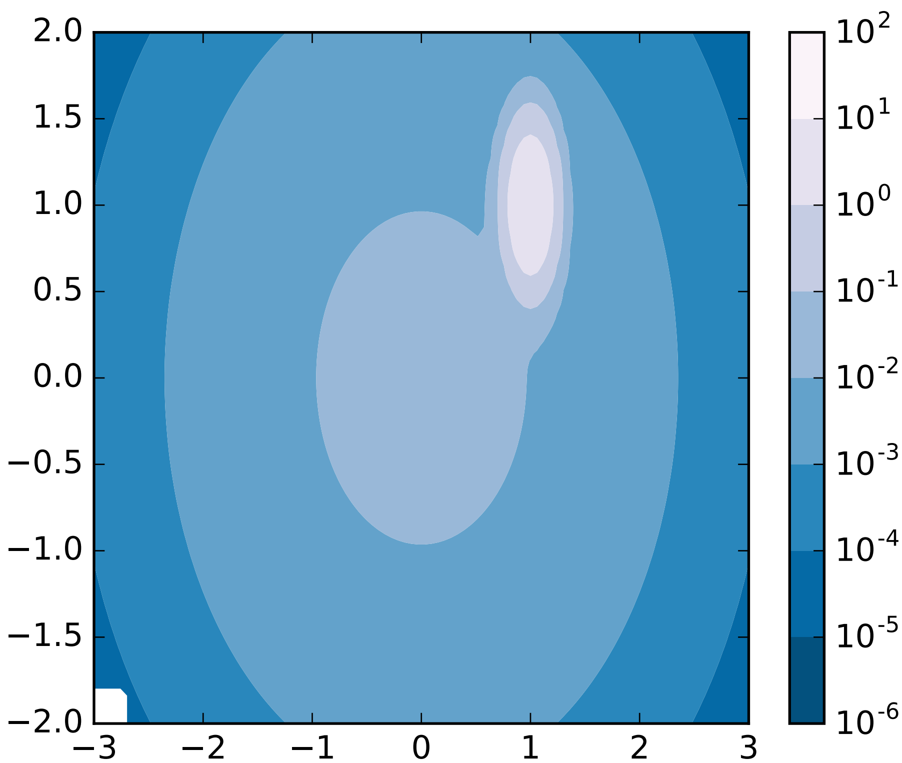
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

EIGHTYFIVE

IMAGES_CONTOURS_AND_FIELDS EXAMPLES

85.1 images_contours_and_fields example code: contourf_log.py




```
'''
Demonstrate use of a log color scale in contourf
'''

import matplotlib.pyplot as plt
import numpy as np
from numpy import ma
from matplotlib import colors, ticker, cm
from matplotlib.mlab import bivariate_normal
```

```
N = 100
x = np.linspace(-3.0, 3.0, N)
y = np.linspace(-2.0, 2.0, N)

X, Y = np.meshgrid(x, y)

# A low hump with a spike coming out of the top right.
# Needs to have z/colour axis on a log scale so we see both hump and spike.
# linear scale only shows the spike.
z = (bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0)
     + 0.1 * bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0))

# Put in some negative values (lower left corner) to cause trouble with logs:
z[:5, :5] = -1

# The following is not strictly essential, but it will eliminate
# a warning.  Comment it out to see the warning.
z = ma.masked_where(z <= 0, z)


# Automatic selection of levels works; setting the
# log locator tells contourf to use a log scale:
cs = plt.contourf(X, Y, z, locator=ticker.LogLocator(), cmap=cm.PuBu_r)

# Alternatively, you can manually set the levels
# and the norm:
#lev_exp = np.arange(np.floor(np.log10(z.min())-1),
#                    np.ceil(np.log10(z.max())+1))
#levs = np.power(10, lev_exp)
#cs = P.contourf(X, Y, z, levs, norm=colors.LogNorm())

# The 'extend' kwarg does not work yet with a log scale.

cbar = plt.colorbar()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

85.2 images_contours_and_fields example code: image_demo.py


```
"""
Simple demo of the imshow function.
"""
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

image_file = cbook.get_sample_data('ada.png')
image = plt.imread(image_file)

plt.imshow(image)
plt.axis('off')  # clear x- and y-axes
plt.show()
```

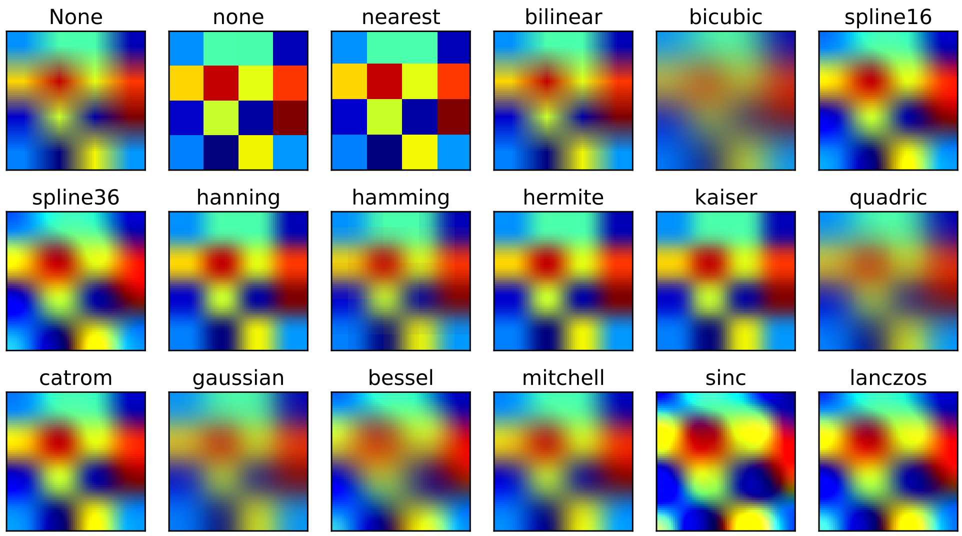
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

85.3 images_contours_and_fields example code: image_demo_clip_path.py

```
"""
Demo of image that's been clipped by a circular patch.
"""
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.cbook as cbook


image_file = cbook.get_sample_data('grace_hopper.png')
image = plt.imread(image_file)

fig, ax = plt.subplots()
im = ax.imshow(image)
patch = patches.Circle((260, 200), radius=200, transform=ax.transData)
im.set_clip_path(patch)

plt.axis('off')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

85.4 images_contours_and_fields example code: interpolation_methods.py

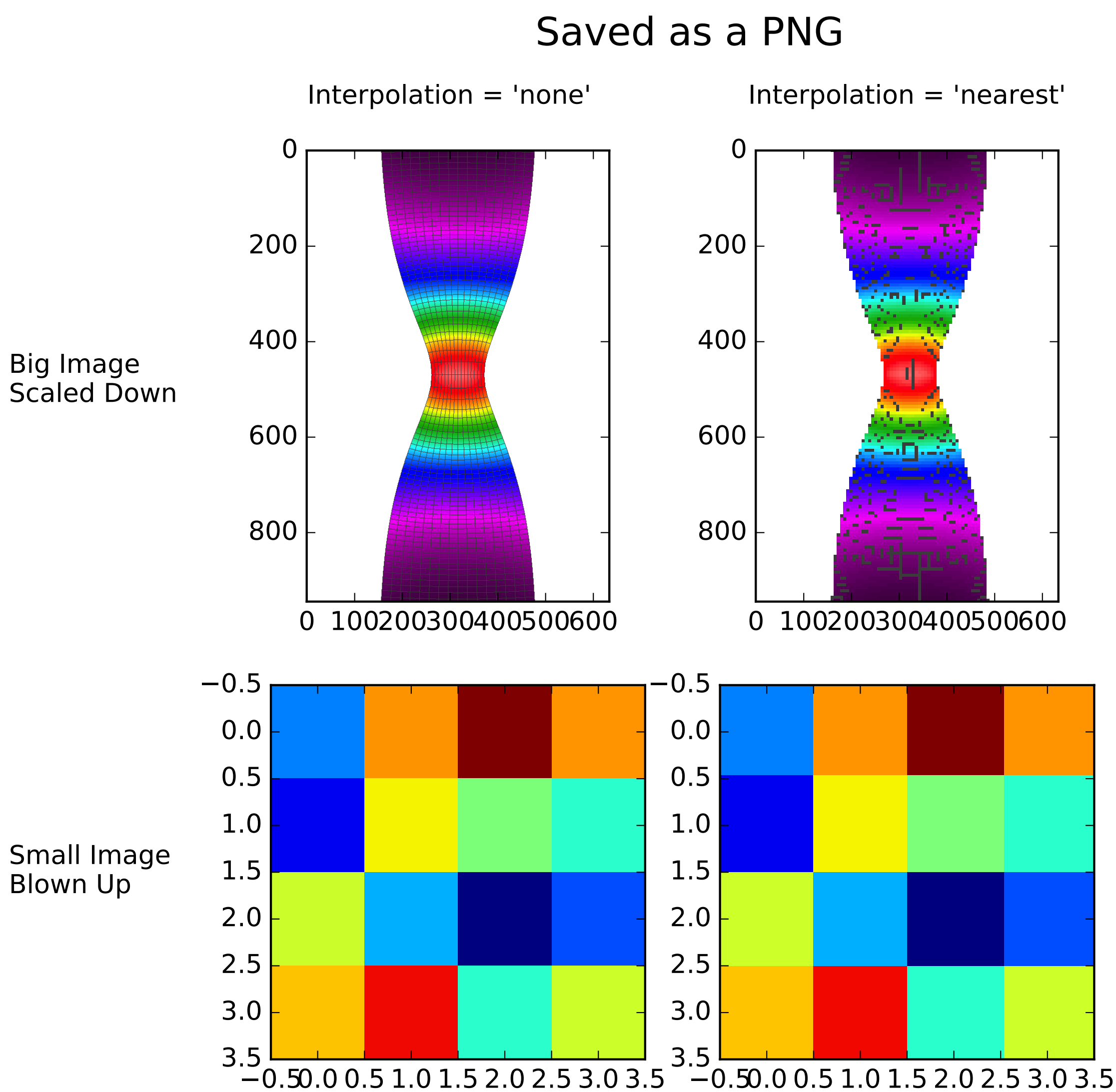


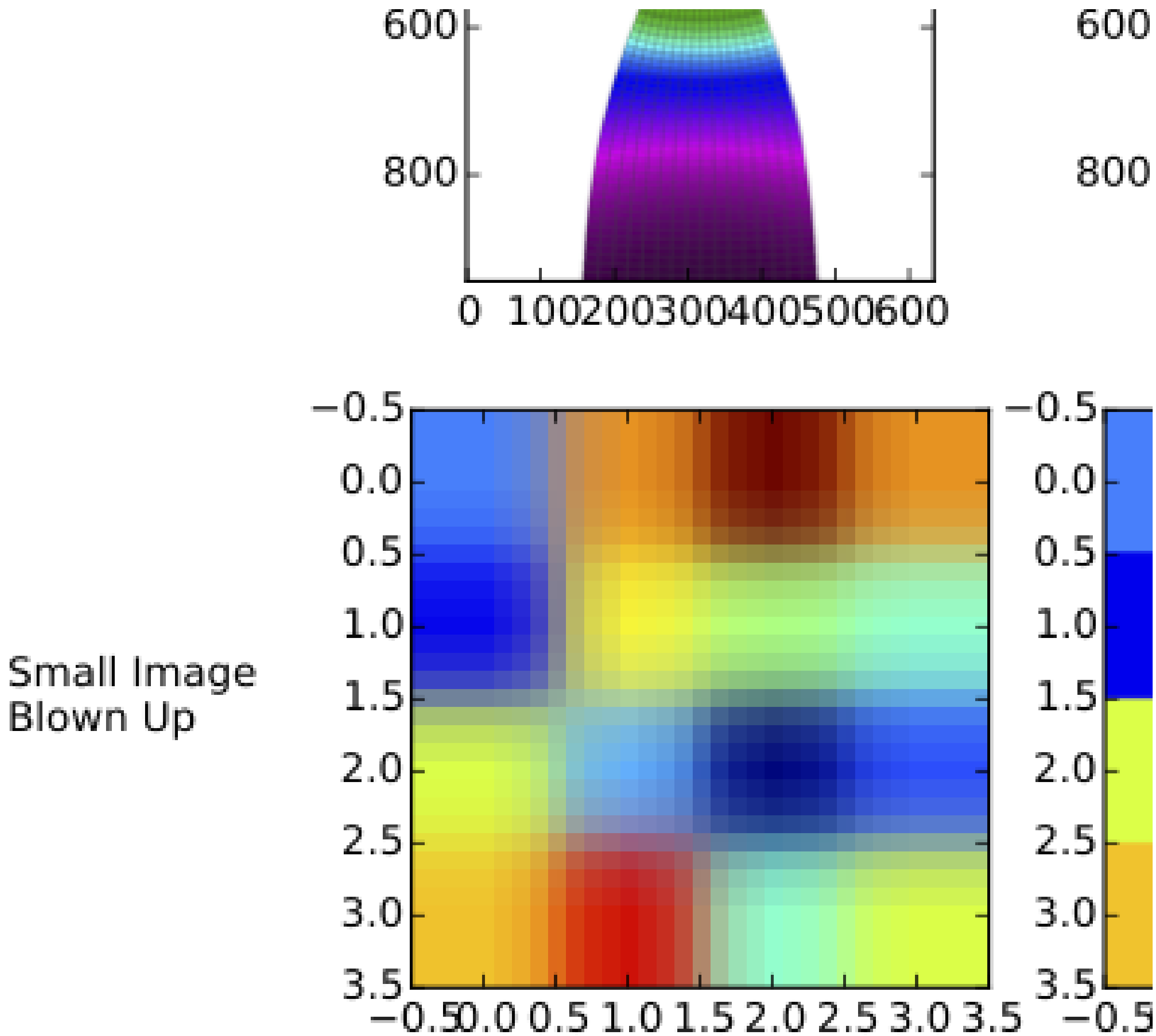

```
'''
Show all different interpolation methods for imshow
'''

import matplotlib.pyplot as plt
import numpy as np

# from the docs:

# If interpolation is None, default to rc image.interpolation. See also
# the filternorm and filterrad parameters. If interpolation is 'none', then
# no interpolation is performed on the Agg, ps and pdf backends. Other
# backends will fall back to 'nearest'.
#
# http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.imshow

methods = [None, 'none', 'nearest', 'bilinear', 'bicubic', 'spline16',
           'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric',
           'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos']

grid = np.random.rand(4, 4)

fig, axes = plt.subplots(3, 6, figsize=(12, 6),
                         subplot_kw={'xticks': [], 'yticks': []})

fig.subplots_adjust(hspace=0.3, wspace=0.05)

for ax, interp_method in zip(axes.flat, methods):
    ax.imshow(grid, interpolation=interp_method)
```

```
    ax.set_title(interp_method)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

85.5 images_contours_and_fields example code: interpolation_none_vs_nearest.py



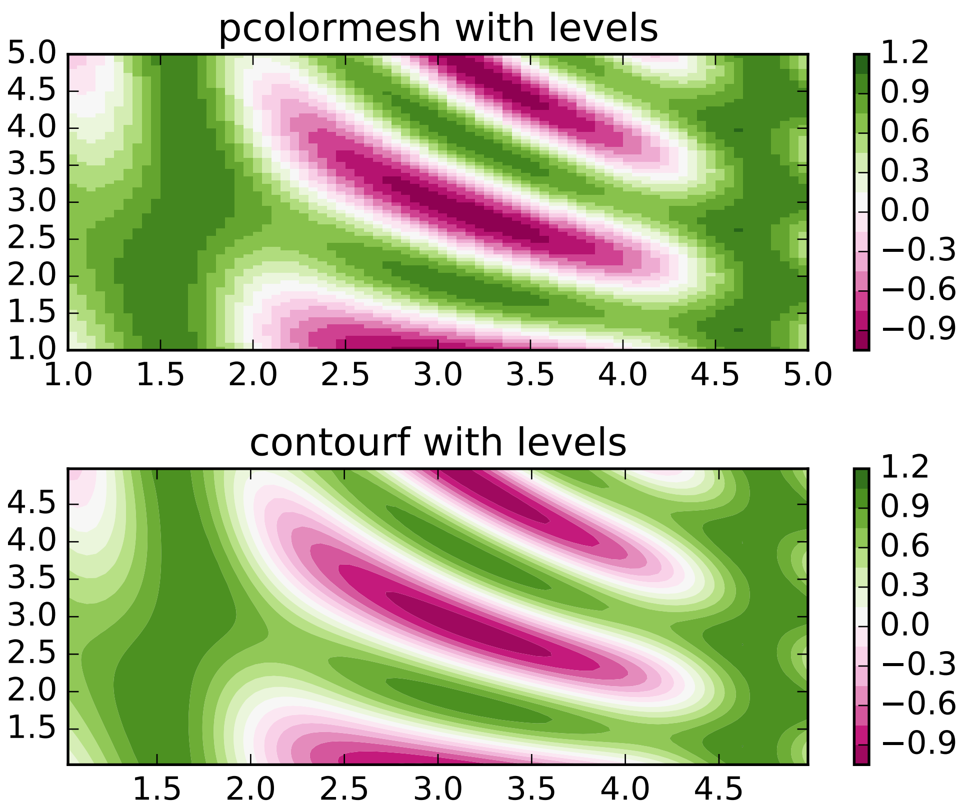



```
"""
Displays the difference between interpolation = 'none' and
interpolation = 'nearest'.

Interpolation = 'none' and interpolation = 'nearest' are equivalent when
converting a figure to an image file, such as a PNG.
Interpolation = 'none' and interpolation = 'nearest' behave quite
differently, however, when converting a figure to a vector graphics file,
such as a PDF.  As shown, Interpolation = 'none' works well when a big
image is scaled down, while interpolation = 'nearest' works well when a
small image is blown up.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
```

```
# Load big image
big_im_path = cbook.get_sample_data('necked_tensile_specimen.png')
big_im = plt.imread(big_im_path)
# Define small image
small_im = np.array([[0.25, 0.75, 1.0, 0.75], [0.1, 0.65, 0.5, 0.4],
                     [0.6, 0.3, 0.0, 0.2], [0.7, 0.9, 0.4, 0.6]])

# Create a 2x2 table of plots
fig = plt.figure(figsize=[8.0, 7.5])
ax = plt.subplot(2, 2, 1)
ax.imshow(big_im, interpolation='none')
ax = plt.subplot(2, 2, 2)
ax.imshow(big_im, interpolation='nearest')
ax = plt.subplot(2, 2, 3)
ax.imshow(small_im, interpolation='none')
ax = plt.subplot(2, 2, 4)
ax.imshow(small_im, interpolation='nearest')
plt.subplots_adjust(left=0.24, wspace=0.2, hspace=0.1,
                    bottom=0.05, top=0.86)

# Label the rows and columns of the table
fig.text(0.03, 0.645, 'Big Image\nScaled Down', ha='left')
fig.text(0.03, 0.225, 'Small Image\nBlown Up', ha='left')
fig.text(0.383, 0.90, "Interpolation = 'none'", ha='center')
fig.text(0.75, 0.90, "Interpolation = 'nearest'", ha='center')

# If you were going to run this example on your local machine, you
# would save the figure as a PNG, save the same figure as a PDF, and
# then compare them.  The following code would suffice.
txt = fig.text(0.452, 0.95, 'Saved as a PNG', fontsize=18)
# plt.savefig('None_vs_nearest-png.png')
# txt.set_text('Saved as a PDF')
# plt.savefig('None_vs_nearest-pdf.pdf')

# Here, however, we need to display the PDF on a webpage, which means
# the PDF must be converted into an image.  For the purposes of this
# example, the 'Nearest_vs_none-pdf.pdf' has been pre-converted into
#'Nearest_vs_none-pdf.png' at 80 dpi.  We simply need to load and
# display it.
pdf_im_path = cbook.get_sample_data('None_vs_nearest-pdf.png')
pdf_im = plt.imread(pdf_im_path)
fig2 = plt.figure(figsize=[8.0, 7.5])
plt.figimage(pdf_im)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

85.6 images_contours_and_fields example code: pcolormesh_levels.py

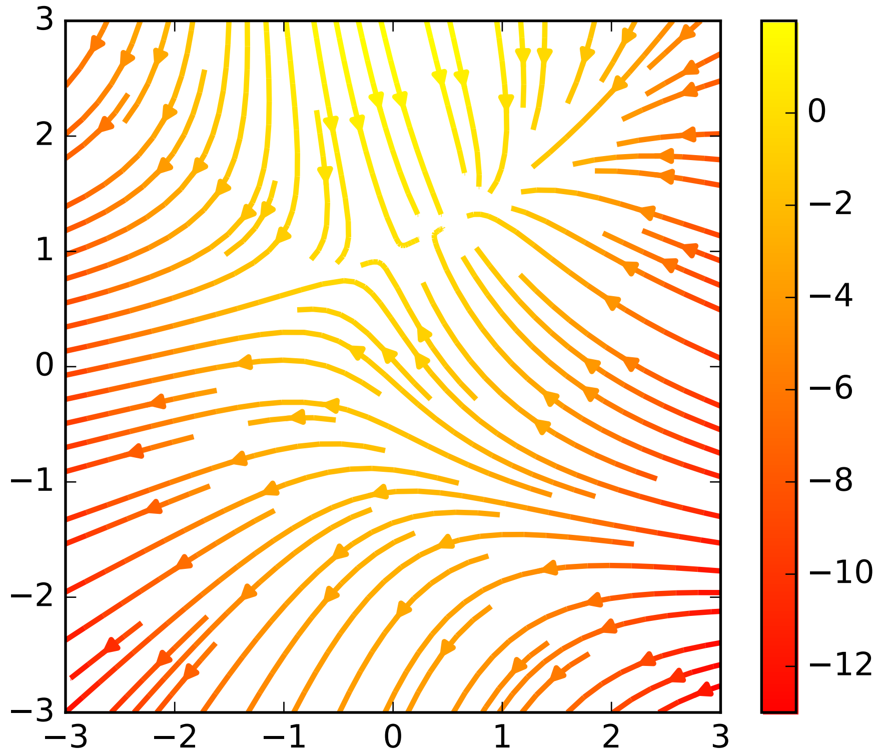


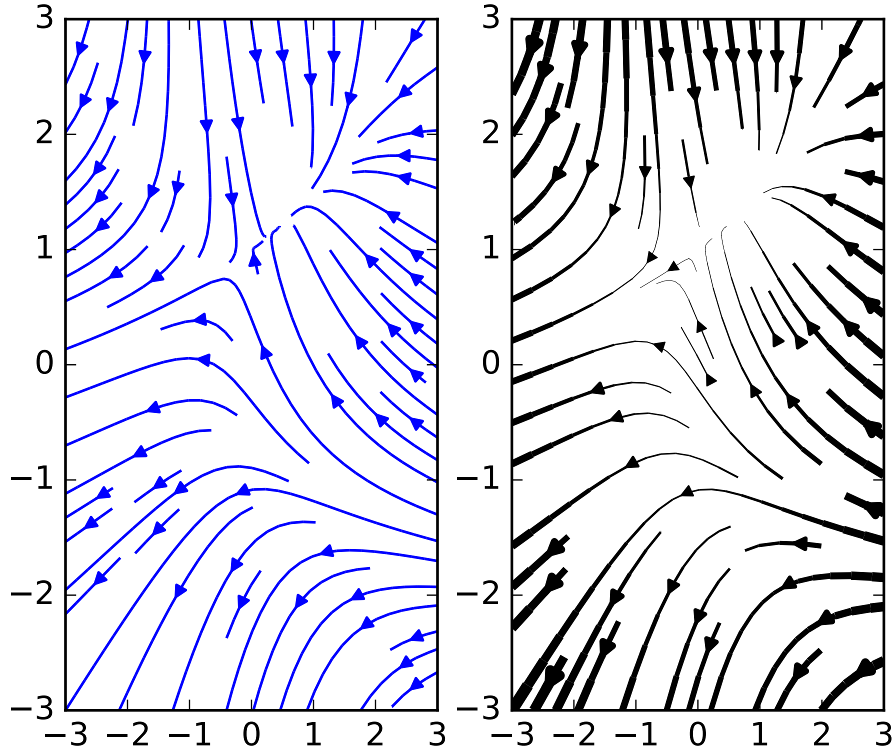

```
"""
Shows how to combine Normalization and Colormap instances to draw
"levels" in pcolor, pcolormesh and imshow type plots in a similar
way to the levels keyword argument to contour/contourf.

"""

import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm
from matplotlib.ticker import MaxNLocator
import numpy as np


# make these smaller to increase the resolution
dx, dy = 0.05, 0.05

# generate 2 2d grids for the x & y bounds
y, x = np.mgrid[slice(1, 5 + dy, dy),
                slice(1, 5 + dx, dx)]

z = np.sin(x)**10 + np.cos(10 + y*x) * np.cos(x)
```

```
# x and y are bounds, so z should be the value *inside* those bounds.
# Therefore, remove the last value from the z array.
z = z[:-1, :-1]
levels = MaxNLocator(nbins=15).tick_values(z.min(), z.max())


# pick the desired colormap, sensible levels, and define a normalization
# instance which takes data values and translates those into levels.
cmap = plt.get_cmap('PiYG')
norm = BoundaryNorm(levels, ncolors=cmap.N, clip=True)

fig, (ax0, ax1) = plt.subplots(nrows=2)

im = ax0.pcolormesh(x, y, z, cmap=cmap, norm=norm)
fig.colorbar(im, ax=ax0)
ax0.set_title('pcolormesh with levels')


# contours are *point* based plots, so convert our bound into point
# centers
cf = ax1.contourf(x[:-1, :-1] + dx/2.,
                  y[:-1, :-1] + dy/2., z, levels=levels,
                  cmap=cmap)
fig.colorbar(cf, ax=ax1)
ax1.set_title('contourf with levels')

# adjust spacing between subplots so `ax1` title and `ax0` tick labels
# don't overlap
fig.tight_layout()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

85.7 images_contours_and_fields example code: streamplot_demo_features.py



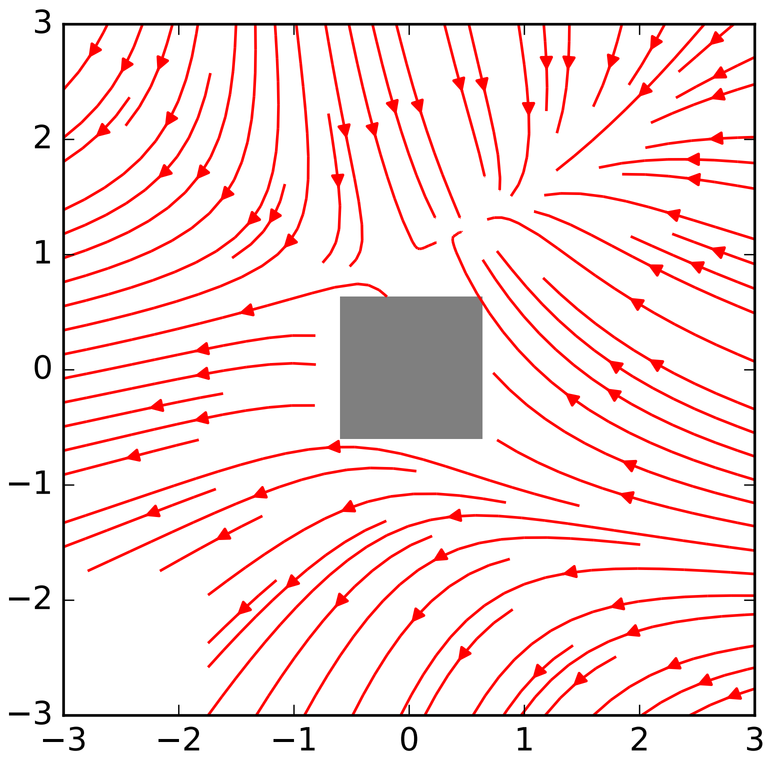



```
"""
Demo of the `streamplot` function.

A streamplot, or streamline plot, is used to display 2D vector fields. This
example shows a few features of the stream plot function:

    * Varying the color along a streamline.
    * Varying the density of streamlines.
    * Varying the line width along a stream line.
"""
import numpy as np
import matplotlib.pyplot as plt

Y, X = np.mgrid[-3:3:100j, -3:3:100j]
U = -1 - X**2 + Y
V = 1 + X - Y**2
speed = np.sqrt(U*U + V*V)

fig0, ax0 = plt.subplots()
strm = ax0.streamplot(X, Y, U, V, color=U, linewidth=2, cmap=plt.cm.autumn)
fig0.colorbar(strm.lines)

fig1, (ax1, ax2) = plt.subplots(ncols=2)
ax1.streamplot(X, Y, U, V, density=[0.5, 1])

lw = 5*speed / speed.max()
```

```
ax2.streamplot(X, Y, U, V, density=0.6, color='k', linewidth=lw)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

85.8 images_contours_and_fields example code: streamplot_demo_masking.py

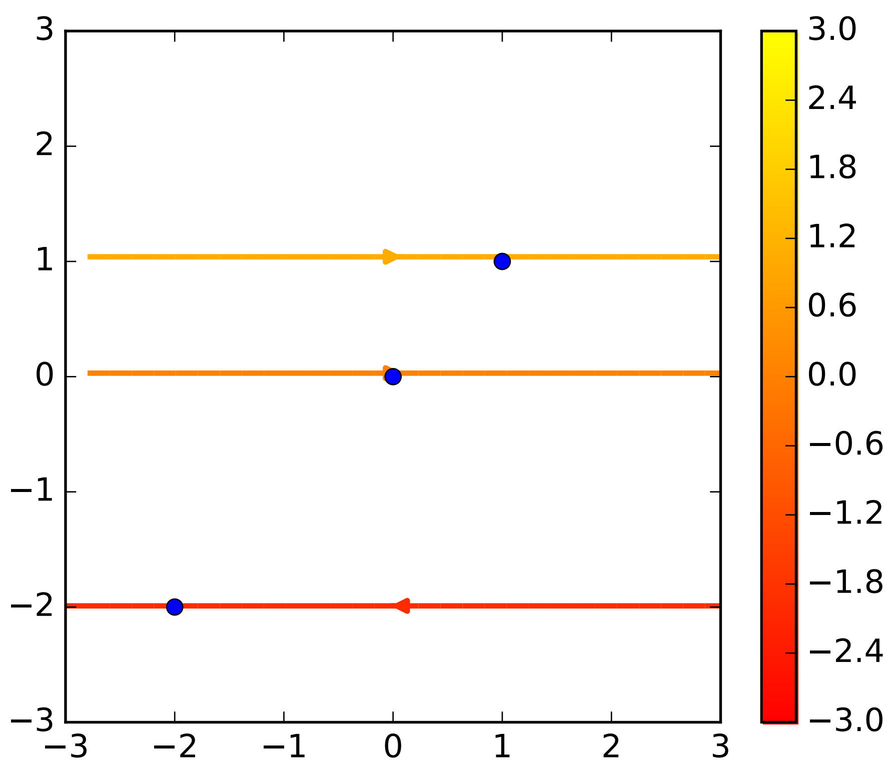



```
"""
Demo of the streamplot function with masking.

This example shows how streamlines created by the streamplot function skips
masked regions and NaN values.
"""
import numpy as np
import matplotlib.pyplot as plt

w = 3
Y, X = np.mgrid[-w:w:100j, -w:w:100j]
U = -1 - X**2 + Y
V = 1 + X - Y**2
```

```
speed = np.sqrt(U*U + V*V)

mask = np.zeros(U.shape, dtype=bool)
mask[40:60, 40:60] = 1
U = np.ma.array(U, mask=mask)
U[:20, :20] = np.nan

plt.streamplot(X, Y, U, V, color='r')

plt.imshow(~mask, extent=(-w, w, -w, w), alpha=0.5,
           interpolation='nearest', cmap=plt.cm.gray)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

85.9 images_contours_and_fields example code: streamplot_demo_start_points.py




```
"""
Demo of the `streamplot` function.
```

```
A streamplot, or streamline plot, is used to display 2D vector fields. This
example shows a few features of the stream plot function:

    * Varying the color along a streamline.
    * Varying the density of streamlines.
    * Varying the line width along a stream line.
"""
import numpy as np
import matplotlib.pyplot as plt

X, Y = (np.linspace(-3, 3, 100),
        np.linspace(-3, 3, 100))

U, V = np.mgrid[-3:3:100j, 0:0:100j]

seed_points = np.array([[-2, 0, 1], [-2, 0, 1]])

fig0, ax0 = plt.subplots()
strm = ax0.streamplot(X, Y, U, V, color=U, linewidth=2,
                      cmap=plt.cm.autumn, start_points=seed_points.T)
fig0.colorbar(strm.lines)

ax0.plot(seed_points[0], seed_points[1], 'bo')

ax0.axis((-3, 3, -3, 3))

plt.show()
```

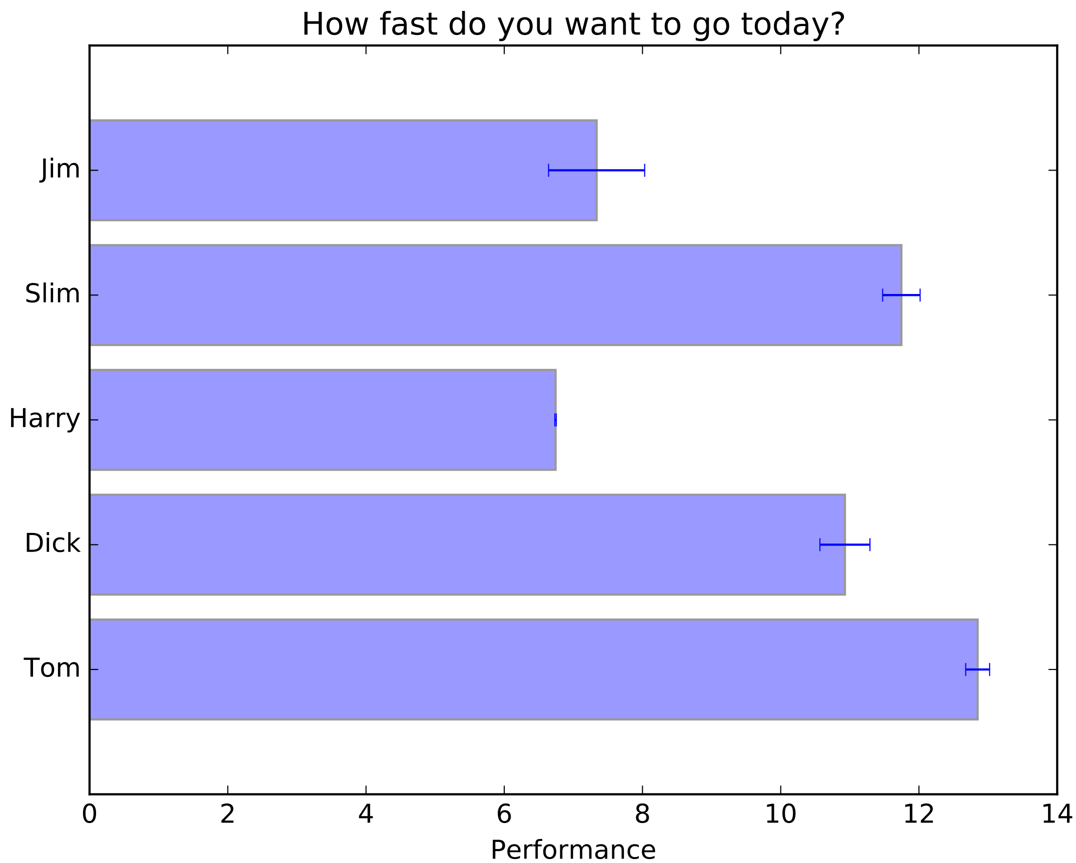
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

EIGHTYSIX

LINES_BARS_AND_MARKERS EXAMPLES

86.1 lines_bars_and_markers example code: barh_demo.py

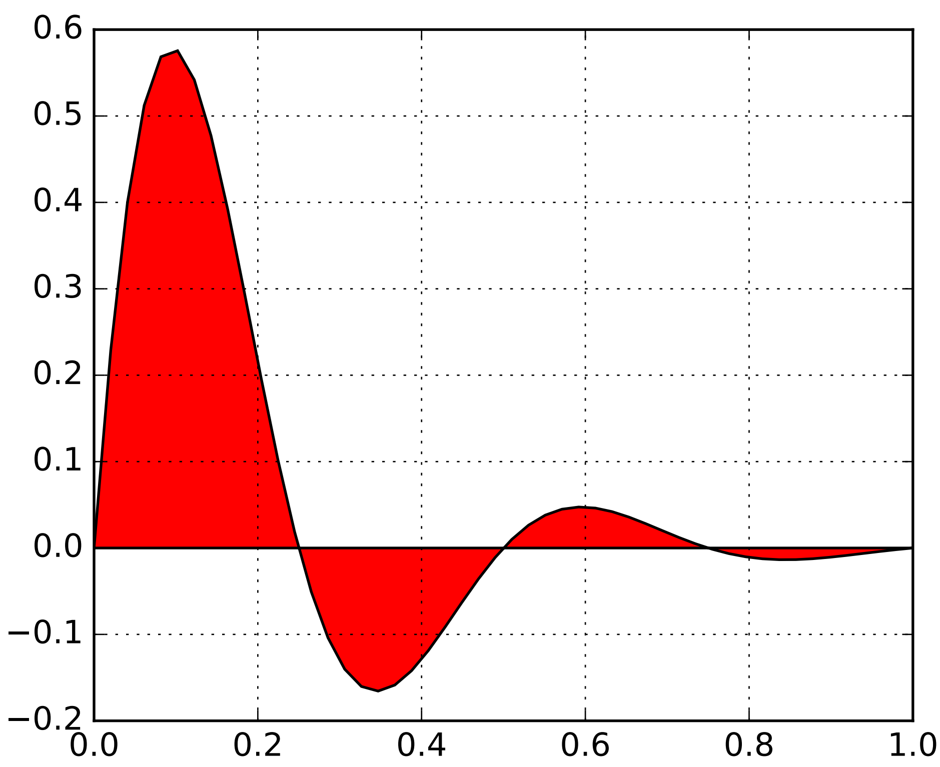



```
"""
Simple demo of a horizontal bar chart.
"""
import matplotlib.pyplot as plt
plt.rcdefaults()
import numpy as np
import matplotlib.pyplot as plt
```

```
# Example data
people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))

plt.barh(y_pos, performance, xerr=error, align='center', alpha=0.4)
plt.yticks(y_pos, people)
plt.xlabel('Performance')
plt.title('How fast do you want to go today?')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

86.2 lines_bars_and_markers example code: fill_demo.py

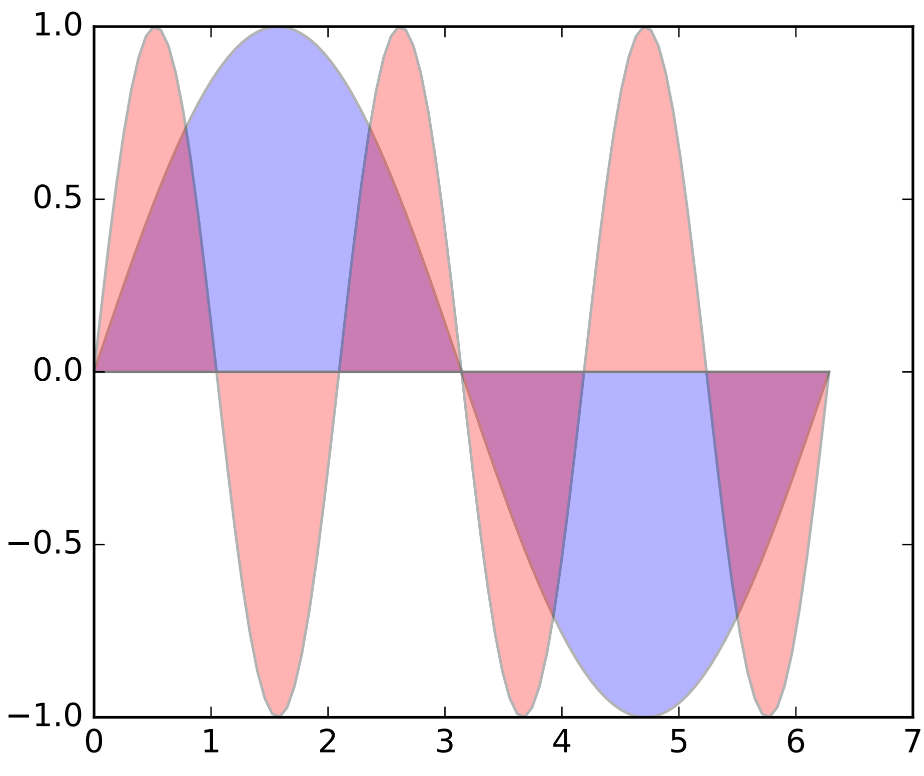



```
"""
Simple demo of the fill function.
"""
```

```
import numpy as np
import matplotlib.pyplot as plt


x = np.linspace(0, 1)
y = np.sin(4 * np.pi * x) * np.exp(-5 * x)

plt.fill(x, y, 'r')
plt.grid(True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

86.3 lines_bars_and_markers example code: fill_demo_features.py

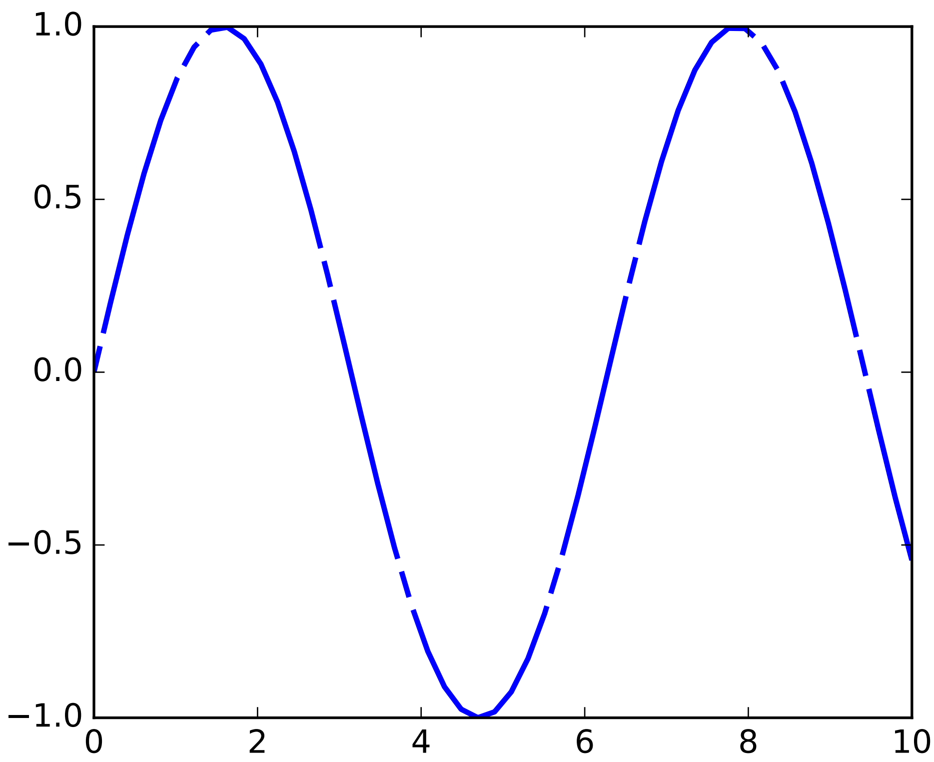



```
"""
Demo of the fill function with a few features.

In addition to the basic fill plot, this demo shows a few optional features:

    * Multiple curves with a single command.
    * Setting the fill color.
```

```
    * Setting the opacity (alpha value).
"""
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.sin(3 * x)
plt.fill(x, y1, 'b', x, y2, 'r', alpha=0.3)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

86.4 lines_bars_and_markers example code: line_demo_dash_control.py

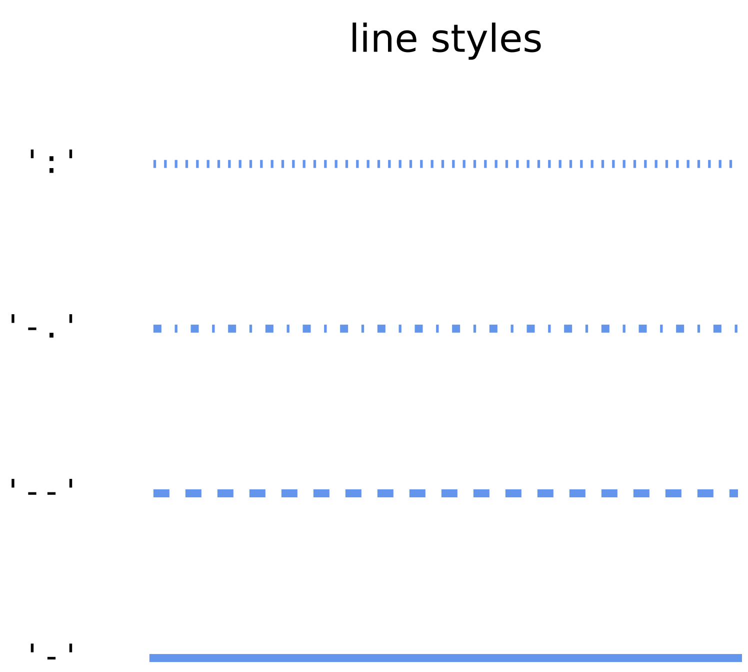



```
"""
Demo of a simple plot with a custom dashed line.

A Line object's ``set_dashes`` method allows you to specify dashes with
a series of on/off lengths (in points).
"""
import numpy as np
```

```
import matplotlib.pyplot as plt


x = np.linspace(0, 10)
line, = plt.plot(x, np.sin(x), '--', linewidth=2)

dashes = [10, 5, 100, 5]  # 10 points on, 5 off, 100 on, 5 off
line.set_dashes(dashes)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

86.5 lines_bars_and_markers example code: line_styles_reference.py

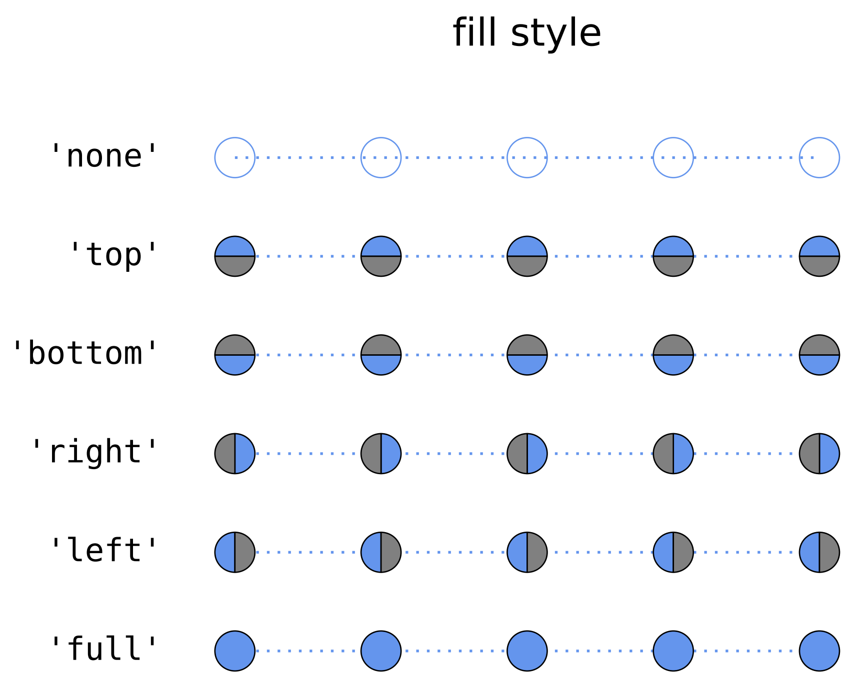



```
"""
Reference for line-styles included with Matplotlib.
"""
import numpy as np
import matplotlib.pyplot as plt
```

```
color = 'cornflowerblue'
points = np.ones(5)  # Draw 5 points for each line
text_style = dict(horizontalalignment='right', verticalalignment='center',
                  fontsize=12, fontdict={'family': 'monospace'})


def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()


def nice_repr(text):
    return repr(text).lstrip('u')


# Plot all line styles.
f, ax = plt.subplots()

linestyles = ['-', '--', '-.', ':']
for y, linestyle in enumerate(linestyles):
    ax.text(-0.5, y, nice_repr(linestyle), **text_style)
    ax.plot(y * points, linestyle=linestyle, color=color, linewidth=3)
    format_axes(ax)
    ax.set_title('line styles')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

86.6 lines_bars_and_markers example code: marker_fillstyle_reference.py

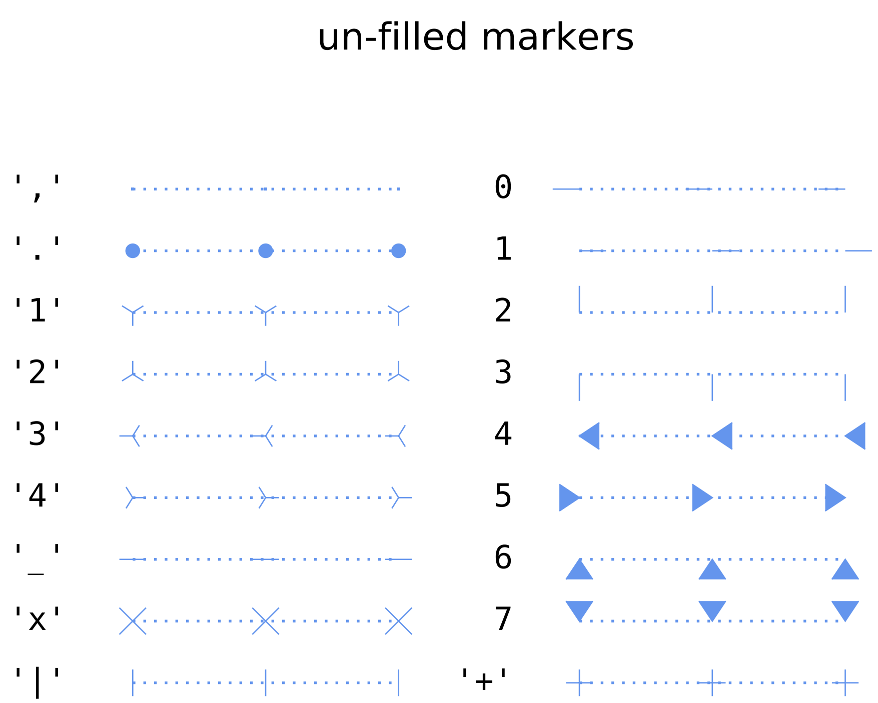


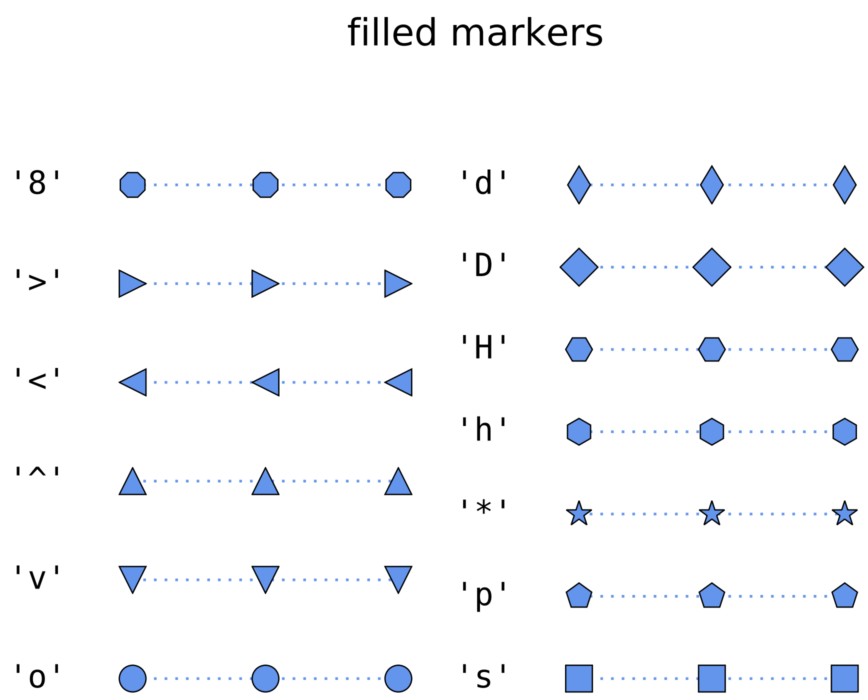

```
"""
Reference for marker fill-styles included with Matplotlib.
"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D


points = np.ones(5)  # Draw 3 points for each line
text_style = dict(horizontalalignment='right', verticalalignment='center',
                  fontsize=12, fontdict={'family': 'monospace'})
marker_style = dict(color='cornflowerblue', linestyle=':', marker='o',
                    markersize=15, markerfacecoloralt='gray')


def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()


def nice_repr(text):
    return repr(text).lstrip('u')
```

```
fig, ax = plt.subplots()

# Plot all fill styles.
for y, fill_style in enumerate(Line2D.fillStyles):
    ax.text(-0.5, y, nice_repr(fill_style), **text_style)
    ax.plot(y * points, fillstyle=fill_style, **marker_style)
    format_axes(ax)
    ax.set_title('fill style')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

86.7 lines_bars_and_markers example code: marker_reference.py



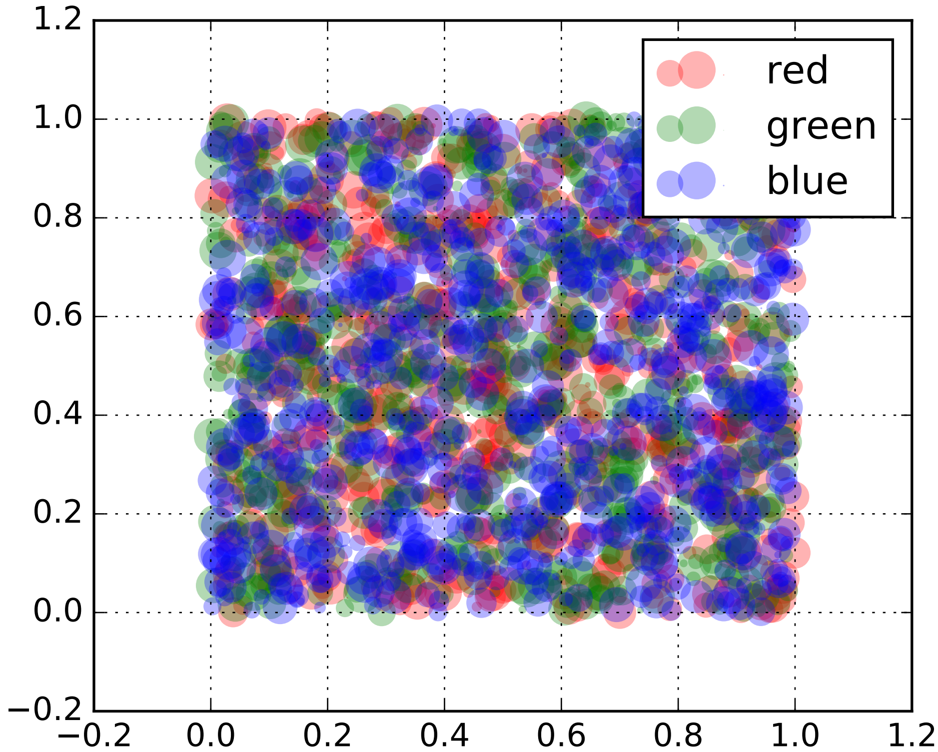



```python
"""
Reference for filled- and unfilled-marker types included with Matplotlib.
"""
from six import iteritems
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D


points = np.ones(3)  # Draw 3 points for each line
text_style = dict(horizontalalignment='right', verticalalignment='center',
                  fontsize=12, fontdict={'family': 'monospace'})
marker_style = dict(linestyle=':', color='cornflowerblue', markersize=10)


def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()


def nice_repr(text):
    return repr(text).lstrip('u')


def split_list(a_list):
    i_half = len(a_list) // 2
```

```
    return (a_list[:i_half], a_list[i_half:])


# Plot all un-filled markers
# --------------------------

fig, axes = plt.subplots(ncols=2)

# Filter out filled markers and marker settings that do nothing.
# We use iteritems from six to make sure that we get an iterator
# in both python 2 and 3
unfilled_markers = [m for m, func in iteritems(Line2D.markers)
                    if func != 'nothing' and m not in Line2D.filled_markers]
# Reverse-sort for pretty. We use our own sort key which is essentially
# a python3 compatible reimplementation of python2 sort.
unfilled_markers = sorted(unfilled_markers,
                          key=lambda x: (str(type(x)), str(x)))[::-1]
for ax, markers in zip(axes, split_list(unfilled_markers)):
    for y, marker in enumerate(markers):
        ax.text(-0.5, y, nice_repr(marker), **text_style)
        ax.plot(y * points, marker=marker, **marker_style)
        format_axes(ax)
fig.suptitle('un-filled markers', fontsize=14)


# Plot all filled markers.
# ------------------------

fig, axes = plt.subplots(ncols=2)
for ax, markers in zip(axes, split_list(Line2D.filled_markers)):
    for y, marker in enumerate(markers):
        ax.text(-0.5, y, nice_repr(marker), **text_style)
        ax.plot(y * points, marker=marker, **marker_style)
        format_axes(ax)
fig.suptitle('filled markers', fontsize=14)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

86.8 lines_bars_and_markers example code: scatter_with_legend.py




```
import matplotlib.pyplot as plt
from numpy.random import rand


for color in ['red', 'green', 'blue']:
    n = 750
    x, y = rand(2, n)
    scale = 200.0 * rand(n)
    plt.scatter(x, y, c=color, s=scale, label=color,
                alpha=0.3, edgecolors='none')

plt.legend()
plt.grid(True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

EIGHTYSEVEN

MISC EXAMPLES

87.1 misc example code: contour_manual.py

[source code]

```
"""
Example of displaying your own contour lines and polygons using ContourSet.
"""
import matplotlib.pyplot as plt
from matplotlib.contour import ContourSet
import matplotlib.cm as cm

# Contour lines for each level are a list/tuple of polygons.
lines0 = [[[0, 0], [0, 4]]]
lines1 = [[[2, 0], [1, 2], [1, 3]]]
lines2 = [[[3, 0], [3, 2]], [[3, 3], [3, 4]]]  # Note two lines.

# Filled contours between two levels are also a list/tuple of polygons.
# Points can be ordered clockwise or anticlockwise.
filled01 = [[[0, 0], [0, 4], [1, 3], [1, 2], [2, 0]]]
filled12 = [[[2, 0], [3, 0], [3, 2], [1, 3], [1, 2]],   # Note two polygons.
            [[1, 4], [3, 4], [3, 3]]]


plt.figure()

# Filled contours using filled=True.
cs = ContourSet(plt.gca(), [0, 1, 2], [filled01, filled12], filled=True, cmap=cm.bone)
cbar = plt.colorbar(cs)

# Contour lines (non-filled).
lines = ContourSet(plt.gca(), [0, 1, 2], [lines0, lines1, lines2], cmap=cm.cool,
                   linewidths=3)
cbar.add_lines(lines)

plt.axis([-0.5, 3.5, -0.5, 4.5])
plt.title('User-specified contours')


# Multiple filled contour lines can be specified in a single list of polygon
```

```
# vertices along with a list of vertex kinds (code types) as described in the
# Path class.  This is particularly useful for polygons with holes.
# Here a code type of 1 is a MOVETO, and 2 is a LINETO.

plt.figure()
filled01 = [[[0, 0], [3, 0], [3, 3], [0, 3], [1, 1], [1, 2], [2, 2], [2, 1]]]
kinds01 = [[1, 2, 2, 2, 1, 2, 2, 2]]
cs = ContourSet(plt.gca(), [0, 1], [filled01], [kinds01], filled=True)
cbar = plt.colorbar(cs)

plt.axis([-0.5, 3.5, -0.5, 3.5])
plt.title('User specified filled contours with holes')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.2 misc example code: font_indexing.py

[source code]

```
"""
A little example that shows how the various indexing into the font
tables relate to one another.  Mainly for mpl developers....

"""
from __future__ import print_function
import matplotlib
from matplotlib.ft2font import FT2Font, KERNING_DEFAULT, KERNING_UNFITTED, KERNING_
↪UNSCALED


#fname = '/usr/share/fonts/sfd/FreeSans.ttf'
fname = matplotlib.get_data_path() + '/fonts/ttf/Vera.ttf'
font = FT2Font(fname)
font.set_charmap(0)

codes = font.get_charmap().items()
#dsu = [(ccode, glyphind) for ccode, glyphind in codes]
#dsu.sort()
#for ccode, glyphind in dsu:
#    try: name = font.get_glyph_name(glyphind)
#    except RuntimeError: pass
#    else: print('% 4d % 4d %s %s' % (glyphind, ccode, hex(int(ccode)), name))


# make a charname to charcode and glyphind dictionary
coded = {}
glyphd = {}
for ccode, glyphind in codes:
    name = font.get_glyph_name(glyphind)
```

```
    coded[name] = ccode
    glyphd[name] = glyphind

code = coded['A']
glyph = font.load_char(code)
#print(glyph.bbox)
print(glyphd['A'], glyphd['V'], coded['A'], coded['V'])
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_DEFAULT))
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_UNFITTED))
print('AV', font.get_kerning(glyphd['A'], glyphd['V'], KERNING_UNSCALED))
print('AV', font.get_kerning(glyphd['A'], glyphd['T'], KERNING_UNSCALED))
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.3 misc example code: ftface_props.py

[source code]

```
#!/usr/bin/env python

from __future__ import print_function
"""
This is a demo script to show you how to use all the properties of an
FT2Font object.  These describe global font properties.  For
individual character metrices, use the Glyp object, as returned by
load_char
"""
import matplotlib
import matplotlib.ft2font as ft


#fname = '/usr/local/share/matplotlib/VeraIt.ttf'
fname = matplotlib.get_data_path() + '/fonts/ttf/VeraIt.ttf'
#fname = '/usr/local/share/matplotlib/cmr10.ttf'

font = ft.FT2Font(fname)

print('Num faces   :', font.num_faces)        # number of faces in file
print('Num glyphs  :', font.num_glyphs)       # number of glyphs in the face
print('Family name :', font.family_name)      # face family name
print('Syle name   :', font.style_name)       # face syle name
print('PS name     :', font.postscript_name)  # the postscript name
print('Num fixed   :', font.num_fixed_sizes)  # number of embedded bitmap in face

# the following are only available if face.scalable
if font.scalable:
    # the face global bounding box (xmin, ymin, xmax, ymax)
    print('Bbox                :', font.bbox)
    # number of font units covered by the EM
    print('EM                  :', font.units_per_EM)
    # the ascender in 26.6 units
```

```
    print('Ascender            :', font.ascender)
    # the descender in 26.6 units
    print('Descender           :', font.descender)
    # the height in 26.6 units
    print('Height              :', font.height)
    # maximum horizontal cursor advance
    print('Max adv width       :', font.max_advance_width)
    # same for vertical layout
    print('Max adv height      :', font.max_advance_height)
    # vertical position of the underline bar
    print('Underline pos       :', font.underline_position)
    # vertical thickness of the underline
    print('Underline thickness :', font.underline_thickness)

for style in ('Italic',
              'Bold',
              'Scalable',
              'Fixed sizes',
              'Fixed width',
              'SFNT',
              'Horizontal',
              'Vertical',
              'Kerning',
              'Fast glyphs',
              'Multiple masters',
              'Glyph names',
              'External stream'):
    bitpos = getattr(ft, style.replace(' ', '_').upper()) - 1
    print('%-17s:' % style, bool(font.style_flags & (1 << bitpos)))

print(dir(font))

cmap = font.get_charmap()
print(font.get_kerning)
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.4 misc example code: image_thumbnail.py

[source code]

```
"""
You can use matplotlib to generate thumbnails from existing images.
matplotlib natively supports PNG files on the input side, and other
image types transparently if your have PIL installed
"""

from __future__ import print_function
# build thumbnails of all images in a directory
import sys
import os
```

```
import glob
import matplotlib.image as image


if len(sys.argv) != 2:
    print('Usage: python %s IMAGEDIR' % __file__)
    raise SystemExit
indir = sys.argv[1]
if not os.path.isdir(indir):
    print('Could not find input directory "%s"' % indir)
    raise SystemExit

outdir = 'thumbs'
if not os.path.exists(outdir):
    os.makedirs(outdir)

for fname in glob.glob(os.path.join(indir, '*.png')):
    basedir, basename = os.path.split(fname)
    outfile = os.path.join(outdir, basename)
    fig = image.thumbnail(fname, outfile, scale=0.15)
    print('saved thumbnail of %s to %s' % (fname, outfile))
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.5 misc example code: longshort.py

[source code]

```
"""
Illustrate the rec array utility funcitons by loading prices from a
csv file, computing the daily returns, appending the results to the
record arrays, joining on date
"""
import urllib
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

# grab the price data off yahoo
u1 = urllib.urlretrieve('http://ichart.finance.yahoo.com/table.csv?s=AAPL&d=9&e=14&
↪f=2008&g=d&a=8&b=7&c=1984&ignore=.csv')
u2 = urllib.urlretrieve('http://ichart.finance.yahoo.com/table.csv?s=GOOG&d=9&e=14&
↪f=2008&g=d&a=8&b=7&c=1984&ignore=.csv')

# load the CSV files into record arrays
r1 = mlab.csv2rec(file(u1[0]))
r2 = mlab.csv2rec(file(u2[0]))

# compute the daily returns and add these columns to the arrays
gains1 = np.zeros_like(r1.adj_close)
gains2 = np.zeros_like(r2.adj_close)
```

```
gains1[1:] = np.diff(r1.adj_close)/r1.adj_close[:-1]
gains2[1:] = np.diff(r2.adj_close)/r2.adj_close[:-1]
r1 = mlab.rec_append_fields(r1, 'gains', gains1)
r2 = mlab.rec_append_fields(r2, 'gains', gains2)

# now join them by date; the default postfixes are 1 and 2.  The
# default jointype is inner so it will do an intersection of dates and
# drop the dates in AAPL which occurred before GOOG started trading in
# 2004.  r1 and r2 are reverse ordered by date since Yahoo returns
# most recent first in the CSV files, but rec_join will sort by key so
# r below will be properly sorted
r = mlab.rec_join('date', r1, r2)


# long appl, short goog
g = r.gains1 - r.gains2
tr = (1 + g).cumprod()  # the total return

# plot the return
fig, ax = plt.subplots()
ax.plot(r.date, tr)
ax.set_title('total return: long APPL, short GOOG')
ax.grid()
fig.autofmt_xdate()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.6 misc example code: multiprocess.py

[source code]

```
# Demo of using multiprocessing for generating data in one process and plotting
# in another.
# Written by Robert Cimrman
# Requires >= Python 2.6 for the multiprocessing module or having the
# standalone processing module installed

from __future__ import print_function
import time
try:
    from multiprocessing import Process, Pipe
except ImportError:
    from processing import Process, Pipe
import numpy as np

import matplotlib
matplotlib.use('GtkAgg')
import matplotlib.pyplot as plt
import gobject
```

```
class ProcessPlotter(object):
    def __init__(self):
        self.x = []
        self.y = []

    def terminate(self):
        plt.close('all')

    def poll_draw(self):

        def call_back():
            while 1:
                if not self.pipe.poll():
                    break

                command = self.pipe.recv()

                if command is None:
                    self.terminate()
                    return False

                else:
                    self.x.append(command[0])
                    self.y.append(command[1])
                    self.ax.plot(self.x, self.y, 'ro')

            self.fig.canvas.draw()
            return True

        return call_back

    def __call__(self, pipe):
        print('starting plotter...')

        self.pipe = pipe
        self.fig, self.ax = plt.subplots()
        self.gid = gobject.timeout_add(1000, self.poll_draw())

        print('...done')
        plt.show()


class NBPlot(object):
    def __init__(self):
        self.plot_pipe, plotter_pipe = Pipe()
        self.plotter = ProcessPlotter()
        self.plot_process = Process(target=self.plotter,
                                    args=(plotter_pipe,))
        self.plot_process.daemon = True
        self.plot_process.start()

    def plot(self, finished=False):
```

```
        send = self.plot_pipe.send
        if finished:
            send(None)
        else:
            data = np.random.random(2)
            send(data)


def main():
    pl = NBPlot()
    for ii in range(10):
        pl.plot()
        time.sleep(0.5)
    raw_input('press Enter...')
    pl.plot(finished=True)

if __name__ == '__main__':
    main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.7 misc example code: rasterization_demo.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt

d = np.arange(100).reshape(10, 10)
x, y = np.meshgrid(np.arange(11), np.arange(11))

theta = 0.25*np.pi
xx = x*np.cos(theta) - y*np.sin(theta)
yy = x*np.sin(theta) + y*np.cos(theta)

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
ax1.set_aspect(1)
ax1.pcolormesh(xx, yy, d)
ax1.set_title("No Rasterization")

ax2.set_aspect(1)
ax2.set_title("Rasterization")

m = ax2.pcolormesh(xx, yy, d)
m.set_rasterized(True)

ax3.set_aspect(1)
ax3.pcolormesh(xx, yy, d)
ax3.text(0.5, 0.5, "Text", alpha=0.2,
         va="center", ha="center", size=50, transform=ax3.transAxes)
```

```
ax3.set_title("No Rasterization")


ax4.set_aspect(1)
m = ax4.pcolormesh(xx, yy, d)
m.set_zorder(-20)

ax4.text(0.5, 0.5, "Text", alpha=0.2,
         zorder=-15,
         va="center", ha="center", size=50, transform=ax4.transAxes)

ax4.set_rasterization_zorder(-10)

ax4.set_title("Rasterization z$<-10$")


# ax2.title.set_rasterized(True) # should display a warning

plt.savefig("test_rasterization.pdf", dpi=150)
plt.savefig("test_rasterization.eps", dpi=150)

if not plt.rcParams["text.usetex"]:
    plt.savefig("test_rasterization.svg", dpi=150)
    # svg backend currently ignores the dpi
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.8 misc example code: rc_traits.py

[source code]

```
# Here is some example code showing how to define some representative
# rc properties and construct a matplotlib artist using traits.
# matplotlib does not ship with enthought.traits, so you will need to
# install it separately.

from __future__ import print_function

import sys
import os
import re
import traits.api as traits
from matplotlib.cbook import is_string_like
from matplotlib.artist import Artist

doprint = True
flexible_true_trait = traits.Trait(
   True,
   {'true': True, 't': True, 'yes': True, 'y': True, 'on': True, True: True,
    'false': False, 'f': False, 'no': False, 'n': False, 'off': False, False: False
    })
```

```python
flexible_false_trait = traits.Trait(False, flexible_true_trait)

colors = {
   'c': '#00bfbf',
   'b': '#0000ff',
   'g': '#008000',
   'k': '#000000',
   'm': '#bf00bf',
   'r': '#ff0000',
   'w': '#ffffff',
   'y': '#bfbf00',
   'gold':        '#FFD700',
   'peachpuff':   '#FFDAB9',
   'navajowhite': '#FFDEAD',
   }


def hex2color(s):
    "Convert hex string (like html uses, eg, #efefef) to a r,g,b tuple"
    return tuple([int(n, 16)/255.0 for n in (s[1:3], s[3:5], s[5:7])])


class RGBA(traits.HasTraits):
    # r,g,b,a in the range 0-1 with default color 0,0,0,1 (black)
    r = traits.Range(0., 1., 0.)
    g = traits.Range(0., 1., 0.)
    b = traits.Range(0., 1., 0.)
    a = traits.Range(0., 1., 1.)

    def __init__(self, r=0., g=0., b=0., a=1.):
        self.r = r
        self.g = g
        self.b = b
        self.a = a

    def __repr__(self):
        return 'r,g,b,a = (%1.2f, %1.2f, %1.2f, %1.2f)' %\
               (self.r, self.g, self.b, self.a)


def tuple_to_rgba(ob, name, val):
    tup = [float(x) for x in val]
    if len(tup) == 3:
        r, g, b = tup
        return RGBA(r, g, b)
    elif len(tup) == 4:
        r, g, b, a = tup
        return RGBA(r, g, b, a)
    else:
        raise ValueError
tuple_to_rgba.info = 'a RGB or RGBA tuple of floats'
```

```
def hex_to_rgba(ob, name, val):
    rgx = re.compile('^#[0-9A-Fa-f]{6}$')

    if not is_string_like(val):
        raise TypeError
    if rgx.match(val) is None:
        raise ValueError
    r, g, b = hex2color(val)
    return RGBA(r, g, b, 1.0)
hex_to_rgba.info = 'a hex color string'


def colorname_to_rgba(ob, name, val):
    hex = colors[val.lower()]
    r, g, b = hex2color(hex)
    return RGBA(r, g, b, 1.0)
colorname_to_rgba.info = 'a named color'


def float_to_rgba(ob, name, val):
    val = float(val)
    return RGBA(val, val, val, 1.)
float_to_rgba.info = 'a grayscale intensity'


Color = traits.Trait(RGBA(), float_to_rgba, colorname_to_rgba, RGBA,
                     hex_to_rgba, tuple_to_rgba)


def file_exists(ob, name, val):
    fh = file(val, 'r')
    return val

linestyles = ('-', '--', '-.', ':', 'steps', 'None')
TICKLEFT, TICKRIGHT, TICKUP, TICKDOWN = range(4)
linemarkers = (None, '.', ',', 'o', '^', 'v', '<', '>', 's',
               '+', 'x', 'd', 'D', '|', '_', 'h', 'H',
               'p', '1', '2', '3', '4',
               TICKLEFT,
               TICKRIGHT,
               TICKUP,
               TICKDOWN,
               'None'
               )


class LineRC(traits.HasTraits):
    linewidth = traits.Float(0.5)
    linestyle = traits.Trait(*linestyles)
    color = Color
    marker = traits.Trait(*linemarkers)
    markerfacecolor = Color
    markeredgecolor = Color
```

```
    markeredgewidth = traits.Float(0.5)
    markersize = traits.Float(6)
    antialiased = flexible_true_trait
    data_clipping = flexible_false_trait


class PatchRC(traits.HasTraits):
    linewidth = traits.Float(1.0)
    facecolor = Color
    edgecolor = Color
    antialiased = flexible_true_trait

timezones = 'UTC', 'US/Central', 'ES/Eastern'  # fixme: and many more
backends = ('GTKAgg', 'Cairo', 'GDK', 'GTK', 'Agg',
            'GTKCairo', 'PS', 'SVG', 'Template', 'TkAgg',
            'WX')


class RC(traits.HasTraits):
    backend = traits.Trait(*backends)
    interactive = flexible_false_trait
    toolbar = traits.Trait('toolbar2', 'classic', None)
    timezone = traits.Trait(*timezones)
    lines = traits.Trait(LineRC())
    patch = traits.Trait(PatchRC())

rc = RC()
rc.lines.color = 'r'
if doprint:
    print('RC')
    rc.print_traits()
    print('RC lines')
    rc.lines.print_traits()
    print('RC patches')
    rc.patch.print_traits()


class Patch(Artist, traits.HasTraits):
    linewidth = traits.Float(0.5)
    facecolor = Color
    fc = facecolor
    edgecolor = Color
    fill = flexible_true_trait

    def __init__(self,
                 edgecolor=None,
                 facecolor=None,
                 linewidth=None,
                 antialiased=None,
                 fill=1,
                 **kwargs
                 ):
        Artist.__init__(self)
```

```
        if edgecolor is None:
            edgecolor = rc.patch.edgecolor
        if facecolor is None:
            facecolor = rc.patch.facecolor
        if linewidth is None:
            linewidth = rc.patch.linewidth
        if antialiased is None:
            antialiased = rc.patch.antialiased

        self.edgecolor = edgecolor
        self.facecolor = facecolor
        self.linewidth = linewidth
        self.antialiased = antialiased
        self.fill = fill


p = Patch()
p.facecolor = '#bfbf00'
p.edgecolor = 'gold'
p.facecolor = (1, .5, .5, .25)
p.facecolor = 0.25
p.fill = 'f'
print('p.facecolor', type(p.facecolor), p.facecolor)
print('p.fill', type(p.fill), p.fill)
if p.fill_:
    print('fill')
else:
    print('no fill')
if doprint:
    print()
    print('Patch')
    p.print_traits()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.9 misc example code: rec_groupby_demo.py

[source code]

```
from __future__ import print_function
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook

datafile = cbook.get_sample_data('aapl.csv', asfileobj=False)
print('loading', datafile)
r = mlab.csv2rec(datafile)
r.sort()
```

```
def daily_return(prices):
    'an array of daily returns from price array'
    g = np.zeros_like(prices)
    g[1:] = (prices[1:] - prices[:-1])/prices[:-1]
    return g


def volume_code(volume):
    'code the continuous volume data categorically'
    ind = np.searchsorted([1e5, 1e6, 5e6, 10e6, 1e7], volume)
    return ind

# a list of (dtype_name, summary_function, output_dtype_name).
# rec_summarize will call on each function on the indicated recarray
# attribute, and the result assigned to output name in the return
# record array.
summaryfuncs = (
    ('date', lambda x: [thisdate.year for thisdate in x], 'years'),
    ('date', lambda x: [thisdate.month for thisdate in x], 'months'),
    ('date', lambda x: [thisdate.weekday() for thisdate in x], 'weekday'),
    ('adj_close', daily_return, 'dreturn'),
    ('volume', volume_code, 'volcode'),
    )

rsum = mlab.rec_summarize(r, summaryfuncs)

# stats is a list of (dtype_name, function, output_dtype_name).
# rec_groupby will summarize the attribute identified by the
# dtype_name over the groups in the groupby list, and assign the
# result to the output_dtype_name
stats = (
    ('dreturn', len, 'rcnt'),
    ('dreturn', np.mean, 'rmean'),
    ('dreturn', np.median, 'rmedian'),
    ('dreturn', np.std, 'rsigma'),
    )

# you can summarize over a single variable, like years or months
print('summary by years')
ry = mlab.rec_groupby(rsum, ('years',), stats)
print(mlab. rec2txt(ry))

print('summary by months')
rm = mlab.rec_groupby(rsum, ('months',), stats)
print(mlab.rec2txt(rm))

# or over multiple variables like years and months
print('summary by year and month')
rym = mlab.rec_groupby(rsum, ('years', 'months'), stats)
print(mlab.rec2txt(rym))

print('summary by volume')
rv = mlab.rec_groupby(rsum, ('volcode',), stats)
```

```
print(mlab.rec2txt(rv))
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.10 misc example code: rec_join_demo.py

[source code]

```
from __future__ import print_function
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook

datafile = cbook.get_sample_data('aapl.csv', asfileobj=False)
print('loading', datafile)
r = mlab.csv2rec(datafile)

r.sort()
r1 = r[-10:]

# Create a new array
r2 = np.empty(12, dtype=[('date', '|O4'), ('high', np.float),
                         ('marker', np.float)])
r2 = r2.view(np.recarray)
r2.date = r.date[-17:-5]
r2.high = r.high[-17:-5]
r2.marker = np.arange(12)

print("r1:")
print(mlab.rec2txt(r1))
print("r2:")
print(mlab.rec2txt(r2))

defaults = {'marker': -1, 'close': np.NaN, 'low': -4444.}

for s in ('inner', 'outer', 'leftouter'):
    rec = mlab.rec_join(['date', 'high'], r1, r2,
                        jointype=s, defaults=defaults)
    print("\n%sjoin :\n%s" % (s, mlab.rec2txt(rec)))
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.11 misc example code: sample_data_demo.py

[source code]

```
"""
Grab mpl data from the ~/.matplotlib/sample_data cache if it exists, else
fetch it from github and cache it
```

```
"""
from __future__ import print_function
import matplotlib.cbook as cbook
import matplotlib.pyplot as plt
fname = cbook.get_sample_data('ada.png', asfileobj=False)

print('fname', fname)
im = plt.imread(fname)
plt.imshow(im)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.12 misc example code: svg_filter_line.py

[source code]

```
"""
Demonstrate SVG filtering effects which might be used with mpl.

Note that the filtering effects are only effective if your svg rederer
support it.
"""

from __future__ import print_function
import matplotlib

matplotlib.use("Svg")

import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms

fig1 = plt.figure()
ax = fig1.add_axes([0.1, 0.1, 0.8, 0.8])

# draw lines
l1, = ax.plot([0.1, 0.5, 0.9], [0.1, 0.9, 0.5], "bo-",
              mec="b", lw=5, ms=10, label="Line 1")
l2, = ax.plot([0.1, 0.5, 0.9], [0.5, 0.2, 0.7], "rs-",
              mec="r", lw=5, ms=10, color="r", label="Line 2")


for l in [l1, l2]:

    # draw shadows with same lines with slight offset and gray colors.

    xx = l.get_xdata()
    yy = l.get_ydata()
    shadow, = ax.plot(xx, yy)
    shadow.update_from(l)
```

```
    # adjust color
    shadow.set_color("0.2")
    # adjust zorder of the shadow lines so that it is drawn below the
    # original lines
    shadow.set_zorder(l.get_zorder() - 0.5)

    # offset transform
    ot = mtransforms.offset_copy(l.get_transform(), fig1,
                                 x=4.0, y=-6.0, units='points')

    shadow.set_transform(ot)

    # set the id for a later use
    shadow.set_gid(l.get_label() + "_shadow")


ax.set_xlim(0., 1.)
ax.set_ylim(0., 1.)

# save the figure as a bytes string in the svg format.
from io import BytesIO
f = BytesIO()
plt.savefig(f, format="svg")


import xml.etree.cElementTree as ET

# filter definition for a gaussian blur
filter_def = """
  <defs  xmlns='http://www.w3.org/2000/svg' xmlns:xlink='http://www.w3.org/1999/xlink'>
    <filter id='dropshadow' height='1.2' width='1.2'>
      <feGaussianBlur result='blur' stdDeviation='3'/>
    </filter>
  </defs>
"""


# read in the saved svg
tree, xmlid = ET.XMLID(f.getvalue())

# insert the filter definition in the svg dom tree.
tree.insert(0, ET.XML(filter_def))

for l in [l1, l2]:
    # pick up the svg element with given id
    shadow = xmlid[l.get_label() + "_shadow"]
    # apply shdow filter
    shadow.set("filter", 'url(#dropshadow)')

fn = "svg_filter_line.svg"
print("Saving '%s'" % fn)
ET.ElementTree(tree).write(fn)
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.13 misc example code: svg_filter_pie.py

[source code]

```
"""
Demonstrate SVG filtering effects which might be used with mpl.
The pie chart drawing code is borrowed from pie_demo.py

Note that the filtering effects are only effective if your svg rederer
support it.
"""


import matplotlib
matplotlib.use("Svg")

import matplotlib.pyplot as plt
from matplotlib.patches import Shadow

# make a square figure and axes
fig1 = plt.figure(1, figsize=(6, 6))
ax = fig1.add_axes([0.1, 0.1, 0.8, 0.8])

labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
fracs = [15, 30, 45, 10]

explode = (0, 0.05, 0, 0)

# We want to draw the shadow for each pie but we will not use "shadow"
# option as it does'n save the references to the shadow patches.
pies = ax.pie(fracs, explode=explode, labels=labels, autopct='%1.1f%%')

for w in pies[0]:
    # set the id with the label.
    w.set_gid(w.get_label())

    # we don't want to draw the edge of the pie
    w.set_ec("none")

for w in pies[0]:
    # create shadow patch
    s = Shadow(w, -0.01, -0.01)
    s.set_gid(w.get_gid() + "_shadow")
    s.set_zorder(w.get_zorder() - 0.1)
    ax.add_patch(s)


# save
from io import BytesIO
f = BytesIO()
```

```
plt.savefig(f, format="svg")

import xml.etree.cElementTree as ET


# filter definition for shadow using a gaussian blur
# and lighteneing effect.
# The lightnening filter is copied from http://www.w3.org/TR/SVG/filters.html

# I tested it with Inkscape and Firefox3. "Gaussian blur" is supported
# in both, but the lightnening effect only in the inkscape. Also note
# that, inkscape's exporting also may not support it.

filter_def = """
  <defs  xmlns='http://www.w3.org/2000/svg' xmlns:xlink='http://www.w3.org/1999/xlink'>
    <filter id='dropshadow' height='1.2' width='1.2'>
      <feGaussianBlur result='blur' stdDeviation='2'/>
    </filter>

    <filter id='MyFilter' filterUnits='objectBoundingBox' x='0' y='0' width='1' height='1
↪'>
      <feGaussianBlur in='SourceAlpha' stdDeviation='4%' result='blur'/>
      <feOffset in='blur' dx='4%' dy='4%' result='offsetBlur'/>
      <feSpecularLighting in='blur' surfaceScale='5' specularConstant='.75'
           specularExponent='20' lighting-color='#bbbbbb' result='specOut'>
        <fePointLight x='-5000%' y='-10000%' z='20000%'/>
      </feSpecularLighting>
      <feComposite in='specOut' in2='SourceAlpha' operator='in' result='specOut'/>
      <feComposite in='SourceGraphic' in2='specOut' operator='arithmetic'
    k1='0' k2='1' k3='1' k4='0'/>
    </filter>
  </defs>
"""


tree, xmlid = ET.XMLID(f.getvalue())

# insert the filter definition in the svg dom tree.
tree.insert(0, ET.XML(filter_def))

for i, pie_name in enumerate(labels):
    pie = xmlid[pie_name]
    pie.set("filter", 'url(#MyFilter)')

    shadow = xmlid[pie_name + "_shadow"]
    shadow.set("filter", 'url(#dropshadow)')

fn = "svg_filter_pie.svg"
print("Saving '%s'" % fn)
ET.ElementTree(tree).write(fn)
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

87.14 misc example code: tight_bbox_test.py

[source code]

```
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np

ax = plt.axes([0.1, 0.3, 0.5, 0.5])

ax.pcolormesh(np.array([[1, 2], [3, 4]]))
plt.yticks([0.5, 1.5], ["long long tick label",
                        "tick label"])
plt.ylabel("My y-label")
plt.title("Check saved figures for their bboxes")
for ext in ["png", "pdf", "svg", "svgz", "eps"]:
    print("saving tight_bbox_test.%s" % (ext,))
    plt.savefig("tight_bbox_test.%s" % (ext,), bbox_inches="tight")
plt.show()
```

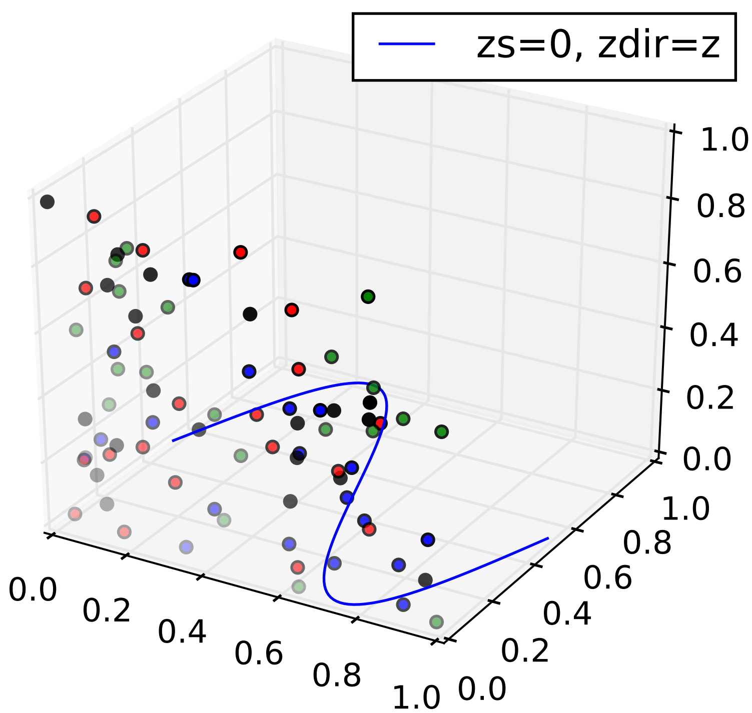
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

EIGHTYEIGHT

MPLOT3D EXAMPLES

88.1 mplot3d example code: 2dcollections3d_demo.py

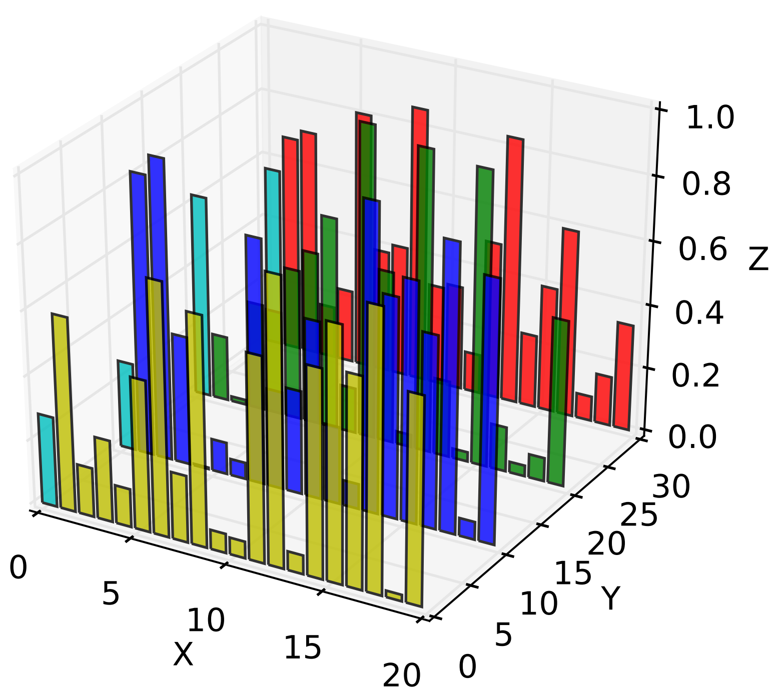



```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.gca(projection='3d')

x = np.linspace(0, 1, 100)
y = np.sin(x * 2 * np.pi) / 2 + 0.5
```

```
ax.plot(x, y, zs=0, zdir='z', label='zs=0, zdir=z')

colors = ('r', 'g', 'b', 'k')
for c in colors:
    x = np.random.sample(20)
    y = np.random.sample(20)
    ax.scatter(x, y, 0, zdir='y', c=c)

ax.legend()
ax.set_xlim3d(0, 1)
ax.set_ylim3d(0, 1)
ax.set_zlim3d(0, 1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.2 mplot3d example code: bars3d_demo.py

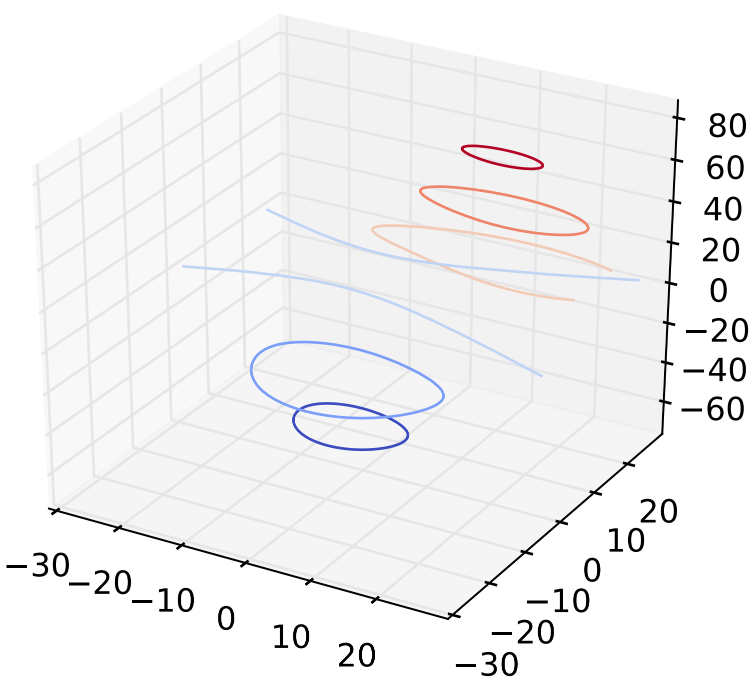



```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for c, z in zip(['r', 'g', 'b', 'y'], [30, 20, 10, 0]):
    xs = np.arange(20)
    ys = np.random.rand(20)

    # You can provide either a single color or an array. To demonstrate this,
    # the first bar of each set will be colored cyan.
    cs = [c] * len(xs)
    cs[0] = 'c'
    ax.bar(xs, ys, zs=z, zdir='y', color=cs, alpha=0.8)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

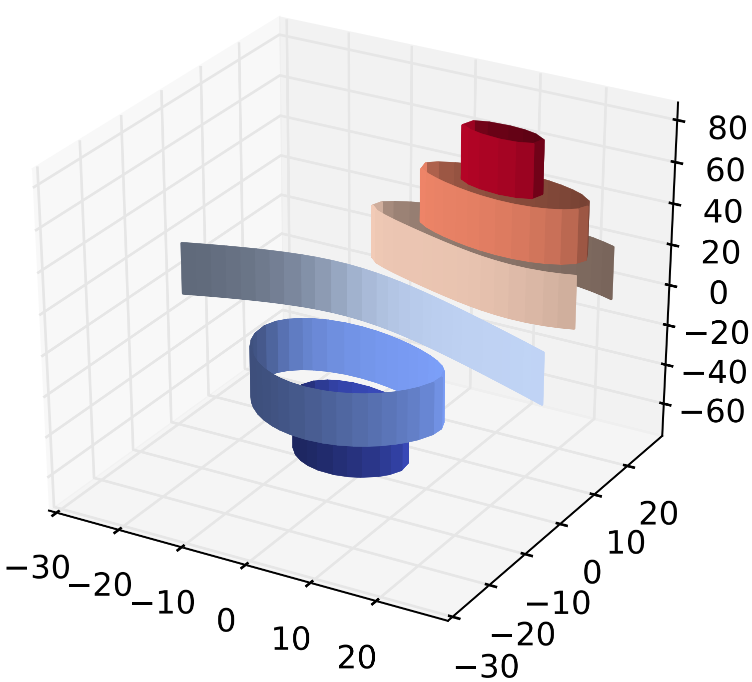
88.3 mplot3d example code: contour3d_demo.py



```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.4 mplot3d example code: contour3d_demo2.py

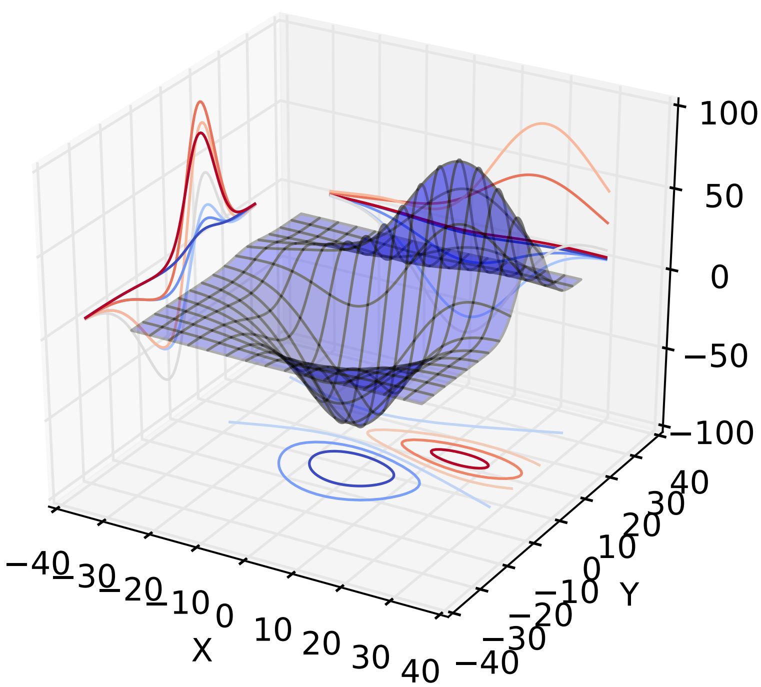



```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
```

```
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, extend3d=True, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.5 mplot3d example code: contour3d_demo3.py

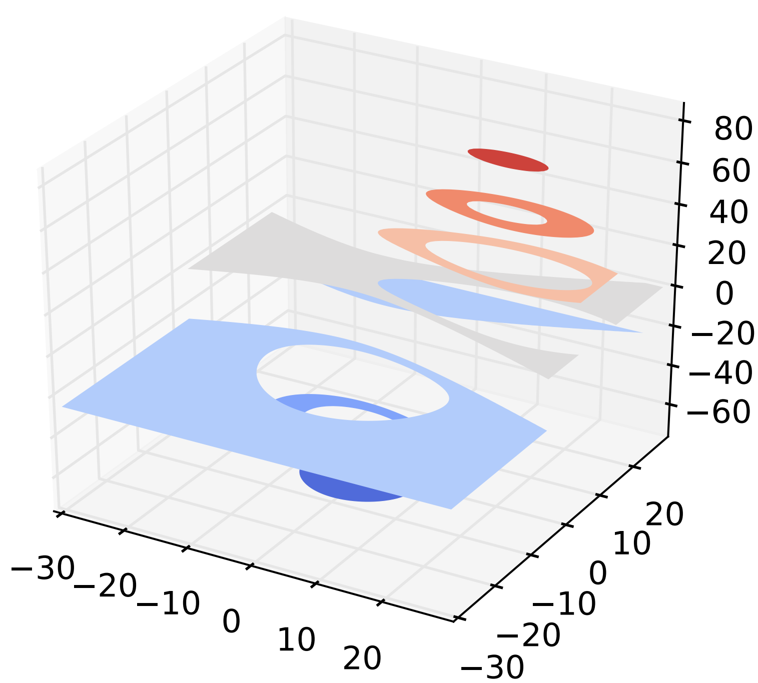



```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contour(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)
```

```
ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.6 mplot3d example code: contourf3d_demo.py

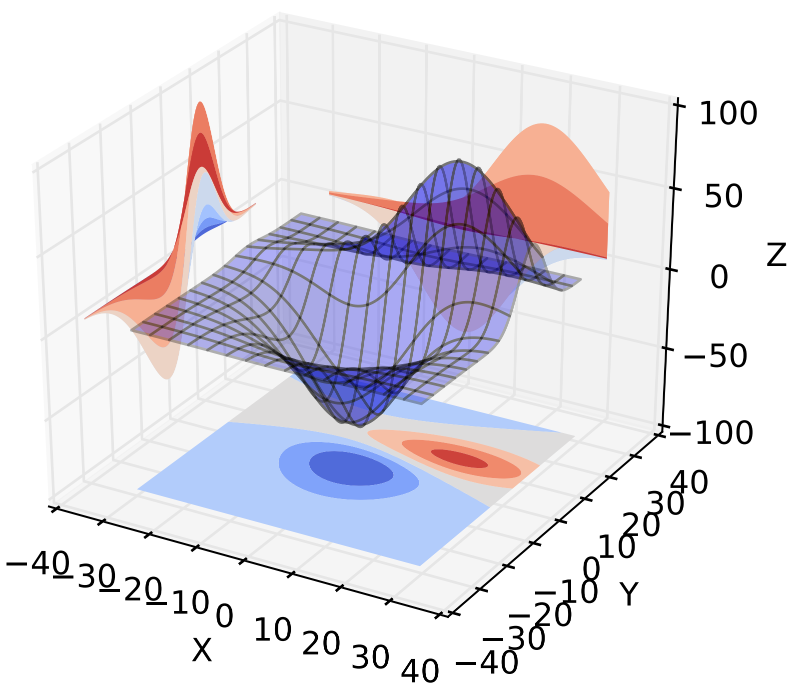



```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contourf(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.7 mplot3d example code: contourf3d_demo2.py

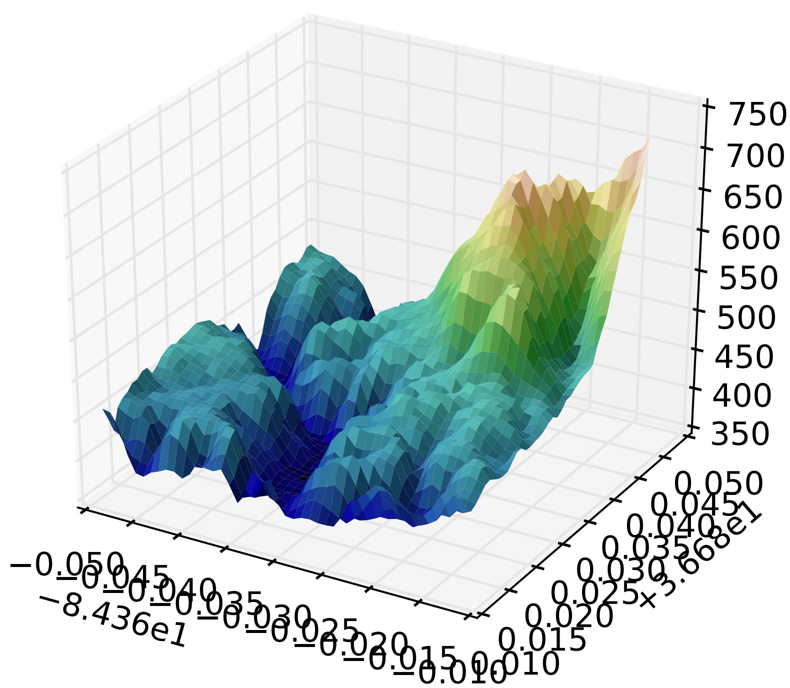



```
"""
.. versionadded:: 1.1.0
   This demo depends on new features added to contourf3d.
"""

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contourf(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
```

```
cset = ax.contourf(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.8 mplot3d example code: custom_shaded_3d_surface.py

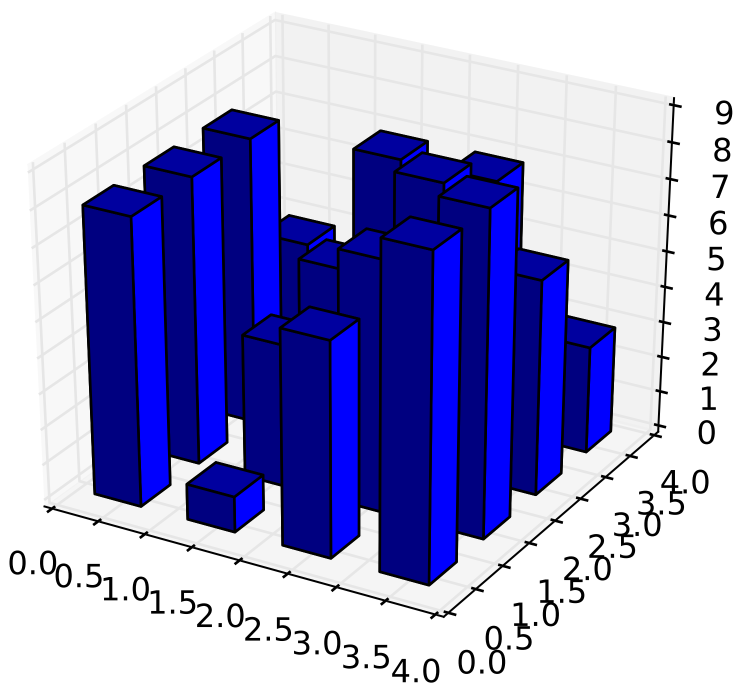



```
"""
Demonstrates using custom hillshading in a 3D surface plot.
"""
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cbook
from matplotlib import cm
from matplotlib.colors import LightSource
```

```
import matplotlib.pyplot as plt
import numpy as np

filename = cbook.get_sample_data('jacksboro_fault_dem.npz', asfileobj=False)
with np.load(filename) as dem:
    z = dem['elevation']
    nrows, ncols = z.shape
    x = np.linspace(dem['xmin'], dem['xmax'], ncols)
    y = np.linspace(dem['ymin'], dem['ymax'], nrows)
    x, y = np.meshgrid(x, y)

region = np.s_[5:50, 5:50]
x, y, z = x[region], y[region], z[region]

fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))

ls = LightSource(270, 45)
# To use a custom hillshading mode, override the built-in shading and pass
# in the rgb colors of the shaded surface calculated from "shade".
rgb = ls.shade(z, cmap=cm.gist_earth, vert_exag=0.1, blend_mode='soft')
surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, facecolors=rgb,
                       linewidth=0, antialiased=False, shade=False)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.9 mplot3d example code: hist3d_demo.py

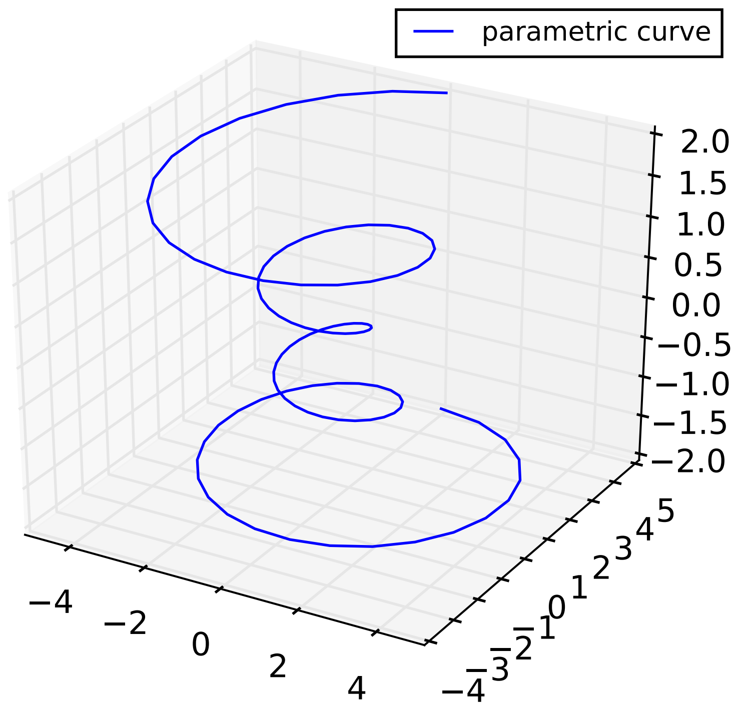



```
'''
Demo of a histogram for 2 dimensional data as a bar graph in 3D.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x, y = np.random.rand(2, 100) * 4
hist, xedges, yedges = np.histogram2d(x, y, bins=4, range=[[0, 4], [0, 4]])

# Construct arrays for the anchor positions of the 16 bars.
# Note: np.meshgrid gives arrays in (ny, nx) so we use 'F' to flatten xpos,
# ypos in column-major order. For numpy >= 1.7, we could instead call meshgrid
# with indexing='ij'.
xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25)
xpos = xpos.flatten('F')
ypos = ypos.flatten('F')
zpos = np.zeros_like(xpos)
```

```
# Construct arrays with the dimensions for the 16 bars.
dx = 0.5 * np.ones_like(zpos)
dy = dx.copy()
dz = hist.flatten()

ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color='b', zsort='average')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.10 mplot3d example code: lines3d_demo.py

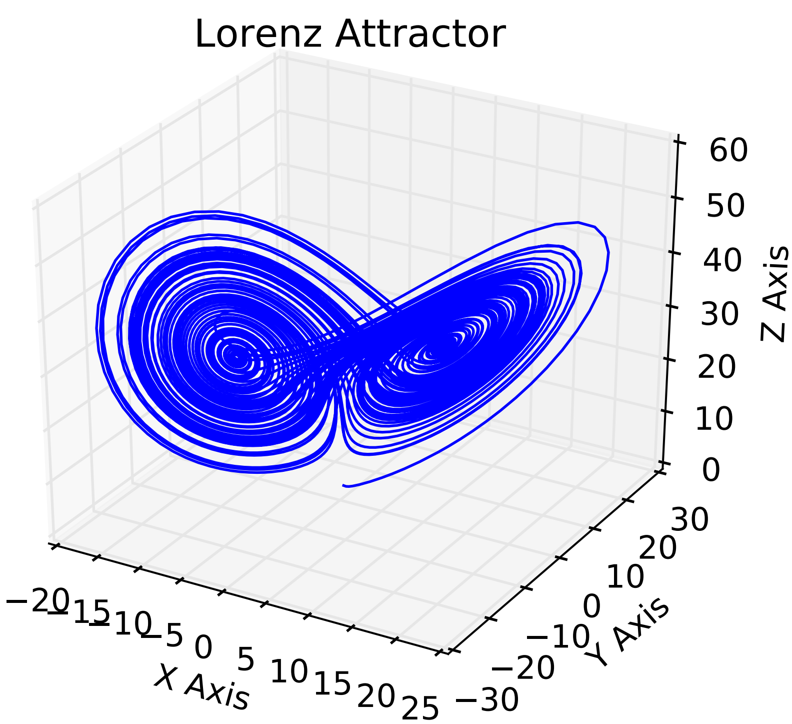



```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt

mpl.rcParams['legend.fontsize'] = 10

fig = plt.figure()
ax = fig.gca(projection='3d')
```

```
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z, label='parametric curve')
ax.legend()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.11 mplot3d example code: lorenz_attractor.py

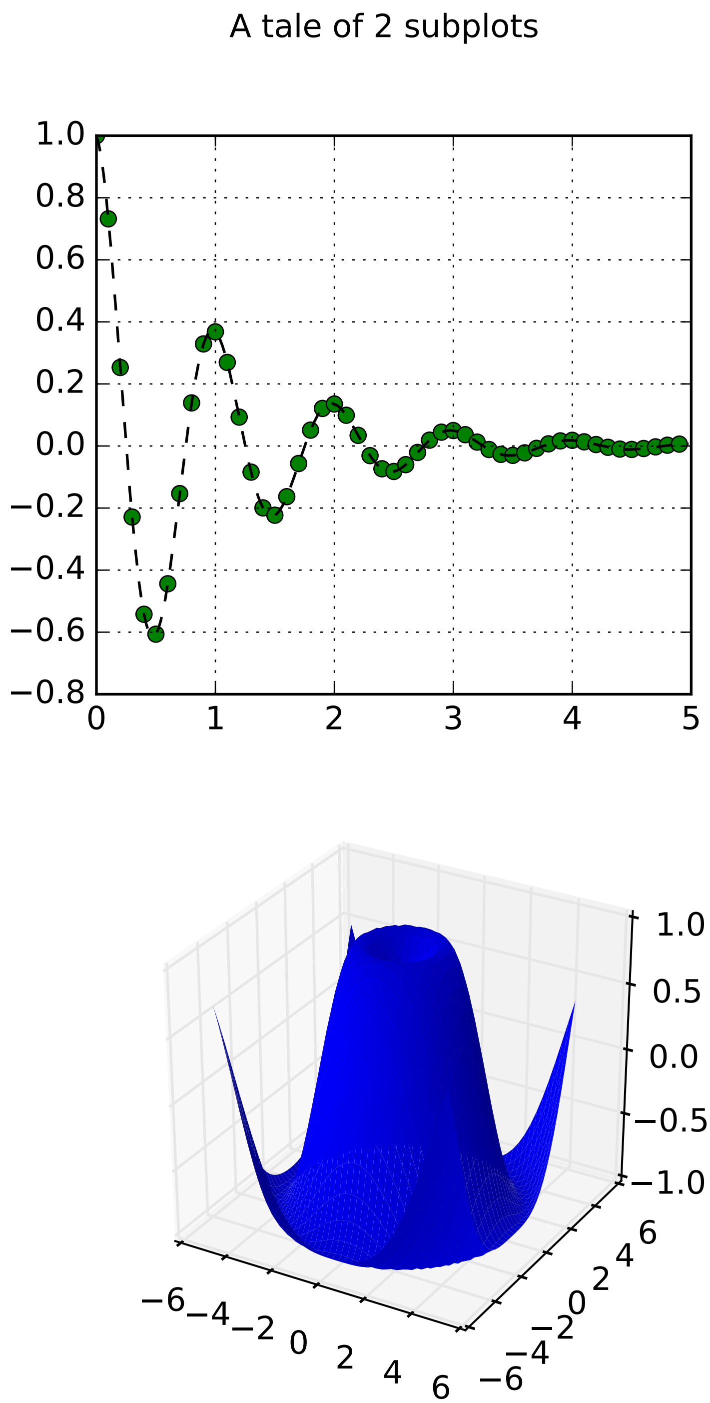



```
# Plot of the Lorenz Attractor based on Edward Lorenz's 1963 "Deterministic
# Nonperiodic Flow" publication.
# http://journals.ametsoc.org/doi/abs/10.1175/1520-0469%281963%29020%3C0130%3ADNF%3E2.0.
↪CO%3B2
#
# Note: Because this is a simple non-linear ODE, it would be more easily
#       done using SciPy's ode solver, but this approach depends only
#       upon NumPy.
```

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def lorenz(x, y, z, s=10, r=28, b=2.667):
    x_dot = s*(y - x)
    y_dot = r*x - y - x*z
    z_dot = x*y - b*z
    return x_dot, y_dot, z_dot


dt = 0.01
stepCnt = 10000

# Need one more for the initial values
xs = np.empty((stepCnt + 1,))
ys = np.empty((stepCnt + 1,))
zs = np.empty((stepCnt + 1,))

# Setting initial values
xs[0], ys[0], zs[0] = (0., 1., 1.05)

# Stepping through "time".
for i in range(stepCnt):
    # Derivatives of the X, Y, Z state
    x_dot, y_dot, z_dot = lorenz(xs[i], ys[i], zs[i])
    xs[i + 1] = xs[i] + (x_dot * dt)
    ys[i + 1] = ys[i] + (y_dot * dt)
    zs[i + 1] = zs[i] + (z_dot * dt)

fig = plt.figure()
ax = fig.gca(projection='3d')

ax.plot(xs, ys, zs)
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("Lorenz Attractor")

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

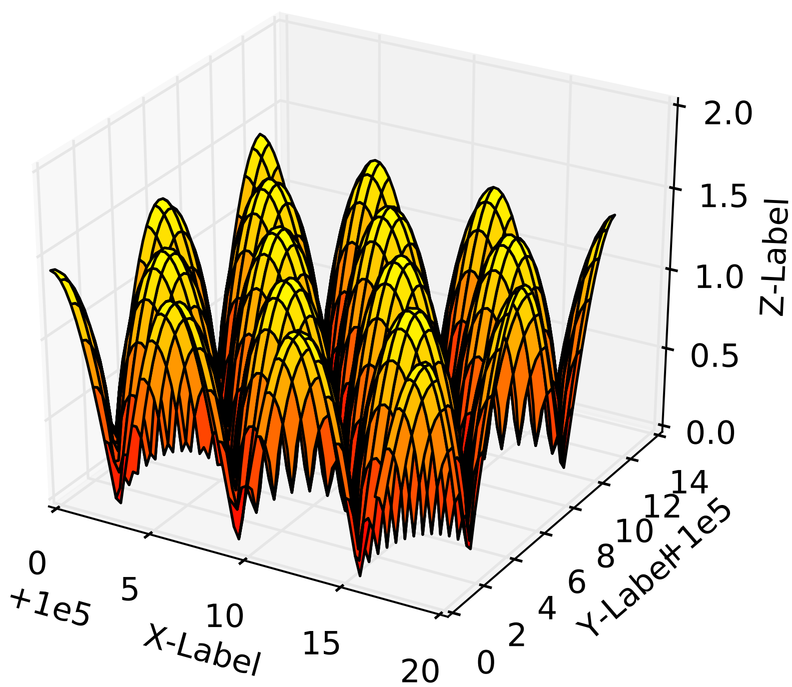
88.12 mplot3d example code: mixed_subplots_demo.py



```
"""
Demonstrate the mixing of 2d and 3d subplots
"""
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np


def f(t):
    s1 = np.cos(2*np.pi*t)
    e1 = np.exp(-t)
    return np.multiply(s1, e1)


################
# First subplot
################
t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)

# Twice as tall as it is wide.
fig = plt.figure(figsize=plt.figaspect(2.))
fig.suptitle('A tale of 2 subplots')
ax = fig.add_subplot(2, 1, 1)
l = ax.plot(t1, f(t1), 'bo',
            t2, f(t2), 'k--', markerfacecolor='green')
ax.grid(True)
ax.set_ylabel('Damped oscillation')


#################
# Second subplot
#################
ax = fig.add_subplot(2, 1, 2, projection='3d')
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                       linewidth=0, antialiased=False)

ax.set_zlim3d(-1, 1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.13 mplot3d example code: offset_demo.py

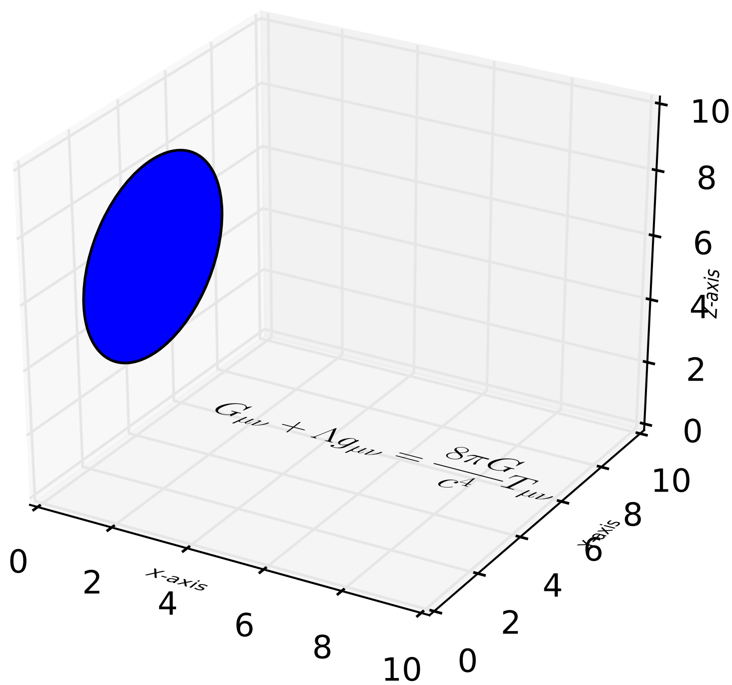



```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

# This example demonstrates mplot3d's offset text display.
# As one rotates the 3D figure, the offsets should remain oriented
# same way as the axis label, and should also be located "away"
# from the center of the plot.
#
# This demo triggers the display of the offset text for the x and
# y axis by adding 1e5 to X and Y. Anything less would not
# automatically trigger it.

fig = plt.figure()
ax = fig.gca(projection='3d')
X, Y = np.mgrid[0:6*np.pi:0.25, 0:4*np.pi:0.25]
Z = np.sqrt(np.abs(np.cos(X) + np.cos(Y)))

surf = ax.plot_surface(X + 1e5, Y + 1e5, Z, cmap='autumn', cstride=2, rstride=2)
ax.set_xlabel("X-Label")
ax.set_ylabel("Y-Label")
ax.set_zlabel("Z-Label")
```

```
ax.set_zlim(0, 2)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.14 mplot3d example code: pathpatch3d_demo.py

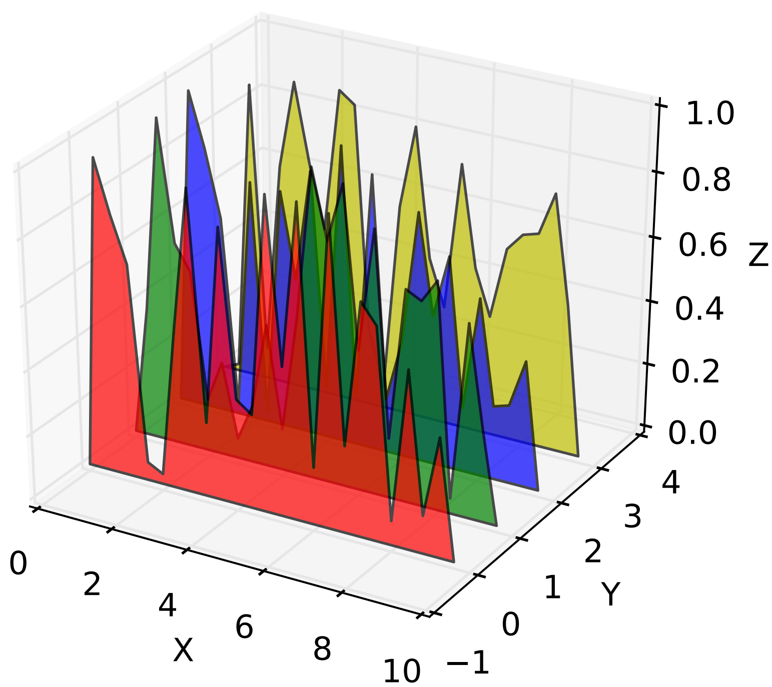



```
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, PathPatch
# register Axes3D class with matplotlib by importing Axes3D
from mpl_toolkits.mplot3d import Axes3D
import mpl_toolkits.mplot3d.art3d as art3d
from matplotlib.text import TextPath
from matplotlib.transforms import Affine2D


def text3d(ax, xyz, s, zdir="z", size=None, angle=0, usetex=False, **kwargs):

    x, y, z = xyz
    if zdir == "y":
        xy1, z1 = (x, z), y
```

```
    elif zdir == "y":
        xy1, z1 = (y, z), x
    else:
        xy1, z1 = (x, y), z

    text_path = TextPath((0, 0), s, size=size, usetex=usetex)
    trans = Affine2D().rotate(angle).translate(xy1[0], xy1[1])

    p1 = PathPatch(trans.transform_path(text_path), **kwargs)
    ax.add_patch(p1)
    art3d.pathpatch_2d_to_3d(p1, z=z1, zdir=zdir)


fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

p = Circle((5, 5), 3)
ax.add_patch(p)
art3d.pathpatch_2d_to_3d(p, z=0, zdir="x")


text3d(ax, (4, -2, 0), "X-axis", zdir="z", size=.5, usetex=False,
       ec="none", fc="k")
text3d(ax, (12, 4, 0), "Y-axis", zdir="z", size=.5, usetex=False,
       angle=.5*3.14159, ec="none", fc="k")
text3d(ax, (12, 10, 4), "Z-axis", zdir="y", size=.5, usetex=False,
       angle=.5*3.14159, ec="none", fc="k")

text3d(ax, (1, 5, 0),
       r"$\displaystyle G_{\mu\nu} + \Lambda g_{\mu\nu} = "
       r"\frac{8\pi G}{c^4} T_{\mu\nu}  $",
       zdir="z", size=1, usetex=True,
       ec="none", fc="k")

ax.set_xlim3d(0, 10)
ax.set_ylim3d(0, 10)
ax.set_zlim3d(0, 10)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.15 mplot3d example code: polys3d_demo.py

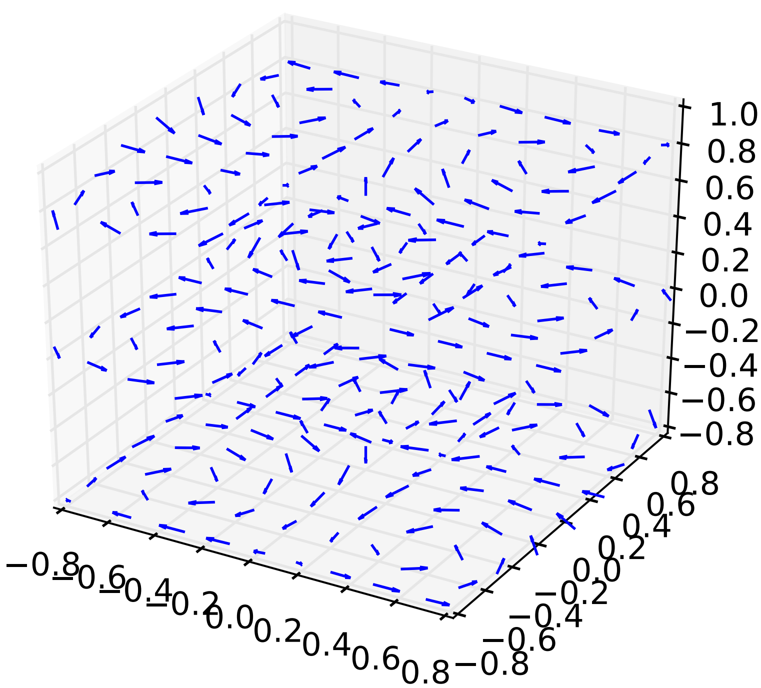



```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
from matplotlib.colors import colorConverter
import matplotlib.pyplot as plt
import numpy as np


fig = plt.figure()
ax = fig.gca(projection='3d')


def cc(arg):
    return colorConverter.to_rgba(arg, alpha=0.6)

xs = np.arange(0, 10, 0.4)
verts = []
zs = [0.0, 1.0, 2.0, 3.0]
for z in zs:
    ys = np.random.rand(len(xs))
    ys[0], ys[-1] = 0, 0
    verts.append(list(zip(xs, ys)))
```

```
poly = PolyCollection(verts, facecolors=[cc('r'), cc('g'), cc('b'),
                                          cc('y')])
poly.set_alpha(0.7)
ax.add_collection3d(poly, zs=zs, zdir='y')

ax.set_xlabel('X')
ax.set_xlim3d(0, 10)
ax.set_ylabel('Y')
ax.set_ylim3d(-1, 4)
ax.set_zlabel('Z')
ax.set_zlim3d(0, 1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.16 mplot3d example code: quiver3d_demo.py

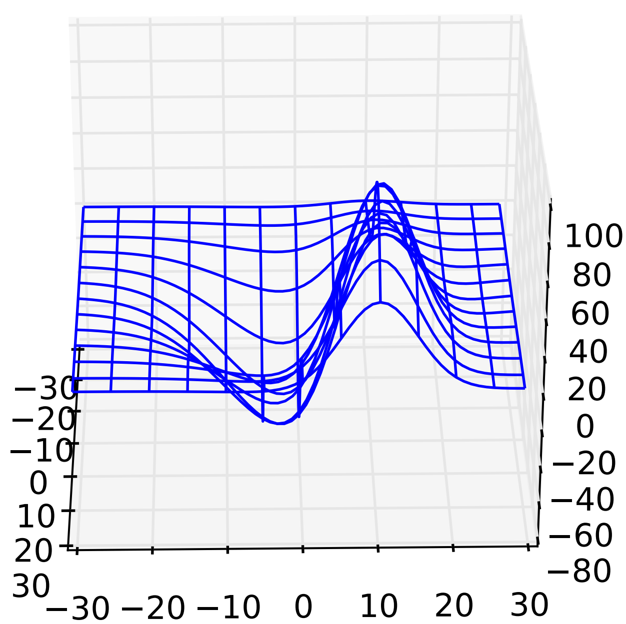



```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
```

```
fig = plt.figure()
ax = fig.gca(projection='3d')

x, y, z = np.meshgrid(np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.8))

u = np.sin(np.pi * x) * np.cos(np.pi * y) * np.cos(np.pi * z)
v = -np.cos(np.pi * x) * np.sin(np.pi * y) * np.cos(np.pi * z)
w = (np.sqrt(2.0 / 3.0) * np.cos(np.pi * x) * np.cos(np.pi * y) *
     np.sin(np.pi * z))

ax.quiver(x, y, z, u, v, w, length=0.1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.17 mplot3d example code: rotate_axes3d_demo.py

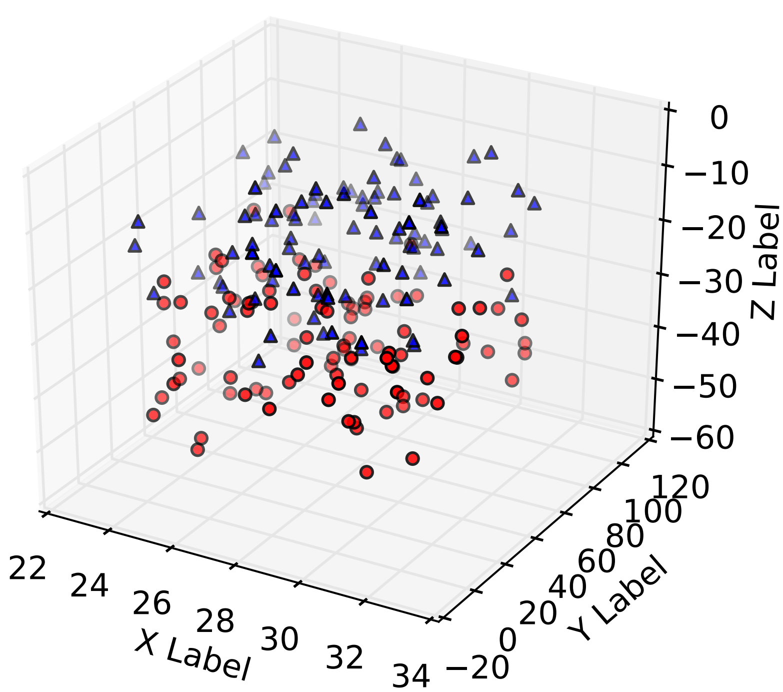



```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
```

```
import numpy as np


fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.1)
ax.plot_wireframe(X, Y, Z, rstride=5, cstride=5)

for angle in range(0, 360):
    ax.view_init(30, angle)
    plt.draw()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.18 mplot3d example code: scatter3d_demo.py

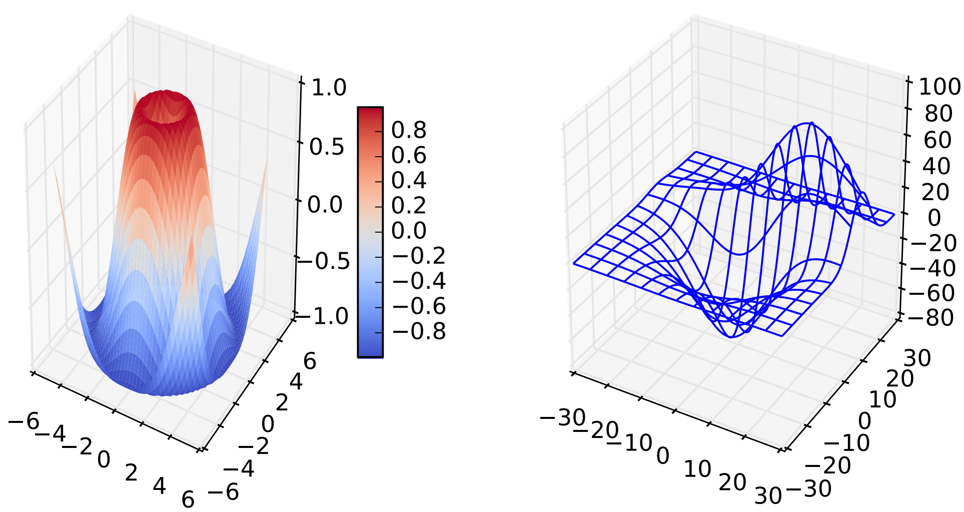



```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt


def randrange(n, vmin, vmax):
```

```
    return (vmax - vmin)*np.random.rand(n) + vmin

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
n = 100
for c, m, zl, zh in [('r', 'o', -50, -25), ('b', '^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zl, zh)
    ax.scatter(xs, ys, zs, c=c, marker=m)

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.19 mplot3d example code: subplot3d_demo.py

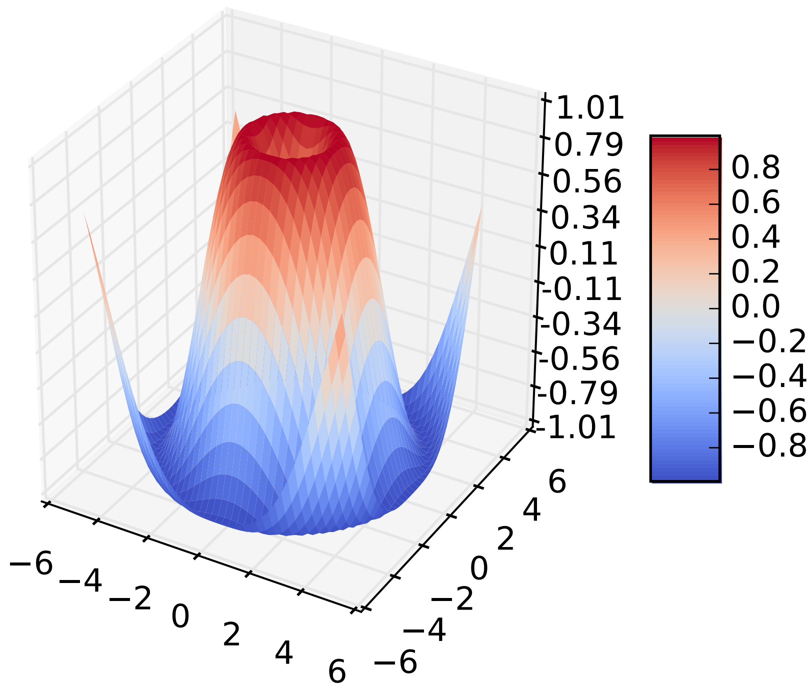



```
from mpl_toolkits.mplot3d.axes3d import Axes3D
import matplotlib.pyplot as plt


# imports specific to the plots in this example
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d.axes3d import get_test_data

# Twice as wide as it is tall.
```

```
fig = plt.figure(figsize=plt.figaspect(0.5))

#---- First subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
ax.set_zlim3d(-1.01, 1.01)

fig.colorbar(surf, shrink=0.5, aspect=10)

#---- Second subplot
ax = fig.add_subplot(1, 2, 2, projection='3d')
X, Y, Z = get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.20 mplot3d example code: surface3d_demo.py

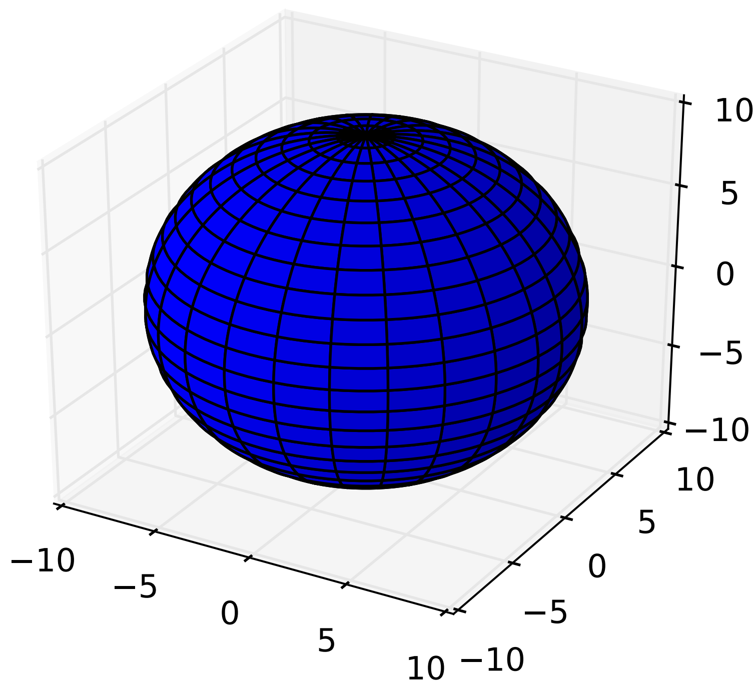



```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
ax.set_zlim(-1.01, 1.01)

ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

fig.colorbar(surf, shrink=0.5, aspect=5)
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

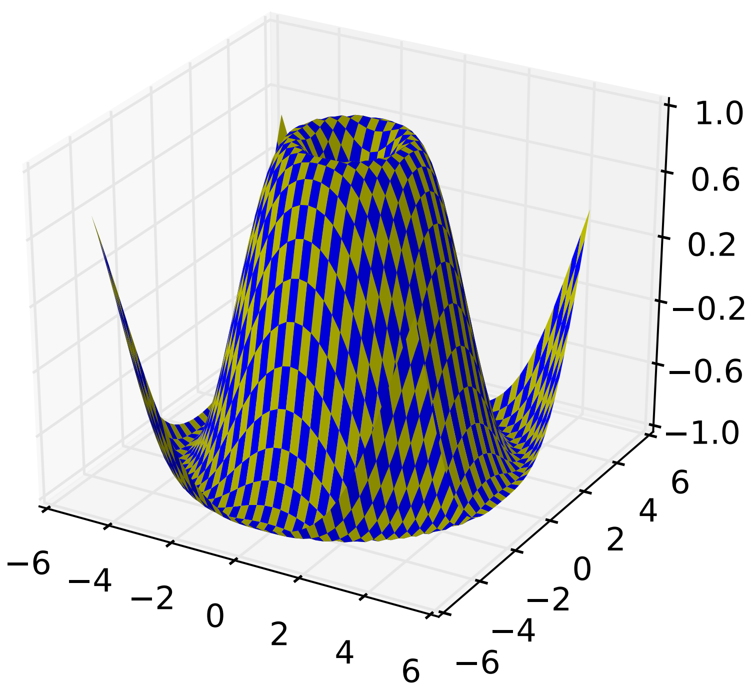
88.21 mplot3d example code: surface3d_demo2.py




```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)

x = 10 * np.outer(np.cos(u), np.sin(v))
y = 10 * np.outer(np.sin(u), np.sin(v))
z = 10 * np.outer(np.ones(np.size(u)), np.cos(v))
ax.plot_surface(x, y, z, rstride=4, cstride=4, color='b')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.22 mplot3d example code: surface3d_demo3.py

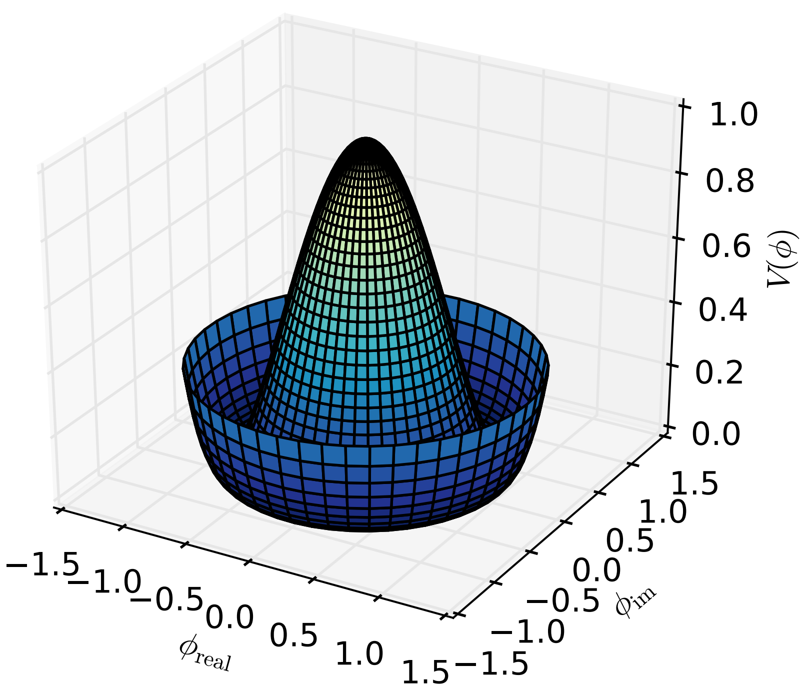



```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

colortuple = ('y', 'b')
colors = np.empty(X.shape, dtype=str)
for y in range(ylen):
```

```
    for x in range(xlen):
        colors[x, y] = colortuple[(x + y) % len(colortuple)]

surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, facecolors=colors,
                       linewidth=0, antialiased=False)

ax.set_zlim3d(-1, 1)
ax.w_zaxis.set_major_locator(LinearLocator(6))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.23 mplot3d example code: surface3d_radial_demo.py

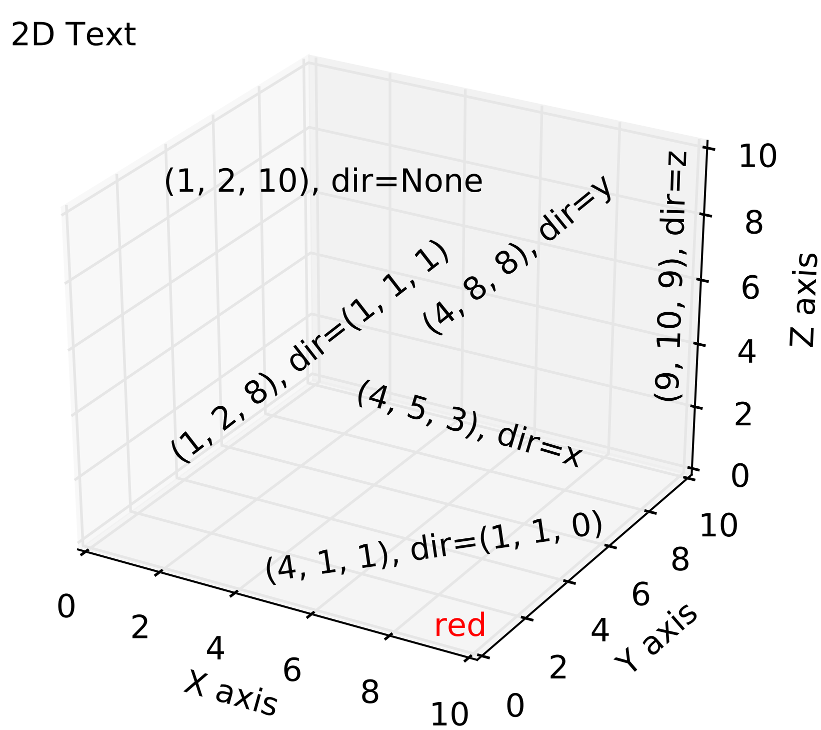



```
# By Armin Moser

from mpl_toolkits.mplot3d import Axes3D
import matplotlib
import numpy as np
from matplotlib import cm
from matplotlib import pyplot as plt
```

```
step = 0.04
maxval = 1.0
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# create supporting points in polar coordinates
r = np.linspace(0, 1.25, 50)
p = np.linspace(0, 2*np.pi, 50)
R, P = np.meshgrid(r, p)
# transform them to cartesian system
X, Y = R*np.cos(P), R*np.sin(P)

Z = ((R**2 - 1)**2)
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.YlGnBu_r)
ax.set_zlim3d(0, 1)
ax.set_xlabel(r'$\phi_\mathrm{real}$')
ax.set_ylabel(r'$\phi_\mathrm{im}$')
ax.set_zlabel(r'$V(\phi)$')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

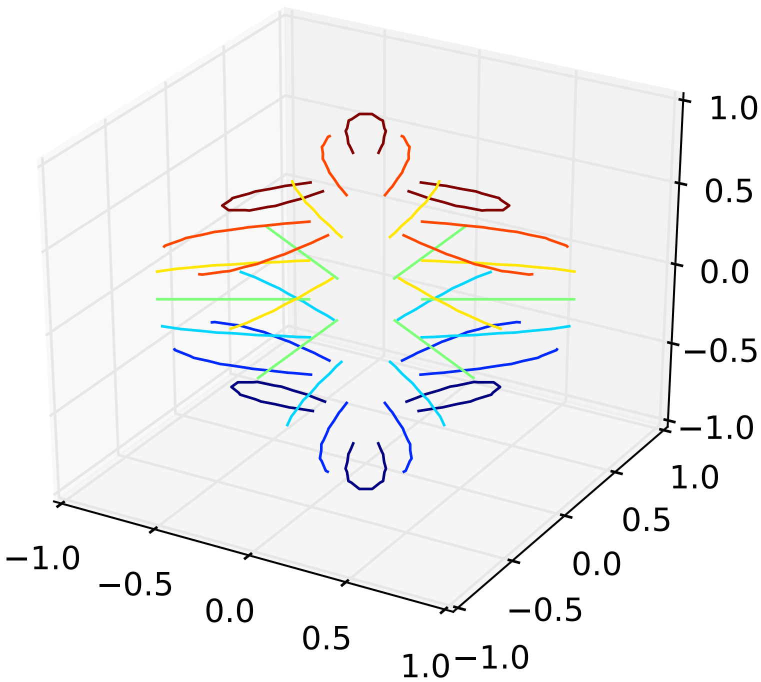
88.24 mplot3d example code: text3d_demo.py



```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.gca(projection='3d')

zdirs = (None, 'x', 'y', 'z', (1, 1, 0), (1, 1, 1))
xs = (1, 4, 4, 9, 4, 1)
ys = (2, 5, 8, 10, 1, 2)
zs = (10, 3, 8, 9, 1, 8)

for zdir, x, y, z in zip(zdirs, xs, ys, zs):
    label = '(%d, %d, %d), dir=%s' % (x, y, z, zdir)
    ax.text(x, y, z, label, zdir)

ax.text(9, 0, 0, "red", color='red')
ax.text2D(0.05, 0.95, "2D Text", transform=ax.transAxes)

ax.set_xlim3d(0, 10)
ax.set_ylim3d(0, 10)
ax.set_zlim3d(0, 10)

ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.25 mplot3d example code: tricontour3d_demo.py

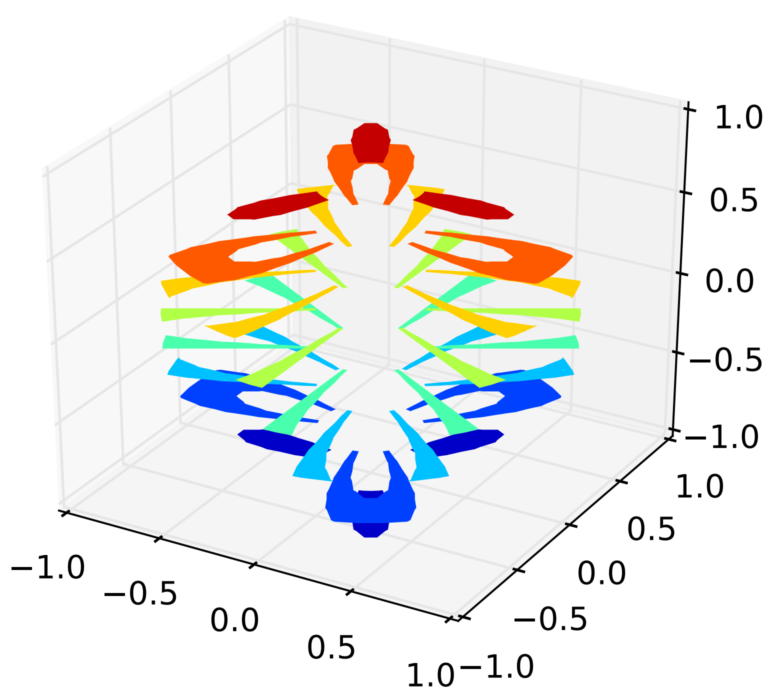



```
"""
Contour plots of unstructured triangular grids.
"""
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as tri
import numpy as np
import math

# First create the x and y coordinates of the points.
n_angles = 48
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()
```

```
# Create a custom triangulation
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

plt.figure()
plt.gca(projection='3d')
plt.tricontour(triang, z)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.26 mplot3d example code: tricontourf3d_demo.py

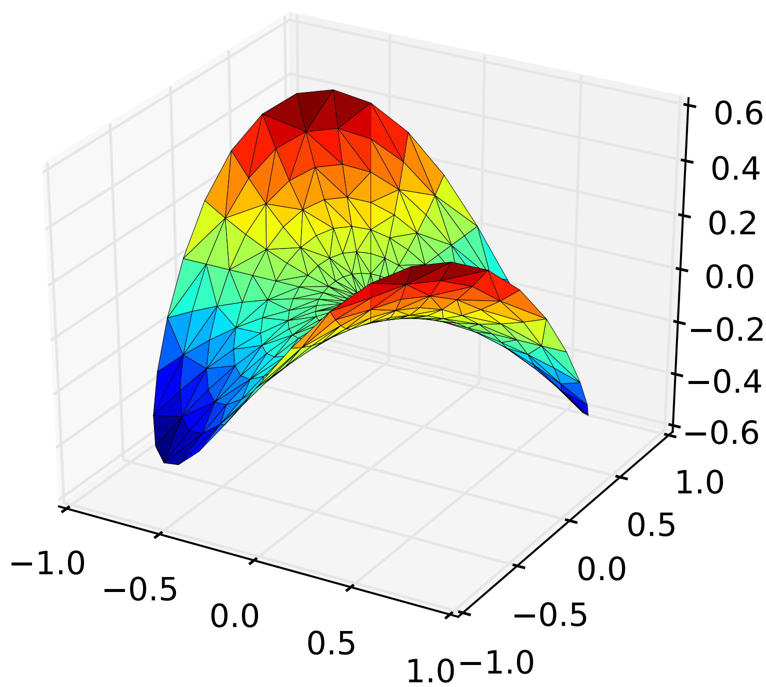



```
"""
Contour plots of unstructured triangular grids.
"""
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as tri
import numpy as np
import math

# First create the x and y coordinates of the points.
n_angles = 48
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create a custom triangulation
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

plt.figure()
plt.gca(projection='3d')
plt.tricontourf(triang, z)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.27 mplot3d example code: trisurf3d_demo.py

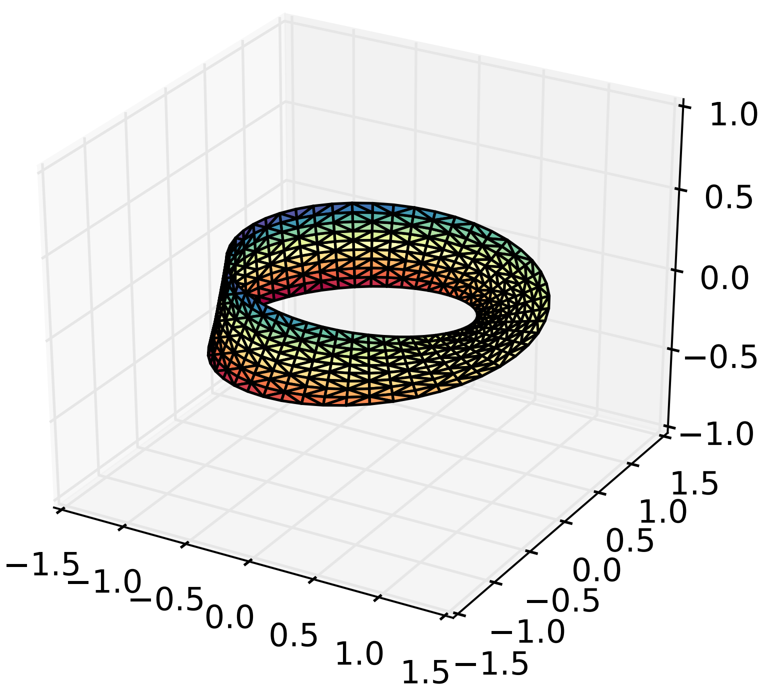


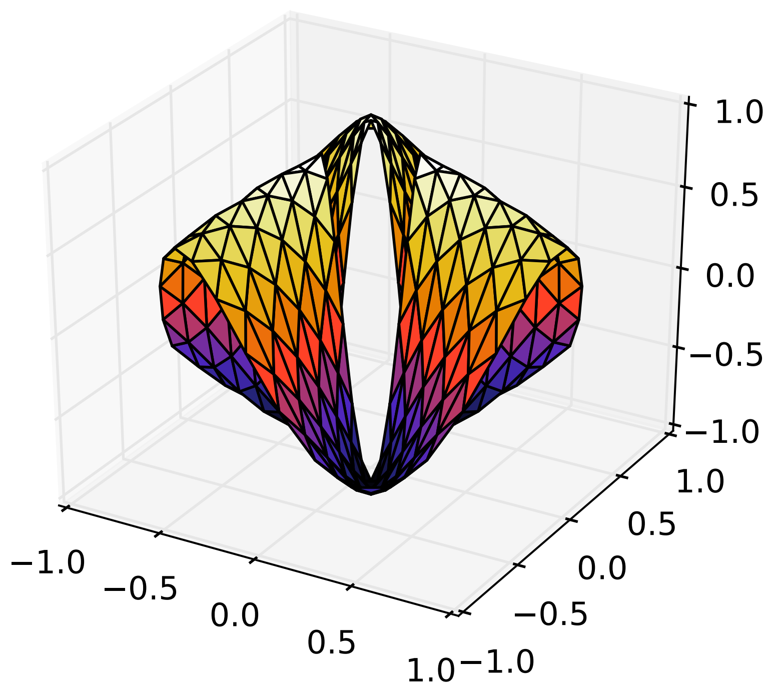

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np

n_angles = 36
n_radii = 8

# An array of radii
# Does not include radius r=0, this is to eliminate duplicate points
radii = np.linspace(0.125, 1.0, n_radii)

# An array of angles
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)

# Repeat all angles for each radius
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)

# Convert polar (radii, angles) coords to cartesian (x, y) coords
# (0, 0) is added here. There are no duplicate points in the (x, y) plane
x = np.append(0, (radii*np.cos(angles)).flatten())
y = np.append(0, (radii*np.sin(angles)).flatten())
```

```
# Pringle surface
z = np.sin(-x*y)

fig = plt.figure()
ax = fig.gca(projection='3d')

ax.plot_trisurf(x, y, z, cmap=cm.jet, linewidth=0.2)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.28 mplot3d example code: trisurf3d_demo2.py



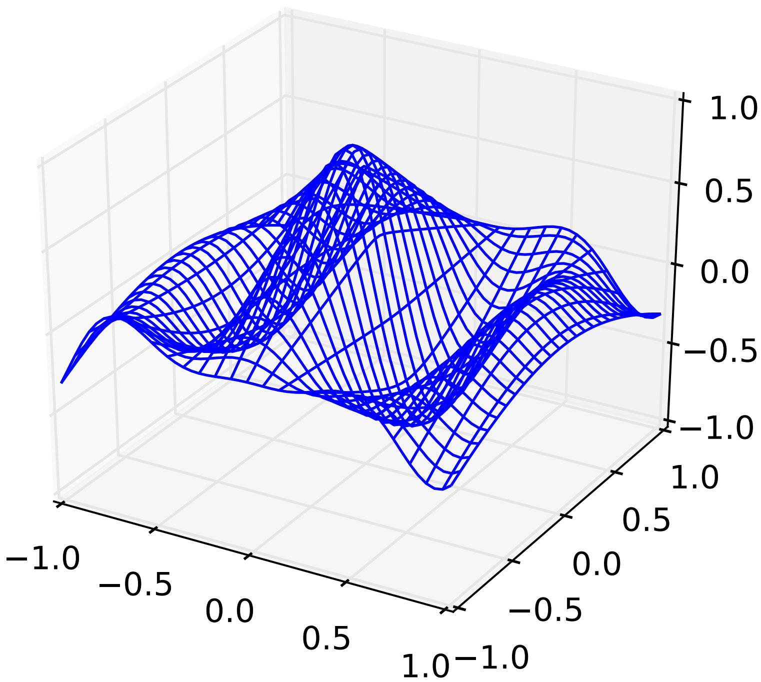



```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as mtri

# u, v are parameterisation variables
u = (np.linspace(0, 2.0 * np.pi, endpoint=True, num=50) * np.ones((10, 1))).flatten()
v = np.repeat(np.linspace(-0.5, 0.5, endpoint=True, num=10), repeats=50).flatten()

# This is the Mobius mapping, taking a u, v pair and returning an x, y, z
# triple
x = (1 + 0.5 * v * np.cos(u / 2.0)) * np.cos(u)
y = (1 + 0.5 * v * np.cos(u / 2.0)) * np.sin(u)
z = 0.5 * v * np.sin(u / 2.0)

# Triangulate parameter space to determine the triangles
tri = mtri.Triangulation(u, v)

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection='3d')

# The triangles in parameter space determine which x, y, z points are
# connected by an edge
ax.plot_trisurf(x, y, z, triangles=tri.triangles, cmap=plt.cm.Spectral)

ax.set_zlim(-1, 1)
```

```
# First create the x and y coordinates of the points.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = mtri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# tripcolor plot.
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection='3d')
ax.plot_trisurf(triang, z, cmap=plt.cm.CMRmap)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.29 mplot3d example code: wire3d_animation_demo.py

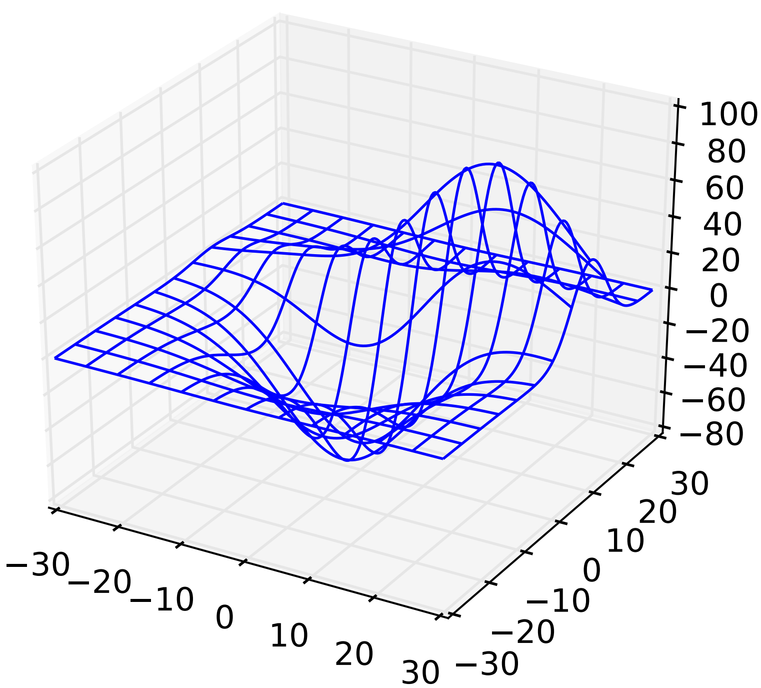



```
from __future__ import print_function
"""
A very simple 'animation' of a 3D plot
"""
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
import time


def generate(X, Y, phi):
    R = 1 - np.sqrt(X**2 + Y**2)
    return np.cos(2 * np.pi * X + phi) * R

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

xs = np.linspace(-1, 1, 50)
ys = np.linspace(-1, 1, 50)
X, Y = np.meshgrid(xs, ys)
Z = generate(X, Y, 0.0)
```

```
wframe = None
tstart = time.time()
for phi in np.linspace(0, 360 / 2 / np.pi, 100):

    oldcol = wframe

    Z = generate(X, Y, phi)
    wframe = ax.plot_wireframe(X, Y, Z, rstride=2, cstride=2)

    # Remove old line collection before drawing
    if oldcol is not None:
        ax.collections.remove(oldcol)

    plt.pause(.001)

print('FPS: %f' % (100 / (time.time() - tstart)))
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

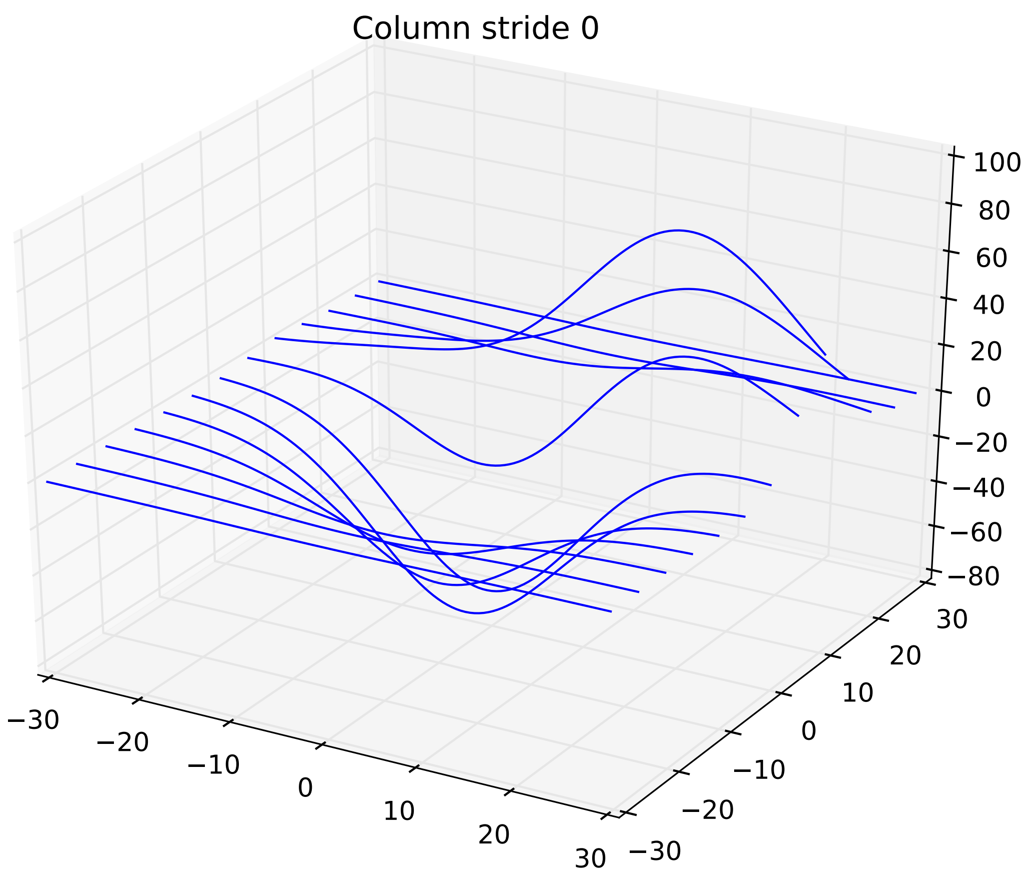
88.30 mplot3d example code: wire3d_demo.py

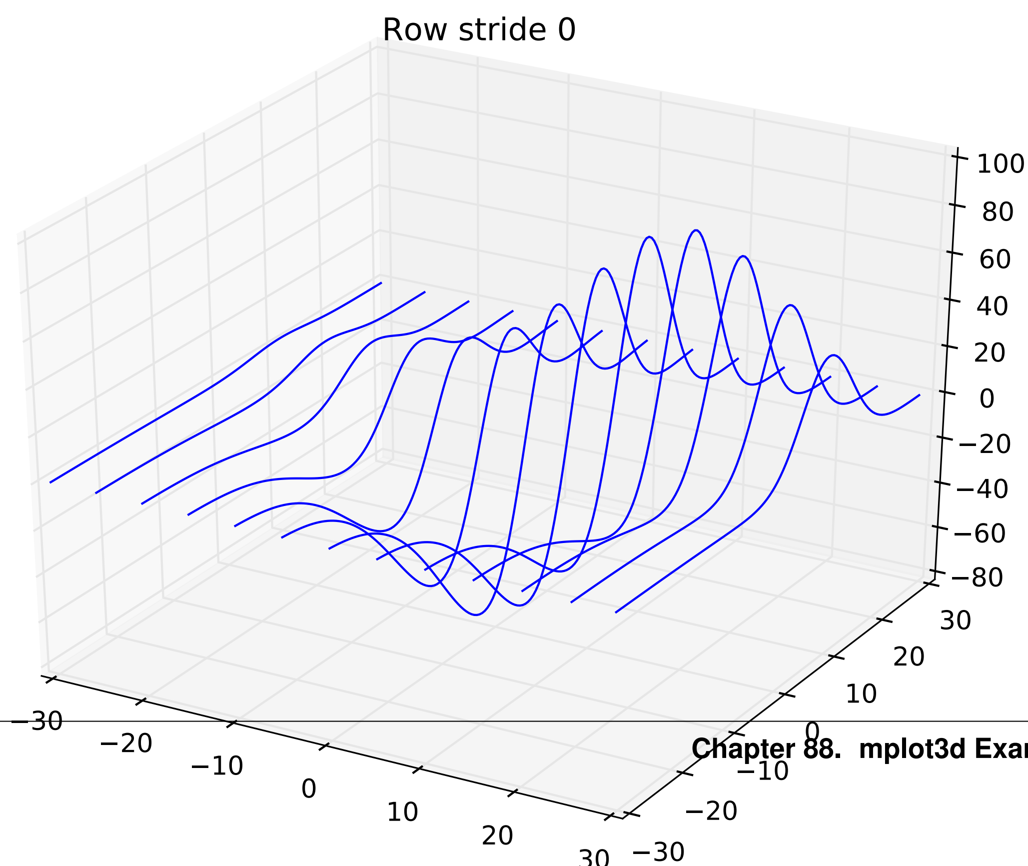


```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

88.31 mplot3d example code: wire3d_zero_stride.py





```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig, [ax1, ax2] = plt.subplots(2, 1, figsize=(8, 12), subplot_kw={'projection': '3d'})
X, Y, Z = axes3d.get_test_data(0.05)
ax1.plot_wireframe(X, Y, Z, rstride=10, cstride=0)
ax1.set_title("Column stride 0")
ax2.plot_wireframe(X, Y, Z, rstride=0, cstride=10)
ax2.set_title("Row stride 0")
plt.tight_layout()
plt.show()
```

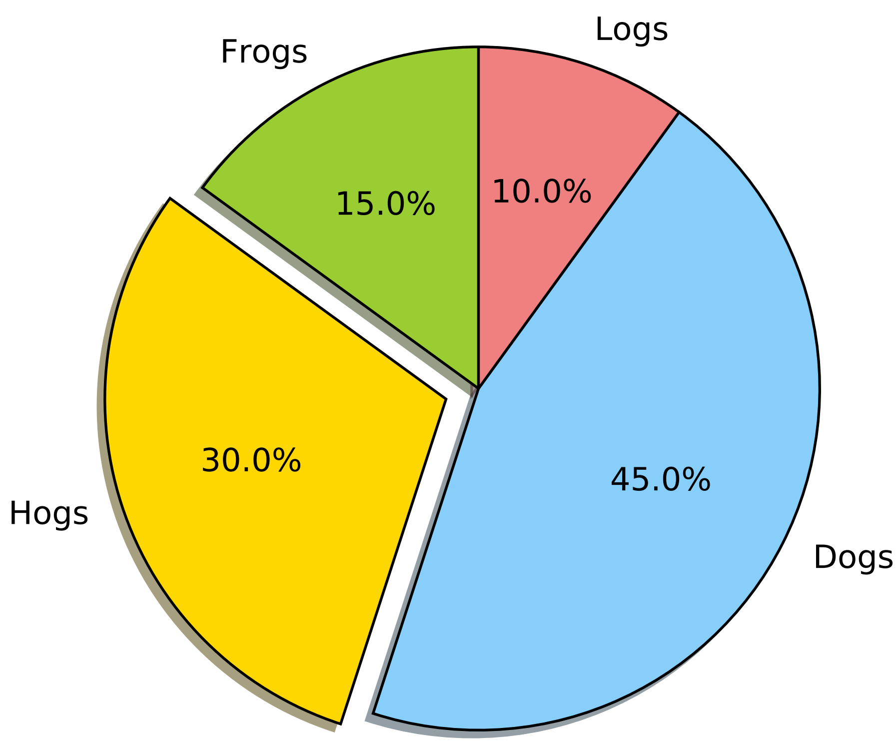
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

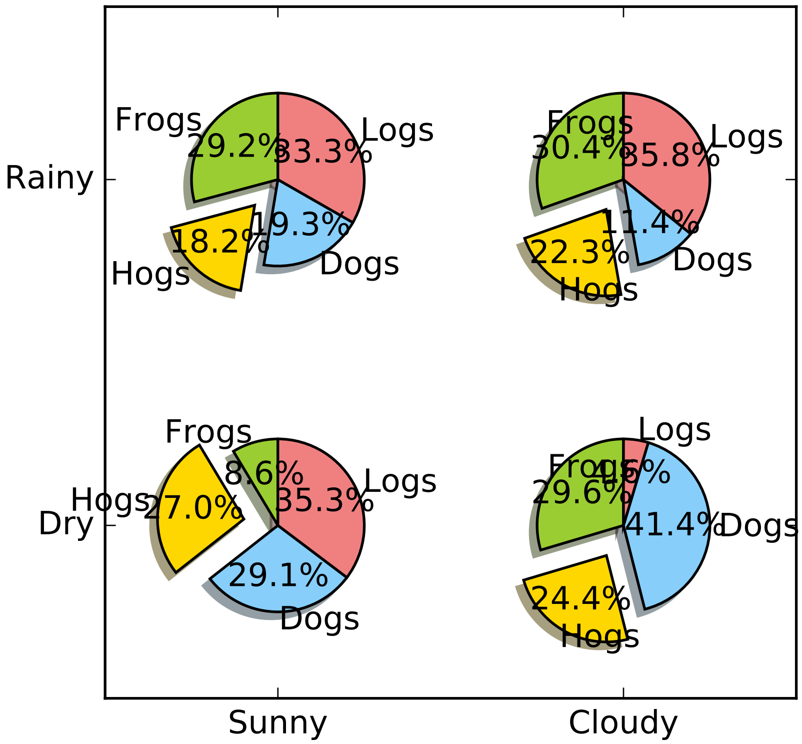
CHAPTER

EIGHTYNINE

PIE_AND_POLAR_CHARTS EXAMPLES

89.1 pie_and_polar_charts example code: pie_demo_features.py



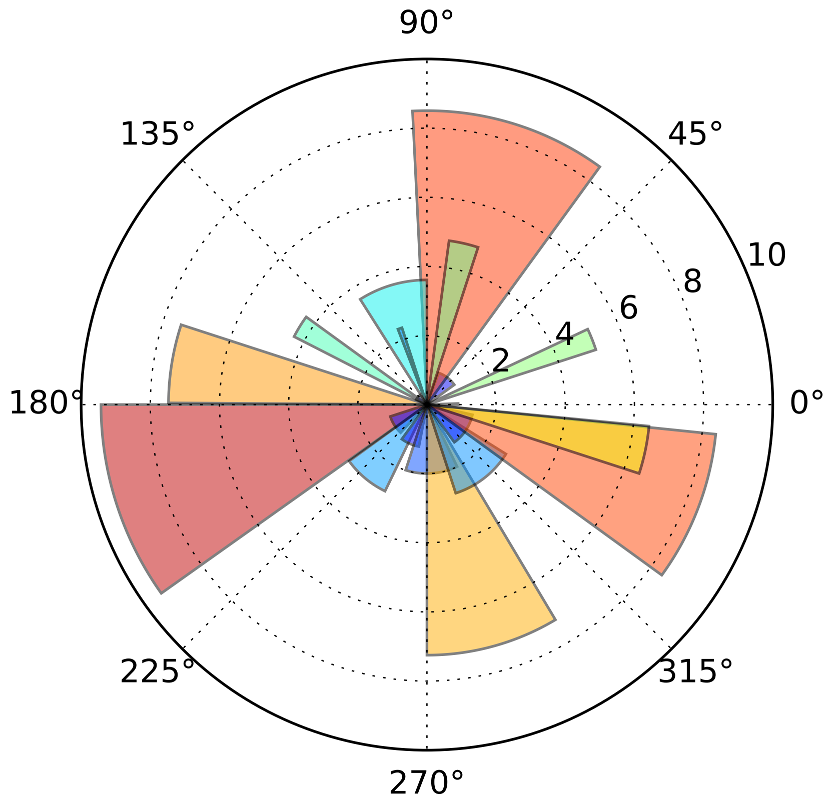



```
"""
Demo of a basic pie chart plus a few additional features.

In addition to the basic pie chart, this demo shows a few optional features:

    * slice labels
    * auto-labeling the percentage
    * offsetting a slice with "explode"
    * drop-shadow
    * custom start angle

Note about the custom start angle:

The default ``startangle`` is 0, which would start the "Frogs" slice on the
positive x-axis. This example sets ``startangle = 90`` such that everything is
rotated counter-clockwise by 90 degrees, and the frog slice starts on the
positive y-axis.
"""
import matplotlib.pyplot as plt

# The slices will be ordered and plotted counter-clockwise.
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')
```

```
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='%1.1f%%', shadow=True, startangle=90)
# Set aspect ratio to be equal so that pie is drawn as a circle.
plt.axis('equal')

fig = plt.figure()
ax = fig.gca()
import numpy as np

ax.pie(np.random.random(4), explode=explode, labels=labels, colors=colors,
       autopct='%1.1f%%', shadow=True, startangle=90,
       radius=0.25, center=(0, 0), frame=True)
ax.pie(np.random.random(4), explode=explode, labels=labels, colors=colors,
       autopct='%1.1f%%', shadow=True, startangle=90,
       radius=0.25, center=(1, 1), frame=True)
ax.pie(np.random.random(4), explode=explode, labels=labels, colors=colors,
       autopct='%1.1f%%', shadow=True, startangle=90,
       radius=0.25, center=(0, 1), frame=True)
ax.pie(np.random.random(4), explode=explode, labels=labels, colors=colors,
       autopct='%1.1f%%', shadow=True, startangle=90,
       radius=0.25, center=(1, 0), frame=True)

ax.set_xticks([0, 1])
ax.set_yticks([0, 1])
ax.set_xticklabels(["Sunny", "Cloudy"])
ax.set_yticklabels(["Dry", "Rainy"])
ax.set_xlim((-0.5, 1.5))
ax.set_ylim((-0.5, 1.5))

# Set aspect ratio to be equal so that pie is drawn as a circle.
ax.set_aspect('equal')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

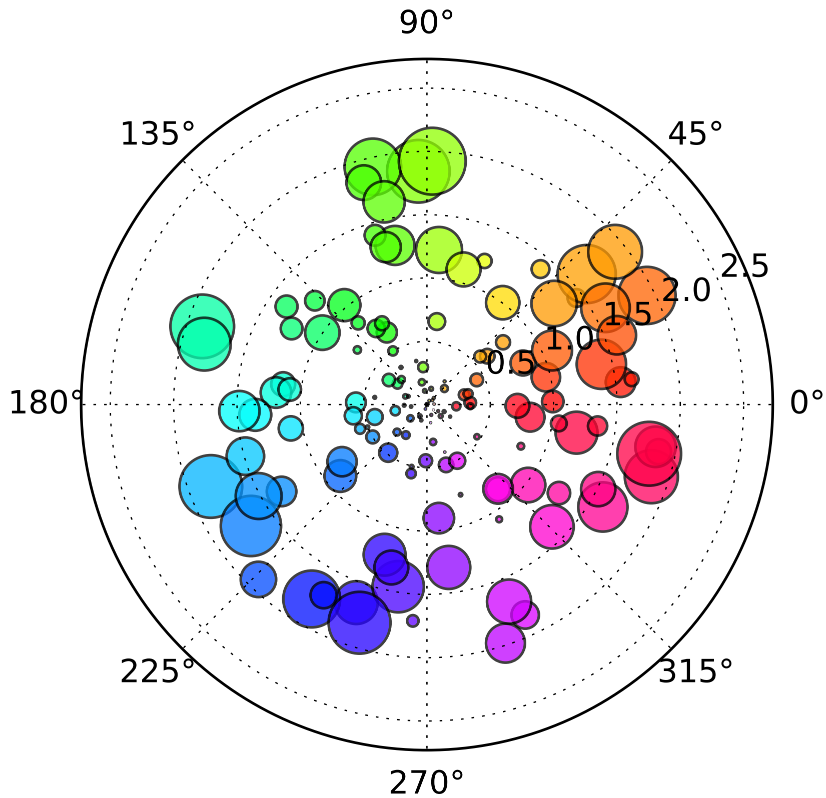
89.2 pie_and_polar_charts example code: polar_bar_demo.py




```
"""
Demo of bar plot on a polar axis.
"""
import numpy as np
import matplotlib.pyplot as plt


N = 20
theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)

ax = plt.subplot(111, projection='polar')
bars = ax.bar(theta, radii, width=width, bottom=0.0)

# Use custom colors and opacity
for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

89.3 pie_and_polar_charts example code: polar_scatter_demo.py




```
"""
Demo of scatter plot on a polar axis.

Size increases radially in this example and color increases with angle (just to
verify the symbols are being scattered correctly).
"""
import numpy as np
import matplotlib.pyplot as plt


N = 150
r = 2 * np.random.rand(N)
theta = 2 * np.pi * np.random.rand(N)
area = 200 * r**2 * np.random.rand(N)
colors = theta

ax = plt.subplot(111, projection='polar')
c = plt.scatter(theta, r, c=colors, s=area, cmap=plt.cm.hsv)
c.set_alpha(0.75)
```

```
plt.show()
```

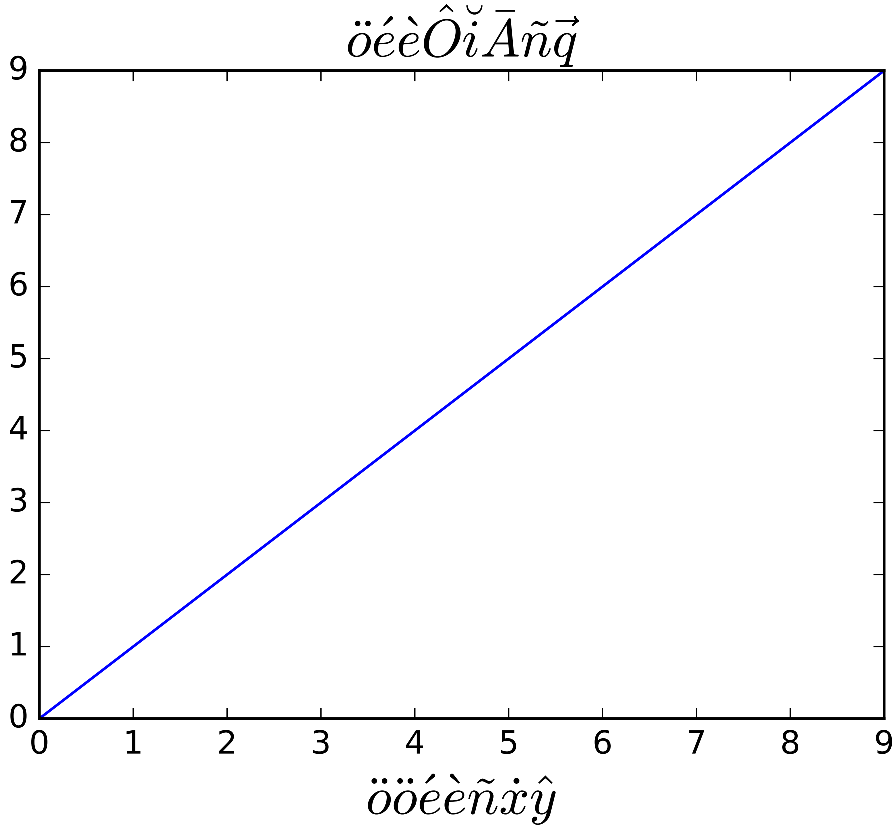
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

NINETY

PYLAB_EXAMPLES EXAMPLES

90.1 pylab_examples example code: accented_text.py

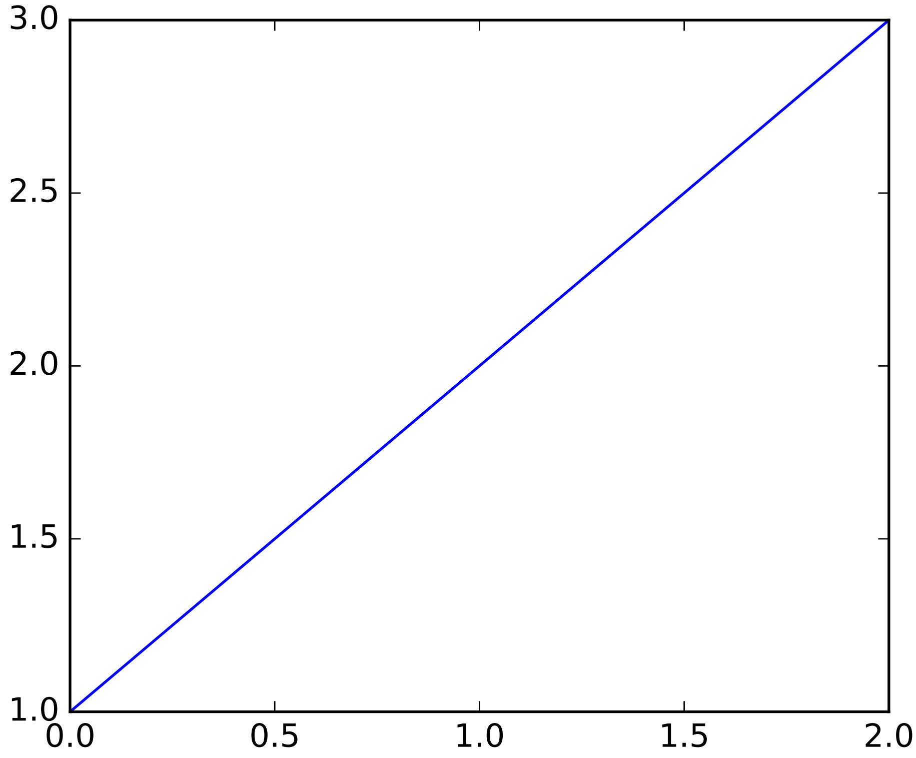



```
#!/usr/bin/env python
"""
matplotlib supports accented characters via TeX mathtext

The following accents are provided: \hat, \breve, \grave, \bar,
\acute, \tilde, \vec, \dot, \ddot.  All of them have the same syntax,
e.g., to make an overbar you do \bar{o} or to make an o umlaut you do
\ddot{o}.  The shortcuts are also provided, e.g.,: \"o \'e \`e \~n \.x
\^y
```

```
"""
import matplotlib.pyplot as plt

plt.axes([0.1, 0.15, 0.8, 0.75])
plt.plot(range(10))

plt.title(r'$\ddot{o}\acute{e}\grave{e}\hat{O}\breve{i}\bar{A}\tilde{n}\vec{q}$',␣
↪fontsize=20)
# shorthand is also supported and curly's are optional
plt.xlabel(r"""$\"o\ddot o \'e\`e\~n\.x\^y$""", fontsize=20)


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.2 pylab_examples example code: agg_buffer.py

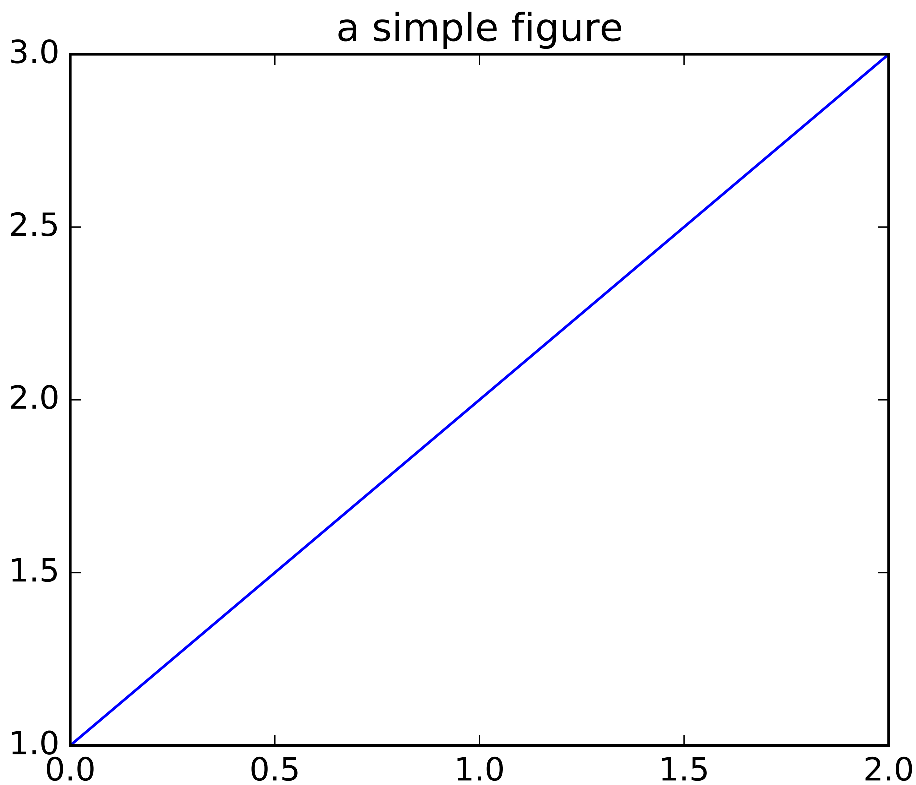


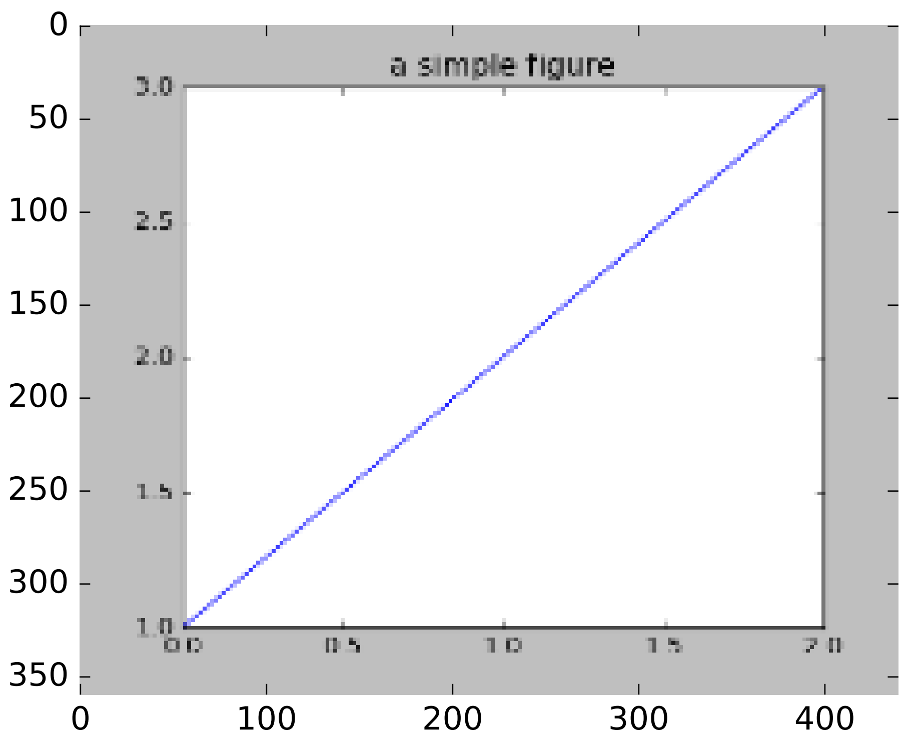

```
#!/usr/bin/env python
"""
Use backend agg to access the figure canvas as an RGB string and then
```

```
convert it to an array and pass it to Pillow for rendering.
"""

import numpy as np

import matplotlib.pyplot as plt
from matplotlib.backends.backend_agg import FigureCanvasAgg


try:
    from PIL import Image
except ImportError:
    raise SystemExit("Pillow must be installed to run this example")

plt.plot([1, 2, 3])

canvas = plt.get_current_fig_manager().canvas

agg = canvas.switch_backends(FigureCanvasAgg)
agg.draw()
s = agg.tostring_rgb()

# get the width and the height to resize the matrix
l, b, w, h = agg.figure.bbox.bounds
w, h = int(w), int(h)

X = np.fromstring(s, np.uint8)
X.shape = h, w, 3

try:
    im = Image.fromstring("RGB", (w, h), s)
except Exception:
    im = Image.frombytes("RGB", (w, h), s)

# Uncomment this line to display the image using ImageMagick's
# `display` tool.
# im.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.3 pylab_examples example code: agg_buffer_to_array.py

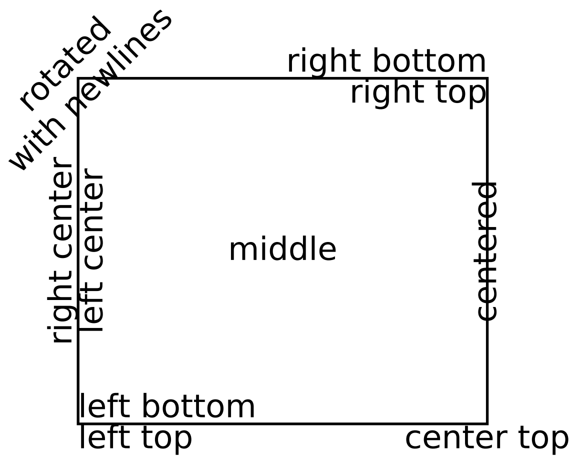





```
import matplotlib.pyplot as plt
import numpy as np

# make an agg figure
fig, ax = plt.subplots()
ax.plot([1, 2, 3])
ax.set_title('a simple figure')
fig.canvas.draw()

# grab the pixel buffer and dump it into a numpy array
X = np.array(fig.canvas.renderer._renderer)

# now display the array X as an Axes in a new figure
fig2 = plt.figure()
ax2 = fig2.add_subplot(111, frameon=False)
ax2.imshow(X)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.4 pylab_examples example code: alignment_test.py



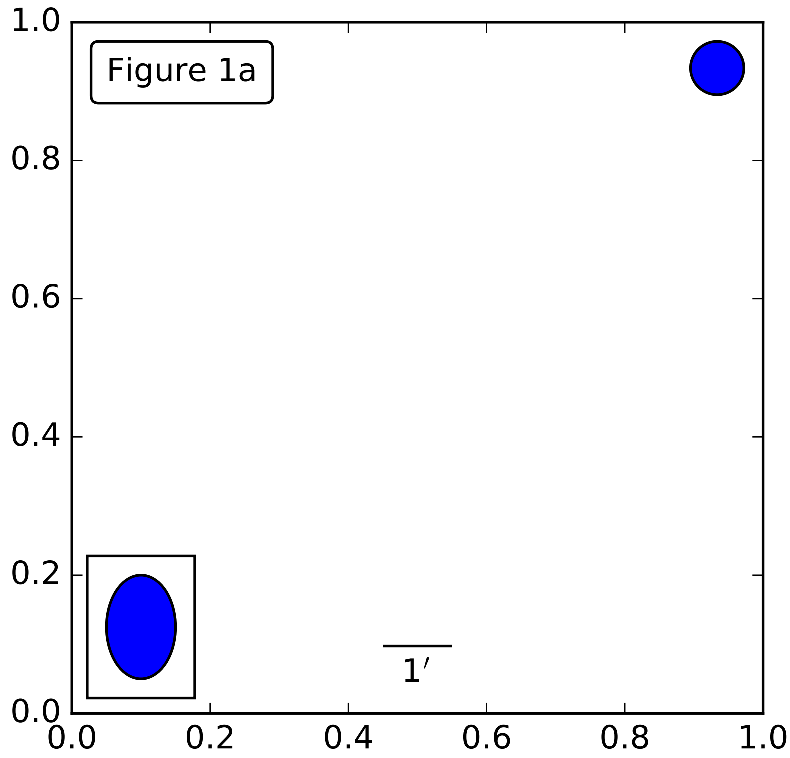

```
#!/usr/bin/env python
"""
You can precisely layout text in data or axes (0,1) coordinates.  This
example shows you some of the alignment and rotation specifications to
layout text
"""

import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.patches import Rectangle

# build a rectangle in axes coords
left, width = .25, .5
bottom, height = .25, .5
right = left + width
top = bottom + height
ax = plt.gca()
p = plt.Rectangle((left, bottom), width, height,
              fill=False,
              )
p.set_transform(ax.transAxes)
p.set_clip_on(False)
```

```
ax.add_patch(p)


ax.text(left, bottom, 'left top',
        horizontalalignment='left',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, bottom, 'left bottom',
        horizontalalignment='left',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right bottom',
        horizontalalignment='right',
        verticalalignment='bottom',
        transform=ax.transAxes)

ax.text(right, top, 'right top',
        horizontalalignment='right',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(right, bottom, 'center top',
        horizontalalignment='center',
        verticalalignment='top',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom + top), 'right center',
        horizontalalignment='right',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, 0.5*(bottom + top), 'left center',
        horizontalalignment='left',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(0.5*(left + right), 0.5*(bottom + top), 'middle',
        horizontalalignment='center',
        verticalalignment='center',
        transform=ax.transAxes)

ax.text(right, 0.5*(bottom + top), 'centered',
        horizontalalignment='center',
        verticalalignment='center',
        rotation='vertical',
        transform=ax.transAxes)

ax.text(left, top, 'rotated\nwith newlines',
        horizontalalignment='center',
```

```
        verticalalignment='center',
        rotation=45,
        transform=ax.transAxes)

plt.axis('off')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.5 pylab_examples example code: anchored_artists.py



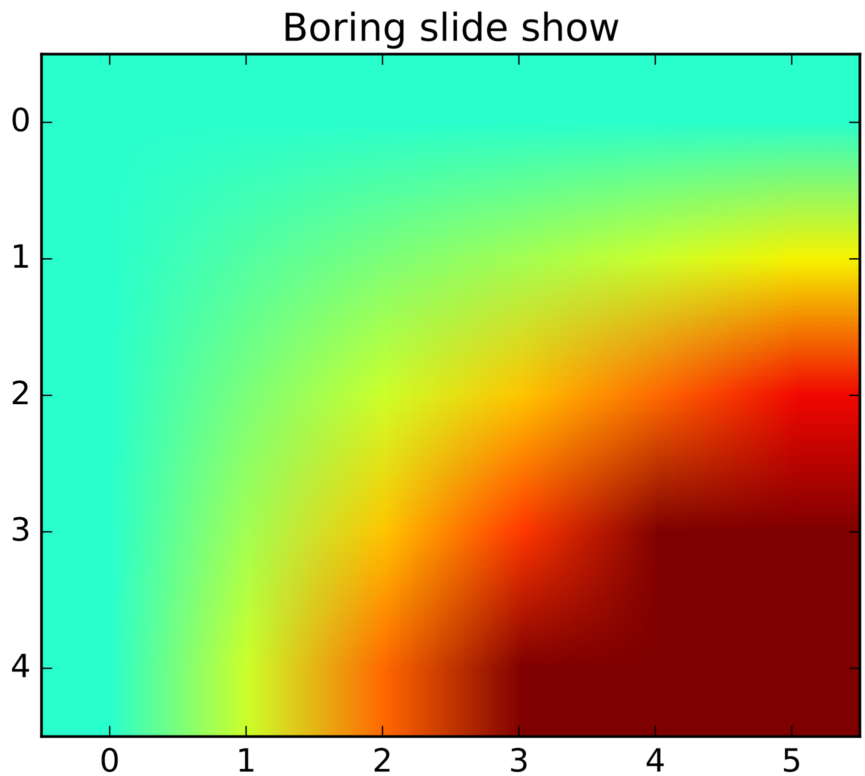

```
from matplotlib.patches import Rectangle, Ellipse

from matplotlib.offsetbox import AnchoredOffsetbox, AuxTransformBox, VPacker,\
    TextArea, DrawingArea


class AnchoredText(AnchoredOffsetbox):
    def __init__(self, s, loc, pad=0.4, borderpad=0.5, prop=None, frameon=True):

        self.txt = TextArea(s,
```

```
                            minimumdescent=False)

        super(AnchoredText, self).__init__(loc, pad=pad, borderpad=borderpad,
                                           child=self.txt,
                                           prop=prop,
                                           frameon=frameon)


class AnchoredSizeBar(AnchoredOffsetbox):
    def __init__(self, transform, size, label, loc,
                 pad=0.1, borderpad=0.1, sep=2, prop=None, frameon=True):
        """
        Draw a horizontal bar with the size in data coordinate of the give axes.
        A label will be drawn underneath (center-aligned).

        pad, borderpad in fraction of the legend font size (or prop)
        sep in points.
        """
        self.size_bar = AuxTransformBox(transform)
        self.size_bar.add_artist(Rectangle((0, 0), size, 0, fc="none"))

        self.txt_label = TextArea(label, minimumdescent=False)

        self._box = VPacker(children=[self.size_bar, self.txt_label],
                            align="center",
                            pad=0, sep=sep)

        AnchoredOffsetbox.__init__(self, loc, pad=pad, borderpad=borderpad,
                                   child=self._box,
                                   prop=prop,
                                   frameon=frameon)


class AnchoredEllipse(AnchoredOffsetbox):
    def __init__(self, transform, width, height, angle, loc,
                 pad=0.1, borderpad=0.1, prop=None, frameon=True):
        """
        Draw an ellipse the size in data coordinate of the give axes.

        pad, borderpad in fraction of the legend font size (or prop)
        """
        self._box = AuxTransformBox(transform)
        self.ellipse = Ellipse((0, 0), width, height, angle)
        self._box.add_artist(self.ellipse)

        AnchoredOffsetbox.__init__(self, loc, pad=pad, borderpad=borderpad,
                                   child=self._box,
                                   prop=prop,
                                   frameon=frameon)


class AnchoredDrawingArea(AnchoredOffsetbox):
    def __init__(self, width, height, xdescent, ydescent,
```

```
                 loc, pad=0.4, borderpad=0.5, prop=None, frameon=True):

        self.da = DrawingArea(width, height, xdescent, ydescent)

        super(AnchoredDrawingArea, self).__init__(loc, pad=pad, borderpad=borderpad,
                                                  child=self.da,
                                                  prop=None,
                                                  frameon=frameon)


if __name__ == "__main__":

    import matplotlib.pyplot as plt

    ax = plt.gca()
    ax.set_aspect(1.)

    at = AnchoredText("Figure 1a",
                      loc=2, frameon=True)
    at.patch.set_boxstyle("round,pad=0.,rounding_size=0.2")
    ax.add_artist(at)

    from matplotlib.patches import Circle
    ada = AnchoredDrawingArea(20, 20, 0, 0,
                              loc=1, pad=0., frameon=False)
    p = Circle((10, 10), 10)
    ada.da.add_artist(p)
    ax.add_artist(ada)

    # draw an ellipse of width=0.1, height=0.15 in the data coordinate
    ae = AnchoredEllipse(ax.transData, width=0.1, height=0.15, angle=0.,
                         loc=3, pad=0.5, borderpad=0.4, frameon=True)

    ax.add_artist(ae)

    # draw a horizontal bar with length of 0.1 in Data coordinate
    # (ax.transData) with a label underneath.
    asb = AnchoredSizeBar(ax.transData,
                          0.1,
                          r"1$^{\prime}$",
                          loc=8,
                          pad=0.1, borderpad=0.5, sep=5,
                          frameon=False)
    ax.add_artist(asb)

    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.6 pylab_examples example code: animation_demo.py

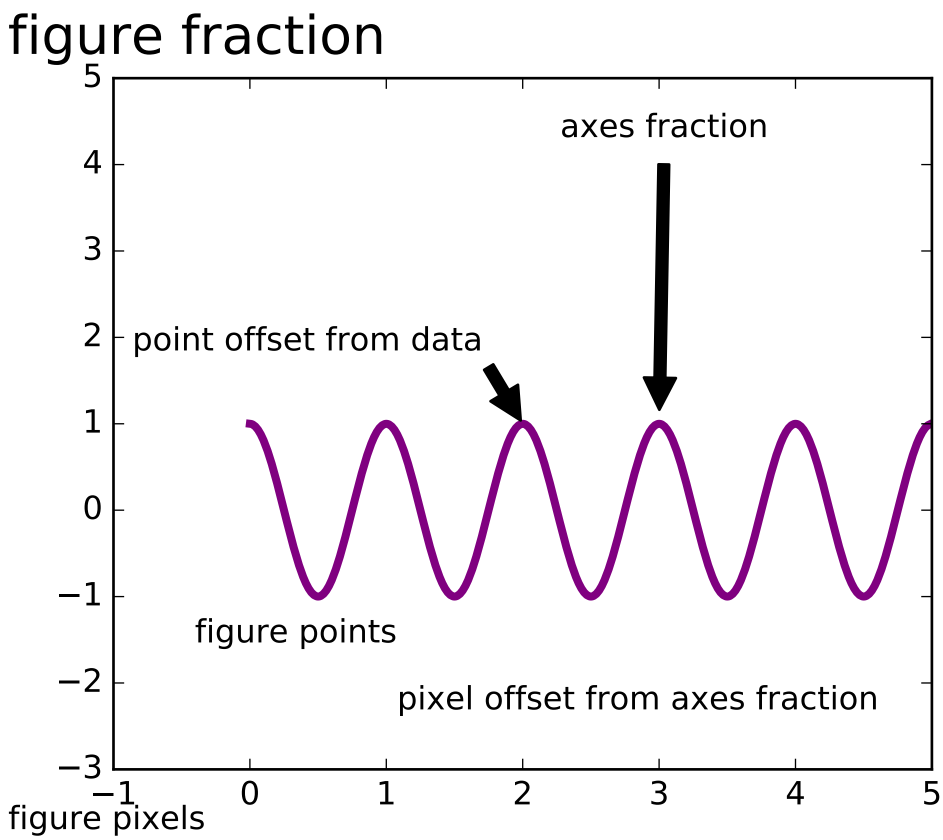


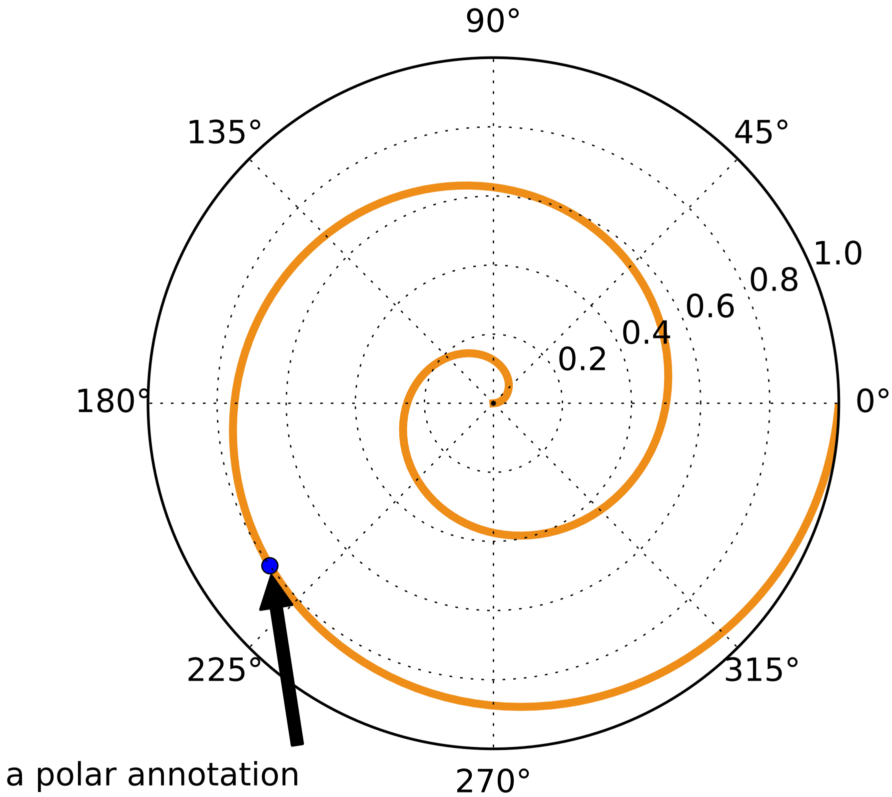

```
"""
Pyplot animation example.

The method shown here is only for very simple, low-performance
use.  For more demanding applications, look at the animation
module and the examples that use it.
"""

import matplotlib.pyplot as plt
import numpy as np

x = np.arange(6)
y = np.arange(5)
z = x * y[:, np.newaxis]

for i in range(5):
    if i == 0:
        p = plt.imshow(z)
        fig = plt.gcf()
        plt.clim()   # clamp the color limits
        plt.title("Boring slide show")
    else:
```

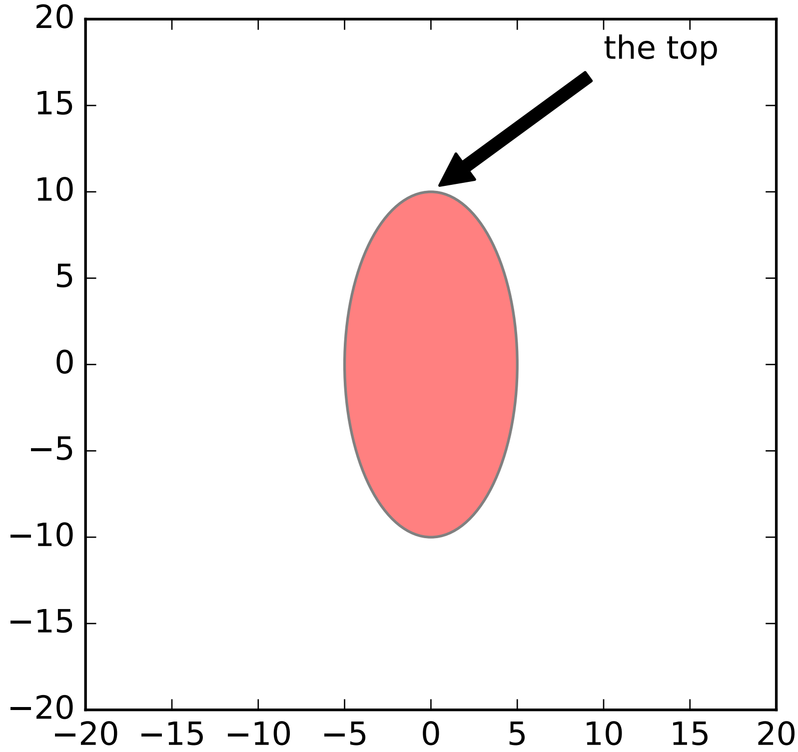
```
        z = z + 2
        p.set_data(z)

    print("step", i)
    plt.pause(0.5)
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.7 pylab_examples example code: annotation_demo.py




90°
135°
45°
180°
0°
225°
315°
270°
0.2
0.4
0.6
0.8
1.0
a polar annotation




```
"""
Some examples of how to annotate points in figures.  You specify an
annotation point xy=(x,y) and a text point xytext=(x,y) for the
annotated points and text location, respectively.  Optionally, you can
specify the coordinate system of xy and xytext with one of the
following strings for xycoords and textcoords (default is 'data')


  'figure points'   : points from the lower left corner of the figure
  'figure pixels'   : pixels from the lower left corner of the figure
  'figure fraction' : 0,0 is lower left of figure and 1,1 is upper, right
  'axes points'     : points from lower left corner of axes
  'axes pixels'     : pixels from lower left corner of axes
  'axes fraction'   : 0,0 is lower left of axes and 1,1 is upper right
  'offset points'   : Specify an offset (in points) from the xy value
  'offset pixels'   : Specify an offset (in pixels) from the xy value
  'data'            : use the axes data coordinate system

Optionally, you can specify arrow properties which draws and arrow
from the text to the annotated point by giving a dictionary of arrow
properties

Valid keys are

          width : the width of the arrow in points
          frac  : the fraction of the arrow length occupied by the head
```

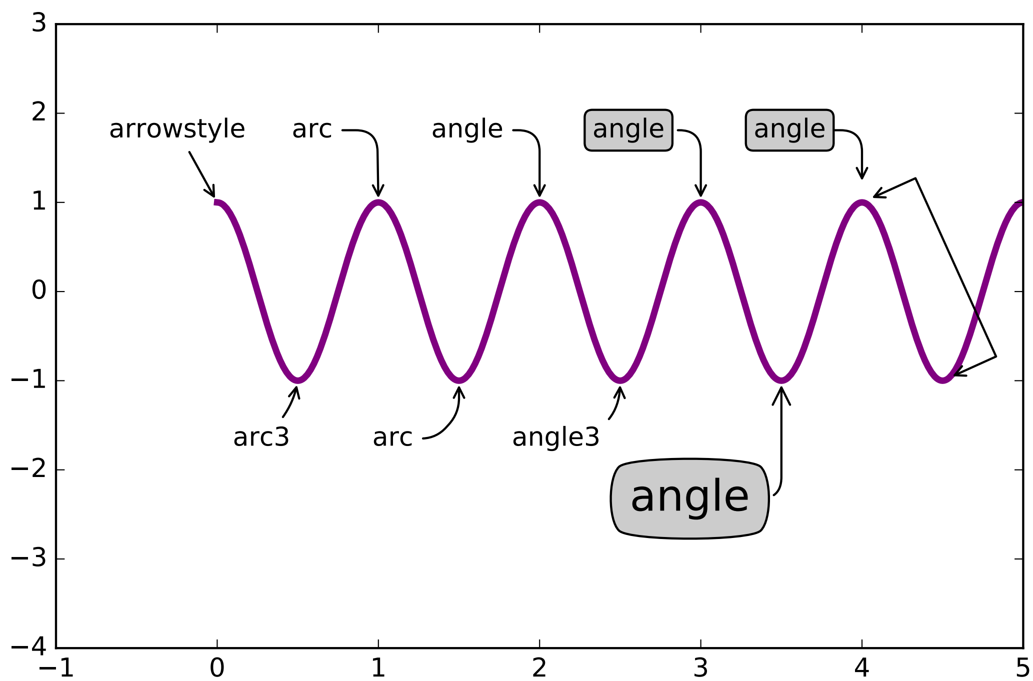
```
        headwidth : the width of the base of the arrow head in points
        shrink : move the tip and base some percent away from the
                 annotated point and text
        any key for matplotlib.patches.polygon  (e.g., facecolor)

For physical coordinate systems (points or pixels) the origin is the
(bottom, left) of the figure or axes.
"""


import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
import numpy as np


if 1:
    # if only one location is given, the text and xypoint being
    # annotated are assumed to be the same
    fig = plt.figure()
    ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1, 5), ylim=(-3, 5))

    t = np.arange(0.0, 5.0, 0.01)
    s = np.cos(2*np.pi*t)
    line, = ax.plot(t, s, lw=3, color='purple')

    ax.annotate('figure pixels',
                xy=(10, 10), xycoords='figure pixels')

    ax.annotate('figure points', xy=(80, 80),
                xycoords='figure points')

    ax.annotate('point offset from data', xy=(2, 1),
                xycoords='data',
                xytext=(-15, 25), textcoords='offset points',
                arrowprops=dict(facecolor='black', shrink=0.05),
                horizontalalignment='right', verticalalignment='bottom',
                )

    ax.annotate('axes fraction', xy=(3, 1), xycoords='data',
                xytext=(0.8, 0.95), textcoords='axes fraction',
                arrowprops=dict(facecolor='black', shrink=0.05),
                horizontalalignment='right', verticalalignment='top',
                )

    ax.annotate('figure fraction', xy=(.025, .975),
                xycoords='figure fraction',
                horizontalalignment='left', verticalalignment='top',
                fontsize=20)

    # use negative points or pixels to specify from right, top -10, 10
    # is 10 points to the left of the right side of the axes and 10
    # points above the bottom
    ax.annotate('pixel offset from axes fraction', xy=(1, 0),
```

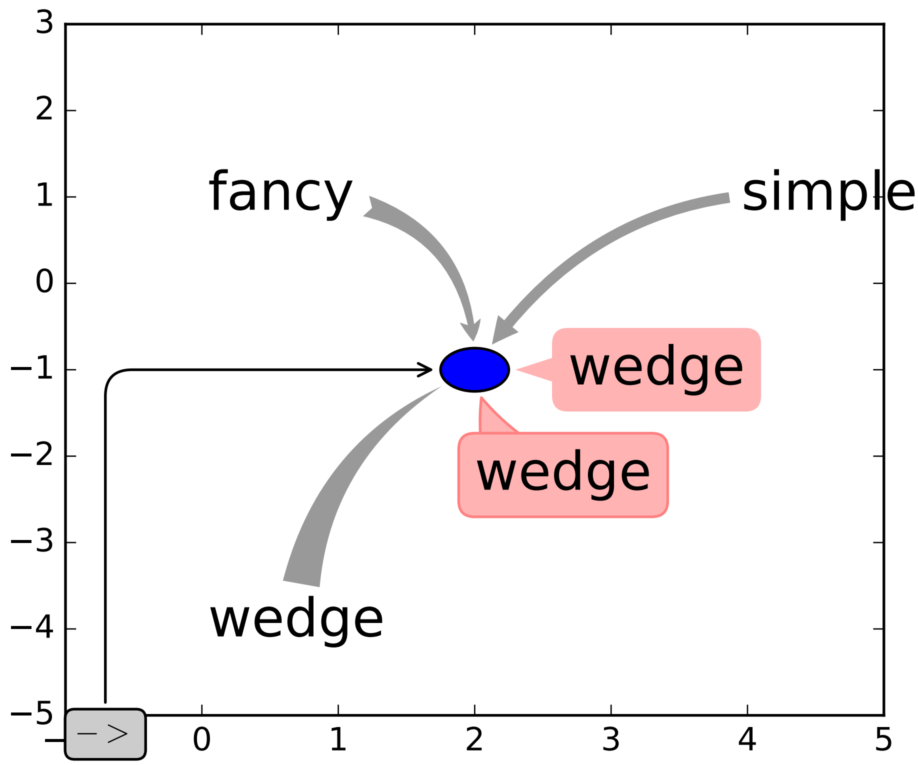
```
                xycoords='axes fraction',
                xytext=(-20, 20),
                textcoords='offset pixels',
                horizontalalignment='right',
                verticalalignment='bottom')


if 1:
    # you can specify the xypoint and the xytext in different
    # positions and coordinate systems, and optionally turn on a
    # connecting line and mark the point with a marker.  Annotations
    # work on polar axes too.  In the example below, the xy point is
    # in native coordinates (xycoords defaults to 'data').  For a
    # polar axes, this is in (theta, radius) space.  The text in this
    # example is placed in the fractional figure coordinate system.
    # Text keyword args like horizontal and vertical alignment are
    # respected
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='polar')
    r = np.arange(0, 1, 0.001)
    theta = 2*2*np.pi*r
    line, = ax.plot(theta, r, color='#ee8d18', lw=3)

    ind = 800
    thisr, thistheta = r[ind], theta[ind]
    ax.plot([thistheta], [thisr], 'o')
    ax.annotate('a polar annotation',
                xy=(thistheta, thisr),  # theta, radius
                xytext=(0.05, 0.05),    # fraction, fraction
                textcoords='figure fraction',
                arrowprops=dict(facecolor='black', shrink=0.05),
                horizontalalignment='left',
                verticalalignment='bottom',
                )


if 1:
    # You can also use polar notation on a cartesian axes.  Here the
    # native coordinate system ('data') is cartesian, so you need to
    # specify the xycoords and textcoords as 'polar' if you want to
    # use (theta, radius)

    el = Ellipse((0, 0), 10, 20, facecolor='r', alpha=0.5)

    fig = plt.figure()
    ax = fig.add_subplot(111, aspect='equal')
    ax.add_artist(el)
    el.set_clip_box(ax.bbox)
    ax.annotate('the top',
                xy=(np.pi/2., 10.),      # theta, radius
                xytext=(np.pi/3, 20.),   # theta, radius
                xycoords='polar',
                textcoords='polar',
```

```
                arrowprops=dict(facecolor='black', shrink=0.05),
                horizontalalignment='left',
                verticalalignment='bottom',
                clip_on=True,  # clip to the axes bounding box
                )

    ax.set_xlim(-20, 20)
    ax.set_ylim(-20, 20)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.8 pylab_examples example code: annotation_demo2.py






```
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
import numpy as np

if 1:
    fig = plt.figure(1, figsize=(8, 5))
    ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1, 5), ylim=(-4, 3))

    t = np.arange(0.0, 5.0, 0.01)
    s = np.cos(2*np.pi*t)
    line, = ax.plot(t, s, lw=3, color='purple')

    ax.annotate('arrowstyle', xy=(0, 1), xycoords='data',
                xytext=(-50, 30), textcoords='offset points',
                arrowprops=dict(arrowstyle="->")
                )

    ax.annotate('arc3', xy=(0.5, -1), xycoords='data',
                xytext=(-30, -30), textcoords='offset points',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="arc3,rad=.2")
                )

    ax.annotate('arc', xy=(1., 1), xycoords='data',
                xytext=(-40, 30), textcoords='offset points',
                arrowprops=dict(arrowstyle="->",
```

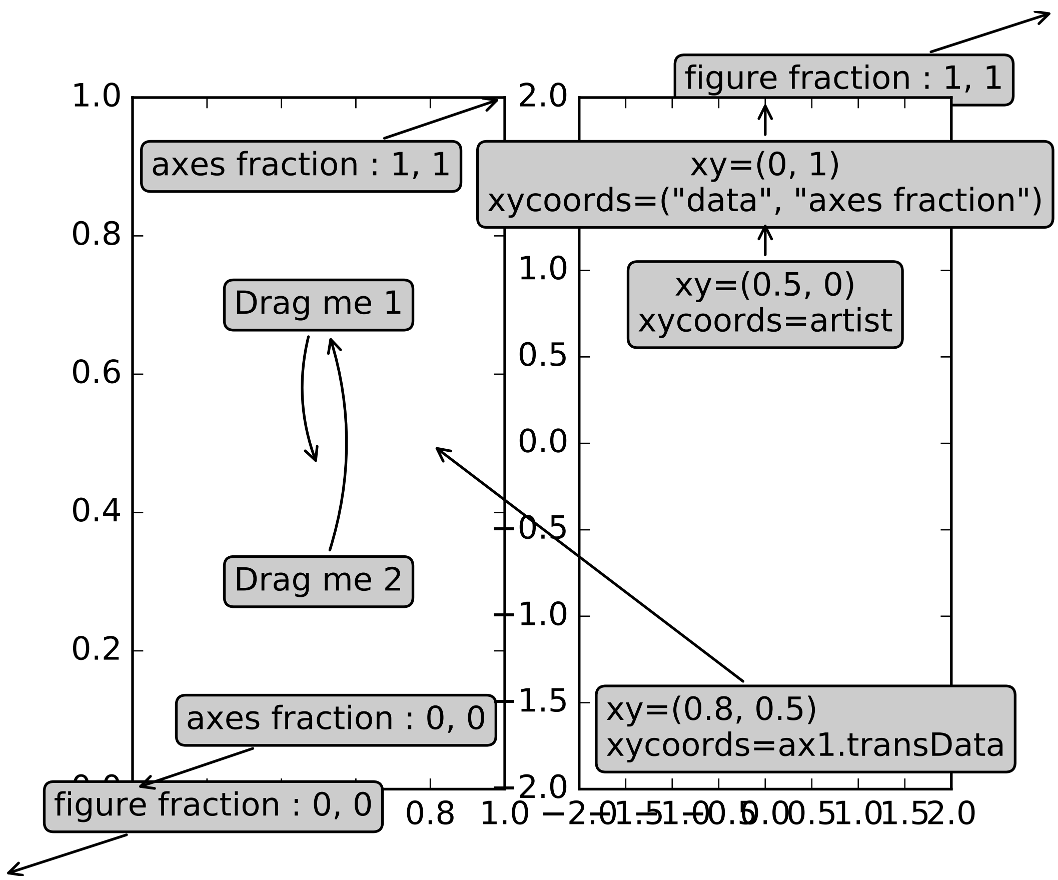
```
                                connectionstyle="arc,angleA=0,armA=30,rad=10"),
                )

    ax.annotate('arc', xy=(1.5, -1), xycoords='data',
                xytext=(-40, -30), textcoords='offset points',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="arc,angleA=0,armA=20,angleB=-
↪90,armB=15,rad=7"),
                )

    ax.annotate('angle', xy=(2., 1), xycoords='data',
                xytext=(-50, 30), textcoords='offset points',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="angle,angleA=0,angleB=90,rad=10"),
                )

    ax.annotate('angle3', xy=(2.5, -1), xycoords='data',
                xytext=(-50, -30), textcoords='offset points',
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="angle3,angleA=0,angleB=-90"),
                )

    ax.annotate('angle', xy=(3., 1), xycoords='data',
                xytext=(-50, 30), textcoords='offset points',
                bbox=dict(boxstyle="round", fc="0.8"),
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="angle,angleA=0,angleB=90,rad=10"),
                )

    ax.annotate('angle', xy=(3.5, -1), xycoords='data',
                xytext=(-70, -60), textcoords='offset points',
                size=20,
                bbox=dict(boxstyle="round4,pad=.5", fc="0.8"),
                arrowprops=dict(arrowstyle="->",
                                connectionstyle="angle,angleA=0,angleB=-90,rad=10"),
                )

    ax.annotate('angle', xy=(4., 1), xycoords='data',
                xytext=(-50, 30), textcoords='offset points',
                bbox=dict(boxstyle="round", fc="0.8"),
                arrowprops=dict(arrowstyle="->",
                                shrinkA=0, shrinkB=10,
                                connectionstyle="angle,angleA=0,angleB=90,rad=10"),
                )

    ann = ax.annotate('', xy=(4., 1.), xycoords='data',
                      xytext=(4.5, -1), textcoords='data',
                      arrowprops=dict(arrowstyle="<->",
                                      connectionstyle="bar",
                                      ec="k",
                                      shrinkA=5, shrinkB=5,
                                      )
                      )
```

```python
if 1:
    fig = plt.figure(2)
    fig.clf()
    ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1, 5), ylim=(-5, 3))

    el = Ellipse((2, -1), 0.5, 0.5)
    ax.add_patch(el)

    ax.annotate('$->$', xy=(2., -1), xycoords='data',
                xytext=(-150, -140), textcoords='offset points',
                bbox=dict(boxstyle="round", fc="0.8"),
                arrowprops=dict(arrowstyle="->",
                                patchB=el,
                                connectionstyle="angle,angleA=90,angleB=0,rad=10"),
                )

    ax.annotate('fancy', xy=(2., -1), xycoords='data',
                xytext=(-100, 60), textcoords='offset points',
                size=20,
                # bbox=dict(boxstyle="round", fc="0.8"),
                arrowprops=dict(arrowstyle="fancy",
                                fc="0.6", ec="none",
                                patchB=el,
                                connectionstyle="angle3,angleA=0,angleB=-90"),
                )

    ax.annotate('simple', xy=(2., -1), xycoords='data',
                xytext=(100, 60), textcoords='offset points',
                size=20,
                # bbox=dict(boxstyle="round", fc="0.8"),
                arrowprops=dict(arrowstyle="simple",
                                fc="0.6", ec="none",
                                patchB=el,
                                connectionstyle="arc3,rad=0.3"),
                )

    ax.annotate('wedge', xy=(2., -1), xycoords='data',
                xytext=(-100, -100), textcoords='offset points',
                size=20,
                # bbox=dict(boxstyle="round", fc="0.8"),
                arrowprops=dict(arrowstyle="wedge,tail_width=0.7",
                                fc="0.6", ec="none",
                                patchB=el,
                                connectionstyle="arc3,rad=-0.3"),
                )

    ann = ax.annotate('wedge', xy=(2., -1), xycoords='data',
                      xytext=(0, -45), textcoords='offset points',
                      size=20,
                      bbox=dict(boxstyle="round", fc=(1.0, 0.7, 0.7), ec=(1., .5, .5)),
                      arrowprops=dict(arrowstyle="wedge,tail_width=1.",
```

```
                                    fc=(1.0, 0.7, 0.7), ec=(1., .5, .5),
                                    patchA=None,
                                    patchB=el,
                                    relpos=(0.2, 0.8),
                                    connectionstyle="arc3,rad=-0.1"),
                    )

    ann = ax.annotate('wedge', xy=(2., -1), xycoords='data',
                    xytext=(35, 0), textcoords='offset points',
                    size=20, va="center",
                    bbox=dict(boxstyle="round", fc=(1.0, 0.7, 0.7), ec="none"),
                    arrowprops=dict(arrowstyle="wedge,tail_width=1.",
                                    fc=(1.0, 0.7, 0.7), ec="none",
                                    patchA=None,
                                    patchB=el,
                                    relpos=(0.2, 0.5),
                                    )
                    )


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.9 pylab_examples example code: annotation_demo3.py



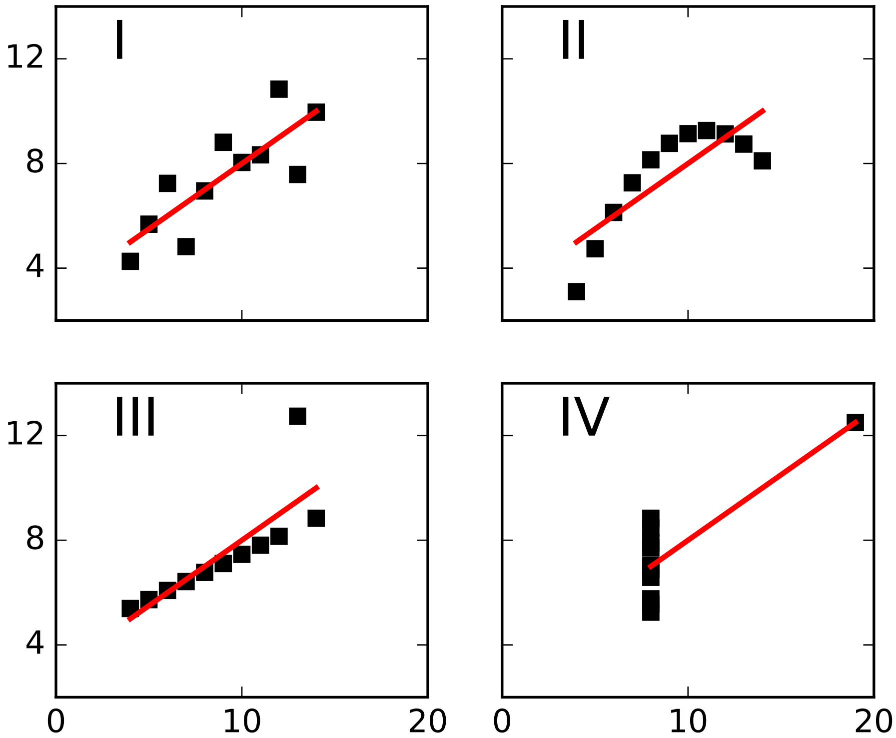

```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)

bbox_args = dict(boxstyle="round", fc="0.8")
arrow_args = dict(arrowstyle="->")

ax1.annotate('figure fraction : 0, 0', xy=(0, 0), xycoords='figure fraction',
             xytext=(20, 20), textcoords='offset points',
             ha="left", va="bottom",
             bbox=bbox_args,
             arrowprops=arrow_args
             )

ax1.annotate('figure fraction : 1, 1', xy=(1, 1), xycoords='figure fraction',
             xytext=(-20, -20), textcoords='offset points',
             ha="right", va="top",
             bbox=bbox_args,
             arrowprops=arrow_args
             )

ax1.annotate('axes fraction : 0, 0', xy=(0, 0), xycoords='axes fraction',
```

```
             xytext=(20, 20), textcoords='offset points',
             ha="left", va="bottom",
             bbox=bbox_args,
             arrowprops=arrow_args
             )

ax1.annotate('axes fraction : 1, 1', xy=(1, 1), xycoords='axes fraction',
             xytext=(-20, -20), textcoords='offset points',
             ha="right", va="top",
             bbox=bbox_args,
             arrowprops=arrow_args
             )


an1 = ax1.annotate('Drag me 1', xy=(.5, .7), xycoords='data',
                   #xytext=(.5, .7), textcoords='data',
                   ha="center", va="center",
                   bbox=bbox_args,
                   #arrowprops=arrow_args
                   )

an2 = ax1.annotate('Drag me 2', xy=(.5, .5), xycoords=an1,
                   xytext=(.5, .3), textcoords='axes fraction',
                   ha="center", va="center",
                   bbox=bbox_args,
                   arrowprops=dict(patchB=an1.get_bbox_patch(),
                                   connectionstyle="arc3,rad=0.2",
                                   **arrow_args)
                   )

an3 = ax1.annotate('', xy=(.5, .5), xycoords=an2,
                   xytext=(.5, .5), textcoords=an1,
                   ha="center", va="center",
                   bbox=bbox_args,
                   arrowprops=dict(patchA=an1.get_bbox_patch(),
                                   patchB=an2.get_bbox_patch(),
                                   connectionstyle="arc3,rad=0.2",
                                   **arrow_args)
                   )


t = ax2.annotate('xy=(0, 1)\nxycoords=("data", "axes fraction")',
                 xy=(0, 1), xycoords=("data", 'axes fraction'),
                 xytext=(0, -20), textcoords='offset points',
                 ha="center", va="top",
                 bbox=bbox_args,
                 arrowprops=arrow_args
                 )

from matplotlib.text import OffsetFrom

ax2.annotate('xy=(0.5, 0)\nxycoords=artist',
             xy=(0.5, 0.), xycoords=t,
```

```
             xytext=(0, -20), textcoords='offset points',
             ha="center", va="top",
             bbox=bbox_args,
             arrowprops=arrow_args
             )

ax2.annotate('xy=(0.8, 0.5)\nxycoords=ax1.transData',
             xy=(0.8, 0.5), xycoords=ax1.transData,
             xytext=(10, 10), textcoords=OffsetFrom(ax2.bbox, (0, 0), "points"),
             ha="left", va="bottom",
             bbox=bbox_args,
             arrowprops=arrow_args
             )

ax2.set_xlim(-2, 2)
ax2.set_ylim(-2, 2)

an1.draggable()
an2.draggable()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.10 pylab_examples example code: anscombe.py

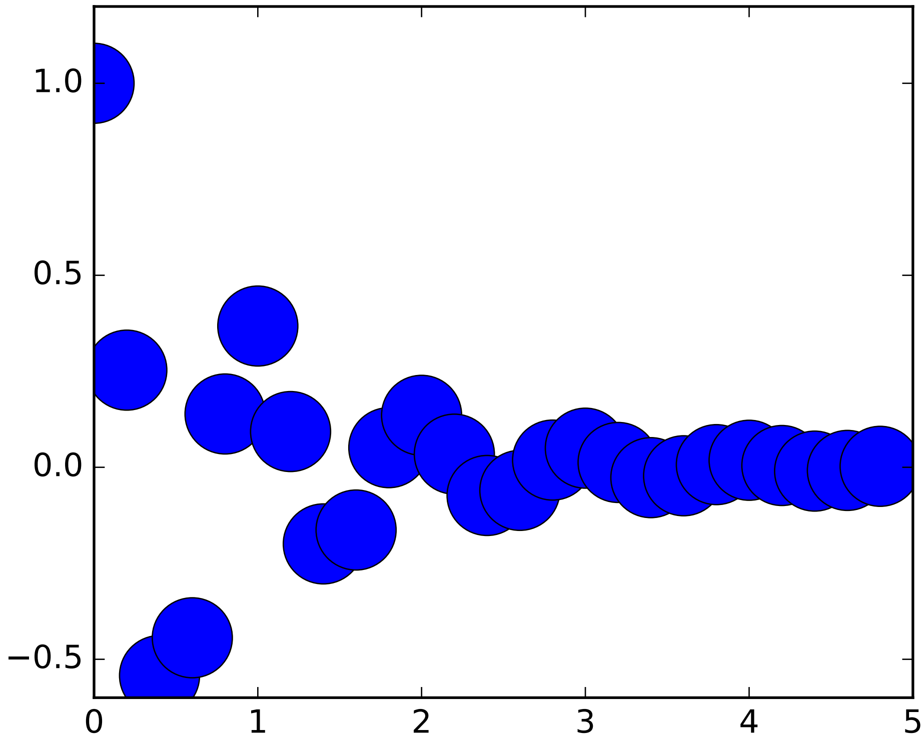



```
#!/usr/bin/env python

from __future__ import print_function
"""
Edward Tufte uses this example from Anscombe to show 4 datasets of x
and y that have the same mean, standard deviation, and regression
line, but which are qualitatively different.

matplotlib fun for a rainy day
"""

import matplotlib.pyplot as plt
import numpy as np

x = np.array([10, 8, 13, 9, 11, 14, 6, 4, 12, 7, 5])
y1 = np.array([8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68])
y2 = np.array([9.14, 8.14, 8.74, 8.77, 9.26, 8.10, 6.13, 3.10, 9.13, 7.26, 4.74])
y3 = np.array([7.46, 6.77, 12.74, 7.11, 7.81, 8.84, 6.08, 5.39, 8.15, 6.42, 5.73])
x4 = np.array([8, 8, 8, 8, 8, 8, 8, 19, 8, 8, 8])
y4 = np.array([6.58, 5.76, 7.71, 8.84, 8.47, 7.04, 5.25, 12.50, 5.56, 7.91, 6.89])
```

```
def fit(x):
    return 3 + 0.5*x


xfit = np.array([np.amin(x), np.amax(x)])

plt.subplot(221)
plt.plot(x, y1, 'ks', xfit, fit(xfit), 'r-', lw=2)
plt.axis([2, 20, 2, 14])
plt.setp(plt.gca(), xticklabels=[], yticks=(4, 8, 12), xticks=(0, 10, 20))
plt.text(3, 12, 'I', fontsize=20)

plt.subplot(222)
plt.plot(x, y2, 'ks', xfit, fit(xfit), 'r-', lw=2)
plt.axis([2, 20, 2, 14])
plt.setp(plt.gca(), xticklabels=[], yticks=(4, 8, 12), yticklabels=[], xticks=(0, 10,
↪20))
plt.text(3, 12, 'II', fontsize=20)

plt.subplot(223)
plt.plot(x, y3, 'ks', xfit, fit(xfit), 'r-', lw=2)
plt.axis([2, 20, 2, 14])
plt.text(3, 12, 'III', fontsize=20)
plt.setp(plt.gca(), yticks=(4, 8, 12), xticks=(0, 10, 20))

plt.subplot(224)

xfit = np.array([np.amin(x4), np.amax(x4)])
plt.plot(x4, y4, 'ks', xfit, fit(xfit), 'r-', lw=2)
plt.axis([2, 20, 2, 14])
plt.setp(plt.gca(), yticklabels=[], yticks=(4, 8, 12), xticks=(0, 10, 20))
plt.text(3, 12, 'IV', fontsize=20)

# verify the stats
pairs = (x, y1), (x, y2), (x, y3), (x4, y4)
for x, y in pairs:
    print('mean=%1.2f, std=%1.2f, r=%1.2f' % (np.mean(y), np.std(y), np.corrcoef(x,
↪y)[0][1]))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

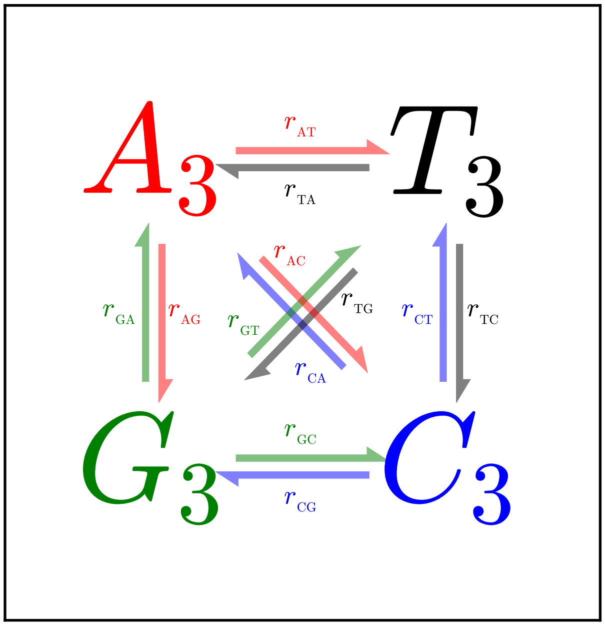
90.11 pylab_examples example code: arctest.py




```
import matplotlib.pyplot as plt
import numpy as np


def f(t):
    'a damped exponential'
    s1 = np.cos(2 * np.pi * t)
    e1 = np.exp(-t)
    return s1 * e1

t1 = np.arange(0.0, 5.0, .2)


l = plt.plot(t1, f(t1), 'ro')
plt.setp(l, 'markersize', 30)
plt.setp(l, 'markerfacecolor', 'b')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.12 pylab_examples example code: arrow_demo.py



```
#!/usr/bin/env python
"""Arrow drawing example for the new fancy_arrow facilities.

Code contributed by: Rob Knight <rob@spot.colorado.edu>

usage:

  python arrow_demo.py realistic|full|sample|extreme


"""
import matplotlib.pyplot as plt
import numpy as np

rates_to_bases = {'r1': 'AT', 'r2': 'TA', 'r3': 'GA', 'r4': 'AG', 'r5': 'CA',
                  'r6': 'AC', 'r7': 'GT', 'r8': 'TG', 'r9': 'CT', 'r10': 'TC',
                  'r11': 'GC', 'r12': 'CG'}
numbered_bases_to_rates = dict([(v, k) for k, v in rates_to_bases.items()])
lettered_bases_to_rates = dict([(v, 'r' + v) for k, v in rates_to_bases.items()])


def add_dicts(d1, d2):
    """Adds two dicts and returns the result."""
    result = d1.copy()
    result.update(d2)
```

```
    return result


def make_arrow_plot(data, size=4, display='length', shape='right',
                    max_arrow_width=0.03, arrow_sep=0.02, alpha=0.5,
                    normalize_data=False, ec=None, labelcolor=None,
                    head_starts_at_zero=True, rate_labels=lettered_bases_to_rates,
                    **kwargs):
    """Makes an arrow plot.

    Parameters:

    data: dict with probabilities for the bases and pair transitions.
    size: size of the graph in inches.
    display: 'length', 'width', or 'alpha' for arrow property to change.
    shape: 'full', 'left', or 'right' for full or half arrows.
    max_arrow_width: maximum width of an arrow, data coordinates.
    arrow_sep: separation between arrows in a pair, data coordinates.
    alpha: maximum opacity of arrows, default 0.8.

    **kwargs can be anything allowed by a Arrow object, e.g.
    linewidth and edgecolor.
    """

    plt.xlim(-0.5, 1.5)
    plt.ylim(-0.5, 1.5)
    plt.gcf().set_size_inches(size, size)
    plt.xticks([])
    plt.yticks([])
    max_text_size = size*12
    min_text_size = size
    label_text_size = size*2.5
    text_params = {'ha': 'center', 'va': 'center', 'family': 'sans-serif',
                   'fontweight': 'bold'}
    r2 = np.sqrt(2)

    deltas = {
        'AT': (1, 0),
        'TA': (-1, 0),
        'GA': (0, 1),
        'AG': (0, -1),
        'CA': (-1/r2, 1/r2),
        'AC': (1/r2, -1/r2),
        'GT': (1/r2, 1/r2),
        'TG': (-1/r2, -1/r2),
        'CT': (0, 1),
        'TC': (0, -1),
        'GC': (1, 0),
        'CG': (-1, 0)
        }

    colors = {
        'AT': 'r',
```

```
        'TA': 'k',
        'GA': 'g',
        'AG': 'r',
        'CA': 'b',
        'AC': 'r',
        'GT': 'g',
        'TG': 'k',
        'CT': 'b',
        'TC': 'k',
        'GC': 'g',
        'CG': 'b'
        }

    label_positions = {
        'AT': 'center',
        'TA': 'center',
        'GA': 'center',
        'AG': 'center',
        'CA': 'left',
        'AC': 'left',
        'GT': 'left',
        'TG': 'left',
        'CT': 'center',
        'TC': 'center',
        'GC': 'center',
        'CG': 'center'
        }

    def do_fontsize(k):
        return float(np.clip(max_text_size*np.sqrt(data[k]),
                              min_text_size, max_text_size))

    A = plt.text(0, 1, '$A_3$', color='r', size=do_fontsize('A'), **text_params)
    T = plt.text(1, 1, '$T_3$', color='k', size=do_fontsize('T'), **text_params)
    G = plt.text(0, 0, '$G_3$', color='g', size=do_fontsize('G'), **text_params)
    C = plt.text(1, 0, '$C_3$', color='b', size=do_fontsize('C'), **text_params)

    arrow_h_offset = 0.25  # data coordinates, empirically determined
    max_arrow_length = 1 - 2*arrow_h_offset

    max_arrow_width = max_arrow_width
    max_head_width = 2.5*max_arrow_width
    max_head_length = 2*max_arrow_width
    arrow_params = {'length_includes_head': True, 'shape': shape,
                    'head_starts_at_zero': head_starts_at_zero}
    ax = plt.gca()
    sf = 0.6  # max arrow size represents this in data coords

    d = (r2/2 + arrow_h_offset - 0.5)/r2  # distance for diags
    r2v = arrow_sep/r2  # offset for diags

    # tuple of x, y for start position
    positions = {
```

```
        'AT': (arrow_h_offset, 1 + arrow_sep),
        'TA': (1 - arrow_h_offset, 1 - arrow_sep),
        'GA': (-arrow_sep, arrow_h_offset),
        'AG': (arrow_sep, 1 - arrow_h_offset),
        'CA': (1 - d - r2v, d - r2v),
        'AC': (d + r2v, 1 - d + r2v),
        'GT': (d - r2v, d + r2v),
        'TG': (1 - d + r2v, 1 - d - r2v),
        'CT': (1 - arrow_sep, arrow_h_offset),
        'TC': (1 + arrow_sep, 1 - arrow_h_offset),
        'GC': (arrow_h_offset, arrow_sep),
        'CG': (1 - arrow_h_offset, -arrow_sep),
        }

    if normalize_data:
        # find maximum value for rates, i.e. where keys are 2 chars long
        max_val = 0
        for k, v in data.items():
            if len(k) == 2:
                max_val = max(max_val, v)
        # divide rates by max val, multiply by arrow scale factor
        for k, v in data.items():
            data[k] = v/max_val*sf

    def draw_arrow(pair, alpha=alpha, ec=ec, labelcolor=labelcolor):
        # set the length of the arrow
        if display == 'length':
            length = max_head_length + data[pair]/sf*(max_arrow_length -
                                                      max_head_length)
        else:
            length = max_arrow_length
        # set the transparency of the arrow
        if display == 'alph':
            alpha = min(data[pair]/sf, alpha)
        else:
            alpha = alpha
        # set the width of the arrow
        if display == 'width':
            scale = data[pair]/sf
            width = max_arrow_width*scale
            head_width = max_head_width*scale
            head_length = max_head_length*scale
        else:
            width = max_arrow_width
            head_width = max_head_width
            head_length = max_head_length

        fc = colors[pair]
        ec = ec or fc

        x_scale, y_scale = deltas[pair]
        x_pos, y_pos = positions[pair]
        plt.arrow(x_pos, y_pos, x_scale*length, y_scale*length,
```

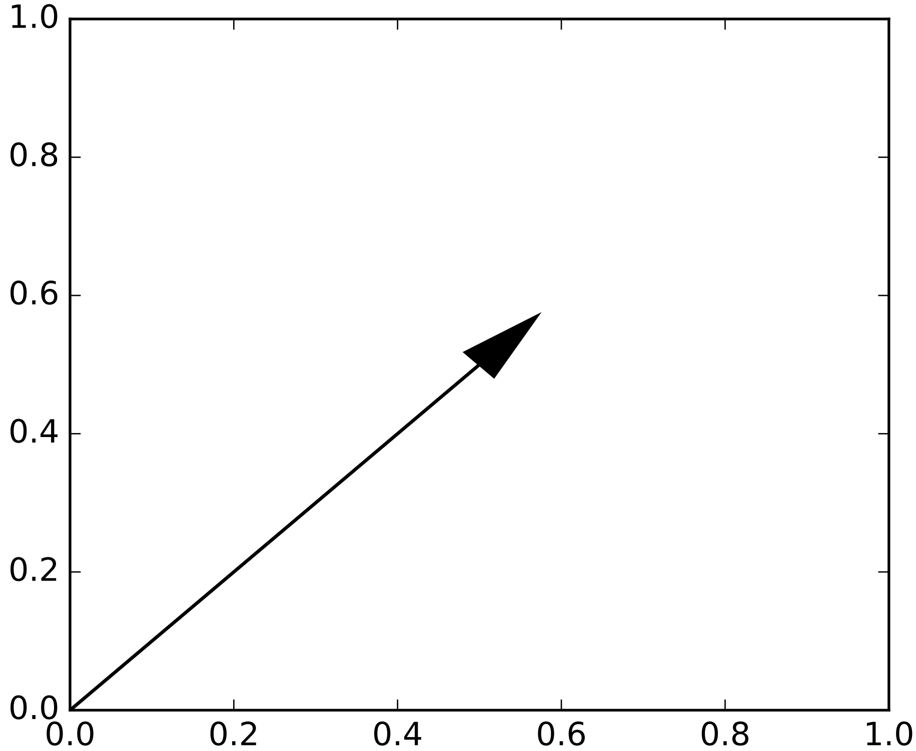
```
              fc=fc, ec=ec, alpha=alpha, width=width, head_width=head_width,
              head_length=head_length, **arrow_params)

        # figure out coordinates for text
        # if drawing relative to base: x and y are same as for arrow
        # dx and dy are one arrow width left and up
        # need to rotate based on direction of arrow, use x_scale and y_scale
        # as sin x and cos x?
        sx, cx = y_scale, x_scale

        where = label_positions[pair]
        if where == 'left':
            orig_position = 3*np.array([[max_arrow_width, max_arrow_width]])
        elif where == 'absolute':
            orig_position = np.array([[max_arrow_length/2.0, 3*max_arrow_width]])
        elif where == 'right':
            orig_position = np.array([[length - 3*max_arrow_width,
                                       3*max_arrow_width]])
        elif where == 'center':
            orig_position = np.array([[length/2.0, 3*max_arrow_width]])
        else:
            raise ValueError("Got unknown position parameter %s" % where)

        M = np.array([[cx, sx], [-sx, cx]])
        coords = np.dot(orig_position, M) + [[x_pos, y_pos]]
        x, y = np.ravel(coords)
        orig_label = rate_labels[pair]
        label = '$%s_{_{\mathrm{%s}}}$' % (orig_label[0], orig_label[1:])

        plt.text(x, y, label, size=label_text_size, ha='center', va='center',
                 color=labelcolor or fc)

    for p in positions.keys():
        draw_arrow(p)

# test data
all_on_max = dict([(i, 1) for i in 'TCAG'] +
                  [(i + j, 0.6) for i in 'TCAG' for j in 'TCAG'])

realistic_data = {
    'A': 0.4,
    'T': 0.3,
    'G': 0.5,
    'C': 0.2,
    'AT': 0.4,
    'AC': 0.3,
    'AG': 0.2,
    'TA': 0.2,
    'TC': 0.3,
    'TG': 0.4,
    'CT': 0.2,
    'CG': 0.3,
    'CA': 0.2,
```

```
    'GA': 0.1,
    'GT': 0.4,
    'GC': 0.1,
    }

extreme_data = {
    'A': 0.75,
    'T': 0.10,
    'G': 0.10,
    'C': 0.05,
    'AT': 0.6,
    'AC': 0.3,
    'AG': 0.1,
    'TA': 0.02,
    'TC': 0.3,
    'TG': 0.01,
    'CT': 0.2,
    'CG': 0.5,
    'CA': 0.2,
    'GA': 0.1,
    'GT': 0.4,
    'GC': 0.2,
    }

sample_data = {
    'A': 0.2137,
    'T': 0.3541,
    'G': 0.1946,
    'C': 0.2376,
    'AT': 0.0228,
    'AC': 0.0684,
    'AG': 0.2056,
    'TA': 0.0315,
    'TC': 0.0629,
    'TG': 0.0315,
    'CT': 0.1355,
    'CG': 0.0401,
    'CA': 0.0703,
    'GA': 0.1824,
    'GT': 0.0387,
    'GC': 0.1106,
    }


if __name__ == '__main__':
    from sys import argv
    d = None
    if len(argv) > 1:
        if argv[1] == 'full':
            d = all_on_max
            scaled = False
        elif argv[1] == 'extreme':
            d = extreme_data
```

```
            scaled = False
        elif argv[1] == 'realistic':
            d = realistic_data
            scaled = False
        elif argv[1] == 'sample':
            d = sample_data
            scaled = True
    if d is None:
        d = all_on_max
        scaled = False
    if len(argv) > 2:
        display = argv[2]
    else:
        display = 'length'

    size = 4
    plt.figure(figsize=(size, size))

    make_arrow_plot(d, display=display, linewidth=0.001, edgecolor=None,
                    normalize_data=scaled, head_starts_at_zero=True, size=size)

    plt.draw()

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

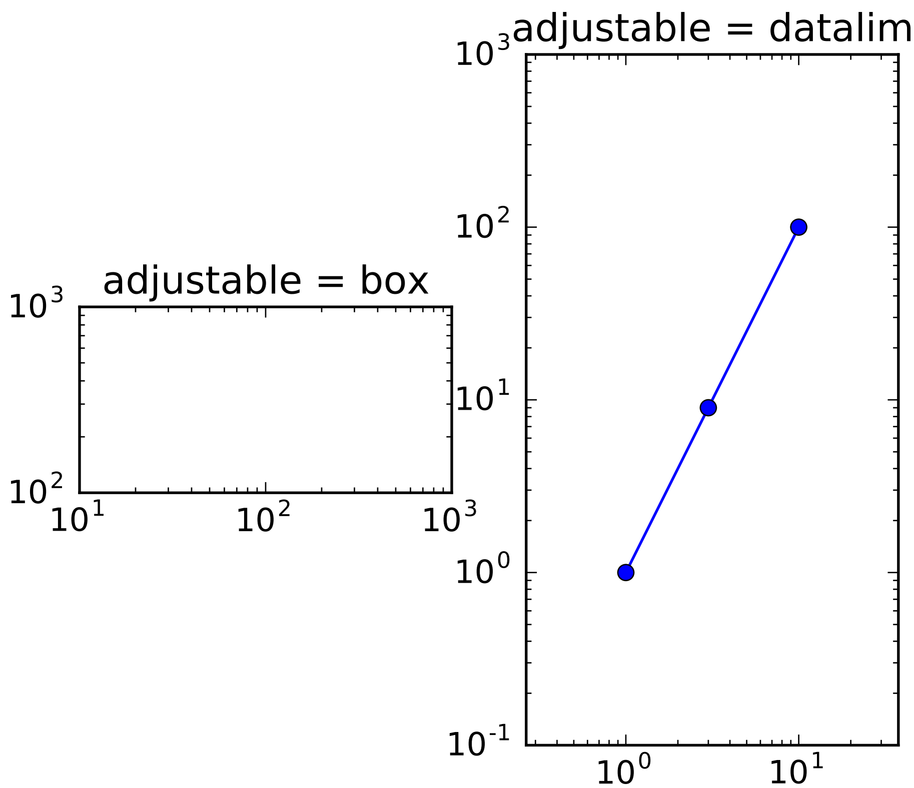
90.13 pylab_examples example code: arrow_simple_demo.py




```
import matplotlib.pyplot as plt

ax = plt.axes()
ax.arrow(0, 0, 0.5, 0.5, head_width=0.05, head_length=0.1, fc='k', ec='k')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

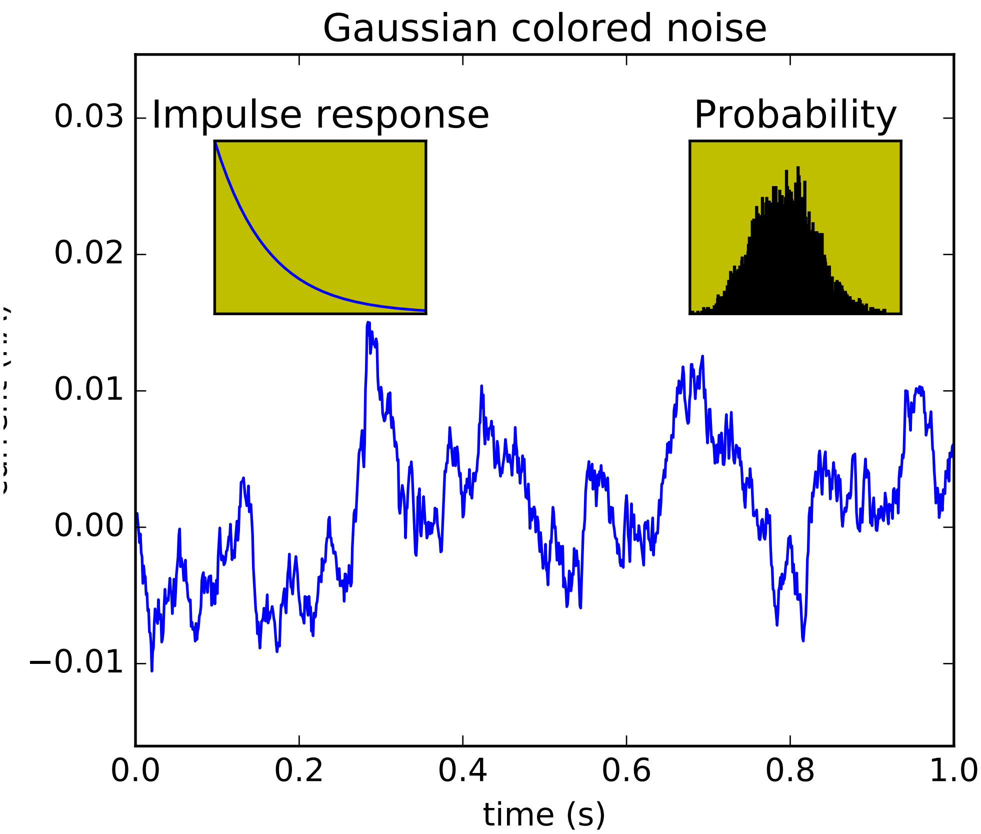
90.14 pylab_examples example code: aspect_loglog.py




```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_xscale("log")
ax1.set_yscale("log")
ax1.set_xlim(1e1, 1e3)
ax1.set_ylim(1e2, 1e3)
ax1.set_aspect(1)
ax1.set_title("adjustable = box")

ax2.set_xscale("log")
ax2.set_yscale("log")
ax2.set_adjustable("datalim")
ax2.plot([1, 3, 10], [1, 9, 100], "o-")
ax2.set_xlim(1e-1, 1e2)
ax2.set_ylim(1e-1, 1e3)
ax2.set_aspect(1)
ax2.set_title("adjustable = datalim")

plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.15 pylab_examples example code: axes_demo.py

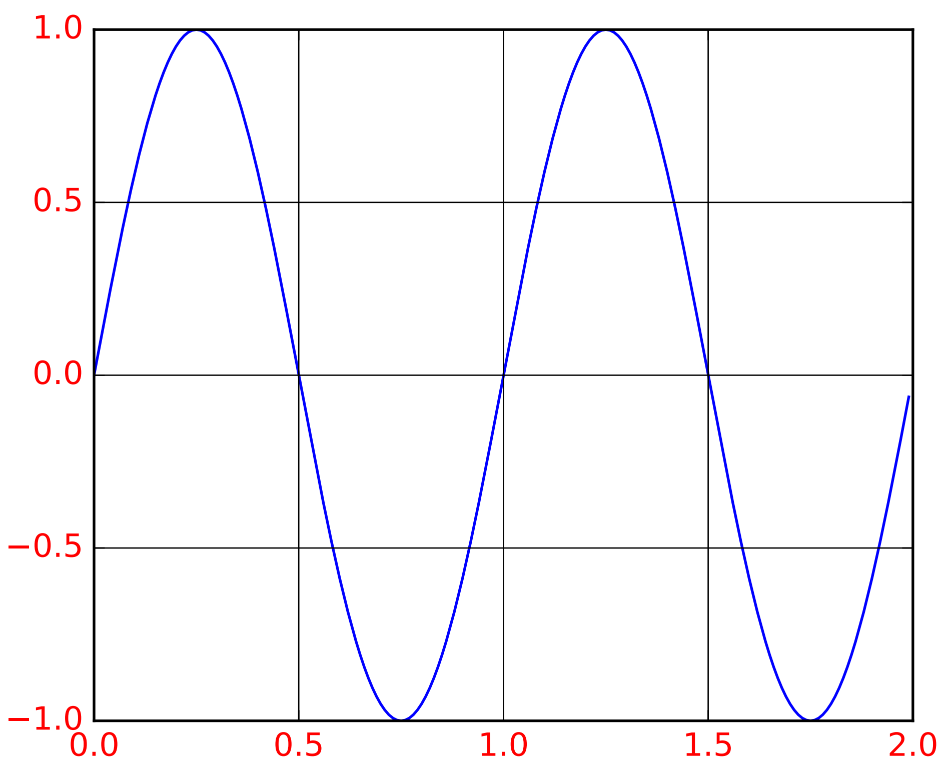



```
import matplotlib.pyplot as plt
import numpy as np

# create some data to use for the plot
dt = 0.001
t = np.arange(0.0, 10.0, dt)
r = np.exp(-t[:1000]/0.05)               # impulse response
x = np.random.randn(len(t))
s = np.convolve(x, r)[:len(x)]*dt  # colored noise

# the main axes is subplot(111) by default
plt.plot(t, s)
plt.axis([0, 1, 1.1*np.amin(s), 2*np.amax(s)])
plt.xlabel('time (s)')
plt.ylabel('current (nA)')
plt.title('Gaussian colored noise')

# this is an inset axes over the main axes
a = plt.axes([.65, .6, .2, .2], axisbg='y')
```

```
n, bins, patches = plt.hist(s, 400, normed=1)
plt.title('Probability')
plt.xticks([])
plt.yticks([])

# this is another inset axes over the main axes
a = plt.axes([0.2, 0.6, .2, .2], axisbg='y')
plt.plot(t[:len(r)], r)
plt.title('Impulse response')
plt.xlim(0, 0.2)
plt.xticks([])
plt.yticks([])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.16 pylab_examples example code: axes_props.py

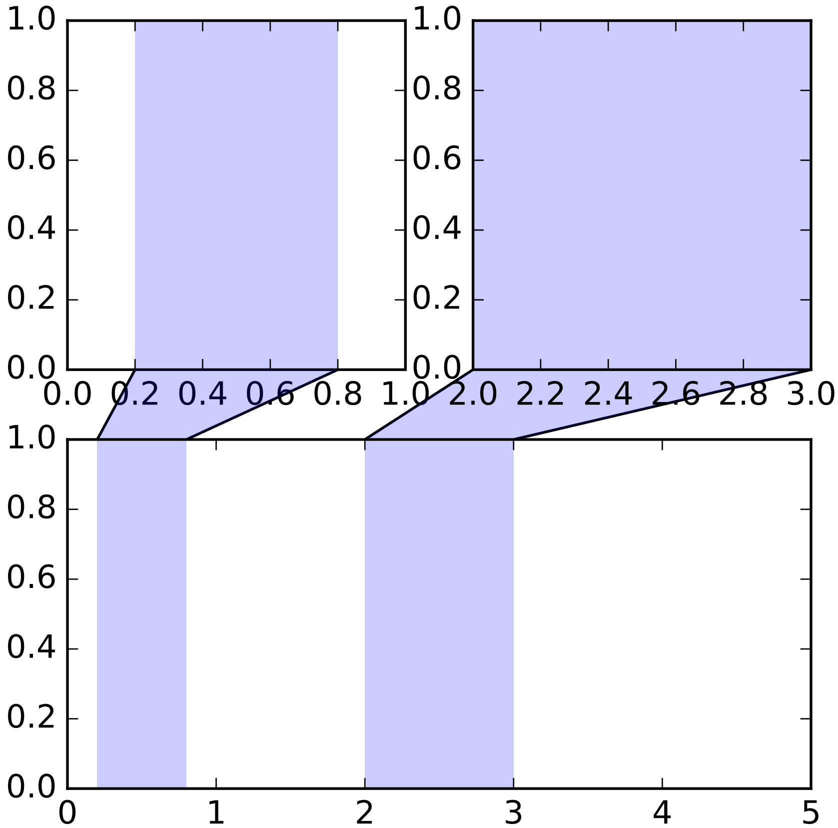



```
"""
You can control the axis tick and grid properties
"""
```

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2 * np.pi * t)
fig, ax = plt.subplots()
ax.plot(t, s)
ax.grid(True)

ticklines = ax.get_xticklines() + ax.get_yticklines()
gridlines = ax.get_xgridlines() + ax.get_ygridlines()
ticklabels = ax.get_xticklabels() + ax.get_yticklabels()

for line in ticklines:
    line.set_linewidth(3)

for line in gridlines:
    line.set_linestyle('-')

for label in ticklabels:
    label.set_color('r')
    label.set_fontsize('medium')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.17 pylab_examples example code: axes_zoom_effect.py



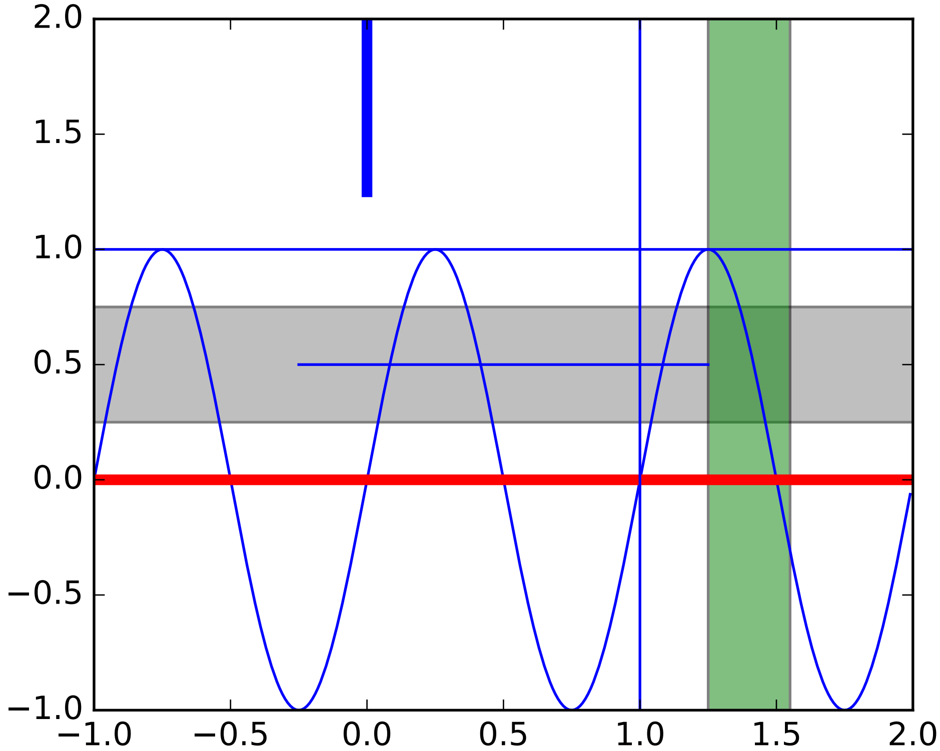

```
from matplotlib.transforms import Bbox, TransformedBbox, \
    blended_transform_factory

from mpl_toolkits.axes_grid1.inset_locator import BboxPatch, BboxConnector,\
    BboxConnectorPatch


def connect_bbox(bbox1, bbox2,
                 loc1a, loc2a, loc1b, loc2b,
                 prop_lines, prop_patches=None):
    if prop_patches is None:
        prop_patches = prop_lines.copy()
        prop_patches["alpha"] = prop_patches.get("alpha", 1)*0.2

    c1 = BboxConnector(bbox1, bbox2, loc1=loc1a, loc2=loc2a, **prop_lines)
    c1.set_clip_on(False)
    c2 = BboxConnector(bbox1, bbox2, loc1=loc1b, loc2=loc2b, **prop_lines)
    c2.set_clip_on(False)
```

```
    bbox_patch1 = BboxPatch(bbox1, **prop_patches)
    bbox_patch2 = BboxPatch(bbox2, **prop_patches)

    p = BboxConnectorPatch(bbox1, bbox2,
                           # loc1a=3, loc2a=2, loc1b=4, loc2b=1,
                           loc1a=loc1a, loc2a=loc2a, loc1b=loc1b, loc2b=loc2b,
                           **prop_patches)
    p.set_clip_on(False)

    return c1, c2, bbox_patch1, bbox_patch2, p


def zoom_effect01(ax1, ax2, xmin, xmax, **kwargs):
    """
    ax1 : the main axes
    ax1 : the zoomed axes
    (xmin,xmax) : the limits of the colored area in both plot axes.

    connect ax1 & ax2. The x-range of (xmin, xmax) in both axes will
    be marked.  The keywords parameters will be used ti create
    patches.

    """

    trans1 = blended_transform_factory(ax1.transData, ax1.transAxes)
    trans2 = blended_transform_factory(ax2.transData, ax2.transAxes)

    bbox = Bbox.from_extents(xmin, 0, xmax, 1)

    mybbox1 = TransformedBbox(bbox, trans1)
    mybbox2 = TransformedBbox(bbox, trans2)

    prop_patches = kwargs.copy()
    prop_patches["ec"] = "none"
    prop_patches["alpha"] = 0.2

    c1, c2, bbox_patch1, bbox_patch2, p = \
        connect_bbox(mybbox1, mybbox2,
                     loc1a=3, loc2a=2, loc1b=4, loc2b=1,
                     prop_lines=kwargs, prop_patches=prop_patches)

    ax1.add_patch(bbox_patch1)
    ax2.add_patch(bbox_patch2)
    ax2.add_patch(c1)
    ax2.add_patch(c2)
    ax2.add_patch(p)

    return c1, c2, bbox_patch1, bbox_patch2, p


def zoom_effect02(ax1, ax2, **kwargs):
    """
    ax1 : the main axes
```

```
    ax1 : the zoomed axes

    Similar to zoom_effect01.  The xmin & xmax will be taken from the
    ax1.viewLim.
    """

    tt = ax1.transScale + (ax1.transLimits + ax2.transAxes)
    trans = blended_transform_factory(ax2.transData, tt)

    mybbox1 = ax1.bbox
    mybbox2 = TransformedBbox(ax1.viewLim, trans)

    prop_patches = kwargs.copy()
    prop_patches["ec"] = "none"
    prop_patches["alpha"] = 0.2

    c1, c2, bbox_patch1, bbox_patch2, p = \
        connect_bbox(mybbox1, mybbox2,
                     loc1a=3, loc2a=2, loc1b=4, loc2b=1,
                     prop_lines=kwargs, prop_patches=prop_patches)

    ax1.add_patch(bbox_patch1)
    ax2.add_patch(bbox_patch2)
    ax2.add_patch(c1)
    ax2.add_patch(c2)
    ax2.add_patch(p)

    return c1, c2, bbox_patch1, bbox_patch2, p


import matplotlib.pyplot as plt

plt.figure(1, figsize=(5, 5))
ax1 = plt.subplot(221)
ax2 = plt.subplot(212)
ax2.set_xlim(0, 1)
ax2.set_xlim(0, 5)
zoom_effect01(ax1, ax2, 0.2, 0.8)


ax1 = plt.subplot(222)
ax1.set_xlim(2, 3)
ax2.set_xlim(0, 5)
zoom_effect02(ax1, ax2)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.18 pylab_examples example code: axhspan_demo.py

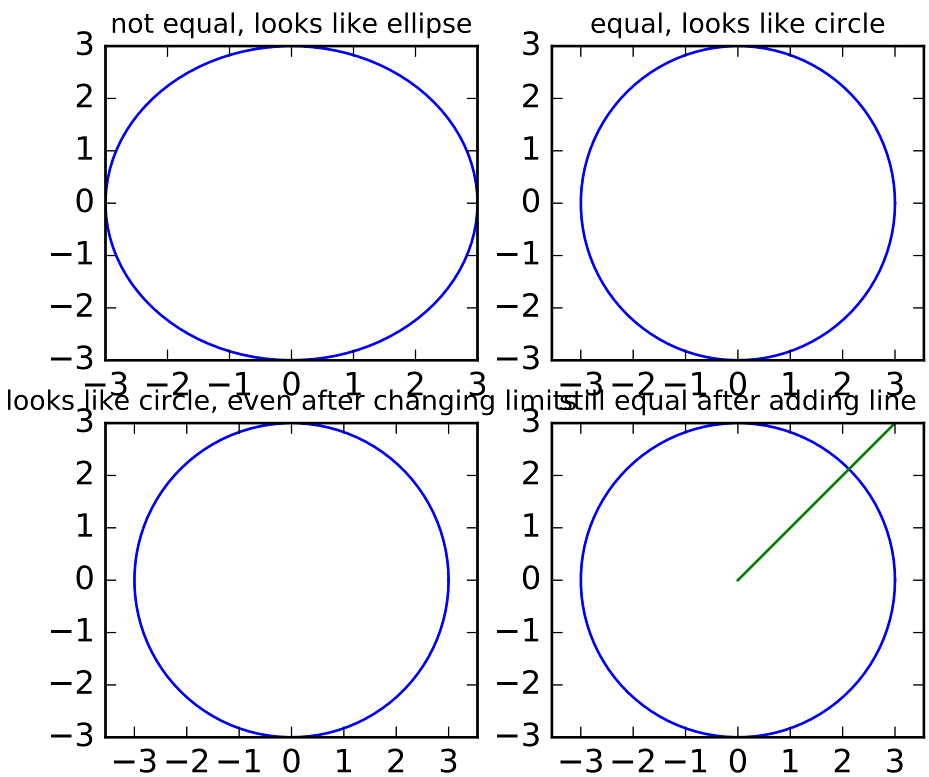



```
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(-1, 2, .01)
s = np.sin(2*np.pi*t)

plt.plot(t, s)
# draw a thick red hline at y=0 that spans the xrange
l = plt.axhline(linewidth=4, color='r')

# draw a default hline at y=1 that spans the xrange
l = plt.axhline(y=1)

# draw a default vline at x=1 that spans the yrange
l = plt.axvline(x=1)

# draw a thick blue vline at x=0 that spans the upper quadrant of
# the yrange
l = plt.axvline(x=0, ymin=0.75, linewidth=4, color='b')

# draw a default hline at y=.5 that spans the middle half of
# the axes
```

```
l = plt.axhline(y=.5, xmin=0.25, xmax=0.75)

p = plt.axhspan(0.25, 0.75, facecolor='0.5', alpha=0.5)

p = plt.axvspan(1.25, 1.55, facecolor='g', alpha=0.5)

plt.axis([-1, 2, -1, 2])


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.19 pylab_examples example code: axis_equal_demo.py

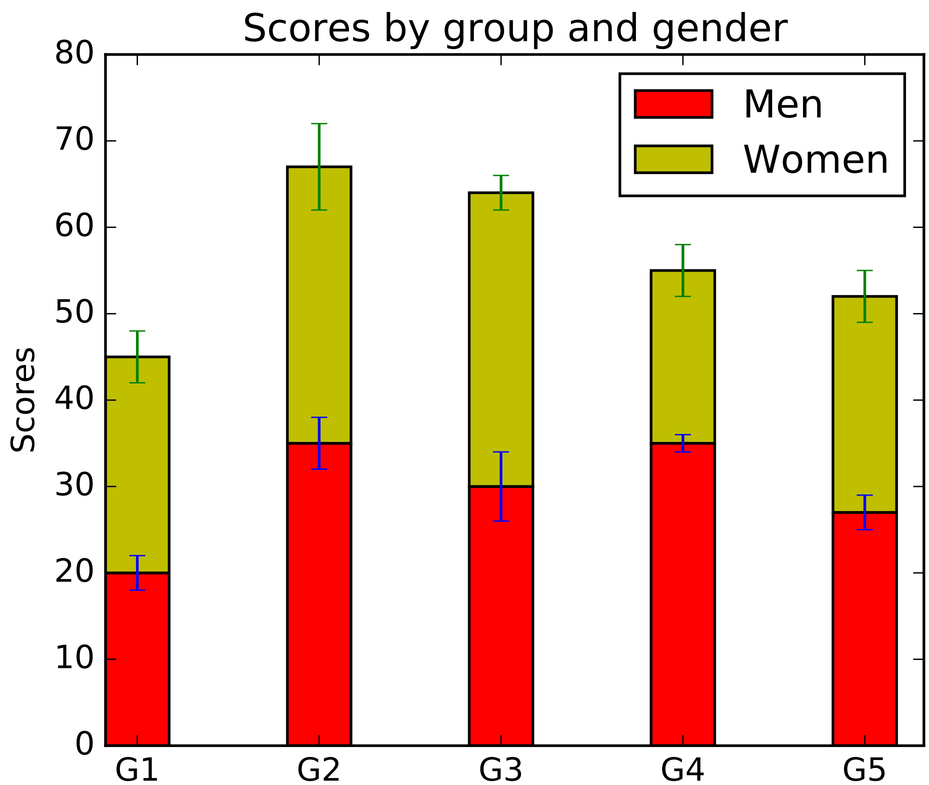



```
'''This example is only interesting when ran in interactive mode'''

import matplotlib.pyplot as plt
import numpy as np

# Plot circle or radius 3
```

```
an = np.linspace(0, 2*np.pi, 100)

plt.subplot(221)
plt.plot(3*np.cos(an), 3*np.sin(an))
plt.title('not equal, looks like ellipse', fontsize=10)

plt.subplot(222)
plt.plot(3*np.cos(an), 3*np.sin(an))
plt.axis('equal')
plt.title('equal, looks like circle', fontsize=10)

plt.subplot(223)
plt.plot(3*np.cos(an), 3*np.sin(an))
plt.axis('equal')
plt.axis([-3, 3, -3, 3])
plt.title('looks like circle, even after changing limits', fontsize=10)

plt.subplot(224)
plt.plot(3*np.cos(an), 3*np.sin(an))
plt.axis('equal')
plt.axis([-3, 3, -3, 3])
plt.plot([0, 4], [0, 4])
plt.title('still equal after adding line', fontsize=10)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.20 pylab_examples example code: bar_stacked.py

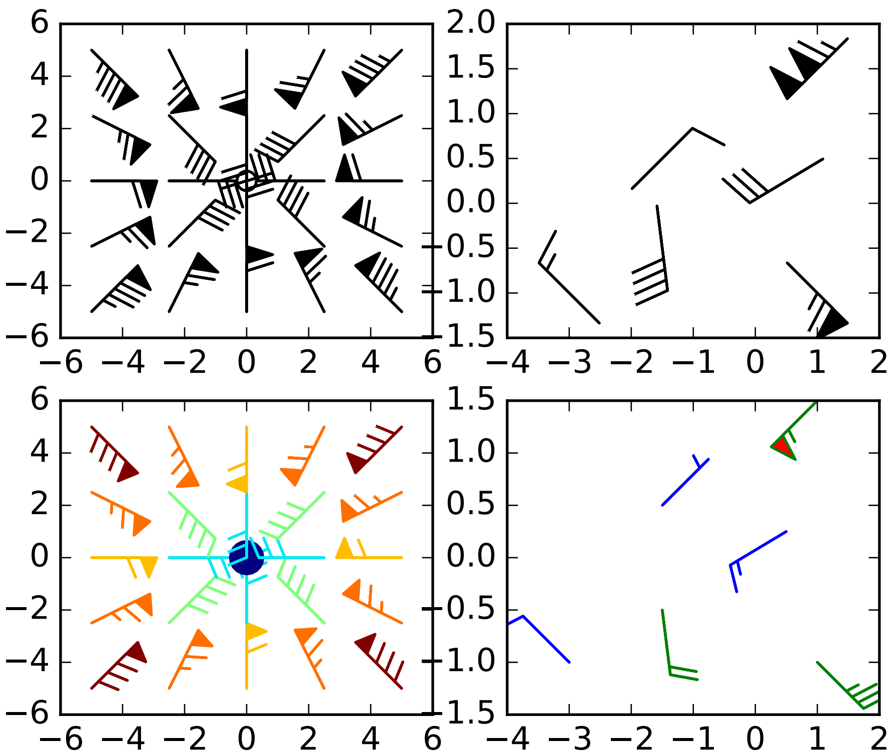


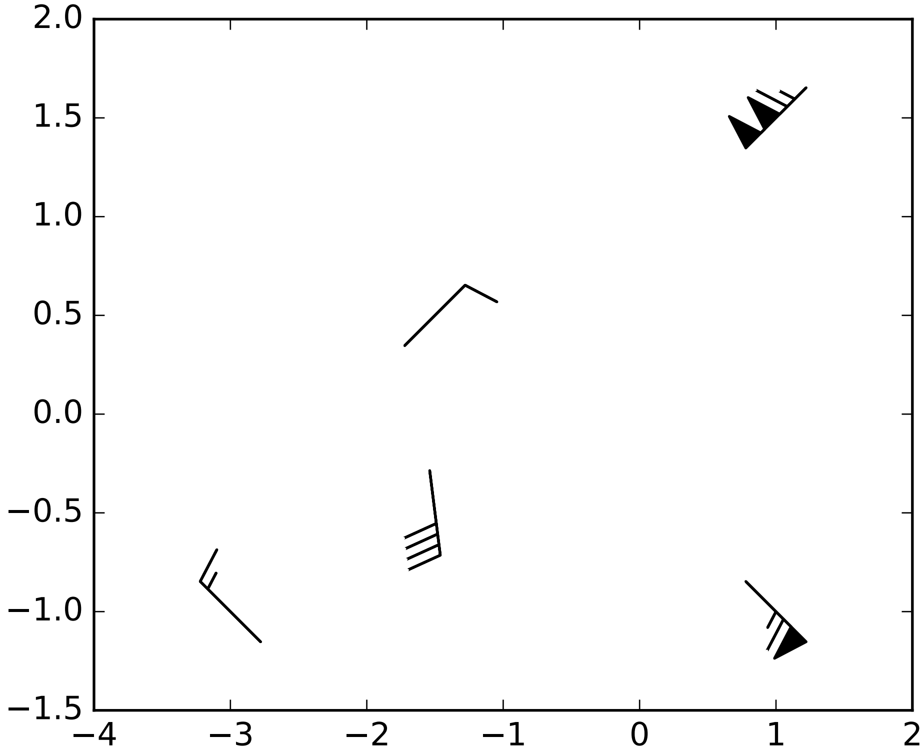

```
#!/usr/bin/env python
# a stacked bar plot with errorbars
import numpy as np
import matplotlib.pyplot as plt


N = 5
menMeans = (20, 35, 30, 35, 27)
womenMeans = (25, 32, 34, 20, 25)
menStd = (2, 3, 4, 1, 2)
womenStd = (3, 5, 2, 3, 3)
ind = np.arange(N)    # the x locations for the groups
width = 0.35       # the width of the bars: can also be len(x) sequence

p1 = plt.bar(ind, menMeans, width, color='r', yerr=menStd)
p2 = plt.bar(ind, womenMeans, width, color='y',
             bottom=menMeans, yerr=womenStd)

plt.ylabel('Scores')
plt.title('Scores by group and gender')
plt.xticks(ind + width/2., ('G1', 'G2', 'G3', 'G4', 'G5'))
plt.yticks(np.arange(0, 81, 10))
```

```
plt.legend((p1[0], p2[0]), ('Men', 'Women'))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.21 pylab_examples example code: barb_demo.py



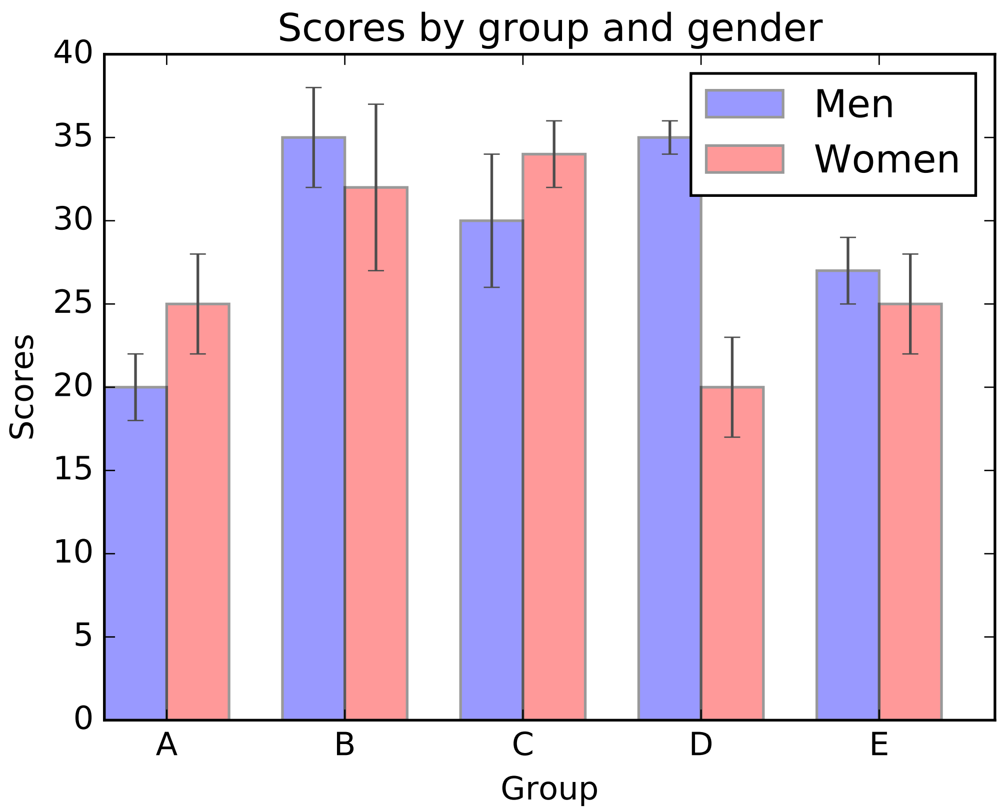



```python
"""
Demonstration of wind barb plots
"""
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-5, 5, 5)
X, Y = np.meshgrid(x, x)
U, V = 12*X, 12*Y

data = [(-1.5, .5, -6, -6),
        (1, -1, -46, 46),
        (-3, -1, 11, -11),
        (1, 1.5, 80, 80),
        (0.5, 0.25, 25, 15),
        (-1.5, -0.5, -5, 40)]

data = np.array(data, dtype=[('x', np.float32), ('y', np.float32),
                             ('u', np.float32), ('v', np.float32)])

# Default parameters, uniform grid
ax = plt.subplot(2, 2, 1)
ax.barbs(X, Y, U, V)

# Arbitrary set of vectors, make them longer and change the pivot point
#(point around which they're rotated) to be the middle
```

```
ax = plt.subplot(2, 2, 2)
ax.barbs(data['x'], data['y'], data['u'], data['v'], length=8, pivot='middle')

# Showing colormapping with uniform grid.  Fill the circle for an empty barb,
# don't round the values, and change some of the size parameters
ax = plt.subplot(2, 2, 3)
ax.barbs(X, Y, U, V, np.sqrt(U*U + V*V), fill_empty=True, rounding=False,
         sizes=dict(emptybarb=0.25, spacing=0.2, height=0.3))

# Change colors as well as the increments for parts of the barbs
ax = plt.subplot(2, 2, 4)
ax.barbs(data['x'], data['y'], data['u'], data['v'], flagcolor='r',
         barbcolor=['b', 'g'], barb_increments=dict(half=10, full=20, flag=100),
         flip_barb=True)

# Masked arrays are also supported
masked_u = np.ma.masked_array(data['u'])
masked_u[4] = 1000  # Bad value that should not be plotted when masked
masked_u[4] = np.ma.masked

# Identical plot to panel 2 in the first figure, but with the point at
#(0.5, 0.25) missing (masked)
fig2 = plt.figure()
ax = fig2.add_subplot(1, 1, 1)
ax.barbs(data['x'], data['y'], masked_u, data['v'], length=8, pivot='middle')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.22 pylab_examples example code: barchart_demo.py

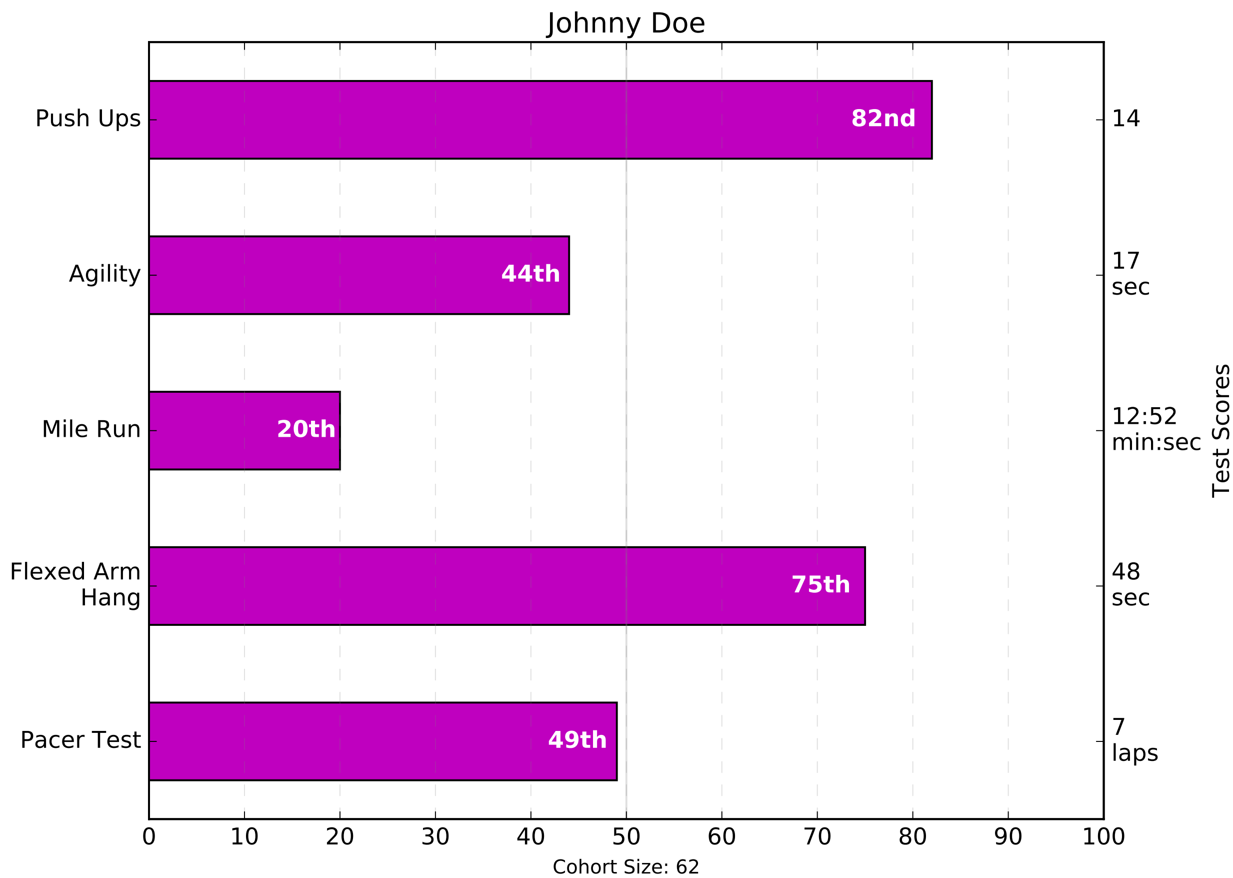



```
"""
Bar chart demo with pairs of bars grouped for easy comparison.
"""
import numpy as np
import matplotlib.pyplot as plt


n_groups = 5

means_men = (20, 35, 30, 35, 27)
std_men = (2, 3, 4, 1, 2)

means_women = (25, 32, 34, 20, 25)
std_women = (3, 5, 2, 3, 3)

fig, ax = plt.subplots()

index = np.arange(n_groups)
bar_width = 0.35

opacity = 0.4
error_config = {'ecolor': '0.3'}
```

```
rects1 = plt.bar(index, means_men, bar_width,
                 alpha=opacity,
                 color='b',
                 yerr=std_men,
                 error_kw=error_config,
                 label='Men')

rects2 = plt.bar(index + bar_width, means_women, bar_width,
                 alpha=opacity,
                 color='r',
                 yerr=std_women,
                 error_kw=error_config,
                 label='Women')

plt.xlabel('Group')
plt.ylabel('Scores')
plt.title('Scores by group and gender')
plt.xticks(index + bar_width, ('A', 'B', 'C', 'D', 'E'))
plt.legend()

plt.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.23 pylab_examples example code: barchart_demo2.py




```
"""
Thanks Josh Hemann for the example

This examples comes from an application in which grade school gym
teachers wanted to be able to show parents how their child did across
a handful of fitness tests, and importantly, relative to how other
children did. To extract the plotting code for demo purposes, we'll
just make up some data for little Johnny Doe...

"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from collections import namedtuple

Student = namedtuple('Student', ['name', 'grade', 'gender'])
Score = namedtuple('Score', ['score', 'percentile'])

# GLOBAL CONSTANTS
testNames = ['Pacer Test', 'Flexed Arm\n Hang', 'Mile Run', 'Agility',
```

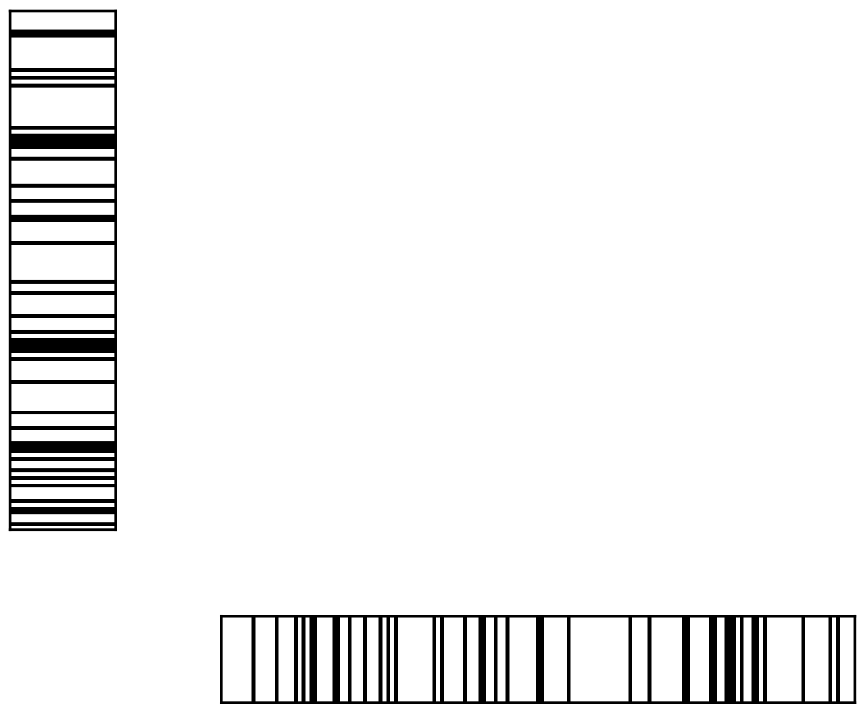
```
             'Push Ups']
testMeta = dict(zip(testNames, ['laps', 'sec', 'min:sec', 'sec', '']))


def attach_ordinal(num):
    """helper function to add ordinal string to integers

    1 -> 1st
    56 -> 56th
    """
    suffixes = dict((str(i), v) for i, v in
                    enumerate(['th', 'st', 'nd', 'rd', 'th',
                               'th', 'th', 'th', 'th', 'th']))

    v = str(num)
    # special case early teens
    if v in {'11', '12', '13'}:
        return v + 'th'
    return v + suffixes[v[-1]]


def format_score(scr, test):
    """
    Build up the score labels for the right Y-axis by first
    appending a carriage return to each string and then tacking on
    the appropriate meta information (i.e., 'laps' vs 'seconds'). We
    want the labels centered on the ticks, so if there is no meta
    info (like for pushups) then don't add the carriage return to
    the string
    """
    md = testMeta[test]
    if md:
        return '{0}\n{1}'.format(scr, md)
    else:
        return scr


def format_ycursor(y):
    y = int(y)
    if y < 0 or y >= len(testNames):
        return ''
    else:
        return testNames[y]


def plot_student_results(student, scores, cohort_size):
    #  create the figure
    fig, ax1 = plt.subplots(figsize=(9, 7))
    fig.subplots_adjust(left=0.115, right=0.88)
    fig.canvas.set_window_title('Eldorado K-8 Fitness Chart')

    pos = np.arange(len(testNames)) + 0.5  # Center bars on the Y-axis ticks
```

```
    rects = ax1.barh(pos, [scores[k].percentile for k in testNames],
                     align='center',
                     height=0.5, color='m',
                     tick_label=testNames)

    ax1.set_title(student.name)

    ax1.set_xlim([0, 100])
    ax1.xaxis.set_major_locator(MaxNLocator(11))
    ax1.xaxis.grid(True, linestyle='--', which='major',
                   color='grey', alpha=.25)

    # Plot a solid vertical gridline to highlight the median position
    ax1.axvline(50, color='grey', alpha=0.25)
    # set X-axis tick marks at the deciles
    cohort_label = ax1.text(.5, -.07, 'Cohort Size: {0}'.format(cohort_size),
                            horizontalalignment='center', size='small',
                            transform=ax1.transAxes)

    # Set the right-hand Y-axis ticks and labels
    ax2 = ax1.twinx()

    scoreLabels = [format_score(scores[k].score, k) for k in testNames]

    # set the tick locations
    ax2.set_yticks(pos)
    # make sure that the limits are set equally on both yaxis so the
    # ticks line up
    ax2.set_ylim(ax1.get_ylim())

    # set the tick labels
    ax2.set_yticklabels(scoreLabels)

    ax2.set_ylabel('Test Scores')

    ax2.set_xlabel(('Percentile Ranking Across '
                    '{grade} Grade {gender}s').format(
                        grade=attach_ordinal(student.grade),
                        gender=student.gender.title()))

    rect_labels = []
    # Lastly, write in the ranking inside each bar to aid in interpretation
    for rect in rects:
        # Rectangle widths are already integer-valued but are floating
        # type, so it helps to remove the trailing decimal point and 0 by
        # converting width to int type
        width = int(rect.get_width())

        rankStr = attach_ordinal(width)
        # The bars aren't wide enough to print the ranking inside
        if (width < 5):
            # Shift the text to the right side of the right edge
            xloc = width + 1
```

```
            # Black against white background
            clr = 'black'
            align = 'left'
        else:
            # Shift the text to the left side of the right edge
            xloc = 0.98*width
            # White on magenta
            clr = 'white'
            align = 'right'

        # Center the text vertically in the bar
        yloc = rect.get_y() + rect.get_height()/2.0
        label = ax1.text(xloc, yloc, rankStr, horizontalalignment=align,
                         verticalalignment='center', color=clr, weight='bold',
                         clip_on=True)
        rect_labels.append(label)

    # make the interactive mouse over give the bar title
    ax2.fmt_ydata = format_ycursor
    # return all of the artists created
    return {'fig': fig,
            'ax': ax1,
            'ax_right': ax2,
            'bars': rects,
            'perc_labels': rect_labels,
            'cohort_label': cohort_label}

student = Student('Johnny Doe', 2, 'boy')
scores = dict(zip(testNames,
                  (Score(v, p) for v, p in
                   zip(['7', '48', '12:52', '17', '14'],
                       np.round(np.random.uniform(0, 1,
                                                  len(testNames))*100, 0)))))
cohort_size = 62  # The number of other 2nd grade boys

arts = plot_student_results(student, scores, cohort_size)
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.24 pylab_examples example code: barcode_demo.py

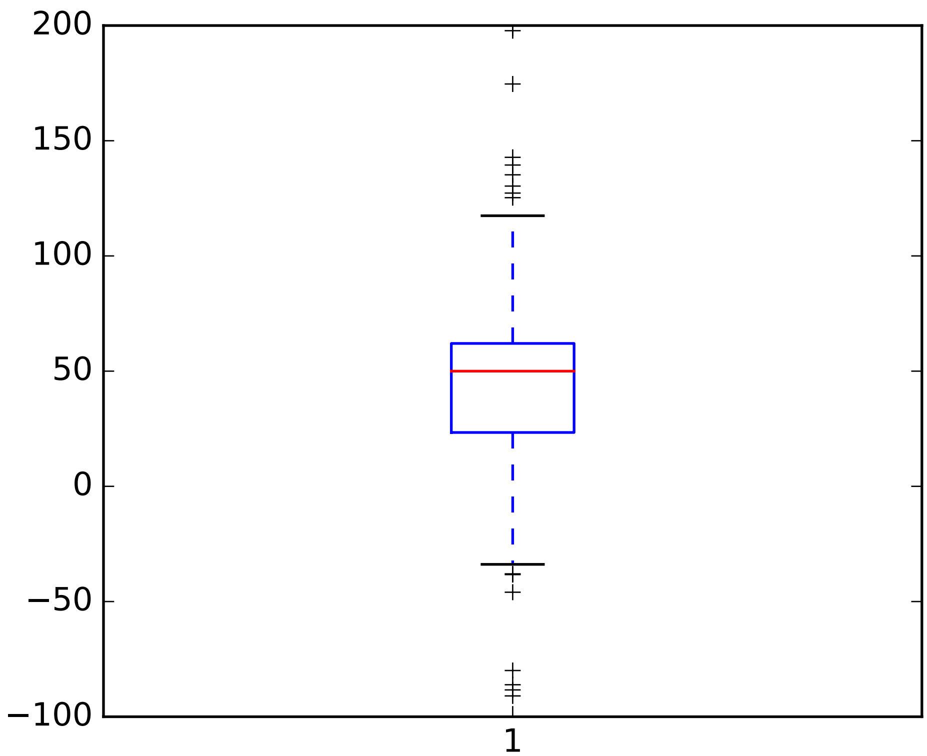


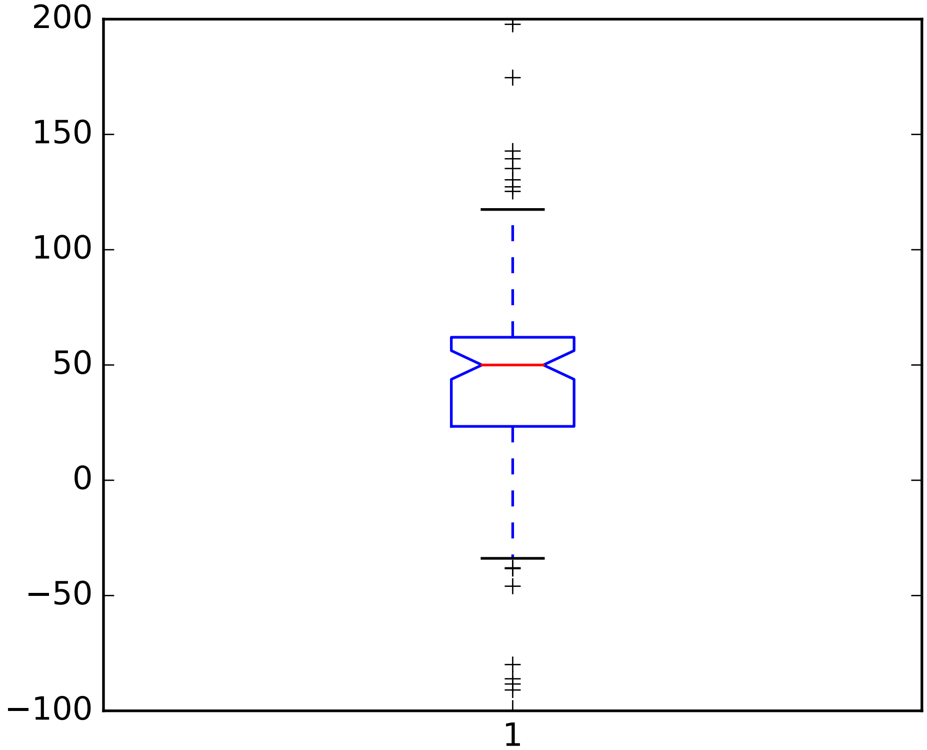
```
import matplotlib.pyplot as plt
import numpy as np

# the bar
x = np.where(np.random.rand(500) > 0.7, 1.0, 0.0)

axprops = dict(xticks=[], yticks=[])
barprops = dict(aspect='auto', cmap=plt.cm.binary, interpolation='nearest')

fig = plt.figure()

# a vertical barcode -- this is broken at present
x.shape = len(x), 1
ax = fig.add_axes([0.1, 0.3, 0.1, 0.6], **axprops)
ax.imshow(x, **barprops)

x = x.copy()
# a horizontal barcode
x.shape = 1, len(x)
ax = fig.add_axes([0.3, 0.1, 0.6, 0.1], **axprops)
ax.imshow(x, **barprops)
```

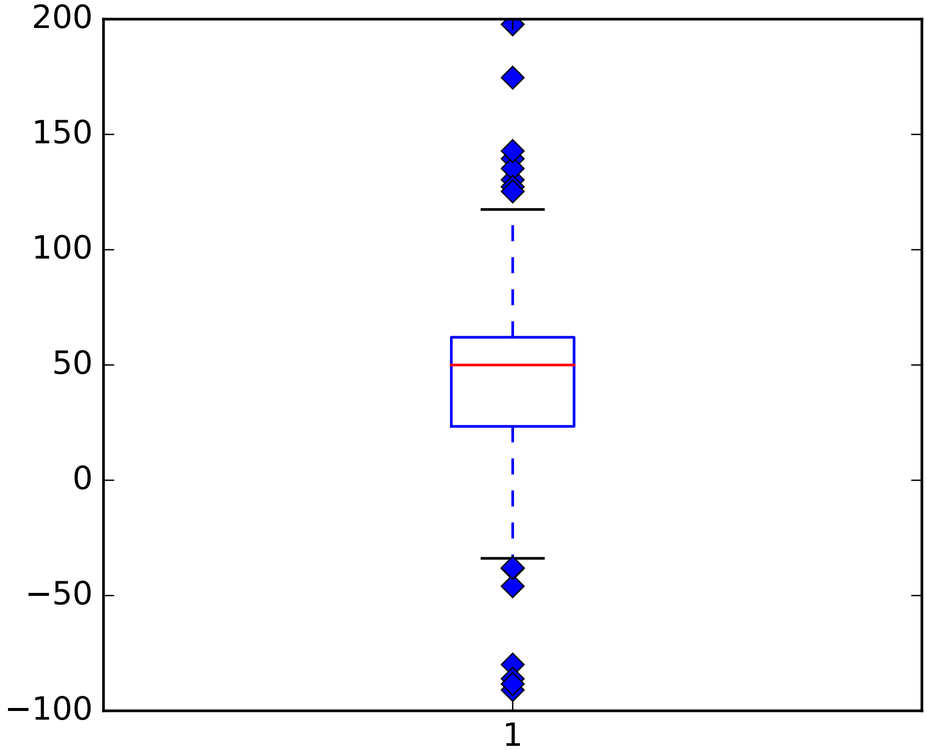
```
plt.show()
```

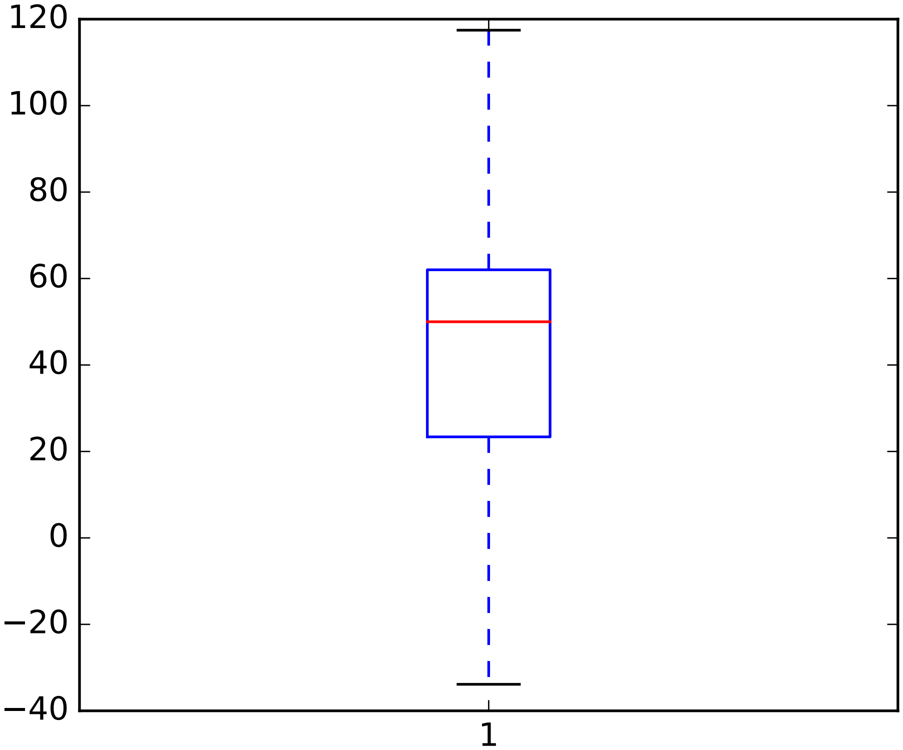
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

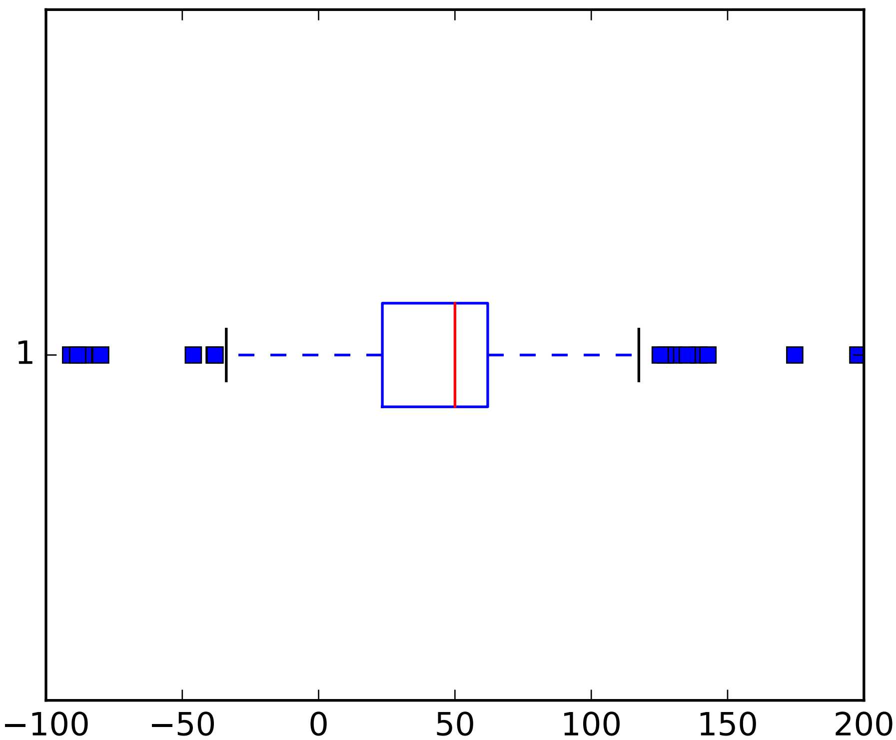
90.25 pylab_examples example code: boxplot_demo.py

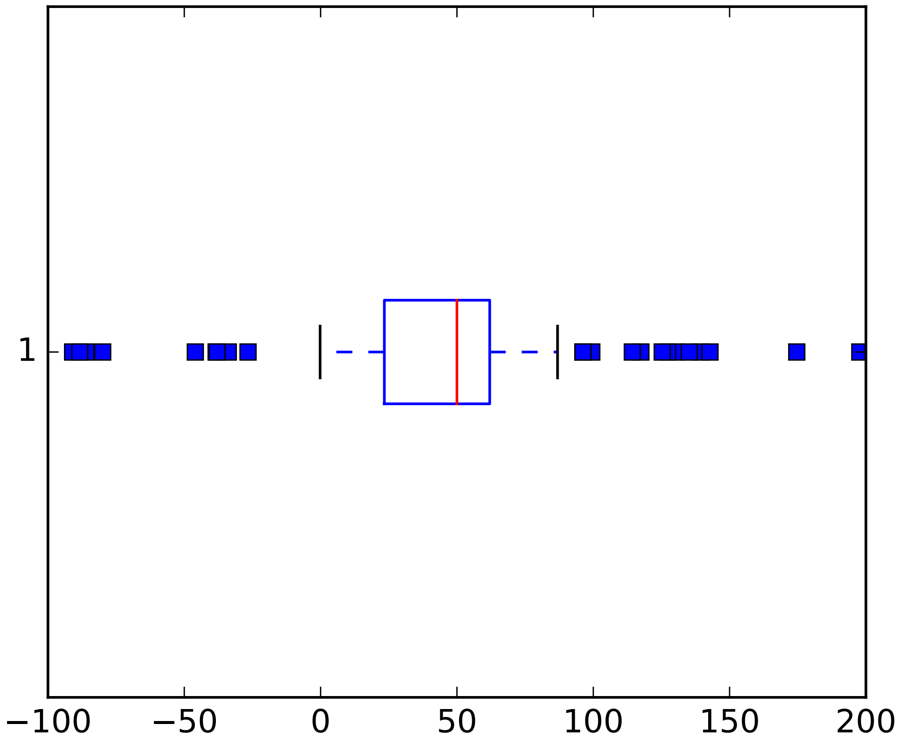


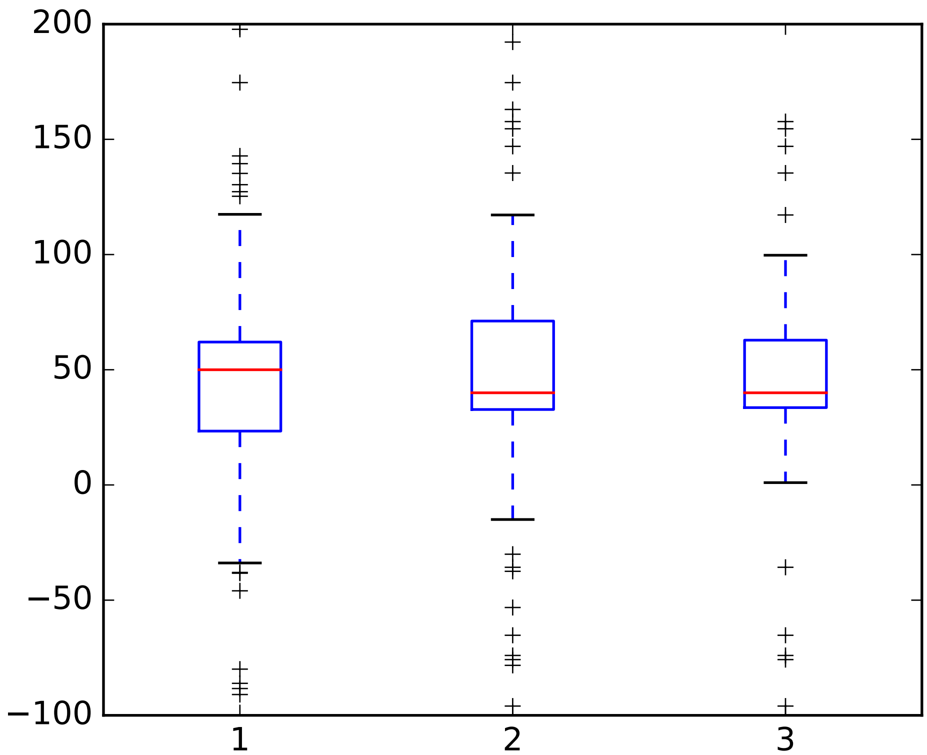

200
150
100
50
0
−50
−100
1


200
150
100
50
0
−50
−100
1


120
100
80
60
40
20
0
−20
−40
1


1
−100
−50
0
50
100
150
200


1
−100
−50
0
50
100
150
200

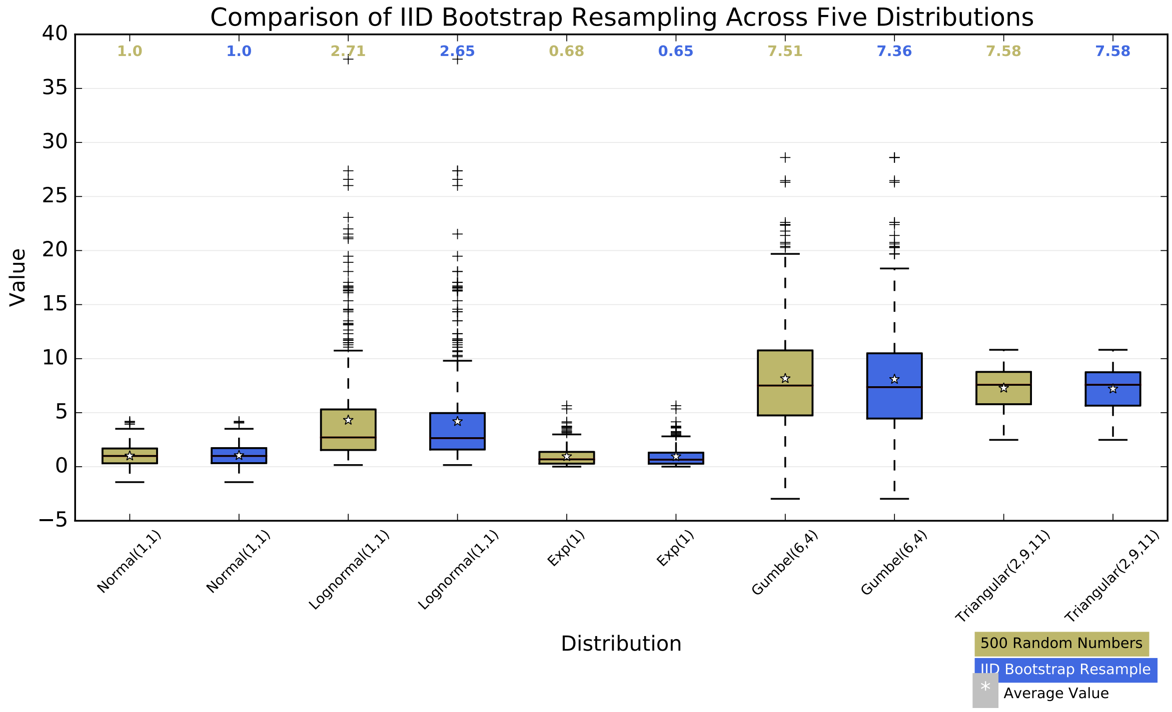



```
import matplotlib.pyplot as plt
import numpy as np

# fake up some data
spread = np.random.rand(50) * 100
center = np.ones(25) * 50
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
data = np.concatenate((spread, center, flier_high, flier_low), 0)

# basic plot
plt.boxplot(data)

# notched plot
plt.figure()
plt.boxplot(data, 1)

# change outlier point symbols
plt.figure()
plt.boxplot(data, 0, 'gD')

# don't show outlier points
plt.figure()
plt.boxplot(data, 0, '')

# horizontal boxes
```

```
plt.figure()
plt.boxplot(data, 0, 'rs', 0)

# change whisker length
plt.figure()
plt.boxplot(data, 0, 'rs', 0, 0.75)

# fake up some more data
spread = np.random.rand(50) * 100
center = np.ones(25) * 40
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
d2 = np.concatenate((spread, center, flier_high, flier_low), 0)
data.shape = (-1, 1)
d2.shape = (-1, 1)
# data = concatenate( (data, d2), 1 )
# Making a 2-D array only works if all the columns are the
# same length.  If they are not, then use a list instead.
# This is actually more efficient because boxplot converts
# a 2-D array into a list of vectors internally anyway.
data = [data, d2, d2[::2, 0]]
# multiple box plots on one figure
plt.figure()
plt.boxplot(data)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.26 pylab_examples example code: boxplot_demo2.py



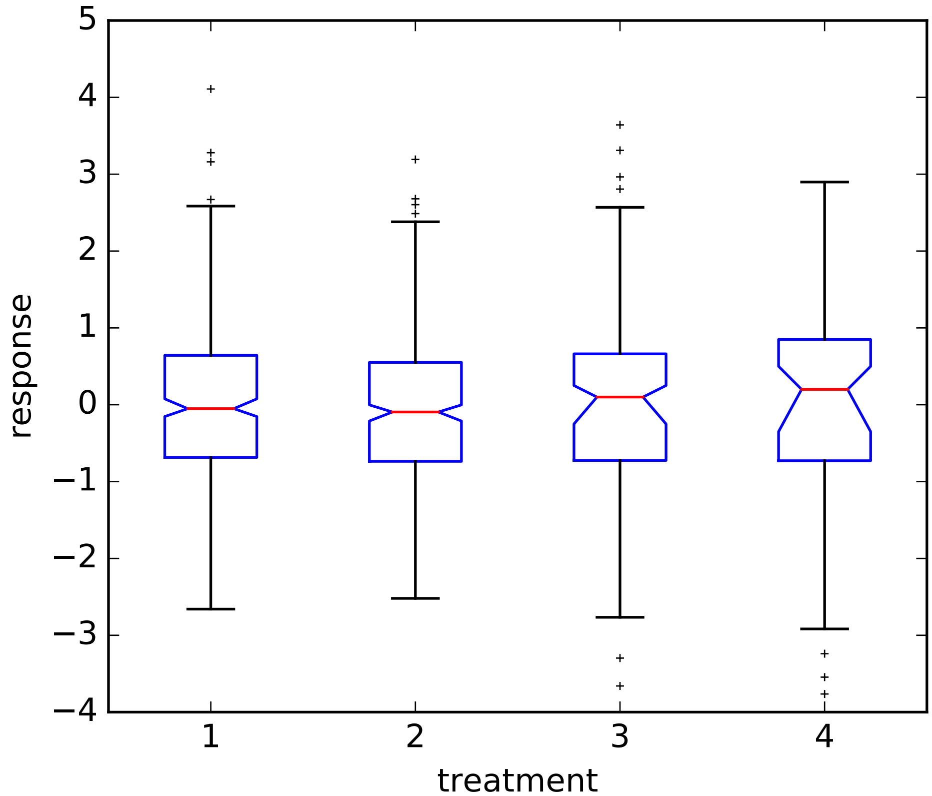

```
"""
Thanks Josh Hemann for the example
"""

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon


# Generate some data from five different probability distributions,
# each with different characteristics. We want to play with how an IID
# bootstrap resample of the data preserves the distributional
# properties of the original sample, and a boxplot is one visual tool
# to make this assessment
numDists = 5
randomDists = ['Normal(1,1)', ' Lognormal(1,1)', 'Exp(1)', 'Gumbel(6,4)',
               'Triangular(2,9,11)']
N = 500
norm = np.random.normal(1, 1, N)
logn = np.random.lognormal(1, 1, N)
expo = np.random.exponential(1, N)
gumb = np.random.gumbel(6, 4, N)
tria = np.random.triangular(2, 9, 11, N)

# Generate some random indices that we'll use to resample the original data
# arrays. For code brevity, just use the same random indices for each array
bootstrapIndices = np.random.random_integers(0, N - 1, N)
```

```
normBoot = norm[bootstrapIndices]
expoBoot = expo[bootstrapIndices]
gumbBoot = gumb[bootstrapIndices]
lognBoot = logn[bootstrapIndices]
triaBoot = tria[bootstrapIndices]

data = [norm, normBoot, logn, lognBoot, expo, expoBoot, gumb, gumbBoot,
        tria, triaBoot]

fig, ax1 = plt.subplots(figsize=(10, 6))
fig.canvas.set_window_title('A Boxplot Example')
plt.subplots_adjust(left=0.075, right=0.95, top=0.9, bottom=0.25)

bp = plt.boxplot(data, notch=0, sym='+', vert=1, whis=1.5)
plt.setp(bp['boxes'], color='black')
plt.setp(bp['whiskers'], color='black')
plt.setp(bp['fliers'], color='red', marker='+')

# Add a horizontal grid to the plot, but make it very light in color
# so we can use it for reading data values but not be distracting
ax1.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
               alpha=0.5)

# Hide these grid behind plot objects
ax1.set_axisbelow(True)
ax1.set_title('Comparison of IID Bootstrap Resampling Across Five Distributions')
ax1.set_xlabel('Distribution')
ax1.set_ylabel('Value')

# Now fill the boxes with desired colors
boxColors = ['darkkhaki', 'royalblue']
numBoxes = numDists*2
medians = list(range(numBoxes))
for i in range(numBoxes):
    box = bp['boxes'][i]
    boxX = []
    boxY = []
    for j in range(5):
        boxX.append(box.get_xdata()[j])
        boxY.append(box.get_ydata()[j])
    boxCoords = list(zip(boxX, boxY))
    # Alternate between Dark Khaki and Royal Blue
    k = i % 2
    boxPolygon = Polygon(boxCoords, facecolor=boxColors[k])
    ax1.add_patch(boxPolygon)
    # Now draw the median lines back over what we just filled in
    med = bp['medians'][i]
    medianX = []
    medianY = []
    for j in range(2):
        medianX.append(med.get_xdata()[j])
        medianY.append(med.get_ydata()[j])
        plt.plot(medianX, medianY, 'k')
```

```
        medians[i] = medianY[0]
    # Finally, overplot the sample averages, with horizontal alignment
    # in the center of each box
    plt.plot([np.average(med.get_xdata())], [np.average(data[i])],
             color='w', marker='*', markeredgecolor='k')

# Set the axes ranges and axes labels
ax1.set_xlim(0.5, numBoxes + 0.5)
top = 40
bottom = -5
ax1.set_ylim(bottom, top)
xtickNames = plt.setp(ax1, xticklabels=np.repeat(randomDists, 2))
plt.setp(xtickNames, rotation=45, fontsize=8)

# Due to the Y-axis scale being different across samples, it can be
# hard to compare differences in medians across the samples. Add upper
# X-axis tick labels with the sample medians to aid in comparison
# (just use two decimal places of precision)
pos = np.arange(numBoxes) + 1
upperLabels = [str(np.round(s, 2)) for s in medians]
weights = ['bold', 'semibold']
for tick, label in zip(range(numBoxes), ax1.get_xticklabels()):
    k = tick % 2
    ax1.text(pos[tick], top - (top*0.05), upperLabels[tick],
             horizontalalignment='center', size='x-small', weight=weights[k],
             color=boxColors[k])

# Finally, add a basic legend
plt.figtext(0.80, 0.08, str(N) + ' Random Numbers',
            backgroundcolor=boxColors[0], color='black', weight='roman',
            size='x-small')
plt.figtext(0.80, 0.045, 'IID Bootstrap Resample',
            backgroundcolor=boxColors[1],
            color='white', weight='roman', size='x-small')
plt.figtext(0.80, 0.015, '*', color='white', backgroundcolor='silver',
            weight='roman', size='medium')
plt.figtext(0.815, 0.013, ' Average Value', color='black', weight='roman',
            size='x-small')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.27 pylab_examples example code: boxplot_demo3.py

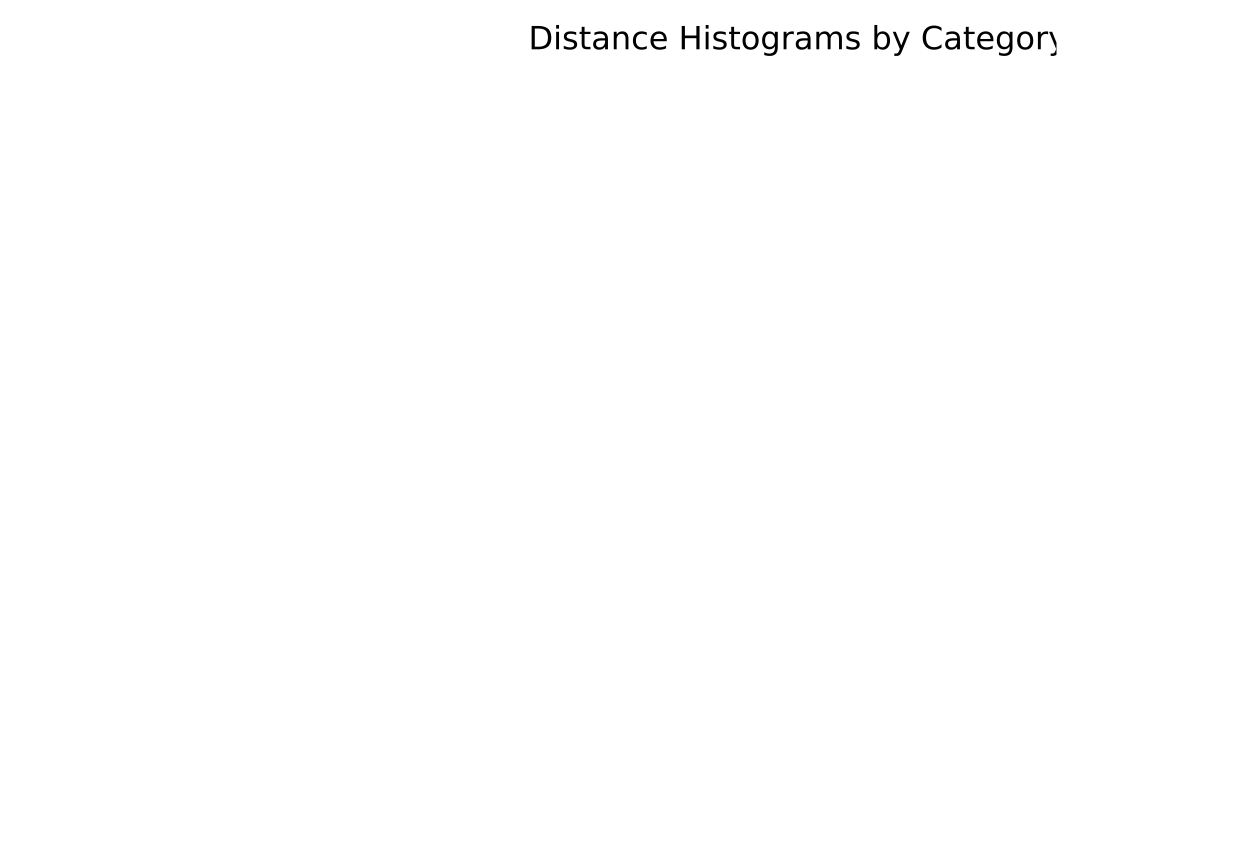



```
import matplotlib.pyplot as plt
import numpy as np


def fakeBootStrapper(n):
    '''
    This is just a placeholder for the user's method of
    bootstrapping the median and its confidence intervals.

    Returns an arbitrary median and confidence intervals
    packed into a tuple
    '''
    if n == 1:
        med = 0.1
        CI = (-0.25, 0.25)
    else:
        med = 0.2
        CI = (-0.35, 0.50)

    return med, CI
```

```
np.random.seed(2)
inc = 0.1
e1 = np.random.normal(0, 1, size=(500,))
e2 = np.random.normal(0, 1, size=(500,))
e3 = np.random.normal(0, 1 + inc, size=(500,))
e4 = np.random.normal(0, 1 + 2*inc, size=(500,))

treatments = [e1, e2, e3, e4]
med1, CI1 = fakeBootStrapper(1)
med2, CI2 = fakeBootStrapper(2)
medians = [None, None, med1, med2]
conf_intervals = [None, None, CI1, CI2]

fig, ax = plt.subplots()
pos = np.array(range(len(treatments))) + 1
bp = ax.boxplot(treatments, sym='k+', positions=pos,
                notch=1, bootstrap=5000,
                usermedians=medians,
                conf_intervals=conf_intervals)

ax.set_xlabel('treatment')
ax.set_ylabel('response')
plt.setp(bp['whiskers'], color='k', linestyle='-')
plt.setp(bp['fliers'], markersize=3.0)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

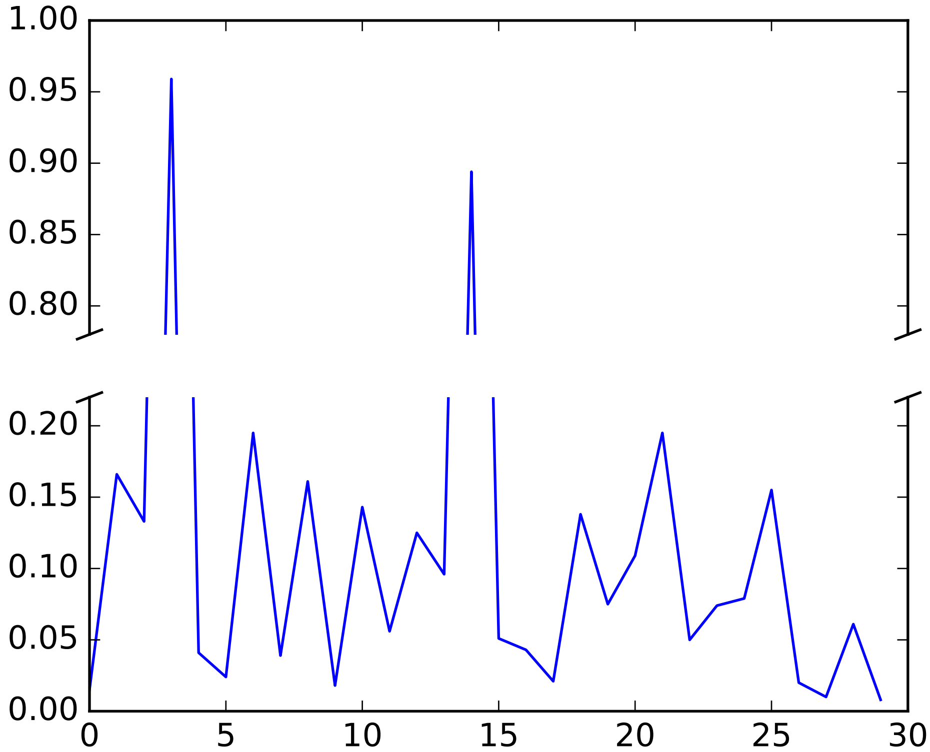
90.28 pylab_examples example code: break.py




```
import matplotlib.pyplot as plt

plt.gcf().text(0.5, 0.95, 'Distance Histograms by Category is \
    a really long title')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.29 pylab_examples example code: broken_axis.py

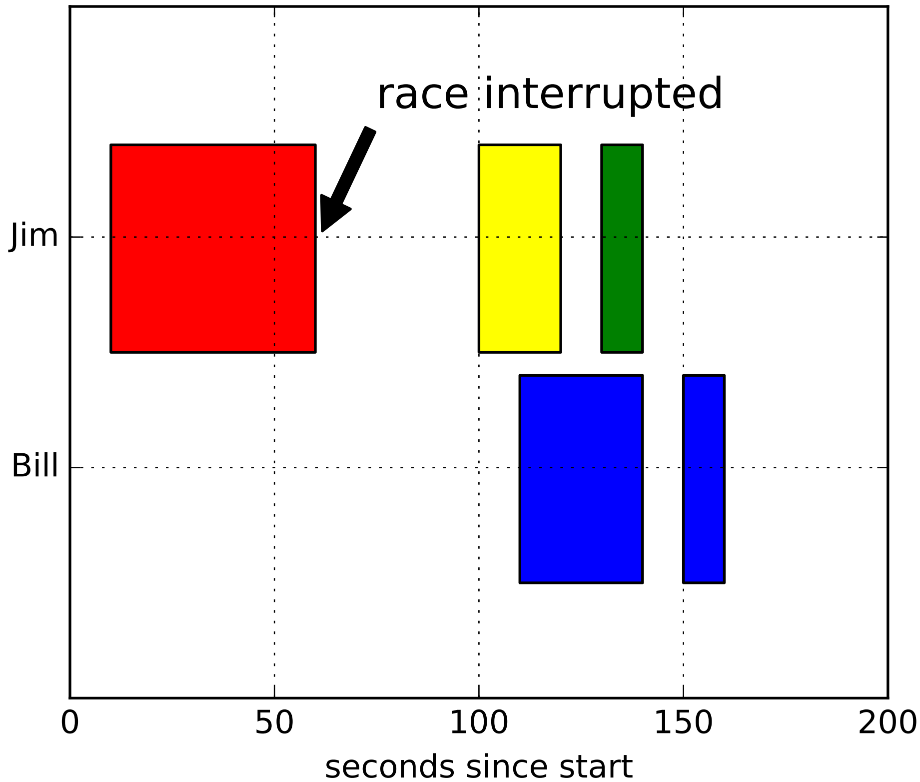



```
"""
Broken axis example, where the y-axis will have a portion cut out.
"""
import matplotlib.pyplot as plt
import numpy as np


# 30 points between 0 0.2] originally made using np.random.rand(30)*.2
pts = np.array([
    0.015, 0.166, 0.133, 0.159, 0.041, 0.024, 0.195, 0.039, 0.161, 0.018,
    0.143, 0.056, 0.125, 0.096, 0.094, 0.051, 0.043, 0.021, 0.138, 0.075,
    0.109, 0.195, 0.050, 0.074, 0.079, 0.155, 0.020, 0.010, 0.061, 0.008])

# Now let's make two outlier points which are far away from everything.
pts[[3, 14]] += .8

# If we were to simply plot pts, we'd lose most of the interesting
# details due to the outliers. So let's 'break' or 'cut-out' the y-axis
# into two portions - use the top (ax) for the outliers, and the bottom
# (ax2) for the details of the majority of our data
f, (ax, ax2) = plt.subplots(2, 1, sharex=True)
```

```
# plot the same data on both axes
ax.plot(pts)
ax2.plot(pts)

# zoom-in / limit the view to different portions of the data
ax.set_ylim(.78, 1.)  # outliers only
ax2.set_ylim(0, .22)  # most of the data

# hide the spines between ax and ax2
ax.spines['bottom'].set_visible(False)
ax2.spines['top'].set_visible(False)
ax.xaxis.tick_top()
ax.tick_params(labeltop='off')  # don't put tick labels at the top
ax2.xaxis.tick_bottom()

# This looks pretty good, and was fairly painless, but you can get that
# cut-out diagonal lines look with just a bit more work. The important
# thing to know here is that in axes coordinates, which are always
# between 0-1, spine endpoints are at these locations (0,0), (0,1),
# (1,0), and (1,1).  Thus, we just need to put the diagonals in the
# appropriate corners of each of our axes, and so long as we use the
# right transform and disable clipping.

d = .015  # how big to make the diagonal lines in axes coordinates
# arguments to pass plot, just so we don't keep repeating them
kwargs = dict(transform=ax.transAxes, color='k', clip_on=False)
ax.plot((-d, +d), (-d, +d), **kwargs)        # top-left diagonal
ax.plot((1 - d, 1 + d), (-d, +d), **kwargs)  # top-right diagonal

kwargs.update(transform=ax2.transAxes)  # switch to the bottom axes
ax2.plot((-d, +d), (1 - d, 1 + d), **kwargs)  # bottom-left diagonal
ax2.plot((1 - d, 1 + d), (1 - d, 1 + d), **kwargs)  # bottom-right diagonal

# What's cool about this is that now if we vary the distance between
# ax and ax2 via f.subplots_adjust(hspace=...) or plt.subplot_tool(),
# the diagonal lines will move accordingly, and stay right at the tips
# of the spines they are 'breaking'

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

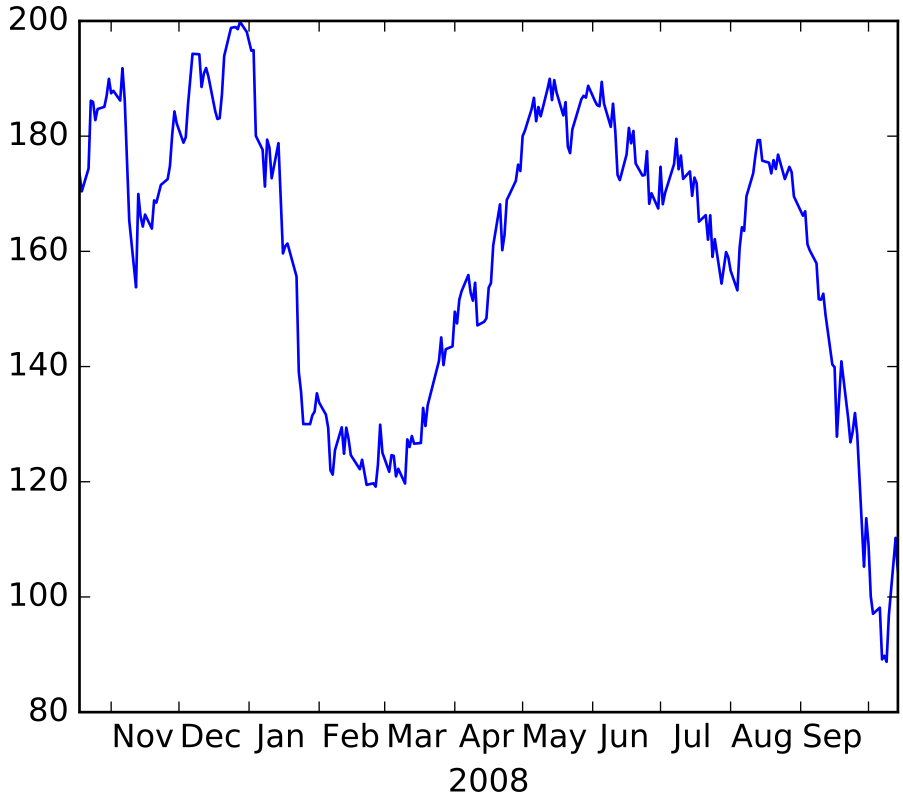
90.30 pylab_examples example code: broken_barh.py




```
"""
Make a "broken" horizontal bar plot, i.e., one with gaps
"""
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.broken_barh([(110, 30), (150, 10)], (10, 9), facecolors='blue')
ax.broken_barh([(10, 50), (100, 20), (130, 10)], (20, 9),
               facecolors=('red', 'yellow', 'green'))
ax.set_ylim(5, 35)
ax.set_xlim(0, 200)
ax.set_xlabel('seconds since start')
ax.set_yticks([15, 25])
ax.set_yticklabels(['Bill', 'Jim'])
ax.grid(True)
ax.annotate('race interrupted', (61, 25),
            xytext=(0.8, 0.9), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize=16,
            horizontalalignment='right', verticalalignment='top')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.31 pylab_examples example code: centered_ticklabels.py

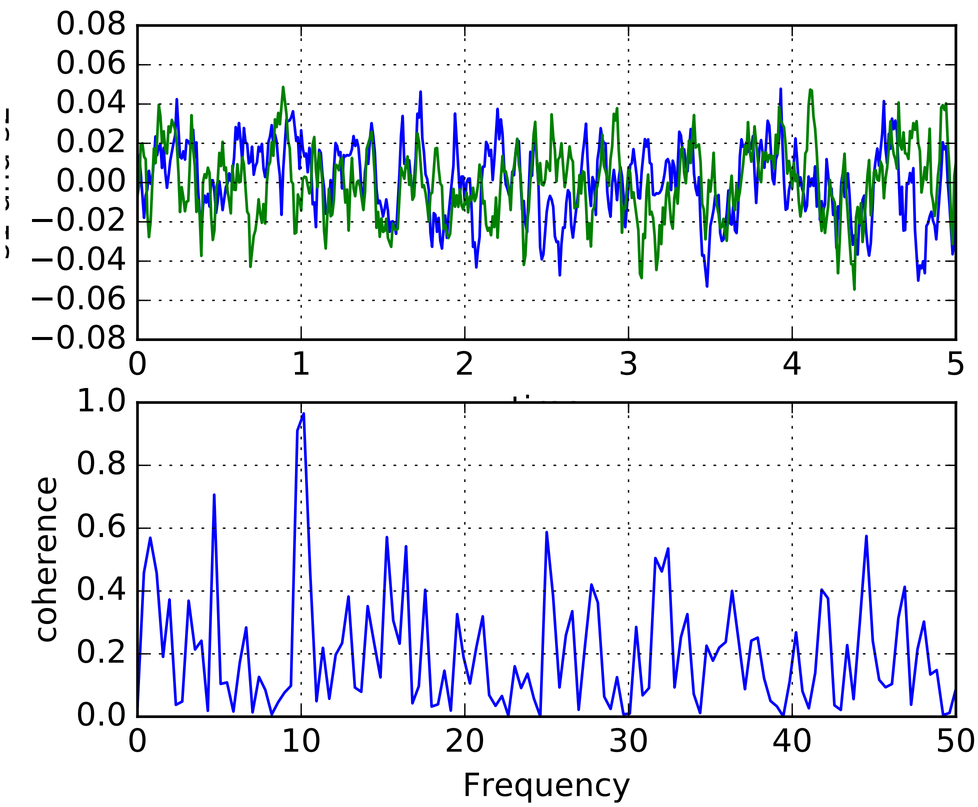



```
# sometimes it is nice to have ticklabels centered.  mpl currently
# associates a label with a tick, and the label can be aligned
# 'center', 'left', or 'right' using the horizontal alignment property:
#
#
#   for label in ax.xaxis.get_xticklabels():
#       label.set_horizontalalignment('right')
#
#
# but this doesn't help center the label between ticks.  One solution
# is to "face it".  Use the minor ticks to place a tick centered
# between the major ticks.  Here is an example that labels the months,
# centered between the ticks

import numpy as np
import matplotlib.cbook as cbook
import matplotlib.dates as dates
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt
```

```
# load some financial data; apple's stock price
fh = cbook.get_sample_data('aapl.npy.gz')
try:
    # Python3 cannot load python2 .npy files with datetime(object) arrays
    # unless the encoding is set to bytes. Hovever this option was
    # not added until numpy 1.10 so this example will only work with
    # python 2 or with numpy 1.10 and later.
    r = np.load(fh, encoding='bytes')
except TypeError:
    r = np.load(fh)
fh.close()
r = r[-250:]  # get the last 250 days

fig, ax = plt.subplots()
ax.plot(r.date, r.adj_close)

ax.xaxis.set_major_locator(dates.MonthLocator())
ax.xaxis.set_minor_locator(dates.MonthLocator(bymonthday=15))

ax.xaxis.set_major_formatter(ticker.NullFormatter())
ax.xaxis.set_minor_formatter(dates.DateFormatter('%b'))

for tick in ax.xaxis.get_minor_ticks():
    tick.tick1line.set_markersize(0)
    tick.tick2line.set_markersize(0)
    tick.label1.set_horizontalalignment('center')

imid = len(r)//2
ax.set_xlabel(str(r.date[imid].year))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.32 pylab_examples example code: cohere_demo.py

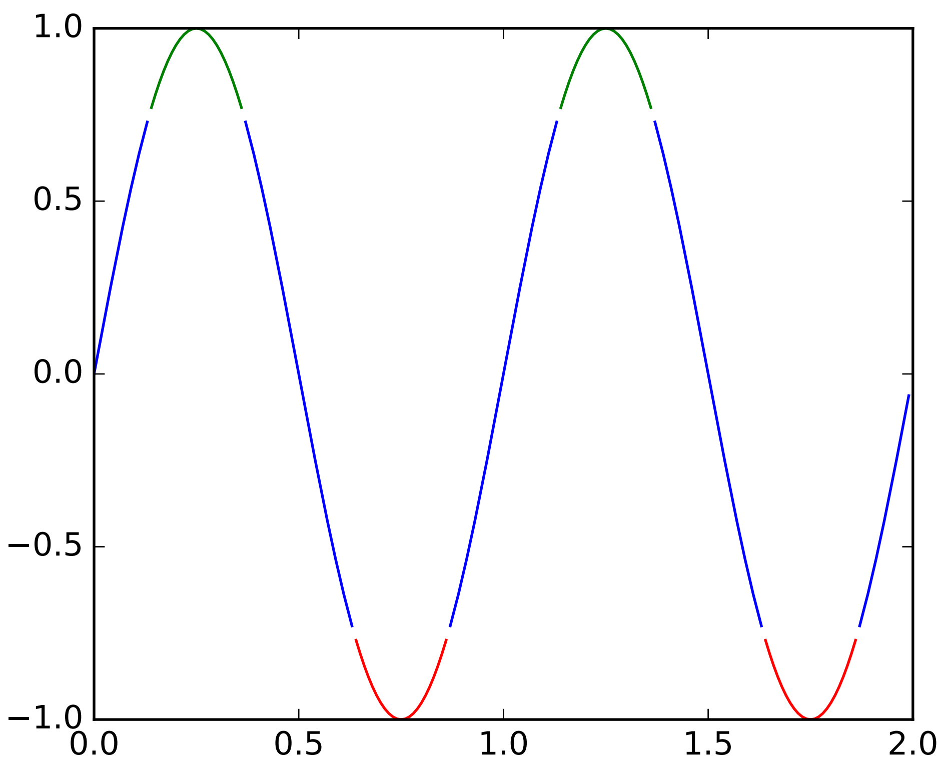



```
#!/usr/bin/env python
"""
Compute the coherence of two signals
"""
import numpy as np
import matplotlib.pyplot as plt

# make a little extra space between the subplots
plt.subplots_adjust(wspace=0.5)

dt = 0.01
t = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t))                 # white noise 1
nse2 = np.random.randn(len(t))                 # white noise 2
r = np.exp(-t/0.05)

cnse1 = np.convolve(nse1, r, mode='same')*dt   # colored noise 1
cnse2 = np.convolve(nse2, r, mode='same')*dt   # colored noise 2

# two signals with a coherent part and a random part
s1 = 0.01*np.sin(2*np.pi*10*t) + cnse1
s2 = 0.01*np.sin(2*np.pi*10*t) + cnse2
```

```
plt.subplot(211)
plt.plot(t, s1, 'b-', t, s2, 'g-')
plt.xlim(0, 5)
plt.xlabel('time')
plt.ylabel('s1 and s2')
plt.grid(True)

plt.subplot(212)
cxy, f = plt.cohere(s1, s2, 256, 1./dt)
plt.ylabel('coherence')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.33 pylab_examples example code: color_by_yvalue.py

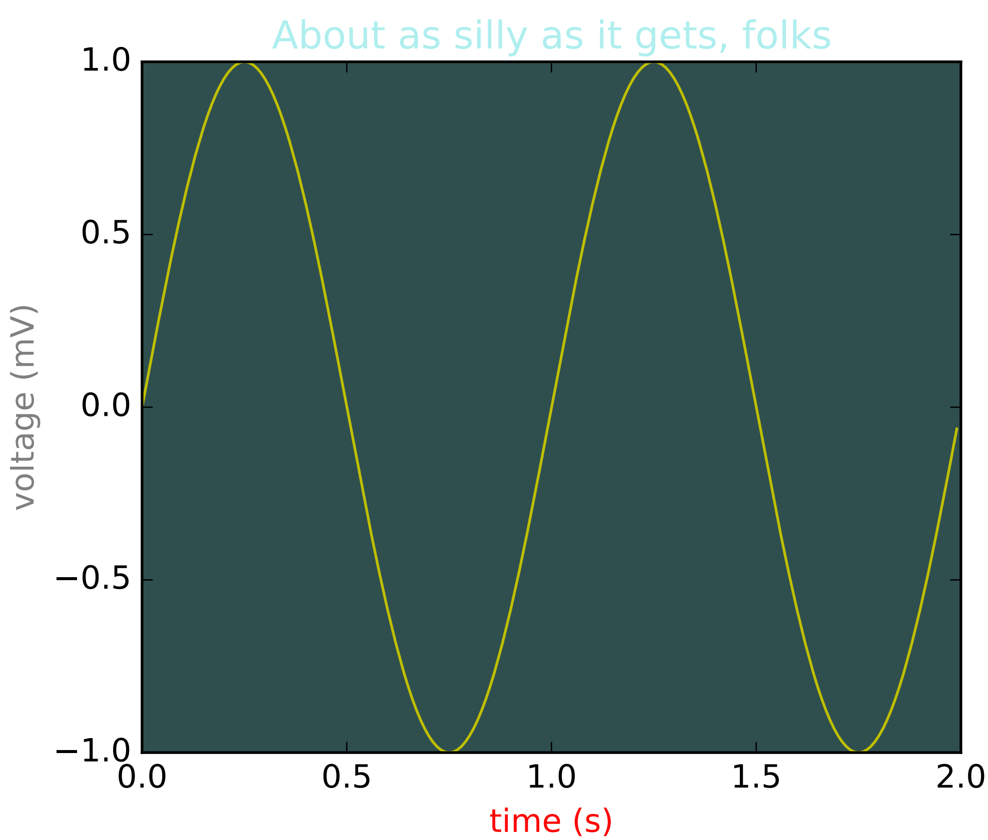



```
# use masked arrays to plot a line with different colors by y-value
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0.0, 2.0, 0.01)
```

```
s = np.sin(2*np.pi*t)

upper = 0.77
lower = -0.77


supper = np.ma.masked_where(s < upper, s)
slower = np.ma.masked_where(s > lower, s)
smiddle = np.ma.masked_where(np.logical_or(s < lower, s > upper), s)

plt.plot(t, slower, 'r', t, smiddle, 'b', t, supper, 'g')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.34 pylab_examples example code: color_demo.py

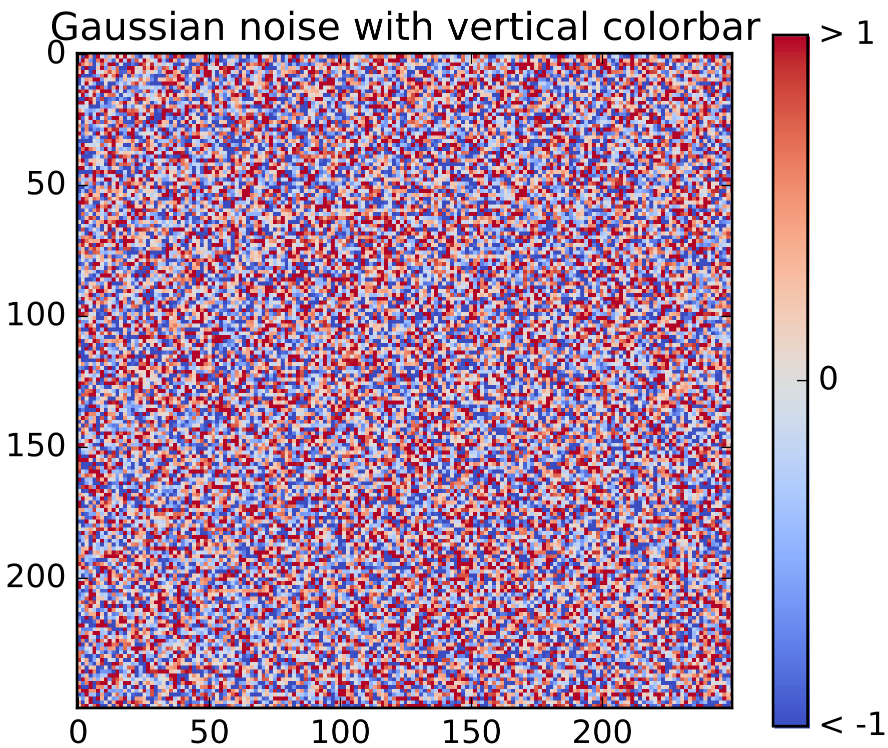


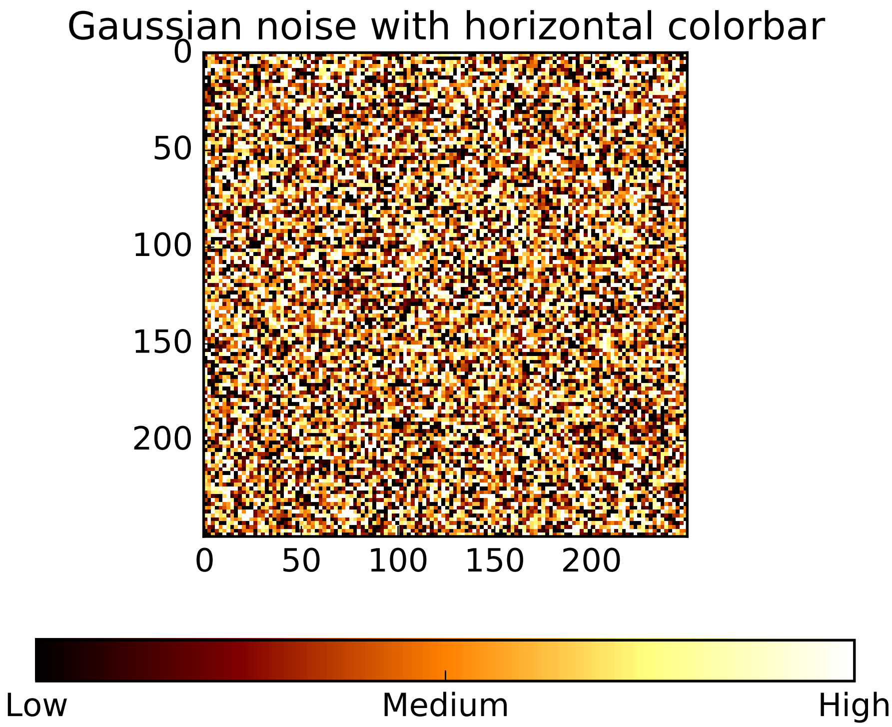

```
#!/usr/bin/env python
"""
matplotlib gives you 4 ways to specify colors,

    1) as a single letter string, ala MATLAB
```

```
    2) as an html style hex string or html color name

    3) as an R,G,B tuple, where R,G,B, range from 0-1

    4) as a string representing a floating point number
       from 0 to 1, corresponding to shades of gray.

See help(colors) for more info.
"""
import matplotlib.pyplot as plt
import numpy as np

plt.subplot(111, axisbg='darkslategray')
#subplot(111, axisbg='#ababab')
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
plt.plot(t, s, 'y')
plt.xlabel('time (s)', color='r')
plt.ylabel('voltage (mV)', color='0.5')  # grayscale color
plt.title('About as silly as it gets, folks', color='#afeeee')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.35 pylab_examples example code: colorbar_tick_labelling_demo.py






```python
"""Produce custom labelling for a colorbar.

Contributed by Scott Sinclair
"""

import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from numpy.random import randn

# Make plot with vertical (default) colorbar
fig, ax = plt.subplots()

data = np.clip(randn(250, 250), -1, 1)

cax = ax.imshow(data, interpolation='nearest', cmap=cm.coolwarm)
ax.set_title('Gaussian noise with vertical colorbar')

# Add colorbar, make sure to specify tick locations to match desired ticklabels
cbar = fig.colorbar(cax, ticks=[-1, 0, 1])
cbar.ax.set_yticklabels(['< -1', '0', '> 1'])  # vertically oriented colorbar

# Make plot with horizontal colorbar
fig, ax = plt.subplots()

data = np.clip(randn(250, 250), -1, 1)
```

```
cax = ax.imshow(data, interpolation='nearest', cmap=cm.afmhot)
ax.set_title('Gaussian noise with horizontal colorbar')

cbar = fig.colorbar(cax, ticks=[-1, 0, 1], orientation='horizontal')
cbar.ax.set_xticklabels(['Low', 'Medium', 'High'])  # horizontal colorbar

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.36 pylab_examples example code: colours.py

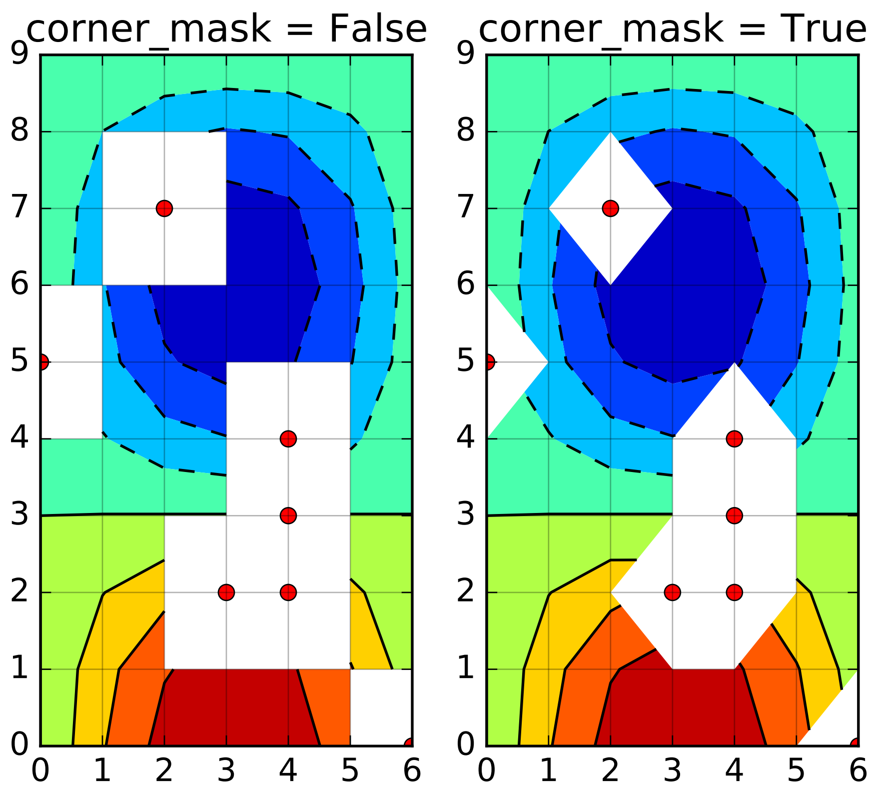
[source code]

```
#!/usr/bin/env python
# -*- noplot -*-
"""
Some simple functions to generate colours.
"""
import numpy as np
from matplotlib.colors import colorConverter


def pastel(colour, weight=2.4):
    """ Convert colour into a nice pastel shade"""
    rgb = np.asarray(colorConverter.to_rgb(colour))
    # scale colour
    maxc = max(rgb)
    if maxc < 1.0 and maxc > 0:
        # scale colour
        scale = 1.0 / maxc
        rgb = rgb * scale
    # now decrease saturation
    total = rgb.sum()
    slack = 0
    for x in rgb:
        slack += 1.0 - x

    # want to increase weight from total to weight
    # pick x s.t.  slack * x == weight - total
    # x = (weight - total) / slack
    x = (weight - total) / slack

    rgb = [c + (x * (1.0 - c)) for c in rgb]

    return rgb


def get_colours(n):
    """ Return n pastel colours. """
```

```
    base = np.asarray([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

    if n <= 3:
        return base[0:n]

    # how many new colours to we need to insert between
    # red and green and between green and blue?
    needed = (((n - 3) + 1) / 2, (n - 3) / 2)

    colours = []
    for start in (0, 1):
        for x in np.linspace(0, 1, needed[start] + 2):
            colours.append((base[start] * (1.0 - x)) +
                           (base[start + 1] * x))

    return [pastel(c) for c in colours[0:n]]
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

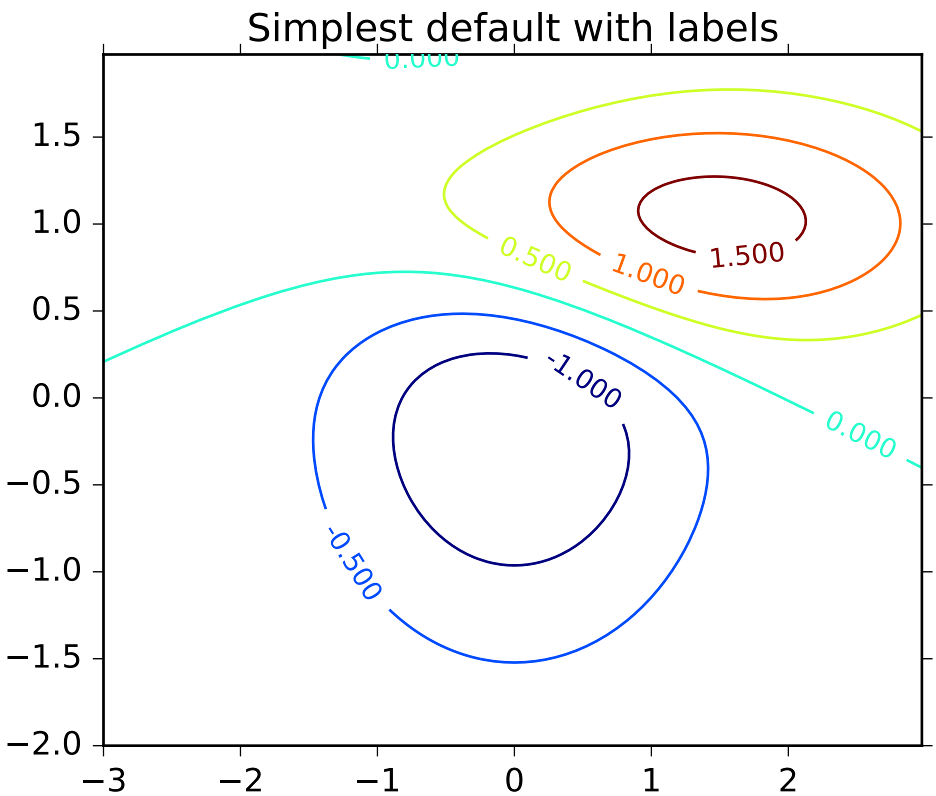
90.37 pylab_examples example code: contour_corner_mask.py

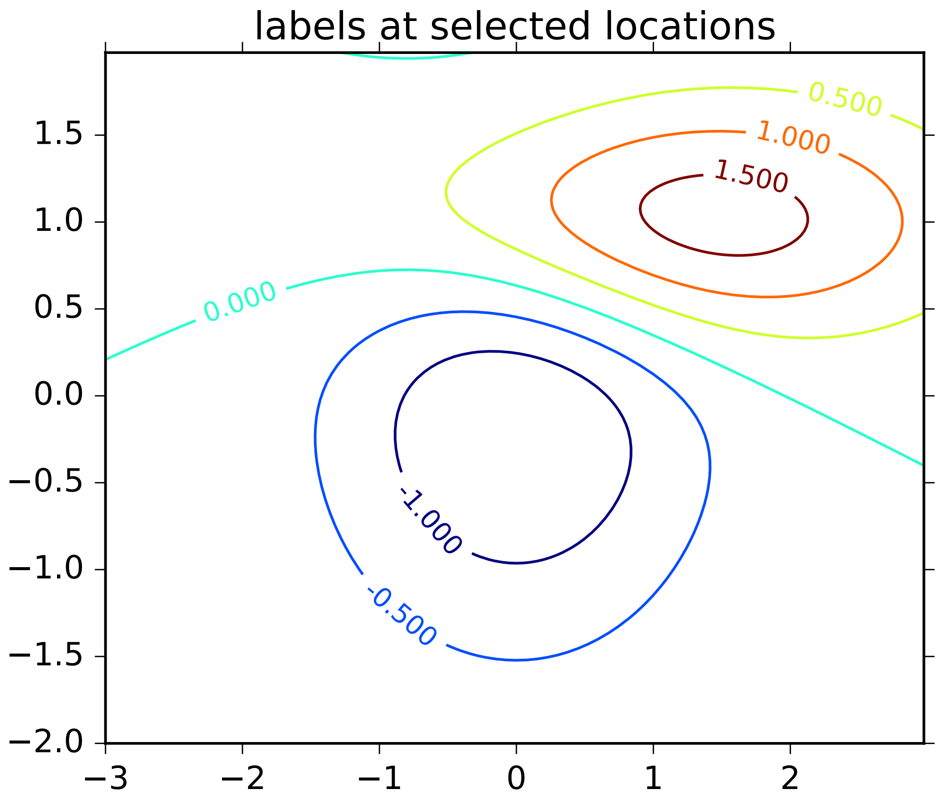


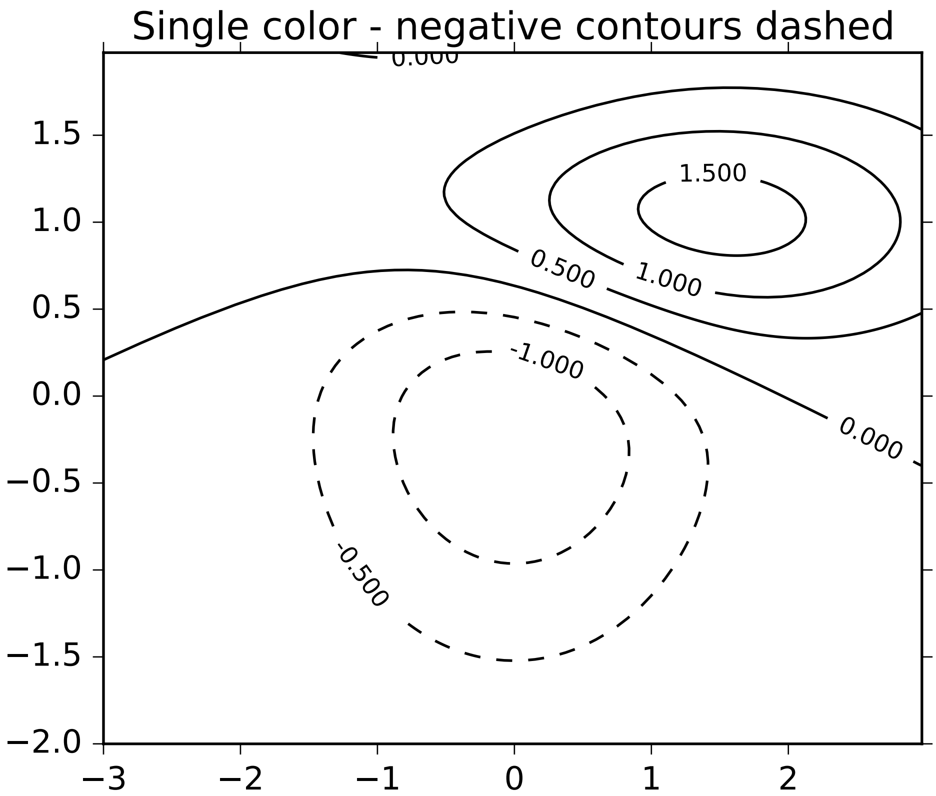
```
#!/usr/bin/env python
"""
Illustrate the difference between corner_mask=False and corner_mask=True
for masked contour plots.
"""
import matplotlib.pyplot as plt
import numpy as np

# Data to plot.
x, y = np.meshgrid(np.arange(7), np.arange(10))
z = np.sin(0.5*x)*np.cos(0.52*y)

# Mask various z values.
mask = np.zeros_like(z, dtype=np.bool)
mask[2, 3:5] = True
mask[3:5, 4] = True
mask[7, 2] = True
mask[5, 0] = True
mask[0, 6] = True
z = np.ma.array(z, mask=mask)

corner_masks = [False, True]
for i, corner_mask in enumerate(corner_masks):
    plt.subplot(1, 2, i+1)
    cs = plt.contourf(x, y, z, corner_mask=corner_mask)
    plt.contour(cs, colors='k')
    plt.title('corner_mask = {0}'.format(corner_mask))

    # Plot grid.
    plt.grid(c='k', ls='-', alpha=0.3)

    # Indicate masked points with red circles.
    plt.plot(np.ma.array(x, mask=~mask), y, 'ro')

plt.show()
```

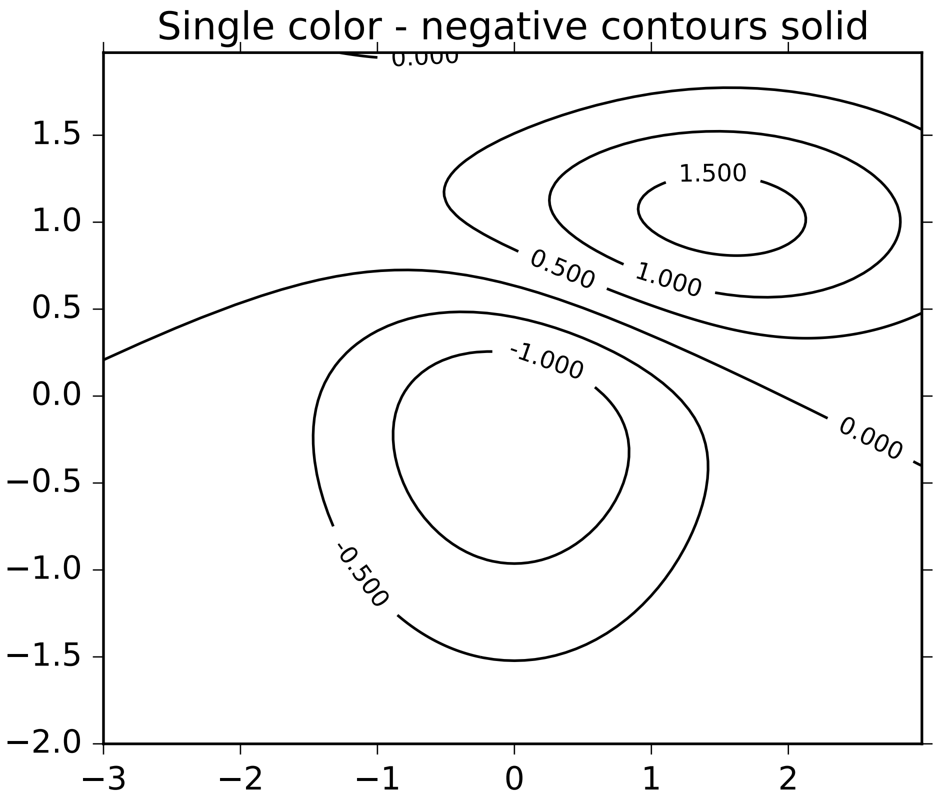
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

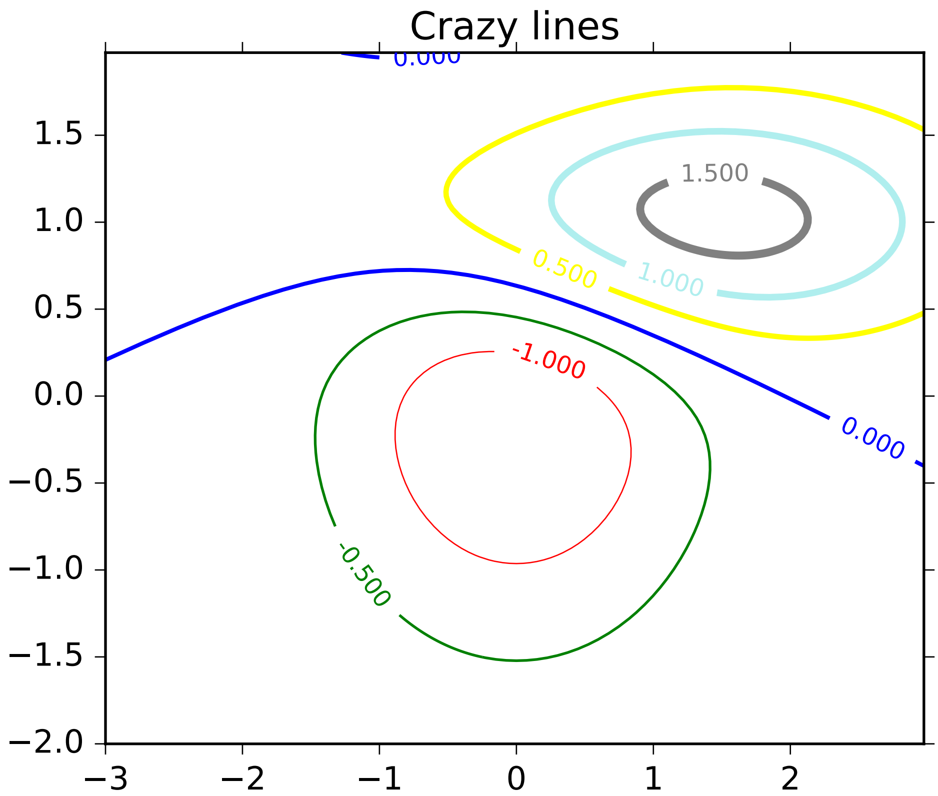
90.38 pylab_examples example code: contour_demo.py

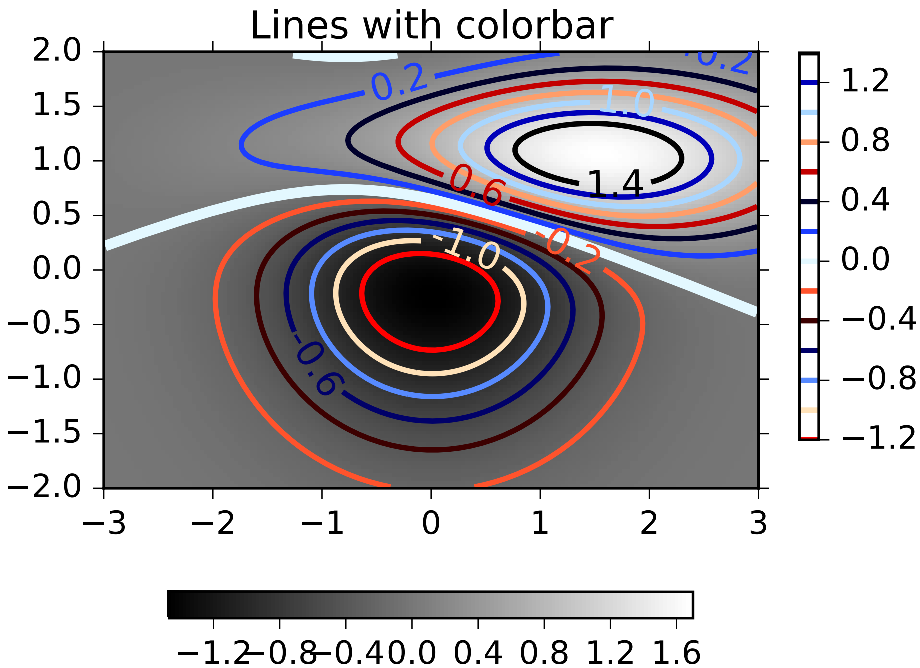



labels at selected locations
0.500
1.000
1.500
0.000
-1.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Single color - negative contours dashed
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Single color - negative contours solid
0.000
1.500
0.500
1.000
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2


Crazy lines
0.000
1.500
1.000
0.500
-1.000
0.000
-0.500
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2



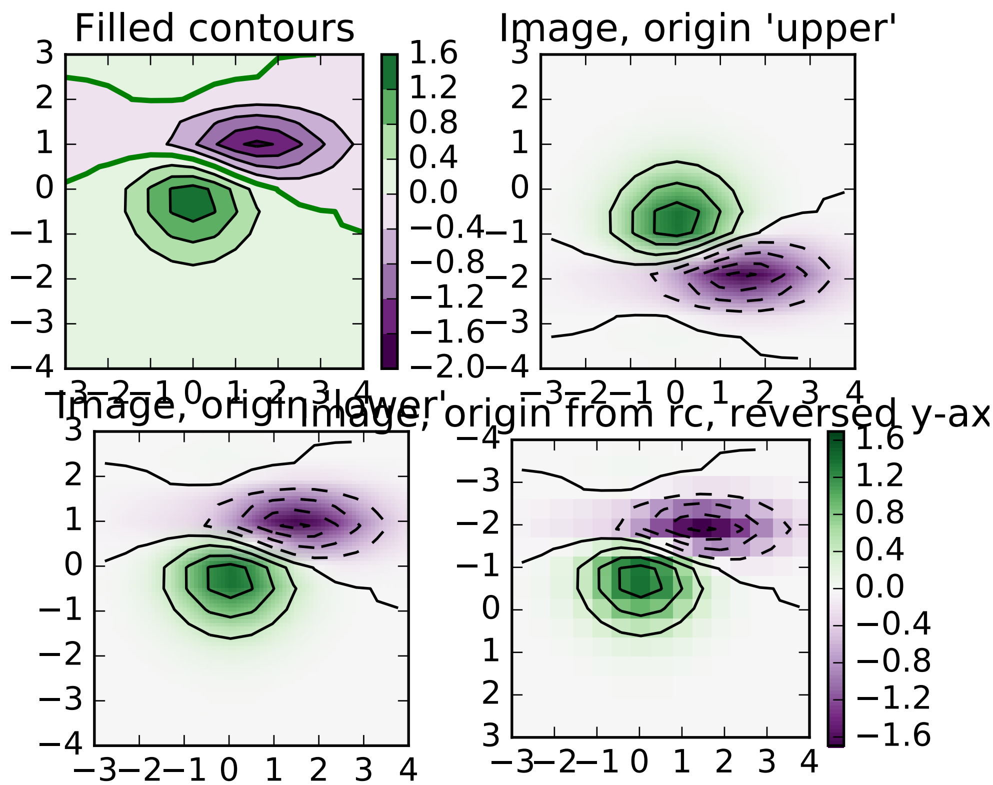

```
#!/usr/bin/env python
"""
Illustrate simple contour plotting, contours on an image with
a colorbar for the contours, and labelled contours.

See also contour_image.py.
"""
import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'

delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
# difference of Gaussians
Z = 10.0 * (Z2 - Z1)
```

```
# Create a simple contour plot with labels using default colors.  The
# inline argument to clabel will control whether the labels are draw
# over the line segments of the contour, removing the lines beneath
# the label
plt.figure()
CS = plt.contour(X, Y, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('Simplest default with labels')


# contour labels can be placed manually by providing list of positions
# (in data coordinate). See ginput_manual_clabel.py for interactive
# placement.
plt.figure()
CS = plt.contour(X, Y, Z)
manual_locations = [(-1, -1.4), (-0.62, -0.7), (-2, 0.5), (1.7, 1.2), (2.0, 1.4), (2.4,␣
↪1.7)]
plt.clabel(CS, inline=1, fontsize=10, manual=manual_locations)
plt.title('labels at selected locations')


# You can force all the contours to be the same color.
plt.figure()
CS = plt.contour(X, Y, Z, 6,
                 colors='k',  # negative contours will be dashed by default
                 )
plt.clabel(CS, fontsize=9, inline=1)
plt.title('Single color - negative contours dashed')

# You can set negative contours to be solid instead of dashed:
matplotlib.rcParams['contour.negative_linestyle'] = 'solid'
plt.figure()
CS = plt.contour(X, Y, Z, 6,
                 colors='k',  # negative contours will be dashed by default
                 )
plt.clabel(CS, fontsize=9, inline=1)
plt.title('Single color - negative contours solid')


# And you can manually specify the colors of the contour
plt.figure()
CS = plt.contour(X, Y, Z, 6,
                 linewidths=np.arange(.5, 4, .5),
                 colors=('r', 'green', 'blue', (1, 1, 0), '#afeeee', '0.5')
                 )
plt.clabel(CS, fontsize=9, inline=1)
plt.title('Crazy lines')


# Or you can use a colormap to specify the colors; the default
# colormap will be used for the contour lines
plt.figure()
im = plt.imshow(Z, interpolation='bilinear', origin='lower',
```

```
                cmap=cm.gray, extent=(-3, 3, -2, 2))
levels = np.arange(-1.2, 1.6, 0.2)
CS = plt.contour(Z, levels,
                 origin='lower',
                 linewidths=2,
                 extent=(-3, 3, -2, 2))

# Thicken the zero contour.
zc = CS.collections[6]
plt.setp(zc, linewidth=4)

plt.clabel(CS, levels[1::2],  # label every second level
           inline=1,
           fmt='%1.1f',
           fontsize=14)

# make a colorbar for the contour lines
CB = plt.colorbar(CS, shrink=0.8, extend='both')

plt.title('Lines with colorbar')
#plt.hot()  # Now change the colormap for the contour lines and colorbar
plt.flag()

# We can still add a colorbar for the image, too.
CBI = plt.colorbar(im, orientation='horizontal', shrink=0.8)

# This makes the original colorbar look a bit out of place,
# so let's improve its position.

l, b, w, h = plt.gca().get_position().bounds
ll, bb, ww, hh = CB.ax.get_position().bounds
CB.ax.set_position([ll, b + 0.1*h, ww, h*0.8])


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.39 pylab_examples example code: contour_image.py



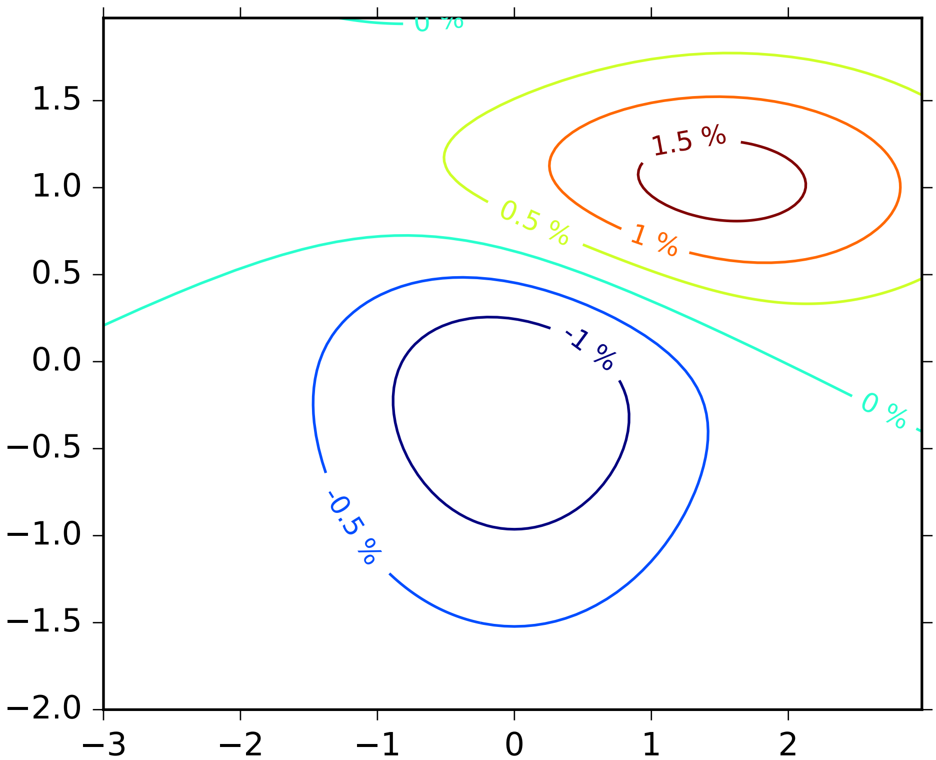

```
#!/usr/bin/env python
'''
Test combinations of contouring, filled contouring, and image plotting.
For contour labelling, see contour_demo.py.

The emphasis in this demo is on showing how to make contours register
correctly on images, and on how to get both of them oriented as
desired.  In particular, note the usage of the "origin" and "extent"
keyword arguments to imshow and contour.
'''
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import mlab, cm

# Default delta is large because that makes it fast, and it illustrates
# the correct registration between image and contours.
delta = 0.5

extent = (-3, 4, -4, 3)

x = np.arange(-3.0, 4.001, delta)
y = np.arange(-4.0, 3.001, delta)
```

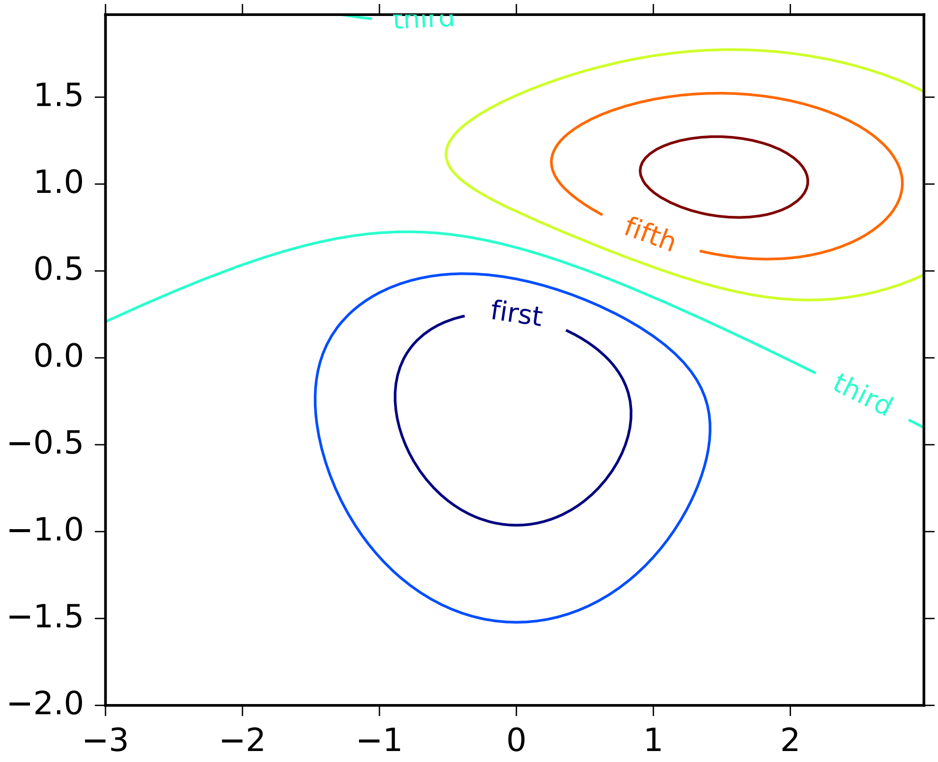
```
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = (Z1 - Z2) * 10

levels = np.arange(-2.0, 1.601, 0.4)  # Boost the upper limit to avoid truncation errors.

norm = cm.colors.Normalize(vmax=abs(Z).max(), vmin=-abs(Z).max())
cmap = cm.PRGn

plt.figure()


plt.subplot(2, 2, 1)

cset1 = plt.contourf(X, Y, Z, levels,
                 cmap=cm.get_cmap(cmap, len(levels) - 1),
                 norm=norm,
                 )
# It is not necessary, but for the colormap, we need only the
# number of levels minus 1.  To avoid discretization error, use
# either this number or a large number such as the default (256).

# If we want lines as well as filled regions, we need to call
# contour separately; don't try to change the edgecolor or edgewidth
# of the polygons in the collections returned by contourf.
# Use levels output from previous call to guarantee they are the same.
cset2 = plt.contour(X, Y, Z, cset1.levels,
                colors='k',
                hold='on')
# We don't really need dashed contour lines to indicate negative
# regions, so let's turn them off.
for c in cset2.collections:
    c.set_linestyle('solid')

# It is easier here to make a separate call to contour than
# to set up an array of colors and linewidths.
# We are making a thick green line as a zero contour.
# Specify the zero level as a tuple with only 0 in it.
cset3 = plt.contour(X, Y, Z, (0,),
                colors='g',
                linewidths=2,
                hold='on')
plt.title('Filled contours')
plt.colorbar(cset1)
#hot()


plt.subplot(2, 2, 2)

plt.imshow(Z, extent=extent, cmap=cmap, norm=norm)
v = plt.axis()
plt.contour(Z, levels, hold='on', colors='k',
```

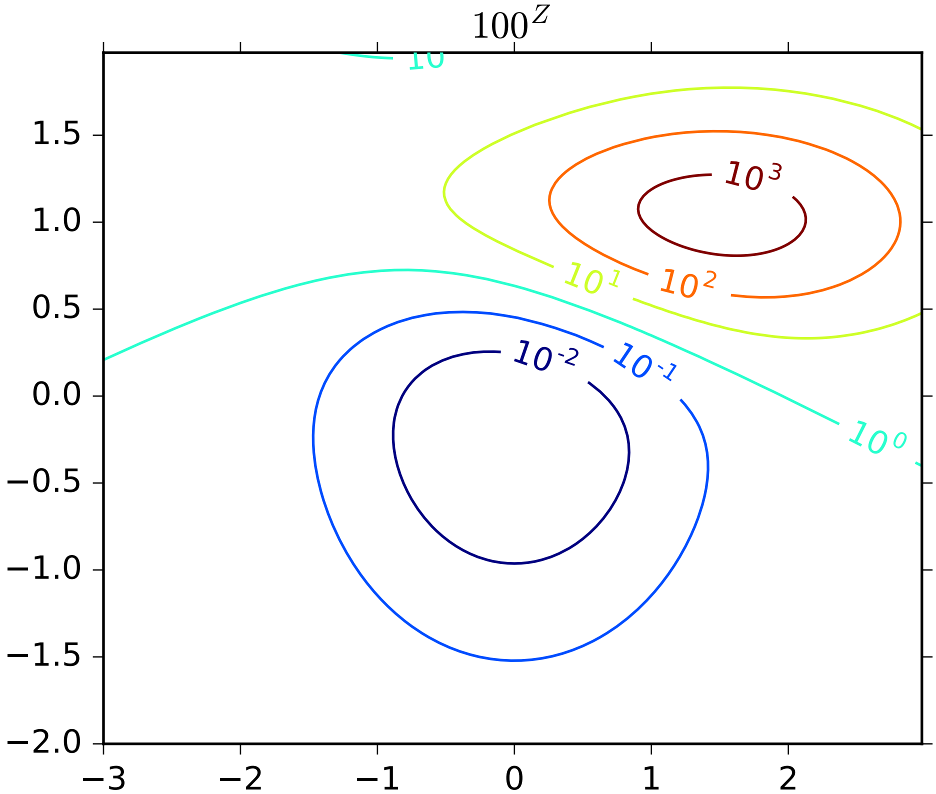
```
        origin='upper', extent=extent)
plt.axis(v)
plt.title("Image, origin 'upper'")

plt.subplot(2, 2, 3)

plt.imshow(Z, origin='lower', extent=extent, cmap=cmap, norm=norm)
v = plt.axis()
plt.contour(Z, levels, hold='on', colors='k',
        origin='lower', extent=extent)
plt.axis(v)
plt.title("Image, origin 'lower'")

plt.subplot(2, 2, 4)

# We will use the interpolation "nearest" here to show the actual
# image pixels.
# Note that the contour lines don't extend to the edge of the box.
# This is intentional. The Z values are defined at the center of each
# image pixel (each color block on the following subplot), so the
# domain that is contoured does not extend beyond these pixel centers.
im = plt.imshow(Z, interpolation='nearest', extent=extent, cmap=cmap, norm=norm)
v = plt.axis()
plt.contour(Z, levels, hold='on', colors='k',
        origin='image', extent=extent)
plt.axis(v)
ylim = plt.get(plt.gca(), 'ylim')
plt.setp(plt.gca(), ylim=ylim[::-1])
plt.title("Image, origin from rc, reversed y-axis")
plt.colorbar(im)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.40 pylab_examples example code: contour_label_demo.py




third
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
fifth
first
third
−3
−2
−1
0
1
2



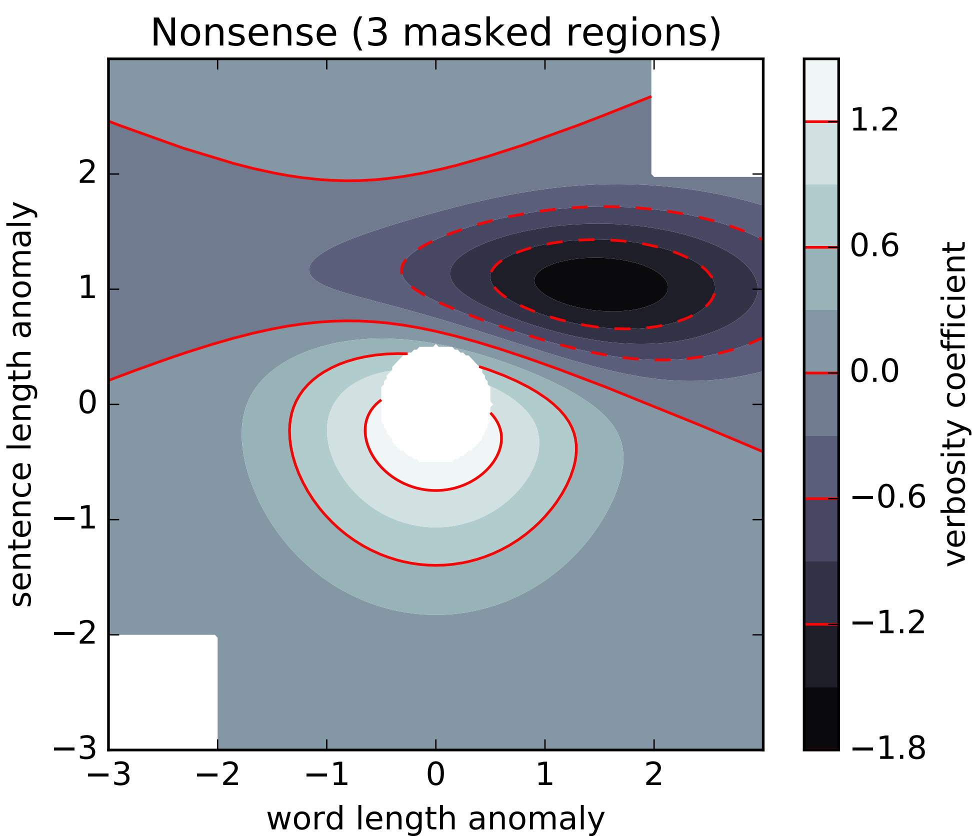

```
#!/usr/bin/env python
"""
Illustrate some of the more advanced things that one can do with
contour labels.

See also contour_demo.py.
"""
import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt

matplotlib.rcParams['xtick.direction'] = 'out'
matplotlib.rcParams['ytick.direction'] = 'out'

##################################################
# Define our surface
##################################################
delta = 0.025
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
```

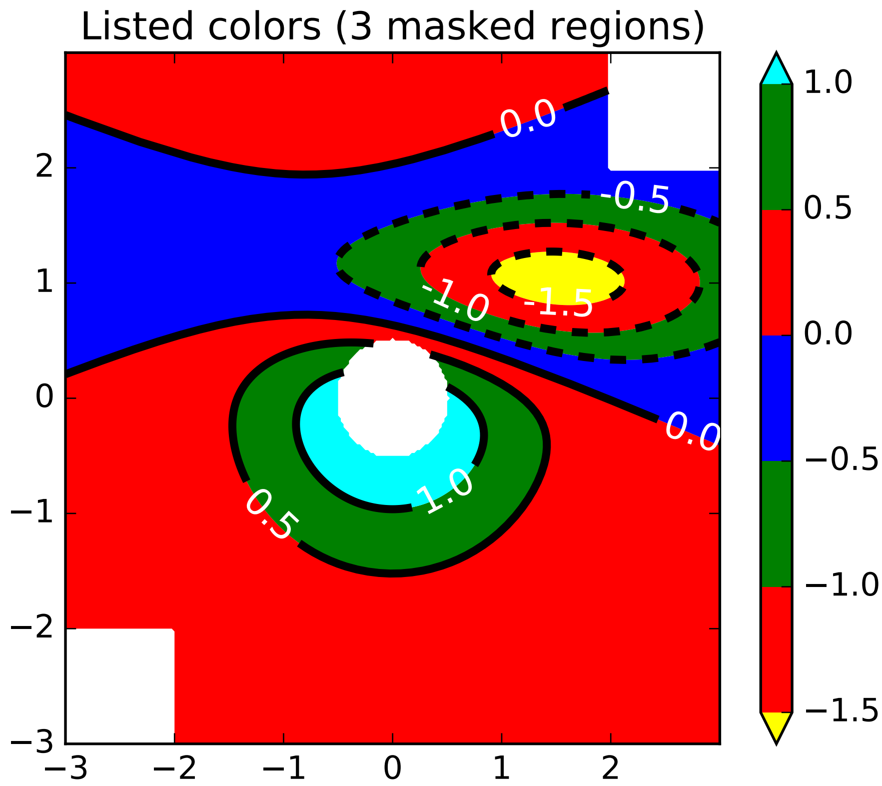
```
# difference of Gaussians
Z = 10.0 * (Z2 - Z1)

##################################################
# Make contour labels using creative float classes
# Follows suggestion of Manuel Metz
##################################################
plt.figure()

# Basic contour plot
CS = plt.contour(X, Y, Z)


# Define a class that forces representation of float to look a certain way
# This remove trailing zero so '1.0' becomes '1'
class nf(float):
    def __repr__(self):
        str = '%.1f' % (self.__float__(),)
        if str[-1] == '0':
            return '%.0f' % self.__float__()
        else:
            return '%.1f' % self.__float__()

# Recast levels to new class
CS.levels = [nf(val) for val in CS.levels]

# Label levels with specially formatted floats
if plt.rcParams["text.usetex"]:
    fmt = r'%r \%%'
else:
    fmt = '%r %%'
plt.clabel(CS, CS.levels, inline=True, fmt=fmt, fontsize=10)

##################################################
# Label contours with arbitrary strings using a
# dictionary
##################################################
plt.figure()

# Basic contour plot
CS = plt.contour(X, Y, Z)

fmt = {}
strs = ['first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh']
for l, s in zip(CS.levels, strs):
    fmt[l] = s

# Label every other level using strings
plt.clabel(CS, CS.levels[::2], inline=True, fmt=fmt, fontsize=10)

# Use a Formatter

plt.figure()
```

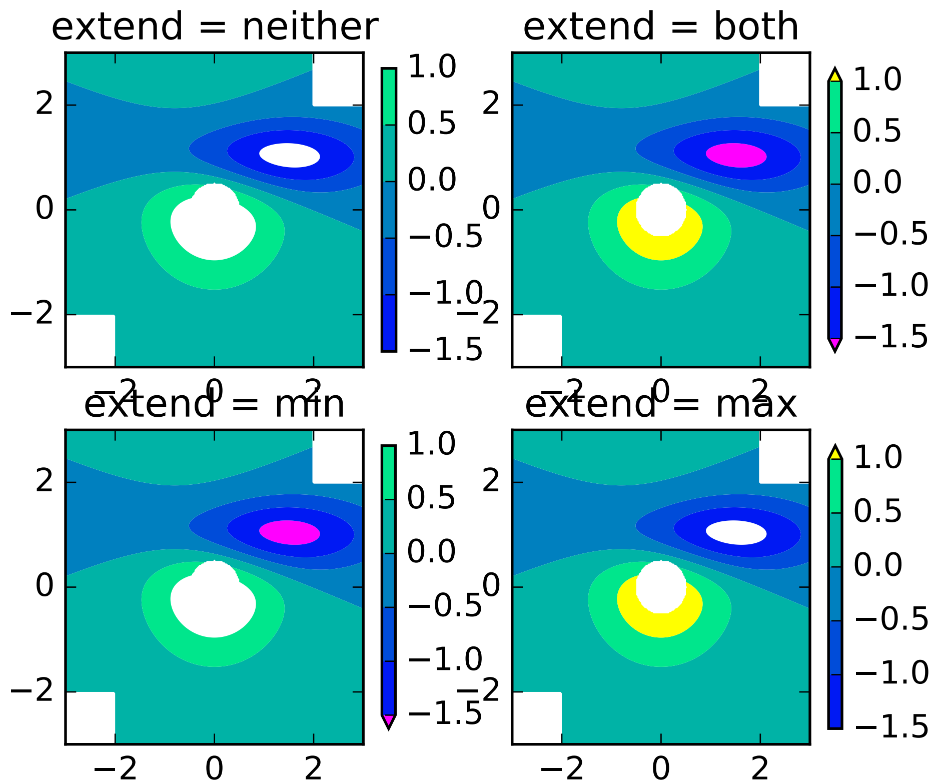
```
CS = plt.contour(X, Y, 100**Z, locator=plt.LogLocator())
fmt = ticker.LogFormatterMathtext()
fmt.create_dummy_axis()
plt.clabel(CS, CS.levels, fmt=fmt)
plt.title("$100^Z$")

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.41 pylab_examples example code: contourf_demo.py




Listed colors (3 masked regions)
0.0
-0.5
-1.0
-1.5
0.0
0.5
1.0
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
1.0
0.5
0.0
−0.5
−1.0
−1.5



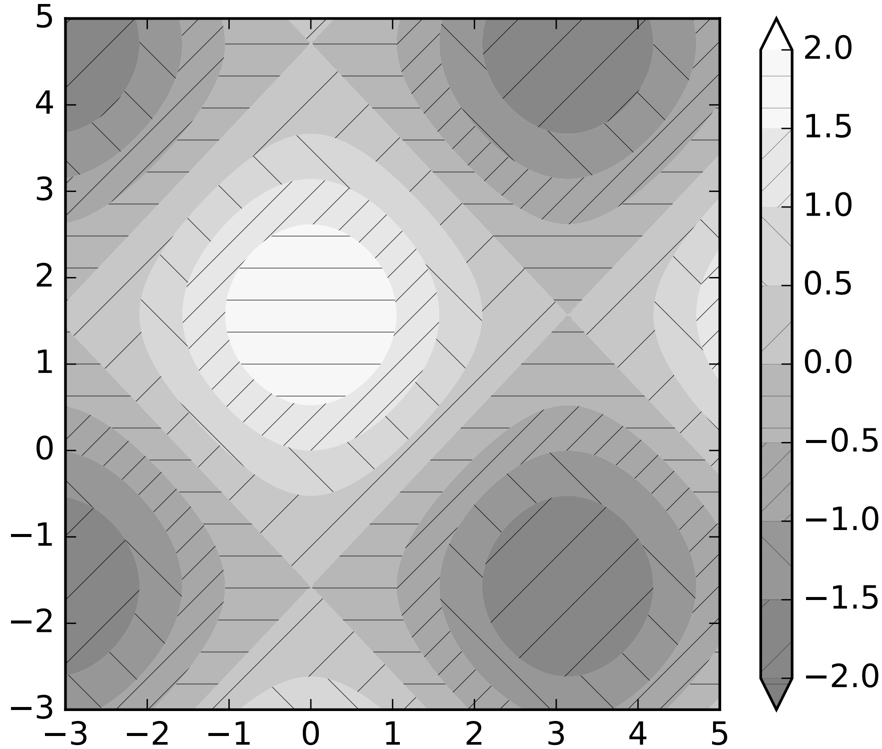

```
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt

origin = 'lower'
#origin = 'upper'

delta = 0.025

x = y = np.arange(-3.0, 3.01, delta)
X, Y = np.meshgrid(x, y)
Z1 = plt.mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = plt.mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = 10 * (Z1 - Z2)

nr, nc = Z.shape

# put NaNs in one corner:
Z[-nr//6:, -nc//6:] = np.nan
# contourf will convert these to masked


Z = np.ma.array(Z)
# mask another corner:
Z[:nr//6, :nc//6] = np.ma.masked
```

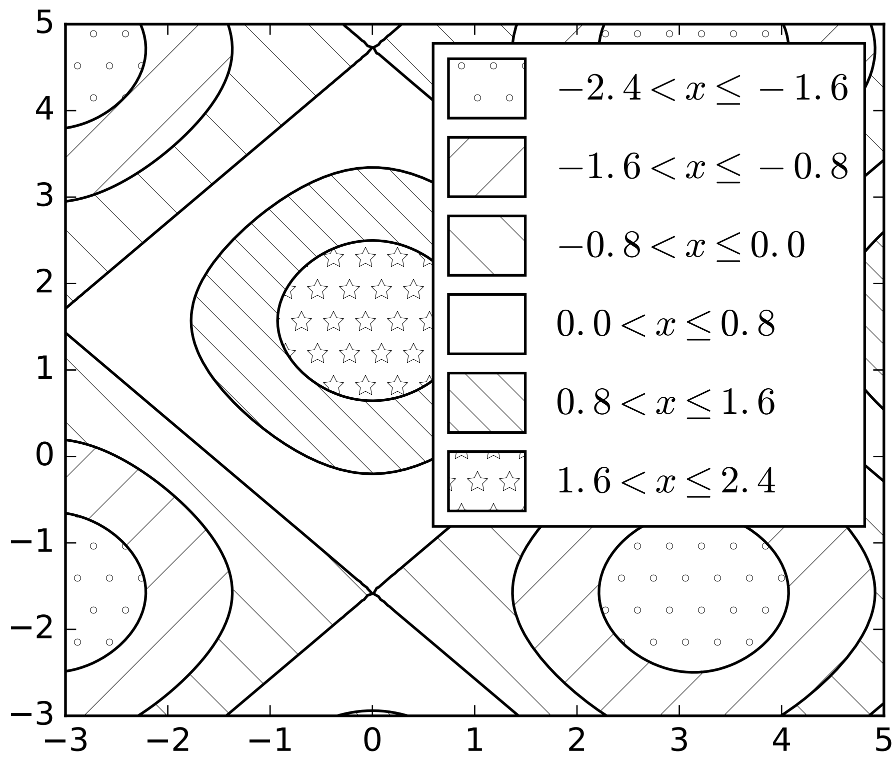
```
# mask a circle in the middle:
interior = np.sqrt((X**2) + (Y**2)) < 0.5
Z[interior] = np.ma.masked


# We are using automatic selection of contour levels;
# this is usually not such a good idea, because they don't
# occur on nice boundaries, but we do it here for purposes
# of illustration.
CS = plt.contourf(X, Y, Z, 10,
                  #[-1, -0.1, 0, 0.1],
                  #alpha=0.5,
                  cmap=plt.cm.bone,
                  origin=origin)

# Note that in the following, we explicitly pass in a subset of
# the contour levels used for the filled contours.  Alternatively,
# We could pass in additional levels to provide extra resolution,
# or leave out the levels kwarg to use all of the original levels.

CS2 = plt.contour(CS, levels=CS.levels[::2],
                  colors='r',
                  origin=origin,
                  hold='on')

plt.title('Nonsense (3 masked regions)')
plt.xlabel('word length anomaly')
plt.ylabel('sentence length anomaly')

# Make a colorbar for the ContourSet returned by the contourf call.
cbar = plt.colorbar(CS)
cbar.ax.set_ylabel('verbosity coefficient')
# Add the contour line levels to the colorbar
cbar.add_lines(CS2)

plt.figure()

# Now make a contour plot with the levels specified,
# and with the colormap generated automatically from a list
# of colors.
levels = [-1.5, -1, -0.5, 0, 0.5, 1]
CS3 = plt.contourf(X, Y, Z, levels,
                   colors=('r', 'g', 'b'),
                   origin=origin,
                   extend='both')
# Our data range extends outside the range of levels; make
# data below the lowest contour level yellow, and above the
# highest level cyan:
CS3.cmap.set_under('yellow')
CS3.cmap.set_over('cyan')

CS4 = plt.contour(X, Y, Z, levels,
                  colors=('k',),
```

```
                  linewidths=(3,),
                  origin=origin)
plt.title('Listed colors (3 masked regions)')
plt.clabel(CS4, fmt='%2.1f', colors='w', fontsize=14)

# Notice that the colorbar command gets all the information it
# needs from the ContourSet object, CS3.
plt.colorbar(CS3)

# Illustrate all 4 possible "extend" settings:
extends = ["neither", "both", "min", "max"]
cmap = plt.cm.get_cmap("winter")
cmap.set_under("magenta")
cmap.set_over("yellow")
# Note: contouring simply excludes masked or nan regions, so
# instead of using the "bad" colormap value for them, it draws
# nothing at all in them.  Therefore the following would have
# no effect:
# cmap.set_bad("red")

fig, axs = plt.subplots(2, 2)
for ax, extend in zip(axs.ravel(), extends):
    cs = ax.contourf(X, Y, Z, levels, cmap=cmap, extend=extend, origin=origin)
    fig.colorbar(cs, ax=ax, shrink=0.9)
    ax.set_title("extend = %s" % extend)
    ax.locator_params(nbins=4)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.42 pylab_examples example code: contourf_hatching.py



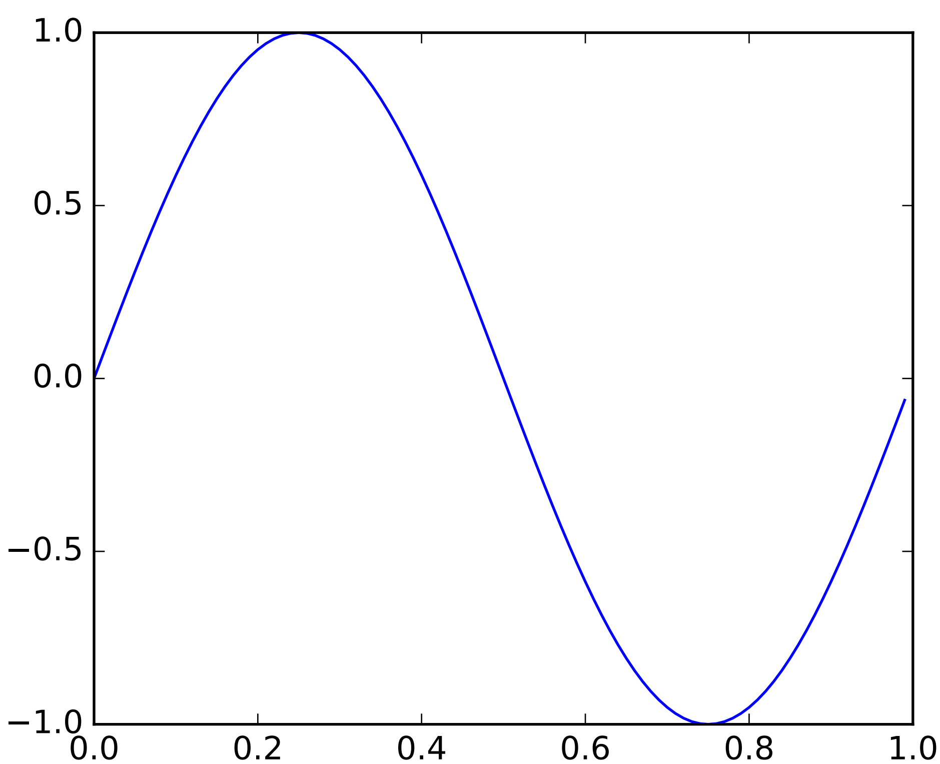



```
import matplotlib.pyplot as plt
import numpy as np


# invent some numbers, turning the x and y arrays into simple
# 2d arrays, which make combining them together easier.
x = np.linspace(-3, 5, 150).reshape(1, -1)
y = np.linspace(-3, 5, 120).reshape(-1, 1)
z = np.cos(x) + np.sin(y)

# we no longer need x and y to be 2 dimensional, so flatten them.
x, y = x.flatten(), y.flatten()


# ---------------------------------------------
# |                 Plot #1                   |
# ---------------------------------------------
# the simplest hatched plot with a colorbar
fig = plt.figure()
cs = plt.contourf(x, y, z, hatches=['-', '/', '\\', '//'],
                  cmap=plt.get_cmap('gray'),
                  extend='both', alpha=0.5
                  )
plt.colorbar()
```

```
# ---------------------------------------------
# |                 Plot #2                   |
# ---------------------------------------------
# a plot of hatches without color with a legend
plt.figure()
n_levels = 6
plt.contour(x, y, z, n_levels, colors='black', linestyles='-')
cs = plt.contourf(x, y, z, n_levels, colors='none',
                  hatches=['.', '/', '\\', None, '\\\\', '*'],
                  extend='lower'
                  )

# create a legend for the contour set
artists, labels = cs.legend_elements()
plt.legend(artists, labels, handleheight=2)


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

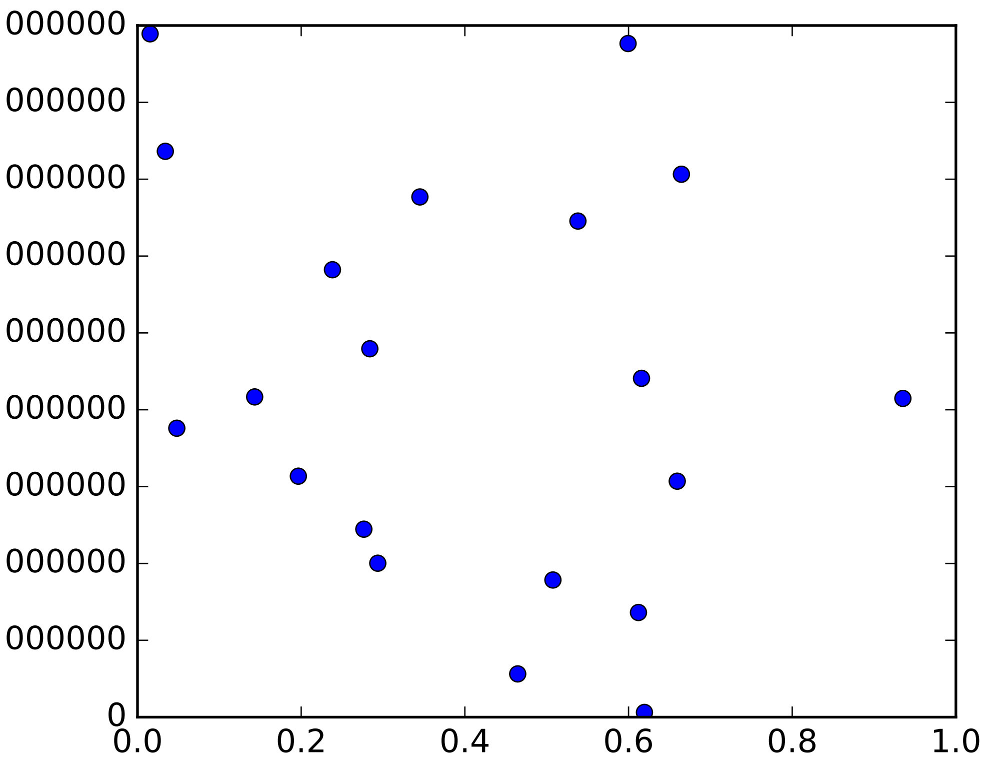
90.43 pylab_examples example code: coords_demo.py



```python
#!/usr/bin/env python

"""
An example of how to interact with the plotting canvas by connecting
to move and click events
"""
from __future__ import print_function
import sys
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
fig, ax = plt.subplots()
ax.plot(t, s)


def on_move(event):
    # get the x and y pixel coords
    x, y = event.x, event.y

    if event.inaxes:
        ax = event.inaxes  # the axes instance
        print('data coords %f %f' % (event.xdata, event.ydata))


def on_click(event):
    # get the x and y coords, flip y from top to bottom
    x, y = event.x, event.y
    if event.button == 1:
        if event.inaxes is not None:
            print('data coords %f %f' % (event.xdata, event.ydata))

binding_id = plt.connect('motion_notify_event', on_move)
plt.connect('button_press_event', on_click)

if "test_disconnect" in sys.argv:
    print("disconnecting console coordinate printout...")
    plt.disconnect(binding_id)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

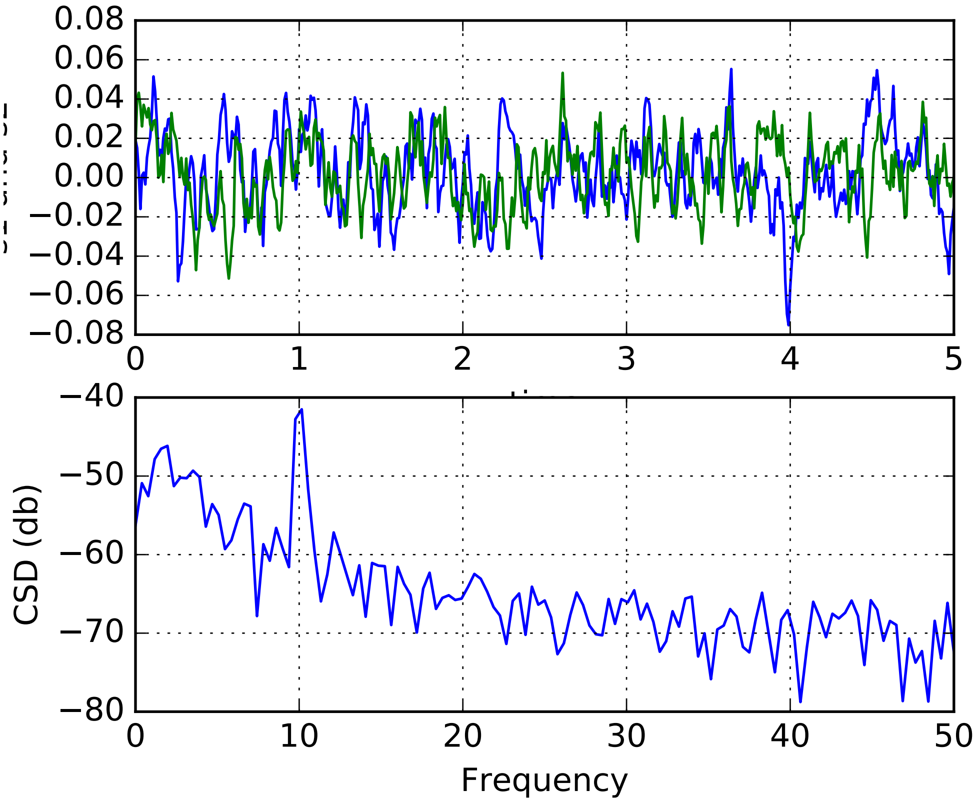
90.44 pylab_examples example code: coords_report.py




```
#!/usr/bin/env python

# override the default reporting of coords

import matplotlib.pyplot as plt
import numpy as np


def millions(x):
    return '$%1.1fM' % (x*1e-6)

x = np.random.rand(20)
y = 1e7*np.random.rand(20)

fig, ax = plt.subplots()
ax.fmt_ydata = millions
plt.plot(x, y, 'o')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.45 pylab_examples example code: csd_demo.py




```
#!/usr/bin/env python
"""
Compute the cross spectral density of two signals
"""
import numpy as np
import matplotlib.pyplot as plt

# make a little extra space between the subplots
plt.subplots_adjust(wspace=0.5)

dt = 0.01
t = np.arange(0, 30, dt)
nse1 = np.random.randn(len(t))                 # white noise 1
nse2 = np.random.randn(len(t))                 # white noise 2
r = np.exp(-t/0.05)

cnse1 = np.convolve(nse1, r, mode='same')*dt   # colored noise 1
cnse2 = np.convolve(nse2, r, mode='same')*dt   # colored noise 2

# two signals with a coherent part and a random part
s1 = 0.01*np.sin(2*np.pi*10*t) + cnse1
s2 = 0.01*np.sin(2*np.pi*10*t) + cnse2
```

```
plt.subplot(211)
plt.plot(t, s1, 'b-', t, s2, 'g-')
plt.xlim(0, 5)
plt.xlabel('time')
plt.ylabel('s1 and s2')
plt.grid(True)

plt.subplot(212)
cxy, f = plt.csd(s1, s2, 256, 1./dt)
plt.ylabel('CSD (db)')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.46 pylab_examples example code: cursor_demo.py

[source code]

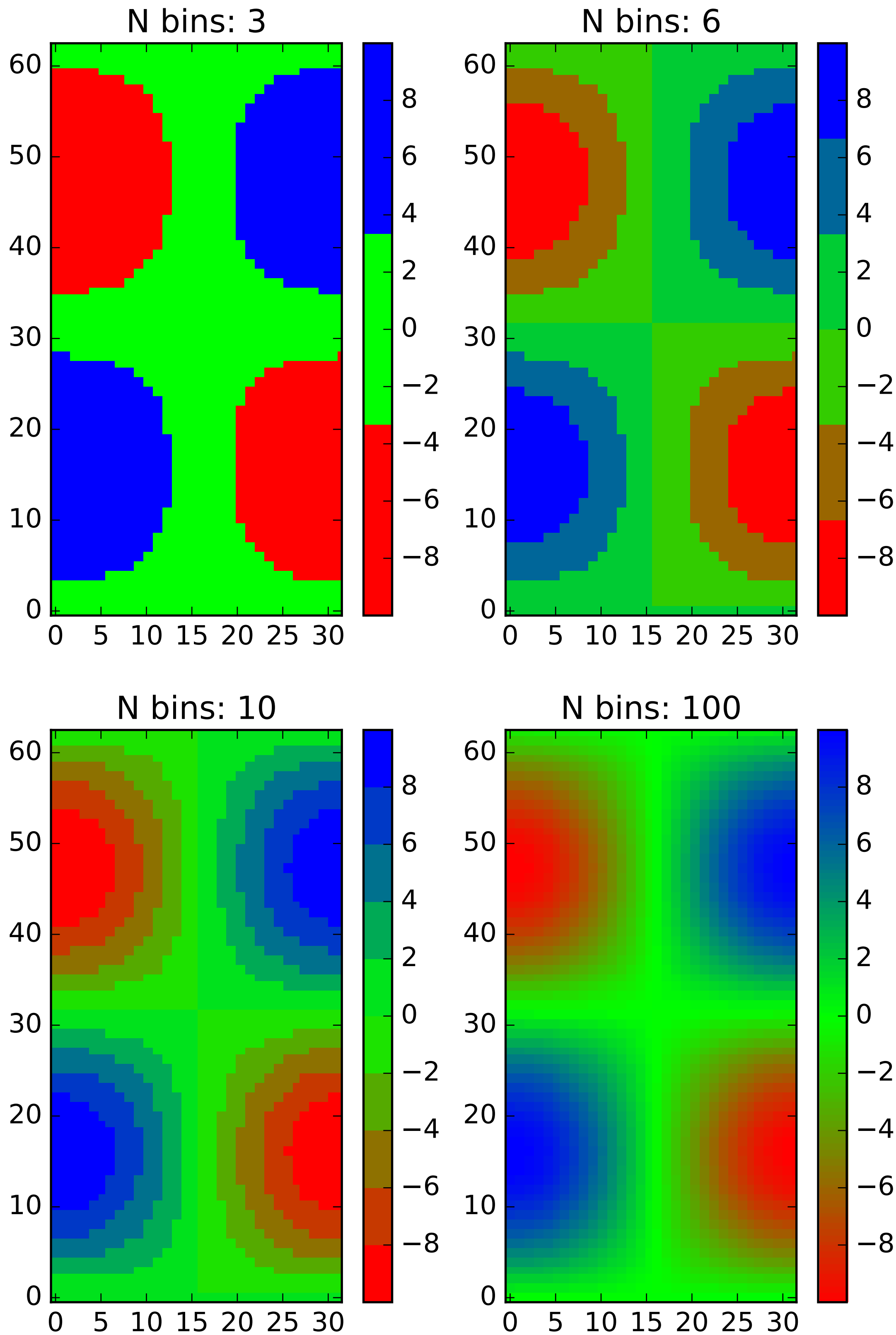
```
# -*- noplot -*-

"""
This example shows how to use matplotlib to provide a data cursor.  It
uses matplotlib to draw the cursor and may be a slow since this
requires redrawing the figure with every mouse move.

Faster cursoring is possible using native GUI drawing, as in
wxcursor_demo.py.

The mpldatacursor and mplcursors third-party packages can be used to achieve a
similar effect.  See
    https://github.com/joferkington/mpldatacursor
    https://github.com/anntzer/mplcursors
"""
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np


class Cursor(object):
    def __init__(self, ax):
        self.ax = ax
        self.lx = ax.axhline(color='k')  # the horiz line
        self.ly = ax.axvline(color='k')  # the vert line

        # text location in axes coords
        self.txt = ax.text(0.7, 0.9, '', transform=ax.transAxes)

    def mouse_move(self, event):
        if not event.inaxes:
            return
```

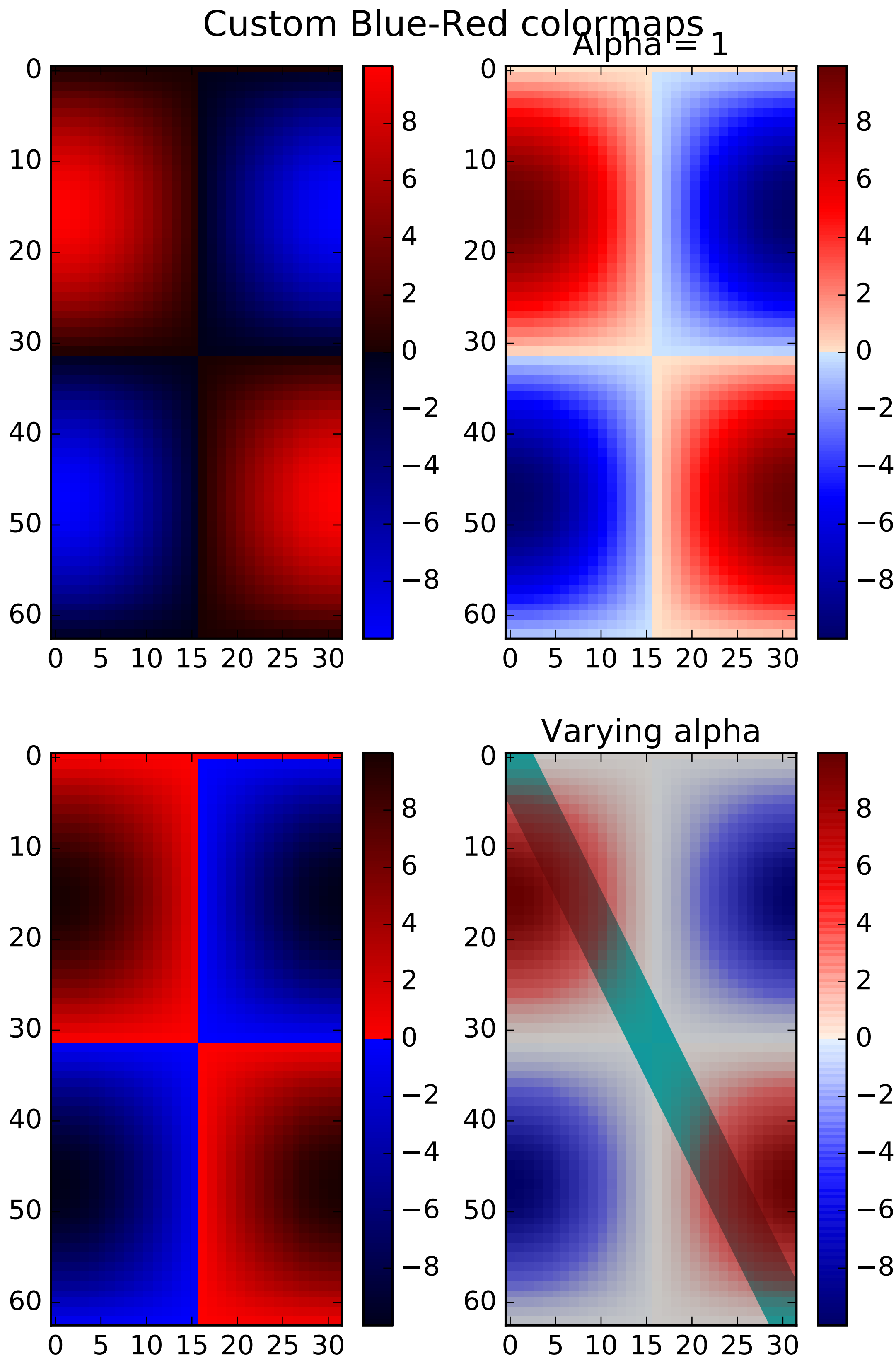
```
        x, y = event.xdata, event.ydata
        # update the line positions
        self.lx.set_ydata(y)
        self.ly.set_xdata(x)

        self.txt.set_text('x=%1.2f, y=%1.2f' % (x, y))
        plt.draw()


class SnaptoCursor(object):
    """
    Like Cursor but the crosshair snaps to the nearest x,y point
    For simplicity, I'm assuming x is sorted
    """

    def __init__(self, ax, x, y):
        self.ax = ax
        self.lx = ax.axhline(color='k')  # the horiz line
        self.ly = ax.axvline(color='k')  # the vert line
        self.x = x
        self.y = y
        # text location in axes coords
        self.txt = ax.text(0.7, 0.9, '', transform=ax.transAxes)

    def mouse_move(self, event):

        if not event.inaxes:
            return

        x, y = event.xdata, event.ydata

        indx = np.searchsorted(self.x, [x])[0]
        x = self.x[indx]
        y = self.y[indx]
        # update the line positions
        self.lx.set_ydata(y)
        self.ly.set_xdata(x)

        self.txt.set_text('x=%1.2f, y=%1.2f' % (x, y))
        print('x=%1.2f, y=%1.2f' % (x, y))
        plt.draw()

t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2*2*np.pi*t)
fig, ax = plt.subplots()

#cursor = Cursor(ax)
cursor = SnaptoCursor(ax, t, s)
plt.connect('motion_notify_event', cursor.mouse_move)

ax.plot(t, s, 'o')
plt.axis([0, 1, -1, 1])
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.47 pylab_examples example code: custom_cmap.py




Custom Blue-Red colormaps
Alpha = 1
Varying alpha
0 10 20 30 40 50 60
0 5 10 15 20 25 30
8 6 4 2 0 −2 −4 −6 −8

```
#!/usr/bin/env python

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap

"""
Creating a colormap from a list of colors
-----------------------------------------
Creating a colormap from a list of colors can be done with the `from_list`
method of `LinearSegmentedColormap`. You must pass a list of RGB tuples that
define the mixture of colors from 0 to 1.


Creating custom colormaps
-------------------------
It is also possible to create a custom mapping for a colormap. This is
accomplished by creating dictionary that specifies how the RGB channels
change from one end of the cmap to the other.

Example: suppose you want red to increase from 0 to 1 over the bottom
half, green to do the same over the middle half, and blue over the top
half.  Then you would use:

cdict = {'red':   ((0.0,  0.0, 0.0),
                   (0.5,  1.0, 1.0),
                   (1.0,  1.0, 1.0)),

         'green': ((0.0,  0.0, 0.0),
                   (0.25, 0.0, 0.0),
                   (0.75, 1.0, 1.0),
                   (1.0,  1.0, 1.0)),

         'blue':  ((0.0,  0.0, 0.0),
                   (0.5,  0.0, 0.0),
                   (1.0,  1.0, 1.0))}

If, as in this example, there are no discontinuities in the r, g, and b
components, then it is quite simple: the second and third element of
each tuple, above, is the same--call it "y".  The first element ("x")
defines interpolation intervals over the full range of 0 to 1, and it
must span that whole range.  In other words, the values of x divide the
0-to-1 range into a set of segments, and y gives the end-point color
values for each segment.

Now consider the green. cdict['green'] is saying that for
0 <= x <= 0.25, y is zero; no green.
0.25 < x <= 0.75, y varies linearly from 0 to 1.
x > 0.75, y remains at 1, full green.

If there are discontinuities, then it is a little more complicated.
Label the 3 elements in each row in the cdict entry for a given color as
(x, y0, y1).  Then for values of x between x[i] and x[i+1] the color
```

```
value is interpolated between y1[i] and y0[i+1].

Going back to the cookbook example, look at cdict['red']; because y0 !=
y1, it is saying that for x from 0 to 0.5, red increases from 0 to 1,
but then it jumps down, so that for x from 0.5 to 1, red increases from
0.7 to 1.  Green ramps from 0 to 1 as x goes from 0 to 0.5, then jumps
back to 0, and ramps back to 1 as x goes from 0.5 to 1.

row i:   x  y0  y1
                /
               /
row i+1: x  y0  y1

Above is an attempt to show that for x in the range x[i] to x[i+1], the
interpolation is between y1[i] and y0[i+1].  So, y0[0] and y1[-1] are
never used.

"""
# Make some illustrative fake data:

x = np.arange(0, np.pi, 0.1)
y = np.arange(0, 2*np.pi, 0.1)
X, Y = np.meshgrid(x, y)
Z = np.cos(X) * np.sin(Y) * 10


# --- Colormaps from a list ---

colors = [(1, 0, 0), (0, 1, 0), (0, 0, 1)]  # R -> G -> B
n_bins = [3, 6, 10, 100]  # Discretizes the interpolation into bins
cmap_name = 'my_list'
fig, axs = plt.subplots(2, 2, figsize=(6, 9))
fig.subplots_adjust(left=0.02, bottom=0.06, right=0.95, top=0.94, wspace=0.05)
for n_bin, ax in zip(n_bins, axs.ravel()):
    # Create the colormap
    cm = LinearSegmentedColormap.from_list(
        cmap_name, colors, N=n_bin)
    # Fewer bins will result in "coarser" colomap interpolation
    im = ax.imshow(Z, interpolation='nearest', origin='lower', cmap=cm)
    ax.set_title("N bins: %s" % n_bin)
    fig.colorbar(im, ax=ax)


# --- Custom colormaps ---

cdict1 = {'red':   ((0.0, 0.0, 0.0),
                   (0.5, 0.0, 0.1),
                   (1.0, 1.0, 1.0)),

         'green': ((0.0, 0.0, 0.0),
                   (1.0, 0.0, 0.0)),

         'blue':  ((0.0, 0.0, 1.0),
```

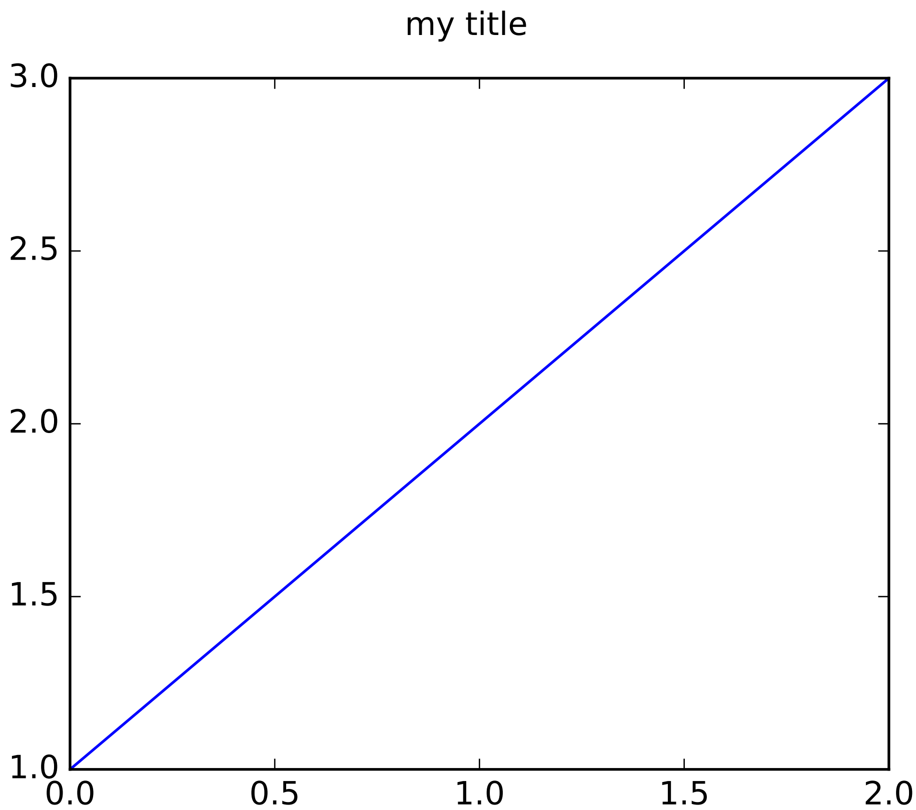
```
                   (0.5, 0.1, 0.0),
                   (1.0, 0.0, 0.0))
        }

cdict2 = {'red':   ((0.0, 0.0, 0.0),
                   (0.5, 0.0, 1.0),
                   (1.0, 0.1, 1.0)),

         'green': ((0.0, 0.0, 0.0),
                   (1.0, 0.0, 0.0)),

         'blue':  ((0.0, 0.0, 0.1),
                   (0.5, 1.0, 0.0),
                   (1.0, 0.0, 0.0))
        }

cdict3 = {'red':  ((0.0, 0.0, 0.0),
                   (0.25, 0.0, 0.0),
                   (0.5, 0.8, 1.0),
                   (0.75, 1.0, 1.0),
                   (1.0, 0.4, 1.0)),

         'green': ((0.0, 0.0, 0.0),
                   (0.25, 0.0, 0.0),
                   (0.5, 0.9, 0.9),
                   (0.75, 0.0, 0.0),
                   (1.0, 0.0, 0.0)),

         'blue':  ((0.0, 0.0, 0.4),
                   (0.25, 1.0, 1.0),
                   (0.5, 1.0, 0.8),
                   (0.75, 0.0, 0.0),
                   (1.0, 0.0, 0.0))
        }

# Make a modified version of cdict3 with some transparency
# in the middle of the range.
cdict4 = cdict3.copy()
cdict4['alpha'] = ((0.0, 1.0, 1.0),
                #   (0.25,1.0, 1.0),
                   (0.5, 0.3, 0.3),
                #   (0.75,1.0, 1.0),
                   (1.0, 1.0, 1.0))


# Now we will use this example to illustrate 3 ways of
# handling custom colormaps.
# First, the most direct and explicit:

blue_red1 = LinearSegmentedColormap('BlueRed1', cdict1)

# Second, create the map explicitly and register it.
# Like the first method, this method works with any kind
```

```
# of Colormap, not just
# a LinearSegmentedColormap:

blue_red2 = LinearSegmentedColormap('BlueRed2', cdict2)
plt.register_cmap(cmap=blue_red2)

# Third, for LinearSegmentedColormap only,
# leave everything to register_cmap:

plt.register_cmap(name='BlueRed3', data=cdict3)  # optional lut kwarg
plt.register_cmap(name='BlueRedAlpha', data=cdict4)

# Make the figure:

fig, axs = plt.subplots(2, 2, figsize=(6, 9))
fig.subplots_adjust(left=0.02, bottom=0.06, right=0.95, top=0.94, wspace=0.05)

# Make 4 subplots:

im1 = axs[0, 0].imshow(Z, interpolation='nearest', cmap=blue_red1)
fig.colorbar(im1, ax=axs[0, 0])

cmap = plt.get_cmap('BlueRed2')
im2 = axs[1, 0].imshow(Z, interpolation='nearest', cmap=cmap)
fig.colorbar(im2, ax=axs[1, 0])

# Now we will set the third cmap as the default.  One would
# not normally do this in the middle of a script like this;
# it is done here just to illustrate the method.

plt.rcParams['image.cmap'] = 'BlueRed3'

im3 = axs[0, 1].imshow(Z, interpolation='nearest')
fig.colorbar(im3, ax=axs[0, 1])
axs[0, 1].set_title("Alpha = 1")

# Or as yet another variation, we can replace the rcParams
# specification *before* the imshow with the following *after*
# imshow.
# This sets the new default *and* sets the colormap of the last
# image-like item plotted via pyplot, if any.
#

# Draw a line with low zorder so it will be behind the image.
axs[1, 1].plot([0, 10*np.pi], [0, 20*np.pi], color='c', lw=20, zorder=-1)

im4 = axs[1, 1].imshow(Z, interpolation='nearest')
fig.colorbar(im4, ax=axs[1, 1])

# Here it is: changing the colormap for the current image and its
# colorbar after they have been plotted.
im4.set_cmap('BlueRedAlpha')
axs[1, 1].set_title("Varying alpha")
```

```
#

fig.suptitle('Custom Blue-Red colormaps', fontsize=16)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.48 pylab_examples example code: custom_figure_class.py

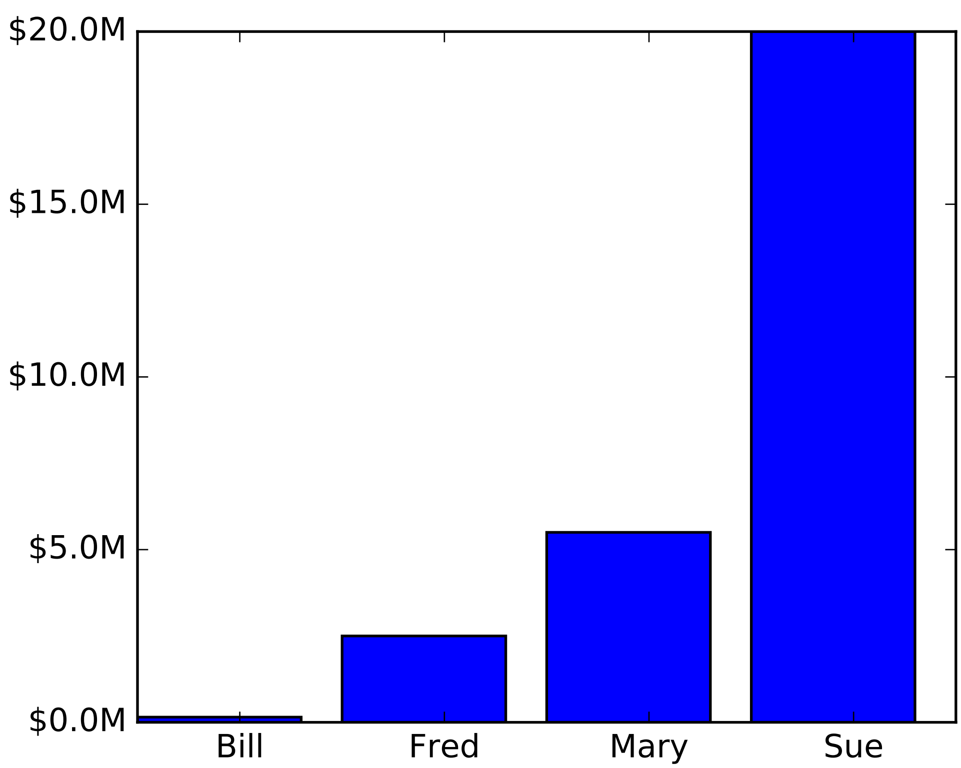



```
"""
You can pass a custom Figure constructor to figure if you want to derive from the␣
↪default Figure.  This simple example creates a figure with a figure title
"""
from matplotlib.pyplot import figure, show
from matplotlib.figure import Figure


class MyFigure(Figure):
    def __init__(self, *args, **kwargs):
        """
        custom kwarg figtitle is a figure title
```

```
        """
        figtitle = kwargs.pop('figtitle', 'hi mom')
        Figure.__init__(self, *args, **kwargs)
        self.text(0.5, 0.95, figtitle, ha='center')

fig = figure(FigureClass=MyFigure, figtitle='my title')
ax = fig.add_subplot(111)
ax.plot([1, 2, 3])

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.49 pylab_examples example code: custom_ticker1.py

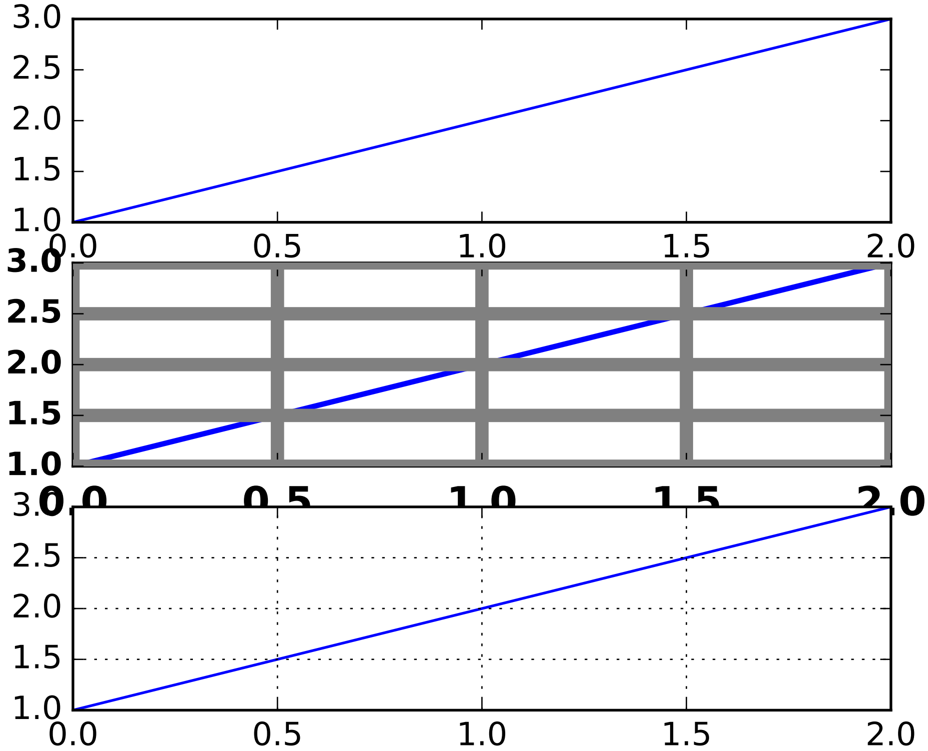



```
#!/usr/bin/env python

"""
The new ticker code was designed to explicitly support user customized
ticking.  The documentation
http://matplotlib.org/matplotlib.ticker.html details this
process.  That code defines a lot of preset tickers but was primarily
```

```
designed to be user extensible.

In this example a user defined function is used to format the ticks in
millions of dollars on the y axis
"""
from matplotlib.ticker import FuncFormatter
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(4)
money = [1.5e5, 2.5e6, 5.5e6, 2.0e7]


def millions(x, pos):
    'The two args are the value and tick position'
    return '$%1.1fM' % (x*1e-6)

formatter = FuncFormatter(millions)

fig, ax = plt.subplots()
ax.yaxis.set_major_formatter(formatter)
plt.bar(x, money)
plt.xticks(x + 0.5, ('Bill', 'Fred', 'Mary', 'Sue'))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.50 pylab_examples example code: customize_rc.py

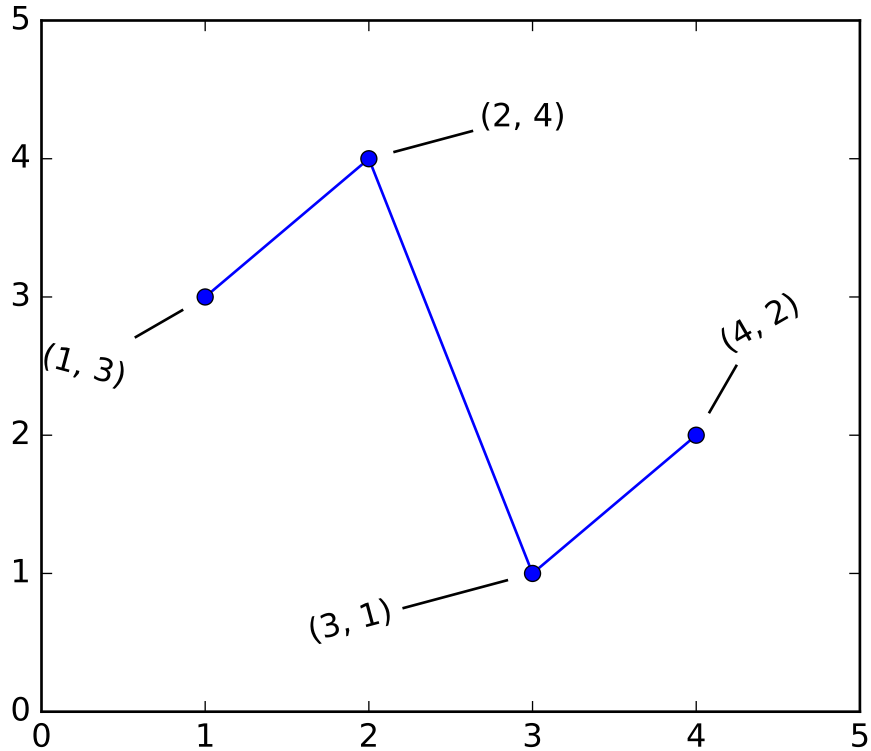



```
"""
I'm not trying to make a good looking figure here, but just to show
some examples of customizing rc params on the fly

If you like to work interactively, and need to create different sets
of defaults for figures (e.g., one set of defaults for publication, one
set for interactive exploration), you may want to define some
functions in a custom module that set the defaults, e.g.,

def set_pub():
    rc('font', weight='bold')    # bold fonts are easier to see
    rc('tick', labelsize=15)     # tick labels bigger
    rc('lines', lw=1, color='k') # thicker black lines (no budget for color!)
    rc('grid', c='0.5', ls='-', lw=0.5)  # solid gray grid lines
    rc('savefig', dpi=300)       # higher res outputs



Then as you are working interactively, you just need to do

>>> set_pub()
>>> subplot(111)
```

```
>>> plot([1,2,3])
>>> savefig('myfig')
>>> rcdefaults()  # restore the defaults

"""
import matplotlib.pyplot as plt

plt.subplot(311)
plt.plot([1, 2, 3])

# the axes attributes need to be set before the call to subplot
plt.rc('font', weight='bold')
plt.rc('xtick.major', size=5, pad=7)
plt.rc('xtick', labelsize=15)

# using aliases for color, linestyle and linewidth; gray, solid, thick
plt.rc('grid', c='0.5', ls='-', lw=5)
plt.rc('lines', lw=2, color='g')
plt.subplot(312)

plt.plot([1, 2, 3])
plt.grid(True)

plt.rcdefaults()
plt.subplot(313)
plt.plot([1, 2, 3])
plt.grid(True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.51 pylab_examples example code: dashpointlabel.py




```
import matplotlib.pyplot as plt

DATA = ((1, 3),
        (2, 4),
        (3, 1),
        (4, 2))
# dash_style =
#     direction, length, (text)rotation, dashrotation, push
# (The parameters are varied to show their effects,
# not for visual appeal).
dash_style = (
    (0, 20, -15, 30, 10),
    (1, 30, 0, 15, 10),
    (0, 40, 15, 15, 10),
    (1, 20, 30, 60, 10),
    )

fig, ax = plt.subplots()

(x, y) = zip(*DATA)
ax.plot(x, y, marker='o')
for i in range(len(DATA)):
```

```
    (x, y) = DATA[i]
    (dd, dl, r, dr, dp) = dash_style[i]
    #print('dashlen call %d' % dl)
    t = ax.text(x, y, str((x, y)), withdash=True,
                dashdirection=dd,
                dashlength=dl,
                rotation=r,
                dashrotation=dr,
                dashpush=dp,
                )

ax.set_xlim((0.0, 5.0))
ax.set_ylim((0.0, 5.0))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.52 pylab_examples example code: data_helper.py

```
#!/usr/bin/env python
# Some functions to load a return data for the plot demos

from numpy import fromstring, argsort, take, array, resize
import matplotlib.cbook as cbook


def get_two_stock_data():
    """
    load stock time and price data for two stocks The return values
    (d1,p1,d2,p2) are the trade time (in days) and prices for stocks 1
    and 2 (intc and aapl)
    """
    ticker1, ticker2 = 'INTC', 'AAPL'

    file1 = cbook.get_sample_data('INTC.dat.gz')
    file2 = cbook.get_sample_data('AAPL.dat.gz')
    M1 = fromstring(file1.read(), '<d')

    M1 = resize(M1, (M1.shape[0]//2, 2))

    M2 = fromstring(file2.read(), '<d')
    M2 = resize(M2, (M2.shape[0]//2, 2))

    d1, p1 = M1[:, 0], M1[:, 1]
    d2, p2 = M2[:, 0], M2[:, 1]
    return (d1, p1, d2, p2)


def get_daily_data():
    """
```

```
    return stock1 and stock2 instances, each of which have attributes

      open, high, low, close, volume

    as numeric arrays

    """
    class C:
        pass

    def get_ticker(ticker):
        vals = []

        datafile = cbook.get_sample_data('%s.csv' % ticker, asfileobj=False)

        lines = open(datafile).readlines()
        for line in lines[1:]:
            vals.append([float(val) for val in line.split(',')[1:]])

        M = array(vals)
        c = C()
        c.open = M[:, 0]
        c.high = M[:, 1]
        c.low = M[:, 2]
        c.close = M[:, 3]
        c.volume = M[:, 4]
        return c
    c1 = get_ticker('intc')
    c2 = get_ticker('msft')
    return c1, c2
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

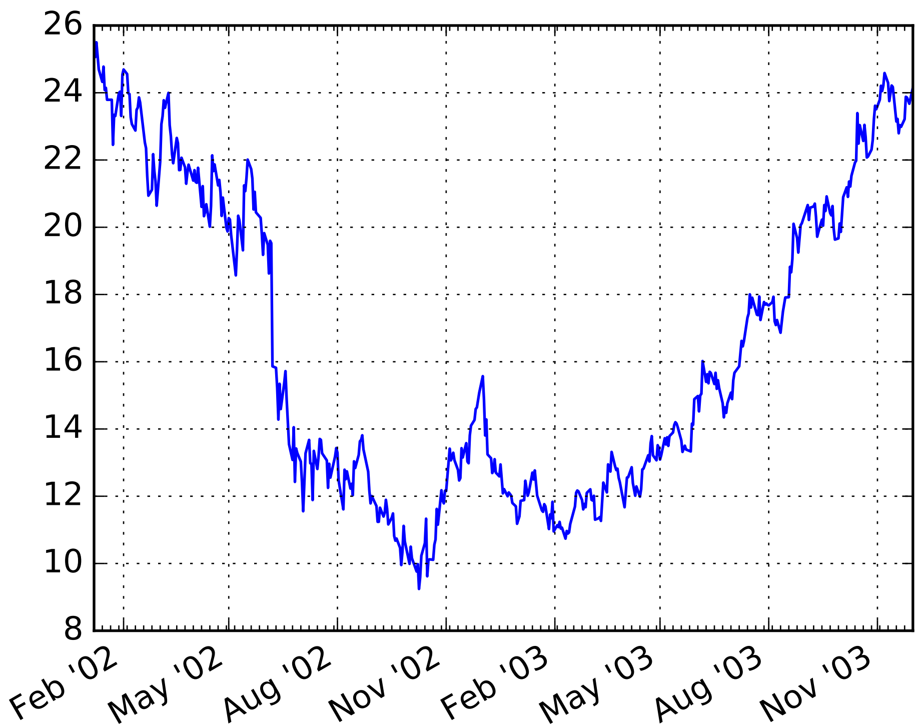
90.53 pylab_examples example code: date_demo1.py


```
#!/usr/bin/env python
"""
Show how to make date plots in matplotlib using date tick locators and
formatters.  See major_minor_demo1.py for more information on
controlling major and minor ticks

All matplotlib date plotting is done by converting date instances into
days since the 0001-01-01 UTC.  The conversion, tick locating and
formatting is done behind the scenes so this is most transparent to
you.  The dates module provides several converter functions date2num
and num2date

This example requires an active internet connection since it uses
yahoo finance to get the data for plotting
"""

import matplotlib.pyplot as plt
from matplotlib.finance import quotes_historical_yahoo_ochl
from matplotlib.dates import YearLocator, MonthLocator, DateFormatter
import datetime
date1 = datetime.date(1995, 1, 1)
date2 = datetime.date(2004, 4, 12)
```

```
years = YearLocator()   # every year
months = MonthLocator()  # every month
yearsFmt = DateFormatter('%Y')

quotes = quotes_historical_yahoo_ochl('INTC', date1, date2)
if len(quotes) == 0:
    raise SystemExit

dates = [q[0] for q in quotes]
opens = [q[1] for q in quotes]

fig, ax = plt.subplots()
ax.plot_date(dates, opens, '-')

# format the ticks
ax.xaxis.set_major_locator(years)
ax.xaxis.set_major_formatter(yearsFmt)
ax.xaxis.set_minor_locator(months)
ax.autoscale_view()


# format the coords message box
def price(x):
    return '$%1.2f' % x
ax.fmt_xdata = DateFormatter('%Y-%m-%d')
ax.fmt_ydata = price
ax.grid(True)

fig.autofmt_xdate()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.54 pylab_examples example code: date_demo2.py

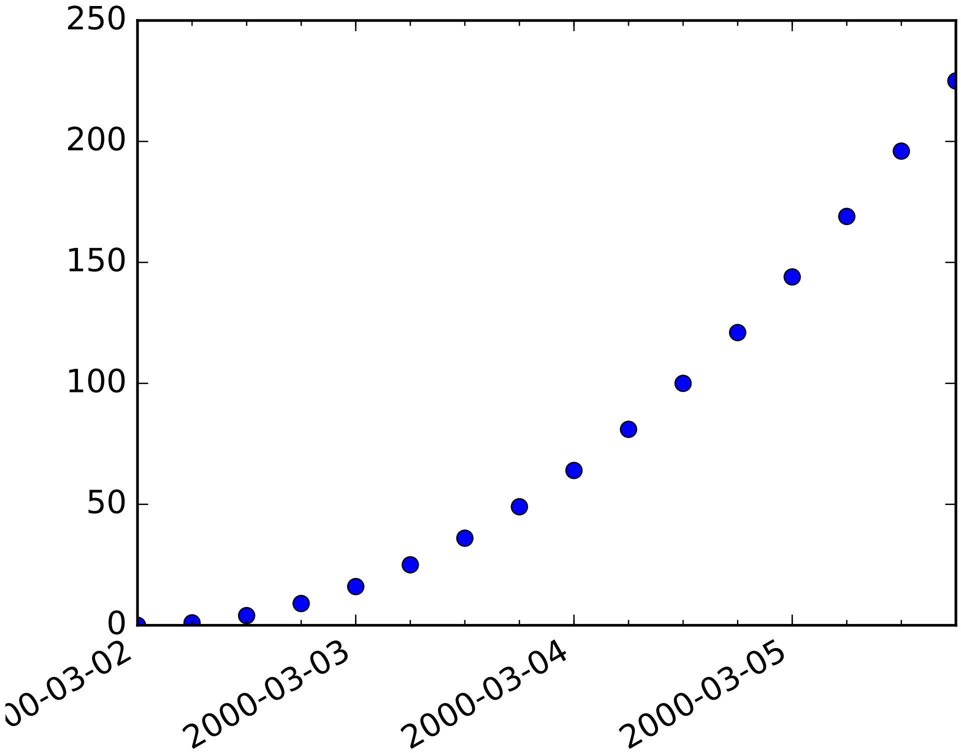



```
#!/usr/bin/env python

"""
Show how to make date plots in matplotlib using date tick locators and
formatters.  See major_minor_demo1.py for more information on
controlling major and minor ticks
"""
from __future__ import print_function
import datetime
import matplotlib.pyplot as plt
from matplotlib.dates import MONDAY
from matplotlib.finance import quotes_historical_yahoo_ochl
from matplotlib.dates import MonthLocator, WeekdayLocator, DateFormatter


date1 = datetime.date(2002, 1, 5)
date2 = datetime.date(2003, 12, 1)

# every monday
mondays = WeekdayLocator(MONDAY)

# every 3rd month
```

```
months = MonthLocator(range(1, 13), bymonthday=1, interval=3)
monthsFmt = DateFormatter("%b '%y")


quotes = quotes_historical_yahoo_ochl('INTC', date1, date2)
if len(quotes) == 0:
    print('Found no quotes')
    raise SystemExit

dates = [q[0] for q in quotes]
opens = [q[1] for q in quotes]

fig, ax = plt.subplots()
ax.plot_date(dates, opens, '-')
ax.xaxis.set_major_locator(months)
ax.xaxis.set_major_formatter(monthsFmt)
ax.xaxis.set_minor_locator(mondays)
ax.autoscale_view()
#ax.xaxis.grid(False, 'major')
#ax.xaxis.grid(True, 'minor')
ax.grid(True)

fig.autofmt_xdate()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.55 pylab_examples example code: date_demo_convert.py

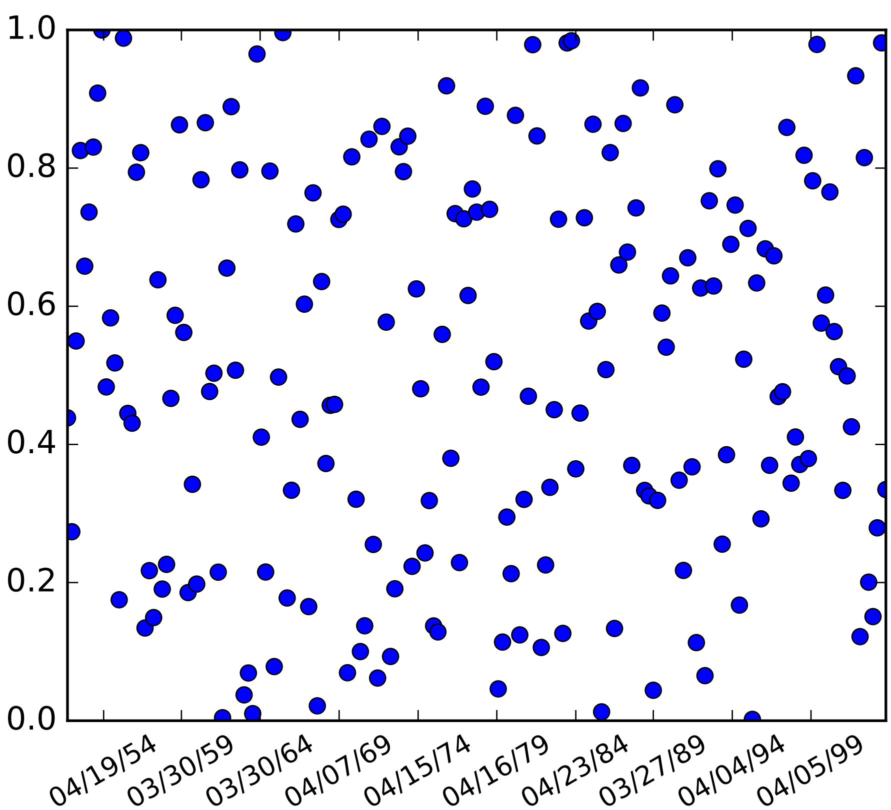



```python
#!/usr/bin/env python

import datetime
import matplotlib.pyplot as plt
from matplotlib.dates import DayLocator, HourLocator, DateFormatter, drange
from numpy import arange

date1 = datetime.datetime(2000, 3, 2)
date2 = datetime.datetime(2000, 3, 6)
delta = datetime.timedelta(hours=6)
dates = drange(date1, date2, delta)

y = arange(len(dates)*1.0)

fig, ax = plt.subplots()
ax.plot_date(dates, y*y)

# this is superfluous, since the autoscaler should get it right, but
# use date2num and num2date to convert between dates and floats if
# you want; both date2num and num2date convert an instance or sequence
ax.set_xlim(dates[0], dates[-1])
```

```
# The hour locator takes the hour or sequence of hours you want to
# tick, not the base multiple

ax.xaxis.set_major_locator(DayLocator())
ax.xaxis.set_minor_locator(HourLocator(arange(0, 25, 6)))
ax.xaxis.set_major_formatter(DateFormatter('%Y-%m-%d'))

ax.fmt_xdata = DateFormatter('%Y-%m-%d %H:%M:%S')
fig.autofmt_xdate()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.56 pylab_examples example code: date_demo_rrule.py

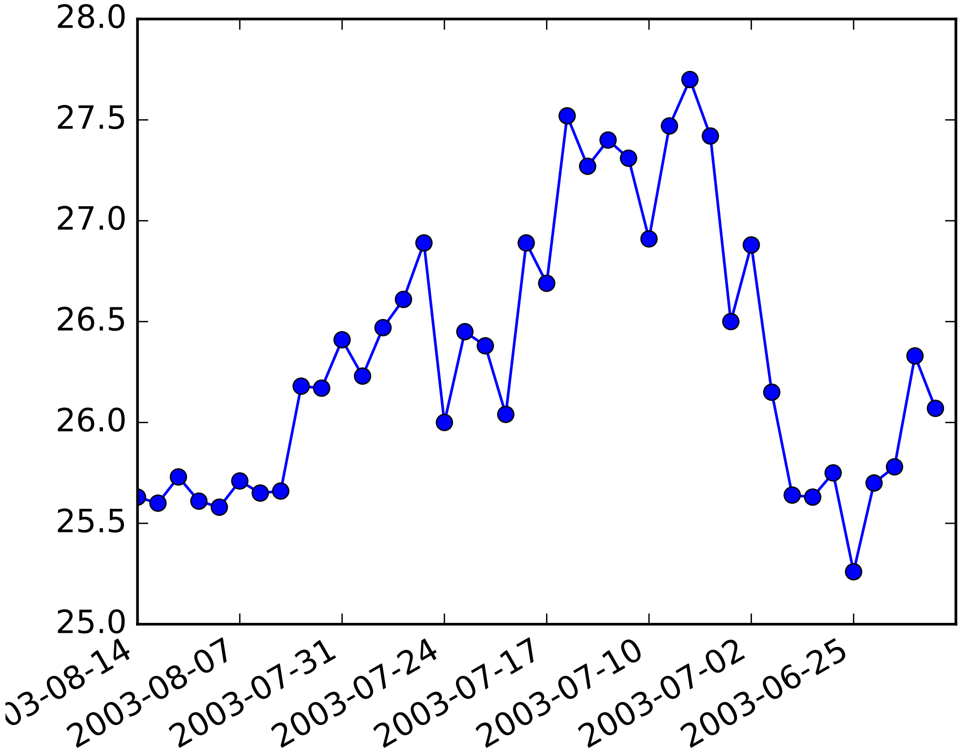



```
#!/usr/bin/env python
"""
Show how to use an rrule instance to make a custom date ticker - here
we put a tick mark on every 5th easter

See https://moin.conectiva.com.br/DateUtil for help with rrules
```

```
"""
import matplotlib.pyplot as plt
from matplotlib.dates import YEARLY, DateFormatter, rrulewrapper, RRuleLocator, drange
import numpy as np
import datetime

# tick every 5th easter
rule = rrulewrapper(YEARLY, byeaster=1, interval=5)
loc = RRuleLocator(rule)
formatter = DateFormatter('%m/%d/%y')
date1 = datetime.date(1952, 1, 1)
date2 = datetime.date(2004, 4, 12)
delta = datetime.timedelta(days=100)

dates = drange(date1, date2, delta)
s = np.random.rand(len(dates))  # make up some random y values


fig, ax = plt.subplots()
plt.plot_date(dates, s)
ax.xaxis.set_major_locator(loc)
ax.xaxis.set_major_formatter(formatter)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=30, fontsize=10)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.57 pylab_examples example code: date_index_formatter.py

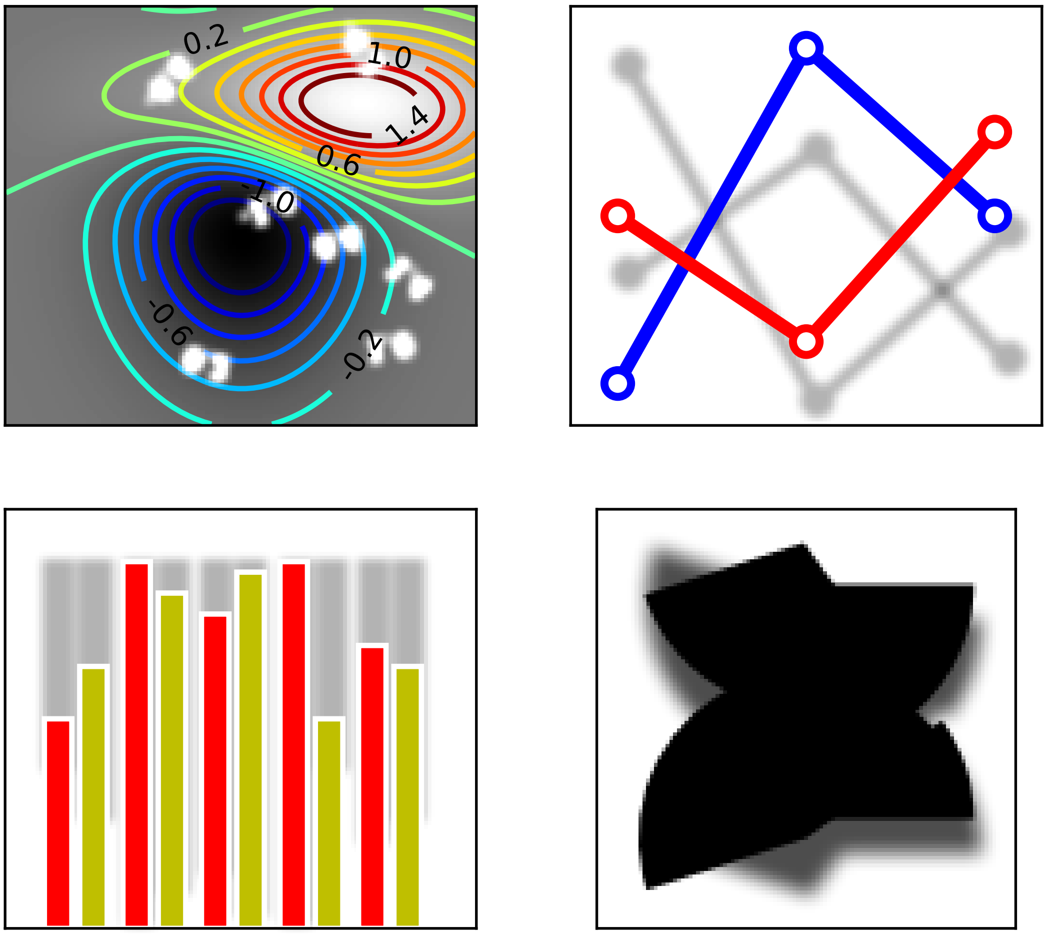



```
"""
When plotting daily data, a frequent request is to plot the data
ignoring skips, e.g., no extra spaces for weekends.  This is particularly
common in financial time series, when you may have data for M-F and
not Sat, Sun and you don't want gaps in the x axis.  The approach is
to simply use the integer index for the xdata and a custom tick
Formatter to get the appropriate date string for a given index.
"""

from __future__ import print_function
import numpy
from matplotlib.mlab import csv2rec
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
from matplotlib.ticker import Formatter

datafile = cbook.get_sample_data('msft.csv', asfileobj=False)
print('loading %s' % datafile)
r = csv2rec(datafile)[-40:]


class MyFormatter(Formatter):
```

```
    def __init__(self, dates, fmt='%Y-%m-%d'):
        self.dates = dates
        self.fmt = fmt

    def __call__(self, x, pos=0):
        'Return the label for time x at position pos'
        ind = int(round(x))
        if ind >= len(self.dates) or ind < 0:
            return ''

        return self.dates[ind].strftime(self.fmt)

formatter = MyFormatter(r.date)

fig, ax = plt.subplots()
ax.xaxis.set_major_formatter(formatter)
ax.plot(numpy.arange(len(r)), r.close, 'o-')
fig.autofmt_xdate()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.58 pylab_examples example code: demo_agg_filter.py




```
import matplotlib.pyplot as plt

import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab


def smooth1d(x, window_len):
    # copied from http://www.scipy.org/Cookbook/SignalSmooth

    s = np.r_[2*x[0] - x[window_len:1:-1], x, 2*x[-1] - x[-1:-window_len:-1]]
    w = np.hanning(window_len)
    y = np.convolve(w/w.sum(), s, mode='same')
```

```
    return y[window_len-1:-window_len+1]


def smooth2d(A, sigma=3):

    window_len = max(int(sigma), 3)*2 + 1
    A1 = np.array([smooth1d(x, window_len) for x in np.asarray(A)])
    A2 = np.transpose(A1)
    A3 = np.array([smooth1d(x, window_len) for x in A2])
    A4 = np.transpose(A3)

    return A4


class BaseFilter(object):
    def prepare_image(self, src_image, dpi, pad):
        ny, nx, depth = src_image.shape
        #tgt_image = np.zeros([pad*2+ny, pad*2+nx, depth], dtype="d")
        padded_src = np.zeros([pad*2 + ny, pad*2 + nx, depth], dtype="d")
        padded_src[pad:-pad, pad:-pad, :] = src_image[:, :, :]

        return padded_src  # , tgt_image

    def get_pad(self, dpi):
        return 0

    def __call__(self, im, dpi):
        pad = self.get_pad(dpi)
        padded_src = self.prepare_image(im, dpi, pad)
        tgt_image = self.process_image(padded_src, dpi)
        return tgt_image, -pad, -pad


class OffsetFilter(BaseFilter):
    def __init__(self, offsets=None):
        if offsets is None:
            self.offsets = (0, 0)
        else:
            self.offsets = offsets

    def get_pad(self, dpi):
        return int(max(*self.offsets)/72.*dpi)

    def process_image(self, padded_src, dpi):
        ox, oy = self.offsets
        a1 = np.roll(padded_src, int(ox/72.*dpi), axis=1)
        a2 = np.roll(a1, -int(oy/72.*dpi), axis=0)
        return a2


class GaussianFilter(BaseFilter):
    "simple gauss filter"
```

```
    def __init__(self, sigma, alpha=0.5, color=None):
        self.sigma = sigma
        self.alpha = alpha
        if color is None:
            self.color = (0, 0, 0)
        else:
            self.color = color

    def get_pad(self, dpi):
        return int(self.sigma*3/72.*dpi)

    def process_image(self, padded_src, dpi):
        #offsetx, offsety = int(self.offsets[0]), int(self.offsets[1])
        tgt_image = np.zeros_like(padded_src)
        aa = smooth2d(padded_src[:, :, -1]*self.alpha,
                      self.sigma/72.*dpi)
        tgt_image[:, :, -1] = aa
        tgt_image[:, :, :-1] = self.color
        return tgt_image


class DropShadowFilter(BaseFilter):
    def __init__(self, sigma, alpha=0.3, color=None, offsets=None):
        self.gauss_filter = GaussianFilter(sigma, alpha, color)
        self.offset_filter = OffsetFilter(offsets)

    def get_pad(self, dpi):
        return max(self.gauss_filter.get_pad(dpi),
                   self.offset_filter.get_pad(dpi))

    def process_image(self, padded_src, dpi):
        t1 = self.gauss_filter.process_image(padded_src, dpi)
        t2 = self.offset_filter.process_image(t1, dpi)
        return t2


from matplotlib.colors import LightSource


class LightFilter(BaseFilter):
    "simple gauss filter"

    def __init__(self, sigma, fraction=0.5):
        self.gauss_filter = GaussianFilter(sigma, alpha=1)
        self.light_source = LightSource()
        self.fraction = fraction

    def get_pad(self, dpi):
        return self.gauss_filter.get_pad(dpi)

    def process_image(self, padded_src, dpi):
        t1 = self.gauss_filter.process_image(padded_src, dpi)
        elevation = t1[:, :, 3]
```

```
        rgb = padded_src[:, :, :3]

        rgb2 = self.light_source.shade_rgb(rgb, elevation,
                                           fraction=self.fraction)

        tgt = np.empty_like(padded_src)
        tgt[:, :, :3] = rgb2
        tgt[:, :, 3] = padded_src[:, :, 3]

        return tgt


class GrowFilter(BaseFilter):
    "enlarge the area"

    def __init__(self, pixels, color=None):
        self.pixels = pixels
        if color is None:
            self.color = (1, 1, 1)
        else:
            self.color = color

    def __call__(self, im, dpi):
        pad = self.pixels
        ny, nx, depth = im.shape
        new_im = np.empty([pad*2 + ny, pad*2 + nx, depth], dtype="d")
        alpha = new_im[:, :, 3]
        alpha.fill(0)
        alpha[pad:-pad, pad:-pad] = im[:, :, -1]
        alpha2 = np.clip(smooth2d(alpha, self.pixels/72.*dpi) * 5, 0, 1)
        new_im[:, :, -1] = alpha2
        new_im[:, :, :-1] = self.color
        offsetx, offsety = -pad, -pad

        return new_im, offsetx, offsety


from matplotlib.artist import Artist


class FilteredArtistList(Artist):
    """
    A simple container to draw filtered artist.
    """

    def __init__(self, artist_list, filter):
        self._artist_list = artist_list
        self._filter = filter
        Artist.__init__(self)

    def draw(self, renderer):
        renderer.start_rasterizing()
        renderer.start_filter()
```

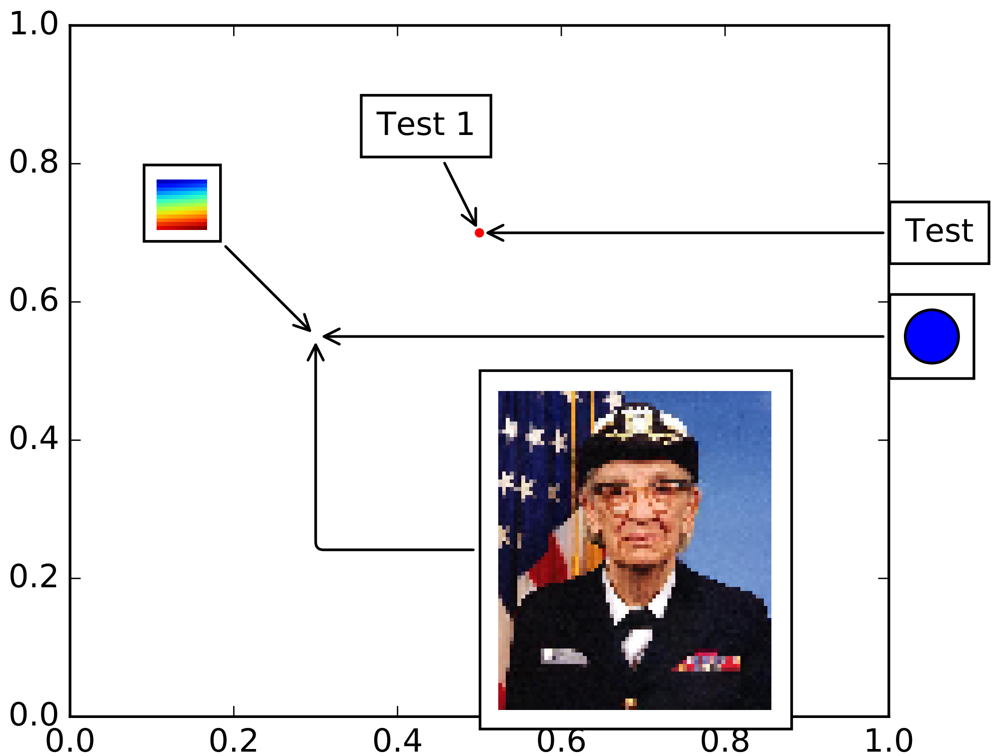
```
        for a in self._artist_list:
            a.draw(renderer)
        renderer.stop_filter(self._filter)
        renderer.stop_rasterizing()


import matplotlib.transforms as mtransforms


def filtered_text(ax):
    # mostly copied from contour_demo.py

    # prepare image
    delta = 0.025
    x = np.arange(-3.0, 3.0, delta)
    y = np.arange(-2.0, 2.0, delta)
    X, Y = np.meshgrid(x, y)
    Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
    Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
    # difference of Gaussians
    Z = 10.0 * (Z2 - Z1)

    # draw
    im = ax.imshow(Z, interpolation='bilinear', origin='lower',
                   cmap=cm.gray, extent=(-3, 3, -2, 2))
    levels = np.arange(-1.2, 1.6, 0.2)
    CS = ax.contour(Z, levels,
                    origin='lower',
                    linewidths=2,
                    extent=(-3, 3, -2, 2))

    ax.set_aspect("auto")

    # contour label
    cl = ax.clabel(CS, levels[1::2],  # label every second level
                   inline=1,
                   fmt='%1.1f',
                   fontsize=11)

    # change clable color to black
    from matplotlib.patheffects import Normal
    for t in cl:
        t.set_color("k")
        # to force TextPath (i.e., same font in all backends)
        t.set_path_effects([Normal()])

    # Add white glows to improve visibility of labels.
    white_glows = FilteredArtistList(cl, GrowFilter(3))
    ax.add_artist(white_glows)
    white_glows.set_zorder(cl[0].get_zorder() - 0.1)

    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
```

```
def drop_shadow_line(ax):
    # copied from examples/misc/svg_filter_line.py

    # draw lines
    l1, = ax.plot([0.1, 0.5, 0.9], [0.1, 0.9, 0.5], "bo-",
                  mec="b", mfc="w", lw=5, mew=3, ms=10, label="Line 1")
    l2, = ax.plot([0.1, 0.5, 0.9], [0.5, 0.2, 0.7], "ro-",
                  mec="r", mfc="w", lw=5, mew=3, ms=10, label="Line 1")

    gauss = DropShadowFilter(4)

    for l in [l1, l2]:

        # draw shadows with same lines with slight offset.

        xx = l.get_xdata()
        yy = l.get_ydata()
        shadow, = ax.plot(xx, yy)
        shadow.update_from(l)

        # offset transform
        ot = mtransforms.offset_copy(l.get_transform(), ax.figure,
                                     x=4.0, y=-6.0, units='points')

        shadow.set_transform(ot)

        # adjust zorder of the shadow lines so that it is drawn below the
        # original lines
        shadow.set_zorder(l.get_zorder() - 0.5)
        shadow.set_agg_filter(gauss)
        shadow.set_rasterized(True)  # to support mixed-mode renderers

    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)

    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)


def drop_shadow_patches(ax):
    # copyed from barchart_demo.py
    N = 5
    menMeans = (20, 35, 30, 35, 27)

    ind = np.arange(N)  # the x locations for the groups
    width = 0.35       # the width of the bars

    rects1 = ax.bar(ind, menMeans, width, color='r', ec="w", lw=2)

    womenMeans = (25, 32, 34, 20, 25)
    rects2 = ax.bar(ind + width + 0.1, womenMeans, width, color='y', ec="w", lw=2)
```

```
    #gauss = GaussianFilter(1.5, offsets=(1,1), )
    gauss = DropShadowFilter(5, offsets=(1, 1), )
    shadow = FilteredArtistList(rects1 + rects2, gauss)
    ax.add_artist(shadow)
    shadow.set_zorder(rects1[0].get_zorder() - 0.1)

    ax.set_xlim(ind[0] - 0.5, ind[-1] + 1.5)
    ax.set_ylim(0, 40)

    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)


def light_filter_pie(ax):
    fracs = [15, 30, 45, 10]
    explode = (0, 0.05, 0, 0)
    pies = ax.pie(fracs, explode=explode)
    ax.patch.set_visible(True)

    light_filter = LightFilter(9)
    for p in pies[0]:
        p.set_agg_filter(light_filter)
        p.set_rasterized(True)  # to support mixed-mode renderers
        p.set(ec="none",
              lw=2)

    gauss = DropShadowFilter(9, offsets=(3, 4), alpha=0.7)
    shadow = FilteredArtistList(pies[0], gauss)
    ax.add_artist(shadow)
    shadow.set_zorder(pies[0][0].get_zorder() - 0.1)


if 1:

    plt.figure(1, figsize=(6, 6))
    plt.subplots_adjust(left=0.05, right=0.95)

    ax = plt.subplot(221)
    filtered_text(ax)

    ax = plt.subplot(222)
    drop_shadow_line(ax)

    ax = plt.subplot(223)
    drop_shadow_patches(ax)

    ax = plt.subplot(224)
    ax.set_aspect(1)
    light_filter_pie(ax)
    ax.set_frame_on(True)

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.59 pylab_examples example code: demo_annotation_box.py



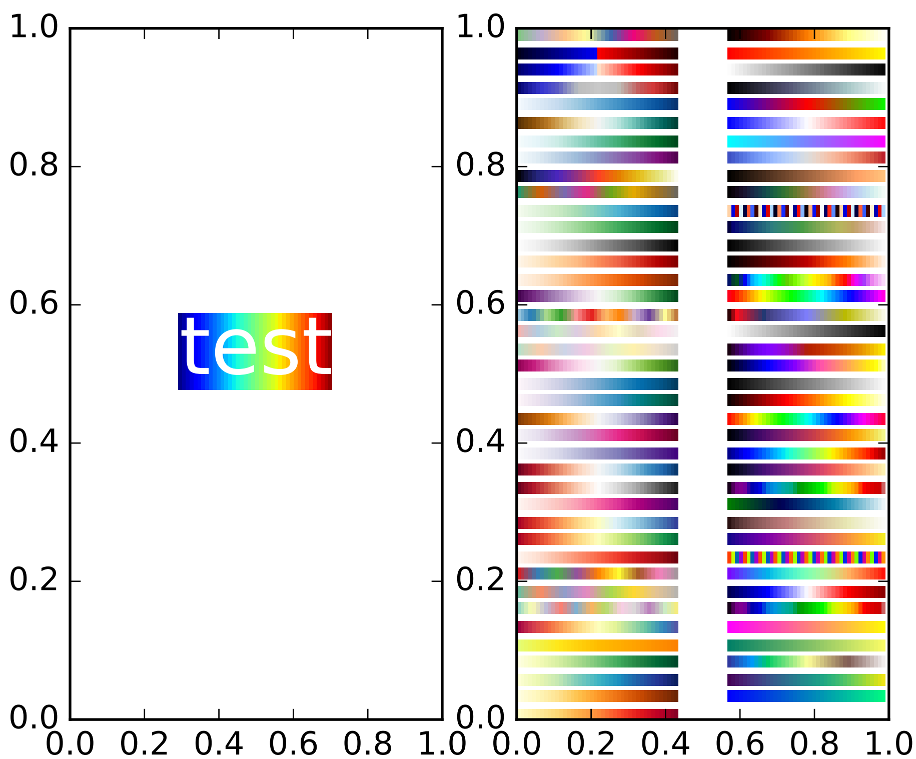

```
import matplotlib.pyplot as plt
from matplotlib.offsetbox import TextArea, DrawingArea, OffsetImage, \
    AnnotationBbox
from matplotlib.cbook import get_sample_data

import numpy as np

if 1:
    fig, ax = plt.subplots()

    offsetbox = TextArea("Test 1", minimumdescent=False)

    xy = (0.5, 0.7)

    ax.plot(xy[0], xy[1], ".r")

    ab = AnnotationBbox(offsetbox, xy,
                        xybox=(-20, 40),
                        xycoords='data',
```

```
                    boxcoords="offset points",
                    arrowprops=dict(arrowstyle="->"))
ax.add_artist(ab)

offsetbox = TextArea("Test", minimumdescent=False)

ab = AnnotationBbox(offsetbox, xy,
                    xybox=(1.02, xy[1]),
                    xycoords='data',
                    boxcoords=("axes fraction", "data"),
                    box_alignment=(0., 0.5),
                    arrowprops=dict(arrowstyle="->"))
ax.add_artist(ab)

from matplotlib.patches import Circle
da = DrawingArea(20, 20, 0, 0)
p = Circle((10, 10), 10)
da.add_artist(p)

xy = [0.3, 0.55]
ab = AnnotationBbox(da, xy,
                    xybox=(1.02, xy[1]),
                    xycoords='data',
                    boxcoords=("axes fraction", "data"),
                    box_alignment=(0., 0.5),
                    arrowprops=dict(arrowstyle="->"))

ax.add_artist(ab)

arr = np.arange(100).reshape((10, 10))
im = OffsetImage(arr, zoom=2)

ab = AnnotationBbox(im, xy,
                    xybox=(-50., 50.),
                    xycoords='data',
                    boxcoords="offset points",
                    pad=0.3,
                    arrowprops=dict(arrowstyle="->"))

ax.add_artist(ab)

# another image

from matplotlib._png import read_png
fn = get_sample_data("grace_hopper.png", asfileobj=False)
arr_lena = read_png(fn)

imagebox = OffsetImage(arr_lena, zoom=0.2)

ab = AnnotationBbox(imagebox, xy,
                    xybox=(120., -80.),
                    xycoords='data',
                    boxcoords="offset points",
```

```
                        pad=0.5,
                        arrowprops=dict(arrowstyle="->",
                                        connectionstyle="angle,angleA=0,angleB=90,rad=3")
                        )

    ax.add_artist(ab)

    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)

    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.60 pylab_examples example code: demo_bboximage.py



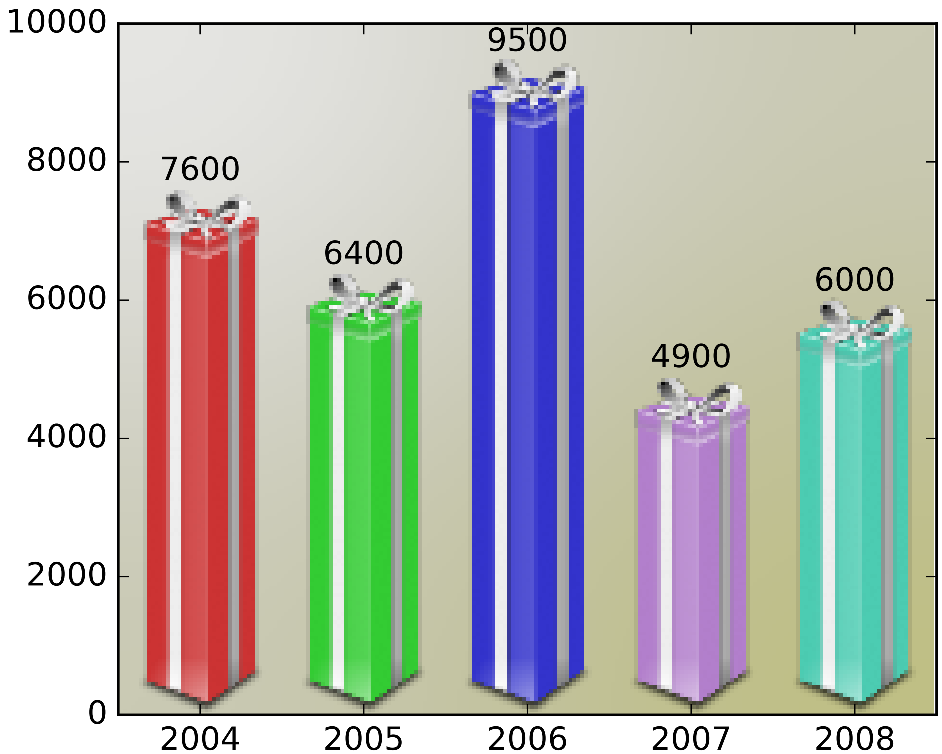

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.image import BboxImage
from matplotlib.transforms import Bbox, TransformedBbox
```

```
if __name__ == "__main__":

    fig = plt.figure(1)
    ax = plt.subplot(121)

    txt = ax.text(0.5, 0.5, "test", size=30, ha="center", color="w")
    kwargs = dict()

    bbox_image = BboxImage(txt.get_window_extent,
                           norm=None,
                           origin=None,
                           clip_on=False,
                           **kwargs
                           )
    a = np.arange(256).reshape(1, 256)/256.
    bbox_image.set_data(a)
    ax.add_artist(bbox_image)

    ax = plt.subplot(122)
    a = np.linspace(0, 1, 256).reshape(1, -1)
    a = np.vstack((a, a))

    maps = sorted(m for m in plt.cm.cmap_d if not m.endswith("_r"))
    #nmaps = len(maps) + 1

    #fig.subplots_adjust(top=0.99, bottom=0.01, left=0.2, right=0.99)

    ncol = 2
    nrow = len(maps)//ncol + 1

    xpad_fraction = 0.3
    dx = 1./(ncol + xpad_fraction*(ncol - 1))

    ypad_fraction = 0.3
    dy = 1./(nrow + ypad_fraction*(nrow - 1))

    for i, m in enumerate(maps):
        ix, iy = divmod(i, nrow)
        #plt.figimage(a, 10, i*10, cmap=plt.get_cmap(m), origin='lower')
        bbox0 = Bbox.from_bounds(ix*dx*(1 + xpad_fraction),
                                 1. - iy*dy*(1 + ypad_fraction) - dy,
                                 dx, dy)
        bbox = TransformedBbox(bbox0, ax.transAxes)

        bbox_image = BboxImage(bbox,
                               cmap=plt.get_cmap(m),
                               norm=None,
                               origin=None,
                               **kwargs
                               )

        bbox_image.set_data(a)
        ax.add_artist(bbox_image)
```

```
    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.61 pylab_examples example code: demo_ribbon_box.py




```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.image import BboxImage

from matplotlib._png import read_png
import matplotlib.colors
from matplotlib.cbook import get_sample_data


class RibbonBox(object):

    original_image = read_png(get_sample_data("Minduka_Present_Blue_Pack.png",
                                              asfileobj=False))
    cut_location = 70
```

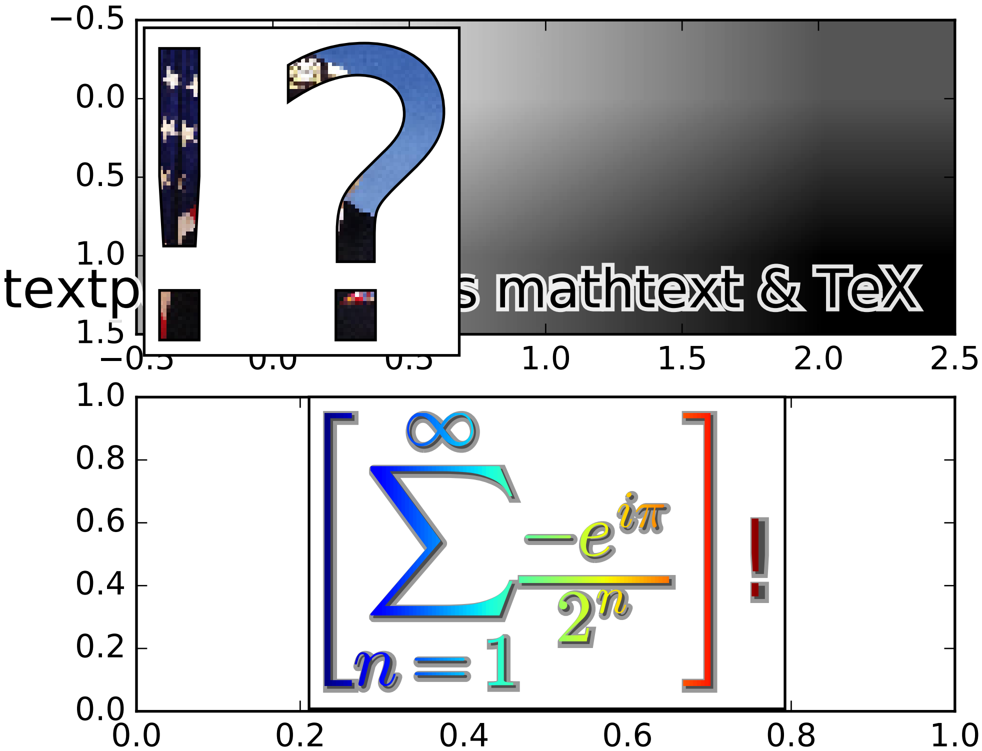
```
    b_and_h = original_image[:, :, 2]
    color = original_image[:, :, 2] - original_image[:, :, 0]
    alpha = original_image[:, :, 3]
    nx = original_image.shape[1]

    def __init__(self, color):
        rgb = matplotlib.colors.colorConverter.to_rgb(color)

        im = np.empty(self.original_image.shape,
                      self.original_image.dtype)

        im[:, :, :3] = self.b_and_h[:, :, np.newaxis]
        im[:, :, :3] -= self.color[:, :, np.newaxis]*(1. - np.array(rgb))
        im[:, :, 3] = self.alpha

        self.im = im

    def get_stretched_image(self, stretch_factor):
        stretch_factor = max(stretch_factor, 1)
        ny, nx, nch = self.im.shape
        ny2 = int(ny*stretch_factor)

        stretched_image = np.empty((ny2, nx, nch),
                                   self.im.dtype)
        cut = self.im[self.cut_location, :, :]
        stretched_image[:, :, :] = cut
        stretched_image[:self.cut_location, :, :] = \
            self.im[:self.cut_location, :, :]
        stretched_image[-(ny - self.cut_location):, :, :] = \
            self.im[-(ny - self.cut_location):, :, :]

        self._cached_im = stretched_image
        return stretched_image


class RibbonBoxImage(BboxImage):
    zorder = 1

    def __init__(self, bbox, color,
                 cmap=None,
                 norm=None,
                 interpolation=None,
                 origin=None,
                 filternorm=1,
                 filterrad=4.0,
                 resample=False,
                 **kwargs
                 ):

        BboxImage.__init__(self, bbox,
                           cmap=cmap,
                           norm=norm,
                           interpolation=interpolation,
```

```
                              origin=origin,
                              filternorm=filternorm,
                              filterrad=filterrad,
                              resample=resample,
                              **kwargs
                              )

        self._ribbonbox = RibbonBox(color)
        self._cached_ny = None

    def draw(self, renderer, *args, **kwargs):

        bbox = self.get_window_extent(renderer)
        stretch_factor = bbox.height / bbox.width

        ny = int(stretch_factor*self._ribbonbox.nx)
        if self._cached_ny != ny:
            arr = self._ribbonbox.get_stretched_image(stretch_factor)
            self.set_array(arr)
            self._cached_ny = ny

        BboxImage.draw(self, renderer, *args, **kwargs)


if 1:
    from matplotlib.transforms import Bbox, TransformedBbox
    from matplotlib.ticker import ScalarFormatter

    fig, ax = plt.subplots()

    years = np.arange(2004, 2009)
    box_colors = [(0.8, 0.2, 0.2),
                  (0.2, 0.8, 0.2),
                  (0.2, 0.2, 0.8),
                  (0.7, 0.5, 0.8),
                  (0.3, 0.8, 0.7),
                  ]
    heights = np.random.random(years.shape) * 7000 + 3000

    fmt = ScalarFormatter(useOffset=False)
    ax.xaxis.set_major_formatter(fmt)

    for year, h, bc in zip(years, heights, box_colors):
        bbox0 = Bbox.from_extents(year - 0.4, 0., year + 0.4, h)
        bbox = TransformedBbox(bbox0, ax.transData)
        rb_patch = RibbonBoxImage(bbox, bc, interpolation="bicubic")

        ax.add_artist(rb_patch)

        ax.annotate(r"%d" % (int(h/100.)*100),
                    (year, h), va="bottom", ha="center")

    patch_gradient = BboxImage(ax.bbox,
```

```
                                interpolation="bicubic",
                                zorder=0.1,
                                )
    gradient = np.zeros((2, 2, 4), dtype=np.float)
    gradient[:, :, :3] = [1, 1, 0.]
    gradient[:, :, 3] = [[0.1, 0.3], [0.3, 0.5]]  # alpha channel
    patch_gradient.set_array(gradient)
    ax.add_artist(patch_gradient)

    ax.set_xlim(years[0] - 0.5, years[-1] + 0.5)
    ax.set_ylim(0, 10000)

    fig.savefig('ribbon_box.png')
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.62 pylab_examples example code: demo_text_path.py




```
# -*- coding: utf-8 -*-

import matplotlib.pyplot as plt
```

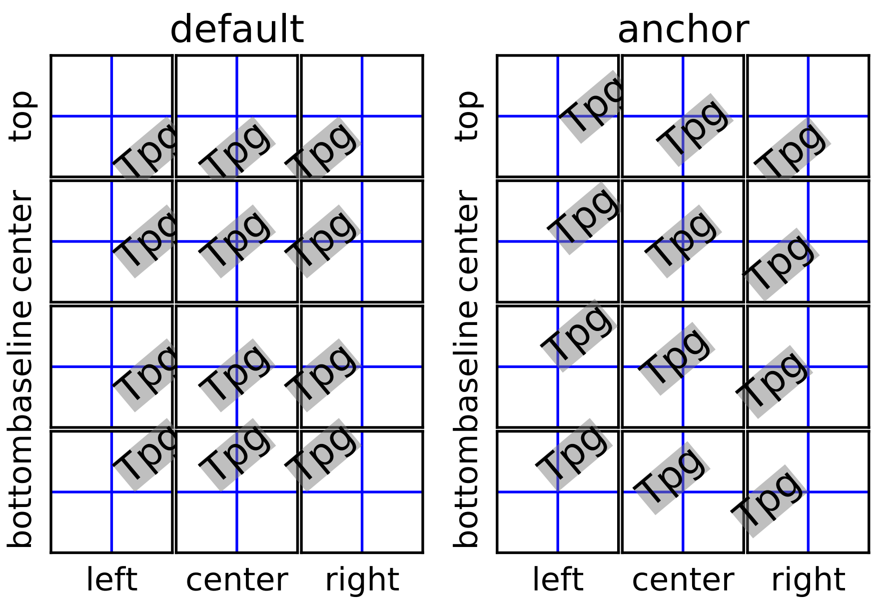
```
from matplotlib.image import BboxImage
import numpy as np
from matplotlib.transforms import IdentityTransform

import matplotlib.patches as mpatches

from matplotlib.offsetbox import AnnotationBbox,\
    AnchoredOffsetbox, AuxTransformBox

from matplotlib.cbook import get_sample_data

from matplotlib.text import TextPath


class PathClippedImagePatch(mpatches.PathPatch):
    """
    The given image is used to draw the face of the patch. Internally,
    it uses BboxImage whose clippath set to the path of the patch.

    FIXME : The result is currently dpi dependent.
    """

    def __init__(self, path, bbox_image, **kwargs):
        mpatches.PathPatch.__init__(self, path, **kwargs)
        self._init_bbox_image(bbox_image)

    def set_facecolor(self, color):
        """simply ignore facecolor"""
        mpatches.PathPatch.set_facecolor(self, "none")

    def _init_bbox_image(self, im):

        bbox_image = BboxImage(self.get_window_extent,
                               norm=None,
                               origin=None,
                               )
        bbox_image.set_transform(IdentityTransform())

        bbox_image.set_data(im)
        self.bbox_image = bbox_image

    def draw(self, renderer=None):

        # the clip path must be updated every draw. any solution? -JJ
        self.bbox_image.set_clip_path(self._path, self.get_transform())
        self.bbox_image.draw(renderer)

        mpatches.PathPatch.draw(self, renderer)


if 1:

    usetex = plt.rcParams["text.usetex"]
```

```
fig = plt.figure(1)

# EXAMPLE 1

ax = plt.subplot(211)

from matplotlib._png import read_png
fn = get_sample_data("grace_hopper.png", asfileobj=False)
arr = read_png(fn)

text_path = TextPath((0, 0), "!?", size=150)
p = PathClippedImagePatch(text_path, arr, ec="k",
                          transform=IdentityTransform())

#p.set_clip_on(False)

# make offset box
offsetbox = AuxTransformBox(IdentityTransform())
offsetbox.add_artist(p)

# make anchored offset box
ao = AnchoredOffsetbox(loc=2, child=offsetbox, frameon=True, borderpad=0.2)
ax.add_artist(ao)

# another text
from matplotlib.patches import PathPatch
if usetex:
    r = r"\mbox{textpath supports mathtext \& \TeX}"
else:
    r = r"textpath supports mathtext & TeX"

text_path = TextPath((0, 0), r,
                     size=20, usetex=usetex)

p1 = PathPatch(text_path, ec="w", lw=3, fc="w", alpha=0.9,
               transform=IdentityTransform())
p2 = PathPatch(text_path, ec="none", fc="k",
               transform=IdentityTransform())

offsetbox2 = AuxTransformBox(IdentityTransform())
offsetbox2.add_artist(p1)
offsetbox2.add_artist(p2)

ab = AnnotationBbox(offsetbox2, (0.95, 0.05),
                    xycoords='axes fraction',
                    boxcoords="offset points",
                    box_alignment=(1., 0.),
                    frameon=False
                    )
ax.add_artist(ab)

ax.imshow([[0, 1, 2], [1, 2, 3]], cmap=plt.cm.gist_gray_r,
```

```
            interpolation="bilinear",
            aspect="auto")

    # EXAMPLE 2

    ax = plt.subplot(212)

    arr = np.arange(256).reshape(1, 256)/256.

    if usetex:
        s = r"$\displaystyle\left[\sum_{n=1}^\infty\frac{-e^{i\pi}}{2^n}\right]$!"
    else:
        s = r"$\left[\sum_{n=1}^\infty\frac{-e^{i\pi}}{2^n}\right]$!"
    text_path = TextPath((0, 0), s, size=40, usetex=usetex)
    text_patch = PathClippedImagePatch(text_path, arr, ec="none",
                                       transform=IdentityTransform())

    shadow1 = mpatches.Shadow(text_patch, 1, -1, props=dict(fc="none", ec="0.6", lw=3))
    shadow2 = mpatches.Shadow(text_patch, 1, -1, props=dict(fc="0.3", ec="none"))

    # make offset box
    offsetbox = AuxTransformBox(IdentityTransform())
    offsetbox.add_artist(shadow1)
    offsetbox.add_artist(shadow2)
    offsetbox.add_artist(text_patch)

    # place the anchored offset box using AnnotationBbox
    ab = AnnotationBbox(offsetbox, (0.5, 0.5),
                        xycoords='data',
                        boxcoords="offset points",
                        box_alignment=(0.5, 0.5),
                        )
    #text_path.set_size(10)

    ax.add_artist(ab)

    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)

    plt.draw()
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.63 pylab_examples example code: demo_text_rotation_mode.py

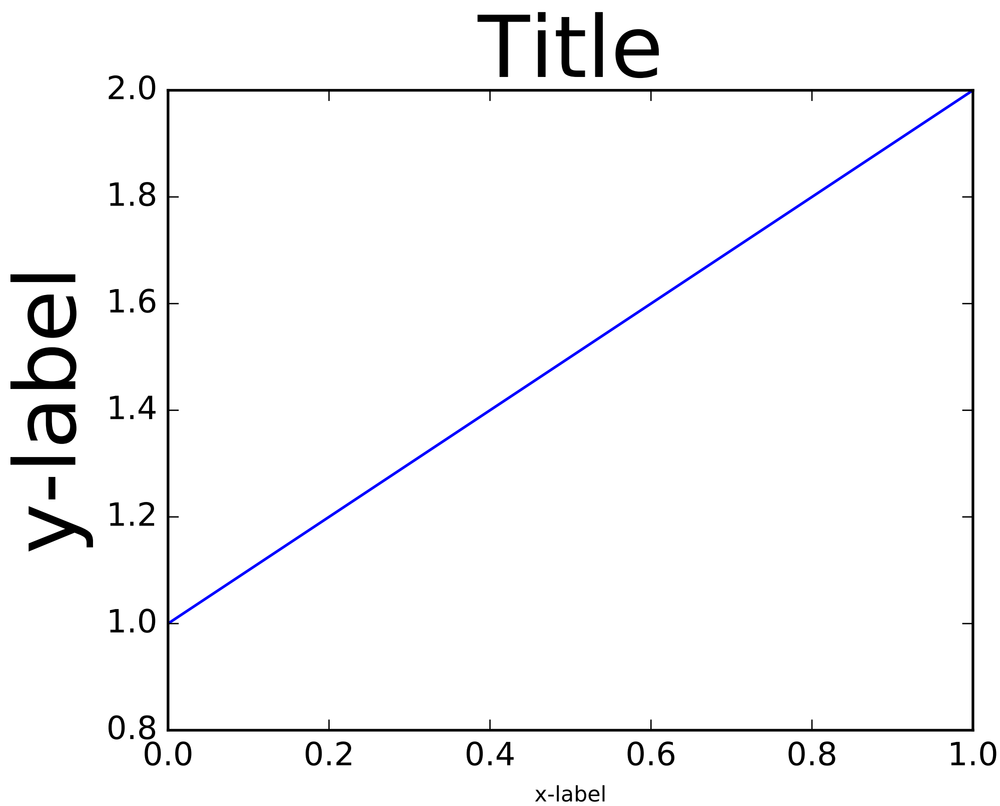


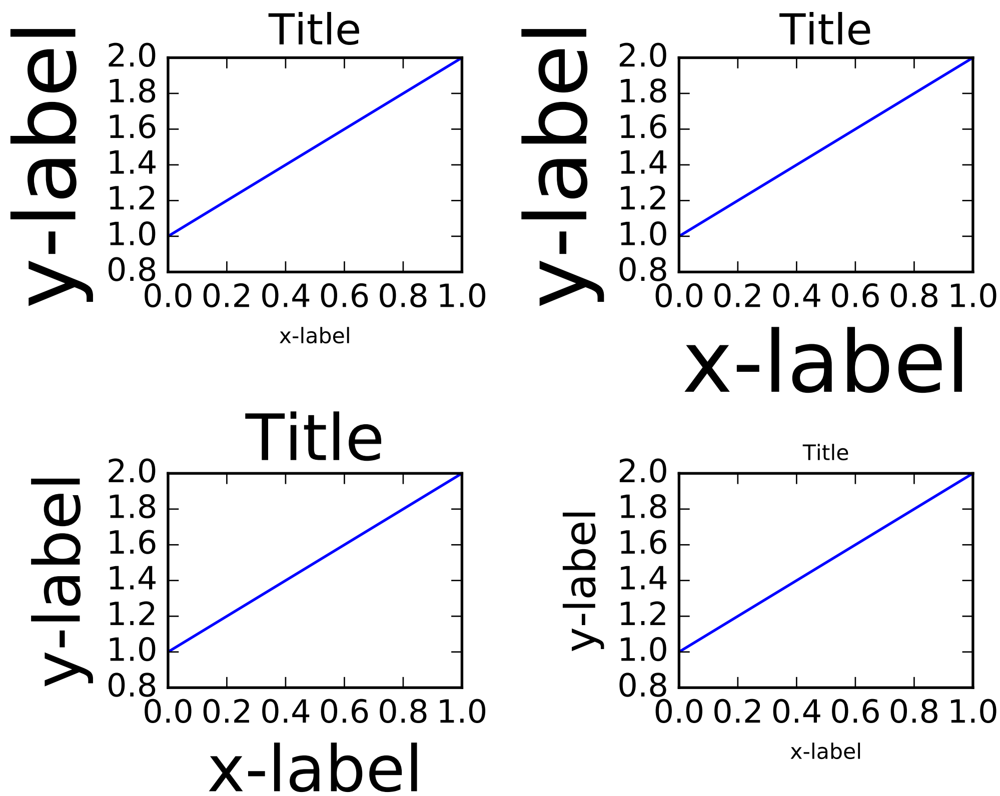

```
from mpl_toolkits.axes_grid1.axes_grid import ImageGrid


def test_rotation_mode(fig, mode, subplot_location):
    ha_list = "left center right".split()
    va_list = "top center baseline bottom".split()
    grid = ImageGrid(fig, subplot_location,
                     nrows_ncols=(len(va_list), len(ha_list)),
                     share_all=True, aspect=True,
                     #label_mode='1',
                     cbar_mode=None)

    for ha, ax in zip(ha_list, grid.axes_row[-1]):
        ax.axis["bottom"].label.set_text(ha)

    grid.axes_row[0][1].set_title(mode, size="large")

    for va, ax in zip(va_list, grid.axes_column[0]):
        ax.axis["left"].label.set_text(va)

    i = 0
    for va in va_list:
        for ha in ha_list:
            ax = grid[i]
            for axis in ax.axis.values():
```

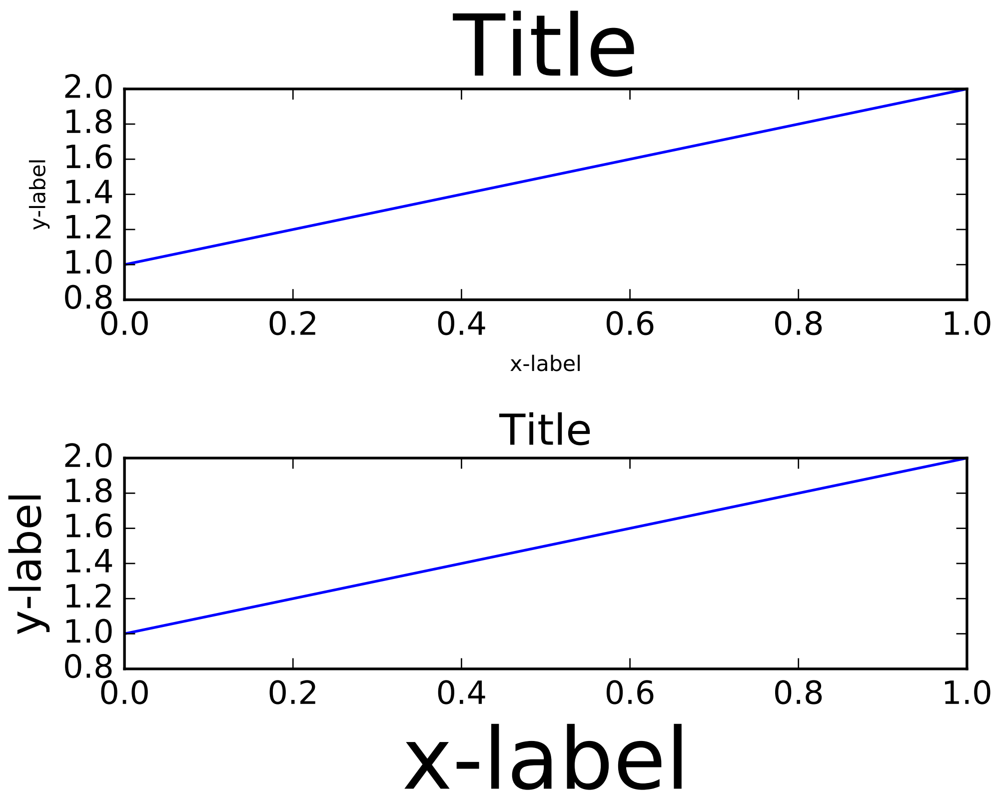
```
                axis.toggle(ticks=False, ticklabels=False)

            ax.text(0.5, 0.5, "Tpg",
                    size="large", rotation=40,
                    bbox=dict(boxstyle="square,pad=0.",
                              ec="none", fc="0.5", alpha=0.5),
                    ha=ha, va=va,
                    rotation_mode=mode)
            ax.axvline(0.5)
            ax.axhline(0.5)
            i += 1

if 1:
    import matplotlib.pyplot as plt
    fig = plt.figure(1, figsize=(5.5, 4))
    fig.clf()

    test_rotation_mode(fig, "default", 121)
    test_rotation_mode(fig, "anchor", 122)
    plt.show()
```

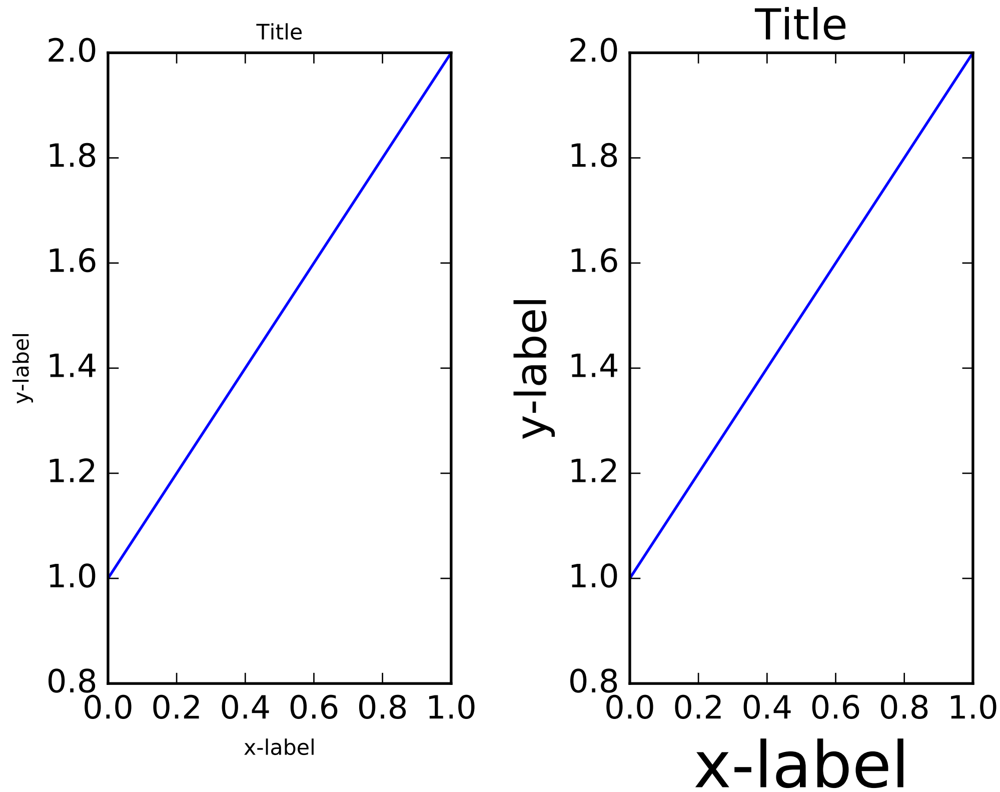
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

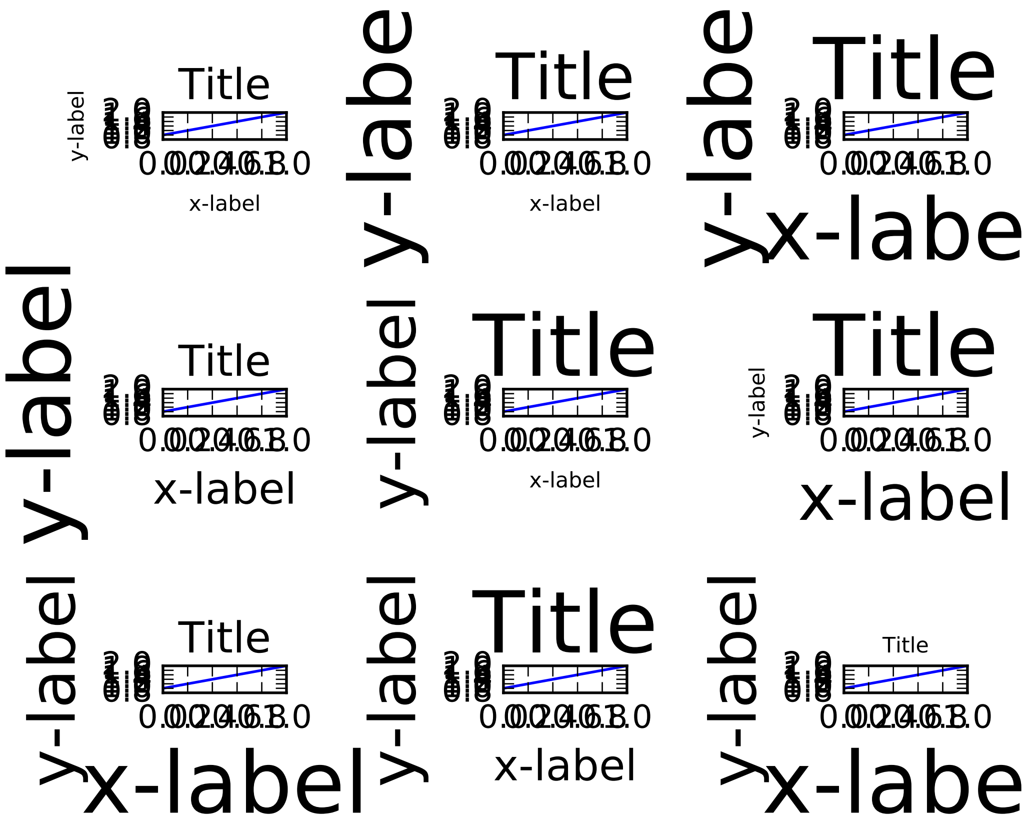
90.64 pylab_examples example code: demo_tight_layout.py

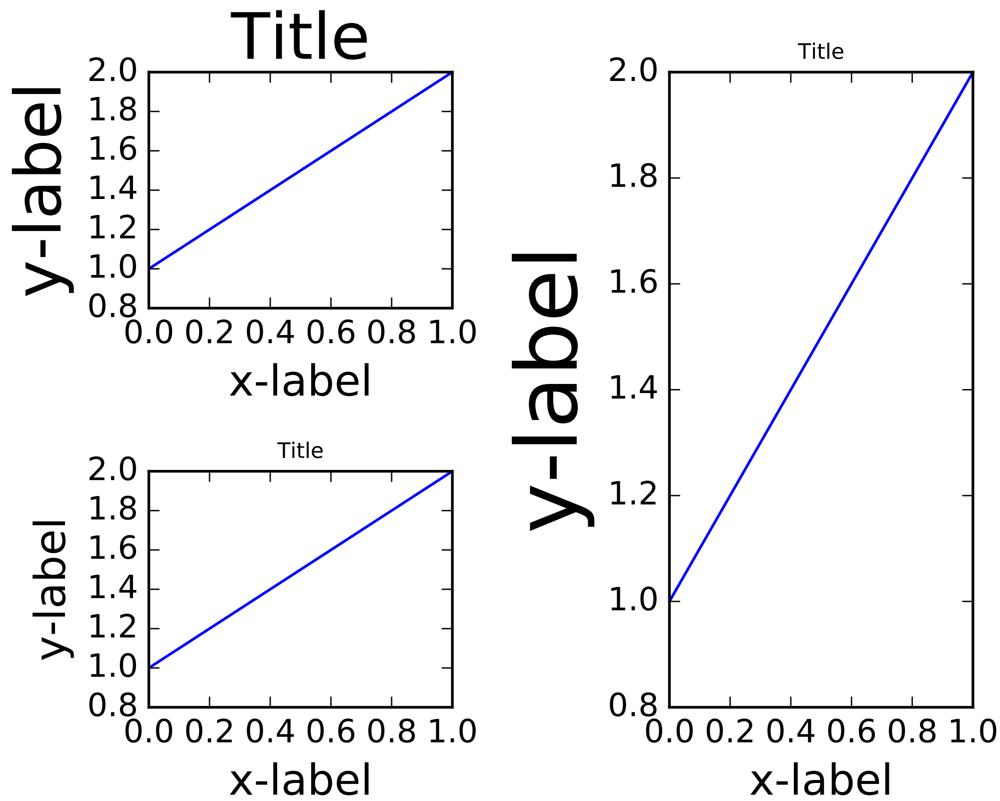



Title
y-label
x-label
Title
y-label
x-label
Title
y-label
x-label
Title
y-label
x-label
0.0 0.2 0.4 0.6 0.8 1.0
0.8 1.0 1.2 1.4 1.6 1.8 2.0

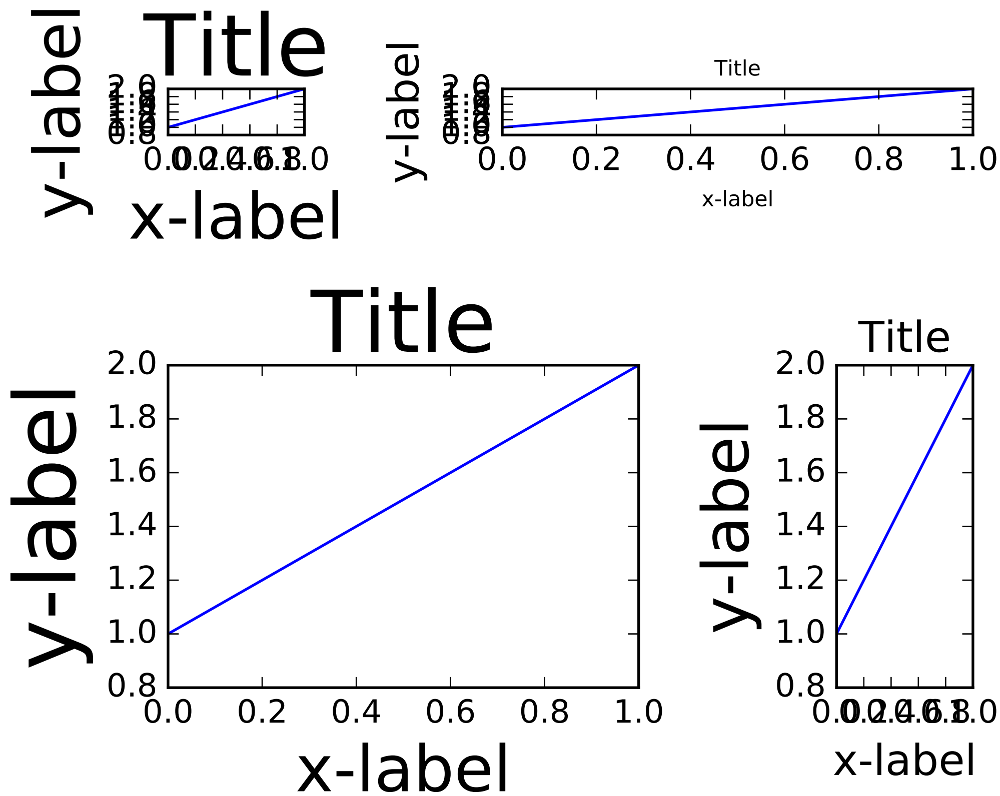

Title
2.0
1.8
1.6
1.4
1.2
1.0
0.8
y-label
0.0
0.2
0.4
0.6
0.8
1.0
x-label
Title
2.0
1.8
1.6
1.4
1.2
1.0
0.8
y-label
0.0
0.2
0.4
0.6
0.8
1.0
x-label

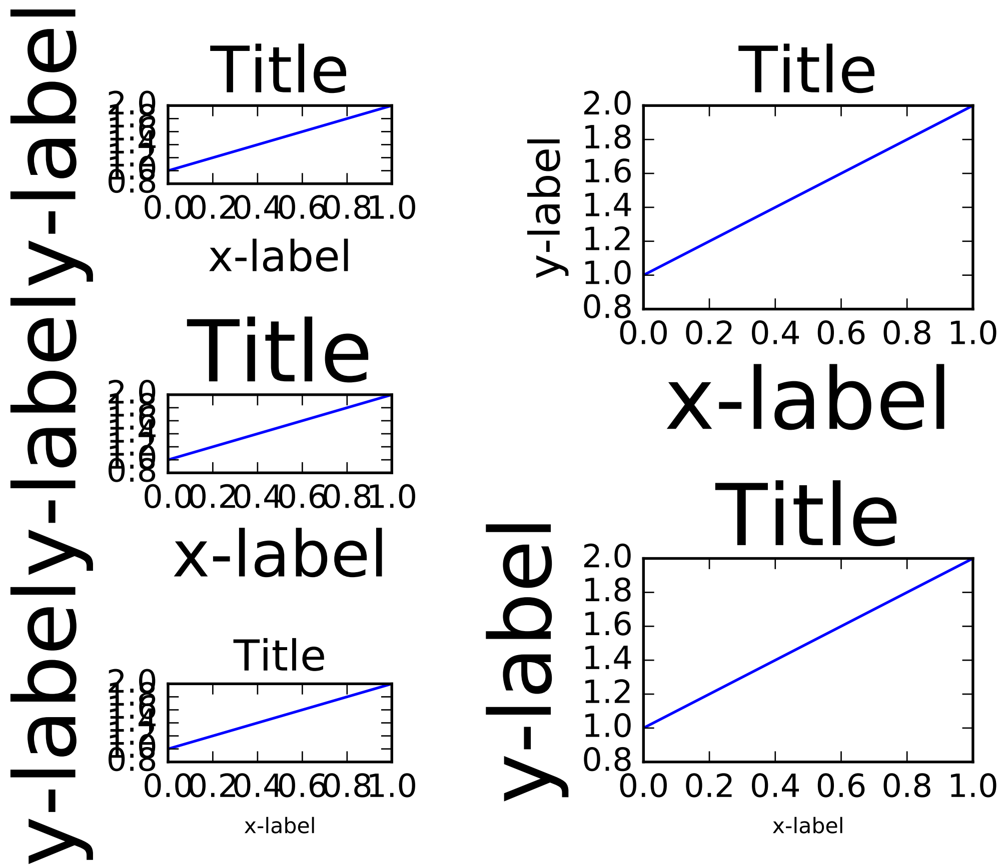

Title
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.0 0.2 0.4 0.6 0.8 1.0
x-label
y-label
Title
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.0 0.2 0.4 0.6 0.8 1.0
x-label
y-label


Title
y-label
x-label
y-labe
x-labe



Title
y-label
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.0 0.2 0.4 0.6 0.8 1.0
x-label
Title
y-label
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.0 0.2 0.4 0.6 0.8 1.0
x-label
Title
y-label
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.0 0.2 0.4 0.6 0.8 1.0
x-label


Title
y-label
x-label
Title
y-label
0.0
0.2
0.4
0.6
0.8
1.0
x-label
Title
2.0
1.8
1.6
1.4
1.2
1.0
0.8
y-label
0.0
0.2
0.4
0.6
0.8
1.0
x-label
Title
2.0
1.8
1.6
1.4
1.2
1.0
0.8
y-label
x-label



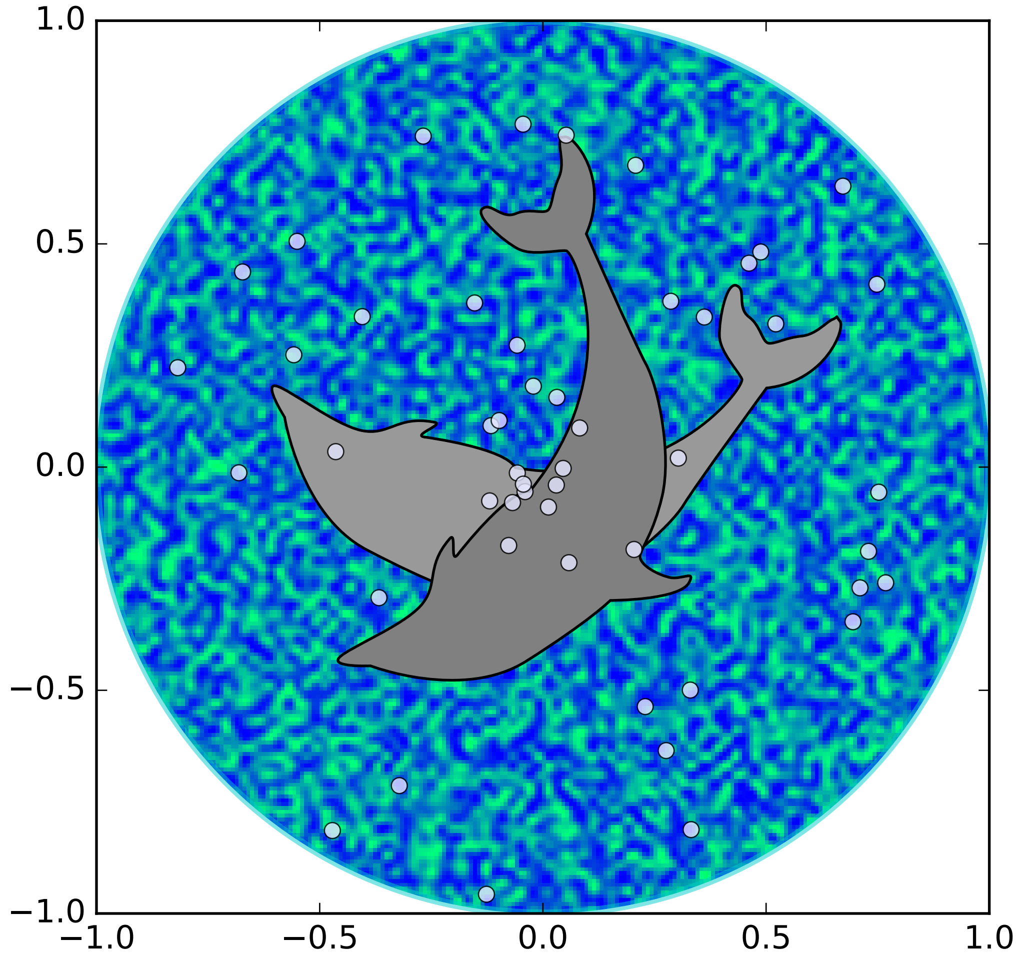

```
import matplotlib.pyplot as plt
import warnings

import random
fontsizes = [8, 16, 24, 32]


def example_plot(ax):
    ax.plot([1, 2])
    ax.set_xlabel('x-label', fontsize=random.choice(fontsizes))
    ax.set_ylabel('y-label', fontsize=random.choice(fontsizes))
    ax.set_title('Title', fontsize=random.choice(fontsizes))

fig, ax = plt.subplots()
example_plot(ax)
plt.tight_layout()

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2)
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)
plt.tight_layout()

fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1)
example_plot(ax1)
```

```
example_plot(ax2)
plt.tight_layout()

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)
example_plot(ax1)
example_plot(ax2)
plt.tight_layout()

fig, axes = plt.subplots(nrows=3, ncols=3)
for row in axes:
    for ax in row:
        example_plot(ax)
plt.tight_layout()


fig = plt.figure()

ax1 = plt.subplot(221)
ax2 = plt.subplot(223)
ax3 = plt.subplot(122)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)

plt.tight_layout()


fig = plt.figure()

ax1 = plt.subplot2grid((3, 3), (0, 0))
ax2 = plt.subplot2grid((3, 3), (0, 1), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 0), colspan=2, rowspan=2)
ax4 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)

example_plot(ax1)
example_plot(ax2)
example_plot(ax3)
example_plot(ax4)

plt.tight_layout()

plt.show()


fig = plt.figure()

import matplotlib.gridspec as gridspec

gs1 = gridspec.GridSpec(3, 1)
ax1 = fig.add_subplot(gs1[0])
ax2 = fig.add_subplot(gs1[1])
ax3 = fig.add_subplot(gs1[2])
```

```
example_plot(ax1)
example_plot(ax2)
example_plot(ax3)

with warnings.catch_warnings():
    warnings.simplefilter("ignore", UserWarning)
    # This raises warnings since tight layout cannot
    # handle gridspec automatically. We are going to
    # do that manually so we can filter the warning.
    gs1.tight_layout(fig, rect=[None, None, 0.45, None])

gs2 = gridspec.GridSpec(2, 1)
ax4 = fig.add_subplot(gs2[0])
ax5 = fig.add_subplot(gs2[1])

example_plot(ax4)
example_plot(ax5)

with warnings.catch_warnings():
    # This raises warnings since tight layout cannot
    # handle gridspec automatically. We are going to
    # do that manually so we can filter the warning.
    warnings.simplefilter("ignore", UserWarning)
    gs2.tight_layout(fig, rect=[0.45, None, None, None])

# now match the top and bottom of two gridspecs.
top = min(gs1.top, gs2.top)
bottom = max(gs1.bottom, gs2.bottom)

gs1.update(top=top, bottom=bottom)
gs2.update(top=top, bottom=bottom)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.65 pylab_examples example code: dolphin.py



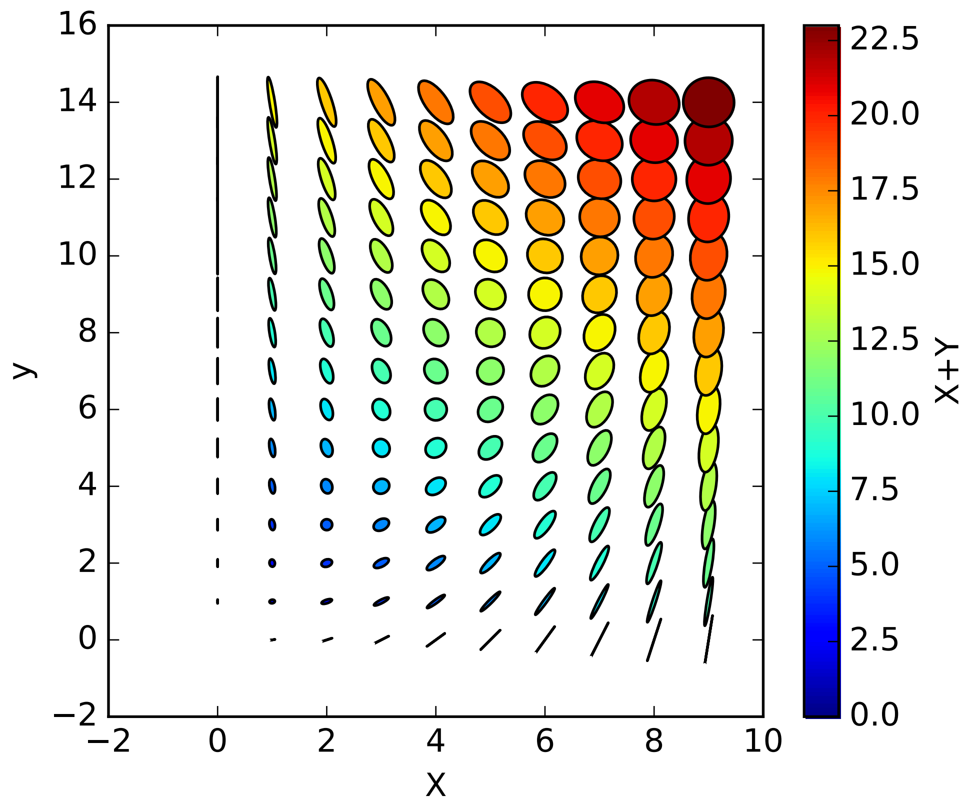

```
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, PathPatch
from matplotlib.path import Path
from matplotlib.transforms import Affine2D
import numpy as np


r = np.random.rand(50)
t = np.random.rand(50) * np.pi * 2.0
x = r * np.cos(t)
y = r * np.sin(t)
```

```
fig, ax = plt.subplots(figsize=(6, 6))
circle = Circle((0, 0), 1, facecolor='none',
                edgecolor=(0, 0.8, 0.8), linewidth=3, alpha=0.5)
ax.add_patch(circle)

im = plt.imshow(np.random.random((100, 100)),
                origin='lower', cmap=cm.winter,
                interpolation='spline36',
                extent=([-1, 1, -1, 1]))
im.set_clip_path(circle)

plt.plot(x, y, 'o', color=(0.9, 0.9, 1.0), alpha=0.8)

# Dolphin from OpenClipart library by Andy Fitzsimon
#       <cc:License rdf:about="http://web.resource.org/cc/PublicDomain">
#         <cc:permits rdf:resource="http://web.resource.org/cc/Reproduction"/>
#         <cc:permits rdf:resource="http://web.resource.org/cc/Distribution"/>
#         <cc:permits rdf:resource="http://web.resource.org/cc/DerivativeWorks"/>
#       </cc:License>

dolphin = """
M -0.59739425,160.18173 C -0.62740401,160.18885 -0.57867129,160.11183
-0.57867129,160.11183 C -0.57867129,160.11183 -0.5438361,159.89315
-0.39514638,159.81496 C -0.24645668,159.73678 -0.18316813,159.71981
-0.18316813,159.71981 C -0.18316813,159.71981 -0.10322971,159.58124
-0.057804323,159.58725 C -0.029723983,159.58913 -0.061841603,159.60356
-0.071265813,159.62815 C -0.080250183,159.65325 -0.082918513,159.70554
-0.061841203,159.71248 C -0.040763903,159.7194 -0.0066711426,159.71091
0.077336307,159.73612 C 0.16879567,159.76377 0.28380306,159.86448
0.31516668,159.91533 C 0.3465303,159.96618 0.5011127,160.1771
0.5011127,160.1771 C 0.63668998,160.19238 0.67763022,160.31259
0.66556395,160.32668 C 0.65339985,160.34212 0.66350443,160.33642
0.64907098,160.33088 C 0.63463742,160.32533 0.61309688,160.297
0.5789627,160.29339 C 0.54348657,160.28968 0.52329693,160.27674
0.50728856,160.27737 C 0.49060916,160.27795 0.48965803,160.31565
0.46114204,160.33673 C 0.43329696,160.35786 0.4570711,160.39871
0.43309565,160.40685 C 0.4105108,160.41442 0.39416631,160.33027
0.3954995,160.2935 C 0.39683269,160.25672 0.43807996,160.21522
0.44567915,160.19734 C 0.45327833,160.17946 0.27946869,159.9424
-0.061852613,159.99845 C -0.083965233,160.0427 -0.26176109,160.06683
-0.26176109,160.06683 C -0.30127962,160.07028 -0.21167141,160.09731
-0.24649368,160.1011 C -0.32642366,160.11569 -0.34521187,160.06895
-0.40622293,160.0819 C -0.467234,160.09485 -0.56738444,160.17461
-0.59739425,160.18173
"""

vertices = []
codes = []
parts = dolphin.split()
i = 0
code_map = {
    'M': (Path.MOVETO, 1),
    'C': (Path.CURVE4, 3),
```

```
    'L': (Path.LINETO, 1)
    }

while i < len(parts):
    code = parts[i]
    path_code, npoints = code_map[code]
    codes.extend([path_code] * npoints)
    vertices.extend([[float(x) for x in y.split(',')] for y in
                     parts[i + 1:i + npoints + 1]])
    i += npoints + 1
vertices = np.array(vertices, np.float)
vertices[:, 1] -= 160

dolphin_path = Path(vertices, codes)
dolphin_patch = PathPatch(dolphin_path, facecolor=(0.6, 0.6, 0.6),
                          edgecolor=(0.0, 0.0, 0.0))
ax.add_patch(dolphin_patch)

vertices = Affine2D().rotate_deg(60).transform(vertices)
dolphin_path2 = Path(vertices, codes)
dolphin_patch2 = PathPatch(dolphin_path2, facecolor=(0.5, 0.5, 0.5),
                           edgecolor=(0.0, 0.0, 0.0))
ax.add_patch(dolphin_patch2)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.66 pylab_examples example code: ellipse_collection.py

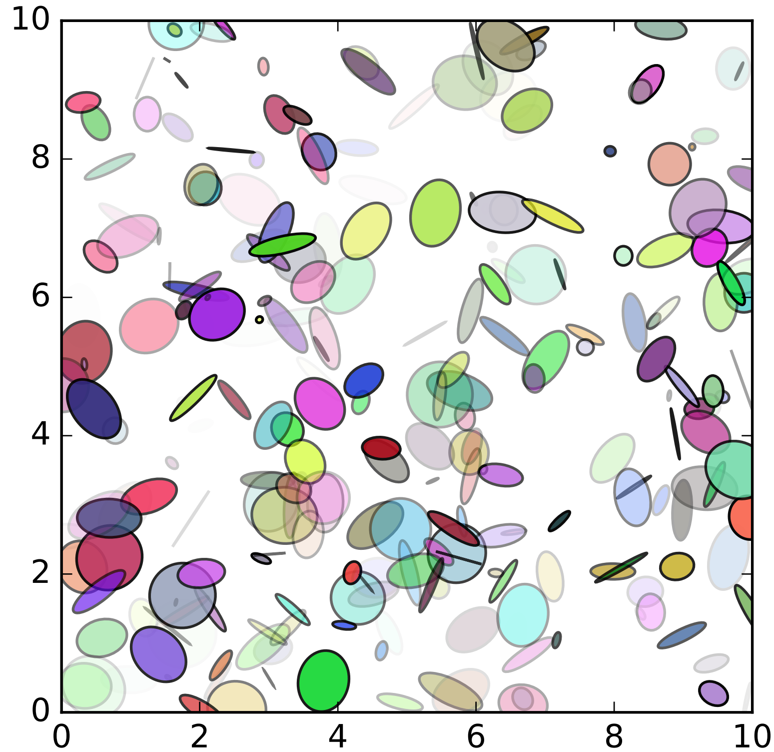



```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.collections import EllipseCollection

x = np.arange(10)
y = np.arange(15)
X, Y = np.meshgrid(x, y)

XY = np.hstack((X.ravel()[:, np.newaxis], Y.ravel()[:, np.newaxis]))

ww = X/10.0
hh = Y/15.0
aa = X*9


fig, ax = plt.subplots()

ec = EllipseCollection(ww, hh, aa, units='x', offsets=XY,
                       transOffset=ax.transData)
ec.set_array((X + Y).ravel())
ax.add_collection(ec)
ax.autoscale_view()
```

```
ax.set_xlabel('X')
ax.set_ylabel('y')
cbar = plt.colorbar(ec)
cbar.set_label('X+Y')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.67 pylab_examples example code: ellipse_demo.py

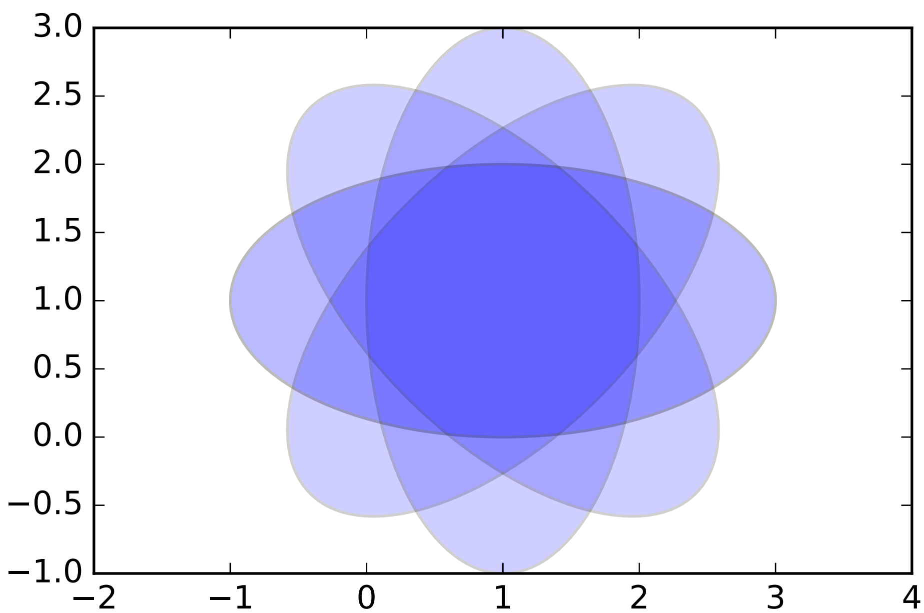



```
import matplotlib.pyplot as plt
import numpy.random as rnd
from matplotlib.patches import Ellipse

NUM = 250

ells = [Ellipse(xy=rnd.rand(2)*10, width=rnd.rand(), height=rnd.rand(), angle=rnd.
↪rand()*360)
        for i in range(NUM)]

fig = plt.figure(0)
ax = fig.add_subplot(111, aspect='equal')
```

```
for e in ells:
    ax.add_artist(e)
    e.set_clip_box(ax.bbox)
    e.set_alpha(rnd.rand())
    e.set_facecolor(rnd.rand(3))

ax.set_xlim(0, 10)
ax.set_ylim(0, 10)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.68 pylab_examples example code: ellipse_rotated.py

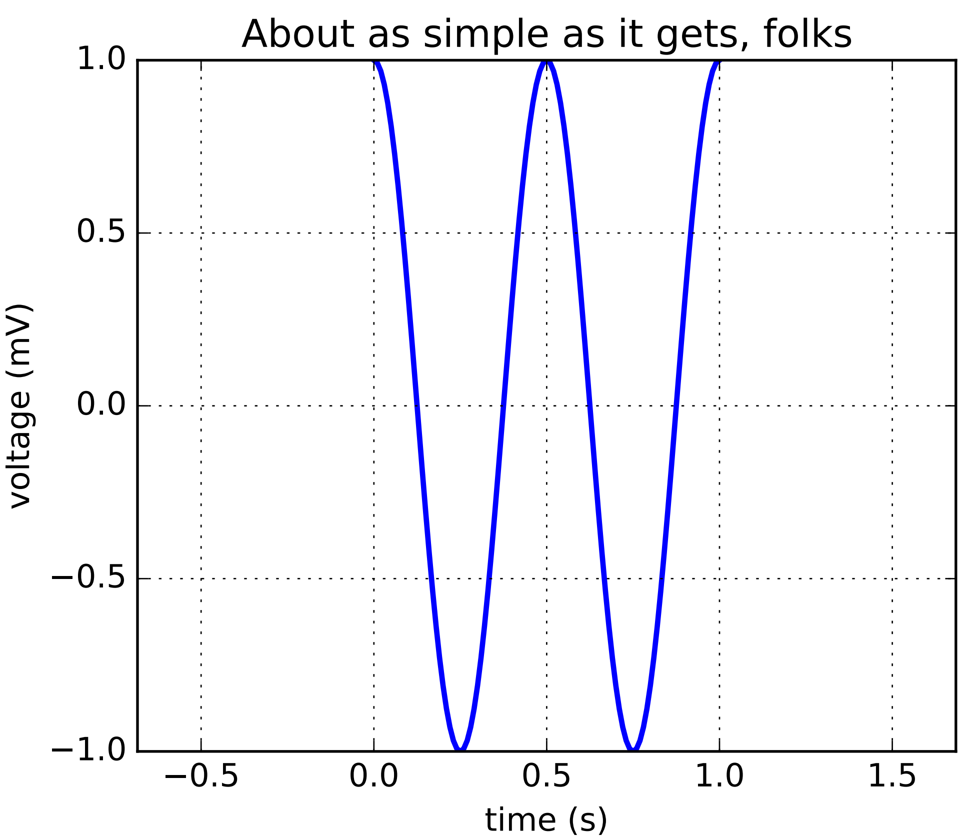



```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Ellipse

delta = 45.0  # degrees

angles = np.arange(0, 360 + delta, delta)
```

```
ells = [Ellipse((1, 1), 4, 2, a) for a in angles]

a = plt.subplot(111, aspect='equal')

for e in ells:
    e.set_clip_box(a.bbox)
    e.set_alpha(0.1)
    a.add_artist(e)

plt.xlim(-2, 4)
plt.ylim(-1, 3)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.69 pylab_examples example code: equal_aspect_ratio.py

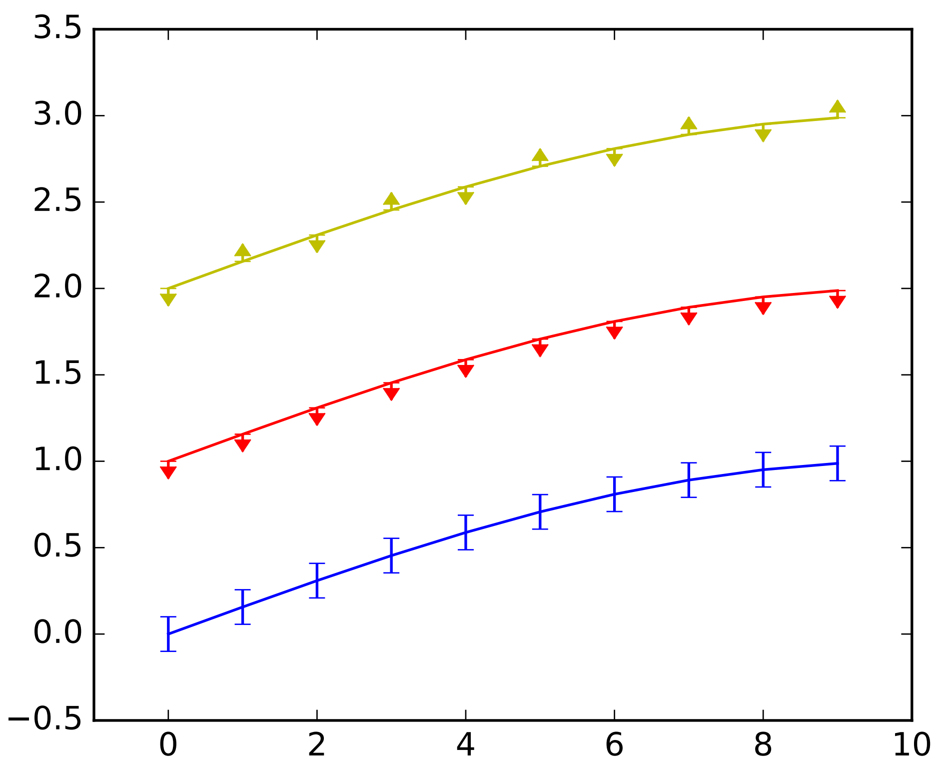


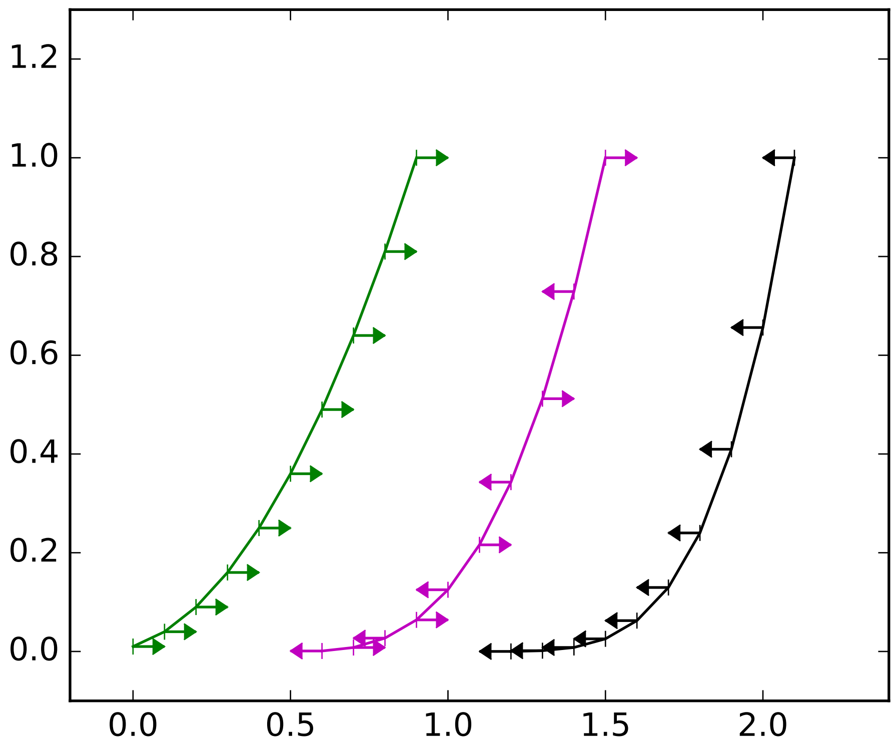

```
#!/usr/bin/env python
"""
Example: simple line plot.
Show how to make a plot that has equal aspect ratio
```

```
"""
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(2*2*np.pi*t)
plt.plot(t, s, '-', lw=2)

plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('About as simple as it gets, folks')
plt.grid(True)

plt.axes().set_aspect('equal', 'datalim')


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.70 pylab_examples example code: errorbar_limits.py



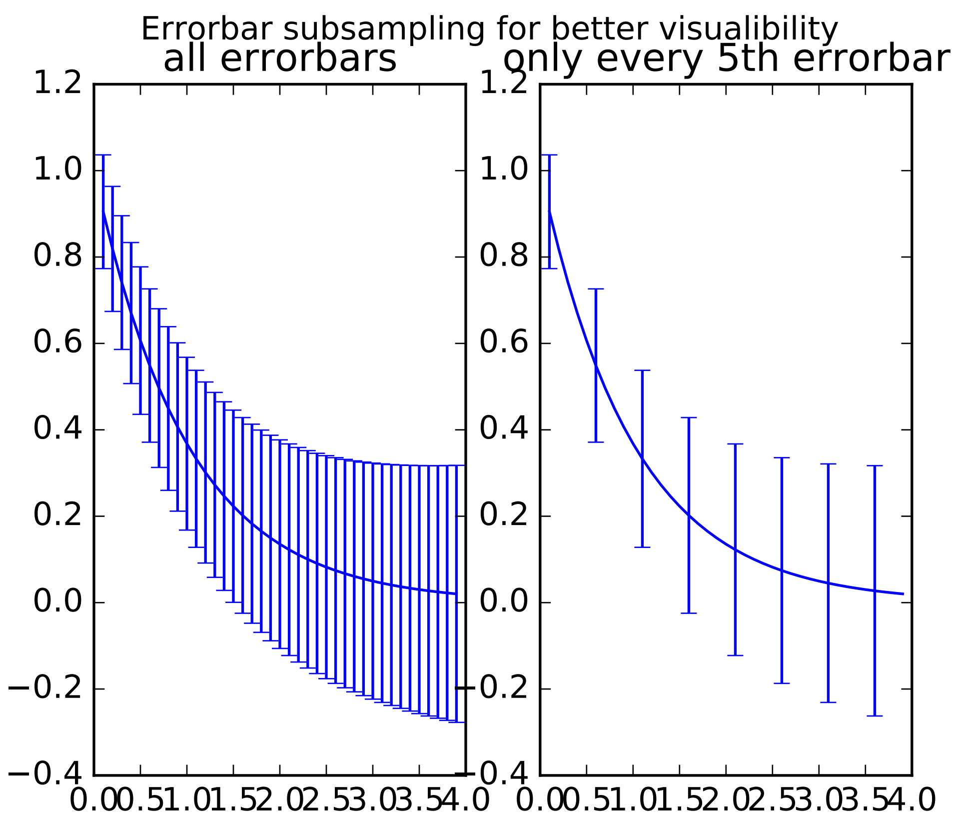



```
"""
Illustration of upper and lower limit symbols on errorbars
"""

import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(0)
x = np.arange(10.0)
y = np.sin(np.arange(10.0)/20.0*np.pi)

plt.errorbar(x, y, yerr=0.1, capsize=3)

y = np.sin(np.arange(10.0)/20.0*np.pi) + 1
plt.errorbar(x, y, yerr=0.1, uplims=True)

y = np.sin(np.arange(10.0)/20.0*np.pi) + 2
upperlimits = np.array([1, 0]*5)
lowerlimits = np.array([0, 1]*5)
plt.errorbar(x, y, yerr=0.1, uplims=upperlimits, lolims=lowerlimits)

plt.xlim(-1, 10)

fig = plt.figure(1)
x = np.arange(10.0)/10.0
y = (x + 0.1)**2
```

```
plt.errorbar(x, y, xerr=0.1, xlolims=True)
y = (x + 0.1)**3

plt.errorbar(x + 0.6, y, xerr=0.1, xuplims=upperlimits, xlolims=lowerlimits)

y = (x + 0.1)**4
plt.errorbar(x + 1.2, y, xerr=0.1, xuplims=True)

plt.xlim(-0.2, 2.4)
plt.ylim(-0.1, 1.3)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.71 pylab_examples example code: errorbar_subsample.py

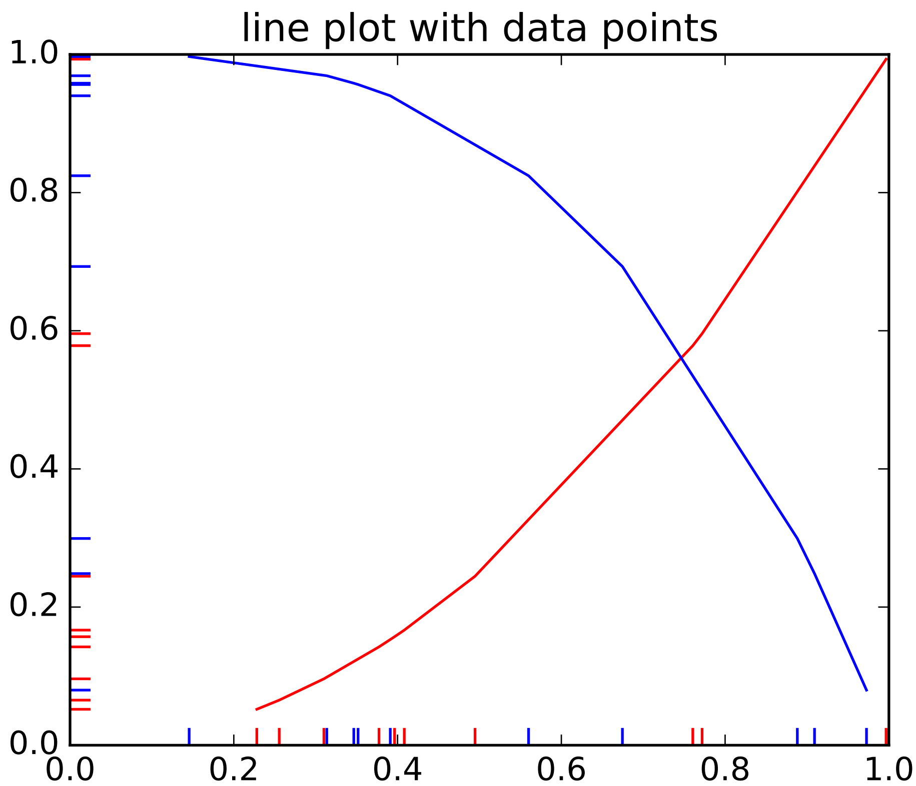



```
'''
Demo for the errorevery keyword to show data full accuracy data plots with
few errorbars.
'''
```

```
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.1)
y = np.exp(-x)

# example variable error bar values
yerr = 0.1 + 0.1*np.sqrt(x)


# Now switch to a more OO interface to exercise more features.
fig, axs = plt.subplots(nrows=1, ncols=2, sharex=True)
ax = axs[0]
ax.errorbar(x, y, yerr=yerr)
ax.set_title('all errorbars')

ax = axs[1]
ax.errorbar(x, y, yerr=yerr, errorevery=5)
ax.set_title('only every 5th errorbar')


fig.suptitle('Errorbar subsampling for better visualibility')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.72 pylab_examples example code: eventcollection_demo.py

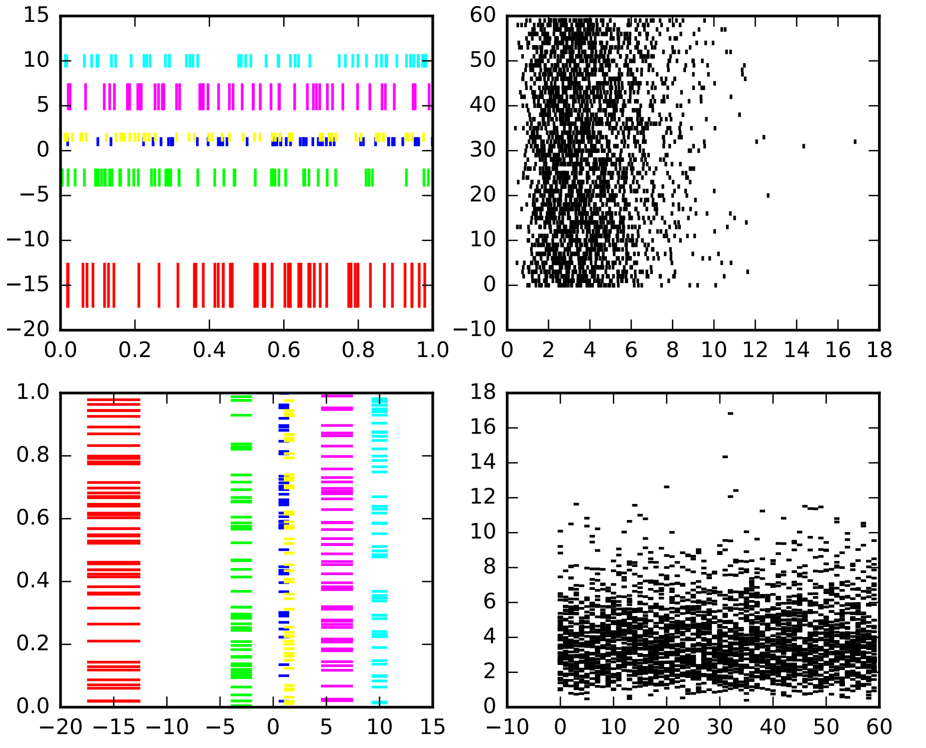



```
#!/usr/bin/env python
# -*- Coding:utf-8 -*-
'''Plot two curves, then use EventCollections to mark the locations of the x
and y data points on the respective axes for each curve'''

import matplotlib.pyplot as plt
from matplotlib.collections import EventCollection
import numpy as np

# create random data
np.random.seed(50)
xdata = np.random.random([2, 10])

# split the data into two parts
xdata1 = xdata[0, :]
xdata2 = xdata[1, :]

# sort the data so it makes clean curves
xdata1.sort()
xdata2.sort()

# create some y data points
```

```
ydata1 = xdata1 ** 2
ydata2 = 1 - xdata2 ** 3

# plot the data
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(xdata1, ydata1, 'r', xdata2, ydata2, 'b')

# create the events marking the x data points
xevents1 = EventCollection(xdata1, color=[1, 0, 0], linelength=0.05)
xevents2 = EventCollection(xdata2, color=[0, 0, 1], linelength=0.05)

# create the events marking the y data points
yevents1 = EventCollection(ydata1, color=[1, 0, 0], linelength=0.05,
                           orientation='vertical')
yevents2 = EventCollection(ydata2, color=[0, 0, 1], linelength=0.05,
                           orientation='vertical')

# add the events to the axis
ax.add_collection(xevents1)
ax.add_collection(xevents2)
ax.add_collection(yevents1)
ax.add_collection(yevents2)

# set the limits
ax.set_xlim([0, 1])
ax.set_ylim([0, 1])

ax.set_title('line plot with data points')

# display the plot
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.73 pylab_examples example code: eventplot_demo.py

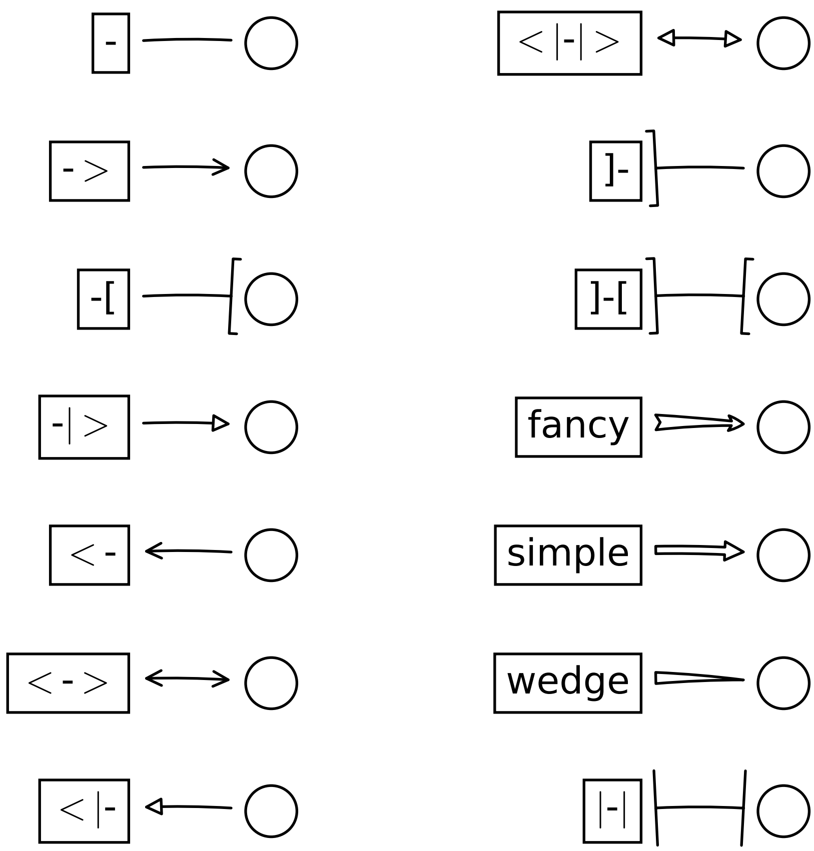



```
#!/usr/bin/env python
# -*- Coding:utf-8 -*-
'''an eventplot showing sequences of events with various line properties
the plot is shown in both horizontal and vertical orientations'''

import matplotlib.pyplot as plt
import numpy as np
import matplotlib
matplotlib.rcParams['font.size'] = 8.0

# set the random seed
np.random.seed(0)

# create random data
data1 = np.random.random([6, 50])

# set different colors for each set of positions
colors1 = np.array([[1, 0, 0],
                    [0, 1, 0],
                    [0, 0, 1],
                    [1, 1, 0],
                    [1, 0, 1],
```

```
                    [0, 1, 1]])

# set different line properties for each set of positions
# note that some overlap
lineoffsets1 = np.array([-15, -3, 1, 1.5, 6, 10])
linelengths1 = [5, 2, 1, 1, 3, 1.5]

fig = plt.figure()

# create a horizontal plot
ax1 = fig.add_subplot(221)
ax1.eventplot(data1, colors=colors1, lineoffsets=lineoffsets1,
              linelengths=linelengths1)


# create a vertical plot
ax2 = fig.add_subplot(223)
ax2.eventplot(data1, colors=colors1, lineoffsets=lineoffsets1,
              linelengths=linelengths1, orientation='vertical')

# create another set of random data.
# the gamma distribution is only used fo aesthetic purposes
data2 = np.random.gamma(4, size=[60, 50])

# use individual values for the parameters this time
# these values will be used for all data sets (except lineoffsets2, which
# sets the increment between each data set in this usage)
colors2 = [[0, 0, 0]]
lineoffsets2 = 1
linelengths2 = 1

# create a horizontal plot
ax1 = fig.add_subplot(222)
ax1.eventplot(data2, colors=colors2, lineoffsets=lineoffsets2,
              linelengths=linelengths2)


# create a vertical plot
ax2 = fig.add_subplot(224)
ax2.eventplot(data2, colors=colors2, lineoffsets=lineoffsets2,
              linelengths=linelengths2, orientation='vertical')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.74 pylab_examples example code: fancyarrow_demo.py

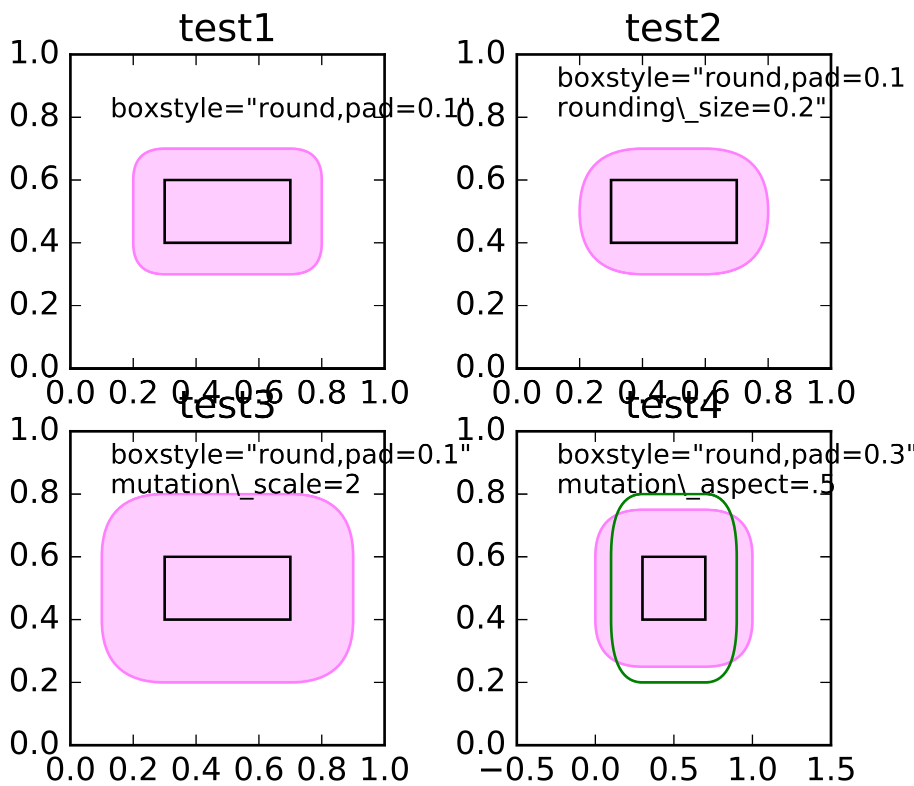



```
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt

styles = mpatches.ArrowStyle.get_styles()

ncol = 2
nrow = (len(styles) + 1) // ncol
figheight = (nrow + 0.5)
fig1 = plt.figure(1, (4.*ncol/1.5, figheight/1.5))
fontsize = 0.2 * 70


ax = fig1.add_axes([0, 0, 1, 1], frameon=False, aspect=1.)

ax.set_xlim(0, 4*ncol)
ax.set_ylim(0, figheight)


def to_texstring(s):
```

```
    s = s.replace("<", r"$<$")
    s = s.replace(">", r"$>$")
    s = s.replace("|", r"$|$")
    return s

for i, (stylename, styleclass) in enumerate(sorted(styles.items())):
    x = 3.2 + (i//nrow)*4
    y = (figheight - 0.7 - i % nrow)  # /figheight
    p = mpatches.Circle((x, y), 0.2, fc="w")
    ax.add_patch(p)

    ax.annotate(to_texstring(stylename), (x, y),
                (x - 1.2, y),
                #xycoords="figure fraction", textcoords="figure fraction",
                ha="right", va="center",
                size=fontsize,
                arrowprops=dict(arrowstyle=stylename,
                                patchB=p,
                                shrinkA=5,
                                shrinkB=5,
                                fc="w", ec="k",
                                connectionstyle="arc3,rad=-0.05",
                                ),
                bbox=dict(boxstyle="square", fc="w"))

ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)


plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.75 pylab_examples example code: fancybox_demo.py




```
import matplotlib.pyplot as plt
import matplotlib.transforms as mtransforms
from matplotlib.patches import FancyBboxPatch


# Bbox object around which the fancy box will be drawn.
bb = mtransforms.Bbox([[0.3, 0.4], [0.7, 0.6]])


def draw_bbox(ax, bb):
    # boxstyle=square with pad=0, i.e. bbox itself.
    p_bbox = FancyBboxPatch((bb.xmin, bb.ymin),
                            abs(bb.width), abs(bb.height),
                            boxstyle="square,pad=0.",
                            ec="k", fc="none", zorder=10.,
                            )
    ax.add_patch(p_bbox)


def test1(ax):

    # a fancy box with round corners. pad=0.1
```

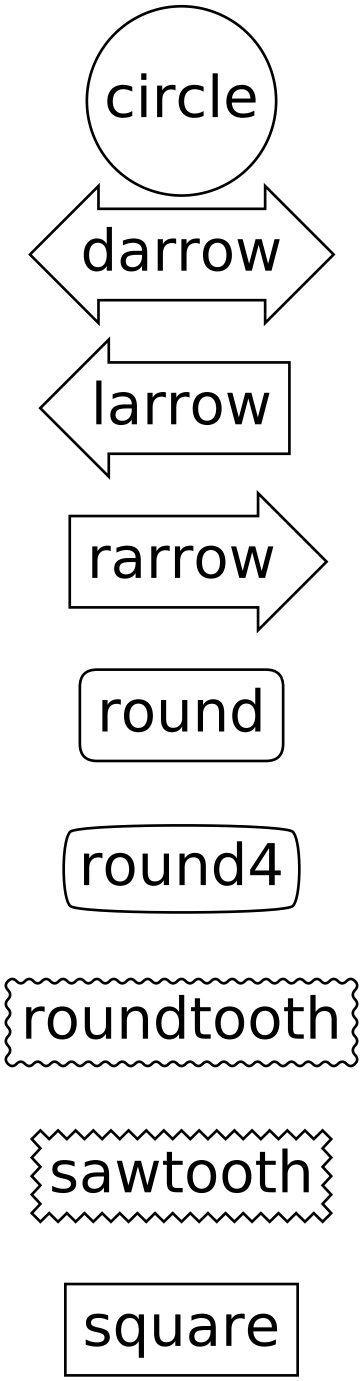
```
    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),
                             abs(bb.width), abs(bb.height),
                             boxstyle="round,pad=0.1",
                             fc=(1., .8, 1.),
                             ec=(1., 0.5, 1.))

    ax.add_patch(p_fancy)

    ax.text(0.1, 0.8,
            r' boxstyle="round,pad=0.1"',
            size=10, transform=ax.transAxes)

    # draws control points for the fancy box.
    #l = p_fancy.get_path().vertices
    #ax.plot(l[:,0], l[:,1], ".")

    # draw the original bbox in black
    draw_bbox(ax, bb)


def test2(ax):

    # bbox=round has two optional argument. pad and rounding_size.
    # They can be set during the initialization.
    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),
                             abs(bb.width), abs(bb.height),
                             boxstyle="round,pad=0.1",
                             fc=(1., .8, 1.),
                             ec=(1., 0.5, 1.))

    ax.add_patch(p_fancy)

    # boxstyle and its argument can be later modified with
    # set_boxstyle method. Note that the old attributes are simply
    # forgotten even if the boxstyle name is same.

    p_fancy.set_boxstyle("round,pad=0.1, rounding_size=0.2")
    # or
    #p_fancy.set_boxstyle("round", pad=0.1, rounding_size=0.2)

    ax.text(0.1, 0.8,
            ' boxstyle="round,pad=0.1\n rounding\\_size=0.2"',
            size=10, transform=ax.transAxes)

    # draws control points for the fancy box.
    #l = p_fancy.get_path().vertices
    #ax.plot(l[:,0], l[:,1], ".")

    draw_bbox(ax, bb)


def test3(ax):
```

```
    # mutation_scale determine overall scale of the mutation,
    # i.e. both pad and rounding_size is scaled according to this
    # value.
    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),
                             abs(bb.width), abs(bb.height),
                             boxstyle="round,pad=0.1",
                             mutation_scale=2.,
                             fc=(1., .8, 1.),
                             ec=(1., 0.5, 1.))

    ax.add_patch(p_fancy)

    ax.text(0.1, 0.8,
            ' boxstyle="round,pad=0.1"\n mutation\\_scale=2',
            size=10, transform=ax.transAxes)

    # draws control points for the fancy box.
    #l = p_fancy.get_path().vertices
    #ax.plot(l[:,0], l[:,1], ".")

    draw_bbox(ax, bb)


def test4(ax):

    # When the aspect ratio of the axes is not 1, the fancy box may
    # not be what you expected (green)

    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),
                             abs(bb.width), abs(bb.height),
                             boxstyle="round,pad=0.2",
                             fc="none",
                             ec=(0., .5, 0.), zorder=4)

    ax.add_patch(p_fancy)

    # You can compensate this by setting the mutation_aspect (pink).
    p_fancy = FancyBboxPatch((bb.xmin, bb.ymin),
                             abs(bb.width), abs(bb.height),
                             boxstyle="round,pad=0.3",
                             mutation_aspect=.5,
                             fc=(1., 0.8, 1.),
                             ec=(1., 0.5, 1.))

    ax.add_patch(p_fancy)

    ax.text(0.1, 0.8,
            ' boxstyle="round,pad=0.3"\n mutation\\_aspect=.5',
            size=10, transform=ax.transAxes)

    draw_bbox(ax, bb)
```

```python
def test_all():
    plt.clf()

    ax = plt.subplot(2, 2, 1)
    test1(ax)
    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.set_title("test1")
    ax.set_aspect(1.)

    ax = plt.subplot(2, 2, 2)
    ax.set_title("test2")
    test2(ax)
    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.set_aspect(1.)

    ax = plt.subplot(2, 2, 3)
    ax.set_title("test3")
    test3(ax)
    ax.set_xlim(0., 1.)
    ax.set_ylim(0., 1.)
    ax.set_aspect(1)

    ax = plt.subplot(2, 2, 4)
    ax.set_title("test4")
    test4(ax)
    ax.set_xlim(-0.5, 1.5)
    ax.set_ylim(0., 1.)
    ax.set_aspect(2.)

    plt.draw()
    plt.show()

test_all()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.76 pylab_examples example code: fancybox_demo2.py

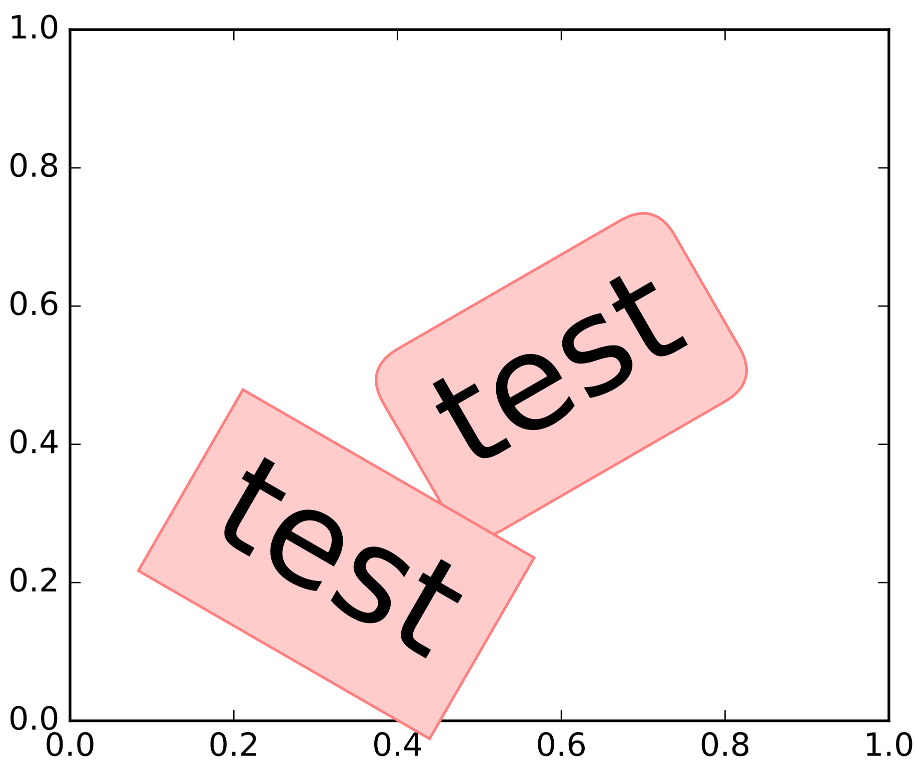



```
import matplotlib.patches as mpatch
import matplotlib.pyplot as plt

styles = mpatch.BoxStyle.get_styles()
```

```
spacing = 1.2

figheight = (spacing * len(styles) + .5)
fig1 = plt.figure(1, (4/1.5, figheight/1.5))
fontsize = 0.3 * 72

for i, stylename in enumerate(sorted(styles.keys())):
    fig1.text(0.5, (spacing * (float(len(styles)) - i) - 0.5)/figheight, stylename,
              ha="center",
              size=fontsize,
              transform=fig1.transFigure,
              bbox=dict(boxstyle=stylename, fc="w", ec="k"))
plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.77 pylab_examples example code: fancytextbox_demo.py

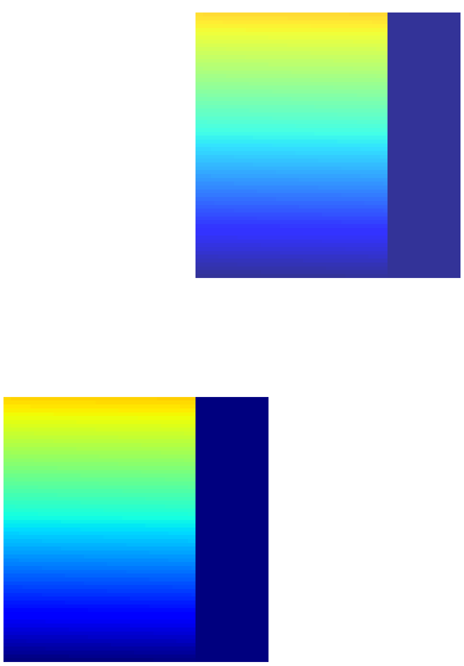



```
import matplotlib.pyplot as plt

plt.text(0.6, 0.5, "test", size=50, rotation=30.,
```

```
         ha="center", va="center",
         bbox=dict(boxstyle="round",
                   ec=(1., 0.5, 0.5),
                   fc=(1., 0.8, 0.8),
                   )
         )

plt.text(0.5, 0.4, "test", size=50, rotation=-30.,
         ha="right", va="top",
         bbox=dict(boxstyle="square",
                   ec=(1., 0.5, 0.5),
                   fc=(1., 0.8, 0.8),
                   )
         )


plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

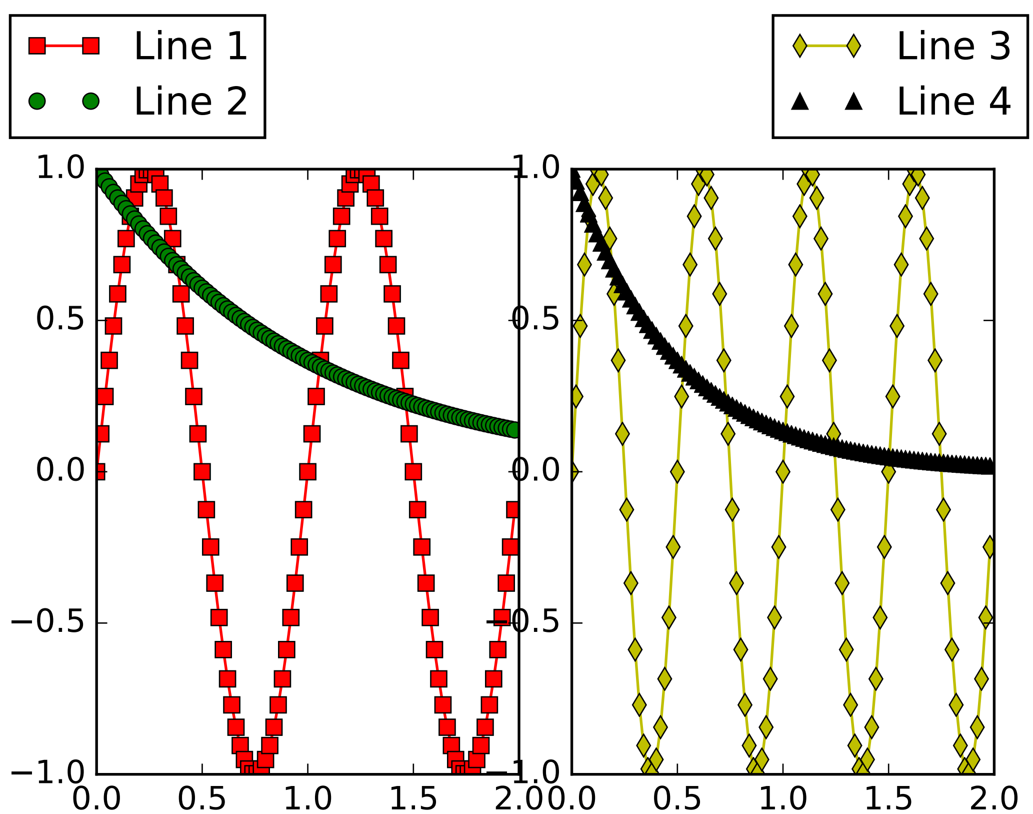
90.78 pylab_examples example code: figimage_demo.py



```
"""
This illustrates placing images directly in the figure, with no axes.

"""
import numpy as np
import matplotlib
import matplotlib.cm as cm
import matplotlib.pyplot as plt


fig = plt.figure()
Z = np.arange(10000.0)
Z.shape = 100, 100
Z[:, 50:] = 1.

im1 = plt.figimage(Z, xo=50, yo=0, cmap=cm.jet, origin='lower')
im2 = plt.figimage(Z, xo=100, yo=100, alpha=.8, cmap=cm.jet, origin='lower')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

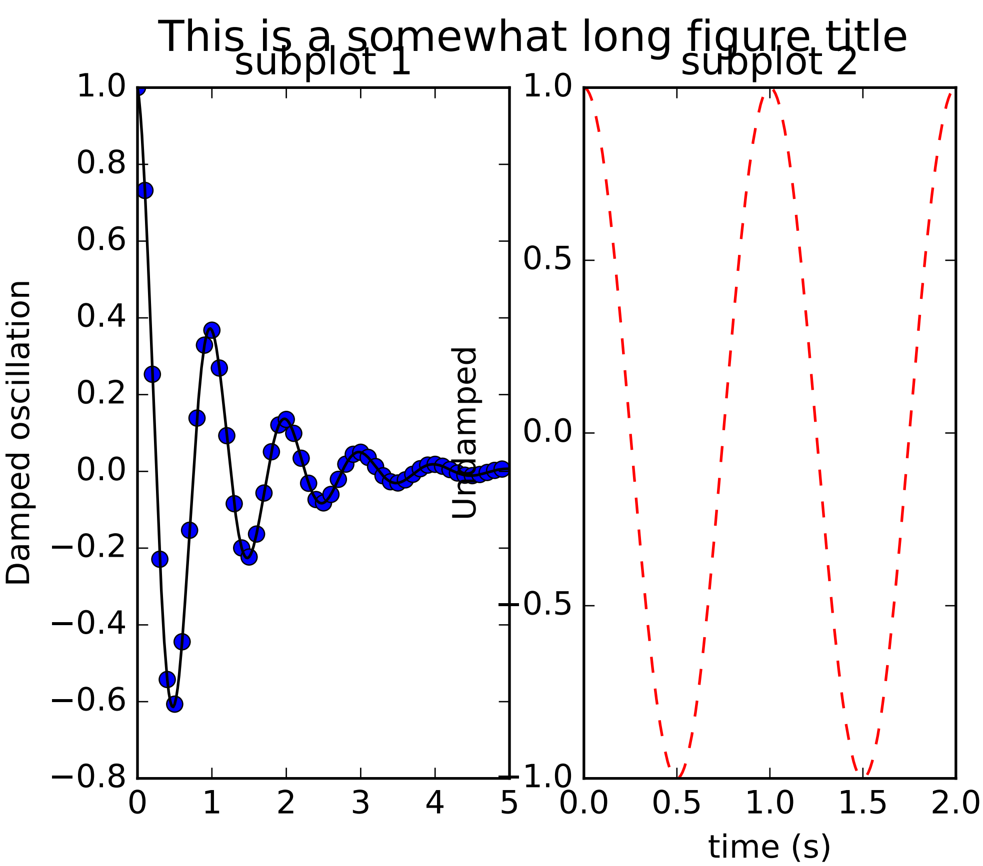
90.79 pylab_examples example code: figlegend_demo.py



```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.1, 0.4, 0.7])
ax2 = fig.add_axes([0.55, 0.1, 0.4, 0.7])

x = np.arange(0.0, 2.0, 0.02)
y1 = np.sin(2*np.pi*x)
y2 = np.exp(-x)
l1, l2 = ax1.plot(x, y1, 'rs-', x, y2, 'go')

y3 = np.sin(4*np.pi*x)
y4 = np.exp(-2*x)
l3, l4 = ax2.plot(x, y3, 'yd-', x, y4, 'k^')

fig.legend((l1, l2), ('Line 1', 'Line 2'), 'upper left')
fig.legend((l3, l4), ('Line 3', 'Line 4'), 'upper right')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

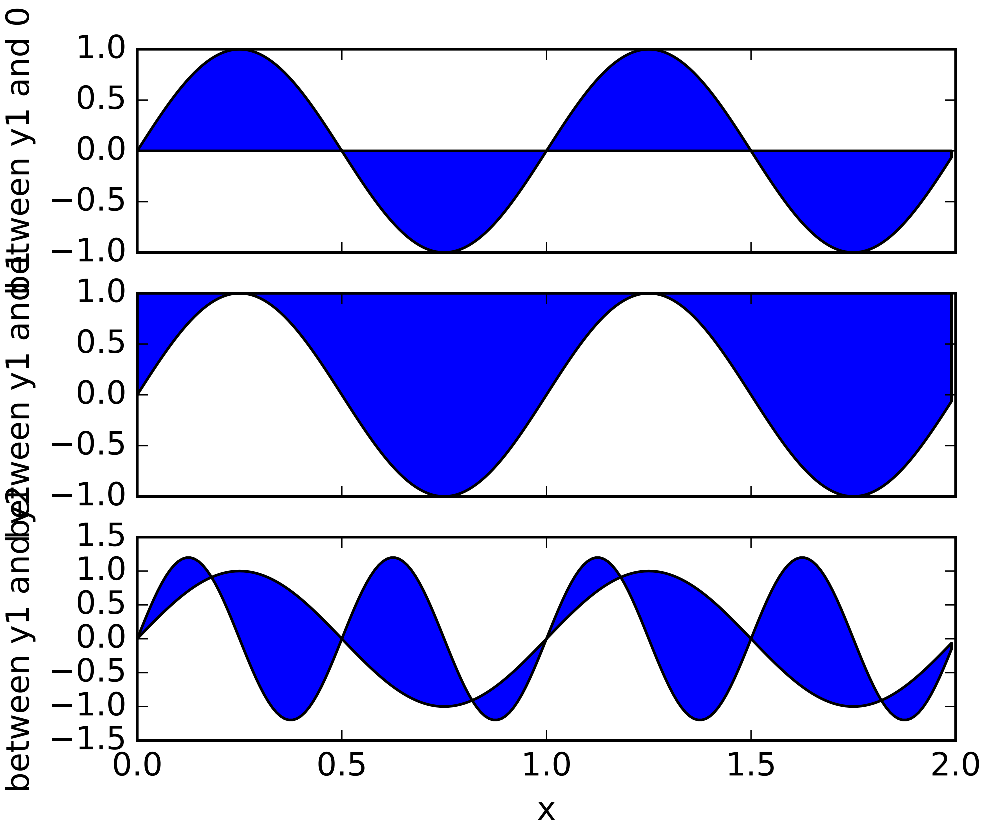
90.80 pylab_examples example code: figure_title.py

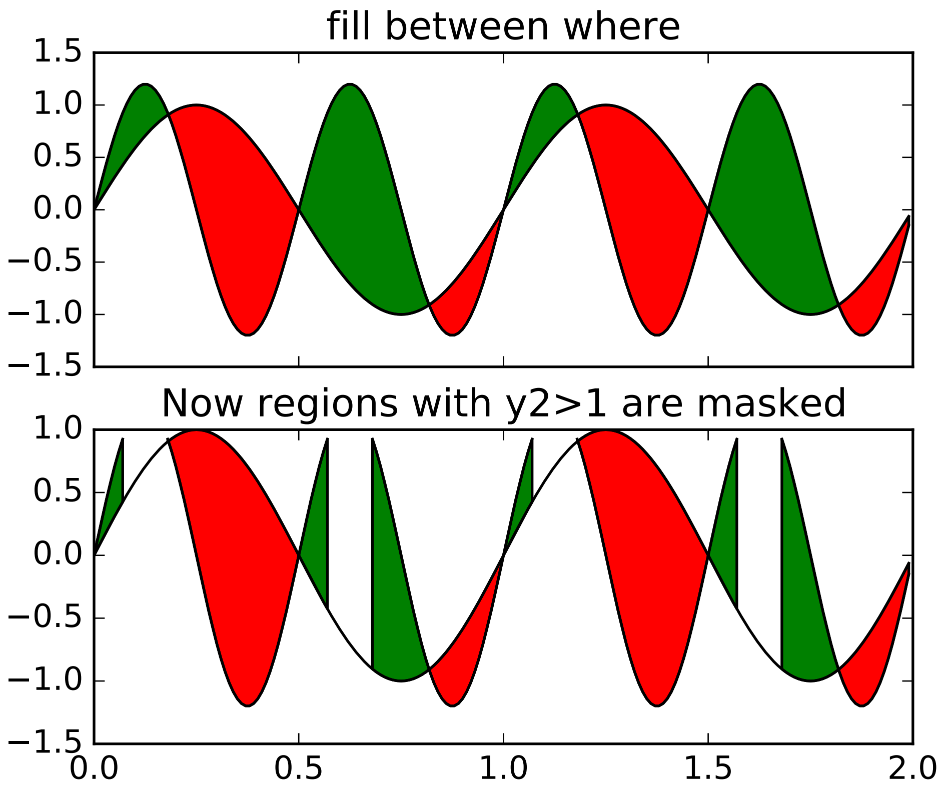


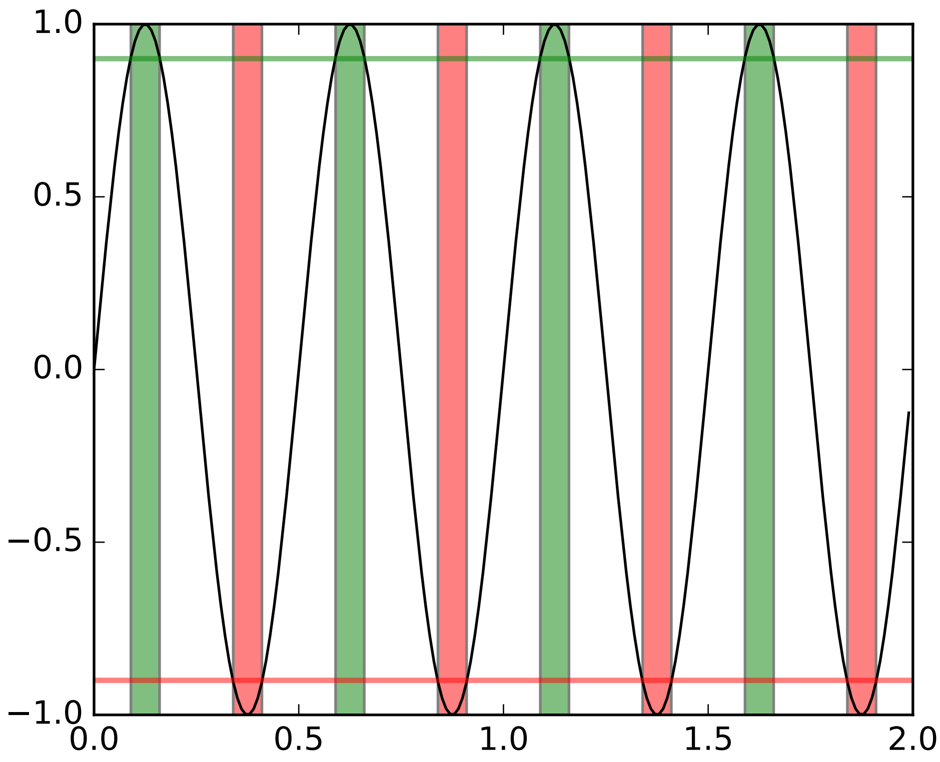
```
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt
import numpy as np


def f(t):
    s1 = np.cos(2*np.pi*t)
    e1 = np.exp(-t)
    return s1 * e1

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)


plt.subplot(121)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.title('subplot 1')
plt.ylabel('Damped oscillation')
plt.suptitle('This is a somewhat long figure title', fontsize=16)


plt.subplot(122)
plt.plot(t3, np.cos(2*np.pi*t3), 'r--')
plt.xlabel('time (s)')
plt.title('subplot 2')
plt.ylabel('Undamped')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.81 pylab_examples example code: fill_between_demo.py




fill between where
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Now regions with y2>1 are masked
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.5
1.0
1.5
2.0

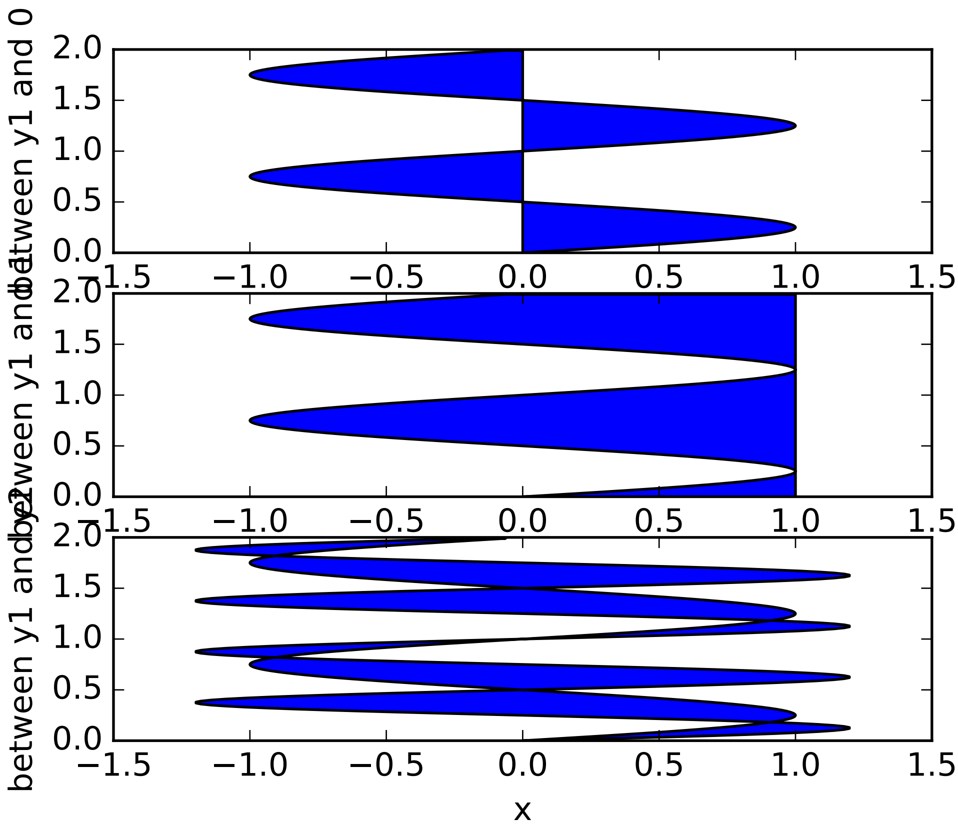


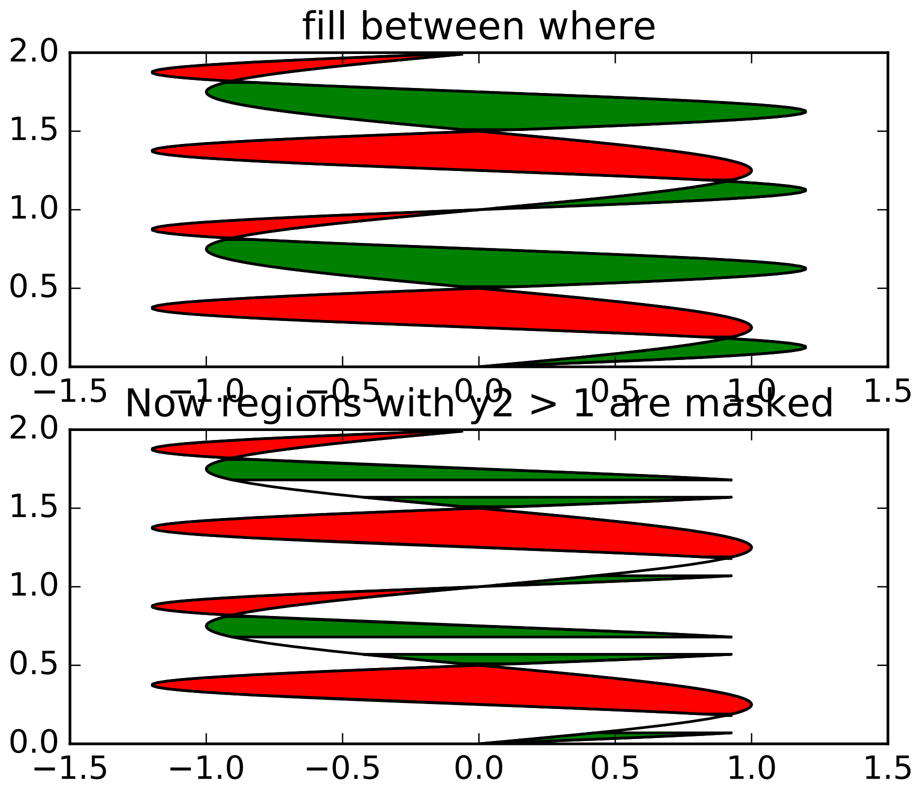

```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)

ax1.fill_between(x, 0, y1)
ax1.set_ylabel('between y1 and 0')

ax2.fill_between(x, y1, 1)
ax2.set_ylabel('between y1 and 1')

ax3.fill_between(x, y1, y2)
ax3.set_ylabel('between y1 and y2')
ax3.set_xlabel('x')

# now fill between y1 and y2 where a logical condition is met.  Note
# this is different than calling
#   fill_between(x[where], y1[where],y2[where]
# because of edge effects over multiple contiguous regions.
fig, (ax, ax1) = plt.subplots(2, 1, sharex=True)
ax.plot(x, y1, x, y2, color='black')
```

```
ax.fill_between(x, y1, y2, where=y2 >= y1, facecolor='green', interpolate=True)
ax.fill_between(x, y1, y2, where=y2 <= y1, facecolor='red', interpolate=True)
ax.set_title('fill between where')

# Test support for masked arrays.
y2 = np.ma.masked_greater(y2, 1.0)
ax1.plot(x, y1, x, y2, color='black')
ax1.fill_between(x, y1, y2, where=y2 >= y1, facecolor='green', interpolate=True)
ax1.fill_between(x, y1, y2, where=y2 <= y1, facecolor='red', interpolate=True)
ax1.set_title('Now regions with y2>1 are masked')

# This example illustrates a problem; because of the data
# gridding, there are undesired unfilled triangles at the crossover
# points.  A brute-force solution would be to interpolate all
# arrays to a very fine grid before plotting.

# show how to use transforms to create axes spans where a certain condition is satisfied
fig, ax = plt.subplots()
y = np.sin(4*np.pi*x)
ax.plot(x, y, color='black')

# use the data coordinates for the x-axis and the axes coordinates for the y-axis
import matplotlib.transforms as mtransforms
trans = mtransforms.blended_transform_factory(ax.transData, ax.transAxes)
theta = 0.9
ax.axhline(theta, color='green', lw=2, alpha=0.5)
ax.axhline(-theta, color='red', lw=2, alpha=0.5)
ax.fill_between(x, 0, 1, where=y > theta, facecolor='green', alpha=0.5, transform=trans)
ax.fill_between(x, 0, 1, where=y < -theta, facecolor='red', alpha=0.5, transform=trans)


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.82 pylab_examples example code: fill_betweenx_demo.py



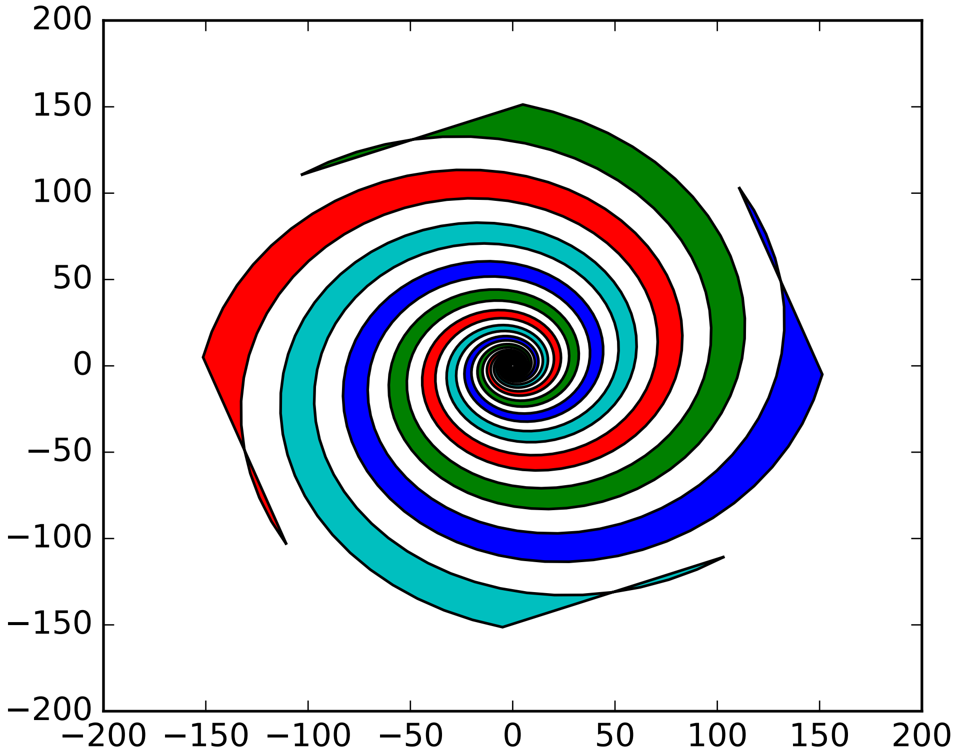



```
"""
Copy of fill_between.py but using fill_betweenx() instead.
"""
import matplotlib.mlab as mlab
from matplotlib.pyplot import figure, show
import numpy as np


x = np.arange(0.0, 2, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = 1.2*np.sin(4*np.pi*x)

fig = figure()
ax1 = fig.add_subplot(311)
ax2 = fig.add_subplot(312, sharex=ax1)
ax3 = fig.add_subplot(313, sharex=ax1)

ax1.fill_betweenx(x, 0, y1)
ax1.set_ylabel('between y1 and 0')

ax2.fill_betweenx(x, y1, 1)
ax2.set_ylabel('between y1 and 1')

ax3.fill_betweenx(x, y1, y2)
ax3.set_ylabel('between y1 and y2')
ax3.set_xlabel('x')
```

```
# now fill between y1 and y2 where a logical condition is met.  Note
# this is different than calling
#   fill_between(x[where], y1[where],y2[where]
# because of edge effects over multiple contiguous regions.
fig = figure()
ax = fig.add_subplot(211)
ax.plot(y1, x, y2, x, color='black')
ax.fill_betweenx(x, y1, y2, where=y2 >= y1, facecolor='green')
ax.fill_betweenx(x, y1, y2, where=y2 <= y1, facecolor='red')
ax.set_title('fill between where')

# Test support for masked arrays.
y2 = np.ma.masked_greater(y2, 1.0)
ax1 = fig.add_subplot(212, sharex=ax)
ax1.plot(y1, x, y2, x, color='black')
ax1.fill_betweenx(x, y1, y2, where=y2 >= y1, facecolor='green')
ax1.fill_betweenx(x, y1, y2, where=y2 <= y1, facecolor='red')
ax1.set_title('Now regions with y2 > 1 are masked')

# This example illustrates a problem; because of the data
# gridding, there are undesired unfilled triangles at the crossover
# points.  A brute-force solution would be to interpolate all
# arrays to a very fine grid before plotting.

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.83 pylab_examples example code: fill_spiral.py

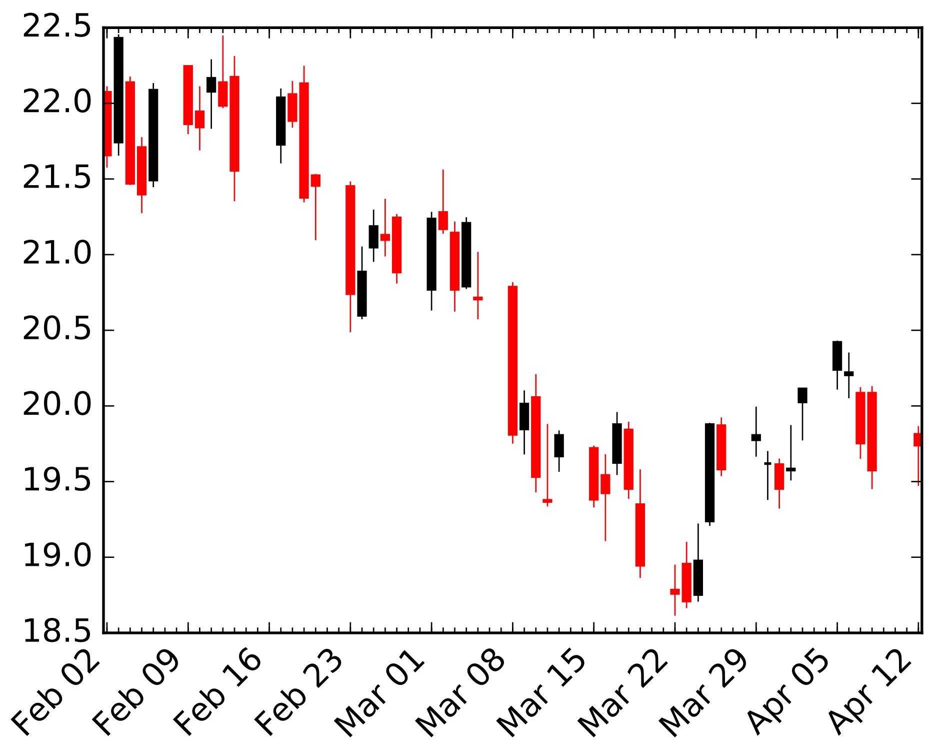



```
import matplotlib.pyplot as plt
import numpy as np

theta = np.arange(0, 8*np.pi, 0.1)
a = 1
b = .2

for dt in np.arange(0, 2*np.pi, np.pi/2.0):

    x = a*np.cos(theta + dt)*np.exp(b*theta)
    y = a*np.sin(theta + dt)*np.exp(b*theta)

    dt = dt + np.pi/4.0

    x2 = a*np.cos(theta + dt)*np.exp(b*theta)
    y2 = a*np.sin(theta + dt)*np.exp(b*theta)

    xf = np.concatenate((x, x2[::-1]))
    yf = np.concatenate((y, y2[::-1]))

    p1 = plt.fill(xf, yf)
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.84 pylab_examples example code: finance_demo.py

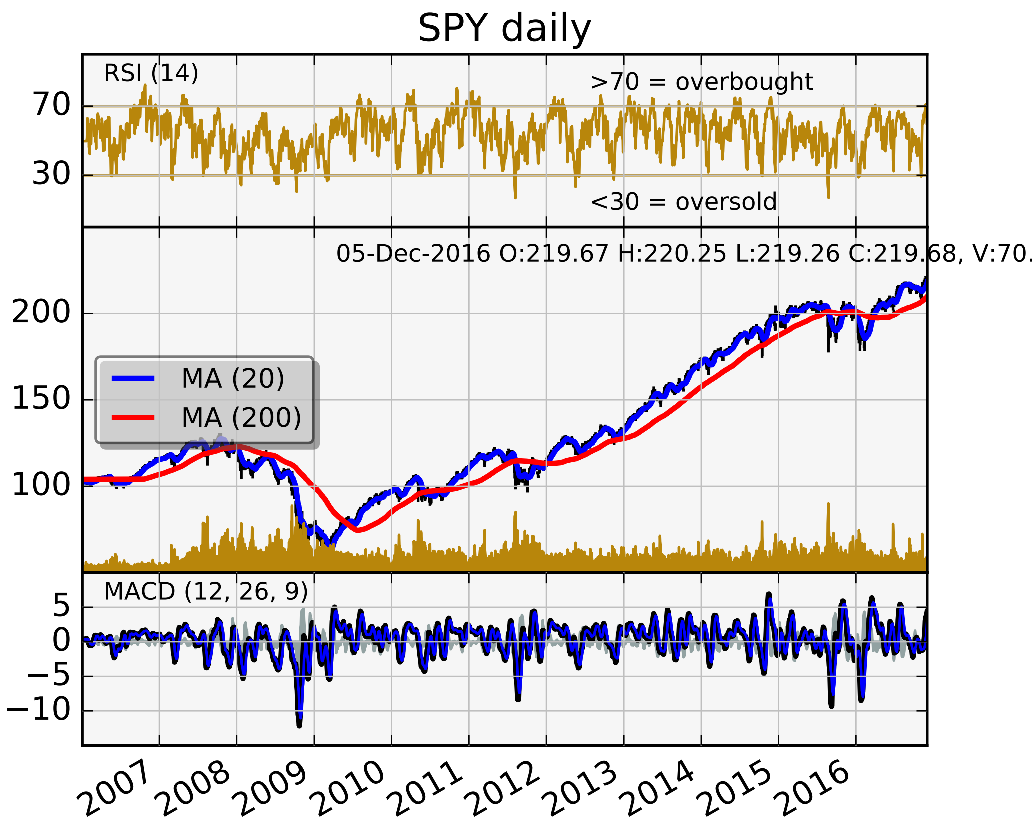



```
#!/usr/bin/env python
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter, WeekdayLocator,\
    DayLocator, MONDAY
from matplotlib.finance import quotes_historical_yahoo_ohlc, candlestick_ohlc


# (Year, month, day) tuples suffice as args for quotes_historical_yahoo
date1 = (2004, 2, 1)
date2 = (2004, 4, 12)


mondays = WeekdayLocator(MONDAY)        # major ticks on the mondays
alldays = DayLocator()              # minor ticks on the days
weekFormatter = DateFormatter('%b %d')  # e.g., Jan 12
dayFormatter = DateFormatter('%d')      # e.g., 12
```

```
quotes = quotes_historical_yahoo_ohlc('INTC', date1, date2)
if len(quotes) == 0:
    raise SystemExit

fig, ax = plt.subplots()
fig.subplots_adjust(bottom=0.2)
ax.xaxis.set_major_locator(mondays)
ax.xaxis.set_minor_locator(alldays)
ax.xaxis.set_major_formatter(weekFormatter)
#ax.xaxis.set_minor_formatter(dayFormatter)

#plot_day_summary(ax, quotes, ticksize=3)
candlestick_ohlc(ax, quotes, width=0.6)

ax.xaxis_date()
ax.autoscale_view()
plt.setp(plt.gca().get_xticklabels(), rotation=45, horizontalalignment='right')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.85 pylab_examples example code: finance_work2.py




```
import datetime
import numpy as np
import matplotlib.colors as colors
import matplotlib.finance as finance
import matplotlib.dates as mdates
import matplotlib.ticker as mticker
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import matplotlib.font_manager as font_manager


startdate = datetime.date(2006, 1, 1)
today = enddate = datetime.date.today()
ticker = 'SPY'


fh = finance.fetch_historical_yahoo(ticker, startdate, enddate)
# a numpy record array with fields: date, open, high, low, close, volume, adj_close)

r = mlab.csv2rec(fh)
fh.close()
r.sort()
```

```
def moving_average(x, n, type='simple'):
    """
    compute an n period moving average.

    type is 'simple' | 'exponential'

    """
    x = np.asarray(x)
    if type == 'simple':
        weights = np.ones(n)
    else:
        weights = np.exp(np.linspace(-1., 0., n))

    weights /= weights.sum()

    a = np.convolve(x, weights, mode='full')[:len(x)]
    a[:n] = a[n]
    return a


def relative_strength(prices, n=14):
    """
    compute the n period relative strength indicator
    http://stockcharts.com/school/doku.php?id=chart_school:glossary_r
 ↪#relativestrengthindex
    http://www.investopedia.com/terms/r/rsi.asp
    """

    deltas = np.diff(prices)
    seed = deltas[:n+1]
    up = seed[seed >= 0].sum()/n
    down = -seed[seed < 0].sum()/n
    rs = up/down
    rsi = np.zeros_like(prices)
    rsi[:n] = 100. - 100./(1. + rs)

    for i in range(n, len(prices)):
        delta = deltas[i - 1]  # cause the diff is 1 shorter

        if delta > 0:
            upval = delta
            downval = 0.
        else:
            upval = 0.
            downval = -delta

        up = (up*(n - 1) + upval)/n
        down = (down*(n - 1) + downval)/n

        rs = up/down
        rsi[i] = 100. - 100./(1. + rs)
```

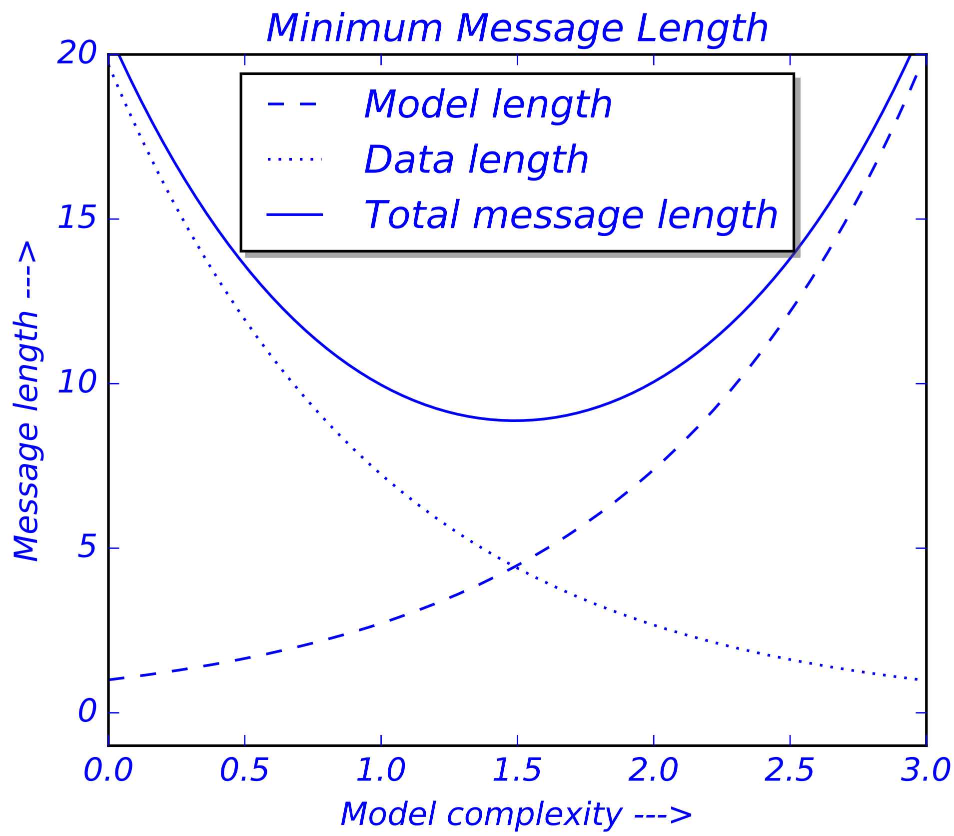
```
    return rsi


def moving_average_convergence(x, nslow=26, nfast=12):
    """
    compute the MACD (Moving Average Convergence/Divergence) using a fast and slow␣
↪exponential moving avg'
    return value is emaslow, emafast, macd which are len(x) arrays
    """
    emaslow = moving_average(x, nslow, type='exponential')
    emafast = moving_average(x, nfast, type='exponential')
    return emaslow, emafast, emafast - emaslow


plt.rc('axes', grid=True)
plt.rc('grid', color='0.75', linestyle='-', linewidth=0.5)

textsize = 9
left, width = 0.1, 0.8
rect1 = [left, 0.7, width, 0.2]
rect2 = [left, 0.3, width, 0.4]
rect3 = [left, 0.1, width, 0.2]


fig = plt.figure(facecolor='white')
axescolor = '#f6f6f6'  # the axes background color

ax1 = fig.add_axes(rect1, axisbg=axescolor)  # left, bottom, width, height
ax2 = fig.add_axes(rect2, axisbg=axescolor, sharex=ax1)
ax2t = ax2.twinx()
ax3 = fig.add_axes(rect3, axisbg=axescolor, sharex=ax1)


# plot the relative strength indicator
prices = r.adj_close
rsi = relative_strength(prices)
fillcolor = 'darkgoldenrod'

ax1.plot(r.date, rsi, color=fillcolor)
ax1.axhline(70, color=fillcolor)
ax1.axhline(30, color=fillcolor)
ax1.fill_between(r.date, rsi, 70, where=(rsi >= 70), facecolor=fillcolor,␣
↪edgecolor=fillcolor)
ax1.fill_between(r.date, rsi, 30, where=(rsi <= 30), facecolor=fillcolor,␣
↪edgecolor=fillcolor)
ax1.text(0.6, 0.9, '>70 = overbought', va='top', transform=ax1.transAxes,␣
↪fontsize=textsize)
ax1.text(0.6, 0.1, '<30 = oversold', transform=ax1.transAxes, fontsize=textsize)
ax1.set_ylim(0, 100)
ax1.set_yticks([30, 70])
ax1.text(0.025, 0.95, 'RSI (14)', va='top', transform=ax1.transAxes, fontsize=textsize)
ax1.set_title('%s daily' % ticker)
```

```
# plot the price and volume data
dx = r.adj_close - r.close
low = r.low + dx
high = r.high + dx

deltas = np.zeros_like(prices)
deltas[1:] = np.diff(prices)
up = deltas > 0
ax2.vlines(r.date[up], low[up], high[up], color='black', label='_nolegend_')
ax2.vlines(r.date[~up], low[~up], high[~up], color='black', label='_nolegend_')
ma20 = moving_average(prices, 20, type='simple')
ma200 = moving_average(prices, 200, type='simple')

linema20, = ax2.plot(r.date, ma20, color='blue', lw=2, label='MA (20)')
linema200, = ax2.plot(r.date, ma200, color='red', lw=2, label='MA (200)')


last = r[-1]
s = '%s O:%1.2f H:%1.2f L:%1.2f C:%1.2f, V:%1.1fM Chg:%+1.2f' % (
    today.strftime('%d-%b-%Y'),
    last.open, last.high,
    last.low, last.close,
    last.volume*1e-6,
    last.close - last.open)
t4 = ax2.text(0.3, 0.9, s, transform=ax2.transAxes, fontsize=textsize)

props = font_manager.FontProperties(size=10)
leg = ax2.legend(loc='center left', shadow=True, fancybox=True, prop=props)
leg.get_frame().set_alpha(0.5)


volume = (r.close*r.volume)/1e6  # dollar volume in millions
vmax = volume.max()
poly = ax2t.fill_between(r.date, volume, 0, label='Volume', facecolor=fillcolor,␣
 ↪edgecolor=fillcolor)
ax2t.set_ylim(0, 5*vmax)
ax2t.set_yticks([])


# compute the MACD indicator
fillcolor = 'darkslategrey'
nslow = 26
nfast = 12
nema = 9
emaslow, emafast, macd = moving_average_convergence(prices, nslow=nslow, nfast=nfast)
ema9 = moving_average(macd, nema, type='exponential')
ax3.plot(r.date, macd, color='black', lw=2)
ax3.plot(r.date, ema9, color='blue', lw=1)
ax3.fill_between(r.date, macd - ema9, 0, alpha=0.5, facecolor=fillcolor,␣
 ↪edgecolor=fillcolor)
```

```
ax3.text(0.025, 0.95, 'MACD (%d, %d, %d)' % (nfast, nslow, nema), va='top',
         transform=ax3.transAxes, fontsize=textsize)

#ax3.set_yticks([])
# turn off upper axis tick labels, rotate the lower ones, etc
for ax in ax1, ax2, ax2t, ax3:
    if ax != ax3:
        for label in ax.get_xticklabels():
            label.set_visible(False)
    else:
        for label in ax.get_xticklabels():
            label.set_rotation(30)
            label.set_horizontalalignment('right')

    ax.fmt_xdata = mdates.DateFormatter('%Y-%m-%d')


class MyLocator(mticker.MaxNLocator):
    def __init__(self, *args, **kwargs):
        mticker.MaxNLocator.__init__(self, *args, **kwargs)

    def __call__(self, *args, **kwargs):
        return mticker.MaxNLocator.__call__(self, *args, **kwargs)

# at most 5 ticks, pruning the upper and lower so they don't overlap
# with other ticks
#ax2.yaxis.set_major_locator(mticker.MaxNLocator(5, prune='both'))
#ax3.yaxis.set_major_locator(mticker.MaxNLocator(5, prune='both'))

ax2.yaxis.set_major_locator(MyLocator(5, prune='both'))
ax3.yaxis.set_major_locator(MyLocator(5, prune='both'))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.86 pylab_examples example code: findobj_demo.py




```
"""
Recursively find all objects that match some criteria
"""
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.text as text

a = np.arange(0, 3, .02)
b = np.arange(0, 3, .02)
c = np.exp(a)
d = c[::-1]

fig, ax = plt.subplots()
plt.plot(a, c, 'k--', a, d, 'k:', a, c + d, 'k')
plt.legend(('Model length', 'Data length', 'Total message length'),
           loc='upper center', shadow=True)
plt.ylim([-1, 20])
plt.grid(False)
plt.xlabel('Model complexity --->')
plt.ylabel('Message length --->')
plt.title('Minimum Message Length')
```

```
# match on arbitrary function
def myfunc(x):
    return hasattr(x, 'set_color') and not hasattr(x, 'set_facecolor')

for o in fig.findobj(myfunc):
    o.set_color('blue')

# match on class instances
for o in fig.findobj(text.Text):
    o.set_fontstyle('italic')


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.87 pylab_examples example code: font_table_ttf.py

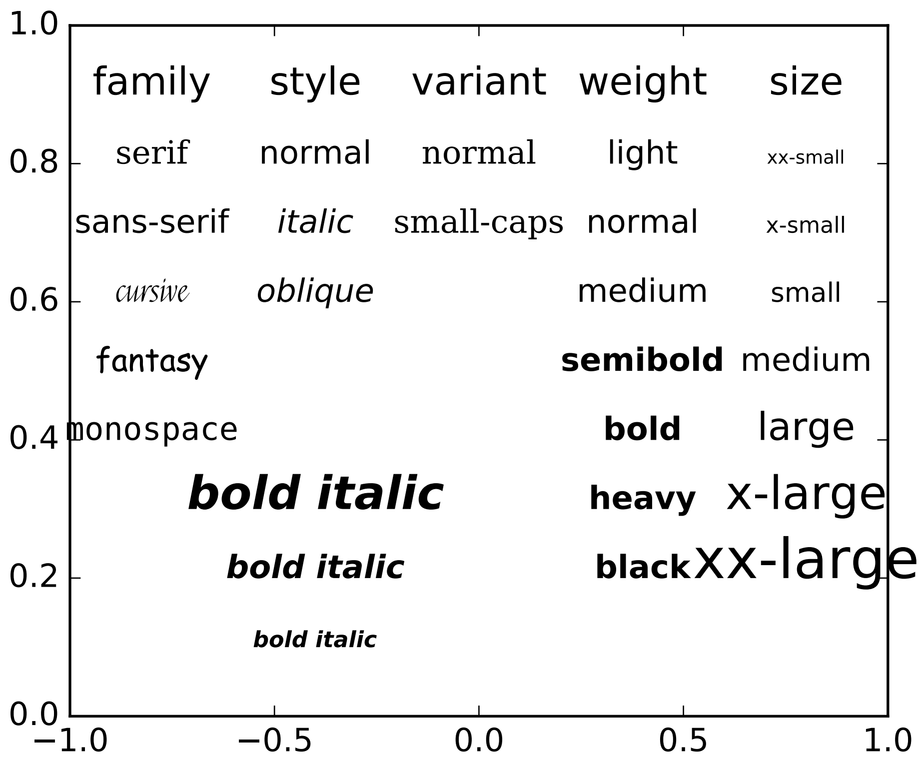
[source code]

```
# -*- noplot -*-
"""
matplotlib has support for freetype fonts.  Here's a little example
using the 'table' command to build a font table that shows the glyphs
by character code.

Usage python font_table_ttf.py somefile.ttf
"""

import sys
import os

import matplotlib
from matplotlib.ft2font import FT2Font
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt

import six
from six import unichr

# the font table grid

labelc = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
          'A', 'B', 'C', 'D', 'E', 'F']
labelr = ['00', '10', '20', '30', '40', '50', '60', '70', '80', '90',
          'A0', 'B0', 'C0', 'D0', 'E0', 'F0']

if len(sys.argv) > 1:
    fontname = sys.argv[1]
else:
```

```
    fontname = os.path.join(matplotlib.get_data_path(),
                            'fonts', 'ttf', 'Vera.ttf')

font = FT2Font(fontname)
codes = list(font.get_charmap().items())
codes.sort()

# a 16,16 array of character strings
chars = [['' for c in range(16)] for r in range(16)]
colors = [[(0.95, 0.95, 0.95) for c in range(16)] for r in range(16)]

plt.figure(figsize=(8, 4), dpi=120)
for ccode, glyphind in codes:
    if ccode >= 256:
        continue
    r, c = divmod(ccode, 16)
    s = unichr(ccode)
    chars[r][c] = s

lightgrn = (0.5, 0.8, 0.5)
plt.title(fontname)
tab = plt.table(cellText=chars,
                rowLabels=labelr,
                colLabels=labelc,
                rowColours=[lightgrn]*16,
                colColours=[lightgrn]*16,
                cellColours=colors,
                cellLoc='center',
                loc='upper left')

for key, cell in tab.get_celld().items():
    row, col = key
    if row > 0 and col > 0:
        cell.set_text_props(fontproperties=FontProperties(fname=fontname))
plt.axis('off')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.88 pylab_examples example code: fonts_demo.py



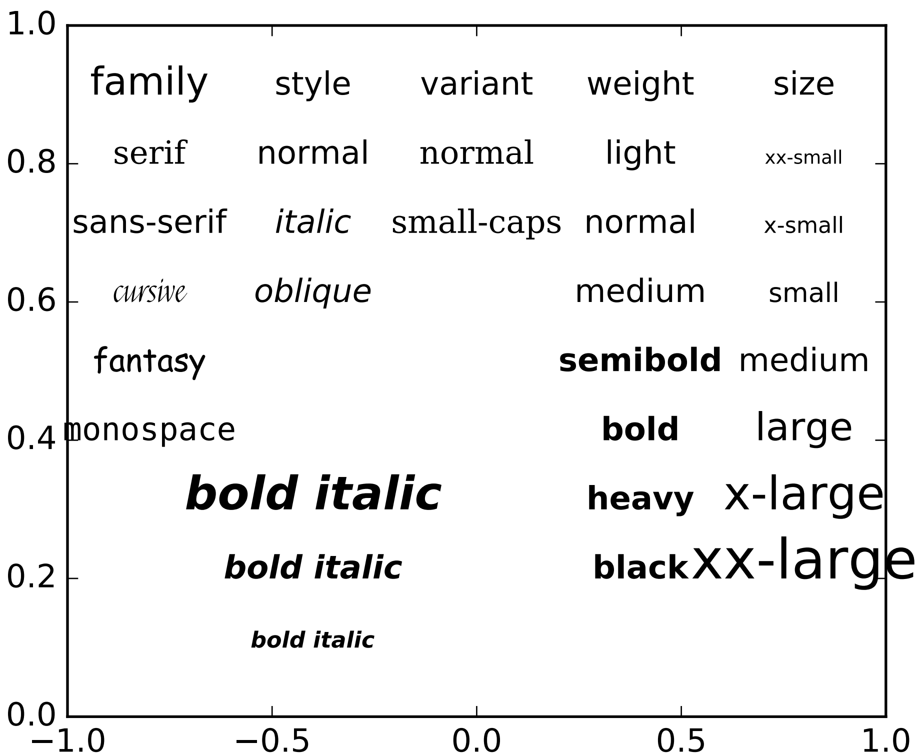

```
"""
Show how to set custom font properties.

For interactive users, you can also use kwargs to the text command,
which requires less typing.  See examples/fonts_demo_kw.py
"""
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt

plt.subplot(111, axisbg='w')

font0 = FontProperties()
alignment = {'horizontalalignment': 'center', 'verticalalignment': 'baseline'}
# Show family options

families = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace']

font1 = font0.copy()
font1.set_size('large')

t = plt.text(-0.8, 0.9, 'family', fontproperties=font1,
             **alignment)
```

```
yp = [0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2]

for k, family in enumerate(families):
    font = font0.copy()
    font.set_family(family)
    t = plt.text(-0.8, yp[k], family, fontproperties=font,
                 **alignment)

# Show style options

styles = ['normal', 'italic', 'oblique']

t = plt.text(-0.4, 0.9, 'style', fontproperties=font1,
             **alignment)

for k, style in enumerate(styles):
    font = font0.copy()
    font.set_family('sans-serif')
    font.set_style(style)
    t = plt.text(-0.4, yp[k], style, fontproperties=font,
                 **alignment)

# Show variant options

variants = ['normal', 'small-caps']

t = plt.text(0.0, 0.9, 'variant', fontproperties=font1,
             **alignment)

for k, variant in enumerate(variants):
    font = font0.copy()
    font.set_family('serif')
    font.set_variant(variant)
    t = plt.text(0.0, yp[k], variant, fontproperties=font,
                 **alignment)

# Show weight options

weights = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']

t = plt.text(0.4, 0.9, 'weight', fontproperties=font1,
             **alignment)

for k, weight in enumerate(weights):
    font = font0.copy()
    font.set_weight(weight)
    t = plt.text(0.4, yp[k], weight, fontproperties=font,
                 **alignment)

# Show size options

sizes = ['xx-small', 'x-small', 'small', 'medium', 'large',
```

```
        'x-large', 'xx-large']

t = plt.text(0.8, 0.9, 'size', fontproperties=font1,
             **alignment)

for k, size in enumerate(sizes):
    font = font0.copy()
    font.set_size(size)
    t = plt.text(0.8, yp[k], size, fontproperties=font,
                 **alignment)

# Show bold italic

font = font0.copy()
font.set_style('italic')
font.set_weight('bold')
font.set_size('x-small')
t = plt.text(-0.4, 0.1, 'bold italic', fontproperties=font,
             **alignment)

font = font0.copy()
font.set_style('italic')
font.set_weight('bold')
font.set_size('medium')
t = plt.text(-0.4, 0.2, 'bold italic', fontproperties=font,
             **alignment)

font = font0.copy()
font.set_style('italic')
font.set_weight('bold')
font.set_size('x-large')
t = plt.text(-0.4, 0.3, 'bold italic', fontproperties=font,
             **alignment)

plt.axis([-1, 1, 0, 1])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.89 pylab_examples example code: fonts_demo_kw.py



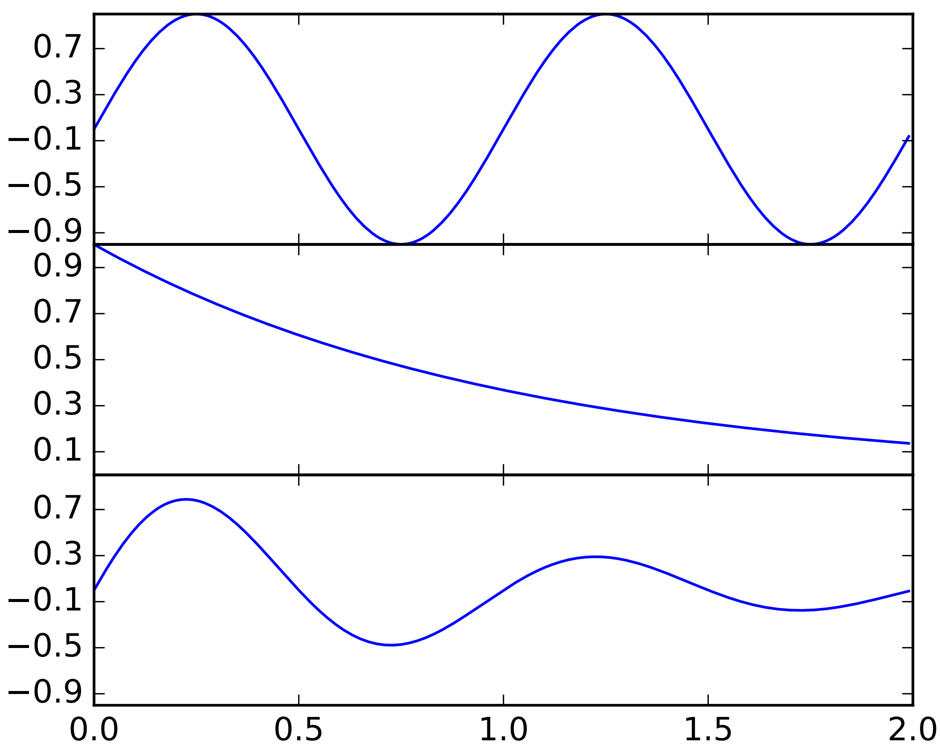

```
"""
Same as fonts_demo using kwargs.  If you prefer a more pythonic, OO
style of coding, see examples/fonts_demo.py.

"""
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt
import numpy as np

plt.subplot(111, axisbg='w')
alignment = {'horizontalalignment': 'center', 'verticalalignment': 'baseline'}

# Show family options

families = ['serif', 'sans-serif', 'cursive', 'fantasy', 'monospace']

t = plt.text(-0.8, 0.9, 'family', size='large', **alignment)

yp = [0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2]

for k, family in enumerate(families):
    t = plt.text(-0.8, yp[k], family, family=family, **alignment)
```

```
# Show style options

styles = ['normal', 'italic', 'oblique']

t = plt.text(-0.4, 0.9, 'style', **alignment)

for k, style in enumerate(styles):
    t = plt.text(-0.4, yp[k], style, family='sans-serif', style=style,
                 **alignment)

# Show variant options

variants = ['normal', 'small-caps']

t = plt.text(0.0, 0.9, 'variant', **alignment)

for k, variant in enumerate(variants):
    t = plt.text(0.0, yp[k], variant, family='serif', variant=variant,
                 **alignment)

# Show weight options

weights = ['light', 'normal', 'medium', 'semibold', 'bold', 'heavy', 'black']

t = plt.text(0.4, 0.9, 'weight', **alignment)

for k, weight in enumerate(weights):
    t = plt.text(0.4, yp[k], weight, weight=weight,
                 **alignment)

# Show size options

sizes = ['xx-small', 'x-small', 'small', 'medium', 'large',
         'x-large', 'xx-large']

t = plt.text(0.8, 0.9, 'size', **alignment)

for k, size in enumerate(sizes):
    t = plt.text(0.8, yp[k], size, size=size,
                 **alignment)

x = -0.4
# Show bold italic
t = plt.text(x, 0.1, 'bold italic', style='italic',
             weight='bold', size='x-small',
             **alignment)

t = plt.text(x, 0.2, 'bold italic',
             style='italic', weight='bold', size='medium',
             **alignment)

t = plt.text(x, 0.3, 'bold italic',
```

```
            style='italic', weight='bold', size='x-large',
            **alignment)

plt.axis([-1, 1, 0, 1])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.90 pylab_examples example code: ganged_plots.py

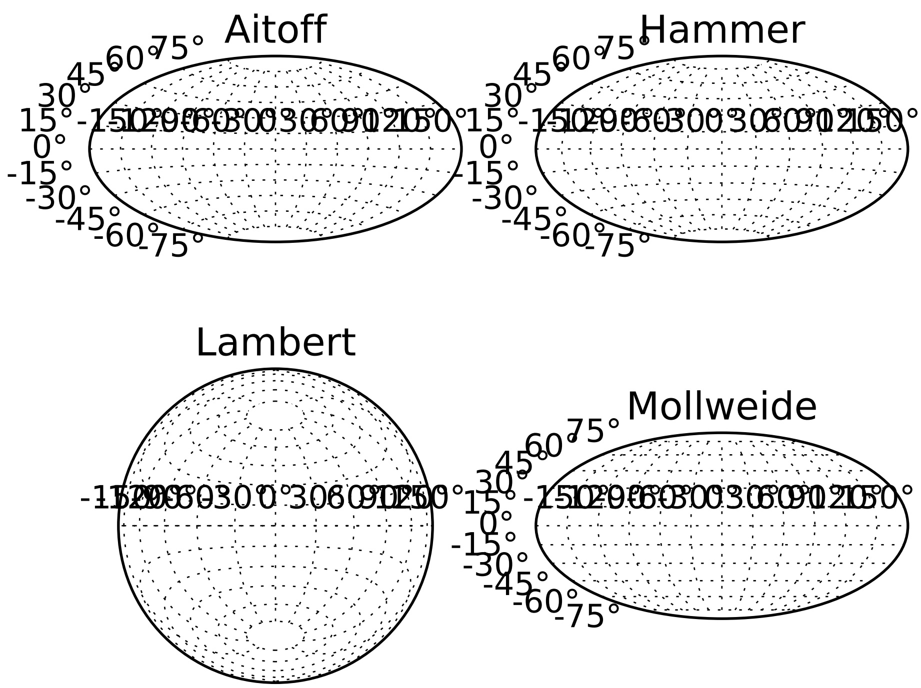



```
"""
To create plots that share a common axes (visually) you can set the
hspace between the subplots close to zero (do not use zero itself).
Normally you'll want to turn off the tick labels on all but one of the
axes.

In this example the plots share a common xaxis but you can follow the
same logic to supply a common y axis.
"""
import matplotlib.pyplot as plt
import numpy as np
```

```
t = np.arange(0.0, 2.0, 0.01)

s1 = np.sin(2*np.pi*t)
s2 = np.exp(-t)
s3 = s1*s2

# axes rect in relative 0,1 coords left, bottom, width, height.  Turn
# off xtick labels on all but the lower plot


f = plt.figure()
plt.subplots_adjust(hspace=0.001)


ax1 = plt.subplot(311)
ax1.plot(t, s1)
plt.yticks(np.arange(-0.9, 1.0, 0.4))
plt.ylim(-1, 1)

ax2 = plt.subplot(312, sharex=ax1)
ax2.plot(t, s2)
plt.yticks(np.arange(0.1, 1.0, 0.2))
plt.ylim(0, 1)

ax3 = plt.subplot(313, sharex=ax1)
ax3.plot(t, s3)
plt.yticks(np.arange(-0.9, 1.0, 0.4))
plt.ylim(-1, 1)

xticklabels = ax1.get_xticklabels() + ax2.get_xticklabels()
plt.setp(xticklabels, visible=False)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.91 pylab_examples example code: geo_demo.py




```python
import matplotlib.pyplot as plt

plt.subplot(221, projection="aitoff")
plt.title("Aitoff")
plt.grid(True)

plt.subplot(222, projection="hammer")
plt.title("Hammer")
plt.grid(True)

plt.subplot(223, projection="lambert")
plt.title("Lambert")
plt.grid(True)

plt.subplot(224, projection="mollweide")
plt.title("Mollweide")
plt.grid(True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.92 pylab_examples example code: ginput_demo.py

[source code]

```
# -*- noplot -*-

from __future__ import print_function

import matplotlib.pyplot as plt
import numpy as np
t = np.arange(10)
plt.plot(t, np.sin(t))
print("Please click")
x = plt.ginput(3)
print("clicked", x)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.93 pylab_examples example code: ginput_manual_clabel.py

[source code]

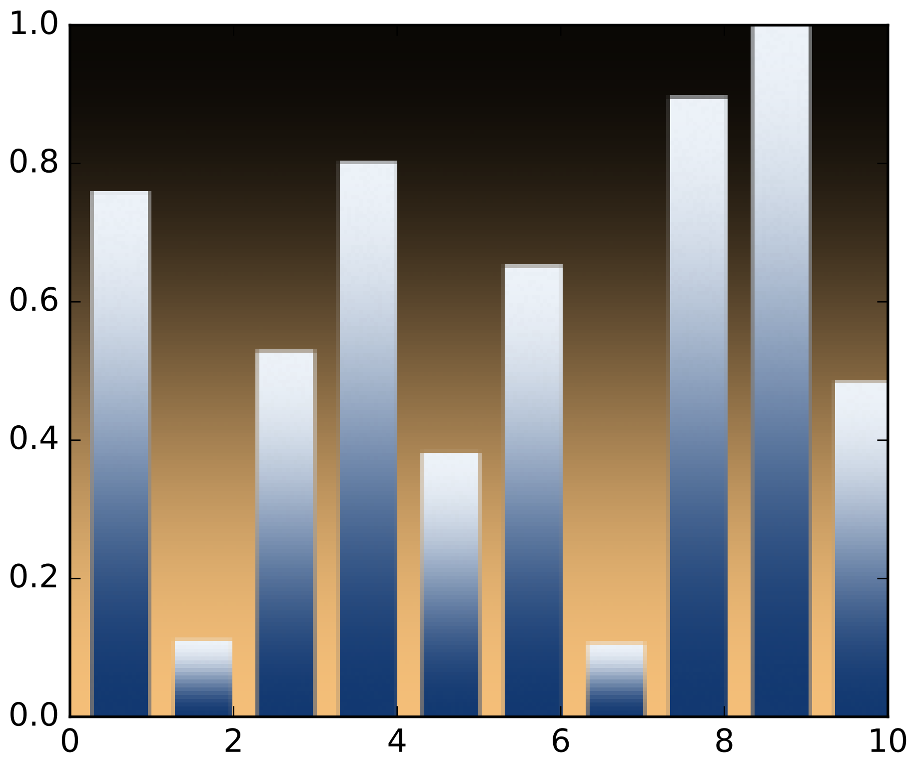
```
#!/usr/bin/env python
# -*- noplot -*-

from __future__ import print_function
"""
This provides examples of uses of interactive functions, such as ginput,
waitforbuttonpress and manual clabel placement.

This script must be run interactively using a backend that has a
graphical user interface (for example, using GTKAgg backend, but not
PS backend).

See also ginput_demo.py
"""
import time
import matplotlib
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt


def tellme(s):
    print(s)
    plt.title(s, fontsize=16)
    plt.draw()

##################################################
```

```
# Define a triangle by clicking three points
##################################################
plt.clf()
plt.axis([-1., 1., -1., 1.])
plt.setp(plt.gca(), autoscale_on=False)

tellme('You will define a triangle, click to begin')

plt.waitforbuttonpress()

happy = False
while not happy:
    pts = []
    while len(pts) < 3:
        tellme('Select 3 corners with mouse')
        pts = np.asarray(plt.ginput(3, timeout=-1))
        if len(pts) < 3:
            tellme('Too few points, starting over')
            time.sleep(1)  # Wait a second

    ph = plt.fill(pts[:, 0], pts[:, 1], 'r', lw=2)

    tellme('Happy? Key click for yes, mouse click for no')

    happy = plt.waitforbuttonpress()

    # Get rid of fill
    if not happy:
        for p in ph:
            p.remove()

##################################################
# Now contour according to distance from triangle
# corners - just an example
##################################################


# Define a nice function of distance from individual pts
def f(x, y, pts):
    z = np.zeros_like(x)
    for p in pts:
        z = z + 1/(np.sqrt((x - p[0])**2 + (y - p[1])**2))
    return 1/z

X, Y = np.meshgrid(np.linspace(-1, 1, 51), np.linspace(-1, 1, 51))
Z = f(X, Y, pts)

CS = plt.contour(X, Y, Z, 20)

tellme('Use mouse to select contour label locations, middle button to finish')
CL = plt.clabel(CS, manual=True)

##################################################
```

```
# Now do a zoom
##################################################
tellme('Now do a nested zoom, click to begin')
plt.waitforbuttonpress()

happy = False
while not happy:
    tellme('Select two corners of zoom, middle mouse button to finish')
    pts = np.asarray(plt.ginput(2, timeout=-1))

    happy = len(pts) < 2
    if happy:
        break

    pts = np.sort(pts, axis=0)
    plt.axis(pts.T.ravel())

tellme('All Done!')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

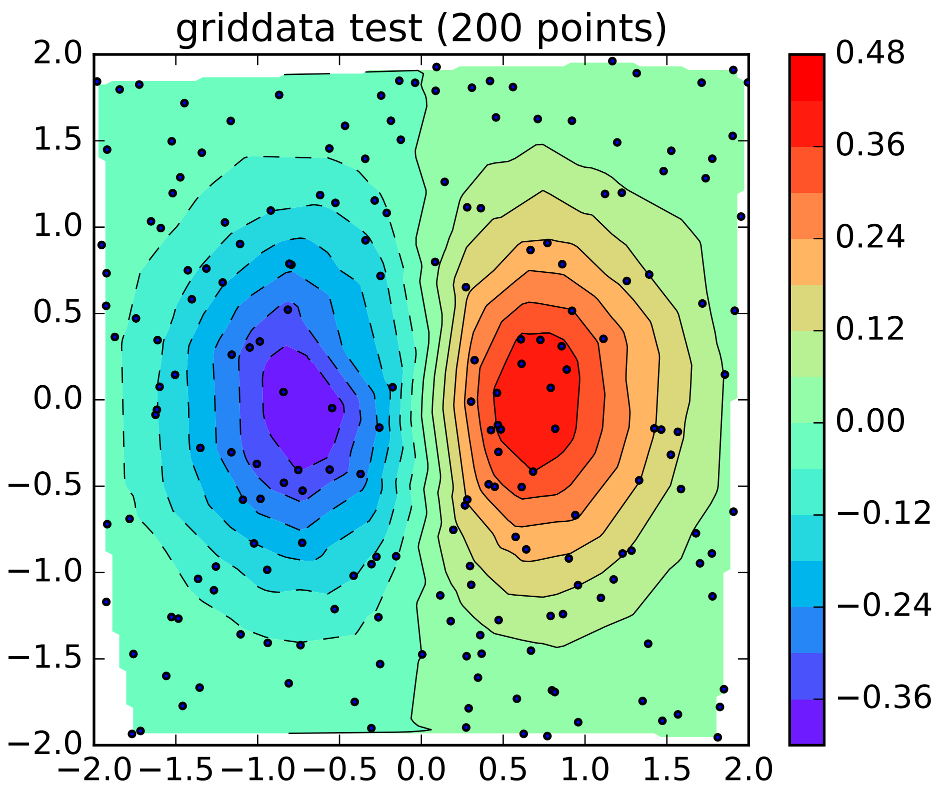
90.94 pylab_examples example code: gradient_bar.py



```
from matplotlib.pyplot import figure, show, cm
from numpy import arange
from numpy.random import rand


def gbar(ax, x, y, width=0.5, bottom=0):
    X = [[.6, .6], [.7, .7]]
    for left, top in zip(x, y):
        right = left + width
        ax.imshow(X, interpolation='bicubic', cmap=cm.Blues,
                  extent=(left, right, bottom, top), alpha=1)

fig = figure()

xmin, xmax = xlim = 0, 10
ymin, ymax = ylim = 0, 1
ax = fig.add_subplot(111, xlim=xlim, ylim=ylim,
                     autoscale_on=False)
X = [[.6, .6], [.7, .7]]

ax.imshow(X, interpolation='bicubic', cmap=cm.copper,
          extent=(xmin, xmax, ymin, ymax), alpha=1)

N = 10
x = arange(N) + 0.25
y = rand(N)
gbar(ax, x, y, width=0.7)
ax.set_aspect('auto')
show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.95 pylab_examples example code: griddata_demo.py

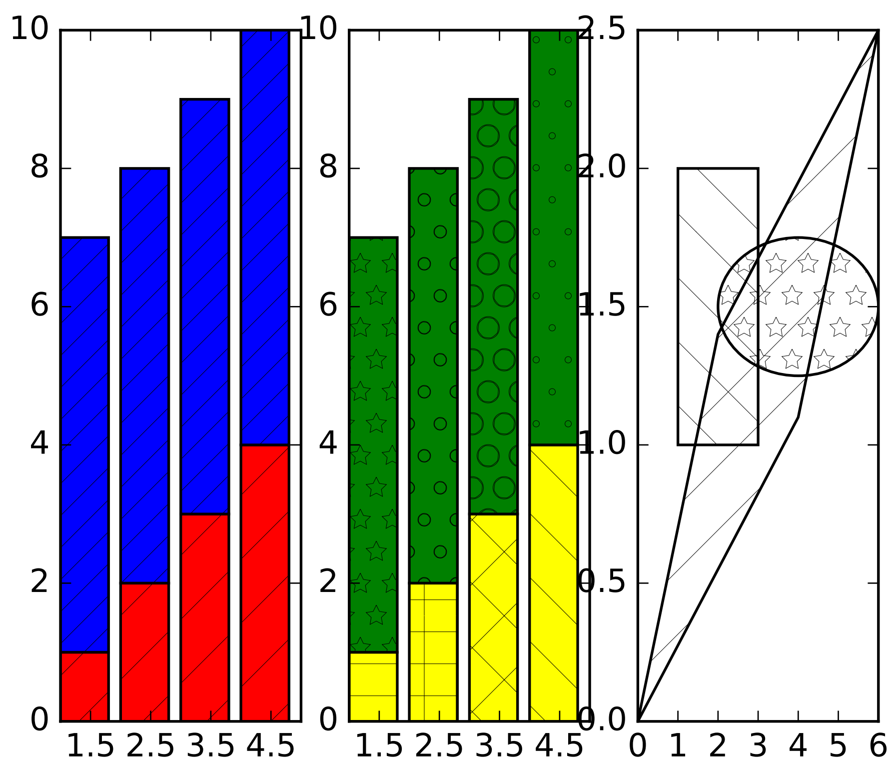



```
from numpy.random import uniform, seed
from matplotlib.mlab import griddata
import matplotlib.pyplot as plt
import numpy as np
# make up data.
#npts = int(raw_input('enter # of random points to plot:'))
seed(0)
npts = 200
x = uniform(-2, 2, npts)
y = uniform(-2, 2, npts)
z = x*np.exp(-x**2 - y**2)
# define grid.
xi = np.linspace(-2.1, 2.1, 100)
yi = np.linspace(-2.1, 2.1, 200)
# grid the data.
zi = griddata(x, y, z, xi, yi, interp='linear')
# contour the gridded data, plotting dots at the nonuniform data points.
CS = plt.contour(xi, yi, zi, 15, linewidths=0.5, colors='k')
CS = plt.contourf(xi, yi, zi, 15, cmap=plt.cm.rainbow,
                  vmax=abs(zi).max(), vmin=-abs(zi).max())
plt.colorbar()  # draw colorbar
# plot data points.
```

```
plt.scatter(x, y, marker='o', c='b', s=5, zorder=10)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('griddata test (%d points)' % npts)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.96 pylab_examples example code: hatch_demo.py

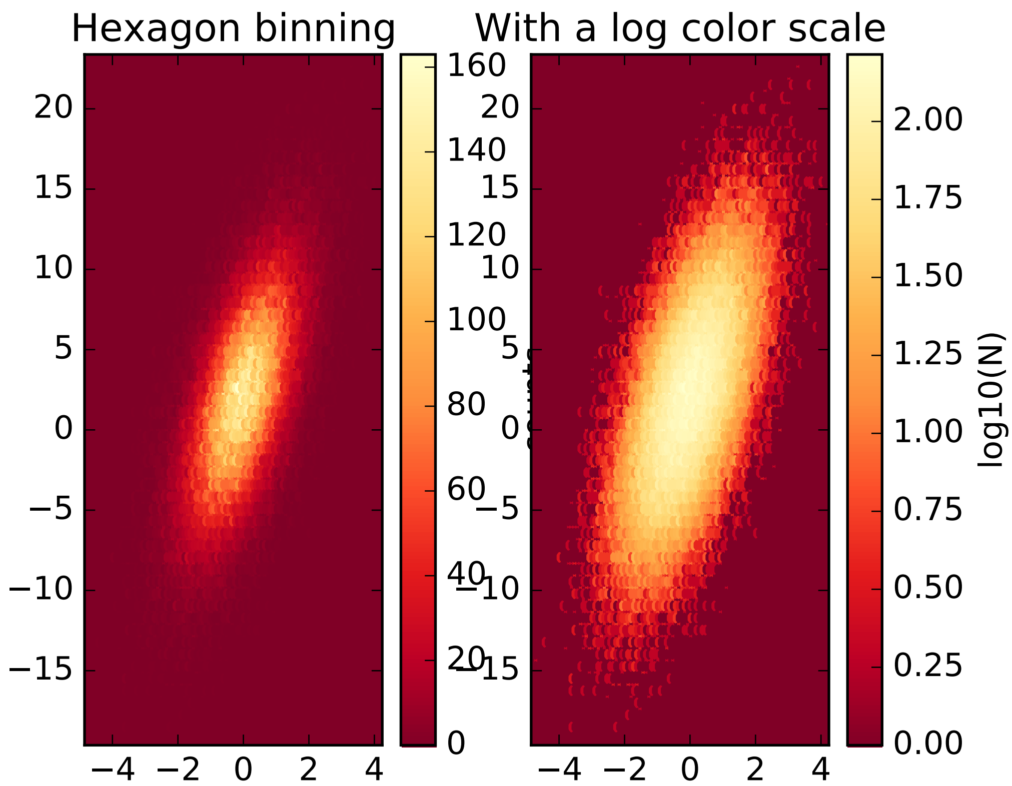



```
"""
Hatching (pattern filled polygons) is supported currently in the PS,
PDF, SVG and Agg backends only.
"""
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse, Polygon

fig = plt.figure()
ax1 = fig.add_subplot(131)
ax1.bar(range(1, 5), range(1, 5), color='red', edgecolor='black', hatch="/")
ax1.bar(range(1, 5), [6] * 4, bottom=range(1, 5), color='blue', edgecolor='black', hatch=
↪'//')
```

```
ax1.set_xticks([1.5, 2.5, 3.5, 4.5])

ax2 = fig.add_subplot(132)
bars = ax2.bar(range(1, 5), range(1, 5), color='yellow', ecolor='black') + \
    ax2.bar(range(1, 5), [6] * 4, bottom=range(1, 5), color='green', ecolor='black')
ax2.set_xticks([1.5, 2.5, 3.5, 4.5])

patterns = ('-', '+', 'x', '\\', '*', 'o', 'O', '.')
for bar, pattern in zip(bars, patterns):
    bar.set_hatch(pattern)

ax3 = fig.add_subplot(133)
ax3.fill([1, 3, 3, 1], [1, 1, 2, 2], fill=False, hatch='\\')
ax3.add_patch(Ellipse((4, 1.5), 4, 0.5, fill=False, hatch='*'))
ax3.add_patch(Polygon([[0, 0], [4, 1.1], [6, 2.5], [2, 1.4]], closed=True,
                      fill=False, hatch='/'))
ax3.set_xlim((0, 6))
ax3.set_ylim((0, 2.5))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.97 pylab_examples example code: hexbin_demo.py

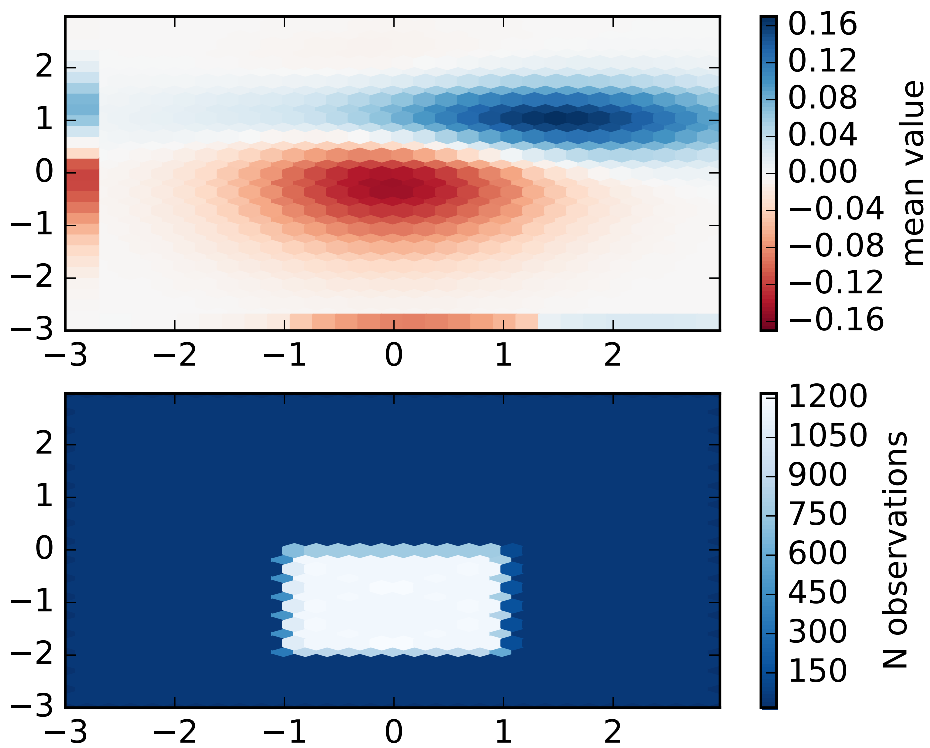



```
"""
hexbin is an axes method or pyplot function that is essentially
a pcolor of a 2-D histogram with hexagonal cells.  It can be
much more informative than a scatter plot; in the first subplot
below, try substituting 'scatter' for 'hexbin'.
"""

import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
n = 100000
x = np.random.standard_normal(n)
y = 2.0 + 3.0 * x + 4.0 * np.random.standard_normal(n)
xmin = x.min()
xmax = x.max()
ymin = y.min()
ymax = y.max()

plt.subplots_adjust(hspace=0.5)
plt.subplot(121)
plt.hexbin(x, y, cmap=plt.cm.YlOrRd_r)
```

```
plt.axis([xmin, xmax, ymin, ymax])
plt.title("Hexagon binning")
cb = plt.colorbar()
cb.set_label('counts')

plt.subplot(122)
plt.hexbin(x, y, bins='log', cmap=plt.cm.YlOrRd_r)
plt.axis([xmin, xmax, ymin, ymax])
plt.title("With a log color scale")
cb = plt.colorbar()
cb.set_label('log10(N)')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.98 pylab_examples example code: hexbin_demo2.py

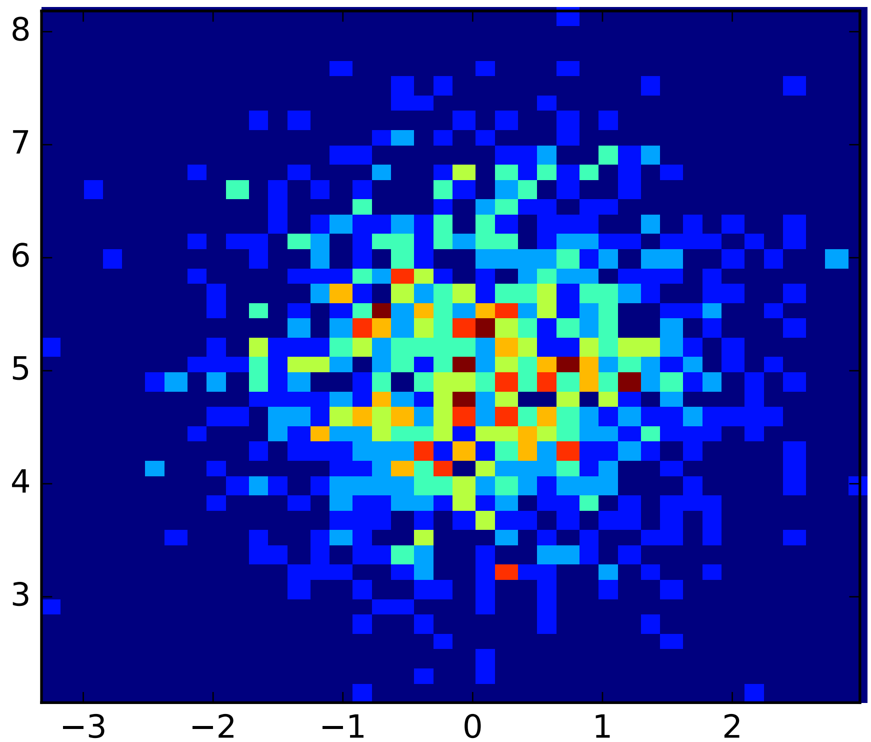



```
"""
hexbin is an axes method or pyplot function that is essentially a
pcolor of a 2-D histogram with hexagonal cells.
"""
```

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2 - Z1  # difference of Gaussians

x = X.ravel()
y = Y.ravel()
z = Z.ravel()

if 1:
    # make some points 20 times more common than others, but same mean
    xcond = (-1 < x) & (x < 1)
    ycond = (-2 < y) & (y < 0)
    cond = xcond & ycond
    xnew = x[cond]
    ynew = y[cond]
    znew = z[cond]
    for i in range(20):
        x = np.hstack((x, xnew))
        y = np.hstack((y, ynew))
        z = np.hstack((z, znew))

xmin = x.min()
xmax = x.max()
ymin = y.min()
ymax = y.max()

gridsize = 30

plt.subplot(211)
plt.hexbin(x, y, C=z, gridsize=gridsize, marginals=True, cmap=plt.cm.RdBu,
           vmax=abs(z).max(), vmin=-abs(z).max())
plt.axis([xmin, xmax, ymin, ymax])
cb = plt.colorbar()
cb.set_label('mean value')


plt.subplot(212)
plt.hexbin(x, y, gridsize=gridsize, cmap=plt.cm.Blues_r)
plt.axis([xmin, xmax, ymin, ymax])
cb = plt.colorbar()
cb.set_label('N observations')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

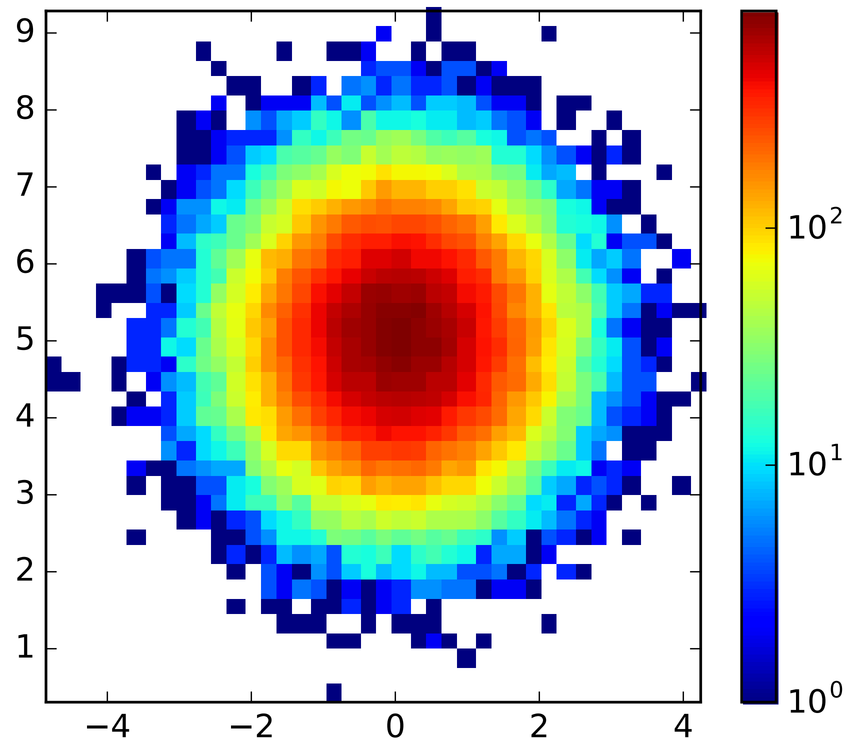
90.99 pylab_examples example code: hist2d_demo.py




```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.randn(1000)
y = np.random.randn(1000) + 5

# normal distribution center at x=0 and y=5
plt.hist2d(x, y, bins=40)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

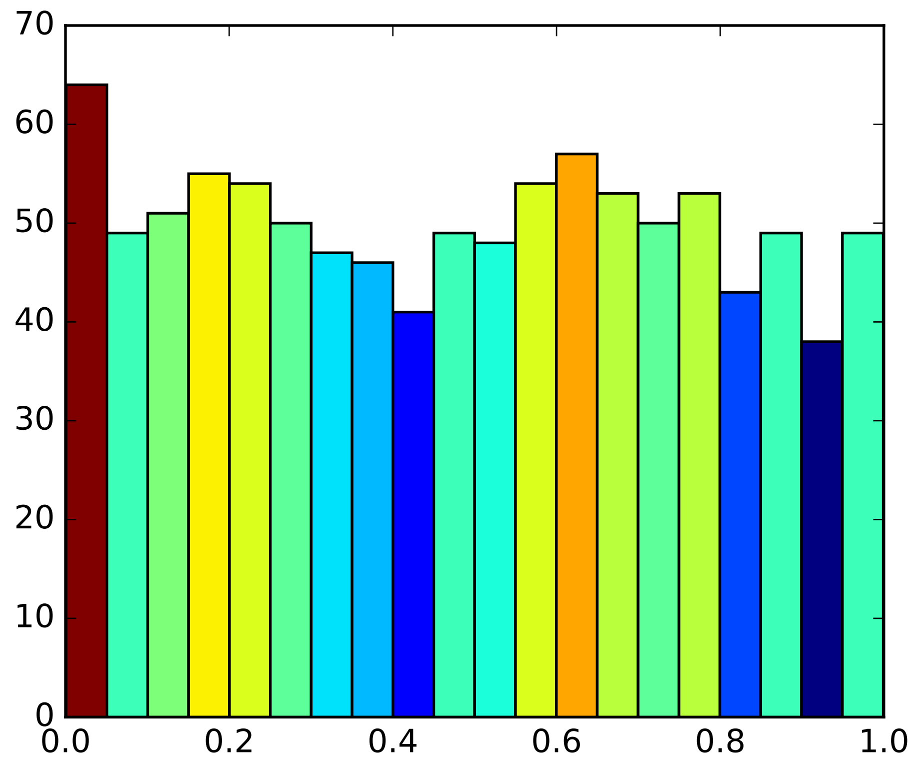
90.100 pylab_examples example code: hist2d_log_demo.py




```
from matplotlib.colors import LogNorm
import matplotlib.pyplot as plt
import numpy as np

# normal distribution center at x=0 and y=5
x = np.random.randn(100000)
y = np.random.randn(100000) + 5

plt.hist2d(x, y, bins=40, norm=LogNorm())
plt.colorbar()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.101 pylab_examples example code: hist_colormapped.py

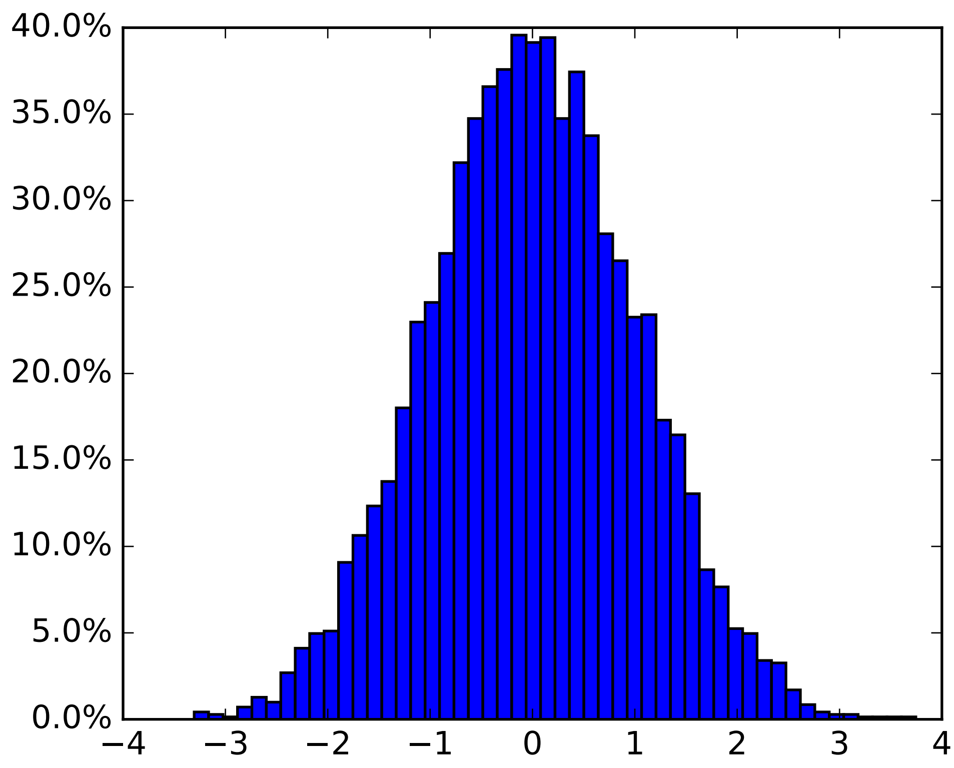



```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as colors

fig, ax = plt.subplots()
Ntotal = 1000
N, bins, patches = ax.hist(np.random.rand(Ntotal), 20)

# I'll color code by height, but you could use any scalar


# we need to normalize the data to 0..1 for the full
# range of the colormap
fracs = N.astype(float)/N.max()
norm = colors.Normalize(fracs.min(), fracs.max())

for thisfrac, thispatch in zip(fracs, patches):
    color = cm.jet(norm(thisfrac))
    thispatch.set_facecolor(color)
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.102 pylab_examples example code: histogram_percent_demo.py




```
import matplotlib
from numpy.random import randn
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter


def to_percent(y, position):
    # Ignore the passed in position. This has the effect of scaling the default
    # tick locations.
    s = str(100 * y)

    # The percent symbol needs escaping in latex
    if matplotlib.rcParams['text.usetex'] is True:
        return s + r'$\%$'
    else:
        return s + '%'
```

```
x = randn(5000)

# Make a normed histogram. It'll be multiplied by 100 later.
plt.hist(x, bins=50, normed=True)

# Create the formatter using the function to_percent. This multiplies all the
# default labels by 100, making them all percentages
formatter = FuncFormatter(to_percent)

# Set the formatter
plt.gca().yaxis.set_major_formatter(formatter)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.103 pylab_examples example code: hyperlinks.py

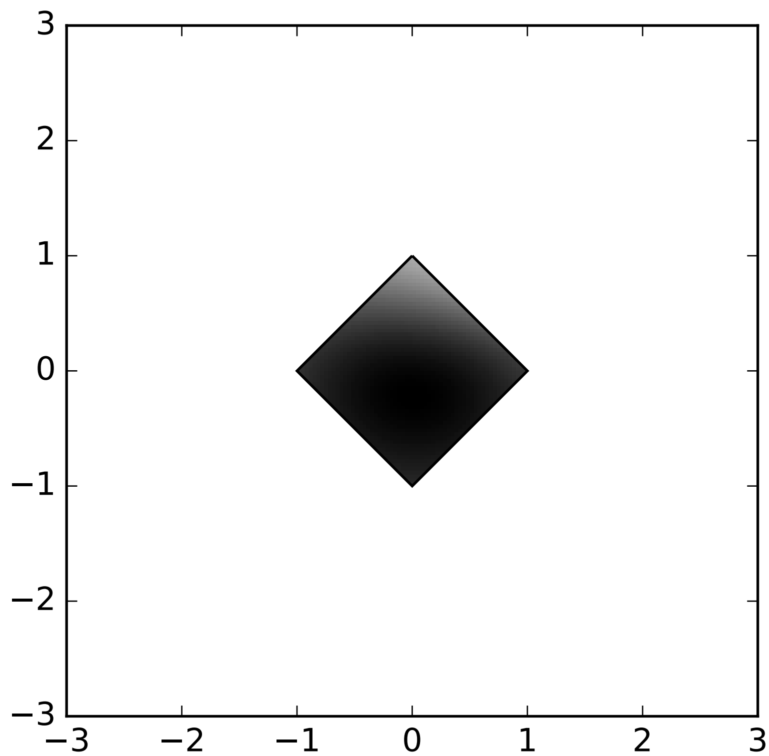
[source code]

```
#!/usr/bin/env python
# -*- noplot -*-

"""
This example demonstrates how to set a hyperlinks on various kinds of elements.

This currently only works with the SVG backend.
"""

import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

f = plt.figure()
s = plt.scatter([1, 2, 3], [4, 5, 6])
s.set_urls(['http://www.bbc.co.uk/news', 'http://www.google.com', None])
f.canvas.print_figure('scatter.svg')

f = plt.figure()
delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2 - Z1  # difference of Gaussians

im = plt.imshow(Z, interpolation='bilinear', cmap=cm.gray,
                origin='lower', extent=[-3, 3, -3, 3])
```

```
im.set_url('http://www.google.com')
f.canvas.print_figure('image.svg')
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.104 pylab_examples example code: image_clip_path.py

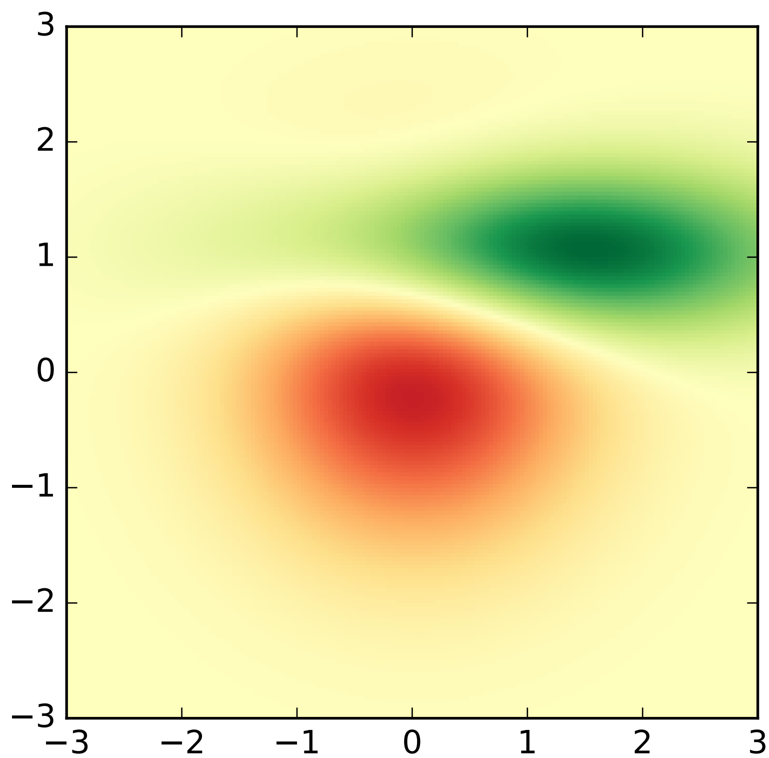



```
#!/usr/bin/env python
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
from matplotlib.path import Path
from matplotlib.patches import PathPatch

delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2 - Z1  # difference of Gaussians
```

```
path = Path([[0, 1], [1, 0], [0, -1], [-1, 0], [0, 1]])
patch = PathPatch(path, facecolor='none')
plt.gca().add_patch(patch)

im = plt.imshow(Z, interpolation='bilinear', cmap=cm.gray,
                origin='lower', extent=[-3, 3, -3, 3],
                clip_path=patch, clip_on=True)
im.set_clip_path(patch)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.105 pylab_examples example code: image_demo.py

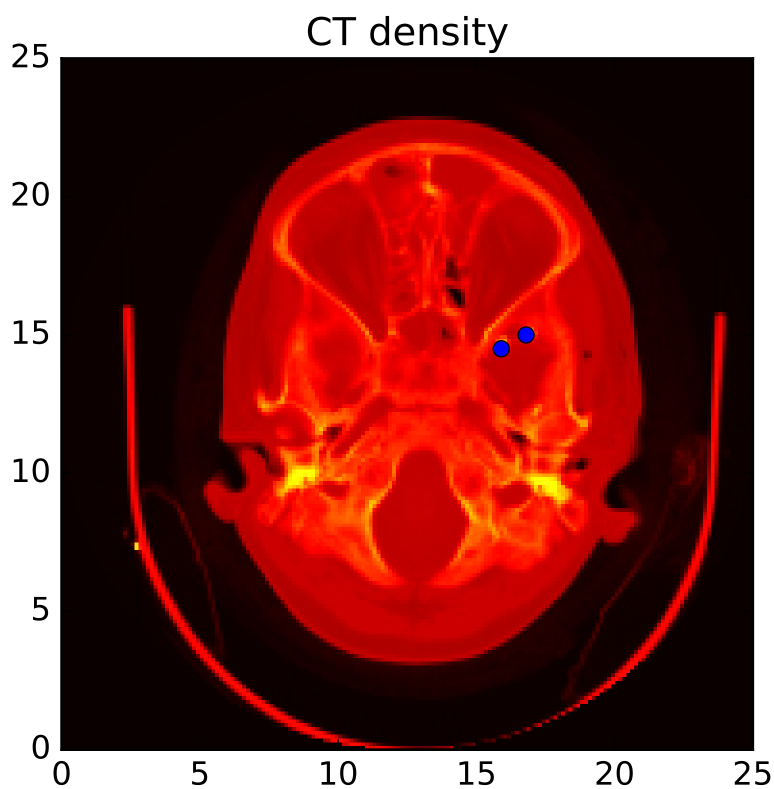



```
#!/usr/bin/env python
import numpy as np
import matplotlib.cm as cm
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

delta = 0.025
```

```
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = Z2 - Z1  # difference of Gaussians

im = plt.imshow(Z, interpolation='bilinear', cmap=cm.RdYlGn,
                origin='lower', extent=[-3, 3, -3, 3],
                vmax=abs(Z).max(), vmin=-abs(Z).max())

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.106 pylab_examples example code: image_demo2.py

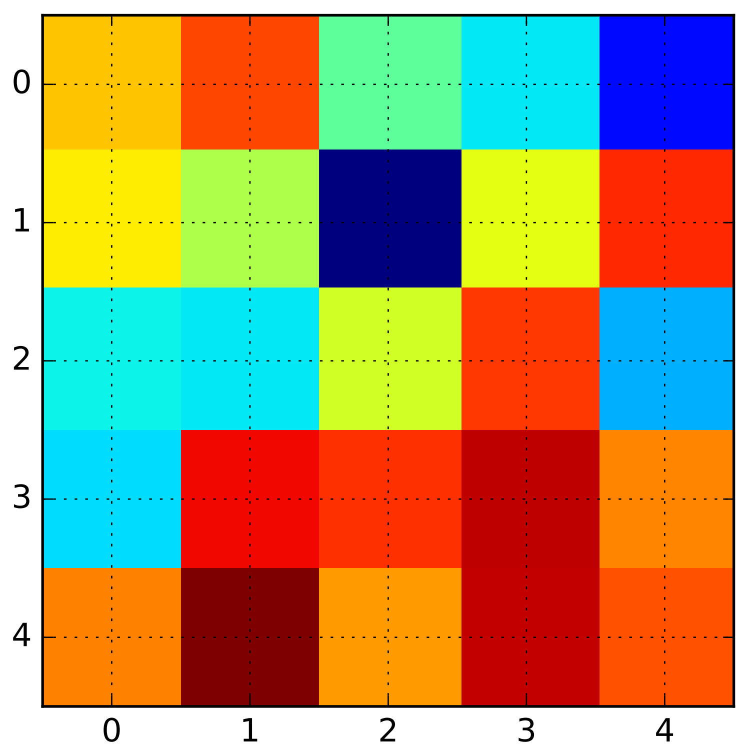


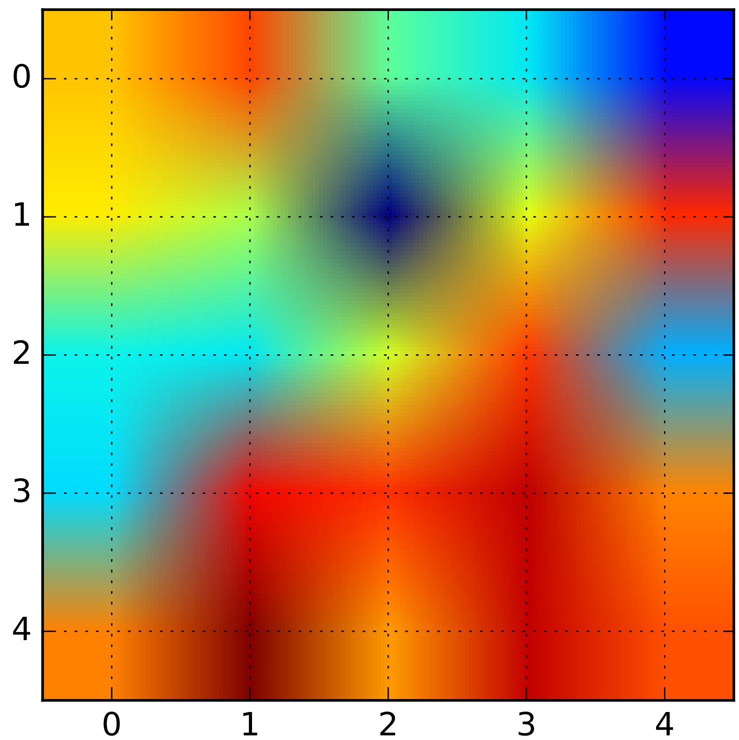

```
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.cbook as cbook

w, h = 512, 512
```

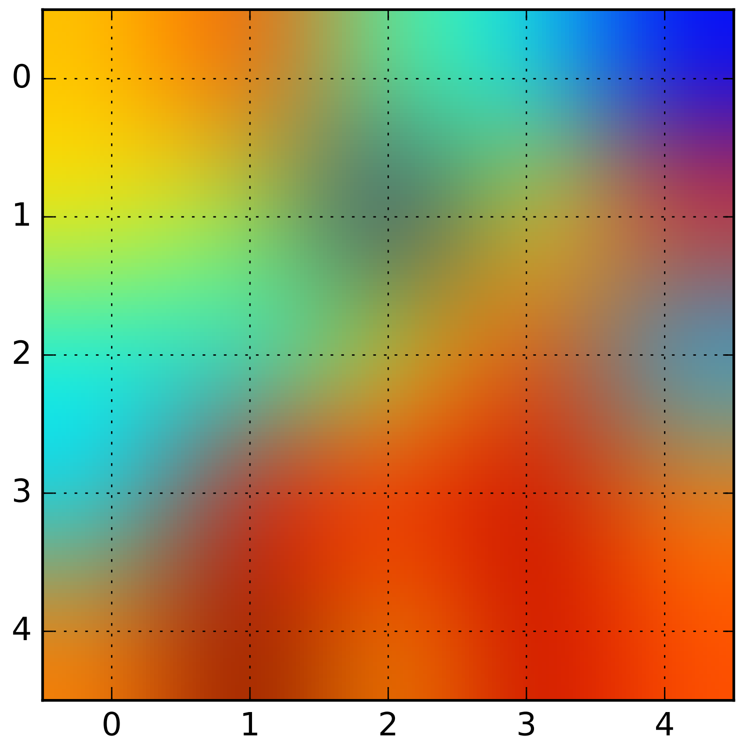
```
datafile = cbook.get_sample_data('ct.raw.gz', asfileobj=True)
s = datafile.read()
A = np.fromstring(s, np.uint16).astype(float)
A *= 1.0 / max(A)
A.shape = w, h

extent = (0, 25, 0, 25)
im = plt.imshow(A, cmap=plt.cm.hot, origin='upper', extent=extent)

markers = [(15.9, 14.5), (16.8, 15)]
x, y = zip(*markers)
plt.plot(x, y, 'o')

plt.title('CT density')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.107 pylab_examples example code: image_interp.py




0
1
2
3
4
0
1
2
3
4

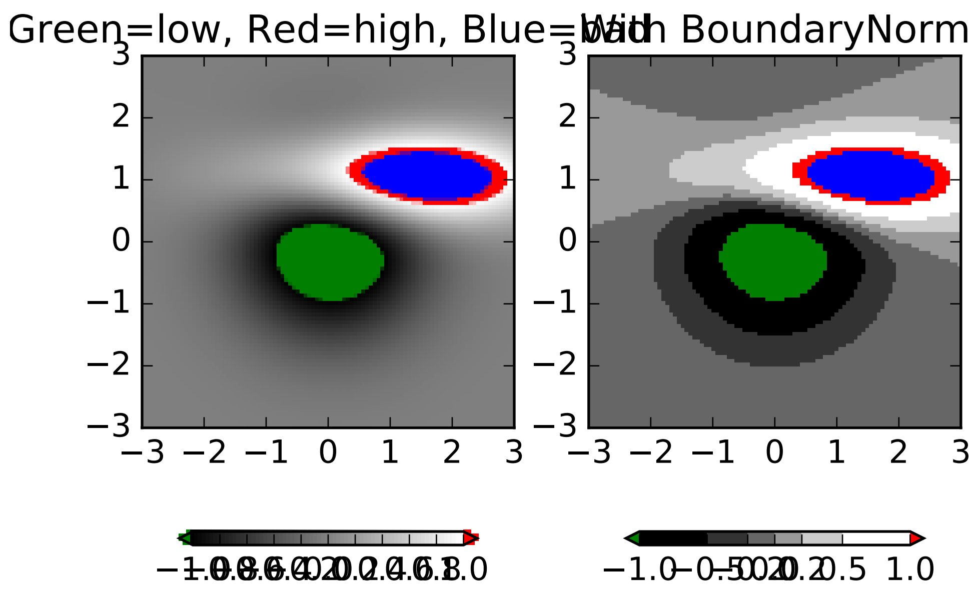



```
"""
The same (small) array, interpolated with three different
interpolation methods.

The center of the pixel at A[i,j] is plotted at i+0.5, i+0.5.  If you
are using interpolation='nearest', the region bounded by (i,j) and
(i+1,j+1) will have the same color.  If you are using interpolation,
the pixel center will have the same color as it does with nearest, but
other pixels will be interpolated between the neighboring pixels.

Earlier versions of matplotlib (<0.63) tried to hide the edge effects
from you by setting the view limits so that they would not be visible.
A recent bugfix in antigrain, and a new implementation in the
matplotlib._image module which takes advantage of this fix, no longer
makes this necessary.  To prevent edge effects, when doing
interpolation, the matplotlib._image module now pads the input array
with identical pixels around the edge.  e.g., if you have a 5x5 array
with colors a-y as below


  a b c d e
  f g h i j
  k l m n o
  p q r s t
  u v w x y
```

```
the _image module creates the padded array,

  a a b c d e e
  a a b c d e e
  f f g h i j j
  k k l m n o o
  p p q r s t t
  o u v w x y y
  o u v w x y y

does the interpolation/resizing, and then extracts the central region.
This allows you to plot the full range of your array w/o edge effects,
and for example to layer multiple images of different sizes over one
another with different interpolation methods - see
examples/layer_images.py.  It also implies a performance hit, as this
new temporary, padded array must be created.  Sophisticated
interpolation also implies a performance hit, so if you need maximal
performance or have very large images, interpolation='nearest' is
suggested.

"""
import matplotlib.pyplot as plt
import numpy as np

A = np.random.rand(5, 5)
plt.figure(1)
plt.imshow(A, interpolation='nearest')
plt.grid(True)

plt.figure(2)
plt.imshow(A, interpolation='bilinear')
plt.grid(True)

plt.figure(3)
plt.imshow(A, interpolation='bicubic')
plt.grid(True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.108 pylab_examples example code: image_masked.py

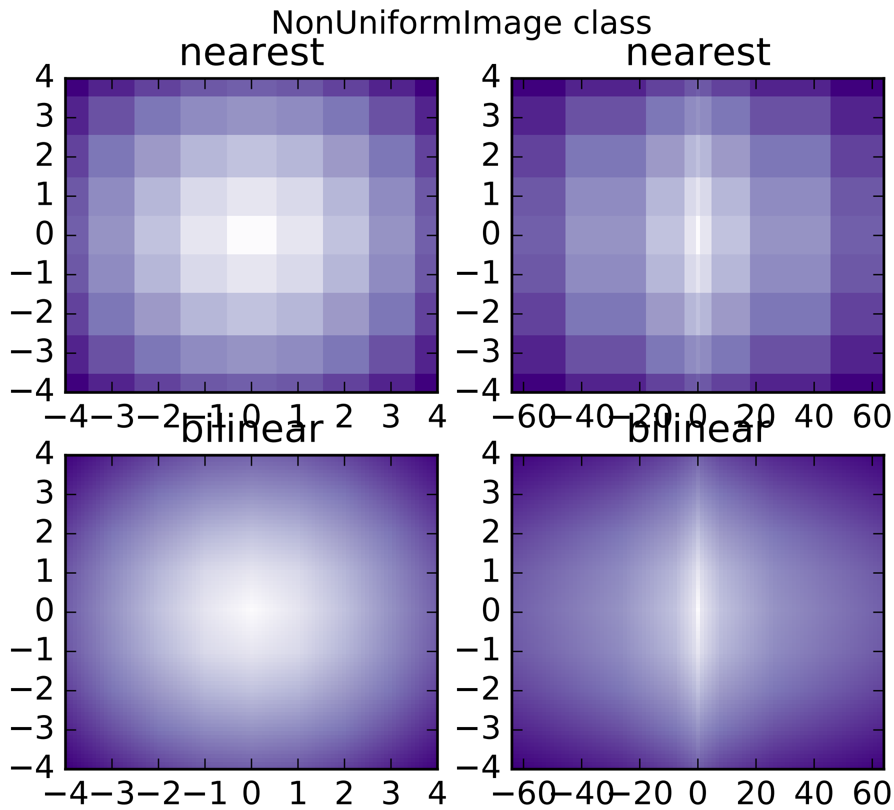



```
"""
imshow with masked array input and out-of-range colors.

The second subplot illustrates the use of BoundaryNorm to
get a filled contour effect.
"""

from numpy import ma
import matplotlib.colors as colors
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import numpy as np

delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y)
Z1 = mlab.bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = mlab.bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = 10*(Z2 - Z1)  # difference of Gaussians

# Set up a colormap:
palette = plt.cm.gray
```

```
palette.set_over('r', 1.0)
palette.set_under('g', 1.0)
palette.set_bad('b', 1.0)
# Alternatively, we could use
# palette.set_bad(alpha = 0.0)
# to make the bad region transparent.  This is the default.
# If you comment out all the palette.set* lines, you will see
# all the defaults; under and over will be colored with the
# first and last colors in the palette, respectively.
Zm = ma.masked_where(Z > 1.2, Z)

# By setting vmin and vmax in the norm, we establish the
# range to which the regular palette color scale is applied.
# Anything above that range is colored based on palette.set_over, etc.

plt.subplot(1, 2, 1)
im = plt.imshow(Zm, interpolation='bilinear',
                cmap=palette,
                norm=colors.Normalize(vmin=-1.0, vmax=1.0, clip=False),
                origin='lower', extent=[-3, 3, -3, 3])
plt.title('Green=low, Red=high, Blue=bad')
plt.colorbar(im, extend='both', orientation='horizontal', shrink=0.8)

plt.subplot(1, 2, 2)
im = plt.imshow(Zm, interpolation='nearest',
                cmap=palette,
                norm=colors.BoundaryNorm([-1, -0.5, -0.2, 0, 0.2, 0.5, 1],
                                         ncolors=256, clip=False),
                origin='lower', extent=[-3, 3, -3, 3])
plt.title('With BoundaryNorm')
plt.colorbar(im, extend='both', spacing='proportional',
             orientation='horizontal', shrink=0.8)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.109 pylab_examples example code: image_nonuniform.py

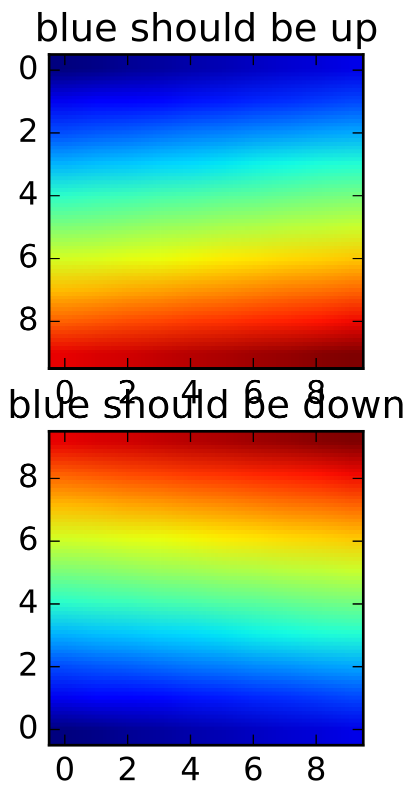



```
'''
This illustrates the NonUniformImage class, which still needs
an axes method interface; either a separate interface, or a
generalization of imshow.
'''

from matplotlib.pyplot import figure, show
import numpy as np
from matplotlib.image import NonUniformImage
from matplotlib import cm

interp = 'nearest'

x = np.linspace(-4, 4, 9)
x2 = x**3
y = np.linspace(-4, 4, 9)
#print('Size %d points' % (len(x) * len(y)))
z = np.sqrt(x[np.newaxis, :]**2 + y[:, np.newaxis]**2)

fig = figure()
fig.suptitle('NonUniformImage class')
ax = fig.add_subplot(221)
```

```
im = NonUniformImage(ax, interpolation=interp, extent=(-4, 4, -4, 4),
                     cmap=cm.Purples)
im.set_data(x, y, z)
ax.images.append(im)
ax.set_xlim(-4, 4)
ax.set_ylim(-4, 4)
ax.set_title(interp)

ax = fig.add_subplot(222)
im = NonUniformImage(ax, interpolation=interp, extent=(-64, 64, -4, 4),
                     cmap=cm.Purples)
im.set_data(x2, y, z)
ax.images.append(im)
ax.set_xlim(-64, 64)
ax.set_ylim(-4, 4)
ax.set_title(interp)

interp = 'bilinear'

ax = fig.add_subplot(223)
im = NonUniformImage(ax, interpolation=interp, extent=(-4, 4, -4, 4),
                     cmap=cm.Purples)
im.set_data(x, y, z)
ax.images.append(im)
ax.set_xlim(-4, 4)
ax.set_ylim(-4, 4)
ax.set_title(interp)

ax = fig.add_subplot(224)
im = NonUniformImage(ax, interpolation=interp, extent=(-64, 64, -4, 4),
                     cmap=cm.Purples)
im.set_data(x2, y, z)
ax.images.append(im)
ax.set_xlim(-64, 64)
ax.set_ylim(-4, 4)
ax.set_title(interp)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.110 pylab_examples example code: image_origin.py

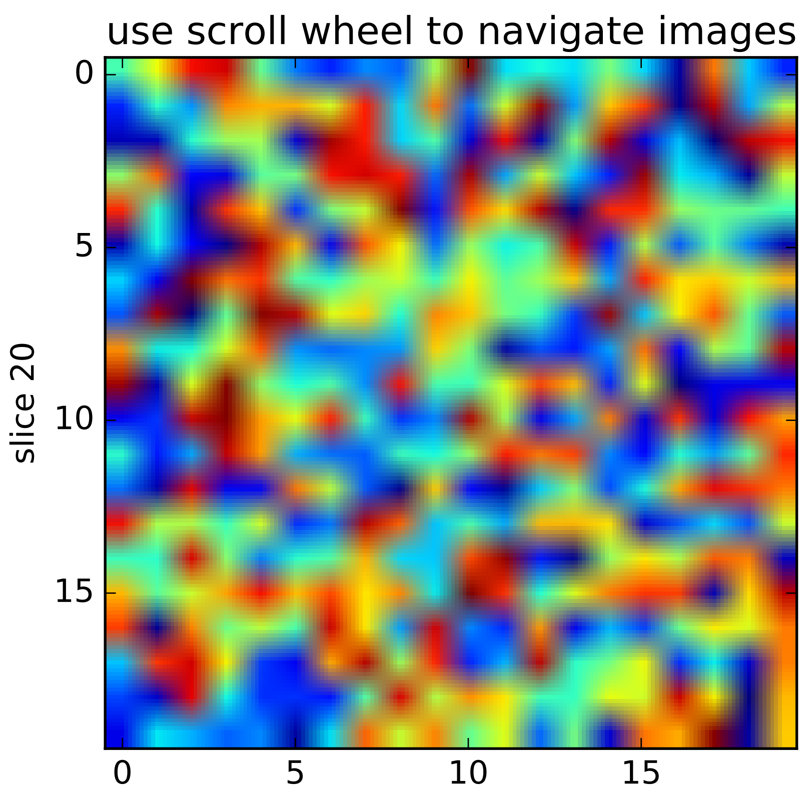



```
"""
You can specify whether images should be plotted with the array origin
x[0,0] in the upper left or upper right by using the origin parameter.
You can also control the default be setting image.origin in your
matplotlibrc file; see http://matplotlib.org/matplotlibrc
"""
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(100.0)
x.shape = (10, 10)

interp = 'bilinear'
#interp = 'nearest'
lim = -2, 11, -2, 6
plt.subplot(211, axisbg='g')
plt.title('blue should be up')
plt.imshow(x, origin='upper', interpolation=interp, cmap='jet')
#plt.axis(lim)

plt.subplot(212, axisbg='y')
plt.title('blue should be down')
```

```
plt.imshow(x, origin='lower', interpolation=interp, cmap='jet')
#plt.axis(lim)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.111 pylab_examples example code: image_slices_viewer.py

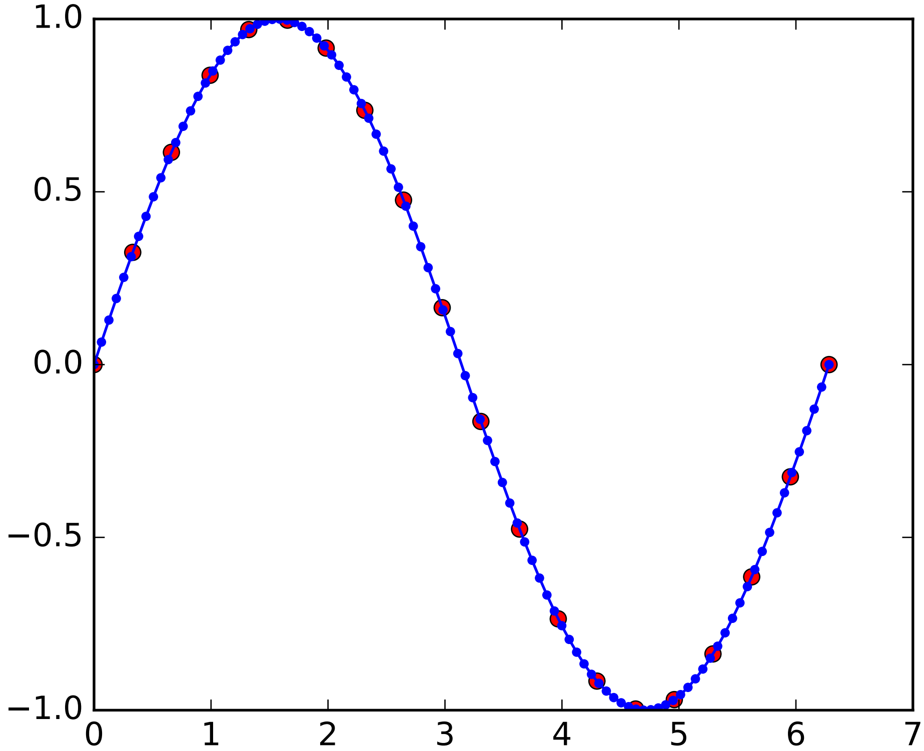



```
from __future__ import print_function
import numpy
from matplotlib.pyplot import figure, show


class IndexTracker(object):
    def __init__(self, ax, X):
        self.ax = ax
        ax.set_title('use scroll wheel to navigate images')

        self.X = X
        rows, cols, self.slices = X.shape
        self.ind = self.slices//2
```

```
        self.im = ax.imshow(self.X[:, :, self.ind])
        self.update()

    def onscroll(self, event):
        print("%s %s" % (event.button, event.step))
        if event.button == 'up':
            self.ind = numpy.clip(self.ind + 1, 0, self.slices - 1)
        else:
            self.ind = numpy.clip(self.ind - 1, 0, self.slices - 1)

        self.update()

    def update(self):
        self.im.set_data(self.X[:, :, self.ind])
        ax.set_ylabel('slice %s' % self.ind)
        self.im.axes.figure.canvas.draw()


fig = figure()
ax = fig.add_subplot(111)

X = numpy.random.rand(20, 20, 40)

tracker = IndexTracker(ax, X)


fig.canvas.mpl_connect('scroll_event', tracker.onscroll)
show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

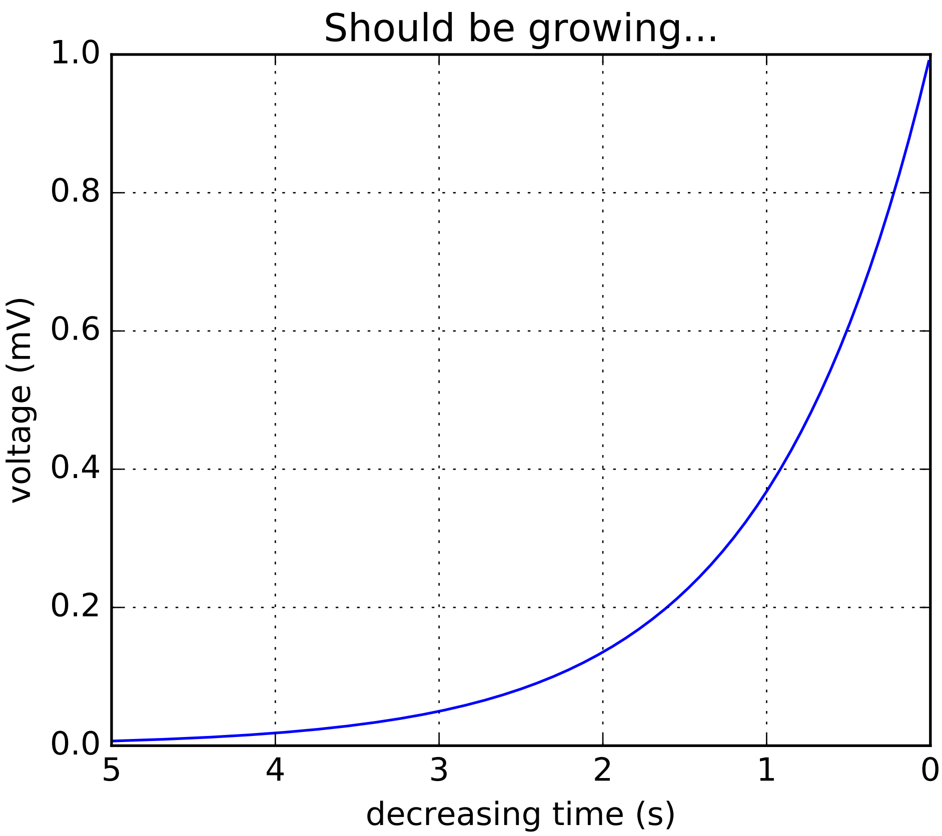
90.112 pylab_examples example code: interp_demo.py




```
import matplotlib.pyplot as plt
from numpy import pi, sin, linspace
from matplotlib.mlab import stineman_interp

x = linspace(0, 2*pi, 20)
y = sin(x)
yp = None
xi = linspace(x[0], x[-1], 100)
yi = stineman_interp(xi, x, y, yp)

fig, ax = plt.subplots()
ax.plot(x, y, 'ro', xi, yi, '-b.')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

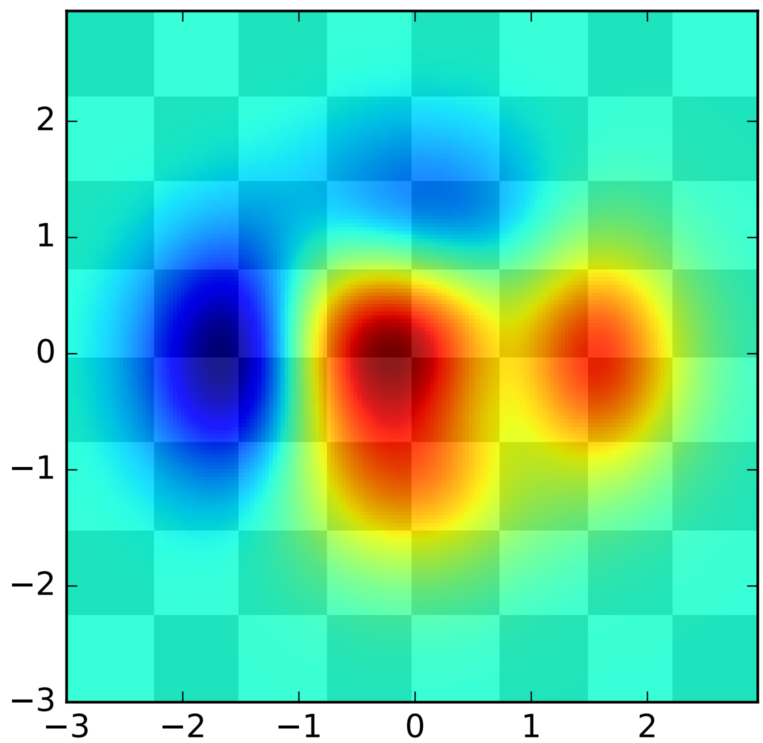
90.113 pylab_examples example code: invert_axes.py




```
"""
You can use decreasing axes by flipping the normal order of the axis
limits
"""

import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.01, 5.0, 0.01)
s = np.exp(-t)
plt.plot(t, s)

plt.xlim(5, 0)  # decreasing time

plt.xlabel('decreasing time (s)')
plt.ylabel('voltage (mV)')
plt.title('Should be growing...')
plt.grid(True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.114 pylab_examples example code: layer_images.py

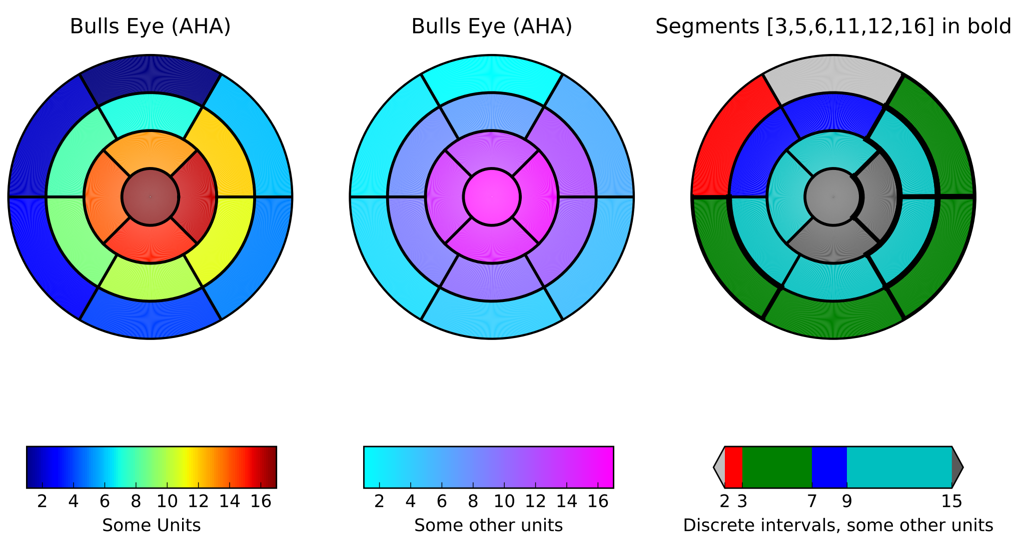



```
"""
Layer images above one another using alpha blending
"""
from __future__ import division
import matplotlib.pyplot as plt
import numpy as np


def func3(x, y):
    return (1 - x/2 + x**5 + y**3)*np.exp(-(x**2 + y**2))

# make these smaller to increase the resolution
dx, dy = 0.05, 0.05

x = np.arange(-3.0, 3.0, dx)
y = np.arange(-3.0, 3.0, dy)
X, Y = np.meshgrid(x, y)

# when layering multiple images, the images need to have the same
# extent.  This does not mean they need to have the same shape, but
# they both need to render to the same coordinate system determined by
# xmin, xmax, ymin, ymax.  Note if you use different interpolations
```

```
# for the images their apparent extent could be different due to
# interpolation edge effects


xmin, xmax, ymin, ymax = np.amin(x), np.amax(x), np.amin(y), np.amax(y)
extent = xmin, xmax, ymin, ymax
fig = plt.figure(frameon=False)

Z1 = np.array(([0, 1]*4 + [1, 0]*4)*4)
Z1.shape = (8, 8)  # chessboard
im1 = plt.imshow(Z1, cmap=plt.cm.gray, interpolation='nearest',
                 extent=extent)
plt.hold(True)

Z2 = func3(X, Y)

im2 = plt.imshow(Z2, cmap=plt.cm.jet, alpha=.9, interpolation='bilinear',
                 extent=extent)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.115 pylab_examples example code: leftventricle_bulleye.py



```
#!/usr/bin/env python
"""
This example demonstrates how to create the 17 segment model for the left
ventricle recommended by the American Heart Association (AHA).
"""

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt


def bullseye_plot(ax, data, segBold=None, cmap=None, norm=None):
    """
    Bullseye representation for the left ventricle.

    Parameters
    ----------
    ax : axes
    data : list of int and float
        The intensity values for each of the 17 segments
    segBold: list of int, optional
        A list with the segments to highlight
    cmap : ColorMap or None, optional
        Optional argument to set the desired colormap
    norm : Normalize or None, optional
        Optional argument to normalize data into the [0.0, 1.0] range


    Notes
    -----
    This function create the 17 segment model for the left ventricle according
    to the American Heart Association (AHA) [1]_

    References
    ----------
    .. [1] M. D. Cerqueira, N. J. Weissman, V. Dilsizian, A. K. Jacobs,
        S. Kaul, W. K. Laskey, D. J. Pennell, J. A. Rumberger, T. Ryan,
        and M. S. Verani, "Standardized myocardial segmentation and
        nomenclature for tomographic imaging of the heart",
        Circulation, vol. 105, no. 4, pp. 539-542, 2002.
    """
    if segBold is None:
        segBold = []

    linewidth = 2
    data = np.array(data).ravel()

    if cmap is None:
        cmap = plt.cm.jet

    if norm is None:
        norm = mpl.colors.Normalize(vmin=data.min(), vmax=data.max())
```

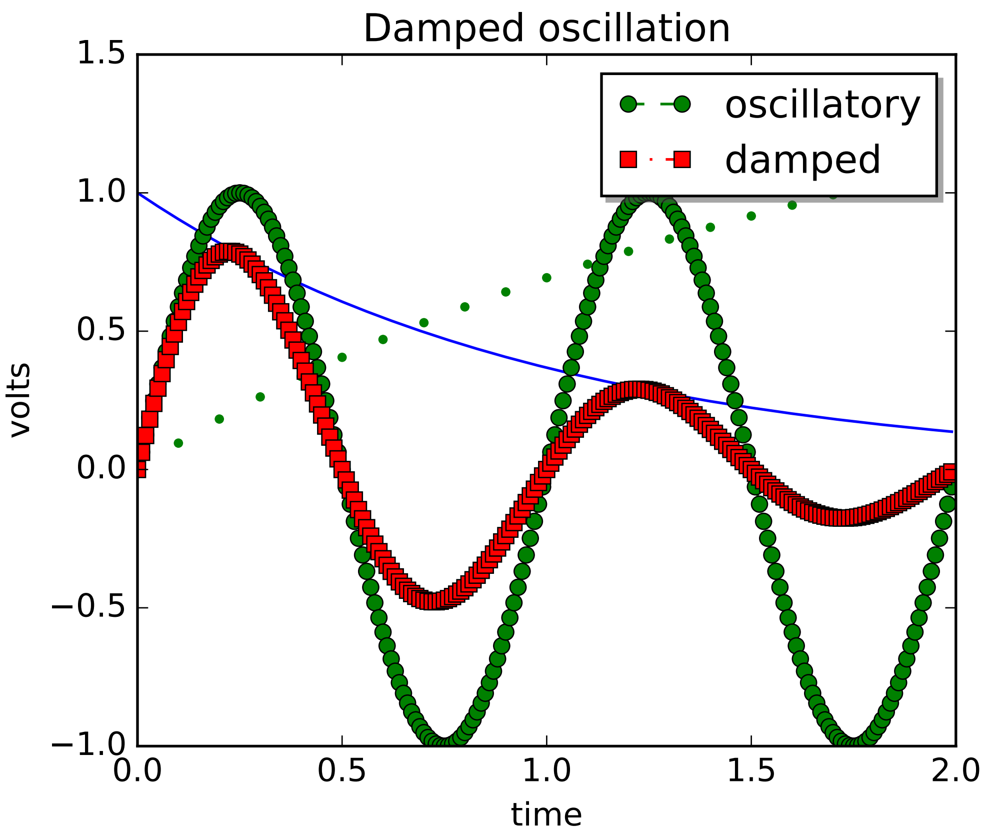
```
    theta = np.linspace(0, 2*np.pi, 768)
    r = np.linspace(0.2, 1, 4)

    # Create the bound for the segment 17
    for i in range(r.shape[0]):
        ax.plot(theta, np.repeat(r[i], theta.shape), '-k', lw=linewidth)

    # Create the bounds for the segments  1-12
    for i in range(6):
        theta_i = i*60*np.pi/180
        ax.plot([theta_i, theta_i], [r[1], 1], '-k', lw=linewidth)

    # Create the bounds for the segmentss 13-16
    for i in range(4):
        theta_i = i*90*np.pi/180 - 45*np.pi/180
        ax.plot([theta_i, theta_i], [r[0], r[1]], '-k', lw=linewidth)

    # Fill the segments 1-6
    r0 = r[2:4]
    r0 = np.repeat(r0[:, np.newaxis], 128, axis=1).T
    for i in range(6):
        # First segment start at 60 degrees
        theta0 = theta[i*128:i*128+128] + 60*np.pi/180
        theta0 = np.repeat(theta0[:, np.newaxis], 2, axis=1)
        z = np.ones((128, 2))*data[i]
        ax.pcolormesh(theta0, r0, z, cmap=cmap, norm=norm)
        if i+1 in segBold:
            ax.plot(theta0, r0, '-k', lw=linewidth+2)
            ax.plot(theta0[0], [r[2], r[3]], '-k', lw=linewidth+1)
            ax.plot(theta0[-1], [r[2], r[3]], '-k', lw=linewidth+1)

    # Fill the segments 7-12
    r0 = r[1:3]
    r0 = np.repeat(r0[:, np.newaxis], 128, axis=1).T
    for i in range(6):
        # First segment start at 60 degrees
        theta0 = theta[i*128:i*128+128] + 60*np.pi/180
        theta0 = np.repeat(theta0[:, np.newaxis], 2, axis=1)
        z = np.ones((128, 2))*data[i+6]
        ax.pcolormesh(theta0, r0, z, cmap=cmap, norm=norm)
        if i+7 in segBold:
            ax.plot(theta0, r0, '-k', lw=linewidth+2)
            ax.plot(theta0[0], [r[1], r[2]], '-k', lw=linewidth+1)
            ax.plot(theta0[-1], [r[1], r[2]], '-k', lw=linewidth+1)

    # Fill the segments 13-16
    r0 = r[0:2]
    r0 = np.repeat(r0[:, np.newaxis], 192, axis=1).T
    for i in range(4):
        # First segment start at 45 degrees
        theta0 = theta[i*192:i*192+192] + 45*np.pi/180
        theta0 = np.repeat(theta0[:, np.newaxis], 2, axis=1)
        z = np.ones((192, 2))*data[i+12]
```

```
        ax.pcolormesh(theta0, r0, z, cmap=cmap, norm=norm)
        if i+13 in segBold:
            ax.plot(theta0, r0, '-k', lw=linewidth+2)
            ax.plot(theta0[0], [r[0], r[1]], '-k', lw=linewidth+1)
            ax.plot(theta0[-1], [r[0], r[1]], '-k', lw=linewidth+1)

    # Fill the segments 17
    if data.size == 17:
        r0 = np.array([0, r[0]])
        r0 = np.repeat(r0[:, np.newaxis], theta.size, axis=1).T
        theta0 = np.repeat(theta[:, np.newaxis], 2, axis=1)
        z = np.ones((theta.size, 2))*data[16]
        ax.pcolormesh(theta0, r0, z, cmap=cmap, norm=norm)
        if 17 in segBold:
            ax.plot(theta0, r0, '-k', lw=linewidth+2)

    ax.set_ylim([0, 1])
    ax.set_yticklabels([])
    ax.set_xticklabels([])


# Create the fake data
data = np.array(range(17)) + 1


# Make a figure and axes with dimensions as desired.
fig, ax = plt.subplots(figsize=(12, 8), nrows=1, ncols=3,
                       subplot_kw=dict(projection='polar'))
fig.canvas.set_window_title('Left Ventricle Bulls Eyes (AHA)')

# Create the axis for the colorbars
axl = fig.add_axes([0.14, 0.15, 0.2, 0.05])
axl2 = fig.add_axes([0.41, 0.15, 0.2, 0.05])
axl3 = fig.add_axes([0.69, 0.15, 0.2, 0.05])


# Set the colormap and norm to correspond to the data for which
# the colorbar will be used.
cmap = mpl.cm.jet
norm = mpl.colors.Normalize(vmin=1, vmax=17)

# ColorbarBase derives from ScalarMappable and puts a colorbar
# in a specified axes, so it has everything needed for a
# standalone colorbar.  There are many more kwargs, but the
# following gives a basic continuous colorbar with ticks
# and labels.
cb1 = mpl.colorbar.ColorbarBase(axl, cmap=cmap, norm=norm,
                                orientation='horizontal')
cb1.set_label('Some Units')


# Set the colormap and norm to correspond to the data for which
# the colorbar will be used.
```

```
cmap2 = mpl.cm.cool
norm2 = mpl.colors.Normalize(vmin=1, vmax=17)

# ColorbarBase derives from ScalarMappable and puts a colorbar
# in a specified axes, so it has everything needed for a
# standalone colorbar.  There are many more kwargs, but the
# following gives a basic continuous colorbar with ticks
# and labels.
cb2 = mpl.colorbar.ColorbarBase(axl2, cmap=cmap2, norm=norm2,
                                orientation='horizontal')
cb2.set_label('Some other units')


# The second example illustrates the use of a ListedColormap, a
# BoundaryNorm, and extended ends to show the "over" and "under"
# value colors.
cmap3 = mpl.colors.ListedColormap(['r', 'g', 'b', 'c'])
cmap3.set_over('0.35')
cmap3.set_under('0.75')

# If a ListedColormap is used, the length of the bounds array must be
# one greater than the length of the color list.  The bounds must be
# monotonically increasing.
bounds = [2, 3, 7, 9, 15]
norm3 = mpl.colors.BoundaryNorm(bounds, cmap3.N)
cb3 = mpl.colorbar.ColorbarBase(axl3, cmap=cmap3, norm=norm3,
                                # to use 'extend', you must
                                # specify two extra boundaries:
                                boundaries=[0]+bounds+[18],
                                extend='both',
                                ticks=bounds,  # optional
                                spacing='proportional',
                                orientation='horizontal')
cb3.set_label('Discrete intervals, some other units')


# Create the 17 segment model
bullseye_plot(ax[0], data, cmap=cmap, norm=norm)
ax[0].set_title('Bulls Eye (AHA)')

bullseye_plot(ax[1], data, cmap=cmap2, norm=norm2)
ax[1].set_title('Bulls Eye (AHA)')

bullseye_plot(ax[2], data, segBold=[3, 5, 6, 11, 12, 16],
              cmap=cmap3, norm=norm3)
ax[2].set_title('Segments [3,5,6,11,12,16] in bold')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

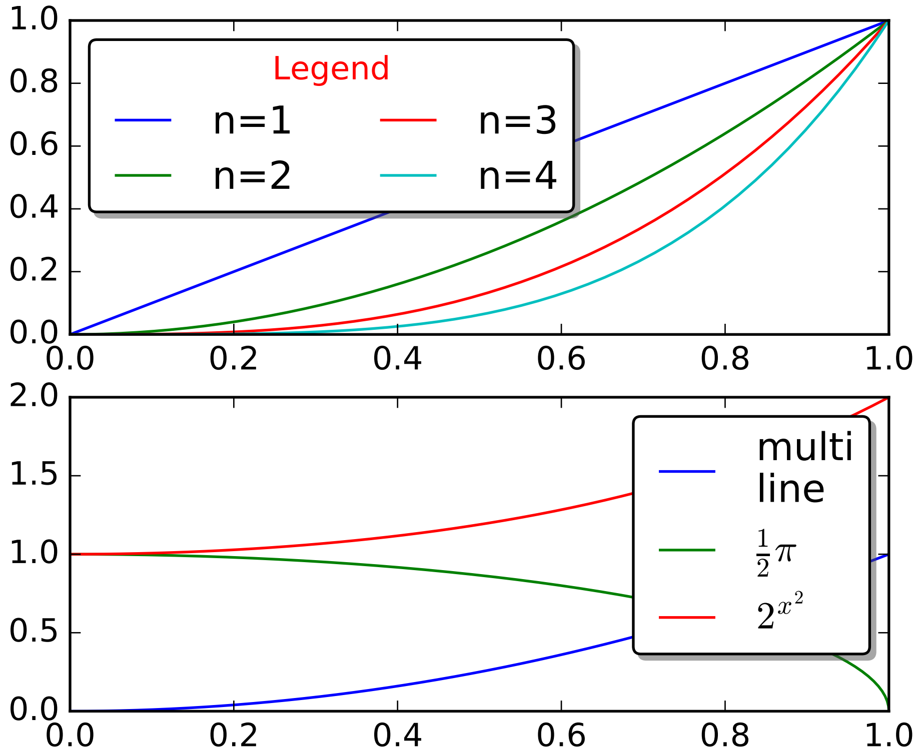
90.116 pylab_examples example code: legend_demo2.py




```
# Make a legend for specific lines.
import matplotlib.pyplot as plt
import numpy as np


t1 = np.arange(0.0, 2.0, 0.1)
t2 = np.arange(0.0, 2.0, 0.01)

# note that plot returns a list of lines.  The "l1, = plot" usage
# extracts the first element of the list into l1 using tuple
# unpacking.  So l1 is a Line2D instance, not a sequence of lines
l1, = plt.plot(t2, np.exp(-t2))
l2, l3 = plt.plot(t2, np.sin(2 * np.pi * t2), '--go', t1, np.log(1 + t1), '.')
l4, = plt.plot(t2, np.exp(-t2) * np.sin(2 * np.pi * t2), 'rs-.')

plt.legend((l2, l4), ('oscillatory', 'damped'), loc='upper right', shadow=True)
plt.xlabel('time')
plt.ylabel('volts')
plt.title('Damped oscillation')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

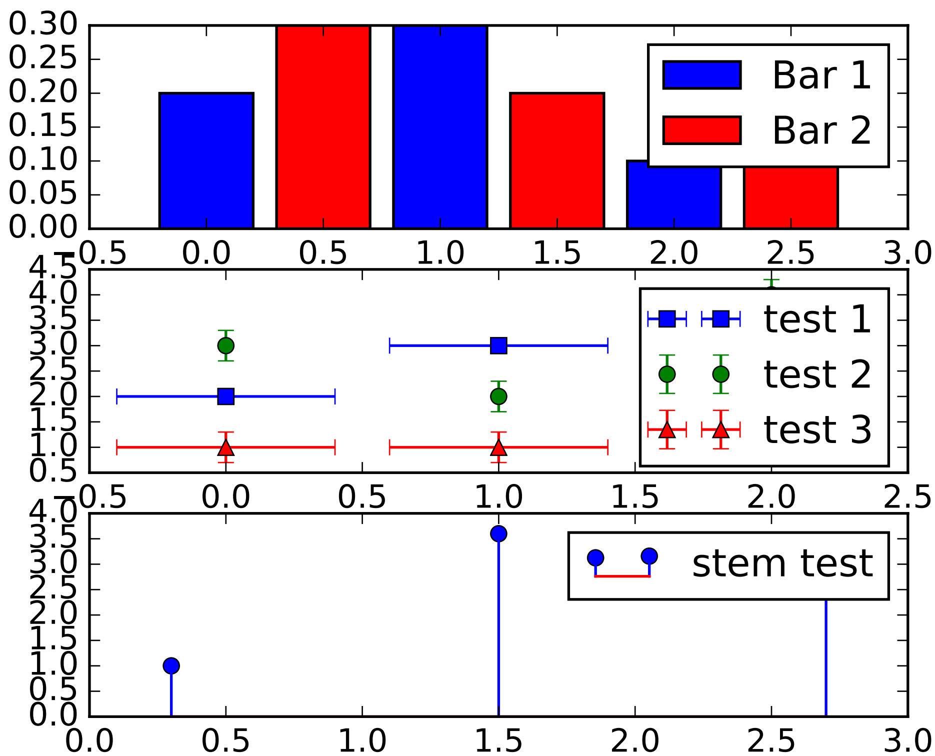
90.117 pylab_examples example code: legend_demo3.py




```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 1)

# Plot the lines y=x**n for n=1..4.
ax = plt.subplot(2, 1, 1)
for n in range(1, 5):
    plt.plot(x, x**n, label="n={0}".format(n))
plt.legend(loc="upper left", bbox_to_anchor=[0, 1],
           ncol=2, shadow=True, title="Legend", fancybox=True)
ax.get_legend().get_title().set_color("red")

# Demonstrate some more complex labels.
ax = plt.subplot(2, 1, 2)
plt.plot(x, x**2, label="multi\nline")
half_pi = np.linspace(0, np.pi / 2)
plt.plot(np.sin(half_pi), np.cos(half_pi), label=r"$\frac{1}{2}\pi$")
plt.plot(x, 2**(x**2), label="$2^{x^2}$")
plt.legend(shadow=True, fancybox=True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.118 pylab_examples example code: legend_demo4.py

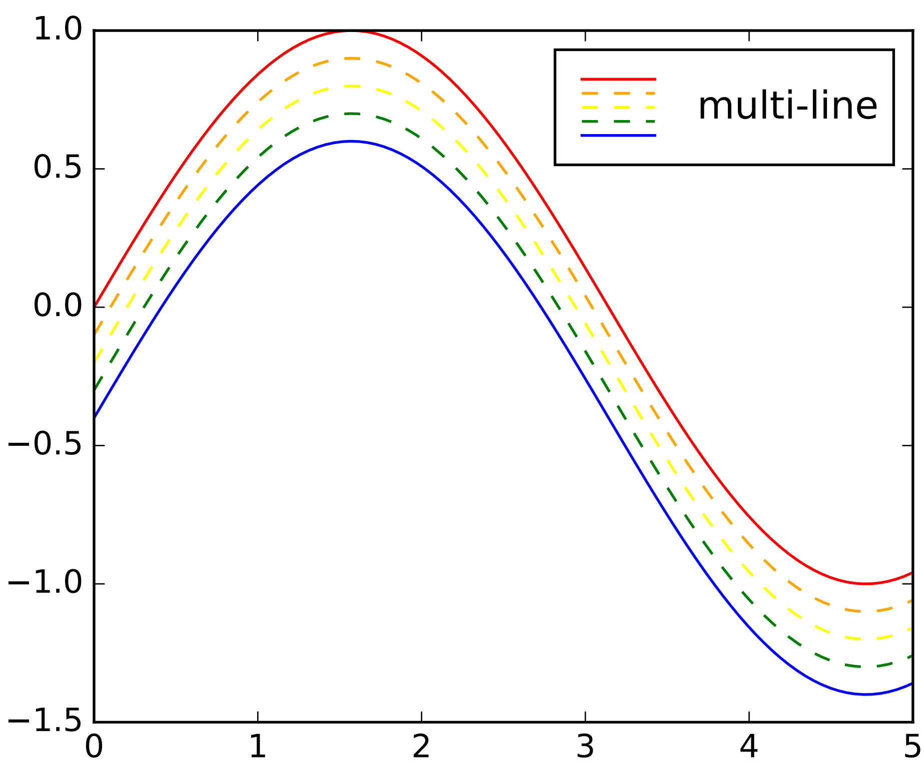



```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(3, 1)
top_ax, middle_ax, bottom_ax = axes

top_ax.bar([0, 1, 2], [0.2, 0.3, 0.1], width=0.4, label="Bar 1",
           align="center")
top_ax.bar([0.5, 1.5, 2.5], [0.3, 0.2, 0.2], color="red", width=0.4,
           label="Bar 2", align="center")
top_ax.legend()

middle_ax.errorbar([0, 1, 2], [2, 3, 1], xerr=0.4, fmt="s", label="test 1")
middle_ax.errorbar([0, 1, 2], [3, 2, 4], yerr=0.3, fmt="o", label="test 2")
middle_ax.errorbar([0, 1, 2], [1, 1, 3], xerr=0.4, yerr=0.3, fmt="^",
                   label="test 3")
middle_ax.legend()

bottom_ax.stem([0.3, 1.5, 2.7], [1, 3.6, 2.7], label="stem test")
bottom_ax.legend()
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.119 pylab_examples example code: legend_demo5.py



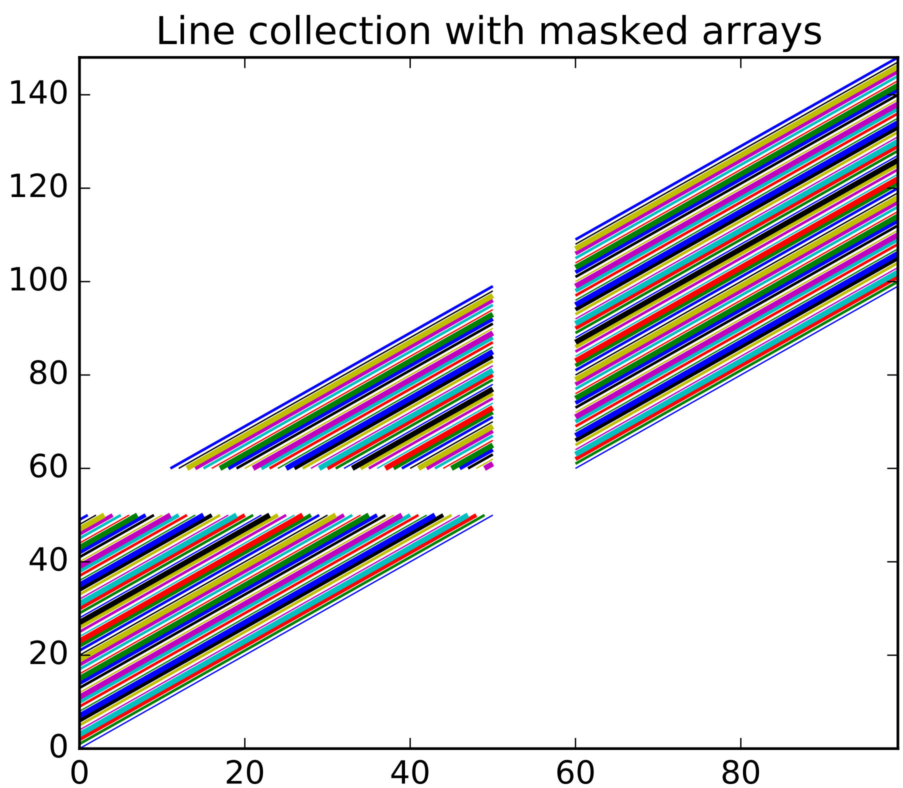

```
from __future__ import (absolute_import, division,
                        print_function, unicode_literals)
import six
from matplotlib import pyplot as plt
import numpy as np
from matplotlib.legend_handler import HandlerLineCollection
import matplotlib.collections as mcol
from matplotlib.lines import Line2D


class HandlerDashedLines(HandlerLineCollection):
    """
    Custom Handler for LineCollection instances.
    """
    def create_artists(self, legend, orig_handle,
```

```
                       xdescent, ydescent, width, height, fontsize, trans):
        # figure out how many lines there are
        numlines = len(orig_handle.get_segments())
        xdata, xdata_marker = self.get_xdata(legend, xdescent, ydescent,
                                             width, height, fontsize)
        leglines = []
        # divide the vertical space where the lines will go
        # into equal parts based on the number of lines
        ydata = ((height) / (numlines + 1)) * np.ones(xdata.shape, float)
        # for each line, create the line at the proper location
        # and set the dash pattern
        for i in range(numlines):
            legline = Line2D(xdata, ydata * (numlines - i) - ydescent)
            self.update_prop(legline, orig_handle, legend)
            # set color, dash pattern, and linewidth to that
            # of the lines in linecollection
            try:
                color = orig_handle.get_colors()[i]
            except IndexError:
                color = orig_handle.get_colors()[0]
            try:
                dashes = orig_handle.get_dashes()[i]
            except IndexError:
                dashes = orig_handle.get_dashes()[0]
            try:
                lw = orig_handle.get_linewidths()[i]
            except IndexError:
                lw = orig_handle.get_linewidths()[0]
            if dashes[0] is not None:
                legline.set_dashes(dashes[1])
            legline.set_color(color)
            legline.set_transform(trans)
            legline.set_linewidth(lw)
            leglines.append(legline)
        return leglines

x = np.linspace(0, 5, 100)

plt.figure()
colors = ['red', 'orange', 'yellow', 'green', 'blue']
styles = ['solid', 'dashed', 'dashed', 'dashed', 'solid']
lines = []
for i, color, style in zip(range(5), colors, styles):
    plt.plot(x, np.sin(x) - .1 * i, c=color, ls=style)


# make proxy artists
# make list of one line -- doesn't matter what the coordinates are
line = [[(0, 0)]]
# set up the proxy artist
lc = mcol.LineCollection(5 * line, linestyles=styles, colors=colors)
# create the legend
plt.legend([lc], ['multi-line'], handler_map={type(lc): HandlerDashedLines()},
```

```
          handlelength=2.5, handleheight=3)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.120 pylab_examples example code: line_collection.py

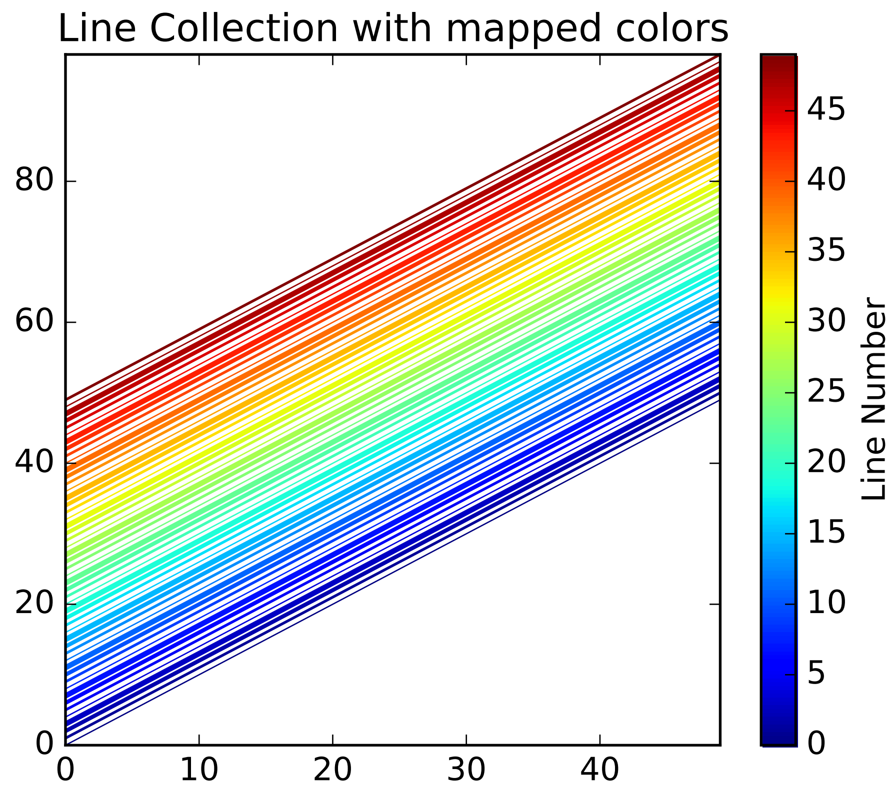



```
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib.colors import colorConverter

import numpy as np

# In order to efficiently plot many lines in a single set of axes,
# Matplotlib has the ability to add the lines all at once. Here is a
# simple example showing how it is done.

x = np.arange(100)
# Here are many sets of y to plot vs x
ys = x[:50, np.newaxis] + x[np.newaxis, :]
```

```
segs = np.zeros((50, 100, 2), float)
segs[:, :, 1] = ys
segs[:, :, 0] = x

# Mask some values to test masked array support:
segs = np.ma.masked_where((segs > 50) & (segs < 60), segs)

# We need to set the plot limits.
ax = plt.axes()
ax.set_xlim(x.min(), x.max())
ax.set_ylim(ys.min(), ys.max())

# colors is sequence of rgba tuples
# linestyle is a string or dash tuple. Legal string values are
#          solid|dashed|dashdot|dotted.  The dash tuple is (offset, onoffseq)
#          where onoffseq is an even length tuple of on and off ink in points.
#          If linestyle is omitted, 'solid' is used
# See matplotlib.collections.LineCollection for more information
colors = [colorConverter.to_rgba(i) for i in 'bgrcmyk']

line_segments = LineCollection(segs, linewidths=(0.5, 1, 1.5, 2),
                               colors=colors, linestyle='solid')
ax.add_collection(line_segments)
ax.set_title('Line collection with masked arrays')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.121 pylab_examples example code: line_collection2.py

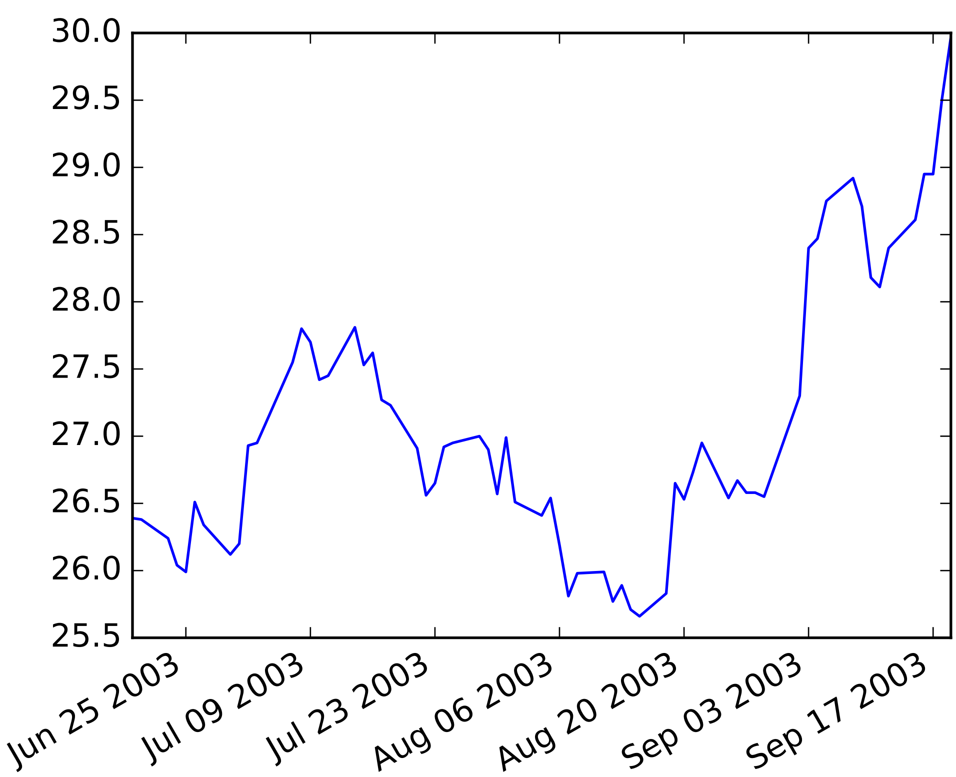



```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.collections import LineCollection

# In order to efficiently plot many lines in a single set of axes,
# Matplotlib has the ability to add the lines all at once. Here is a
# simple example showing how it is done.

N = 50
x = np.arange(N)
# Here are many sets of y to plot vs x
ys = [x + i for i in x]

# We need to set the plot limits, they will not autoscale
ax = plt.axes()
ax.set_xlim((np.amin(x), np.amax(x)))
ax.set_ylim((np.amin(np.amin(ys)), np.amax(np.amax(ys))))

# colors is sequence of rgba tuples
# linestyle is a string or dash tuple. Legal string values are
#          solid|dashed|dashdot|dotted.  The dash tuple is (offset, onoffseq)
#          where onoffseq is an even length tuple of on and off ink in points.
```

```
#          If linestyle is omitted, 'solid' is used
# See matplotlib.collections.LineCollection for more information

# Make a sequence of x,y pairs
line_segments = LineCollection([list(zip(x, y)) for y in ys],
                               linewidths=(0.5, 1, 1.5, 2),
                               linestyles='solid')
line_segments.set_array(x)
ax.add_collection(line_segments)
fig = plt.gcf()
axcb = fig.colorbar(line_segments)
axcb.set_label('Line Number')
ax.set_title('Line Collection with mapped colors')
plt.sci(line_segments)  # This allows interactive changing of the colormap.
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.122 pylab_examples example code: load_converter.py

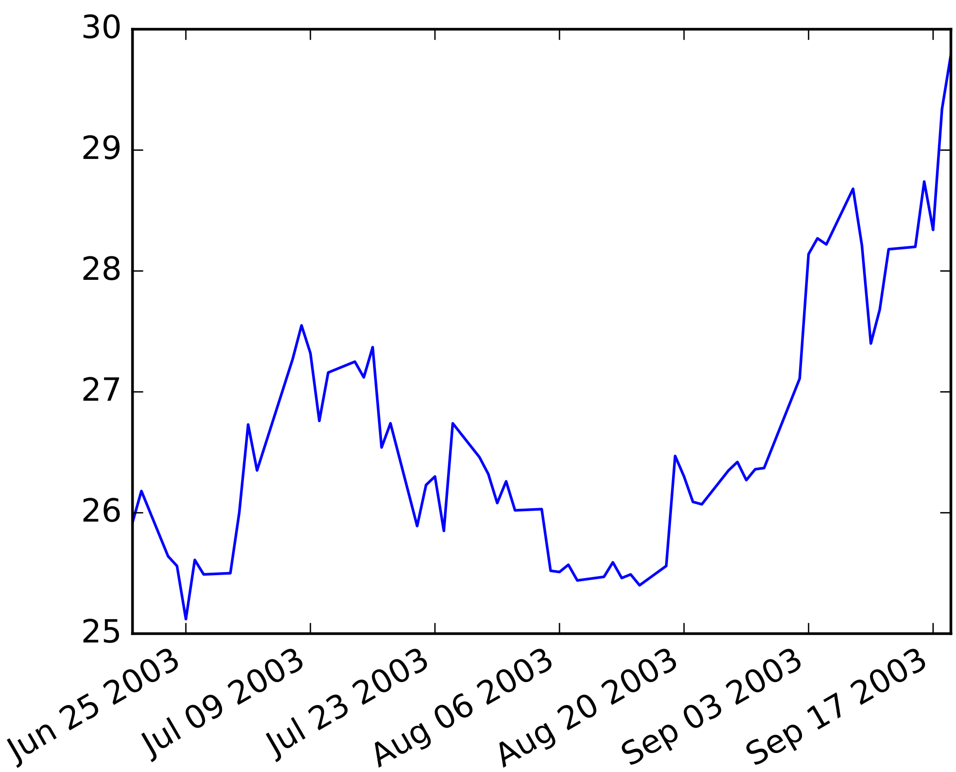



```
from __future__ import print_function
import numpy as np
```

```
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import matplotlib.dates as mdates
from matplotlib.dates import bytespdate2num

datafile = cbook.get_sample_data('msft.csv', asfileobj=False)
print('loading', datafile)

dates, closes = np.loadtxt(datafile, delimiter=',',
                           converters={0: bytespdate2num('%d-%b-%y')},
                           skiprows=1, usecols=(0, 2), unpack=True)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot_date(dates, closes, '-')
fig.autofmt_xdate()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

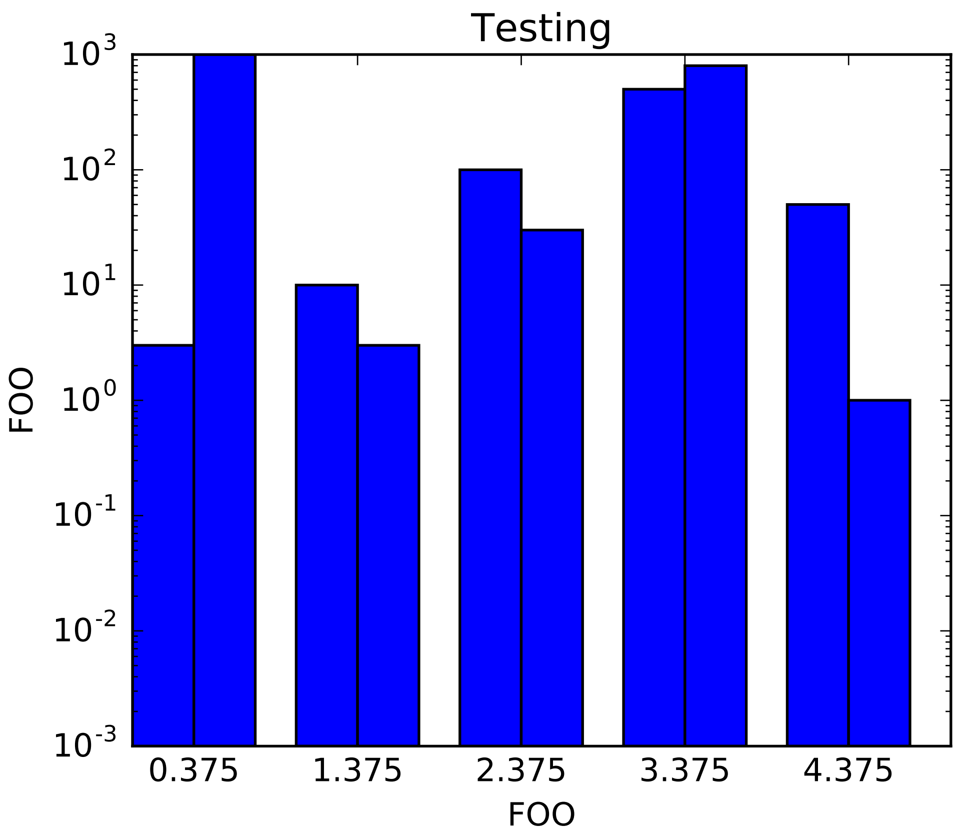
90.123 pylab_examples example code: loadrec.py



```
from __future__ import print_function
from matplotlib import mlab
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

datafile = cbook.get_sample_data('msft.csv', asfileobj=False)
print('loading', datafile)
a = mlab.csv2rec(datafile)
a.sort()
print(a.dtype)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(a.date, a.adj_close, '-')
fig.autofmt_xdate()

# if you have xlwt installed, you can output excel
try:
    import mpl_toolkits.exceltools as exceltools
    exceltools.rec2excel(a, 'test.xls')
except ImportError:
    pass
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

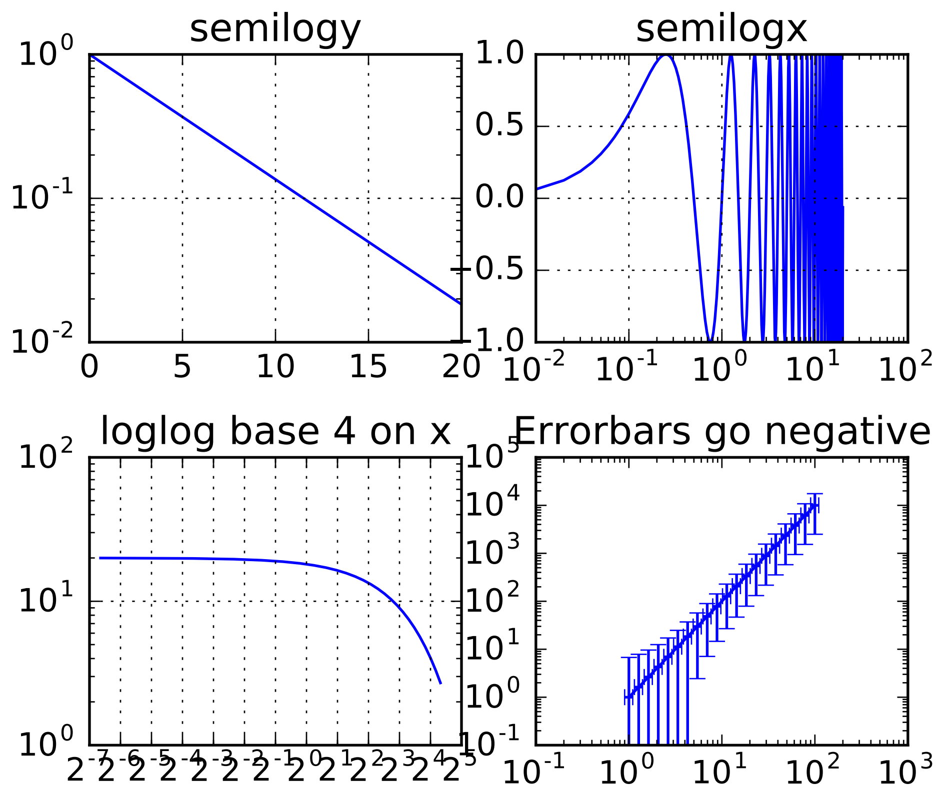
90.124 pylab_examples example code: log_bar.py




```
import matplotlib.pyplot as plt
import numpy as np

data = ((3, 1000), (10, 3), (100, 30), (500, 800), (50, 1))

plt.xlabel("FOO")
plt.ylabel("FOO")
plt.title("Testing")
plt.yscale('log')

dim = len(data[0])
w = 0.75
dimw = w / dim

x = np.arange(len(data))
for i in range(len(data[0])):
    y = [d[i] for d in data]
    b = plt.bar(x + i * dimw, y, dimw, bottom=0.001)
plt.xticks(x + w / 2)
plt.ylim((0.001, 1000))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.125 pylab_examples example code: log_demo.py

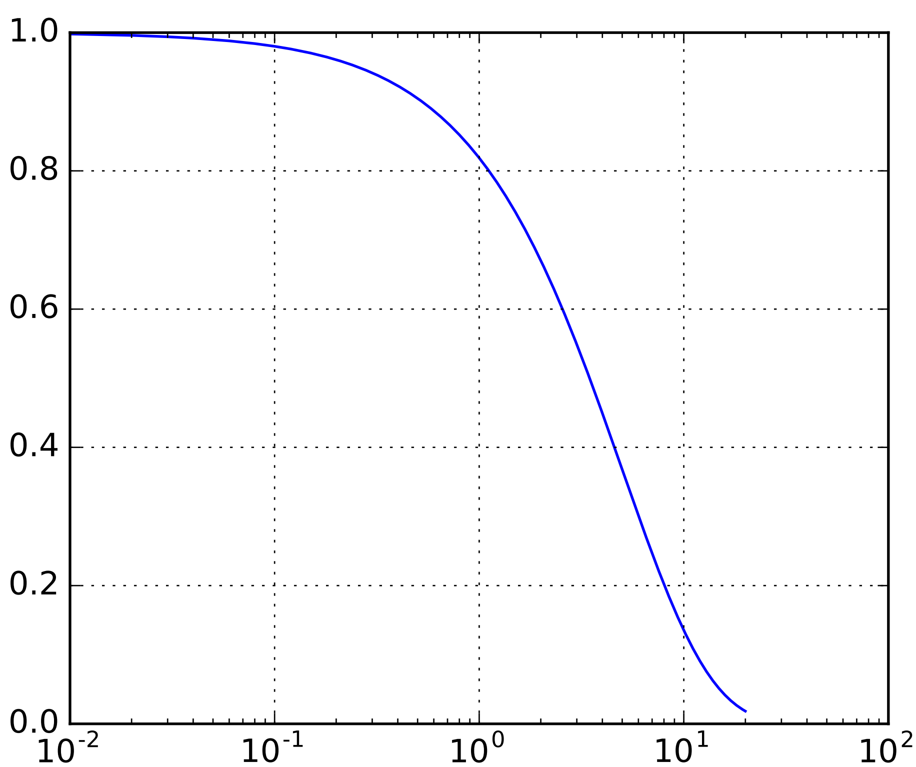



```
import numpy as np
import matplotlib.pyplot as plt

plt.subplots_adjust(hspace=0.4)
t = np.arange(0.01, 20.0, 0.01)

# log y axis
plt.subplot(221)
plt.semilogy(t, np.exp(-t/5.0))
plt.title('semilogy')
plt.grid(True)

# log x axis
plt.subplot(222)
plt.semilogx(t, np.sin(2*np.pi*t))
plt.title('semilogx')
plt.grid(True)

# log x and y axis
```

```
plt.subplot(223)
plt.loglog(t, 20*np.exp(-t/10.0), basex=2)
plt.grid(True)
plt.title('loglog base 4 on x')

# with errorbars: clip non-positive values
ax = plt.subplot(224)
ax.set_xscale("log", nonposx='clip')
ax.set_yscale("log", nonposy='clip')

x = 10.0**np.linspace(0.0, 2.0, 20)
y = x**2.0
plt.errorbar(x, y, xerr=0.1*x, yerr=5.0 + 0.75*y)
ax.set_ylim(ymin=0.1)
ax.set_title('Errorbars go negative')


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

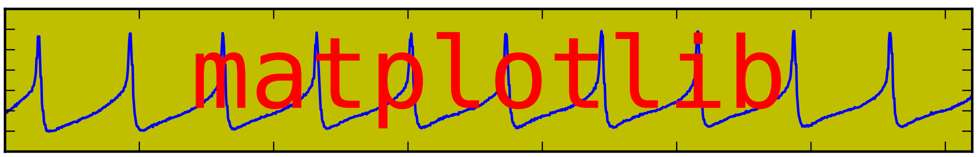
90.126 pylab_examples example code: log_test.py



```
import matplotlib.pyplot as plt
import numpy as np

dt = 0.01
t = np.arange(dt, 20.0, dt)

plt.semilogx(t, np.exp(-t/5.0))
plt.grid(True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

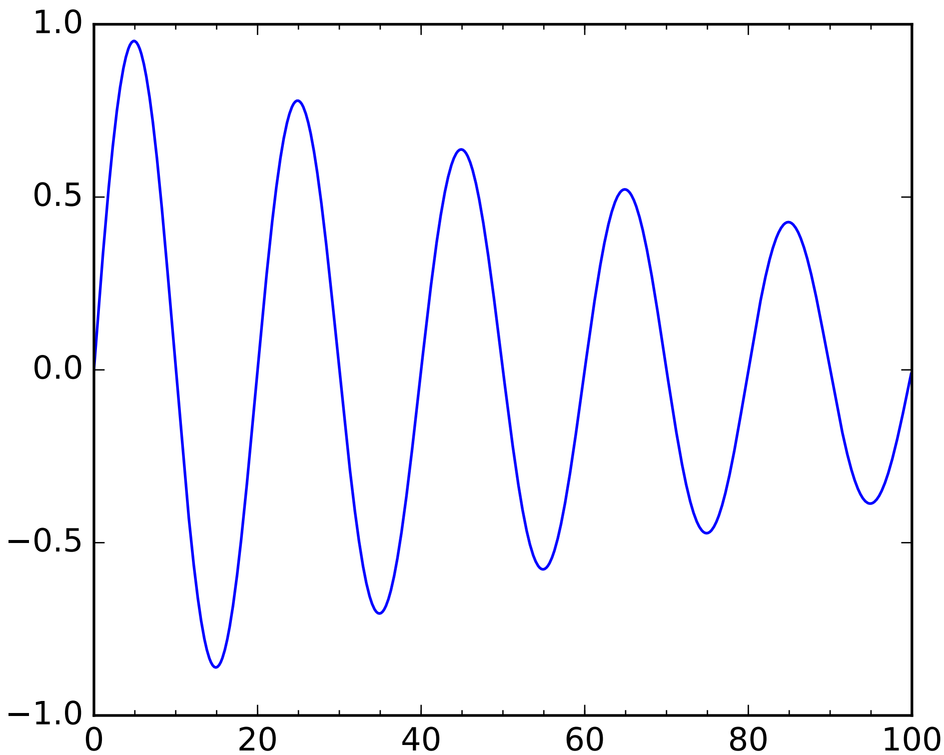
90.127 pylab_examples example code: logo.py




```
# This file generates an old version of the matplotlib logo

from __future__ import print_function
# Above import not necessary for Python 3 onwards. Recommend taking this
# out in examples in the future, since we should all move to Python 3.
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.cbook as cbook

# convert data to mV
datafile = cbook.get_sample_data('membrane.dat', asfileobj=False)
print('loading', datafile)

x = 1000 * 0.1 * np.fromstring(open(datafile, 'rb').read(), np.float32)
# 0.0005 is the sample interval
t = 0.0005 * np.arange(len(x))
plt.figure(1, figsize=(7, 1), dpi=100)
ax = plt.subplot(111, axisbg='y')
plt.plot(t, x)
plt.text(0.5, 0.5, 'matplotlib', color='r',
         fontsize=40, fontname=['Courier', 'Bitstream Vera Sans Mono'],
         horizontalalignment='center',
         verticalalignment='center',
         transform=ax.transAxes,
         )
plt.axis([1, 1.72, -60, 10])
plt.gca().set_xticklabels([])
plt.gca().set_yticklabels([])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.128 pylab_examples example code: major_minor_demo1.py

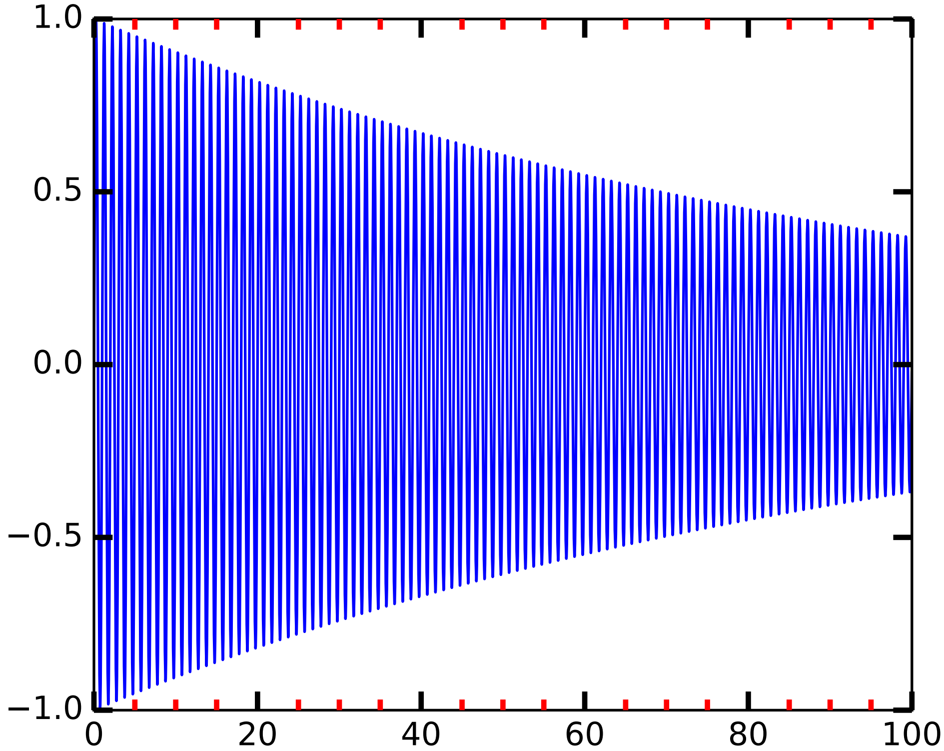



```
"""
Demonstrate how to use major and minor tickers.

The two relevant userland classes are Locators and Formatters.
Locators determine where the ticks are and formatters control the
formatting of ticks.

Minor ticks are off by default (NullLocator and NullFormatter).  You
can turn minor ticks on w/o labels by setting the minor locator.  You
can also turn labeling on for the minor ticker by setting the minor
formatter

Make a plot with major ticks that are multiples of 20 and minor ticks
that are multiples of 5.  Label major ticks with %d formatting but
don't label minor ticks

The MultipleLocator ticker class is used to place ticks on multiples of
```

```
some base.  The FormatStrFormatter uses a string format string (e.g.,
'%d' or '%1.2f' or '%1.1f cm' ) to format the tick

The pyplot interface grid command changes the grid settings of the
major ticks of the y and y axis together.  If you want to control the
grid of the minor ticks for a given axis, use for example

  ax.xaxis.grid(True, which='minor')

Note, you should not use the same locator between different Axis
because the locator stores references to the Axis data and view limits

"""

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import MultipleLocator, FormatStrFormatter

majorLocator = MultipleLocator(20)
majorFormatter = FormatStrFormatter('%d')
minorLocator = MultipleLocator(5)


t = np.arange(0.0, 100.0, 0.1)
s = np.sin(0.1*np.pi*t)*np.exp(-t*0.01)

fig, ax = plt.subplots()
plt.plot(t, s)

ax.xaxis.set_major_locator(majorLocator)
ax.xaxis.set_major_formatter(majorFormatter)

# for the minor ticks, use no labels; default NullFormatter
ax.xaxis.set_minor_locator(minorLocator)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.129 pylab_examples example code: major_minor_demo2.py

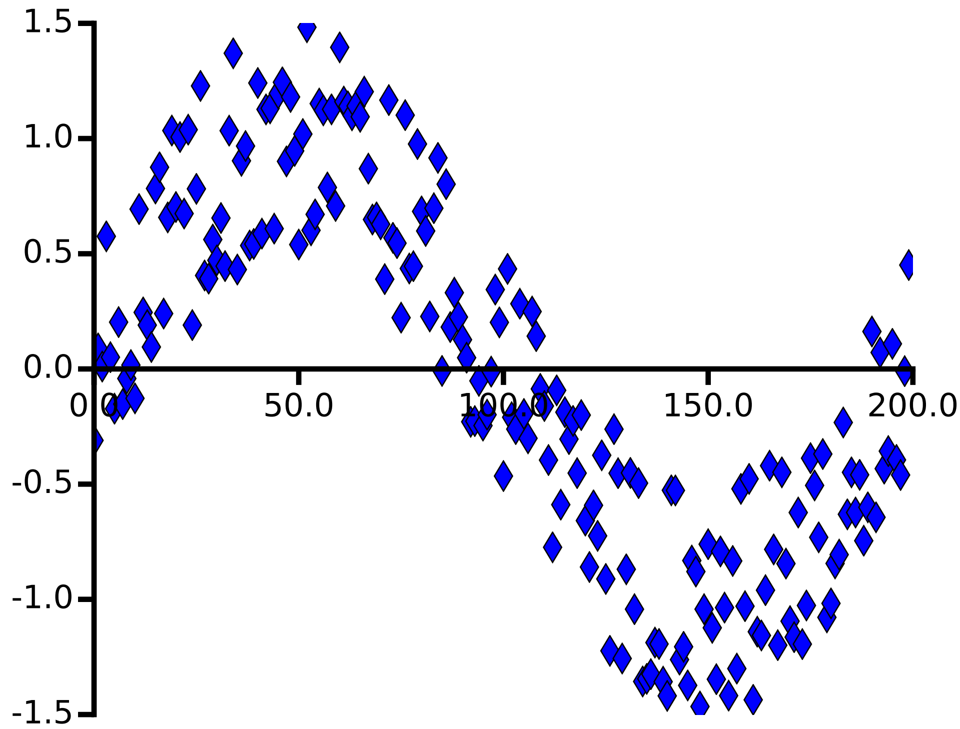



```
#!/usr/bin/env python
"""
Automatic tick selection for major and minor ticks.

Use interactive pan and zoom to see how the tick intervals
change. There will be either 4 or 5 minor tick intervals
per major interval, depending on the major interval.
"""

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import AutoMinorLocator

# One can supply an argument to AutoMinorLocator to
# specify a fixed number of minor intervals per major interval, e.g.:
# minorLocator = AutoMinorLocator(2)
# would lead to a single minor tick between major ticks.

minorLocator = AutoMinorLocator()


t = np.arange(0.0, 100.0, 0.01)
```

```
s = np.sin(2*np.pi*t)*np.exp(-t*0.01)

fig, ax = plt.subplots()
plt.plot(t, s)

ax.xaxis.set_minor_locator(minorLocator)

plt.tick_params(which='both', width=2)
plt.tick_params(which='major', length=7)
plt.tick_params(which='minor', length=4, color='r')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.130 pylab_examples example code: manual_axis.py

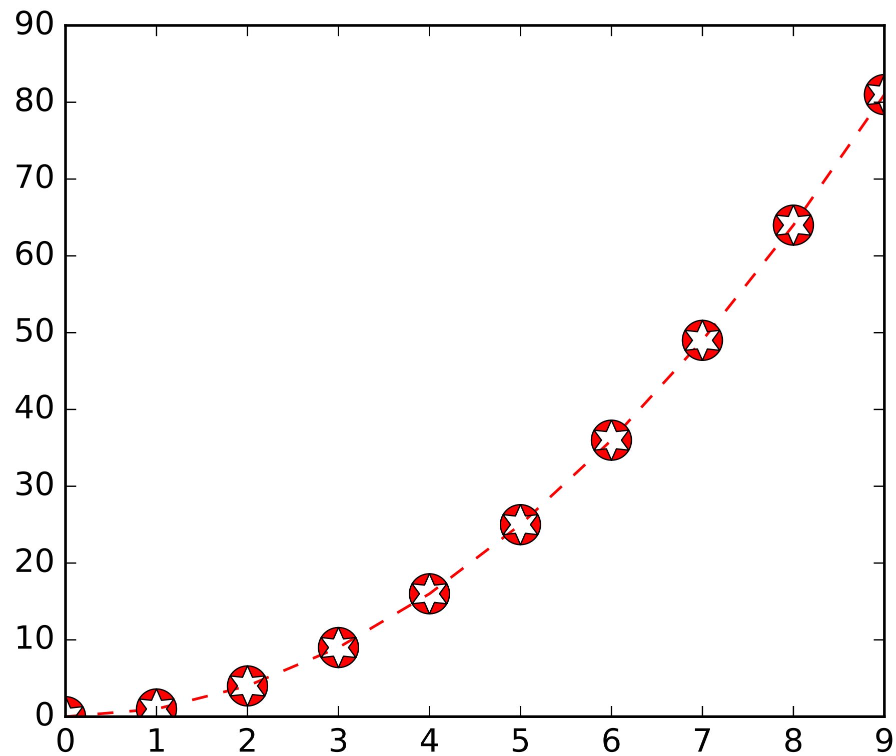



```
"""
The techniques here are no longer required with the new support for
spines in matplotlib -- see
http://matplotlib.org/examples/pylab_examples/spine_placement_demo.html.
```

```
This example should be considered deprecated and is left just for demo
purposes for folks wanting to make a pseudo-axis
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines


def make_xaxis(ax, yloc, offset=0.05, **props):
    xmin, xmax = ax.get_xlim()
    locs = [loc for loc in ax.xaxis.get_majorticklocs()
            if loc >= xmin and loc <= xmax]
    tickline, = ax.plot(locs, [yloc]*len(locs), linestyle='',
                        marker=lines.TICKDOWN, **props)
    axline, = ax.plot([xmin, xmax], [yloc, yloc], **props)
    tickline.set_clip_on(False)
    axline.set_clip_on(False)
    for loc in locs:
        ax.text(loc, yloc - offset, '%1.1f' % loc,
                horizontalalignment='center',
                verticalalignment='top')


def make_yaxis(ax, xloc=0, offset=0.05, **props):
    ymin, ymax = ax.get_ylim()
    locs = [loc for loc in ax.yaxis.get_majorticklocs()
            if loc >= ymin and loc <= ymax]
    tickline, = ax.plot([xloc]*len(locs), locs, linestyle='',
                        marker=lines.TICKLEFT, **props)
    axline, = ax.plot([xloc, xloc], [ymin, ymax], **props)
    tickline.set_clip_on(False)
    axline.set_clip_on(False)

    for loc in locs:
        ax.text(xloc - offset, loc, '%1.1f' % loc,
                verticalalignment='center',
                horizontalalignment='right')


props = dict(color='black', linewidth=2, markeredgewidth=2)
x = np.arange(200.)
y = np.sin(2*np.pi*x/200.) + np.random.rand(200) - 0.5
fig = plt.figure(facecolor='white')
ax = fig.add_subplot(111, frame_on=False)
ax.axison = False
ax.plot(x, y, 'd', markersize=8, markerfacecolor='blue')
ax.set_xlim(0, 200)
ax.set_ylim(-1.5, 1.5)
make_xaxis(ax, 0, offset=0.1, **props)
make_yaxis(ax, 0, offset=5, **props)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

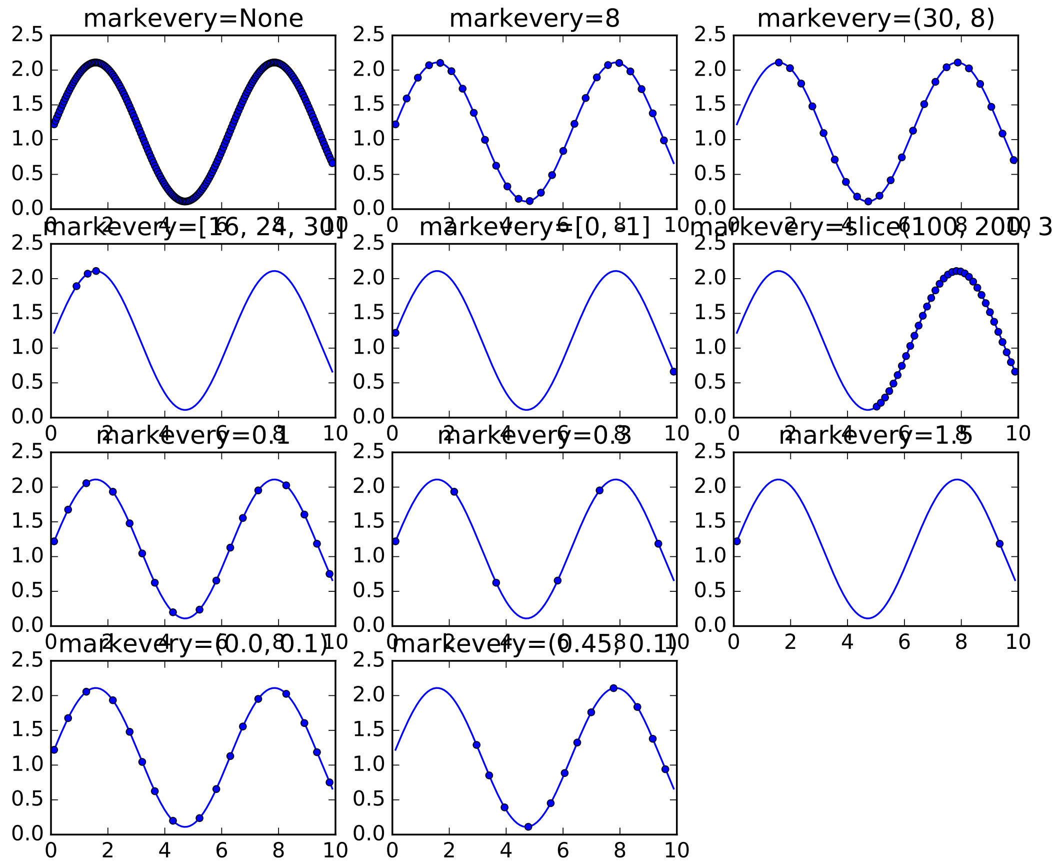
90.131 pylab_examples example code: marker_path.py

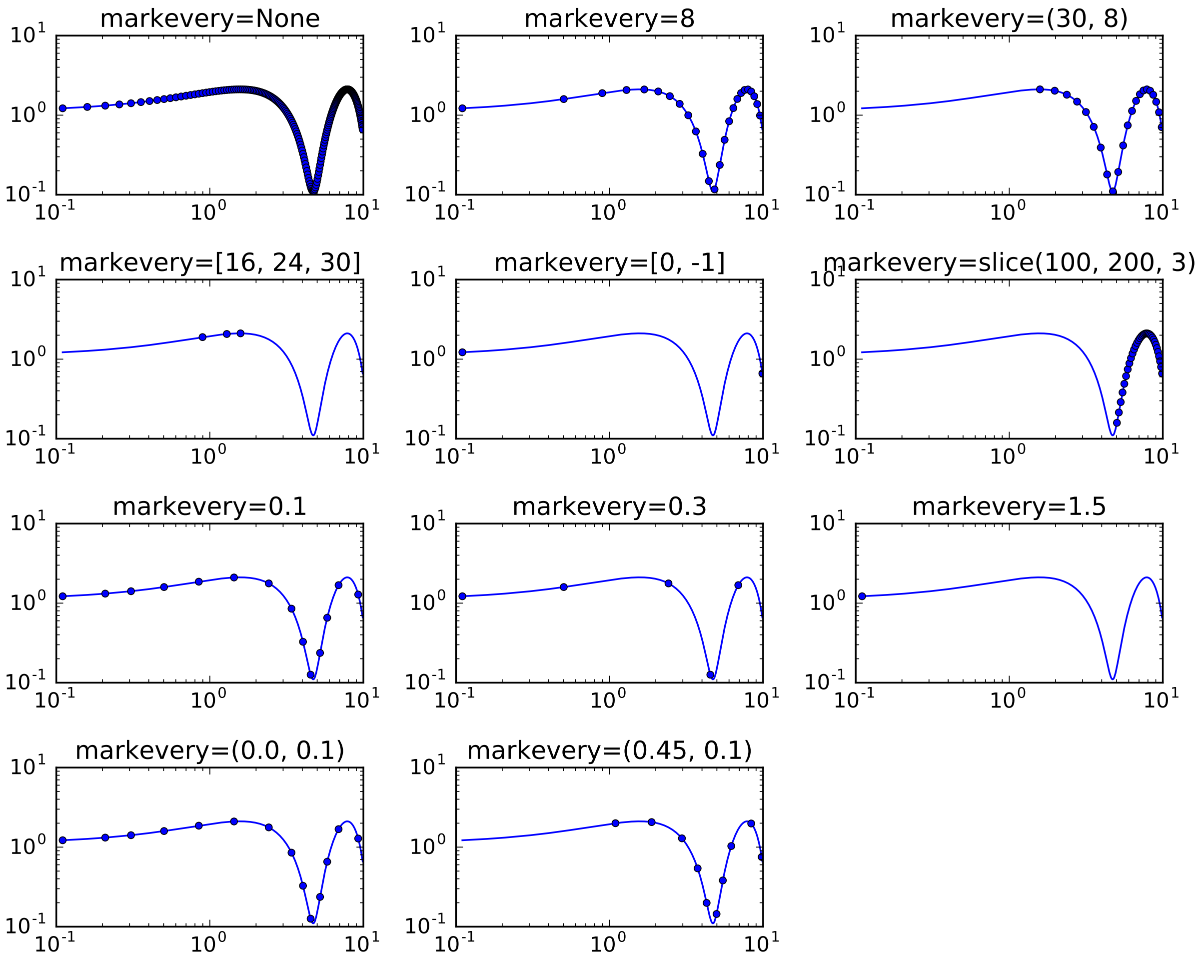


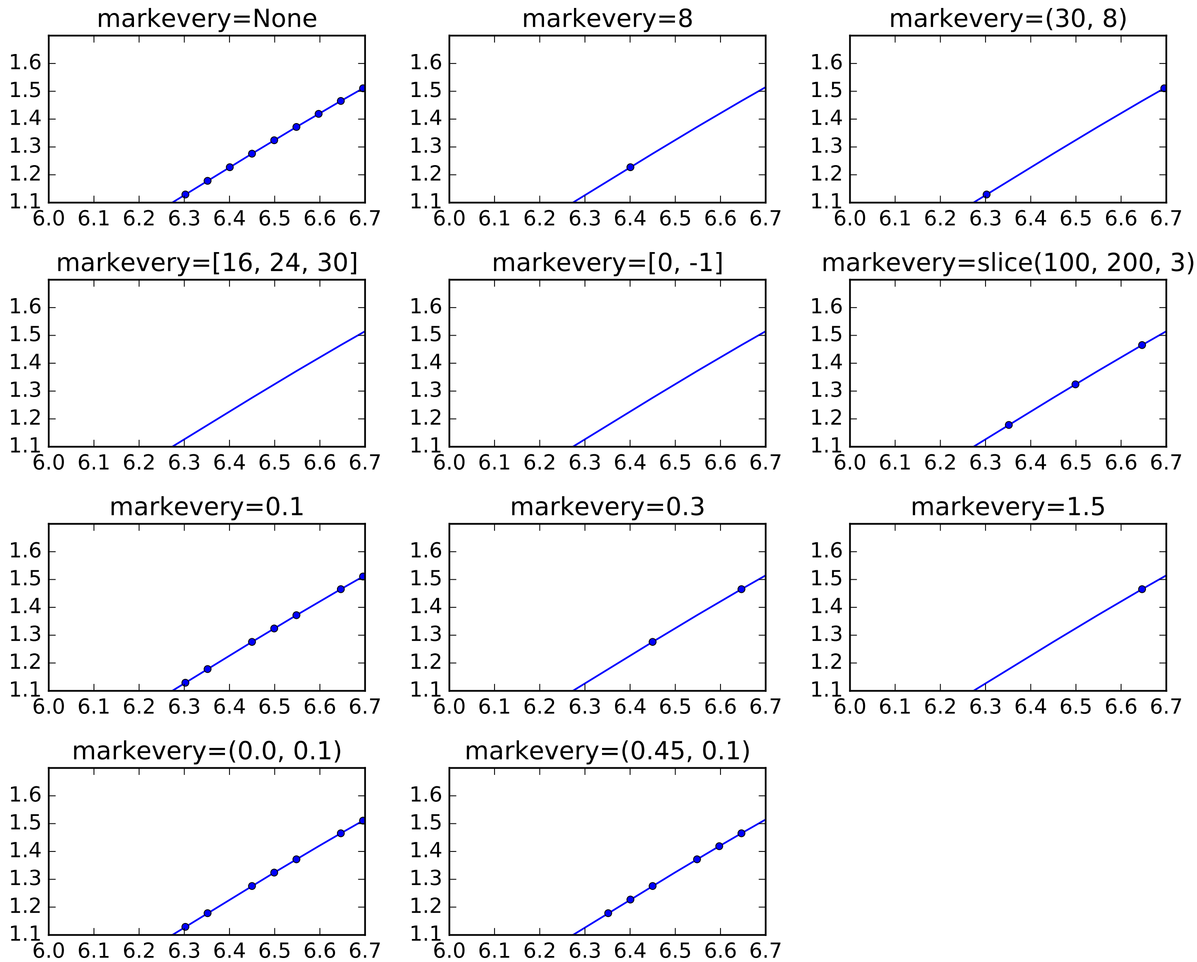

```
import matplotlib.pyplot as plt
import matplotlib.path as mpath
import numpy as np


star = mpath.Path.unit_regular_star(6)
circle = mpath.Path.unit_circle()
# concatenate the circle with an internal cutout of the star
verts = np.concatenate([circle.vertices, star.vertices[::-1, ...]])
codes = np.concatenate([circle.codes, star.codes])
cut_star = mpath.Path(verts, codes)


plt.plot(np.arange(10)**2, '--r', marker=cut_star, markersize=15)

plt.show()
```

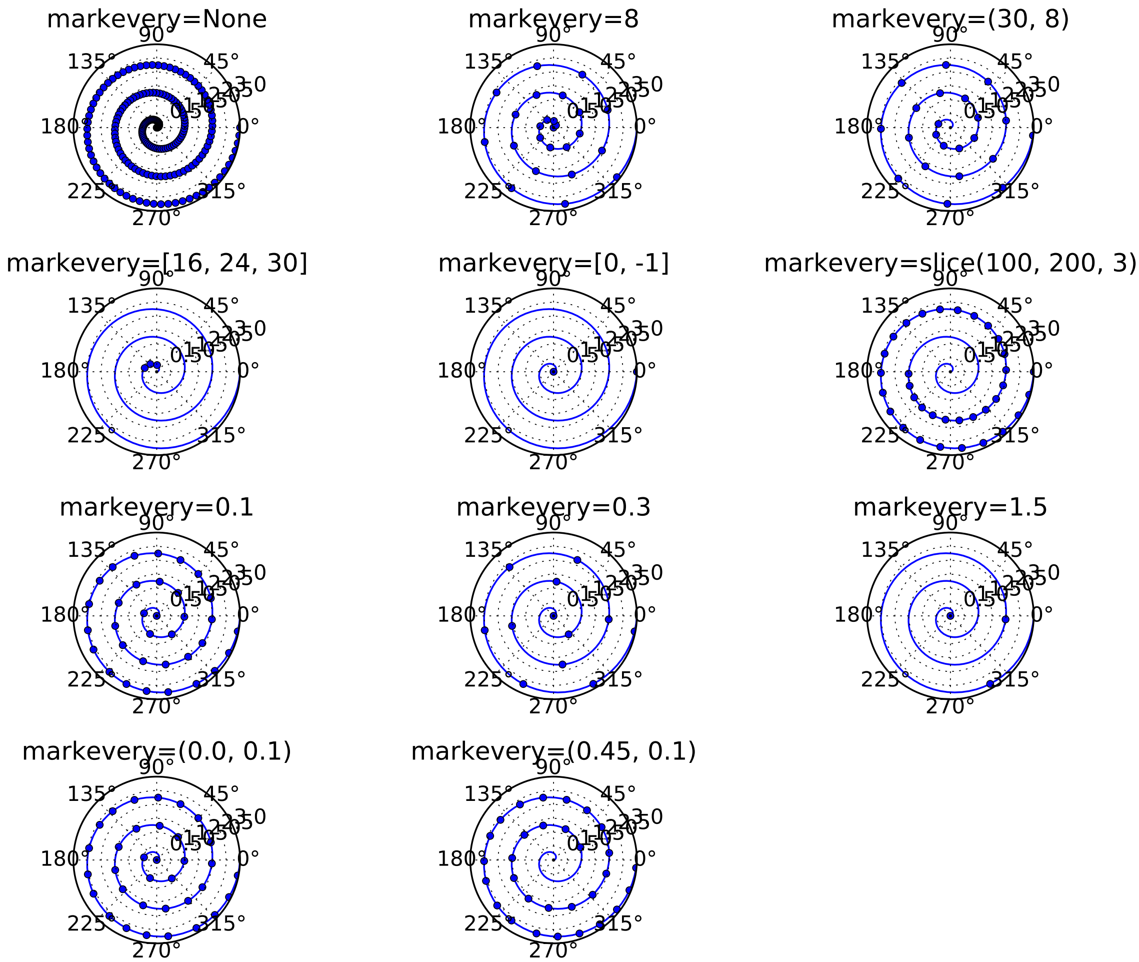
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.132 pylab_examples example code: markevery_demo.py




markevery=None
markevery=8
markevery=(30, 8)
markevery=[16, 24, 30]
markevery=[0, -1]
markevery=slice(100, 200, 3)
markevery=0.1
markevery=0.3
markevery=1.5
markevery=(0.0, 0.1)
markevery=(0.45, 0.1)


markevery=None
markevery=8
markevery=(30, 8)
markevery=[16, 24, 30]
markevery=[0, -1]
markevery=slice(100, 200, 3)
markevery=0.1
markevery=0.3
markevery=1.5
markevery=(0.0, 0.1)
markevery=(0.45, 0.1)



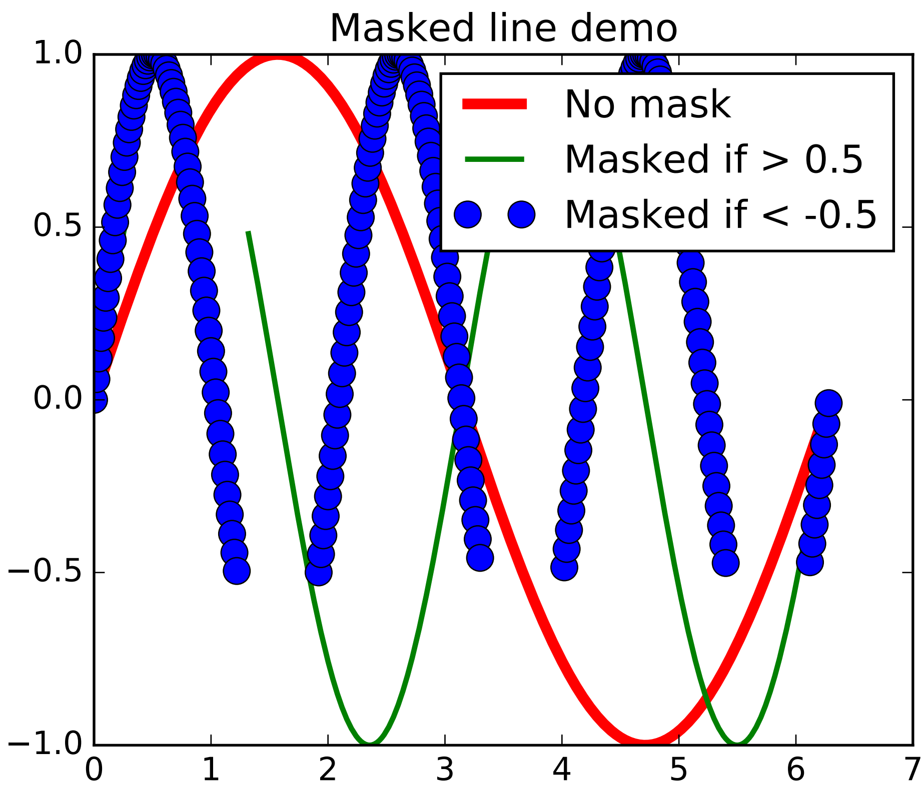

```python
"""
This example demonstrates the various options for showing a marker at a
subset of data points using the `markevery` property of a Line2D object.

Integer arguments are fairly intuitive.  e.g. `markevery`=5 will plot every
5th marker starting from the first data point.

Float arguments allow markers to be spaced at approximately equal distances
along the line.  The theoretical distance along the line between markers is
determined by multiplying the display-coordinate distance of the axes
bounding-box diagonal by the value of `markevery`.  The data points closest
to the theoretical distances will be shown.

A slice or list/array can also be used with `markevery` to specify the markers
to show.

"""

from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
```

```
import matplotlib.gridspec as gridspec

# define a list of markevery cases to plot
cases = [None,
         8,
         (30, 8),
         [16, 24, 30], [0, -1],
         slice(100, 200, 3),
         0.1, 0.3, 1.5,
         (0.0, 0.1), (0.45, 0.1)]

# define the figure size and grid layout properties
figsize = (10, 8)
cols = 3
gs = gridspec.GridSpec(len(cases) // cols + 1, cols)

# define the data for cartesian plots
delta = 0.11
x = np.linspace(0, 10 - 2 * delta, 200) + delta
y = np.sin(x) + 1.0 + delta

# plot each markevery case for linear x and y scales
fig1 = plt.figure(num=1, figsize=figsize)
ax = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    ax.append(fig1.add_subplot(gs[row, col]))
    ax[-1].set_title('markevery=%s' % str(case))
    ax[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)
#fig1.tight_layout()

# plot each markevery case for log x and y scales
fig2 = plt.figure(num=2, figsize=figsize)
axlog = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    axlog.append(fig2.add_subplot(gs[row, col]))
    axlog[-1].set_title('markevery=%s' % str(case))
    axlog[-1].set_xscale('log')
    axlog[-1].set_yscale('log')
    axlog[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)
fig2.tight_layout()

# plot each markevery case for linear x and y scales but zoomed in
# note the behaviour when zoomed in.  When a start marker offset is specified
# it is always interpreted with respect to the first data point which might be
# different to the first visible data point.
fig3 = plt.figure(num=3, figsize=figsize)
axzoom = []
for i, case in enumerate(cases):
    row = (i // cols)
```

```
    col = i % cols
    axzoom.append(fig3.add_subplot(gs[row, col]))
    axzoom[-1].set_title('markevery=%s' % str(case))
    axzoom[-1].plot(x, y, 'o', ls='-', ms=4, markevery=case)
    axzoom[-1].set_xlim((6, 6.7))
    axzoom[-1].set_ylim((1.1, 1.7))
fig3.tight_layout()

# define data for polar plots
r = np.linspace(0, 3.0, 200)
theta = 2 * np.pi * r

# plot each markevery case for polar plots
fig4 = plt.figure(num=4, figsize=figsize)
axpolar = []
for i, case in enumerate(cases):
    row = (i // cols)
    col = i % cols
    axpolar.append(fig4.add_subplot(gs[row, col], projection='polar'))
    axpolar[-1].set_title('markevery=%s' % str(case))
    axpolar[-1].plot(theta, r, 'o', ls='-', ms=4, markevery=case)
fig4.tight_layout()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.133 pylab_examples example code: masked_demo.py

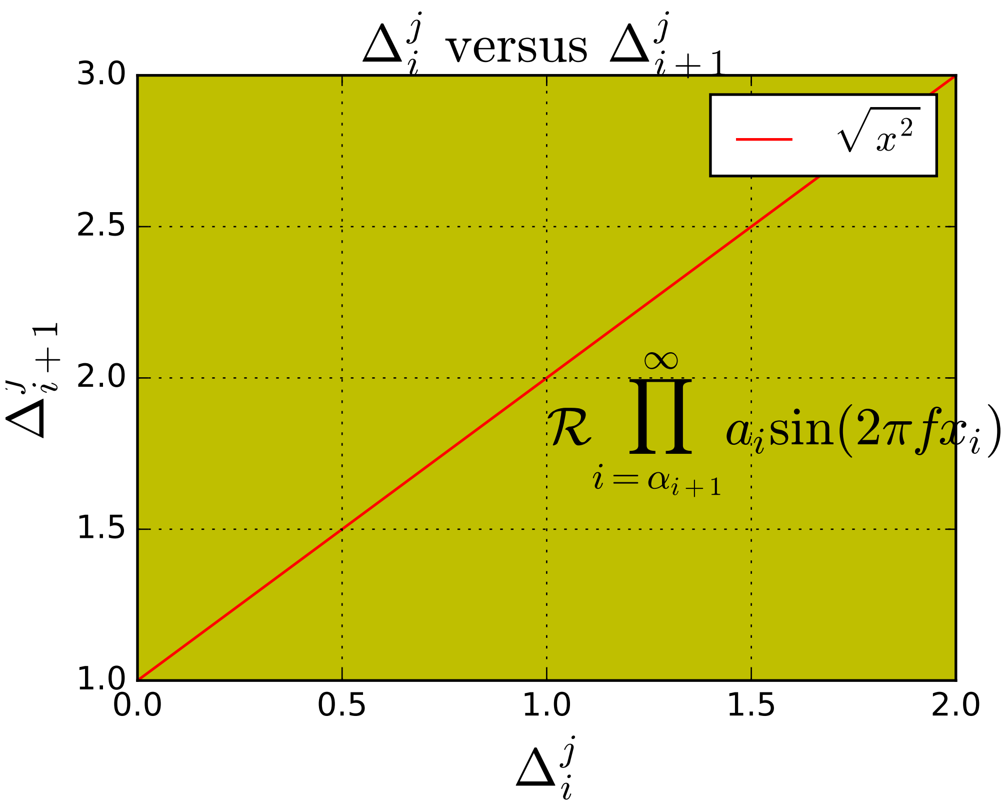



```
'''
Plot lines with points masked out.

This would typically be used with gappy data, to
break the line at the data gaps.
'''

import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 2*np.pi, 0.02)
y = np.sin(x)
y1 = np.sin(2*x)
y2 = np.sin(3*x)
ym1 = np.ma.masked_where(y1 > 0.5, y1)
ym2 = np.ma.masked_where(y2 < -0.5, y2)

lines = plt.plot(x, y, 'r', x, ym1, 'g', x, ym2, 'bo')
plt.setp(lines[0], linewidth=4)
plt.setp(lines[1], linewidth=2)
plt.setp(lines[2], markersize=10)
```

```
plt.legend(('No mask', 'Masked if > 0.5', 'Masked if < -0.5'),
           loc='upper right')
plt.title('Masked line demo')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.134 pylab_examples example code: mathtext_demo.py

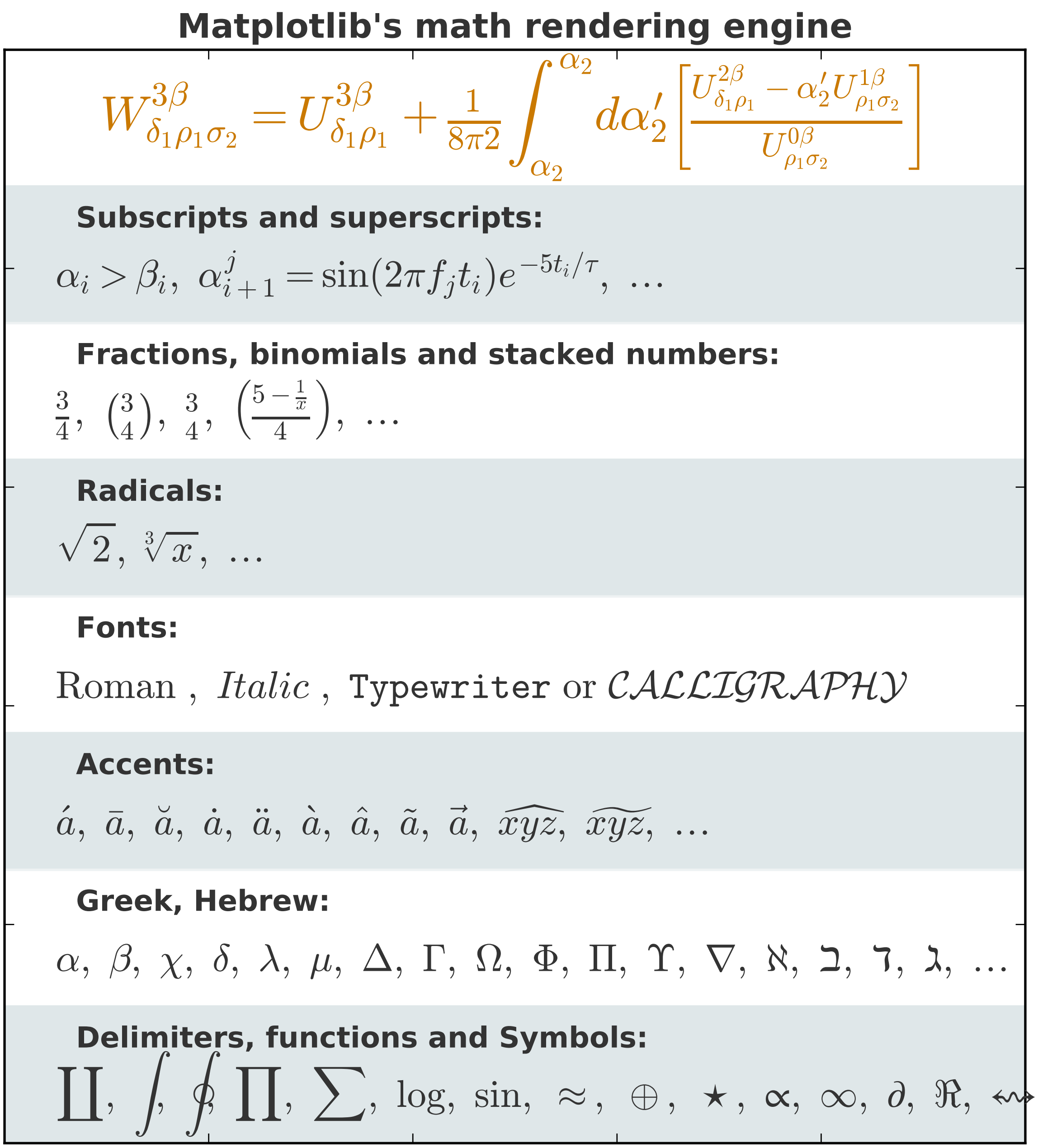



```
#!/usr/bin/env python
"""
Use matplotlib's internal LaTeX parser and layout engine.  For true
latex rendering, see the text.usetex option
"""
import numpy as np
from matplotlib.pyplot import figure, show

fig = figure()
fig.subplots_adjust(bottom=0.2)

ax = fig.add_subplot(111, axisbg='y')
ax.plot([1, 2, 3], 'r')
```

```
x = np.arange(0.0, 3.0, 0.1)

ax.grid(True)
ax.set_xlabel(r'$\Delta_i^j$', fontsize=20)
ax.set_ylabel(r'$\Delta_{i+1}^j$', fontsize=20)
tex = r'$\mathcal{R}\prod_{i=\alpha_{i+1}}^\infty a_i\sin(2 \pi f x_i)$'

ax.text(1, 1.6, tex, fontsize=20, va='bottom')

ax.legend([r"$\sqrt{x^2}$"])

ax.set_title(r'$\Delta_i^j \hspace{0.4} \mathrm{versus} \hspace{0.4} \Delta_{i+1}^j$',
 ↪fontsize=20)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.135 pylab_examples example code: mathtext_examples.py




```
"""
Selected features of Matplotlib's math rendering engine.
"""
from __future__ import print_function
import matplotlib.pyplot as plt
import os
import sys
```

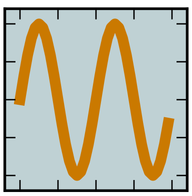
```
import re
import gc

# Selection of features following "Writing mathematical expressions" tutorial
mathtext_titles = {
    0: "Header demo",
    1: "Subscripts and superscripts",
    2: "Fractions, binomials and stacked numbers",
    3: "Radicals",
    4: "Fonts",
    5: "Accents",
    6: "Greek, Hebrew",
    7: "Delimiters, functions and Symbols"}
n_lines = len(mathtext_titles)

# Randomly picked examples
mathext_demos = {
    0: r"$W^{3\beta}_{\delta_1 \rho_1 \sigma_2} = "
    r"U^{3\beta}_{\delta_1 \rho_1} + \frac{1}{8 \pi 2} "
    r"\int^{\alpha_2}_{\alpha_2} d \alpha^\prime_2 \left[\frac{ "
    r"U^{2\beta}_{\delta_1 \rho_1} - \alpha^\prime_2U^{1\beta}_"
    r"{\rho_1 \sigma_2} }{U^{0\beta}_{\rho_1 \sigma_2}}\right]$",

    1: r"$\alpha_i > \beta_i,\ "
    r"\alpha_{i+1}^j = {\rm sin}(2\pi f_j t_i) e^{-5 t_i/\tau},\ "
    r"\ldots$",

    2: r"$\frac{3}{4},\ \binom{3}{4},\ \stackrel{3}{4},\ "
    r"\left(\frac{5 - \frac{1}{x}}{4}\right),\ \ldots$",

    3: r"$\sqrt{2},\ \sqrt[3]{x},\ \ldots$",

    4: r"$\mathrm{Roman}\ , \ \mathit{Italic}\ , \ \mathtt{Typewriter} \ "
    r"\mathrm{or}\ \mathcal{CALLIGRAPHY}$",

    5: r"$\acute a,\ \bar a,\ \breve a,\ \dot a,\ \ddot a, \ \grave a, \ "
    r"\hat a,\ \tilde a,\ \vec a,\ \widehat{xyz},\ \widetilde{xyz},\ "
    r"\ldots$",

    6: r"$\alpha,\ \beta,\ \chi,\ \delta,\ \lambda,\ \mu,\ "
    r"\Delta,\ \Gamma,\ \Omega,\ \Phi,\ \Pi,\ \Upsilon,\ \nabla,\ "
    r"\aleph,\ \beth,\ \daleth,\ \gimel,\ \ldots$",

    7: r"$\coprod,\ \int,\ \oint,\ \prod,\ \sum,\ "
    r"\log,\ \sin,\ \approx,\ \oplus,\ \star,\ \varpropto,\ "
    r"\infty,\ \partial,\ \Re,\ \leftrightsquigarrow, \ \ldots$"}


def doall():
    # Colors used in mpl online documentation.
    mpl_blue_rvb = (191./255., 209./256., 212./255.)
    mpl_orange_rvb = (202/255., 121/256., 0./255.)
    mpl_grey_rvb = (51./255., 51./255., 51./255.)
```

```
    # Creating figure and axis.
    plt.figure(figsize=(6, 7))
    plt.axes([0.01, 0.01, 0.98, 0.90], axisbg="white", frameon=True)
    plt.gca().set_xlim(0., 1.)
    plt.gca().set_ylim(0., 1.)
    plt.gca().set_title("Matplotlib's math rendering engine",
                        color=mpl_grey_rvb, fontsize=14, weight='bold')
    plt.gca().set_xticklabels("", visible=False)
    plt.gca().set_yticklabels("", visible=False)

    # Gap between lines in axes coords
    line_axesfrac = (1. / (n_lines))

    # Plotting header demonstration formula
    full_demo = mathext_demos[0]
    plt.annotate(full_demo,
                 xy=(0.5, 1. - 0.59*line_axesfrac),
                 xycoords='data', color=mpl_orange_rvb, ha='center',
                 fontsize=20)

    # Plotting features demonstration formulae
    for i_line in range(1, n_lines):
        baseline = 1. - (i_line)*line_axesfrac
        baseline_next = baseline - line_axesfrac*1.
        title = mathtext_titles[i_line] + ":"
        fill_color = ['white', mpl_blue_rvb][i_line % 2]
        plt.fill_between([0., 1.], [baseline, baseline],
                         [baseline_next, baseline_next],
                         color=fill_color, alpha=0.5)
        plt.annotate(title,
                     xy=(0.07, baseline - 0.3*line_axesfrac),
                     xycoords='data', color=mpl_grey_rvb, weight='bold')
        demo = mathext_demos[i_line]
        plt.annotate(demo,
                     xy=(0.05, baseline - 0.75*line_axesfrac),
                     xycoords='data', color=mpl_grey_rvb,
                     fontsize=16)

    for i in range(n_lines):
        s = mathext_demos[i]
        print(i, s)
    plt.show()

if '--latex' in sys.argv:
    # Run: python mathtext_examples.py --latex
    # Need amsmath and amssymb packages.
    fd = open("mathtext_examples.ltx", "w")
    fd.write("\\documentclass{article}\n")
    fd.write("\\usepackage{amsmath, amssymb}\n")
    fd.write("\\begin{document}\n")
    fd.write("\\begin{enumerate}\n")
```

```
    for i in range(n_lines):
        s = mathext_demos[i]
        s = re.sub(r"(?<!\\)\$", "$$", s)
        fd.write("\\item %s\n" % s)

    fd.write("\\end{enumerate}\n")
    fd.write("\\end{document}\n")
    fd.close()

    os.system("pdflatex mathtext_examples.ltx")
else:
    doall()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

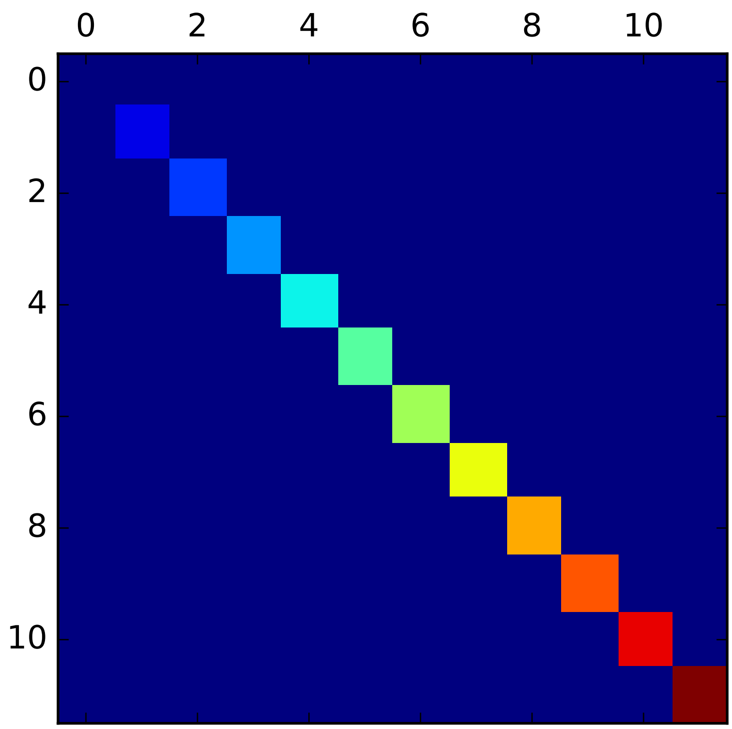
90.136 pylab_examples example code: matplotlib_icon.py

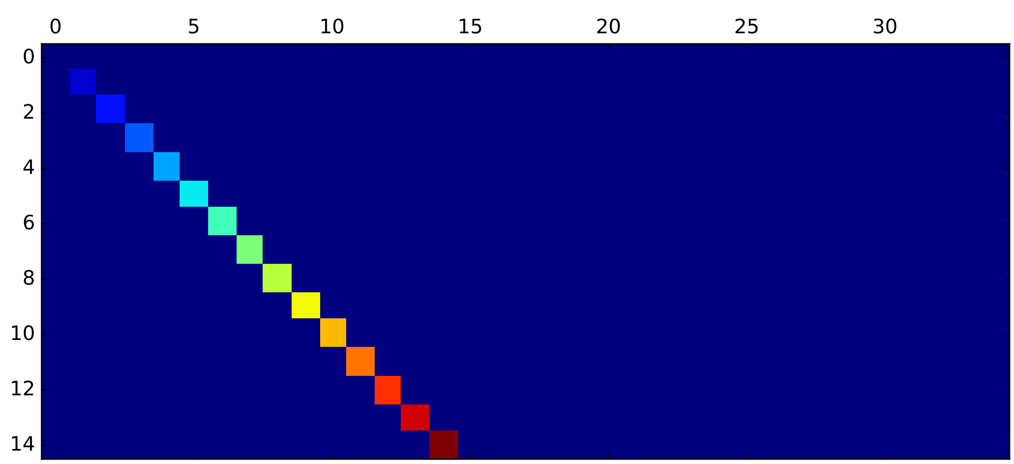


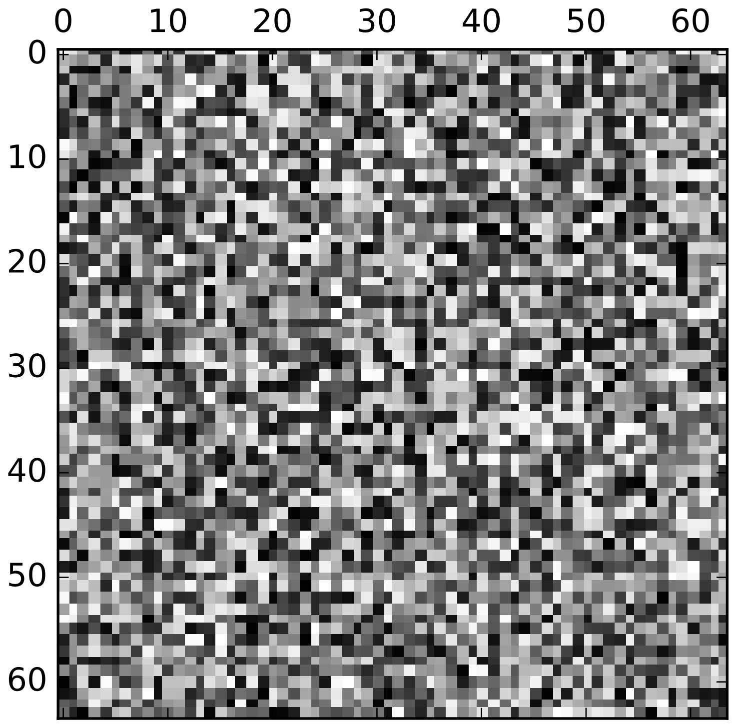
```
"""
make the matplotlib svg minimization icon
"""
import matplotlib.pyplot as plt
import matplotlib
import numpy as np

matplotlib.rc('grid', ls='-', lw=2, color='k')
fig = plt.figure(figsize=(1, 1), dpi=72)
plt.axes([0.025, 0.025, 0.95, 0.95], axisbg='#bfd1d4')

t = np.arange(0, 2, 0.05)
s = np.sin(2*np.pi*t)
plt.plot(t, s, linewidth=4, color="#ca7900")
plt.axis([-.2, 2.2, -1.2, 1.2])

ax = plt.gca()
ax.set_xticklabels([])
ax.set_yticklabels([])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.137 pylab_examples example code: matshow.py








```
"""Simple matshow() example."""
import matplotlib.pyplot as plt
import numpy as np


def samplemat(dims):
    """Make a matrix with all zeros and increasing elements on the diagonal"""
    aa = np.zeros(dims)
    for i in range(min(dims)):
        aa[i, i] = i
    return aa

# Display 2 matrices of different sizes
dimlist = [(12, 12), (15, 35)]
for d in dimlist:
    plt.matshow(samplemat(d))

# Display a random matrix with a specified figure number and a grayscale
# colormap
plt.matshow(np.random.rand(64, 64), fignum=100, cmap=plt.cm.gray)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

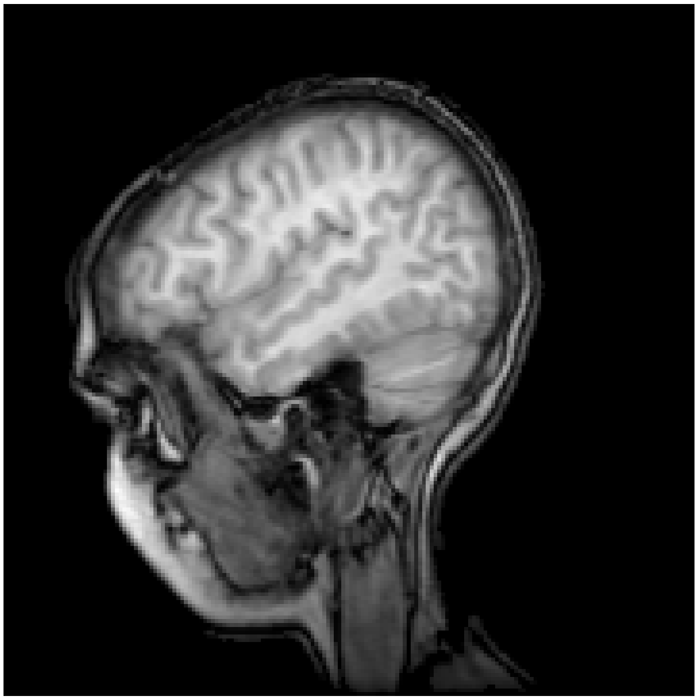
90.138 pylab_examples example code: movie_demo.py

[source code]

```
#!/usr/bin/env python
# -*- noplot -*-

from __future__ import print_function

import os
import matplotlib.pyplot as plt
import numpy as np

files = []

fig, ax = plt.subplots(figsize=(5, 5))
for i in range(50):  # 50 frames
    plt.cla()
    plt.imshow(np.random.rand(5, 5), interpolation='nearest')
    fname = '_tmp%03d.png' % i
    print('Saving frame', fname)
    plt.savefig(fname)
    files.append(fname)

print('Making movie animation.mpg - this make take a while')
os.system("mencoder 'mf://_tmp*.png' -mf type=png:fps=10 -ovc lavc -lavcopts vcodec=wmv2␣
↪-oac copy -o animation.mpg")
#os.system("convert _tmp*.png animation.mng")

# cleanup
for fname in files:
    os.remove(fname)
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

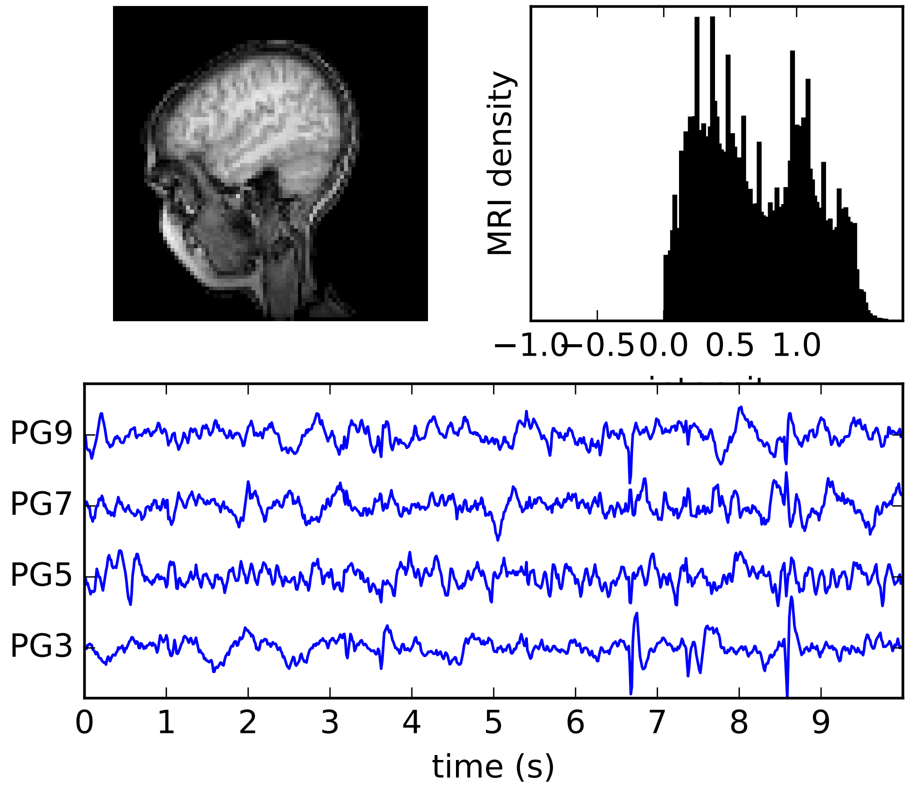
90.139 pylab_examples example code: mri_demo.py



```
from __future__ import print_function
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import numpy as np
# data are 256x256 16 bit integers
dfile = cbook.get_sample_data('s1045.ima.gz')
im = np.fromstring(dfile.read(), np.uint16).astype(float)
im.shape = 256, 256

plt.imshow(im, cmap=plt.cm.gray)
plt.axis('off')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.140 pylab_examples example code: mri_with_eeg.py



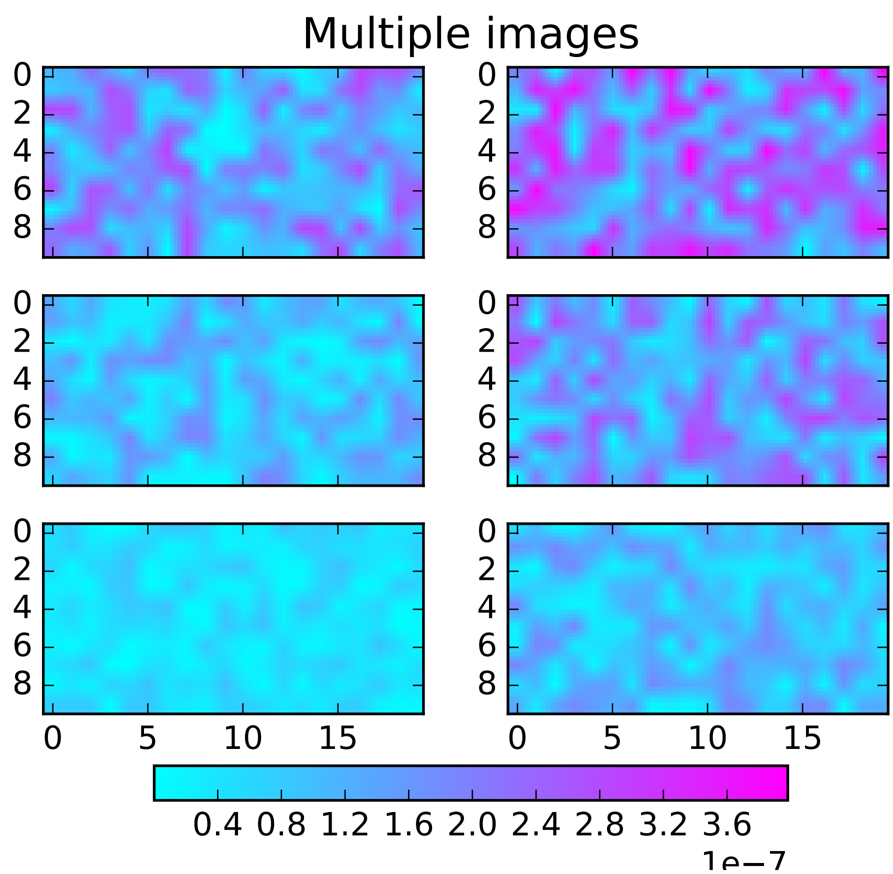

```
#!/usr/bin/env python

"""
This now uses the imshow command instead of pcolor which *is much
faster*
"""
from __future__ import division, print_function

import numpy as np

from matplotlib.pyplot import *
from matplotlib.collections import LineCollection
import matplotlib.cbook as cbook
# I use if 1 to break up the different regions of code visually

if 1:   # load the data
    # data are 256x256 16 bit integers
    dfile = cbook.get_sample_data('s1045.ima.gz')
    im = np.fromstring(dfile.read(), np.uint16).astype(float)
    im.shape = 256, 256

if 1:  # plot the MRI in pcolor
```

```
    subplot(221)
    imshow(im, cmap=cm.gray)
    axis('off')

if 1:  # plot the histogram of MRI intensity
    subplot(222)
    im = np.ravel(im)
    im = im[np.nonzero(im)]  # ignore the background
    im = im/(2.0**15)  # normalize
    hist(im, 100)
    xticks([-1, -.5, 0, .5, 1])
    yticks([])
    xlabel('intensity')
    ylabel('MRI density')

if 1:   # plot the EEG
    # load the data

    numSamples, numRows = 800, 4
    eegfile = cbook.get_sample_data('eeg.dat', asfileobj=False)
    print('loading eeg %s' % eegfile)
    data = np.fromstring(open(eegfile, 'rb').read(), float)
    data.shape = numSamples, numRows
    t = 10.0 * np.arange(numSamples, dtype=float)/numSamples
    ticklocs = []
    ax = subplot(212)
    xlim(0, 10)
    xticks(np.arange(10))
    dmin = data.min()
    dmax = data.max()
    dr = (dmax - dmin)*0.7  # Crowd them a bit.
    y0 = dmin
    y1 = (numRows - 1) * dr + dmax
    ylim(y0, y1)

    segs = []
    for i in range(numRows):
        segs.append(np.hstack((t[:, np.newaxis], data[:, i, np.newaxis])))
        ticklocs.append(i*dr)

    offsets = np.zeros((numRows, 2), dtype=float)
    offsets[:, 1] = ticklocs

    lines = LineCollection(segs, offsets=offsets,
                           transOffset=None,
                           )

    ax.add_collection(lines)

    # set the yticks to use axes coords on the y axis
    ax.set_yticks(ticklocs)
    ax.set_yticklabels(['PG3', 'PG5', 'PG7', 'PG9'])
```

```
    xlabel('time (s)')

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.141 pylab_examples example code: multi_image.py



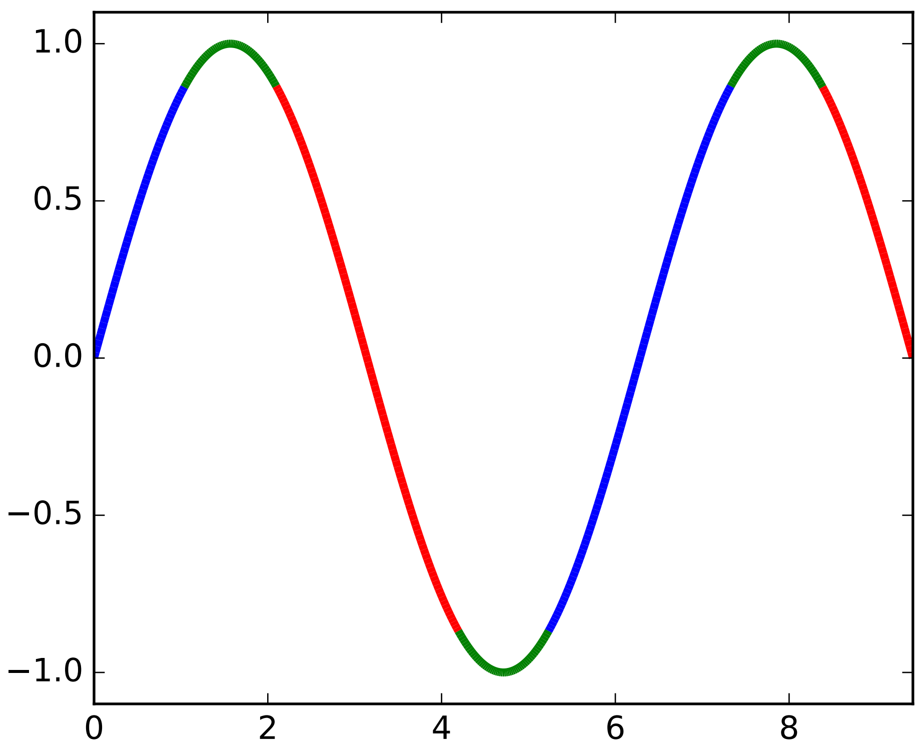

```
#!/usr/bin/env python
'''
Make a set of images with a single colormap, norm, and colorbar.

It also illustrates colorbar tick labelling with a multiplier.
'''

from matplotlib.pyplot import figure, show, axes, sci
from matplotlib import cm, colors
from matplotlib.font_manager import FontProperties
from numpy import amin, amax, ravel
from numpy.random import rand

Nr = 3
```

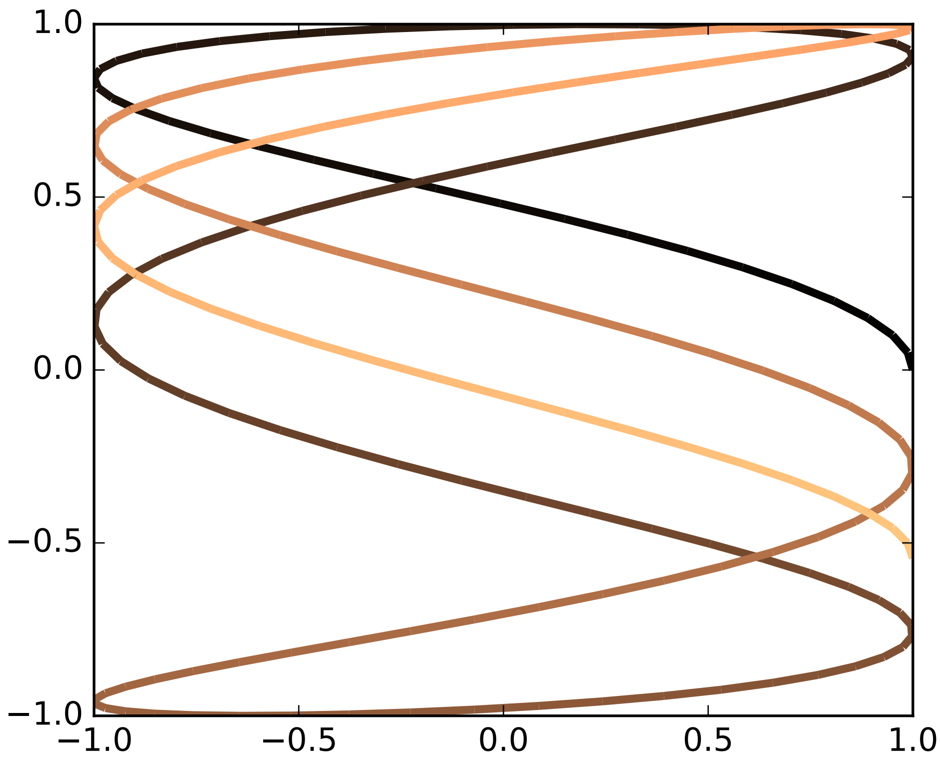
```
Nc = 2

fig = figure()
cmap = cm.cool

figtitle = 'Multiple images'
t = fig.text(0.5, 0.95, figtitle,
             horizontalalignment='center',
             fontproperties=FontProperties(size=16))

cax = fig.add_axes([0.2, 0.08, 0.6, 0.04])

w = 0.4
h = 0.22
ax = []
images = []
vmin = 1e40
vmax = -1e40
for i in range(Nr):
    for j in range(Nc):
        pos = [0.075 + j*1.1*w, 0.18 + i*1.2*h, w, h]
        a = fig.add_axes(pos)
        if i > 0:
            a.set_xticklabels([])
        # Make some fake data with a range that varies
        # somewhat from one plot to the next.
        data = ((1 + i + j)/10.0)*rand(10, 20)*1e-6
        dd = ravel(data)
        # Manually find the min and max of all colors for
        # use in setting the color scale.
        vmin = min(vmin, amin(dd))
        vmax = max(vmax, amax(dd))
        images.append(a.imshow(data, cmap=cmap))

        ax.append(a)

# Set the first image as the master, with all the others
# observing it for changes in cmap or norm.


class ImageFollower(object):
    'update image in response to changes in clim or cmap on another image'

    def __init__(self, follower):
        self.follower = follower

    def __call__(self, leader):
        self.follower.set_cmap(leader.get_cmap())
        self.follower.set_clim(leader.get_clim())

norm = colors.Normalize(vmin=vmin, vmax=vmax)
for i, im in enumerate(images):
    im.set_norm(norm)
```

```
    if i > 0:
        images[0].callbacksSM.connect('changed', ImageFollower(im))

# The colorbar is also based on this master image.
fig.colorbar(images[0], cax, orientation='horizontal')

# We need the following only if we want to run this interactively and
# modify the colormap:

axes(ax[0])     # Return the current axes to the first one,
sci(images[0])  # because the current image must be in current axes.

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.142 pylab_examples example code: multicolored_line.py



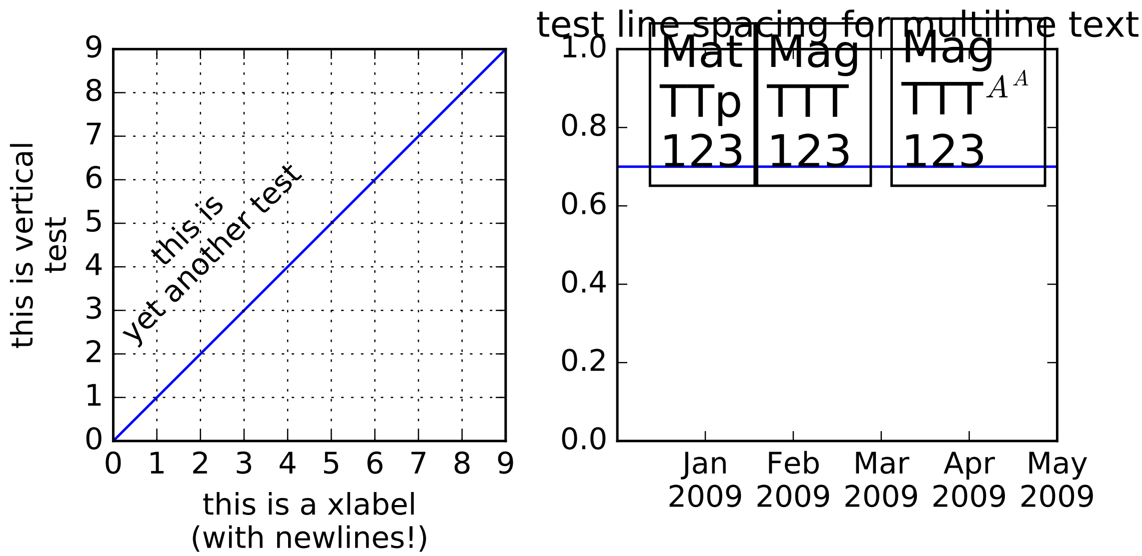



```
#!/usr/bin/env python
'''
Color parts of a line based on its properties, e.g., slope.
'''
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib.colors import ListedColormap, BoundaryNorm

x = np.linspace(0, 3 * np.pi, 500)
y = np.sin(x)
z = np.cos(0.5 * (x[:-1] + x[1:]))  # first derivative

# Create a colormap for red, green and blue and a norm to color
# f' < -0.5 red, f' > 0.5 blue, and the rest green
cmap = ListedColormap(['r', 'g', 'b'])
norm = BoundaryNorm([-1, -0.5, 0.5, 1], cmap.N)

# Create a set of line segments so that we can color them individually
# This creates the points as a N x 1 x 2 array so that we can stack points
# together easily to get the segments. The segments array for line collection
# needs to be numlines x points per line x 2 (x and y)
points = np.array([x, y]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)

# Create the line collection object, setting the colormapping parameters.
```

```
# Have to set the actual values used for colormapping separately.
lc = LineCollection(segments, cmap=cmap, norm=norm)
lc.set_array(z)
lc.set_linewidth(3)

fig1 = plt.figure()
plt.gca().add_collection(lc)
plt.xlim(x.min(), x.max())
plt.ylim(-1.1, 1.1)

# Now do a second plot coloring the curve using a continuous colormap
t = np.linspace(0, 10, 200)
x = np.cos(np.pi * t)
y = np.sin(t)
points = np.array([x, y]).T.reshape(-1, 1, 2)
segments = np.concatenate([points[:-1], points[1:]], axis=1)

lc = LineCollection(segments, cmap=plt.get_cmap('copper'),
                    norm=plt.Normalize(0, 10))
lc.set_array(t)
lc.set_linewidth(3)

fig2 = plt.figure()
plt.gca().add_collection(lc)
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.143 pylab_examples example code: multiline.py




```
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(7, 4))
ax = plt.subplot(121)
ax.set_aspect(1)
plt.plot(np.arange(10))
plt.xlabel('this is a xlabel\n(with newlines!)')
plt.ylabel('this is vertical\ntest', multialignment='center')
plt.text(2, 7, 'this is\nyet another test',
         rotation=45,
         horizontalalignment='center',
         verticalalignment='top',
         multialignment='center')

plt.grid(True)

plt.subplot(122)

plt.text(0.29, 0.7, "Mat\nTTp\n123", size=18,
         va="baseline", ha="right", multialignment="left",
         bbox=dict(fc="none"))

plt.text(0.34, 0.7, "Mag\nTTT\n123", size=18,
         va="baseline", ha="left", multialignment="left",
         bbox=dict(fc="none"))

plt.text(0.95, 0.7, "Mag\nTTT$^{A^A}$\n123", size=18,
```

```
         va="baseline", ha="right", multialignment="left",
         bbox=dict(fc="none"))

plt.xticks([0.2, 0.4, 0.6, 0.8, 1.],
           ["Jan\n2009", "Feb\n2009", "Mar\n2009", "Apr\n2009", "May\n2009"])

plt.axhline(0.7)
plt.title("test line spacing for multiline text")

plt.subplots_adjust(bottom=0.25, top=0.8)
plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

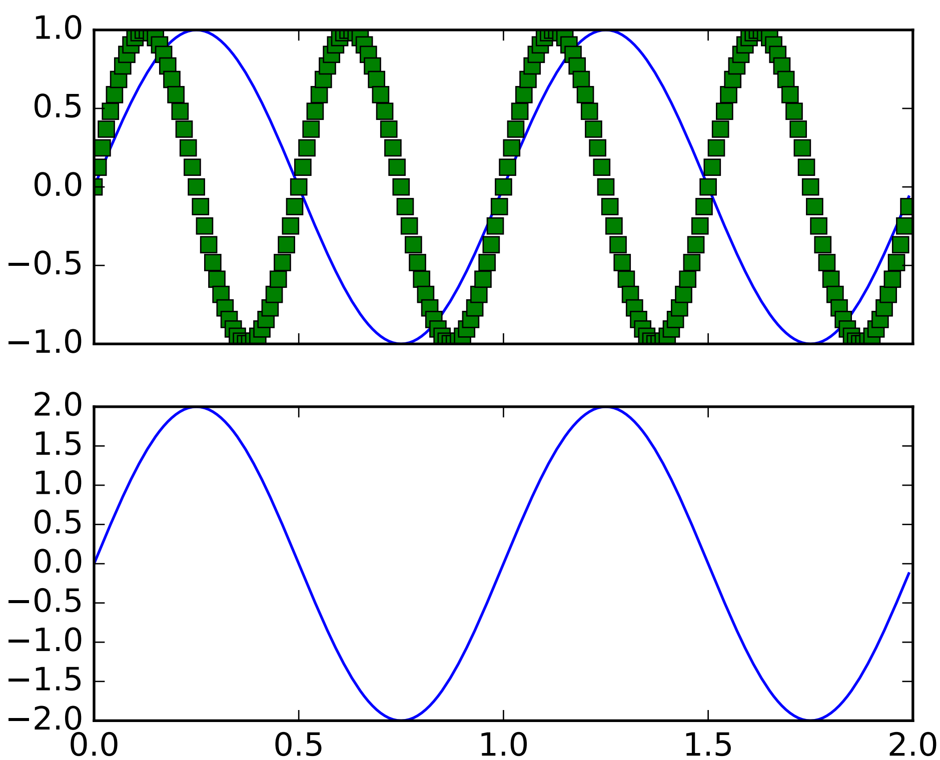
90.144 pylab_examples example code: multipage_pdf.py

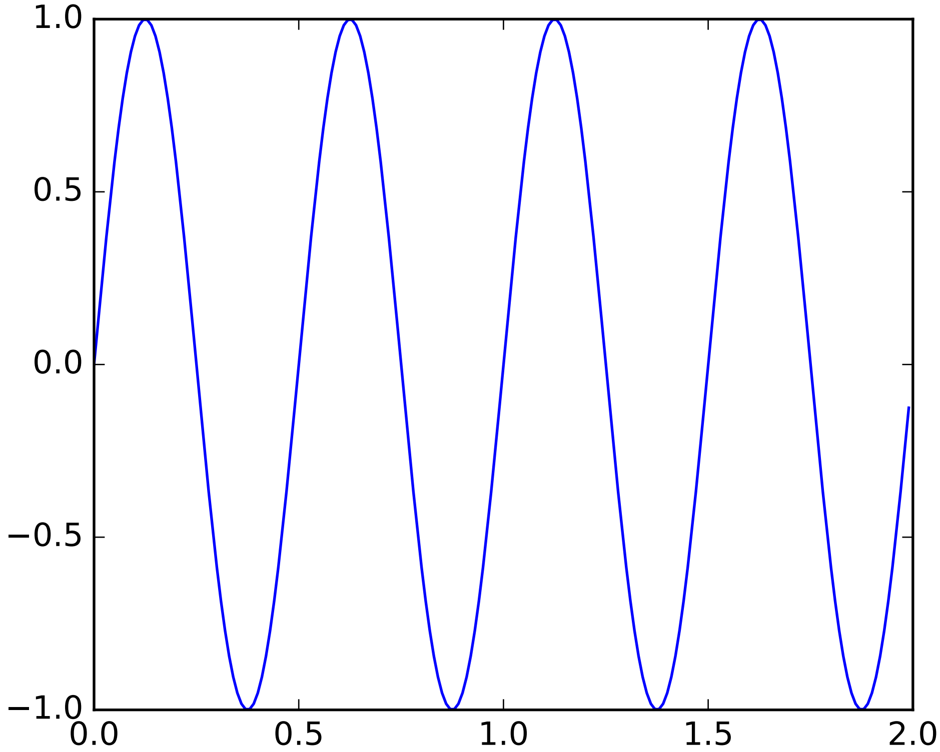
```
"""
This is a demo of creating a pdf file with several pages,
as well as adding metadata and annotations to pdf files.
"""

import datetime
import numpy as np
from matplotlib.backends.backend_pdf import PdfPages
import matplotlib.pyplot as plt

# Create the PdfPages object to which we will save the pages:
# The with statement makes sure that the PdfPages object is closed properly at
# the end of the block, even if an Exception occurs.
with PdfPages('multipage_pdf.pdf') as pdf:
    plt.figure(figsize=(3, 3))
    plt.plot(range(7), [3, 1, 4, 1, 5, 9, 2], 'r-o')
    plt.title('Page One')
    pdf.savefig()  # saves the current figure into a pdf page
    plt.close()

    plt.rc('text', usetex=True)
    plt.figure(figsize=(8, 6))
    x = np.arange(0, 5, 0.1)
    plt.plot(x, np.sin(x), 'b-')
    plt.title('Page Two')
    pdf.attach_note("plot of sin(x)")  # you can add a pdf note to
                                       # attach metadata to a page
    pdf.savefig()
    plt.close()

    plt.rc('text', usetex=False)
    fig = plt.figure(figsize=(4, 5))
    plt.plot(x, x*x, 'ko')
    plt.title('Page Three')
```

```
    pdf.savefig(fig)  # or you can pass a Figure object to pdf.savefig
    plt.close()

    # We can also set the file's metadata via the PdfPages object:
    d = pdf.infodict()
    d['Title'] = 'Multipage PDF Example'
    d['Author'] = u'Jouni K. Sepp\xe4nen'
    d['Subject'] = 'How to create a multipage pdf file and set its metadata'
    d['Keywords'] = 'PdfPages multipage keywords author title subject'
    d['CreationDate'] = datetime.datetime(2009, 11, 13)
    d['ModDate'] = datetime.datetime.today()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.145 pylab_examples example code: multiple_figs_demo.py

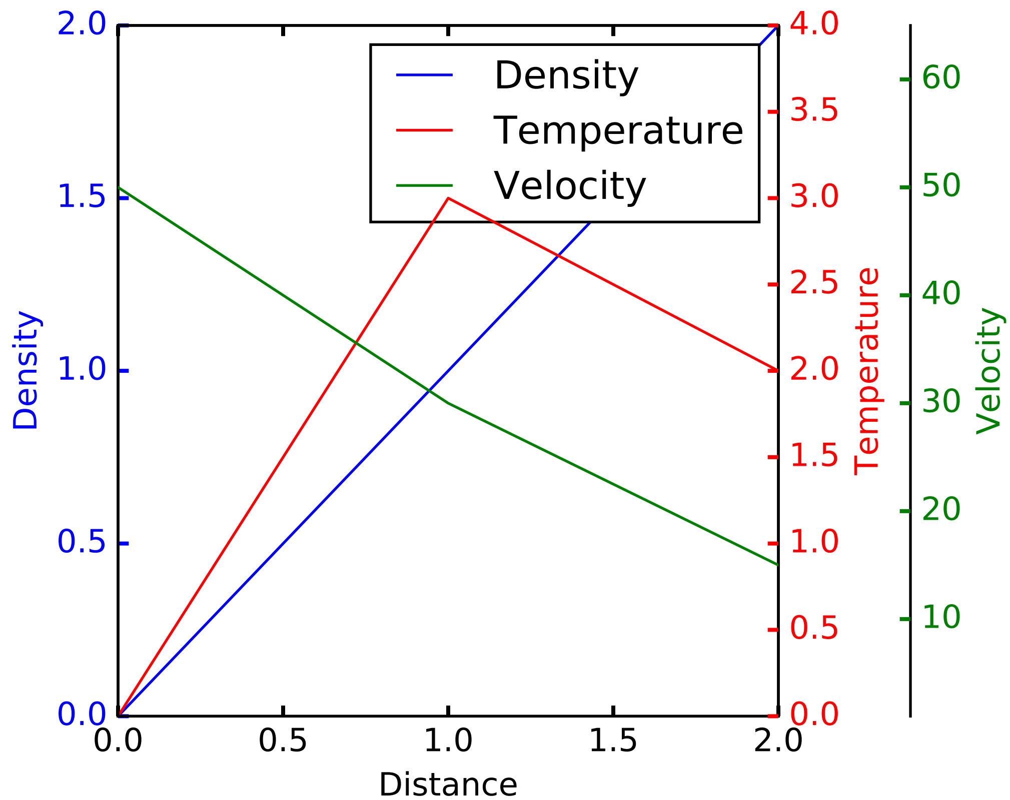





```
# Working with multiple figure windows and subplots
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(4*np.pi*t)

plt.figure(1)
plt.subplot(211)
plt.plot(t, s1)
plt.subplot(212)
plt.plot(t, 2*s1)

plt.figure(2)
plt.plot(t, s2)

# now switch back to figure 1 and make some changes
plt.figure(1)
plt.subplot(211)
plt.plot(t, s2, 'gs')
ax = plt.gca()
ax.set_xticklabels([])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.146 pylab_examples example code: multiple_yaxis_with_spines.py

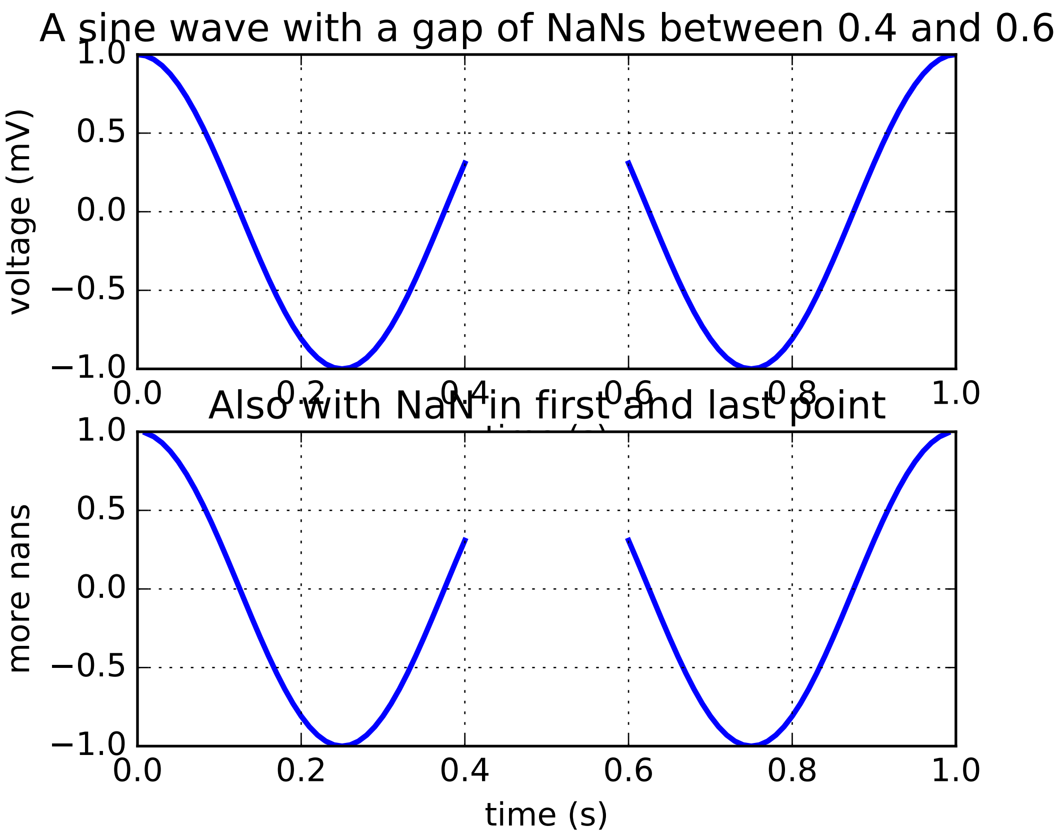



```
import matplotlib.pyplot as plt


def make_patch_spines_invisible(ax):
    ax.set_frame_on(True)
    ax.patch.set_visible(False)
    for sp in ax.spines.values():
        sp.set_visible(False)

fig, host = plt.subplots()
fig.subplots_adjust(right=0.75)

par1 = host.twinx()
par2 = host.twinx()

# Offset the right spine of par2.  The ticks and label have already been
# placed on the right by twinx above.
par2.spines["right"].set_position(("axes", 1.2))
# Having been created by twinx, par2 has its frame off, so the line of its
```

```
# detached spine is invisible.  First, activate the frame but make the patch
# and spines invisible.
make_patch_spines_invisible(par2)
# Second, show the right spine.
par2.spines["right"].set_visible(True)

p1, = host.plot([0, 1, 2], [0, 1, 2], "b-", label="Density")
p2, = par1.plot([0, 1, 2], [0, 3, 2], "r-", label="Temperature")
p3, = par2.plot([0, 1, 2], [50, 30, 15], "g-", label="Velocity")

host.set_xlim(0, 2)
host.set_ylim(0, 2)
par1.set_ylim(0, 4)
par2.set_ylim(1, 65)

host.set_xlabel("Distance")
host.set_ylabel("Density")
par1.set_ylabel("Temperature")
par2.set_ylabel("Velocity")

host.yaxis.label.set_color(p1.get_color())
par1.yaxis.label.set_color(p2.get_color())
par2.yaxis.label.set_color(p3.get_color())

tkw = dict(size=4, width=1.5)
host.tick_params(axis='y', colors=p1.get_color(), **tkw)
par1.tick_params(axis='y', colors=p2.get_color(), **tkw)
par2.tick_params(axis='y', colors=p3.get_color(), **tkw)
host.tick_params(axis='x', **tkw)

lines = [p1, p2, p3]

host.legend(lines, [l.get_label() for l in lines])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.147 pylab_examples example code: nan_test.py

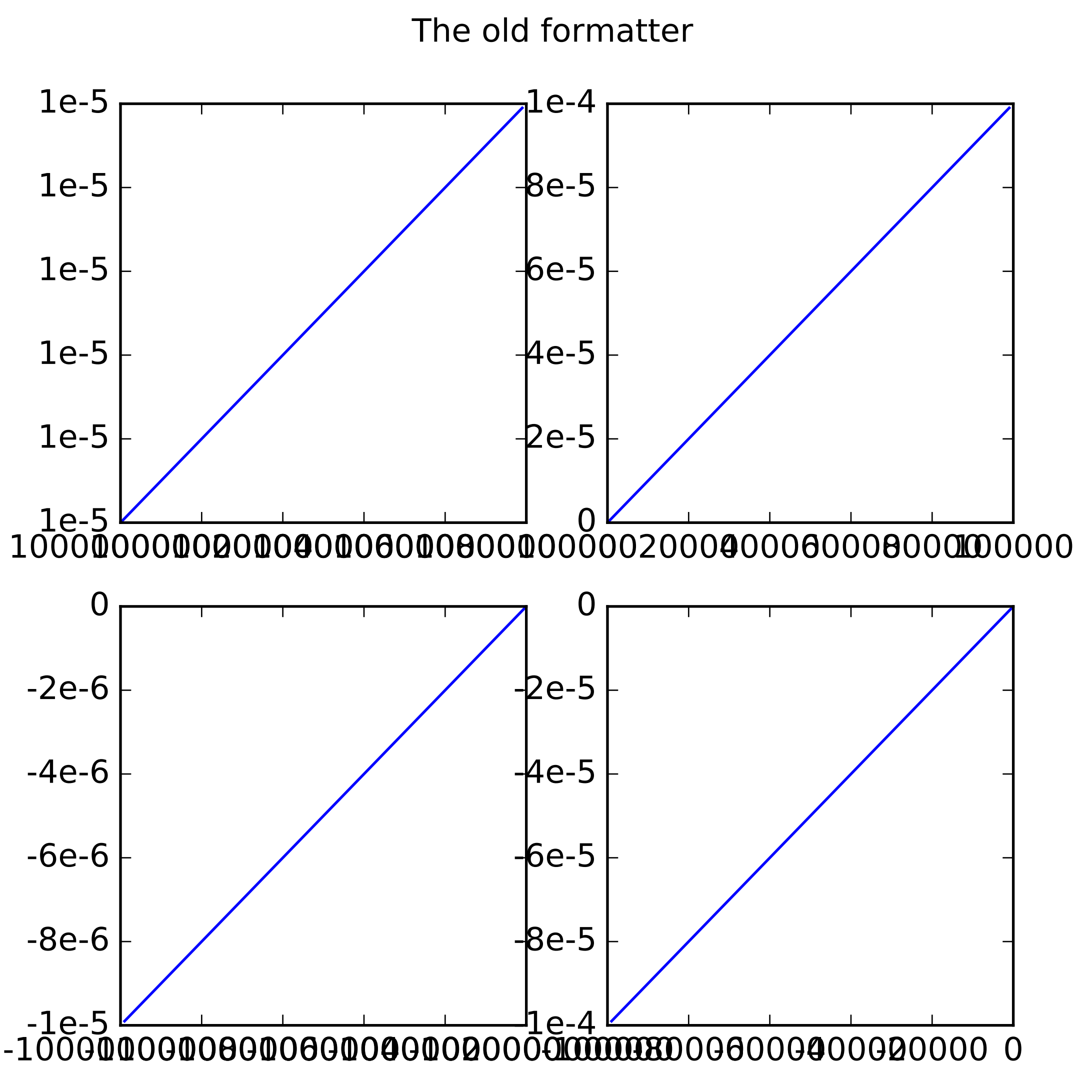


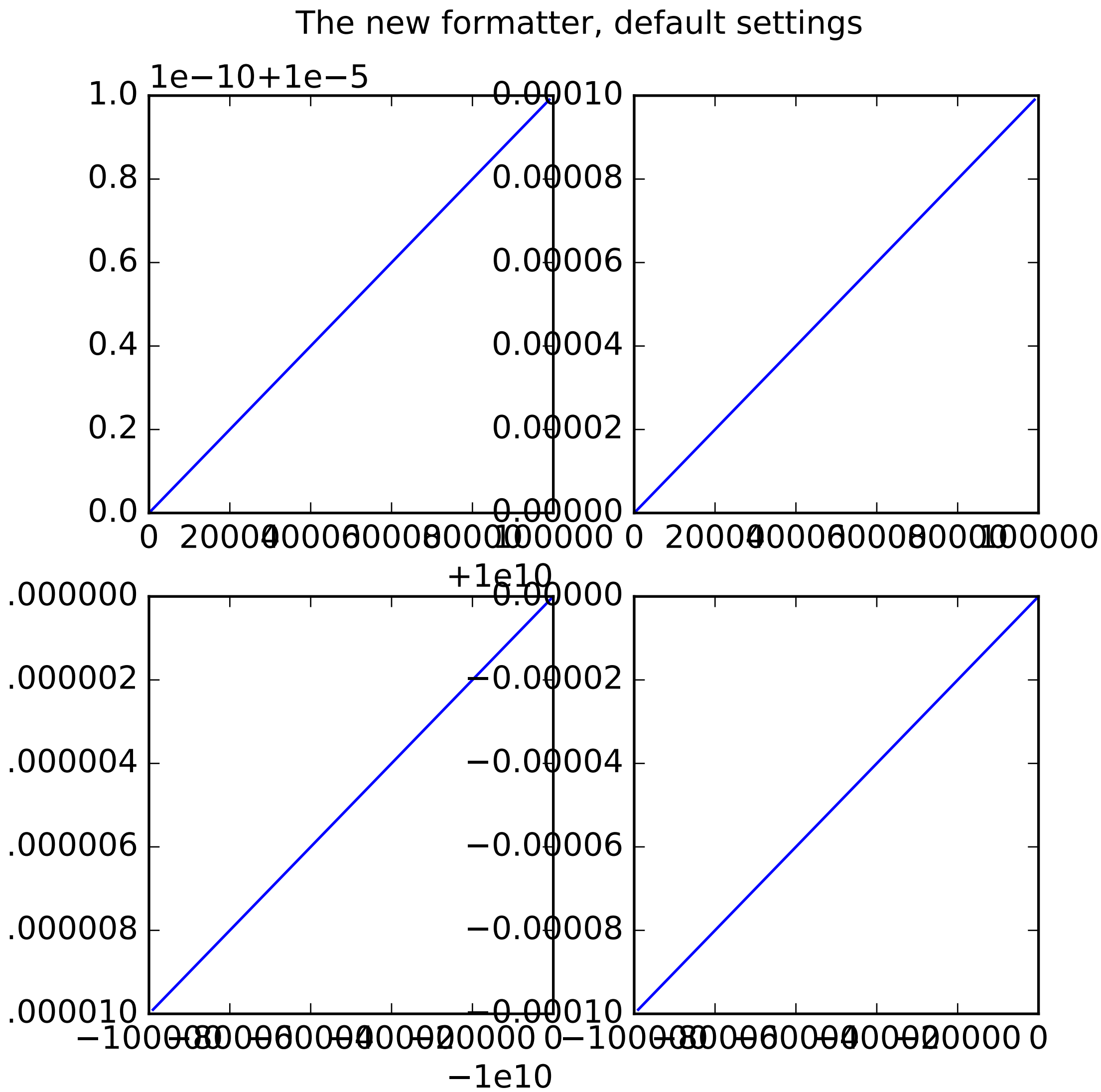

```python
"""
Example: simple line plots with NaNs inserted.
"""
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(2 * 2 * np.pi * t)
t[41:60] = np.nan

plt.subplot(2, 1, 1)
plt.plot(t, s, '-', lw=2)

plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('A sine wave with a gap of NaNs between 0.4 and 0.6')
plt.grid(True)

plt.subplot(2, 1, 2)
t[0] = np.nan
t[-1] = np.nan
plt.plot(t, s, '-', lw=2)
```

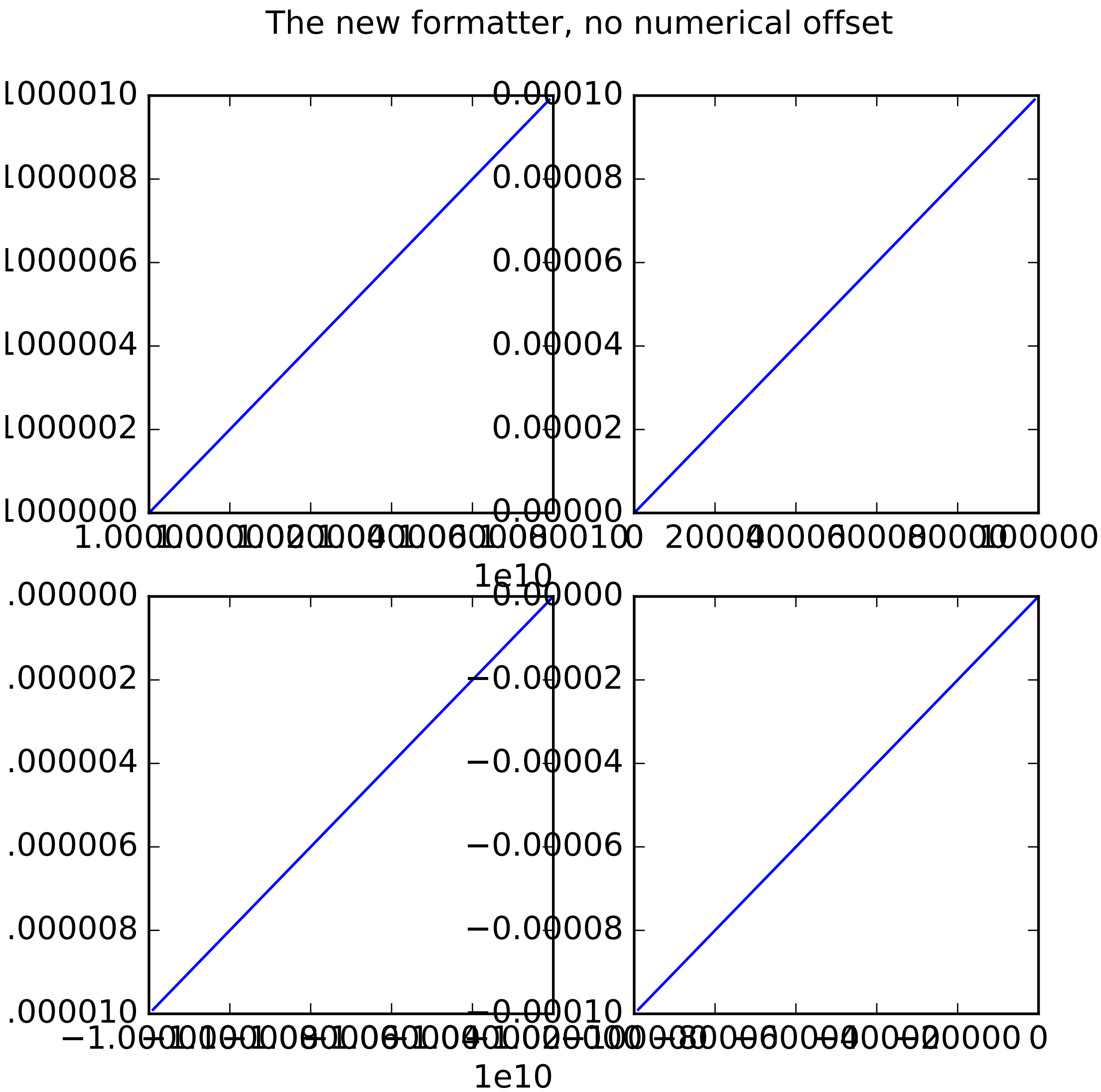
```
plt.title('Also with NaN in first and last point')

plt.xlabel('time (s)')
plt.ylabel('more nans')
plt.grid(True)

plt.show()
```

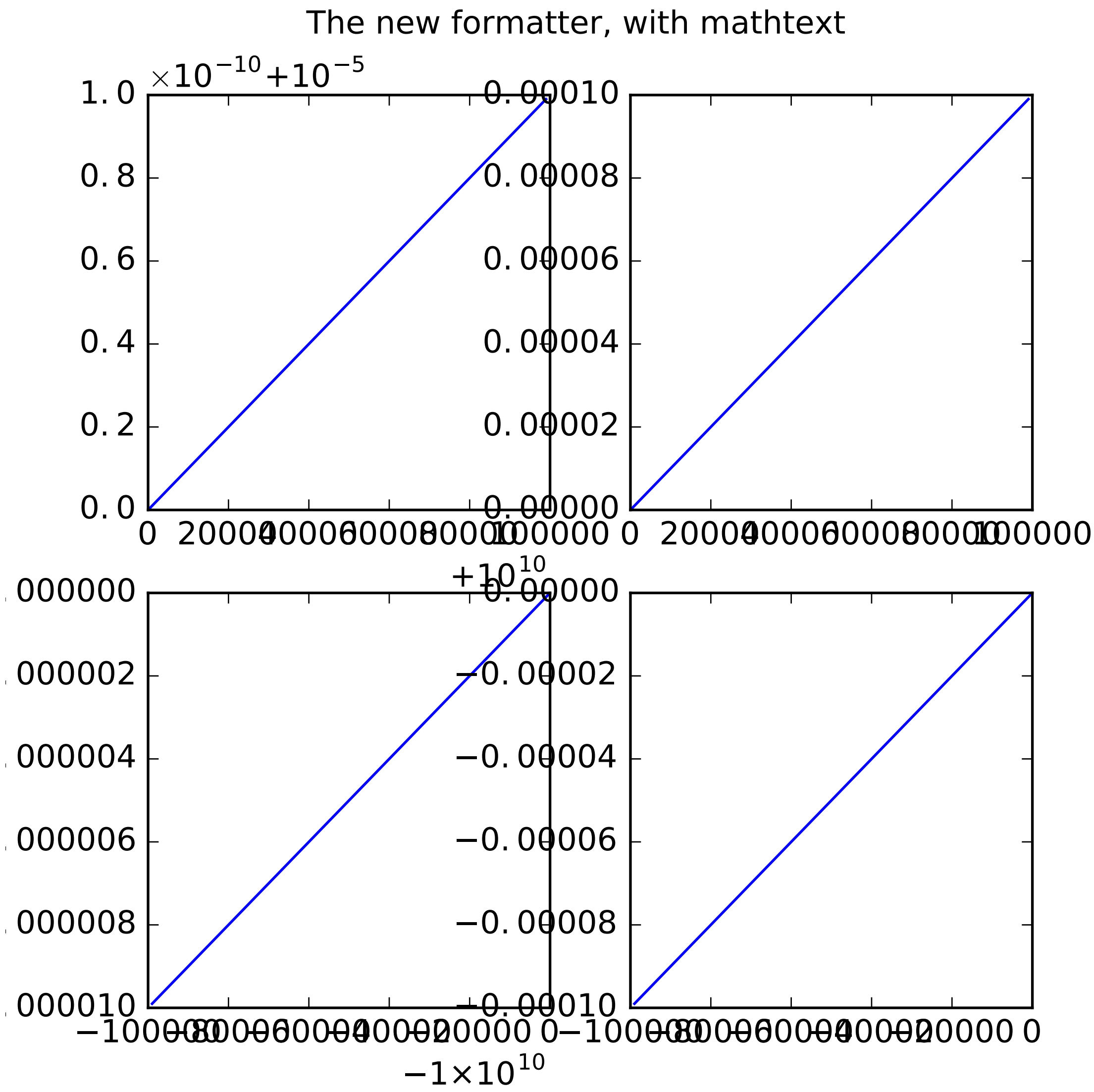
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.148 pylab_examples example code: newscalarformatter_demo.py




The new formatter, default settings
1e−10+1e−5
1.0
0.8
0.6
0.4
0.2
0.0
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
+1e10
.000000
.000002
.000004
.000006
.000008
.000010
−0.00002
−0.00004
−0.00006
−0.00008
−0.00010
−1e10


The new formatter, no numerical offset
1e10
1e10



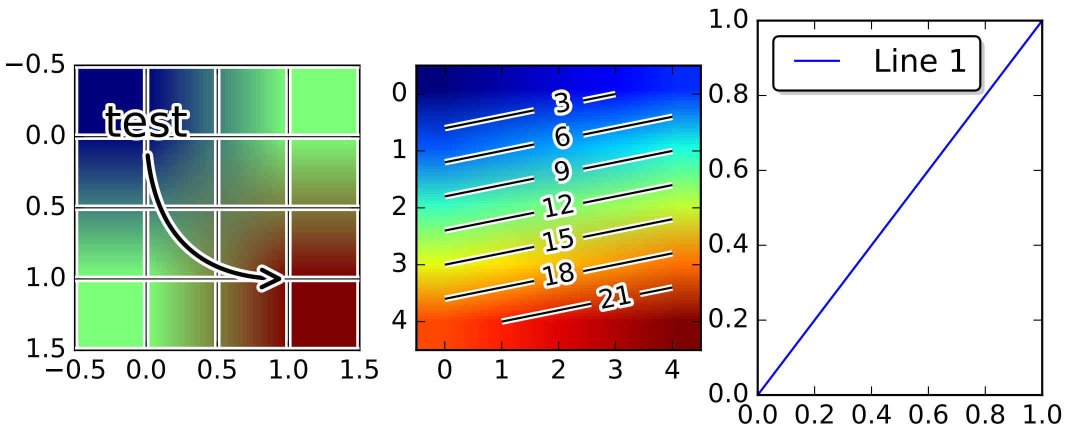

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import OldScalarFormatter, ScalarFormatter

# Example 1
x = np.arange(0, 1, .01)
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2, figsize=(6, 6))
fig.text(0.5, 0.975, 'The old formatter',
         horizontalalignment='center', verticalalignment='top')
ax1.plot(x * 1e5 + 1e10, x * 1e-10 + 1e-5)
ax1.xaxis.set_major_formatter(OldScalarFormatter())
ax1.yaxis.set_major_formatter(OldScalarFormatter())

ax2.plot(x * 1e5, x * 1e-4)
ax2.xaxis.set_major_formatter(OldScalarFormatter())
ax2.yaxis.set_major_formatter(OldScalarFormatter())
```

```
ax3.plot(-x * 1e5 - 1e10, -x * 1e-5 - 1e-10)
ax3.xaxis.set_major_formatter(OldScalarFormatter())
ax3.yaxis.set_major_formatter(OldScalarFormatter())

ax4.plot(-x * 1e5, -x * 1e-4)
ax4.xaxis.set_major_formatter(OldScalarFormatter())
ax4.yaxis.set_major_formatter(OldScalarFormatter())

# Example 2
x = np.arange(0, 1, .01)
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2, figsize=(6, 6))
fig.text(0.5, 0.975, 'The new formatter, default settings',
         horizontalalignment='center',
         verticalalignment='top')

ax1.plot(x * 1e5 + 1e10, x * 1e-10 + 1e-5)
ax1.xaxis.set_major_formatter(ScalarFormatter())
ax1.yaxis.set_major_formatter(ScalarFormatter())

ax2.plot(x * 1e5, x * 1e-4)
ax2.xaxis.set_major_formatter(ScalarFormatter())
ax2.yaxis.set_major_formatter(ScalarFormatter())

ax3.plot(-x * 1e5 - 1e10, -x * 1e-5 - 1e-10)
ax3.xaxis.set_major_formatter(ScalarFormatter())
ax3.yaxis.set_major_formatter(ScalarFormatter())

ax4.plot(-x * 1e5, -x * 1e-4)
ax4.xaxis.set_major_formatter(ScalarFormatter())
ax4.yaxis.set_major_formatter(ScalarFormatter())

# Example 3
x = np.arange(0, 1, .01)
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2, figsize=(6, 6))
fig.text(0.5, 0.975, 'The new formatter, no numerical offset',
         horizontalalignment='center',
         verticalalignment='top')

ax1.plot(x * 1e5 + 1e10, x * 1e-10 + 1e-5)
ax1.xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
ax1.yaxis.set_major_formatter(ScalarFormatter(useOffset=False))

ax2.plot(x * 1e5, x * 1e-4)
ax2.xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
ax2.yaxis.set_major_formatter(ScalarFormatter(useOffset=False))

ax3.plot(-x * 1e5 - 1e10, -x * 1e-5 - 1e-10)
ax3.xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
ax3.yaxis.set_major_formatter(ScalarFormatter(useOffset=False))

ax4.plot(-x * 1e5, -x * 1e-4)
ax4.xaxis.set_major_formatter(ScalarFormatter(useOffset=False))
```

```
ax4.yaxis.set_major_formatter(ScalarFormatter(useOffset=False))

# Example 4
x = np.arange(0, 1, .01)
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2, figsize=(6, 6))
fig.text(0.5, 0.975, 'The new formatter, with mathtext',
         horizontalalignment='center',
         verticalalignment='top')

ax1.plot(x * 1e5 + 1e10, x * 1e-10 + 1e-5)
ax1.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
ax1.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

ax2.plot(x * 1e5, x * 1e-4)
ax2.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
ax2.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

ax3.plot(-x * 1e5 - 1e10, -x * 1e-5 - 1e-10)
ax3.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
ax3.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))

ax4.plot(-x * 1e5, -x * 1e-4)
ax4.xaxis.set_major_formatter(ScalarFormatter(useMathText=True))
ax4.yaxis.set_major_formatter(ScalarFormatter(useMathText=True))
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.149 pylab_examples example code: patheffect_demo.py

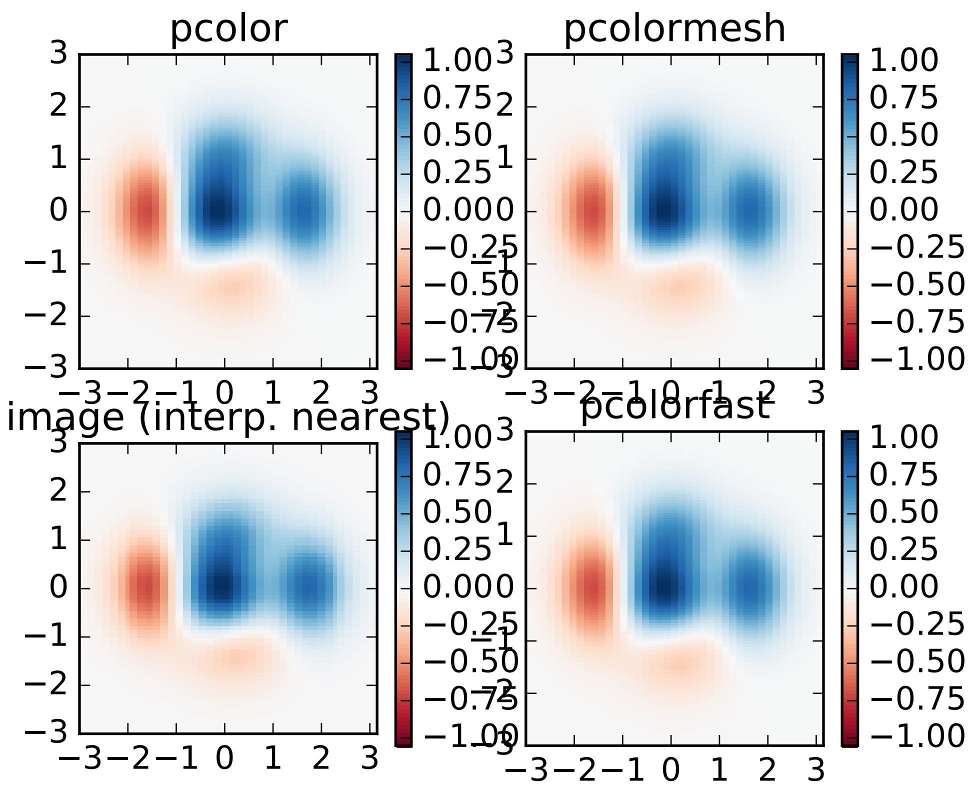



```
import matplotlib.pyplot as plt
import matplotlib.patheffects as PathEffects
import numpy as np

if 1:
```

```
    plt.figure(1, figsize=(8, 3))
    ax1 = plt.subplot(131)
    ax1.imshow([[1, 2], [2, 3]])
    txt = ax1.annotate("test", (1., 1.), (0., 0),
                       arrowprops=dict(arrowstyle="->",
                                       connectionstyle="angle3", lw=2),
                       size=20, ha="center", path_effects=[PathEffects.
↪withStroke(linewidth=3,
                                                                          ␣
↪foreground="w")])
    txt.arrow_patch.set_path_effects([
        PathEffects.Stroke(linewidth=5, foreground="w"),
        PathEffects.Normal()])

    ax1.grid(True, linestyle="-")

    pe = [PathEffects.withStroke(linewidth=3,
                                 foreground="w")]
    for l in ax1.get_xgridlines() + ax1.get_ygridlines():
        l.set_path_effects(pe)

    ax2 = plt.subplot(132)
    arr = np.arange(25).reshape((5, 5))
    ax2.imshow(arr)
    cntr = ax2.contour(arr, colors="k")

    plt.setp(cntr.collections, path_effects=[
        PathEffects.withStroke(linewidth=3, foreground="w")])

    clbls = ax2.clabel(cntr, fmt="%2.0f", use_clabeltext=True)
    plt.setp(clbls, path_effects=[
        PathEffects.withStroke(linewidth=3, foreground="w")])

    # shadow as a path effect
    ax3 = plt.subplot(133)
    p1, = ax3.plot([0, 1], [0, 1])
    leg = ax3.legend([p1], ["Line 1"], fancybox=True, loc=2)
    leg.legendPatch.set_path_effects([PathEffects.withSimplePatchShadow()])

    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.150 pylab_examples example code: pcolor_demo.py

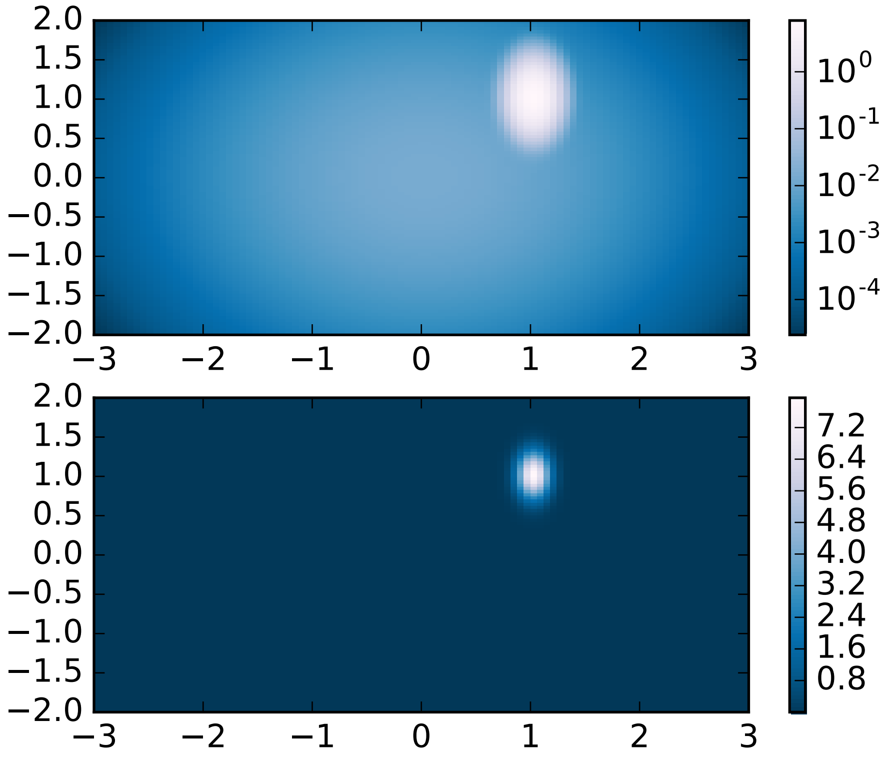



```
"""
Demonstrates similarities between pcolor, pcolormesh, imshow and pcolorfast
for drawing quadrilateral grids.

"""
import matplotlib.pyplot as plt
import numpy as np

# make these smaller to increase the resolution
dx, dy = 0.15, 0.05

# generate 2 2d grids for the x & y bounds
y, x = np.mgrid[slice(-3, 3 + dy, dy),
                slice(-3, 3 + dx, dx)]
z = (1 - x / 2. + x ** 5 + y ** 3) * np.exp(-x ** 2 - y ** 2)
# x and y are bounds, so z should be the value *inside* those bounds.
# Therefore, remove the last value from the z array.
z = z[:-1, :-1]
z_min, z_max = -np.abs(z).max(), np.abs(z).max()


plt.subplot(2, 2, 1)
```

```
plt.pcolor(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
plt.title('pcolor')
# set the limits of the plot to the limits of the data
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.colorbar()


plt.subplot(2, 2, 2)
plt.pcolormesh(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
plt.title('pcolormesh')
# set the limits of the plot to the limits of the data
plt.axis([x.min(), x.max(), y.min(), y.max()])
plt.colorbar()


plt.subplot(2, 2, 3)
plt.imshow(z, cmap='RdBu', vmin=z_min, vmax=z_max,
           extent=[x.min(), x.max(), y.min(), y.max()],
           interpolation='nearest', origin='lower')
plt.title('image (interp. nearest)')
plt.colorbar()


ax = plt.subplot(2, 2, 4)
ax.pcolorfast(x, y, z, cmap='RdBu', vmin=z_min, vmax=z_max)
plt.title('pcolorfast')
plt.colorbar()


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.151 pylab_examples example code: pcolor_log.py

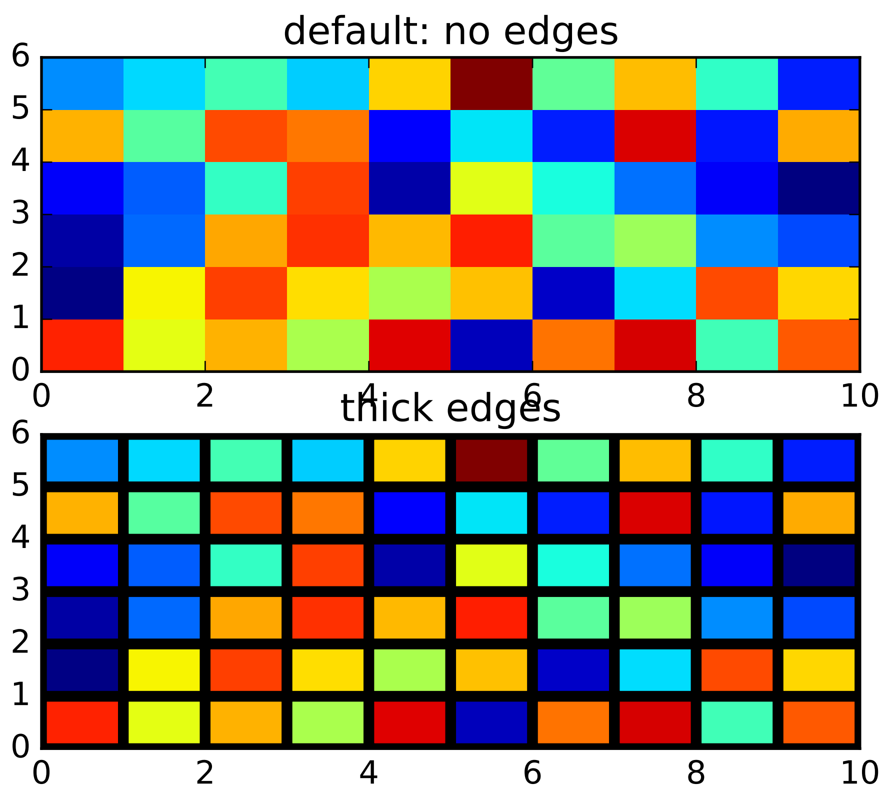



```
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import numpy as np
from matplotlib.mlab import bivariate_normal


N = 100
X, Y = np.mgrid[-3:3:complex(0, N), -2:2:complex(0, N)]

# A low hump with a spike coming out of the top right.
# Needs to have z/colour axis on a log scale so we see both hump and spike.
# linear scale only shows the spike.
Z1 = bivariate_normal(X, Y, 0.1, 0.2, 1.0, 1.0) + 0.1 * bivariate_normal(X, Y, 1.0, 1.0,
    0.0, 0.0)

plt.subplot(2, 1, 1)
plt.pcolor(X, Y, Z1, norm=LogNorm(vmin=Z1.min(), vmax=Z1.max()), cmap='PuBu_r')
plt.colorbar()

plt.subplot(2, 1, 2)
plt.pcolor(X, Y, Z1, cmap='PuBu_r')
plt.colorbar()
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

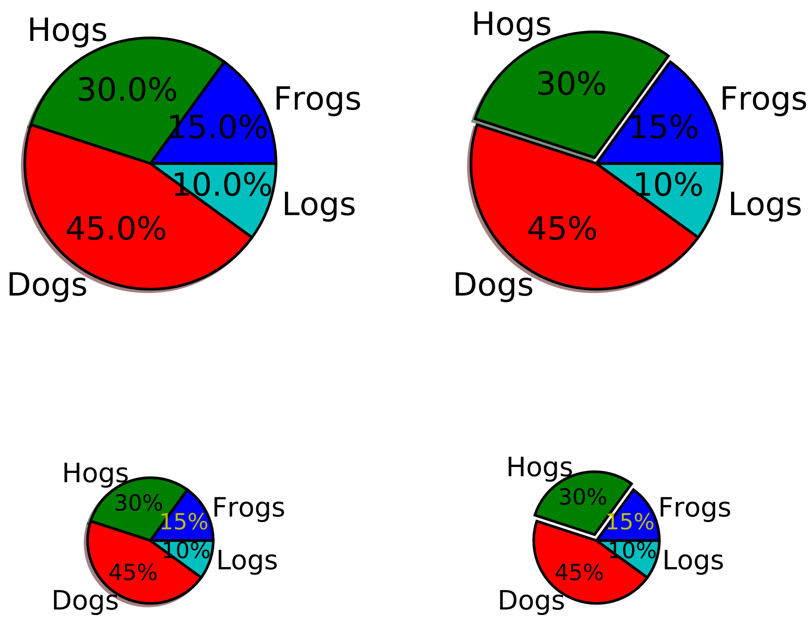
90.152 pylab_examples example code: pcolor_small.py




```
import matplotlib.pyplot as plt
from numpy.random import rand

Z = rand(6, 10)

plt.subplot(2, 1, 1)
c = plt.pcolor(Z)
plt.title('default: no edges')

plt.subplot(2, 1, 2)
c = plt.pcolor(Z, edgecolors='k', linewidths=4)
plt.title('thick edges')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.153 pylab_examples example code: pie_demo2.py

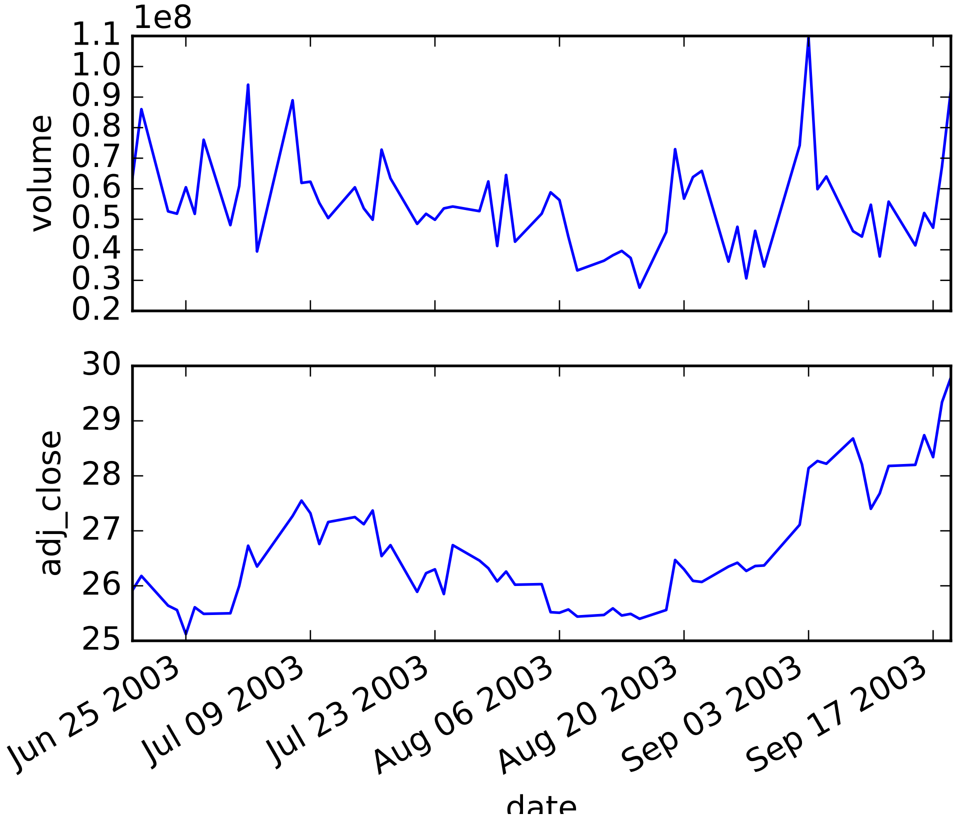


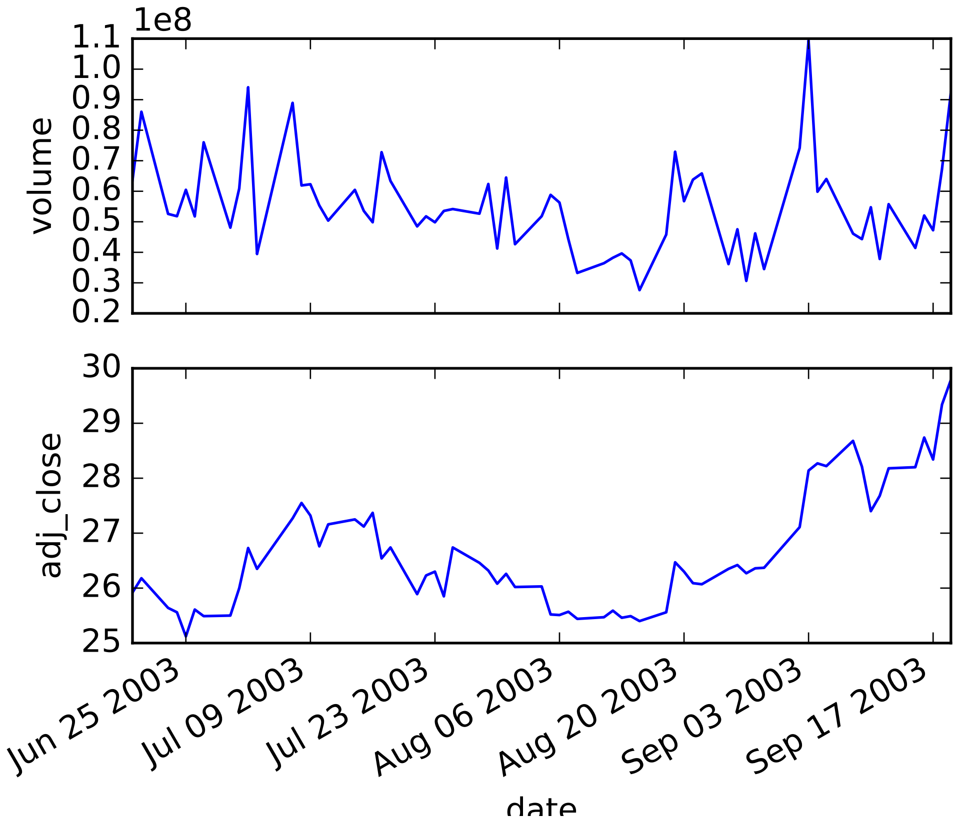

```
"""
Make a pie charts of varying size - see
http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.pie for the docstring.

This example shows a basic pie charts with labels optional features,
like autolabeling the percentage, offsetting a slice with "explode"
and adding a shadow, in different sizes.

"""
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

# Some data

labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
fracs = [15, 30, 45, 10]

explode = (0, 0.05, 0, 0)
```

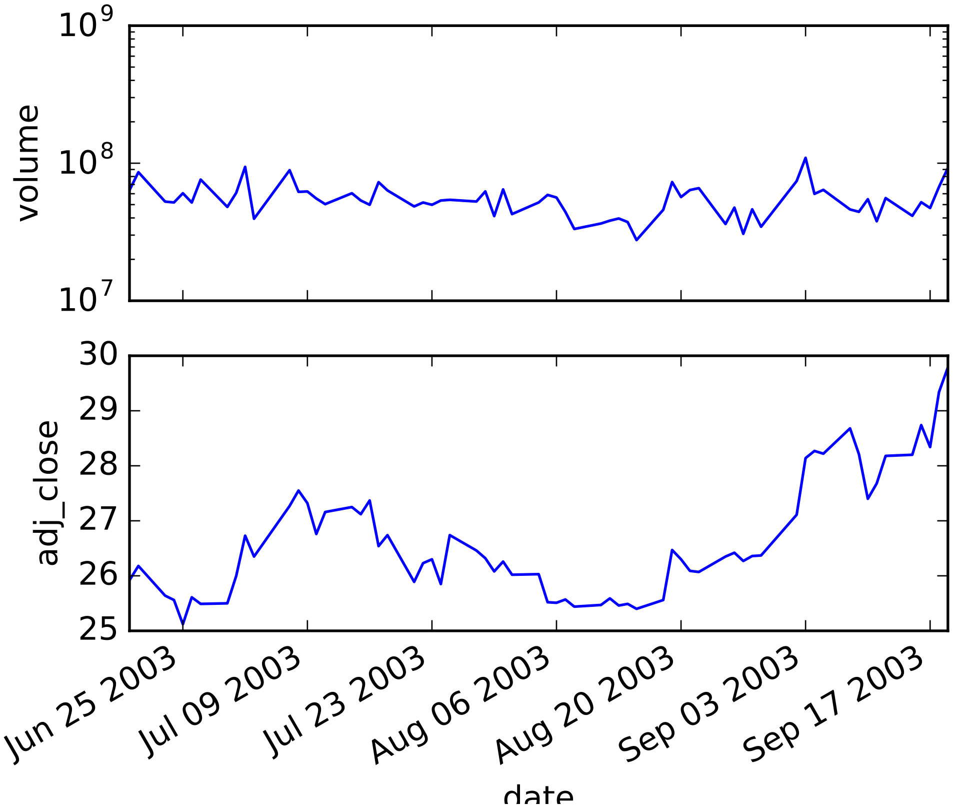
```
# Make square figures and axes

the_grid = GridSpec(2, 2)

plt.subplot(the_grid[0, 0], aspect=1)

plt.pie(fracs, labels=labels, autopct='%1.1f%%', shadow=True)

plt.subplot(the_grid[0, 1], aspect=1)

plt.pie(fracs, explode=explode, labels=labels, autopct='%.0f%%', shadow=True)

plt.subplot(the_grid[1, 0], aspect=1)

patches, texts, autotexts = plt.pie(fracs, labels=labels,
                                    autopct='%.0f%%',
                                    shadow=True, radius=0.5)

# Make the labels on the small plot easier to read.
for t in texts:
    t.set_size('smaller')
for t in autotexts:
    t.set_size('x-small')
autotexts[0].set_color('y')

plt.subplot(the_grid[1, 1], aspect=1)

# Turn off shadow for tiny plot with exploded slice.
patches, texts, autotexts = plt.pie(fracs, explode=explode,
                                    labels=labels, autopct='%.0f%%',
                                    shadow=False, radius=0.5)
for t in texts:
    t.set_size('smaller')
for t in autotexts:
    t.set_size('x-small')
autotexts[0].set_color('y')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

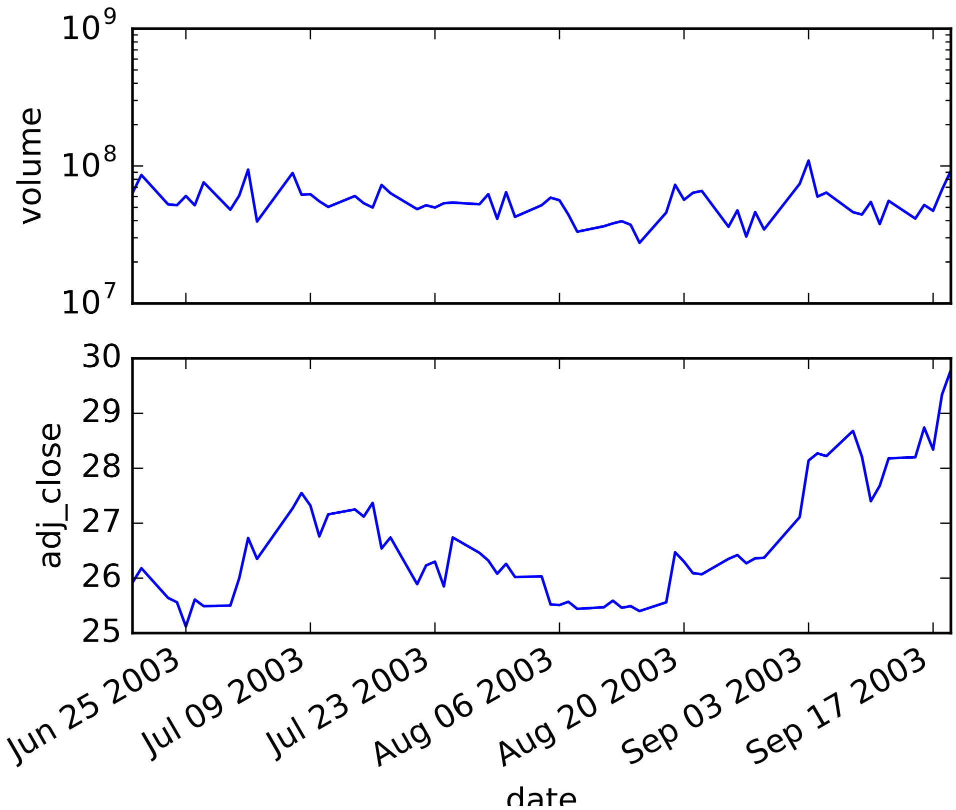
90.154 pylab_examples example code: plotfile_demo.py



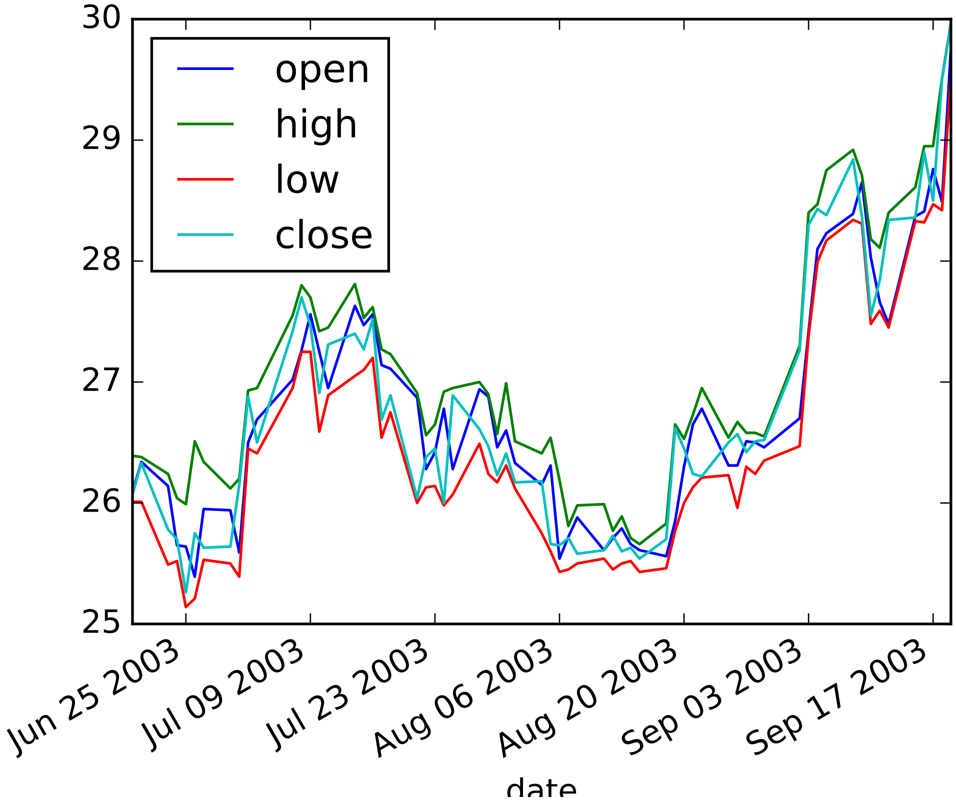

1e8
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
volume
30
29
28
27
26
25
adj_close
Jun 25 2003
Jul 09 2003
Jul 23 2003
Aug 06 2003
Aug 20 2003
Sep 03 2003
Sep 17 2003
date

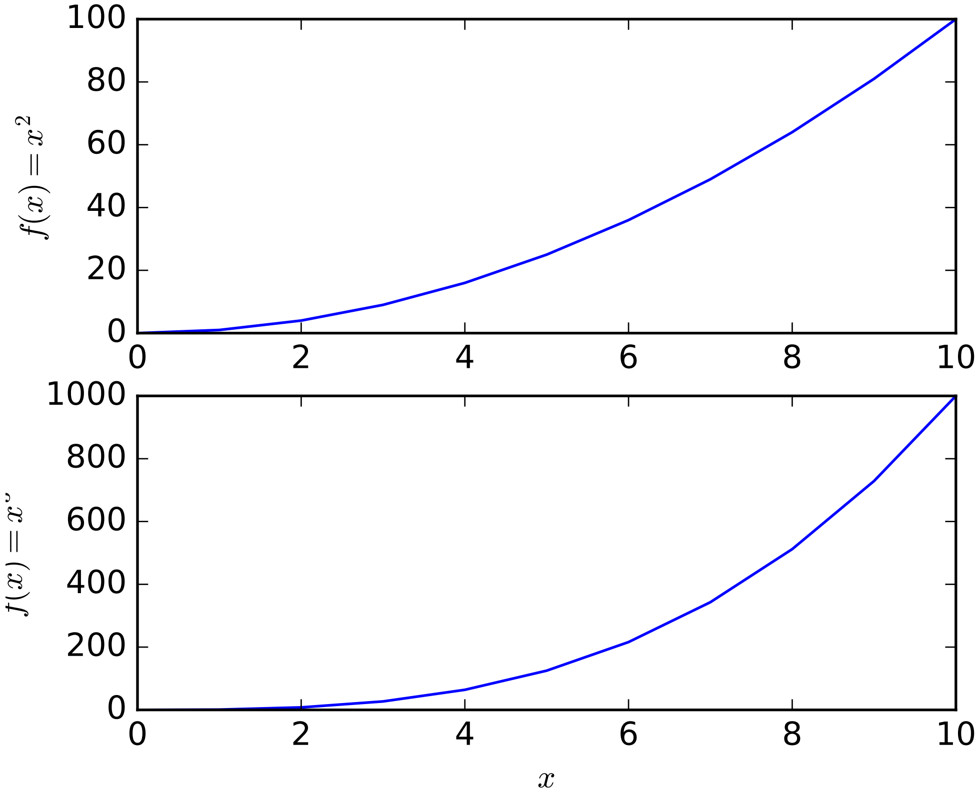


volume
10^9
10^8
10^7
adj_close
30
29
28
27
26
25
Jun 25 2003
Jul 09 2003
Jul 23 2003
Aug 06 2003
Aug 20 2003
Sep 03 2003
Sep 17 2003
date

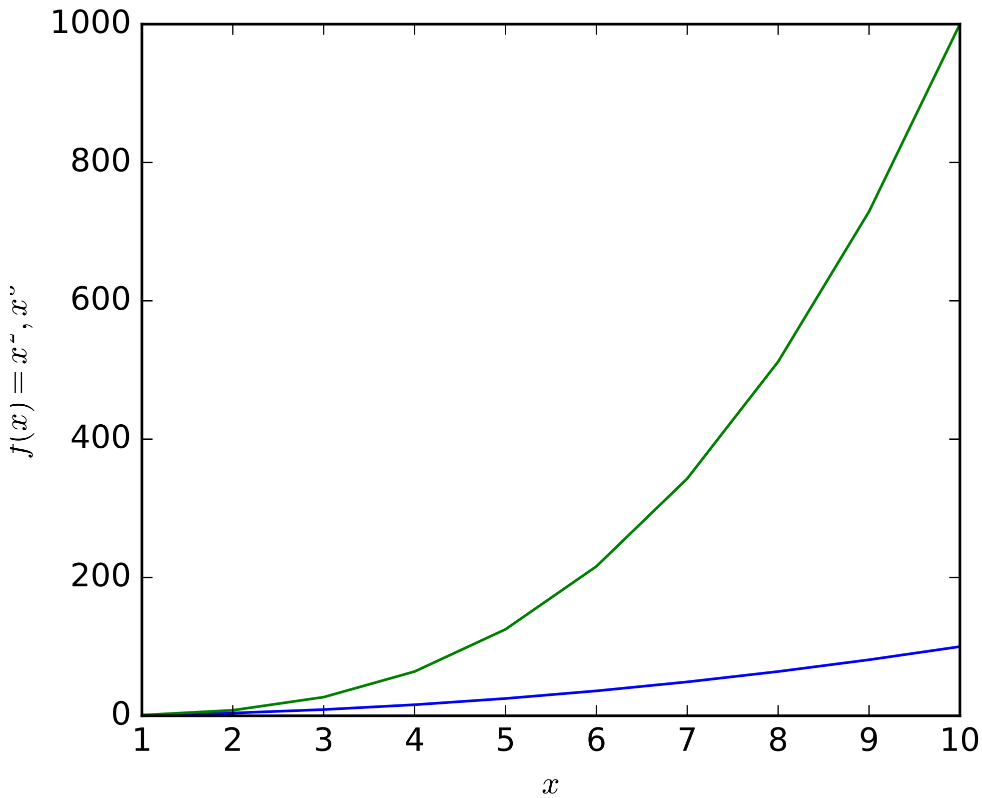


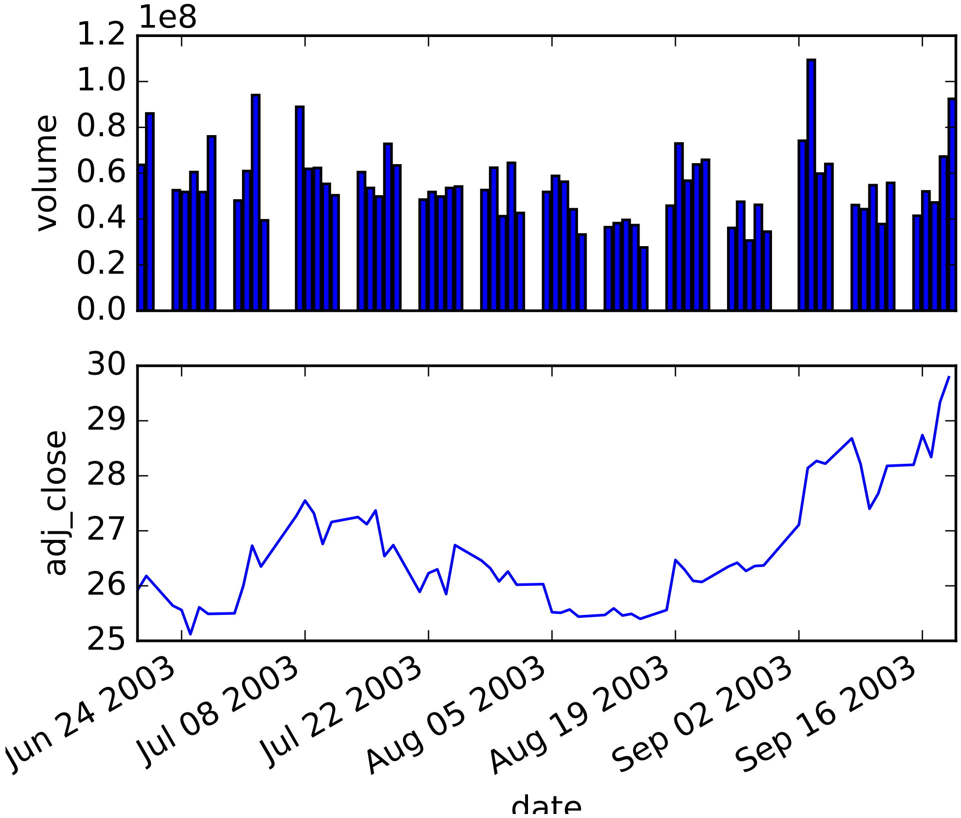
10^9
10^8
10^7
volume
30
29
28
27
26
25
adj_close
Jun 25 2003
Jul 09 2003
Jul 23 2003
Aug 06 2003
Aug 20 2003
Sep 03 2003
Sep 17 2003
date


open
high
low
close
30
29
28
27
26
25
Jun 25 2003
Jul 09 2003
Jul 23 2003
Aug 06 2003
Aug 20 2003
Sep 03 2003
Sep 17 2003
date



100
80
60
40
20
0
$f(x) = x^2$
0
2
4
6
8
10
1000
800
600
400
200
0
$f(x) = x^3$
0
2
4
6
8
10
x


1000
800
600
400
200
0
1
2
3
4
5
6
7
8
9
10
x
$f(x) = x^2, x^3$

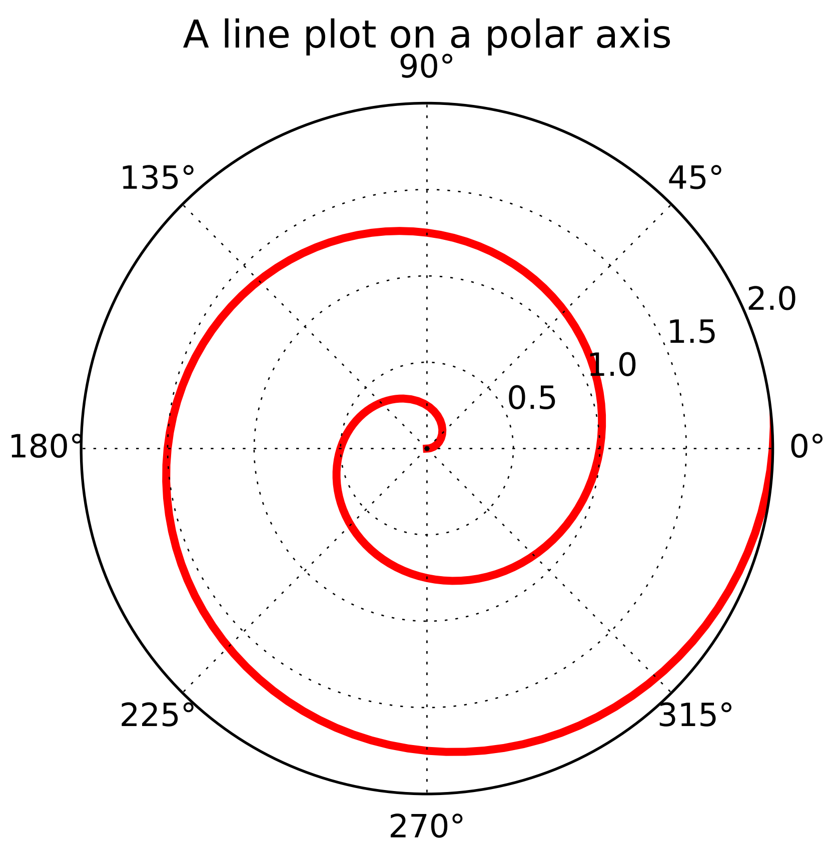



```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib.cbook as cbook

fname = cbook.get_sample_data('msft.csv', asfileobj=False)
fname2 = cbook.get_sample_data('data_x_x2_x3.csv', asfileobj=False)

# test 1; use ints
plt.plotfile(fname, (0, 5, 6))

# test 2; use names
plt.plotfile(fname, ('date', 'volume', 'adj_close'))

# test 3; use semilogy for volume
plt.plotfile(fname, ('date', 'volume', 'adj_close'),
             plotfuncs={'volume': 'semilogy'})

# test 4; use semilogy for volume
plt.plotfile(fname, (0, 5, 6), plotfuncs={5: 'semilogy'})

# test 5; single subplot
plt.plotfile(fname, ('date', 'open', 'high', 'low', 'close'), subplots=False)

# test 6; labeling, if no names in csv-file
plt.plotfile(fname2, cols=(0, 1, 2), delimiter=' ',
```

```
            names=['$x$', '$f(x)=x^2$', '$f(x)=x^3$'])

# test 7; more than one file per figure--illustrated here with a single file
plt.plotfile(fname2, cols=(0, 1), delimiter=' ')
plt.plotfile(fname2, cols=(0, 2), newfig=False,
             delimiter=' ')  # use current figure
plt.xlabel(r'$x$')
plt.ylabel(r'$f(x) = x^2, x^3$')

# test 8; use bar for volume
plt.plotfile(fname, (0, 5, 6), plotfuncs={5: 'bar'})

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.155 pylab_examples example code: polar_demo.py

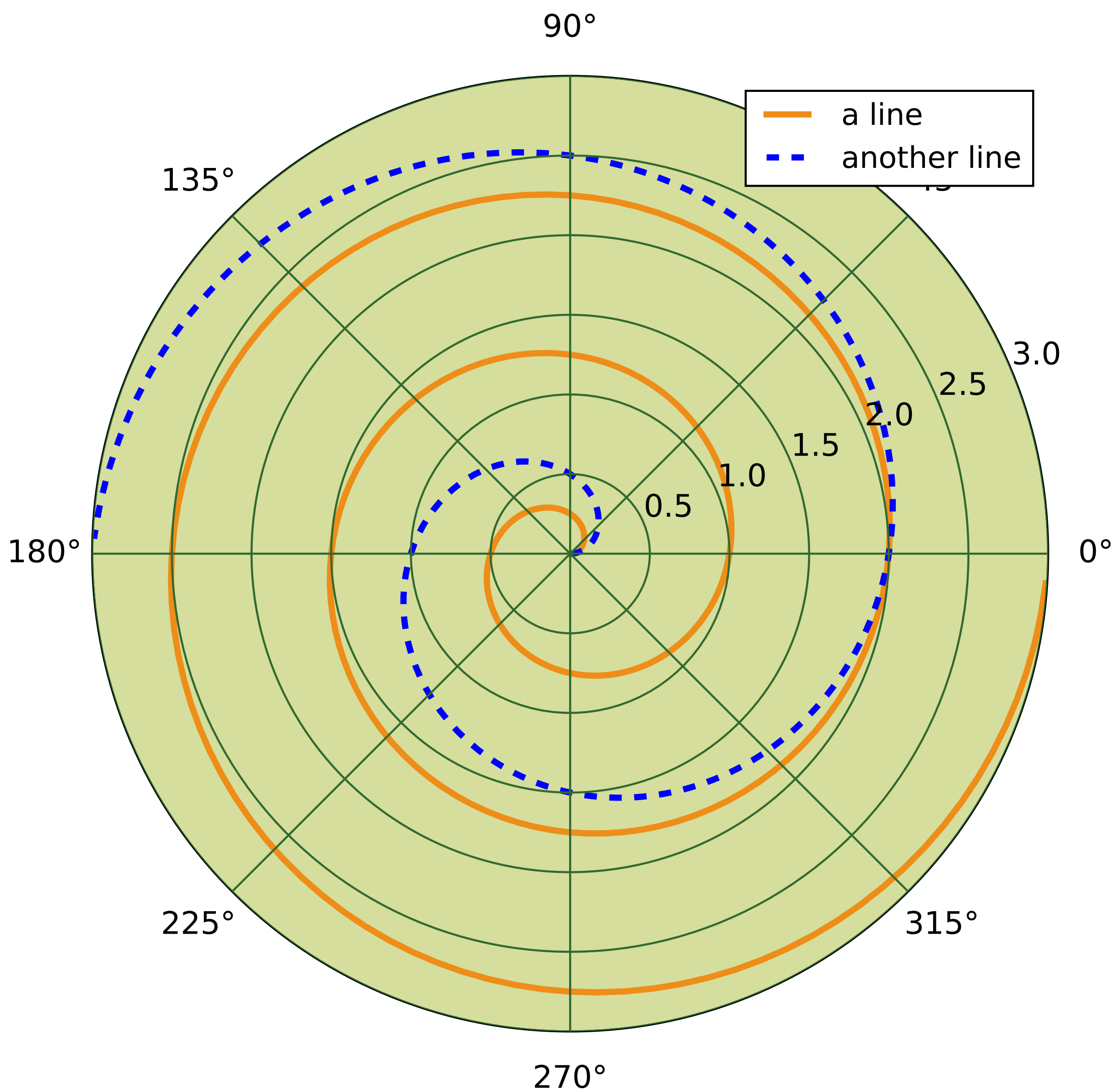



```
"""
Demo of a line plot on a polar axis.
"""
import numpy as np
```

```
import matplotlib.pyplot as plt


r = np.arange(0, 3.0, 0.01)
theta = 2 * np.pi * r

ax = plt.subplot(111, projection='polar')
ax.plot(theta, r, color='r', linewidth=3)
ax.set_rmax(2.0)
ax.grid(True)

ax.set_title("A line plot on a polar axis", va='bottom')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.156 pylab_examples example code: polar_legend.py




```
#!/usr/bin/env python

import numpy as np
from matplotlib.pyplot import figure, show, rc

# radar green, solid grid lines
rc('grid', color='#316931', linewidth=1, linestyle='-')
rc('xtick', labelsize=15)
rc('ytick', labelsize=15)

# force square figure and square axes looks better for polar, IMO
```

```
fig = figure(figsize=(8, 8))
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], projection='polar', axisbg='#d5de9c')

r = np.arange(0, 3.0, 0.01)
theta = 2*np.pi*r
ax.plot(theta, r, color='#ee8d18', lw=3, label='a line')
ax.plot(0.5*theta, r, color='blue', ls='--', lw=3, label='another line')
ax.legend()

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

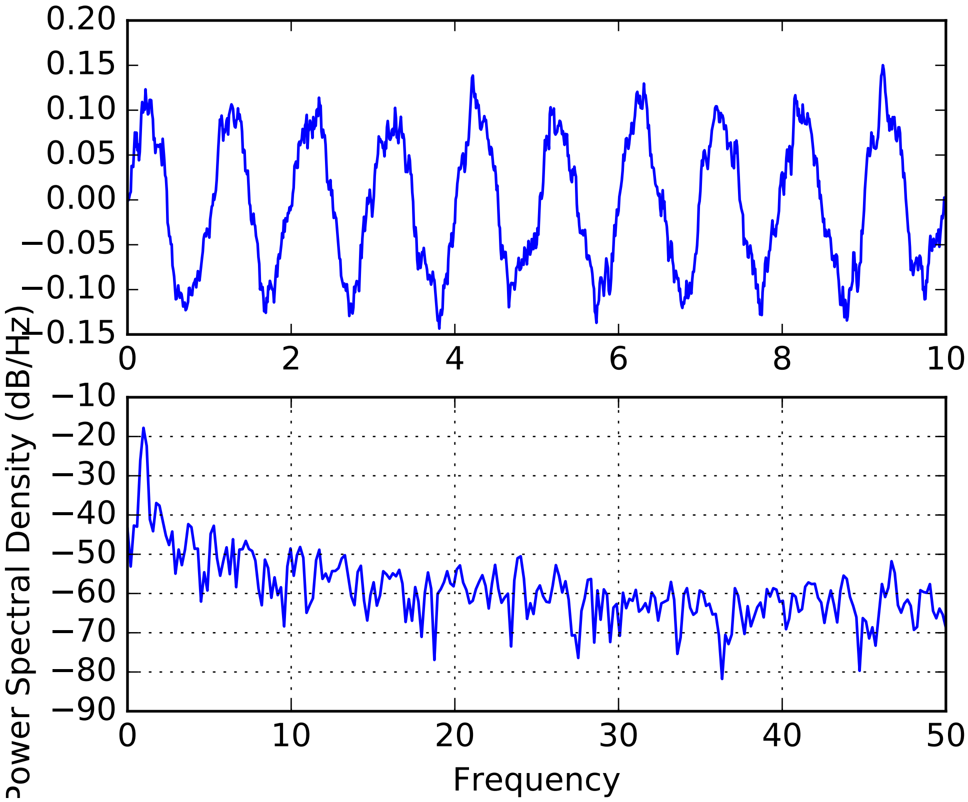
90.157 pylab_examples example code: print_stdout.py

[source code]

```
# -*- noplot -*-
# print png to standard out
# usage: python print_stdout.py > somefile.png

import sys
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

plt.plot([1, 2, 3])

plt.savefig(sys.stdout)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.158 pylab_examples example code: psd_demo.py

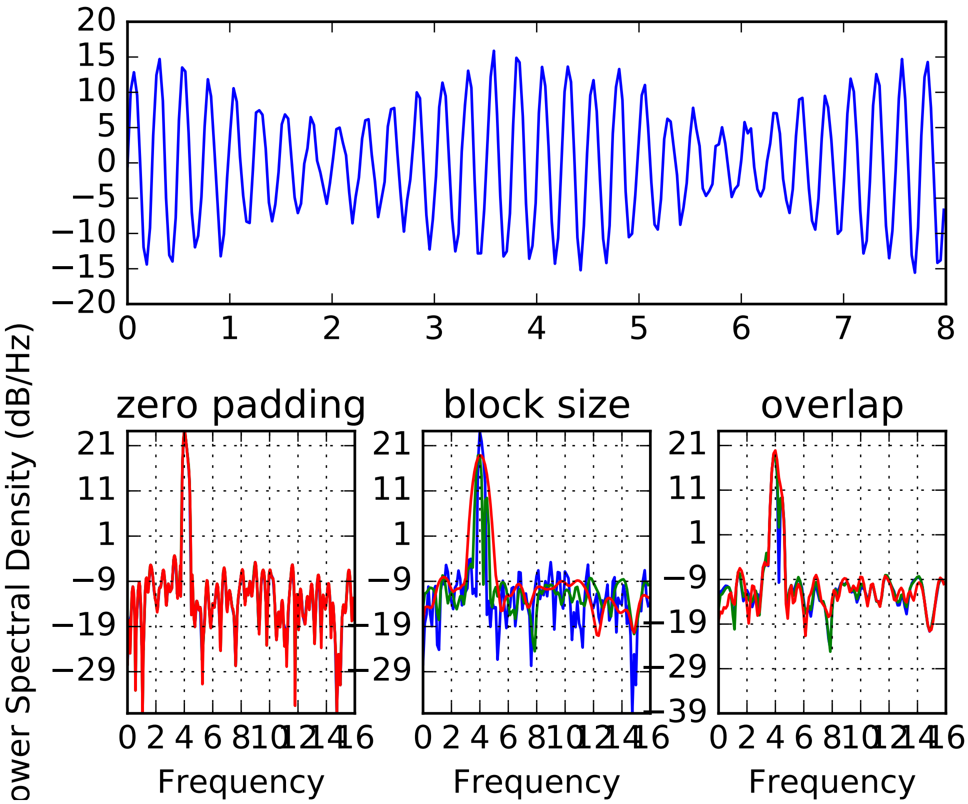



```
import matplotlib.pyplot as plt
import numpy as np

dt = 0.01
t = np.arange(0, 10, dt)
nse = np.random.randn(len(t))
r = np.exp(-t/0.05)

cnse = np.convolve(nse, r)*dt
cnse = cnse[:len(t)]
s = 0.1*np.sin(2*np.pi*t) + cnse

plt.subplot(211)
plt.plot(t, s)
plt.subplot(212)
plt.psd(s, 512, 1/dt)

plt.show()

"""
% compare with MATLAB
dt = 0.01;
```

```
t = [0:dt:10];
nse = randn(size(t));
r = exp(-t/0.05);
cnse = conv(nse, r)*dt;
cnse = cnse(1:length(t));
s = 0.1*sin(2*pi*t) + cnse;

subplot(211)
plot(t,s)
subplot(212)
psd(s, 512, 1/dt)

"""
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.159 pylab_examples example code: psd_demo2.py

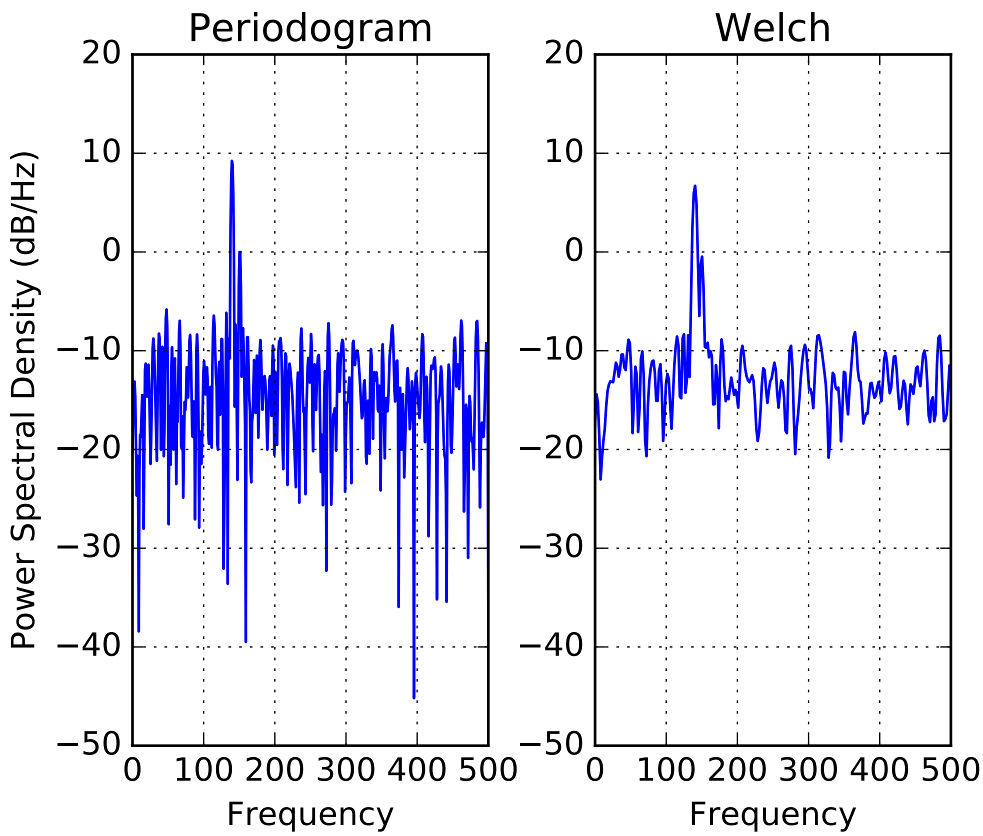



```
# This example shows the effects of some of the different PSD parameters
import numpy as np
import matplotlib.pyplot as plt
```

```
dt = np.pi / 100.
fs = 1. / dt
t = np.arange(0, 8, dt)
y = 10. * np.sin(2 * np.pi * 4 * t) + 5. * np.sin(2 * np.pi * 4.25 * t)
y = y + np.random.randn(*t.shape)

# Plot the raw time series
fig = plt.figure()
fig.subplots_adjust(hspace=0.45, wspace=0.3)
ax = fig.add_subplot(2, 1, 1)
ax.plot(t, y)

# Plot the PSD with different amounts of zero padding. This uses the entire
# time series at once
ax2 = fig.add_subplot(2, 3, 4)
ax2.psd(y, NFFT=len(t), pad_to=len(t), Fs=fs)
ax2.psd(y, NFFT=len(t), pad_to=len(t)*2, Fs=fs)
ax2.psd(y, NFFT=len(t), pad_to=len(t)*4, Fs=fs)
plt.title('zero padding')

# Plot the PSD with different block sizes, Zero pad to the length of the
# original data sequence.
ax3 = fig.add_subplot(2, 3, 5, sharex=ax2, sharey=ax2)
ax3.psd(y, NFFT=len(t), pad_to=len(t), Fs=fs)
ax3.psd(y, NFFT=len(t)//2, pad_to=len(t), Fs=fs)
ax3.psd(y, NFFT=len(t)//4, pad_to=len(t), Fs=fs)
ax3.set_ylabel('')
plt.title('block size')

# Plot the PSD with different amounts of overlap between blocks
ax4 = fig.add_subplot(2, 3, 6, sharex=ax2, sharey=ax2)
ax4.psd(y, NFFT=len(t)//2, pad_to=len(t), noverlap=0, Fs=fs)
ax4.psd(y, NFFT=len(t)//2, pad_to=len(t), noverlap=int(0.05*len(t)/2.), Fs=fs)
ax4.psd(y, NFFT=len(t)//2, pad_to=len(t), noverlap=int(0.2*len(t)/2.), Fs=fs)
ax4.set_ylabel('')
plt.title('overlap')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.160 pylab_examples example code: psd_demo3.py

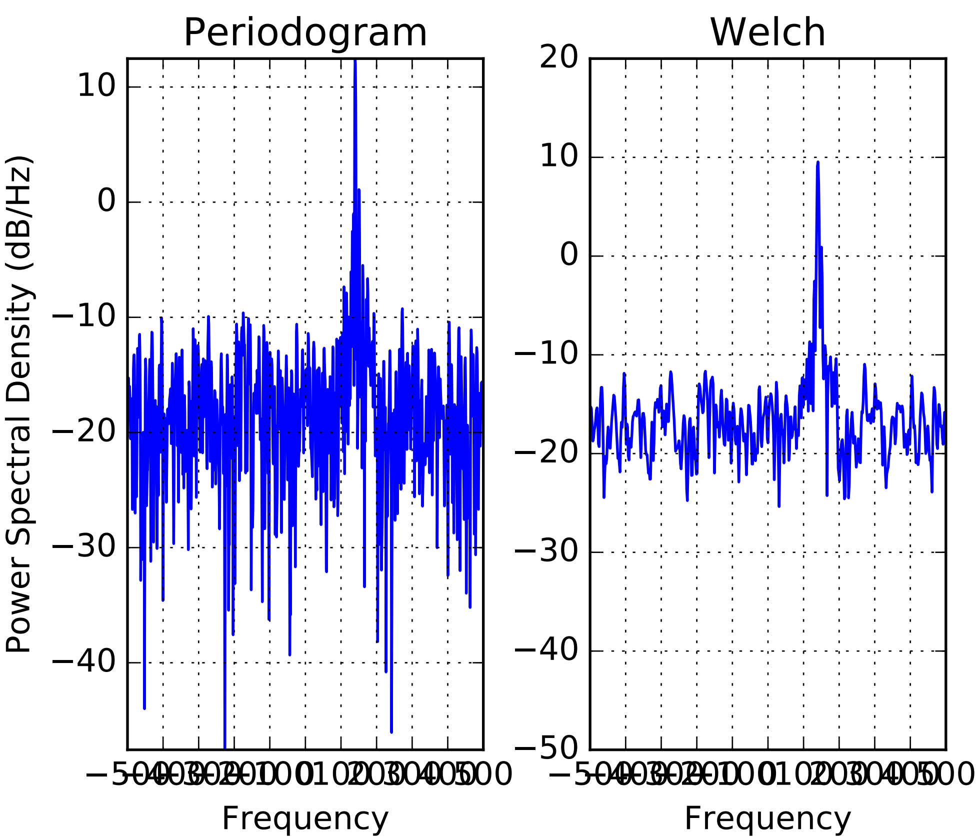



```
# This is a ported version of a MATLAB example from the signal processing
# toolbox that showed some difference at one time between Matplotlib's and
# MATLAB's scaling of the PSD.

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

fs = 1000
t = np.linspace(0, 0.3, 301)
A = np.array([2, 8]).reshape(-1, 1)
f = np.array([150, 140]).reshape(-1, 1)
xn = (A * np.sin(2 * np.pi * f * t)).sum(axis=0) + 5 * np.random.randn(*t.shape)

yticks = np.arange(-50, 30, 10)
xticks = np.arange(0, 550, 100)
plt.subplots_adjust(hspace=0.45, wspace=0.3)
plt.subplot(1, 2, 1)

plt.psd(xn, NFFT=301, Fs=fs, window=mlab.window_none, pad_to=1024,
        scale_by_freq=True)
plt.title('Periodogram')
```

```
plt.yticks(yticks)
plt.xticks(xticks)
plt.grid(True)

plt.subplot(1, 2, 2)
plt.psd(xn, NFFT=150, Fs=fs, window=mlab.window_none, noverlap=75, pad_to=512,
        scale_by_freq=True)
plt.title('Welch')
plt.xticks(xticks)
plt.yticks(yticks)
plt.ylabel('')
plt.grid(True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.161 pylab_examples example code: psd_demo_complex.py

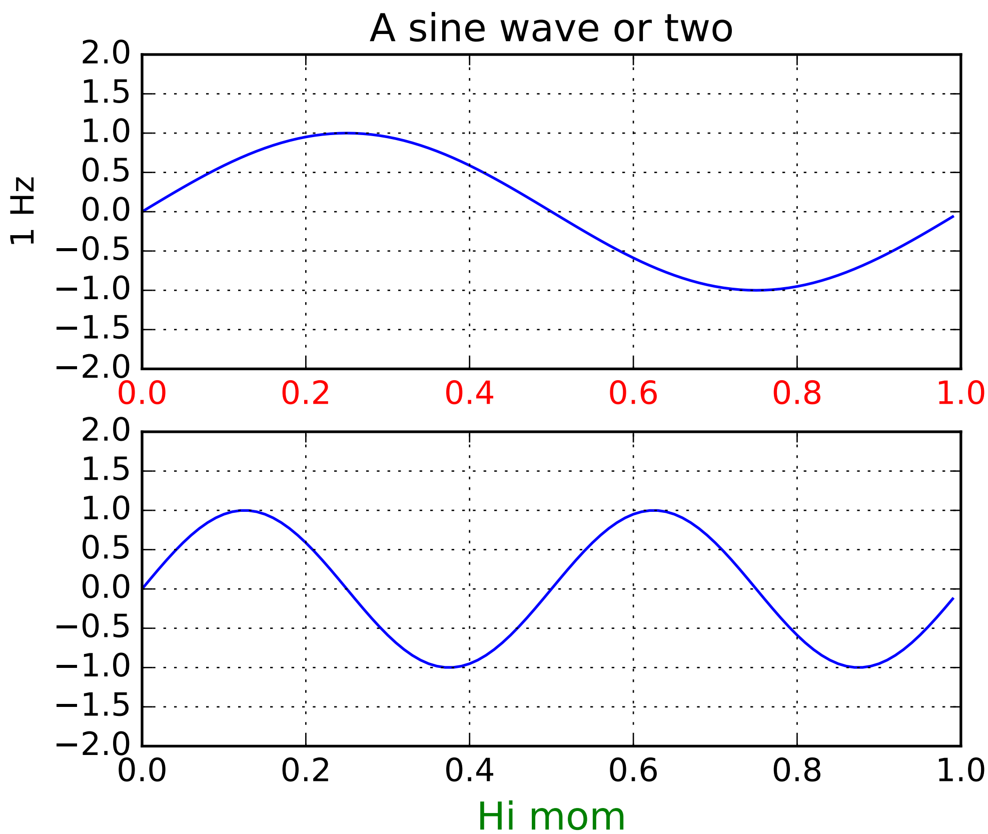



```
# This is a ported version of a MATLAB example from the signal processing
# toolbox that showed some difference at one time between Matplotlib's and
# MATLAB's scaling of the PSD.  This differs from psd_demo3.py in that
```

```python
# this uses a complex signal, so we can see that complex PSD's work properly
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

fs = 1000
t = np.linspace(0, 0.3, 301)
A = np.array([2, 8]).reshape(-1, 1)
f = np.array([150, 140]).reshape(-1, 1)
xn = (A * np.exp(2j * np.pi * f * t)).sum(axis=0) + 5 * np.random.randn(*t.shape)

yticks = np.arange(-50, 30, 10)
xticks = np.arange(-500, 550, 100)
plt.subplots_adjust(hspace=0.45, wspace=0.3)
ax = plt.subplot(1, 2, 1)

plt.psd(xn, NFFT=301, Fs=fs, window=mlab.window_none, pad_to=1024,
        scale_by_freq=True)
plt.title('Periodogram')
plt.yticks(yticks)
plt.xticks(xticks)
plt.grid(True)
plt.xlim(-500, 500)

plt.subplot(1, 2, 2, sharex=ax, sharey=ax)
plt.psd(xn, NFFT=150, Fs=fs, window=mlab.window_none, noverlap=75, pad_to=512,
        scale_by_freq=True)
plt.title('Welch')
plt.xticks(xticks)
plt.yticks(yticks)
plt.ylabel('')
plt.grid(True)
plt.xlim(-500, 500)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.162 pylab_examples example code: pythonic_matplotlib.py



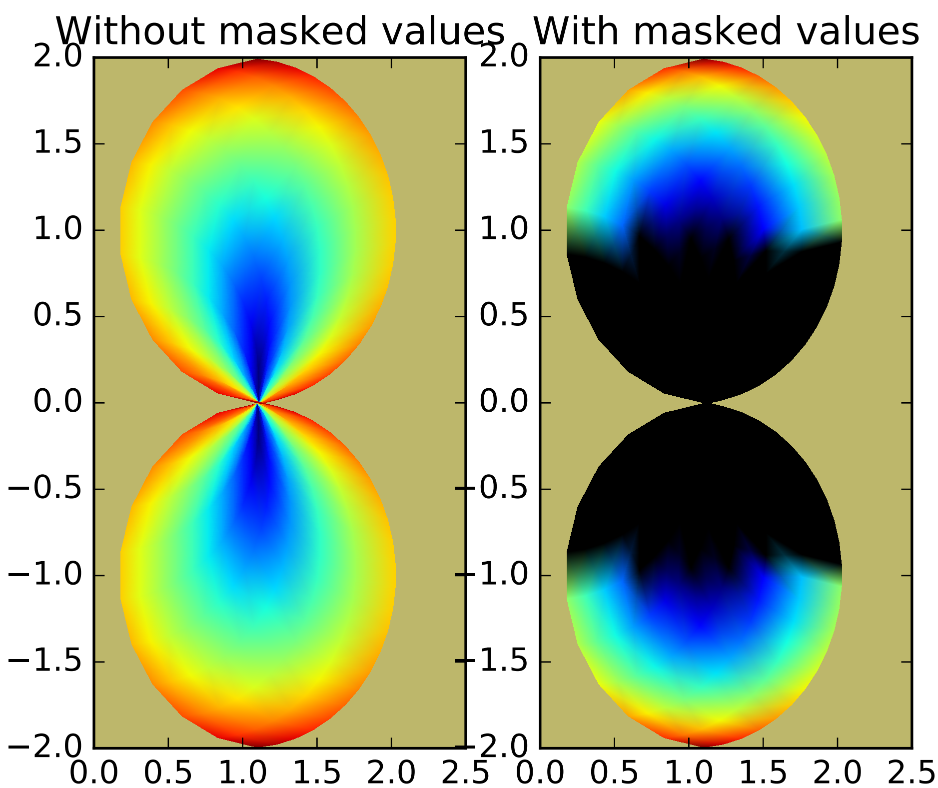

```
"""
Some people prefer to write more pythonic, object-oriented code
rather than use the pyplot interface to matplotlib.  This example shows
you how.

Unless you are an application developer, I recommend using part of the
pyplot interface, particularly the figure, close, subplot, axes, and
show commands.  These hide a lot of complexity from you that you don't
need to see in normal figure creation, like instantiating DPI
instances, managing the bounding boxes of the figure elements,
creating and reaslizing GUI windows and embedding figures in them.


If you are an application developer and want to embed matplotlib in
your application, follow the lead of examples/embedding_in_wx.py,
examples/embedding_in_gtk.py or examples/embedding_in_tk.py.  In this
case you will want to control the creation of all your figures,
embedding them in application windows, etc.

If you are a web application developer, you may want to use the
example in webapp_demo.py, which shows how to use the backend agg
figure canvase directly, with none of the globals (current figure,
```

```
current axes) that are present in the pyplot interface.  Note that
there is no reason why the pyplot interface won't work for web
application developers, however.

If you see an example in the examples dir written in pyplot interface,
and you want to emulate that using the true python method calls, there
is an easy mapping.  Many of those examples use 'set' to control
figure properties.  Here's how to map those commands onto instance
methods

The syntax of set is

  plt.setp(object or sequence, somestring, attribute)

if called with an object, set calls

  object.set_somestring(attribute)

if called with a sequence, set does

  for object in sequence:
       object.set_somestring(attribute)

So for your example, if a is your axes object, you can do

  a.set_xticklabels([])
  a.set_yticklabels([])
  a.set_xticks([])
  a.set_yticks([])
"""


from matplotlib.pyplot import figure, show
from numpy import arange, sin, pi

t = arange(0.0, 1.0, 0.01)

fig = figure(1)

ax1 = fig.add_subplot(211)
ax1.plot(t, sin(2*pi*t))
ax1.grid(True)
ax1.set_ylim((-2, 2))
ax1.set_ylabel('1 Hz')
ax1.set_title('A sine wave or two')

for label in ax1.get_xticklabels():
    label.set_color('r')


ax2 = fig.add_subplot(212)
ax2.plot(t, sin(2*2*pi*t))
ax2.grid(True)
```

```
ax2.set_ylim((-2, 2))
l = ax2.set_xlabel('Hi mom')
l.set_color('g')
l.set_fontsize('large')

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.163 pylab_examples example code: quadmesh_demo.py

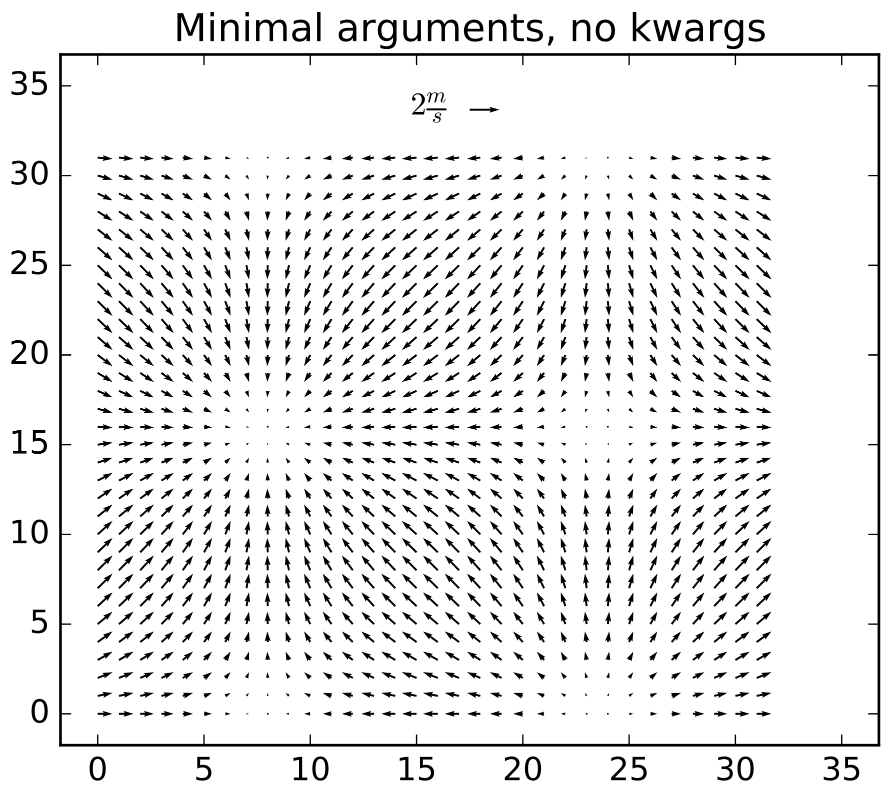


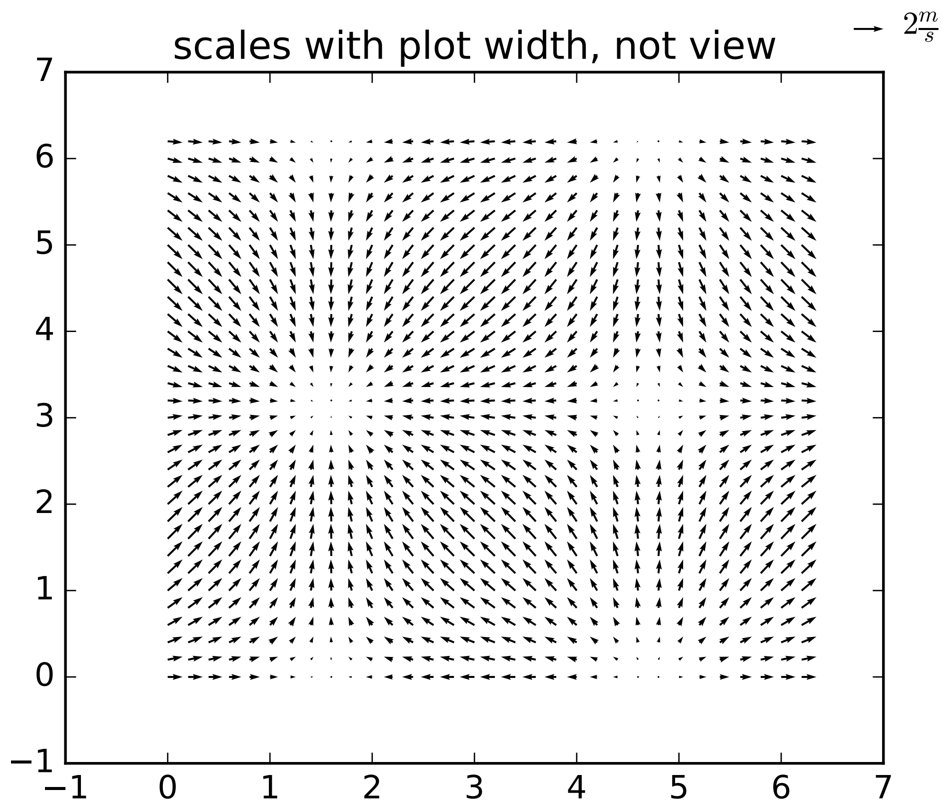

```
#!/usr/bin/env python
"""
pcolormesh uses a QuadMesh, a faster generalization of pcolor, but
with some restrictions.

This demo illustrates a bug in quadmesh with masked data.
"""

import numpy as np
from matplotlib.pyplot import figure, show, savefig
from matplotlib import cm, colors
```

```python
from numpy import ma

n = 12
x = np.linspace(-1.5, 1.5, n)
y = np.linspace(-1.5, 1.5, n*2)
X, Y = np.meshgrid(x, y)
Qx = np.cos(Y) - np.cos(X)
Qz = np.sin(Y) + np.sin(X)
Qx = (Qx + 1.1)
Z = np.sqrt(X**2 + Y**2)/5
Z = (Z - Z.min()) / (Z.max() - Z.min())

# The color array can include masked values:
Zm = ma.masked_where(np.fabs(Qz) < 0.5*np.amax(Qz), Z)


fig = figure()
ax = fig.add_subplot(121)
ax.set_axis_bgcolor("#bdb76b")
ax.pcolormesh(Qx, Qz, Z, shading='gouraud')
ax.set_title('Without masked values')

ax = fig.add_subplot(122)
ax.set_axis_bgcolor("#bdb76b")
#  You can control the color of the masked region:
#cmap = cm.jet
#cmap.set_bad('r', 1.0)
#ax.pcolormesh(Qx,Qz,Zm, cmap=cmap)
#  Or use the default, which is transparent:
col = ax.pcolormesh(Qx, Qz, Zm, shading='gouraud')
ax.set_title('With masked values')


show()
```

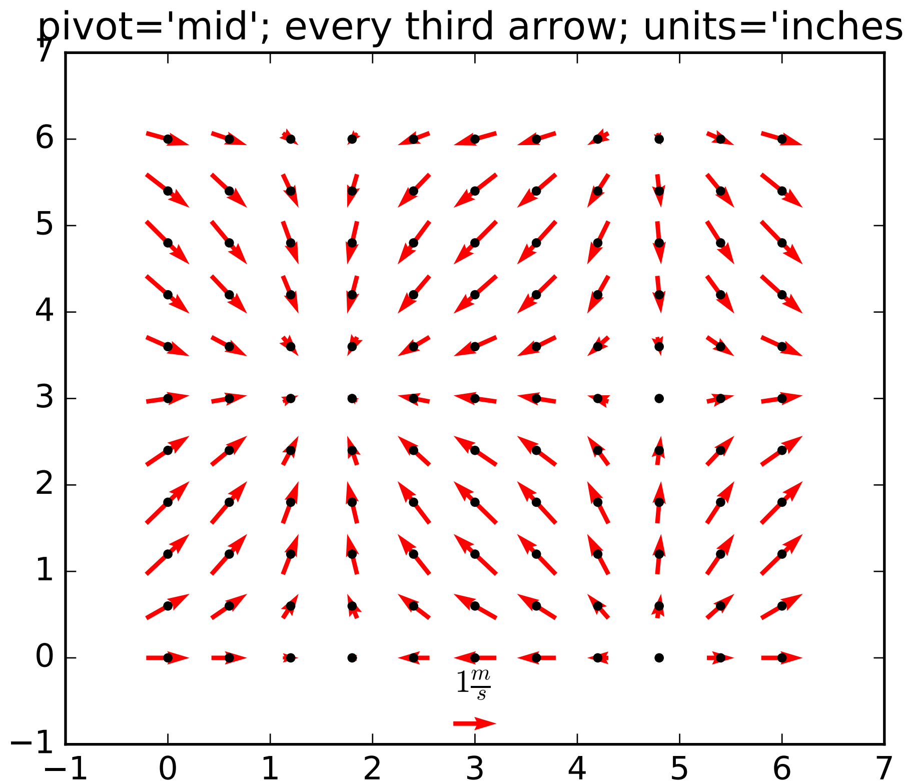
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

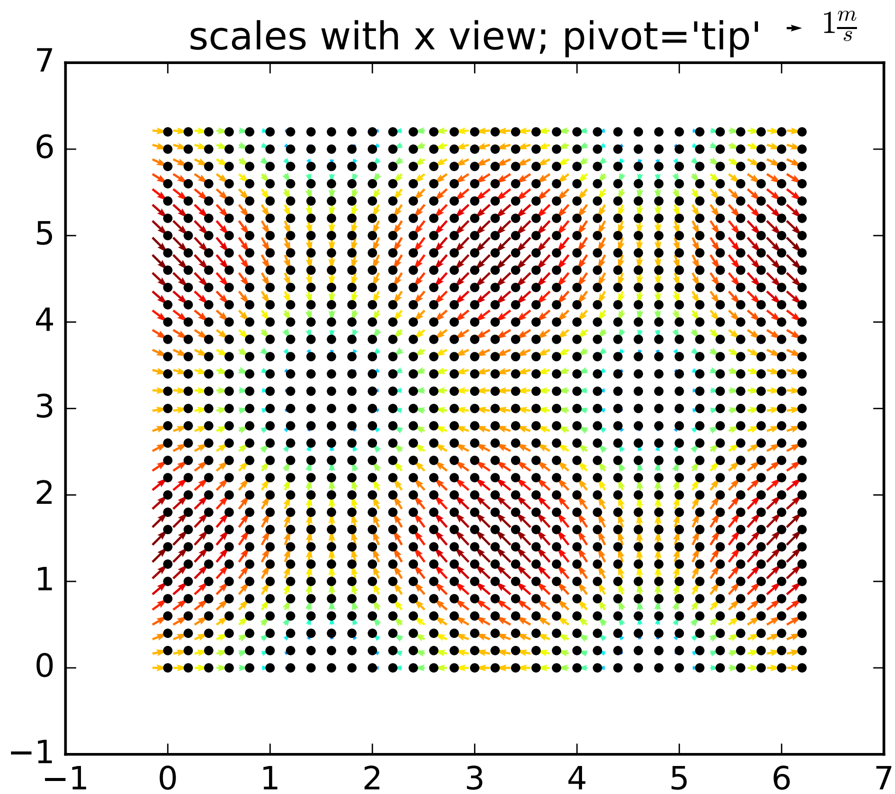
90.164 pylab_examples example code: quiver_demo.py

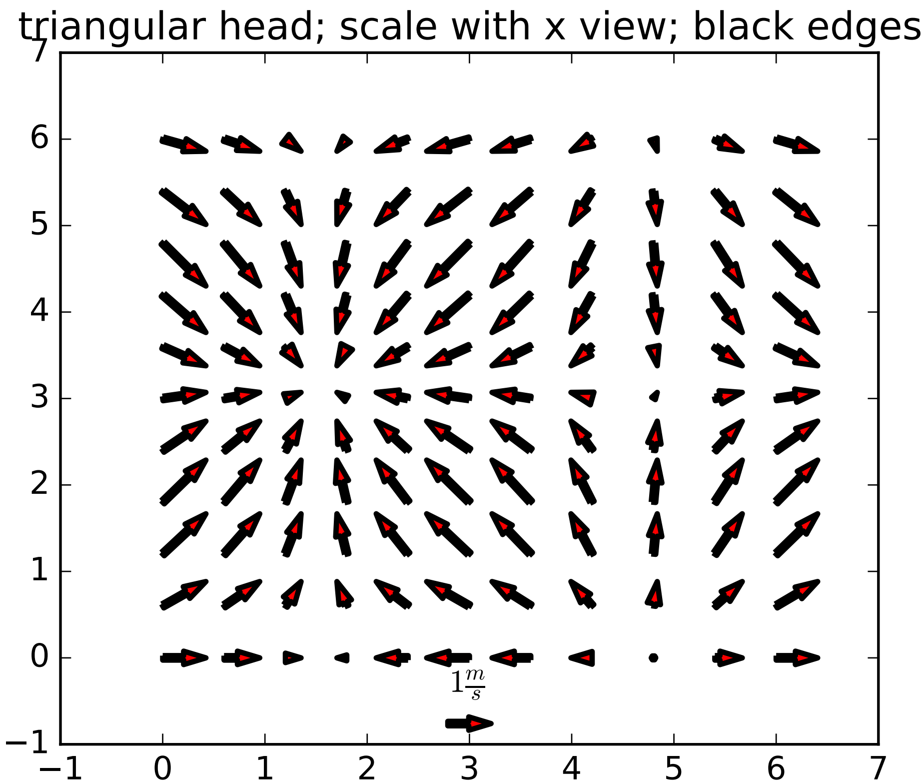


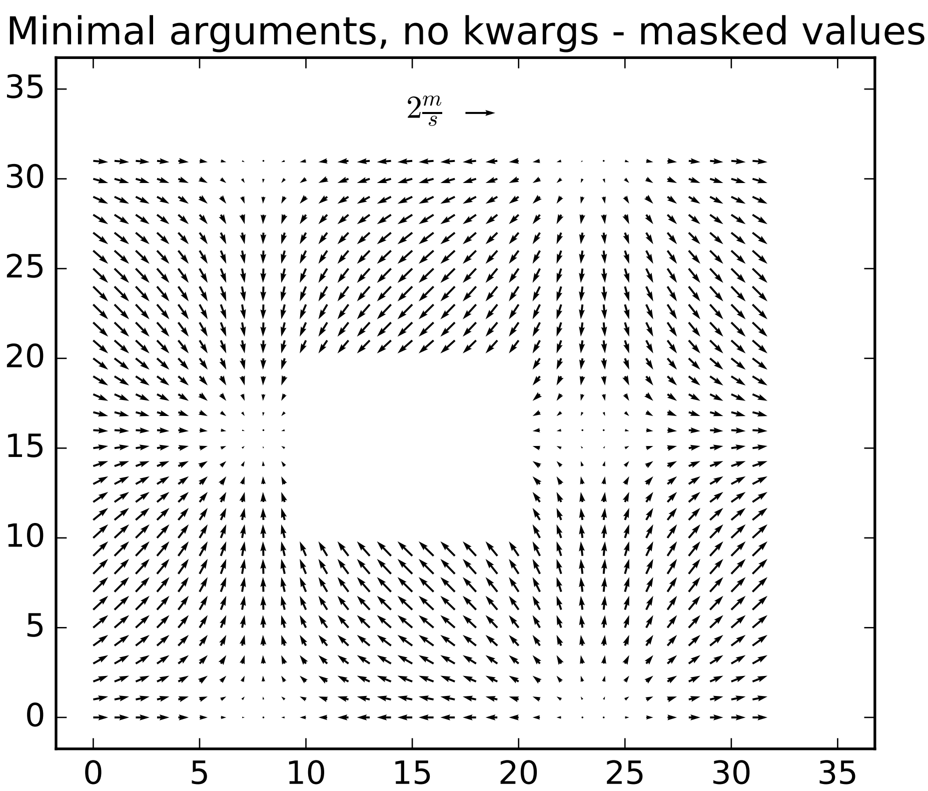


$2\frac{m}{s}$
scales with plot width, not view
7
6
5
4
3
2
1
0
−1
−1
0
1
2
3
4
5
6
7


pivot='mid'; every third arrow; units='inches'
7
6
5
4
3
2
1
0
−1
−1
0
1
2
3
4
5
6
7
$1\frac{m}{s}$


scales with x view; pivot='tip'
$1\frac{m}{s}$
7
6
5
4
3
2
1
0
−1
−1
0
1
2
3
4
5
6
7


triangular head; scale with x view; black edges
7
6
5
4
3
2
1
0
−1
−1
0
1
2
3
4
5
6
7
$1\frac{m}{s}$



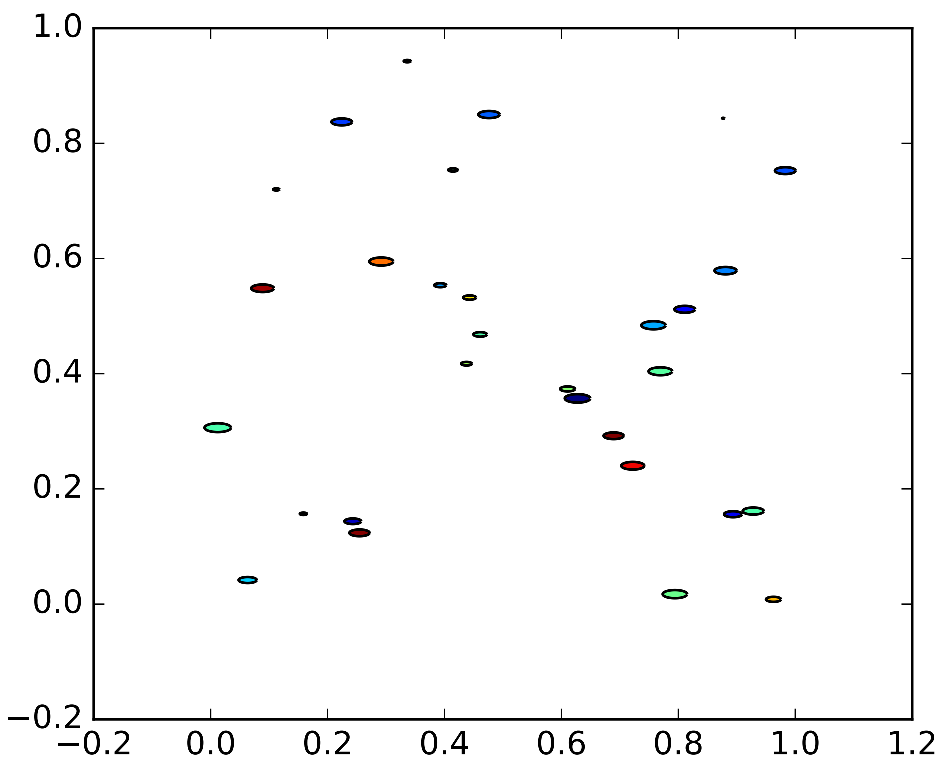

```
'''
Demonstration of quiver and quiverkey functions. This is using the
new version coming from the code in quiver.py.

Known problem: the plot autoscaling does not take into account
the arrows, so those on the boundaries are often out of the picture.
This is *not* an easy problem to solve in a perfectly general way.
The workaround is to manually expand the axes.

'''
import matplotlib.pyplot as plt
import numpy as np
from numpy import ma

X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))
U = np.cos(X)
V = np.sin(Y)

# 1
plt.figure()
Q = plt.quiver(U, V)
qk = plt.quiverkey(Q, 0.5, 0.92, 2, r'$2 \frac{m}{s}$', labelpos='W',
                   fontproperties={'weight': 'bold'})
l, r, b, t = plt.axis()
dx, dy = r - l, t - b
plt.axis([l - 0.05*dx, r + 0.05*dx, b - 0.05*dy, t + 0.05*dy])
```

```
plt.title('Minimal arguments, no kwargs')

# 2
plt.figure()
Q = plt.quiver(X, Y, U, V, units='width')
qk = plt.quiverkey(Q, 0.9, 0.95, 2, r'$2 \frac{m}{s}$',
                   labelpos='E',
                   coordinates='figure',
                   fontproperties={'weight': 'bold'})
plt.axis([-1, 7, -1, 7])
plt.title('scales with plot width, not view')

# 3
plt.figure()
Q = plt.quiver(X[::3, ::3], Y[::3, ::3], U[::3, ::3], V[::3, ::3],
               pivot='mid', color='r', units='inches')
qk = plt.quiverkey(Q, 0.5, 0.03, 1, r'$1 \frac{m}{s}$',
                   fontproperties={'weight': 'bold'})
plt.plot(X[::3, ::3], Y[::3, ::3], 'k.')
plt.axis([-1, 7, -1, 7])
plt.title("pivot='mid'; every third arrow; units='inches'")

# 4
plt.figure()
M = np.hypot(U, V)
Q = plt.quiver(X, Y, U, V, M,
               units='x',
               pivot='tip',
               width=0.022,
               scale=1 / 0.15)
qk = plt.quiverkey(Q, 0.9, 1.05, 1, r'$1 \frac{m}{s}$',
                   labelpos='E',
                   fontproperties={'weight': 'bold'})
plt.plot(X, Y, 'k.')
plt.axis([-1, 7, -1, 7])
plt.title("scales with x view; pivot='tip'")

# 5
plt.figure()
Q = plt.quiver(X[::3, ::3], Y[::3, ::3], U[::3, ::3], V[::3, ::3],
               color='r', units='x',
               linewidths=(2,), edgecolors=('k'), headaxislength=5)
qk = plt.quiverkey(Q, 0.5, 0.03, 1, r'$1 \frac{m}{s}$',
                   fontproperties={'weight': 'bold'})
plt.axis([-1, 7, -1, 7])
plt.title("triangular head; scale with x view; black edges")

# 6
plt.figure()
M = np.zeros(U.shape, dtype='bool')
M[U.shape[0]//3:2*U.shape[0]//3,
  U.shape[1]//3:2*U.shape[1]//3] = True
```

```
U = ma.masked_array(U, mask=M)
V = ma.masked_array(V, mask=M)
Q = plt.quiver(U, V)
qk = plt.quiverkey(Q, 0.5, 0.92, 2, r'$2 \frac{m}{s}$', labelpos='W',
                   fontproperties={'weight': 'bold'})
l, r, b, t = plt.axis()
dx, dy = r - l, t - b
plt.axis([l - 0.05 * dx, r + 0.05 * dx, b - 0.05 * dy, t + 0.05 * dy])
plt.title('Minimal arguments, no kwargs - masked values')


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.165 pylab_examples example code: scatter_custom_symbol.py

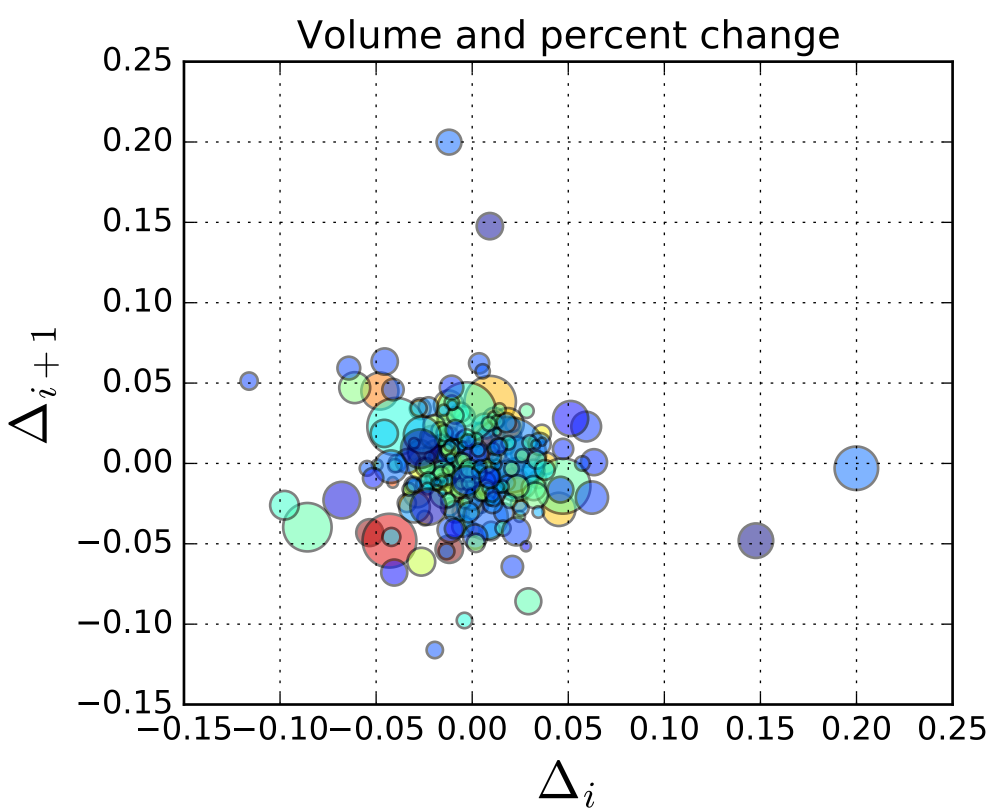



```
import matplotlib.pyplot as plt
from numpy import arange, pi, cos, sin
from numpy.random import rand

# unit area ellipse
```

```
rx, ry = 3., 1.
area = rx * ry * pi
theta = arange(0, 2*pi + 0.01, 0.1)
verts = list(zip(rx/area*cos(theta), ry/area*sin(theta)))

x, y, s, c = rand(4, 30)
s *= 10**2.

fig, ax = plt.subplots()
ax.scatter(x, y, s, c, marker=None, verts=verts)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.166 pylab_examples example code: scatter_demo2.py

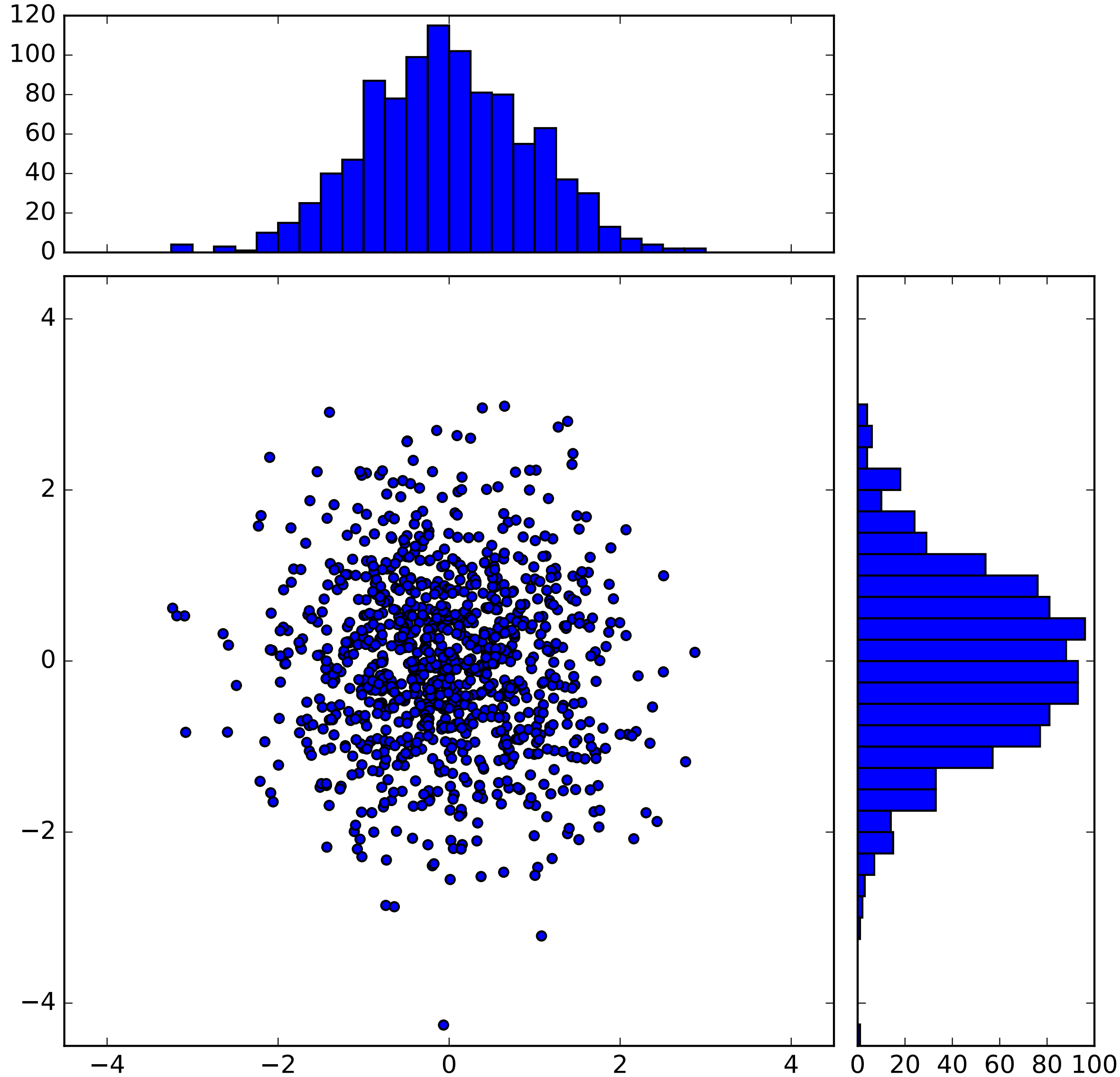



```
"""
Demo of scatter plot with varying marker colors and sizes.
"""
import numpy as np
import matplotlib.pyplot as plt
```

```python
import matplotlib.cbook as cbook

# Load a numpy record array from yahoo csv data with fields date,
# open, close, volume, adj_close from the mpl-data/example directory.
# The record array stores python datetime.date as an object array in
# the date column
datafile = cbook.get_sample_data('goog.npy')
try:
    # Python3 cannot load python2 .npy files with datetime(object) arrays
    # unless the encoding is set to bytes. Hovever this option was
    # not added until numpy 1.10 so this example will only work with
    # python 2 or with numpy 1.10 and later
    price_data = np.load(datafile, encoding='bytes').view(np.recarray)
except TypeError:
    price_data = np.load(datafile).view(np.recarray)
price_data = price_data[-250:]  # get the most recent 250 trading days

delta1 = np.diff(price_data.adj_close)/price_data.adj_close[:-1]

# Marker size in units of points^2
volume = (15 * price_data.volume[:-2] / price_data.volume[0])**2
close = 0.003 * price_data.close[:-2] / 0.003 * price_data.open[:-2]

fig, ax = plt.subplots()
ax.scatter(delta1[:-1], delta1[1:], c=close, s=volume, alpha=0.5)

ax.set_xlabel(r'$\Delta_i$', fontsize=20)
ax.set_ylabel(r'$\Delta_{i+1}$', fontsize=20)
ax.set_title('Volume and percent change')

ax.grid(True)
fig.tight_layout()

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.167 pylab_examples example code: scatter_hist.py

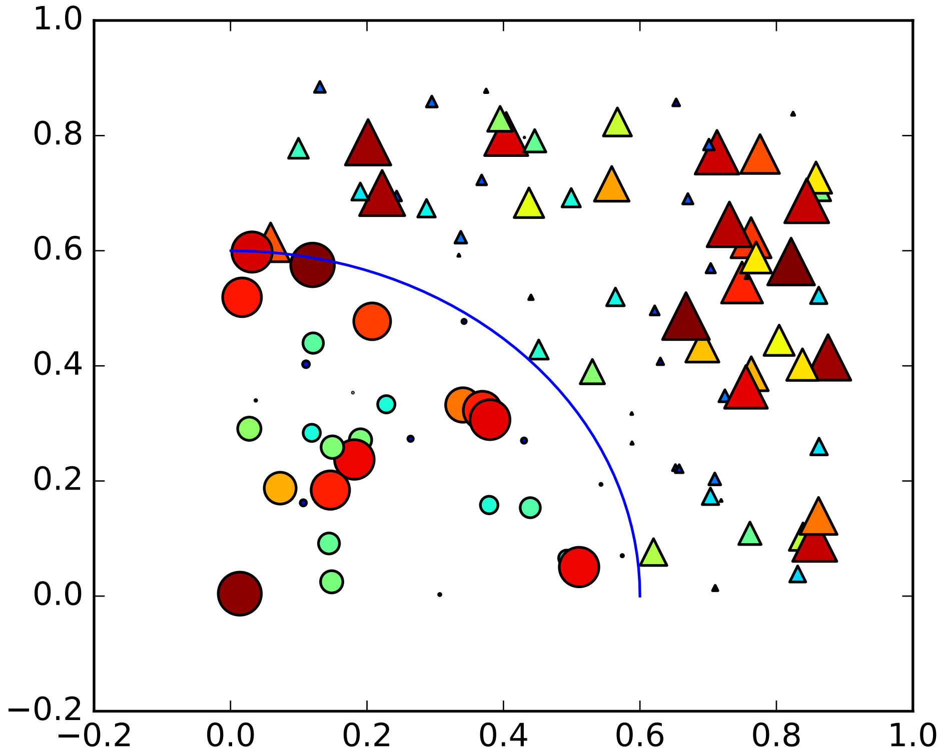



```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import NullFormatter

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)

nullfmt = NullFormatter()         # no labels

# definitions for the axes
```

```
left, width = 0.1, 0.65
bottom, height = 0.1, 0.65
bottom_h = left_h = left + width + 0.02

rect_scatter = [left, bottom, width, height]
rect_histx = [left, bottom_h, width, 0.2]
rect_histy = [left_h, bottom, 0.2, height]

# start with a rectangular Figure
plt.figure(1, figsize=(8, 8))

axScatter = plt.axes(rect_scatter)
axHistx = plt.axes(rect_histx)
axHisty = plt.axes(rect_histy)

# no labels
axHistx.xaxis.set_major_formatter(nullfmt)
axHisty.yaxis.set_major_formatter(nullfmt)

# the scatter plot:
axScatter.scatter(x, y)

# now determine nice limits by hand:
binwidth = 0.25
xymax = np.max([np.max(np.fabs(x)), np.max(np.fabs(y))])
lim = (int(xymax/binwidth) + 1) * binwidth

axScatter.set_xlim((-lim, lim))
axScatter.set_ylim((-lim, lim))

bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

axHistx.set_xlim(axScatter.get_xlim())
axHisty.set_ylim(axScatter.get_ylim())

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

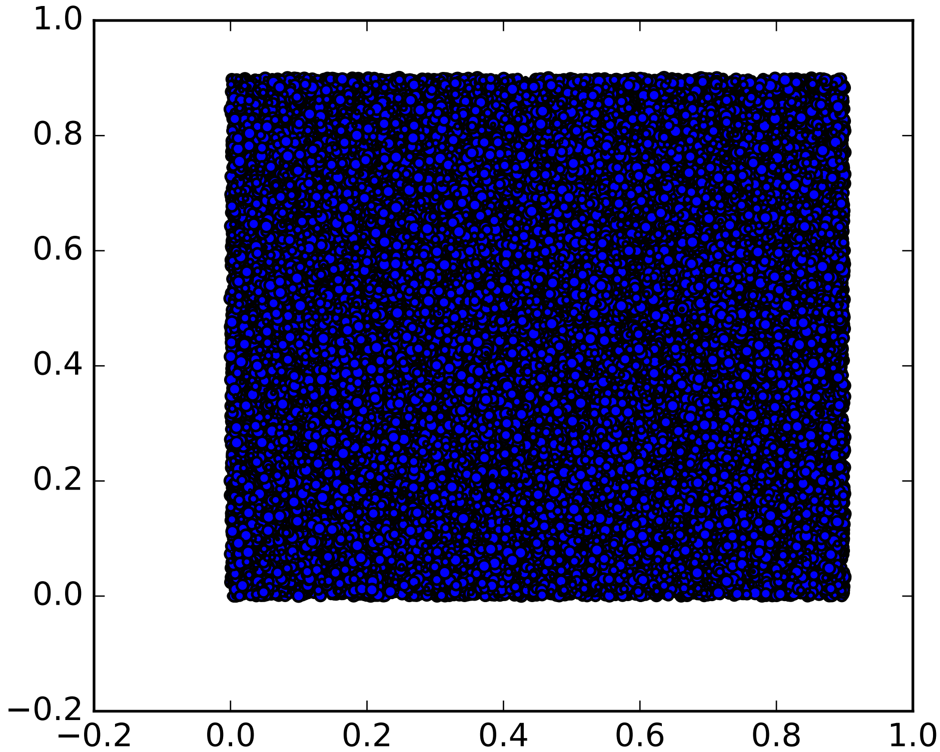
90.168 pylab_examples example code: scatter_masked.py




```
import matplotlib.pyplot as plt
import numpy as np

N = 100
r0 = 0.6
x = 0.9*np.random.rand(N)
y = 0.9*np.random.rand(N)
area = np.pi*(10 * np.random.rand(N))**2  # 0 to 10 point radiuses
c = np.sqrt(area)
r = np.sqrt(x*x + y*y)
area1 = np.ma.masked_where(r < r0, area)
area2 = np.ma.masked_where(r >= r0, area)
plt.scatter(x, y, s=area1, marker='^', c=c, hold='on')
plt.scatter(x, y, s=area2, marker='o', c=c)
# Show the boundary between the regions:
theta = np.arange(0, np.pi/2, 0.01)
plt.plot(r0*np.cos(theta), r0*np.sin(theta))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

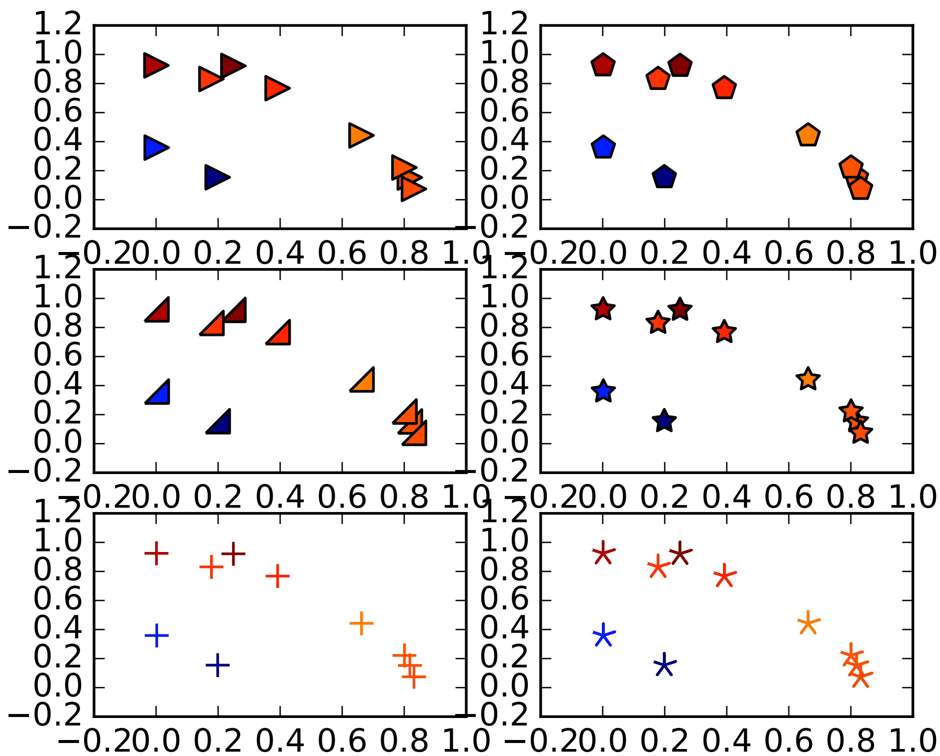
90.169 pylab_examples example code: scatter_profile.py




```
"""
N       Classic     Base renderer    Ext renderer
20       0.22           0.14            0.14
100      0.16           0.14            0.13
1000     0.45           0.26            0.17
10000    3.30           1.31            0.53
50000    19.30          6.53            1.98
"""
from __future__ import print_function  # only needed for python 2.x
import matplotlib.pyplot as plt
import numpy as np

import time

for N in (20, 100, 1000, 10000, 50000):
    tstart = time.time()
    x = 0.9*np.random.rand(N)
    y = 0.9*np.random.rand(N)
    s = 20*np.random.rand(N)
    plt.scatter(x, y, s)
    print('%d symbols in %1.2f s' % (N, time.time() - tstart))
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.170 pylab_examples example code: scatter_star_poly.py

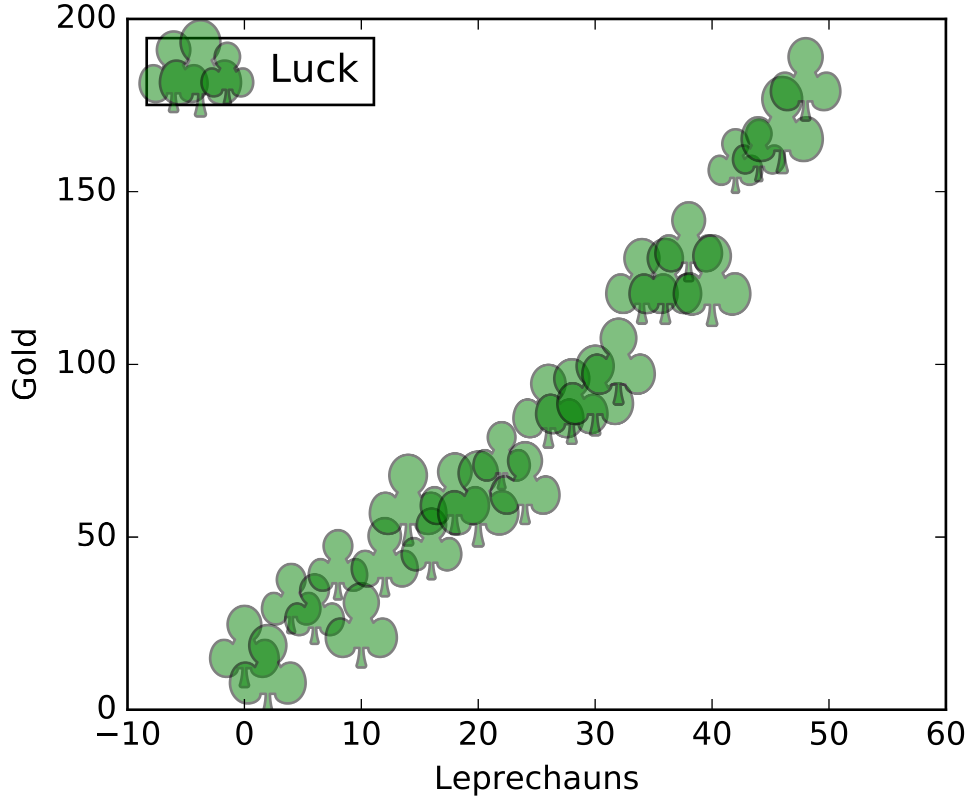


```
import numpy as np
import matplotlib.pyplot as plt


x = np.random.rand(10)
y = np.random.rand(10)
z = np.sqrt(x**2 + y**2)

plt.subplot(321)
plt.scatter(x, y, s=80, c=z, marker=">")

plt.subplot(322)
plt.scatter(x, y, s=80, c=z, marker=(5, 0))

verts = list(zip([-1., 1., 1., -1.], [-1., -1., 1., -1.]))
plt.subplot(323)
plt.scatter(x, y, s=80, c=z, marker=(verts, 0))
# equivalent:
#plt.scatter(x,y,s=80, c=z, marker=None, verts=verts)
```

```
plt.subplot(324)
plt.scatter(x, y, s=80, c=z, marker=(5, 1))

plt.subplot(325)
plt.scatter(x, y, s=80, c=z, marker='+')

plt.subplot(326)
plt.scatter(x, y, s=80, c=z, marker=(5, 2))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.171 pylab_examples example code: scatter_symbol.py

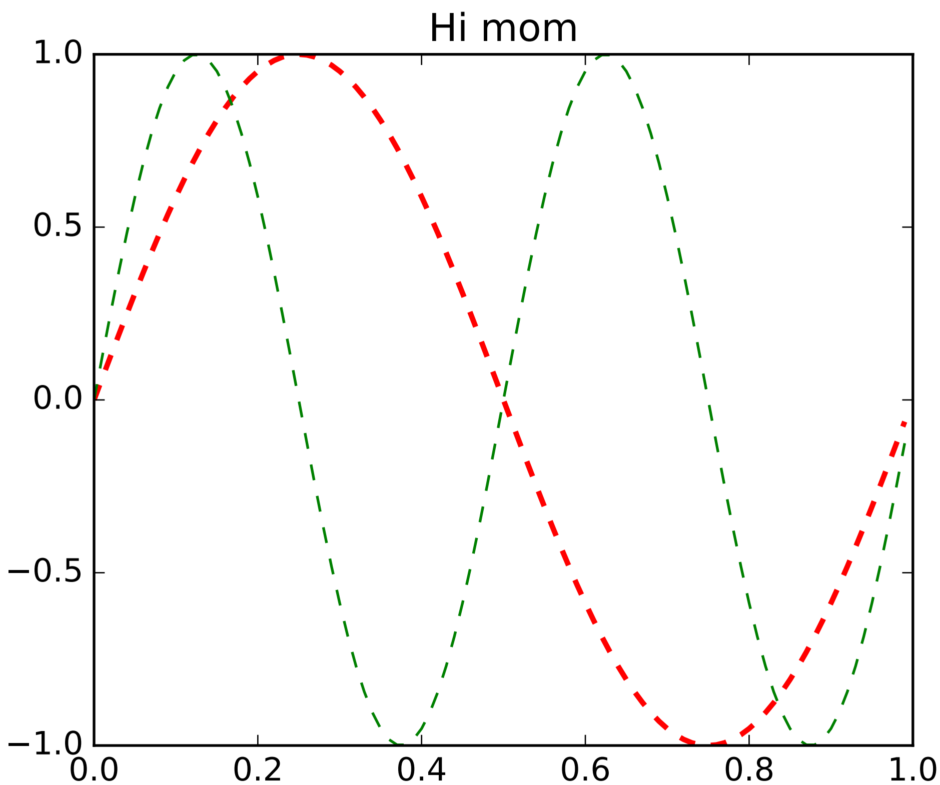



```
from matplotlib import pyplot as plt
import numpy as np
import matplotlib

x = np.arange(0.0, 50.0, 2.0)
y = x ** 1.3 + np.random.rand(*x.shape) * 30.0
```

```
s = np.random.rand(*x.shape) * 800 + 500

plt.scatter(x, y, s, c="g", alpha=0.5, marker=r'$\clubsuit$',
            label="Luck")
plt.xlabel("Leprechauns")
plt.ylabel("Gold")
plt.legend(loc=2)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.172 pylab_examples example code: set_and_get.py



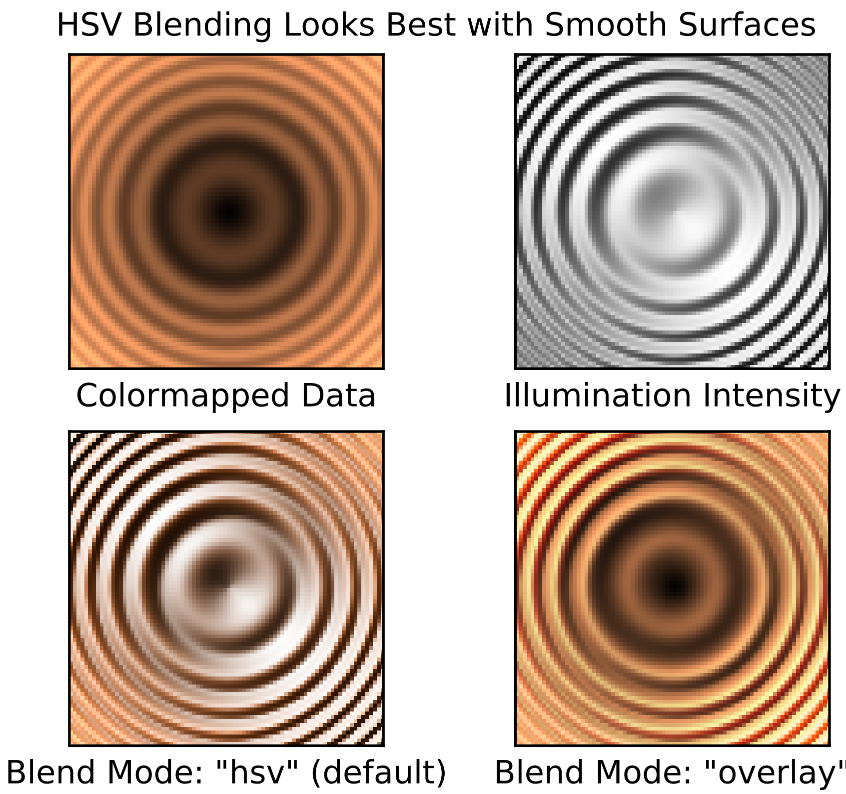

```
"""

The pyplot interface allows you to use setp and getp to set and get
object properties, as well as to do introspection on the object

set:
    To set the linestyle of a line to be dashed, you can do

      >>> line, = plt.plot([1,2,3])
```

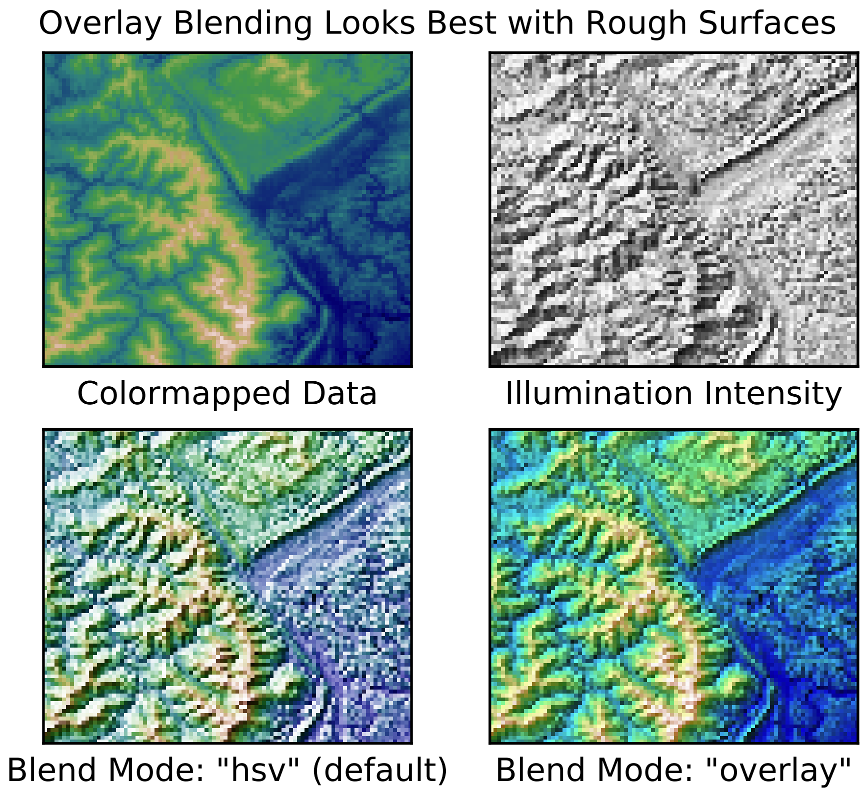
```
      >>> plt.setp(line, linestyle='--')

    If you want to know the valid types of arguments, you can provide the
    name of the property you want to set without a value

      >>> plt.setp(line, 'linestyle')
          linestyle: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]

    If you want to see all the properties that can be set, and their
    possible values, you can do

        >>> plt.setp(line)

    set operates on a single instance or a list of instances.  If you are
    in query mode introspecting the possible values, only the first
    instance in the sequence is used.  When actually setting values, all
    the instances will be set.  e.g., suppose you have a list of two lines,
    the following will make both lines thicker and red

        >>> x = np.arange(0,1.0,0.01)
        >>> y1 = np.sin(2*np.pi*x)
        >>> y2 = np.sin(4*np.pi*x)
        >>> lines = plt.plot(x, y1, x, y2)
        >>> plt.setp(lines, linewidth=2, color='r')


get:

    get returns the value of a given attribute.  You can use get to query
    the value of a single attribute

        >>> plt.getp(line, 'linewidth')
            0.5

    or all the attribute/value pairs

    >>> plt.getp(line)
        aa = True
        alpha = 1.0
        antialiased = True
        c = b
        clip_on = True
        color = b
        ... long listing skipped ...

Aliases:

  To reduce keystrokes in interactive mode, a number of properties
  have short aliases, e.g., 'lw' for 'linewidth' and 'mec' for
  'markeredgecolor'.  When calling set or get in introspection mode,
  these properties will be listed as 'fullname or aliasname', as in
```

```
"""

from __future__ import print_function

import matplotlib.pyplot as plt
import numpy as np


x = np.arange(0, 1.0, 0.01)
y1 = np.sin(2*np.pi*x)
y2 = np.sin(4*np.pi*x)
lines = plt.plot(x, y1, x, y2)
l1, l2 = lines
plt.setp(lines, linestyle='--')       # set both to dashed
plt.setp(l1, linewidth=2, color='r')  # line1 is thick and red
plt.setp(l2, linewidth=1, color='g')  # line2 is thicker and green


print('Line setters')
plt.setp(l1)
print('Line getters')
plt.getp(l1)

print('Rectangle setters')
plt.setp(plt.gca().patch)
print('Rectangle getters')
plt.getp(plt.gca().patch)

t = plt.title('Hi mom')
print('Text setters')
plt.setp(t)
print('Text getters')
plt.getp(t)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.173 pylab_examples example code: shading_example.py



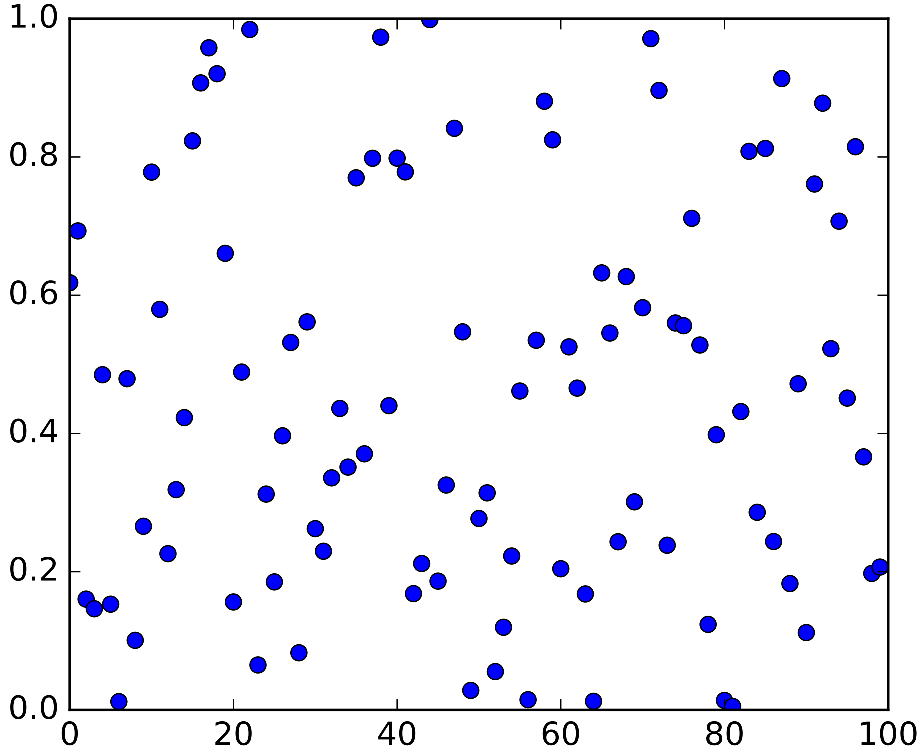


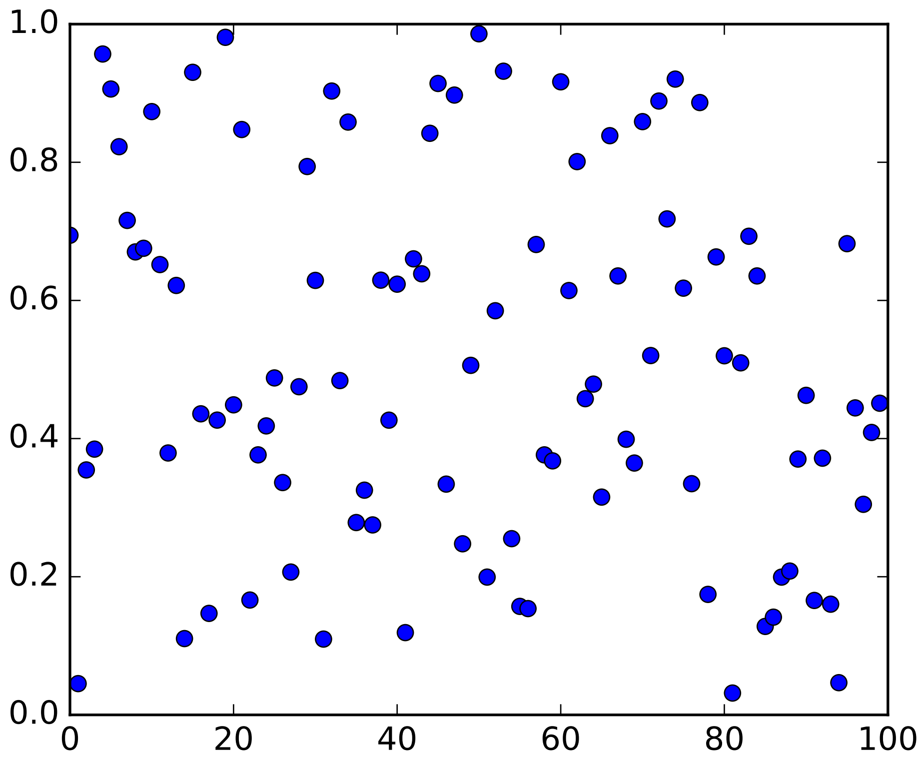

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LightSource
from matplotlib.cbook import get_sample_data

# Example showing how to make shaded relief plots
# like Mathematica
# (http://reference.wolfram.com/mathematica/ref/ReliefPlot.html)
# or Generic Mapping Tools
# (http://gmt.soest.hawaii.edu/gmt/doc/gmt/html/GMT_Docs/node145.html)


def main():
    # Test data
    x, y = np.mgrid[-5:5:0.05, -5:5:0.05]
    z = 5 * (np.sqrt(x**2 + y**2) + np.sin(x**2 + y**2))

    filename = get_sample_data('jacksboro_fault_dem.npz', asfileobj=False)
    with np.load(filename) as dem:
        elev = dem['elevation']

    fig = compare(z, plt.cm.copper)
    fig.suptitle('HSV Blending Looks Best with Smooth Surfaces', y=0.95)

    fig = compare(elev, plt.cm.gist_earth, ve=0.05)
    fig.suptitle('Overlay Blending Looks Best with Rough Surfaces', y=0.95)
```

```
    plt.show()


def compare(z, cmap, ve=1):
    # Create subplots and hide ticks
    fig, axes = plt.subplots(ncols=2, nrows=2)
    for ax in axes.flat:
        ax.set(xticks=[], yticks=[])

    # Illuminate the scene from the northwest
    ls = LightSource(azdeg=315, altdeg=45)

    axes[0, 0].imshow(z, cmap=cmap)
    axes[0, 0].set(xlabel='Colormapped Data')

    axes[0, 1].imshow(ls.hillshade(z, vert_exag=ve), cmap='gray')
    axes[0, 1].set(xlabel='Illumination Intensity')

    rgb = ls.shade(z, cmap=cmap, vert_exag=ve, blend_mode='hsv')
    axes[1, 0].imshow(rgb)
    axes[1, 0].set(xlabel='Blend Mode: "hsv" (default)')

    rgb = ls.shade(z, cmap=cmap, vert_exag=ve, blend_mode='overlay')
    axes[1, 1].imshow(rgb)
    axes[1, 1].set(xlabel='Blend Mode: "overlay"')

    return fig

if __name__ == '__main__':
    main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.174 pylab_examples example code: shared_axis_across_figures.py



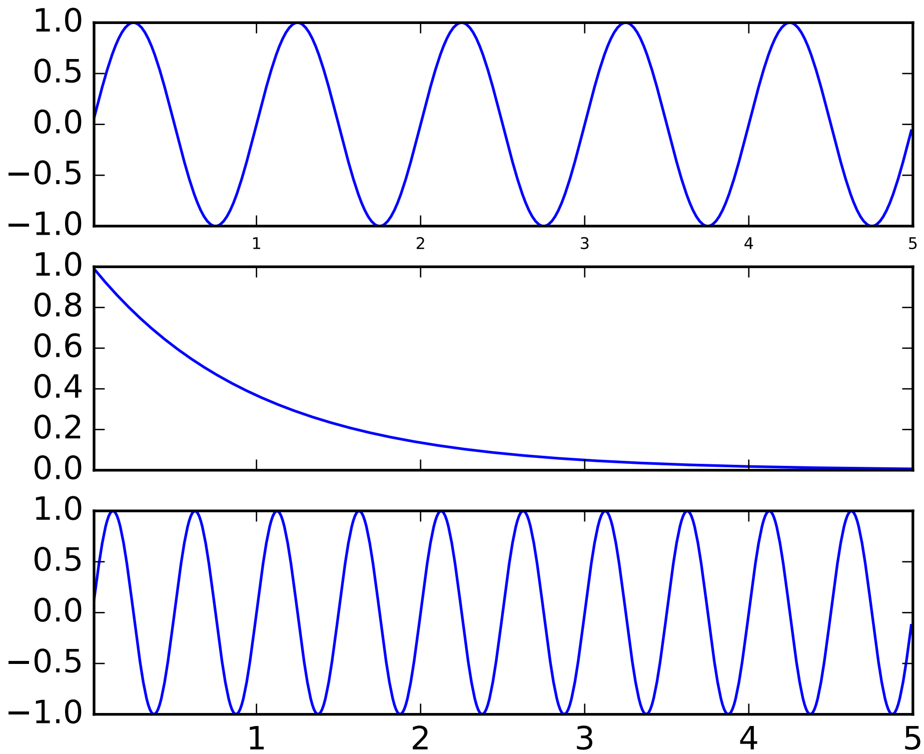



```python
"""
connect the data limits on the axes in one figure with the axes in
another.  This is not the right way to do this for two axes in the
same figure -- use the sharex and sharey property in that case
"""
import numpy as np
import matplotlib.pyplot as plt

fig1 = plt.figure()
fig2 = plt.figure()

ax1 = fig1.add_subplot(111)
ax2 = fig2.add_subplot(111, sharex=ax1, sharey=ax1)

ax1.plot(np.random.rand(100), 'o')
ax2.plot(np.random.rand(100), 'o')

# In the latest release, it is no longer necessary to do anything
# special to share axes across figures:

# ax1.sharex_foreign(ax2)
# ax2.sharex_foreign(ax1)

# ax1.sharey_foreign(ax2)
# ax2.sharey_foreign(ax1)
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.175 pylab_examples example code: shared_axis_demo.py

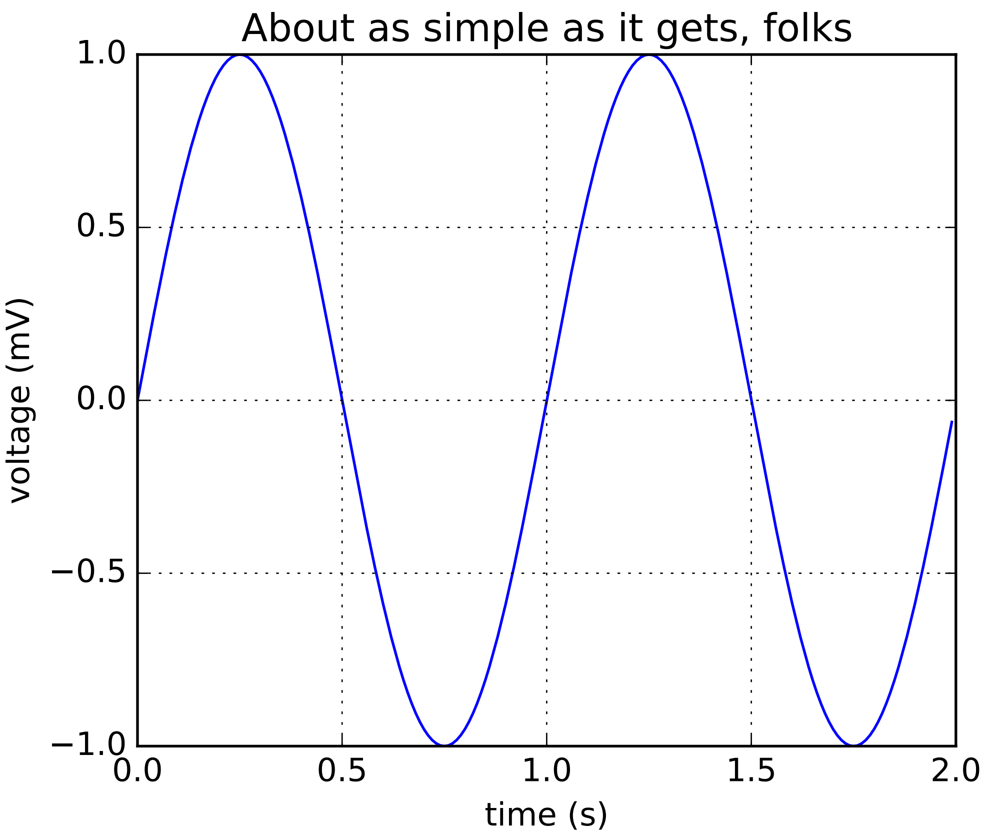



```
"""
You can share the x or y axis limits for one axis with another by
passing an axes instance as a sharex or sharey kwarg.

Changing the axis limits on one axes will be reflected automatically
in the other, and vice-versa, so when you navigate with the toolbar
the axes will follow each other on their shared axes.  Ditto for
changes in the axis scaling (e.g., log vs linear).  However, it is
possible to have differences in tick labeling, e.g., you can selectively
turn off the tick labels on one axes.

The example below shows how to customize the tick labels on the
various axes.  Shared axes share the tick locator, tick formatter,
view limits, and transformation (e.g., log, linear).  But the ticklabels
themselves do not share properties.  This is a feature and not a bug,
because you may want to make the tick labels smaller on the upper
```

```
axes, e.g., in the example below.

If you want to turn off the ticklabels for a given axes (e.g., on
subplot(211) or subplot(212), you cannot do the standard trick

   setp(ax2, xticklabels=[])

because this changes the tick Formatter, which is shared among all
axes.  But you can alter the visibility of the labels, which is a
property

  setp( ax2.get_xticklabels(), visible=False)

"""
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.01, 5.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.exp(-t)
s3 = np.sin(4*np.pi*t)

ax1 = plt.subplot(311)
plt.plot(t, s1)
plt.setp(ax1.get_xticklabels(), fontsize=6)

# share x only
ax2 = plt.subplot(312, sharex=ax1)
plt.plot(t, s2)
# make these tick labels invisible
plt.setp(ax2.get_xticklabels(), visible=False)

# share x and y
ax3 = plt.subplot(313, sharex=ax1, sharey=ax1)
plt.plot(t, s3)
plt.xlim(0.01, 5.0)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

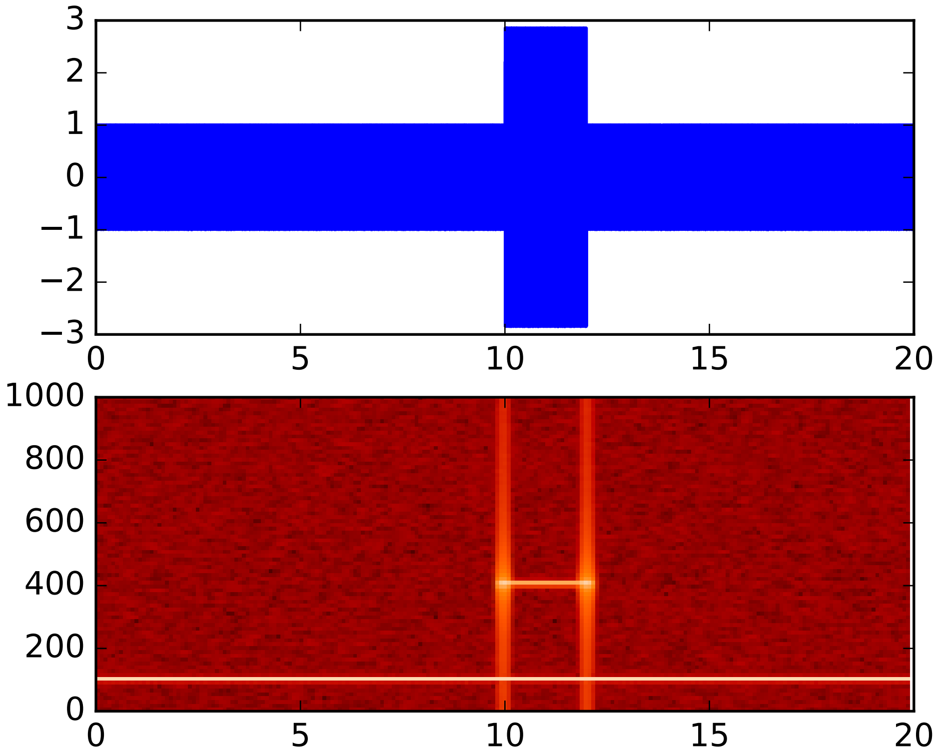
90.176 pylab_examples example code: simple_plot.py




```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
plt.plot(t, s)

plt.xlabel('time (s)')
plt.ylabel('voltage (mV)')
plt.title('About as simple as it gets, folks')
plt.grid(True)
plt.savefig("test.png")
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.177 pylab_examples example code: specgram_demo.py

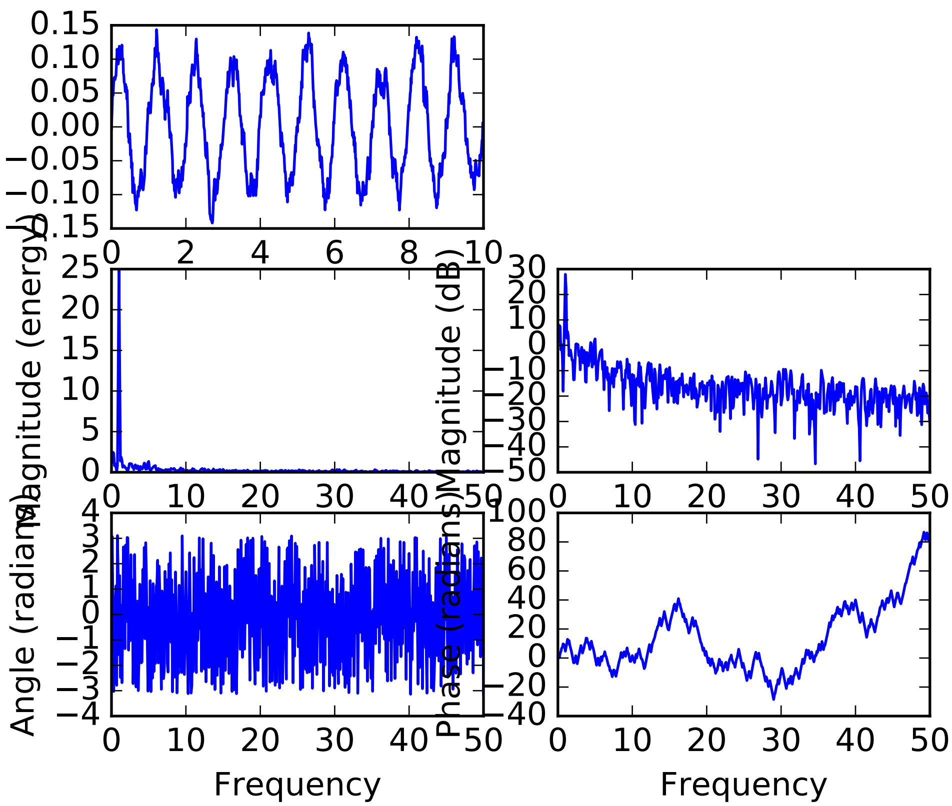



```
import matplotlib.pyplot as plt
import numpy as np

dt = 0.0005
t = np.arange(0.0, 20.0, dt)
s1 = np.sin(2*np.pi*100*t)
s2 = 2*np.sin(2*np.pi*400*t)

# create a transient "chirp"
mask = np.where(np.logical_and(t > 10, t < 12), 1.0, 0.0)
s2 = s2 * mask

# add some noise into the mix
nse = 0.01*np.random.random(size=len(t))

x = s1 + s2 + nse  # the signal
NFFT = 1024       # the length of the windowing segments
Fs = int(1.0/dt)  # the sampling frequency

# Pxx is the segments x freqs array of instantaneous power, freqs is
# the frequency vector, bins are the centers of the time bins in which
# the power is computed, and im is the matplotlib.image.AxesImage
```

```
# instance

ax1 = plt.subplot(211)
plt.plot(t, x)
plt.subplot(212, sharex=ax1)
Pxx, freqs, bins, im = plt.specgram(x, NFFT=NFFT, Fs=Fs, noverlap=900,
                                    cmap=plt.cm.gist_heat)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.178 pylab_examples example code: spectrum_demo.py

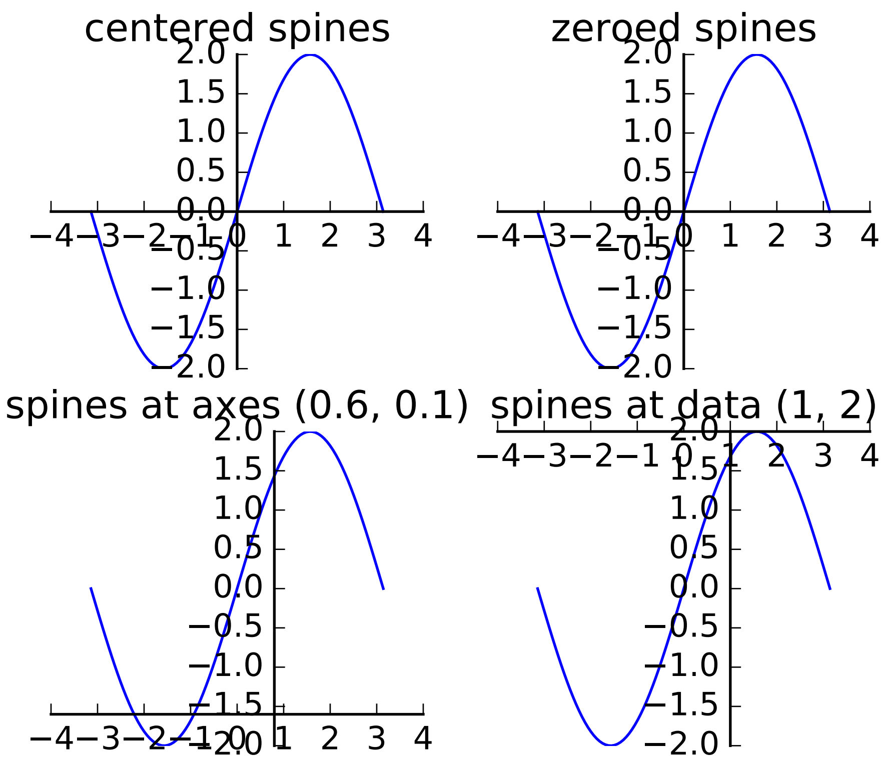


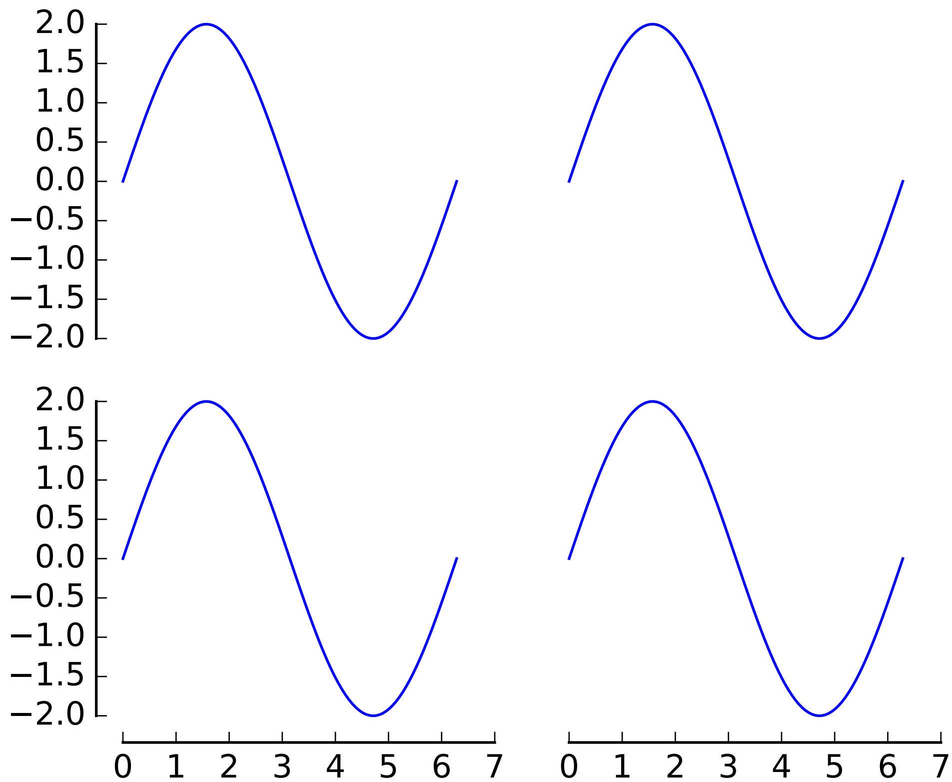

```
import matplotlib.pyplot as plt
import numpy as np

dt = 0.01
Fs = 1/dt
t = np.arange(0, 10, dt)
nse = np.random.randn(len(t))
r = np.exp(-t/0.05)
```

```
cnse = np.convolve(nse, r)*dt
cnse = cnse[:len(t)]
s = 0.1*np.sin(2*np.pi*t) + cnse

plt.subplot(3, 2, 1)
plt.plot(t, s)

plt.subplot(3, 2, 3)
plt.magnitude_spectrum(s, Fs=Fs)

plt.subplot(3, 2, 4)
plt.magnitude_spectrum(s, Fs=Fs, scale='dB')

plt.subplot(3, 2, 5)
plt.angle_spectrum(s, Fs=Fs)

plt.subplot(3, 2, 6)
plt.phase_spectrum(s, Fs=Fs)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.179 pylab_examples example code: spine_placement_demo.py





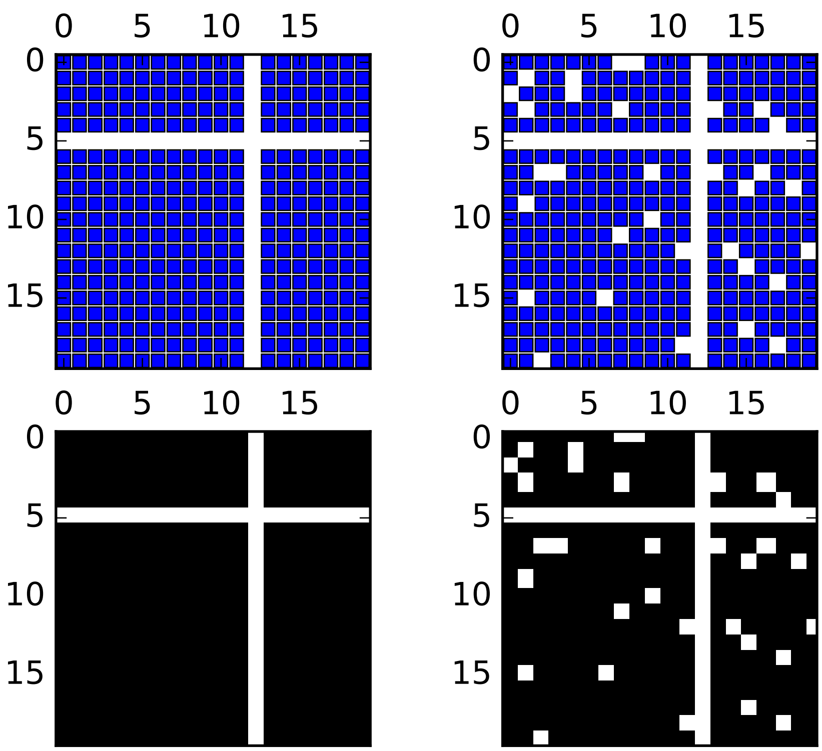

```
import numpy as np
import matplotlib.pyplot as plt


fig = plt.figure()
x = np.linspace(-np.pi, np.pi, 100)
y = 2*np.sin(x)

ax = fig.add_subplot(2, 2, 1)
ax.set_title('centered spines')
ax.plot(x, y)
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('center')
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ax = fig.add_subplot(2, 2, 2)
ax.set_title('zeroed spines')
ax.plot(x, y)
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
```

```
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ax = fig.add_subplot(2, 2, 3)
ax.set_title('spines at axes (0.6, 0.1)')
ax.plot(x, y)
ax.spines['left'].set_position(('axes', 0.6))
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position(('axes', 0.1))
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ax = fig.add_subplot(2, 2, 4)
ax.set_title('spines at data (1, 2)')
ax.plot(x, y)
ax.spines['left'].set_position(('data', 1))
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position(('data', 2))
ax.spines['top'].set_color('none')
ax.spines['left'].set_smart_bounds(True)
ax.spines['bottom'].set_smart_bounds(True)
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
# ----------------------------------------------------


def adjust_spines(ax, spines):
    for loc, spine in ax.spines.items():
        if loc in spines:
            spine.set_position(('outward', 10))  # outward by 10 points
            spine.set_smart_bounds(True)
        else:
            spine.set_color('none')  # don't draw spine

    # turn off ticks where there is no spine
    if 'left' in spines:
        ax.yaxis.set_ticks_position('left')
    else:
        # no yaxis ticks
        ax.yaxis.set_ticks([])

    if 'bottom' in spines:
        ax.xaxis.set_ticks_position('bottom')
    else:
        # no xaxis ticks
        ax.xaxis.set_ticks([])
```

```
fig = plt.figure()

x = np.linspace(0, 2*np.pi, 100)
y = 2*np.sin(x)

ax = fig.add_subplot(2, 2, 1)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, ['left'])

ax = fig.add_subplot(2, 2, 2)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, [])

ax = fig.add_subplot(2, 2, 3)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, ['left', 'bottom'])

ax = fig.add_subplot(2, 2, 4)
ax.plot(x, y, clip_on=False)
adjust_spines(ax, ['bottom'])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.180 pylab_examples example code: spy_demos.py

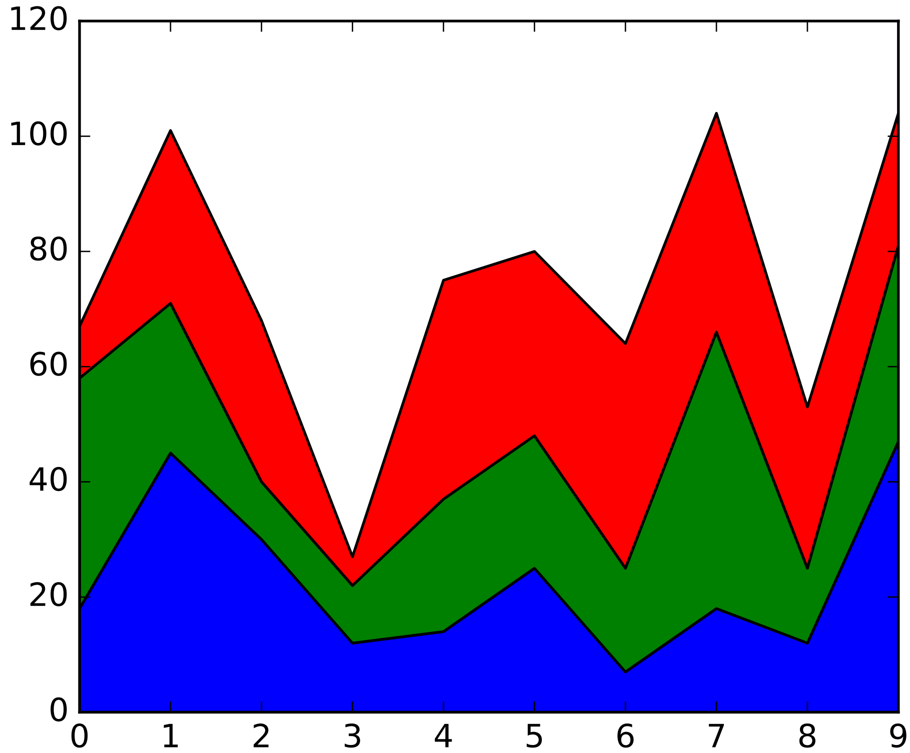


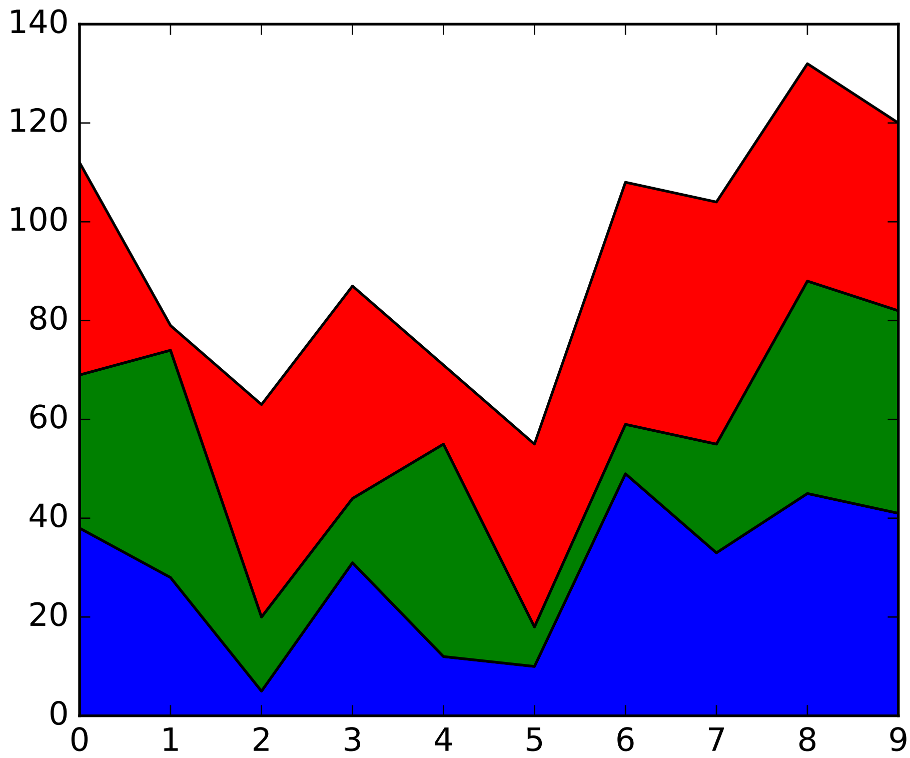

```
"""
Plot the sparsity pattern of arrays
"""

from matplotlib.pyplot import figure, show
import numpy

fig = figure()
ax1 = fig.add_subplot(221)
ax2 = fig.add_subplot(222)
ax3 = fig.add_subplot(223)
ax4 = fig.add_subplot(224)

x = numpy.random.randn(20, 20)
x[5] = 0.
x[:, 12] = 0.

ax1.spy(x, markersize=5)
ax2.spy(x, precision=0.1, markersize=5)

ax3.spy(x)
ax4.spy(x, precision=0.1)
```

```
show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.181 pylab_examples example code: stackplot_demo.py

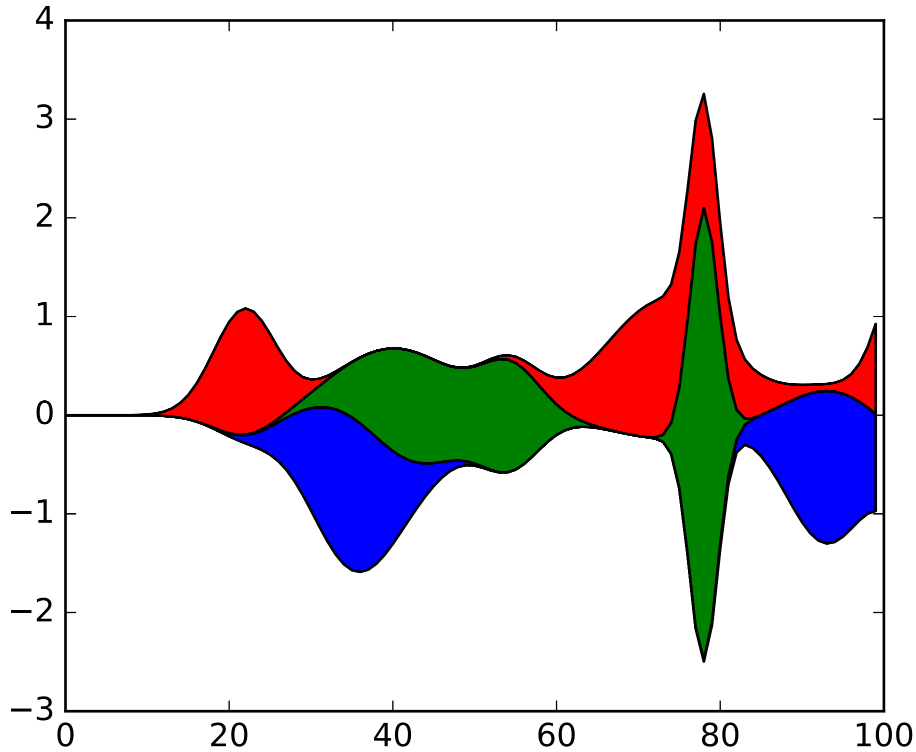





```
import numpy as np
import matplotlib.pyplot as plt


def fnx():
    return np.random.randint(5, 50, 10)

y = np.row_stack((fnx(), fnx(), fnx()))
x = np.arange(10)

y1, y2, y3 = fnx(), fnx(), fnx()

fig, ax = plt.subplots()
ax.stackplot(x, y)
plt.show()

fig, ax = plt.subplots()
ax.stackplot(x, y1, y2, y3)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.182 pylab_examples example code: stackplot_demo2.py

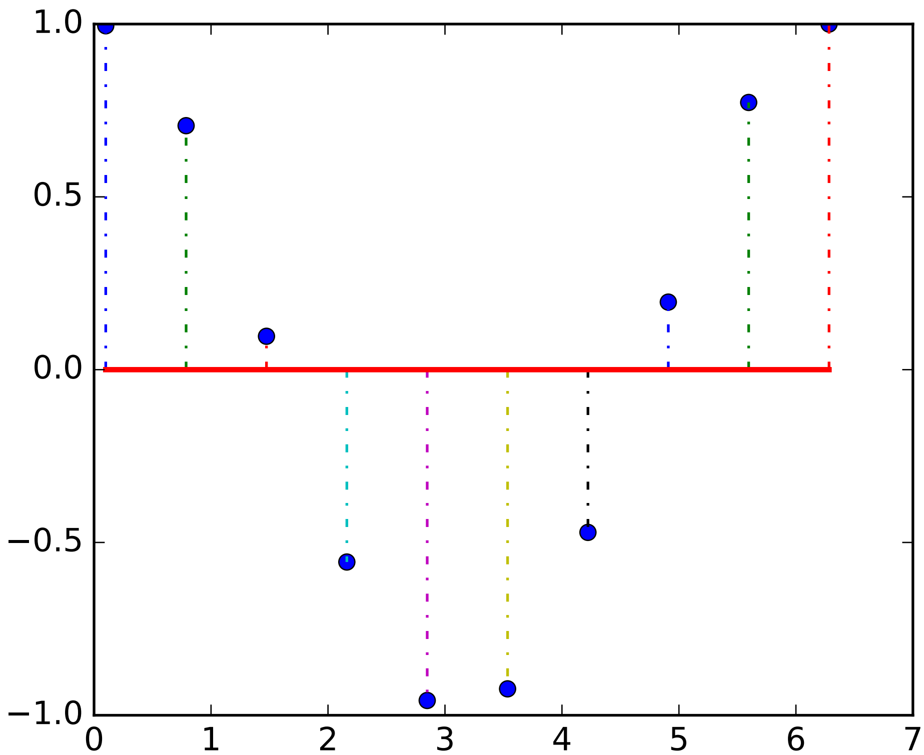



```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)


def layers(n, m):
    """
    Return *n* random Gaussian mixtures, each of length *m*.
    """
    def bump(a):
        x = 1 / (.1 + np.random.random())
        y = 2 * np.random.random() - .5
        z = 10 / (.1 + np.random.random())
        for i in range(m):
            w = (i / float(m) - y) * z
            a[i] += x * np.exp(-w * w)
    a = np.zeros((m, n))
    for i in range(n):
        for j in range(5):
            bump(a[:, i])
    return a
```

```
d = layers(3, 100)

plt.subplots()
plt.stackplot(range(100), d.T, baseline='wiggle')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

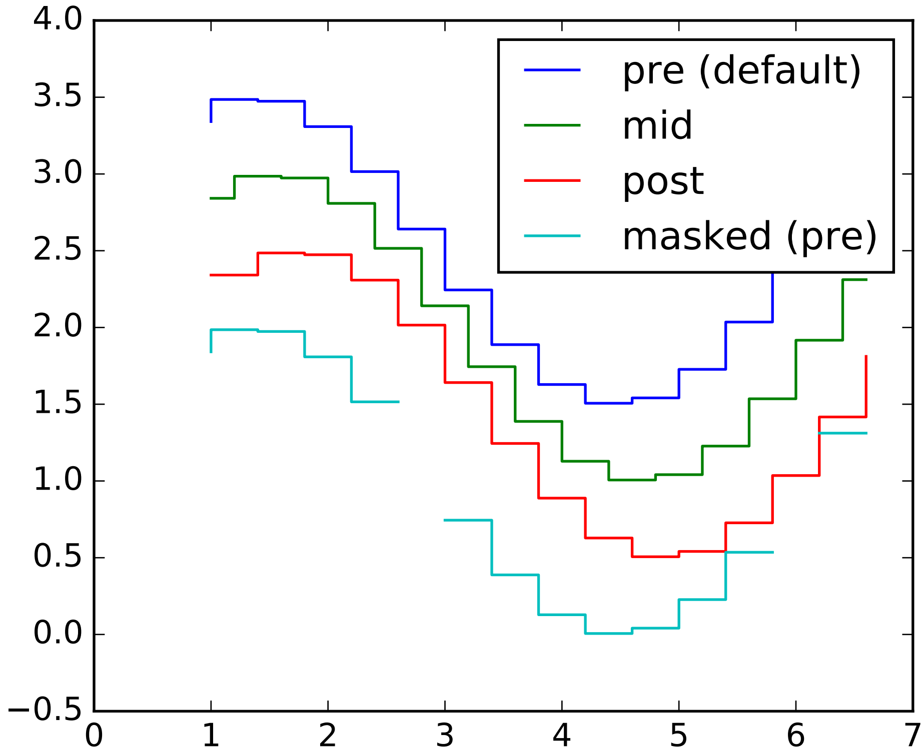
90.183 pylab_examples example code: stem_plot.py




```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0.1, 2*np.pi, 10)
markerline, stemlines, baseline = plt.stem(x, np.cos(x), '-.')
plt.setp(markerline, 'markerfacecolor', 'b')
plt.setp(baseline, 'color', 'r', 'linewidth', 2)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.184 pylab_examples example code: step_demo.py

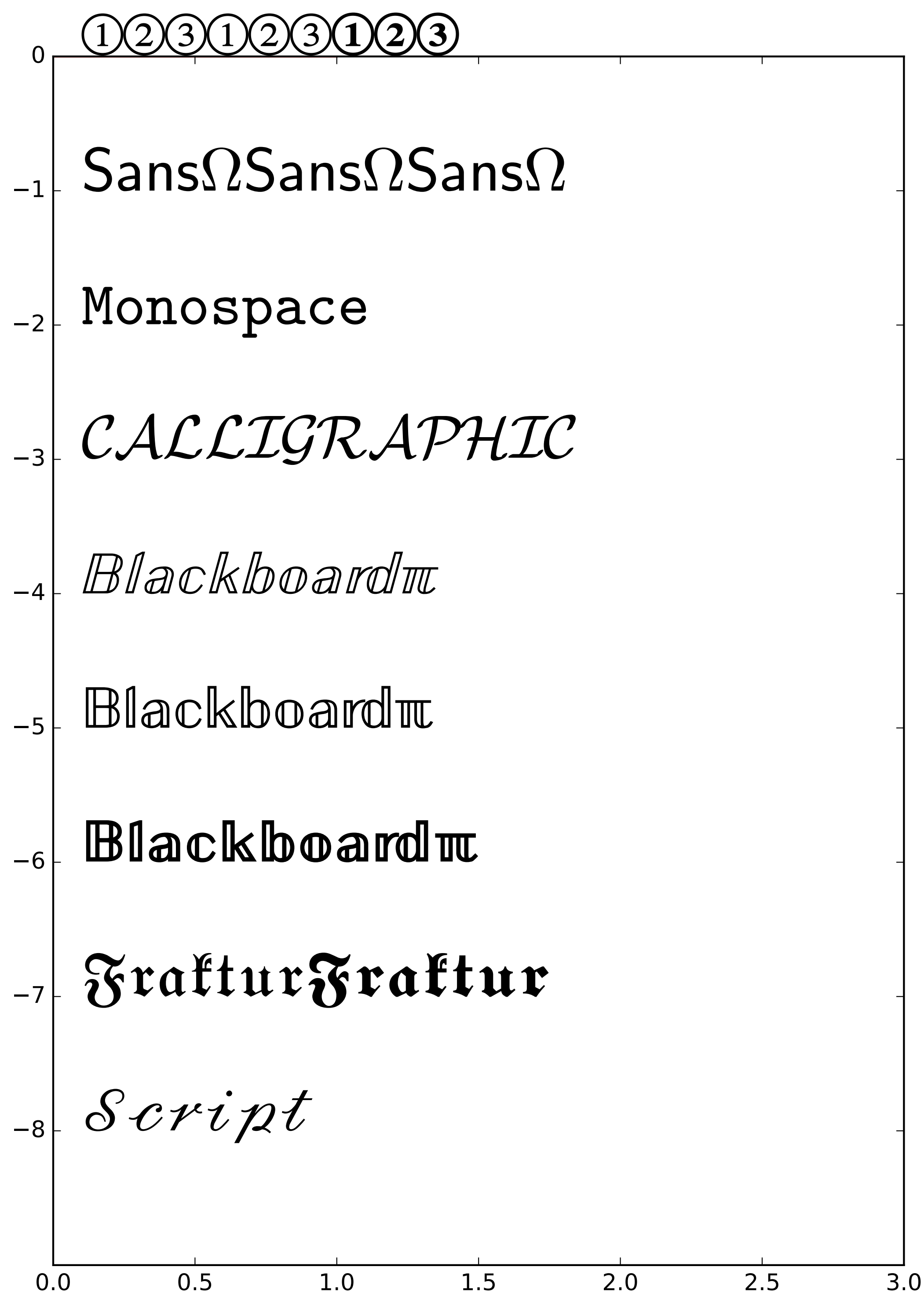



```
import numpy as np
from numpy import ma
import matplotlib.pyplot as plt

x = np.arange(1, 7, 0.4)
y0 = np.sin(x)
y = y0.copy() + 2.5

plt.step(x, y, label='pre (default)')

y -= 0.5
plt.step(x, y, where='mid', label='mid')

y -= 0.5
plt.step(x, y, where='post', label='post')

y = ma.masked_where((y0 > -0.15) & (y0 < 0.15), y - 0.5)
plt.step(x, y, label='masked (pre)')

plt.legend()

plt.xlim(0, 7)
```

```
plt.ylim(-0.5, 4)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.185 pylab_examples example code: stix_fonts_demo.py

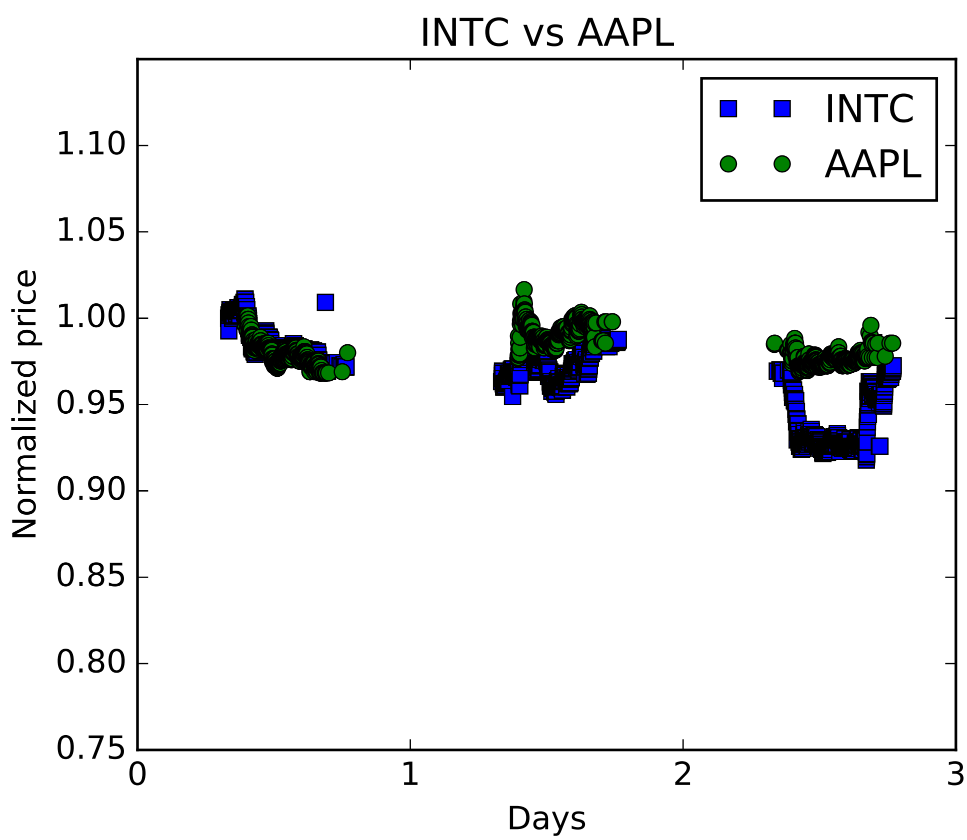


```
from __future__ import unicode_literals

import os
import sys
import re
import gc
import matplotlib.pyplot as plt
import numpy as np

stests = [
    r'$\mathcircled{123} \mathrm{\mathcircled{123}}'
    r' \mathbf{\mathcircled{123}}$',
    r'$\mathsf{Sans \Omega} \mathrm{\mathsf{Sans \Omega}}'
    r' \mathbf{\mathsf{Sans \Omega}}$',
    r'$\mathtt{Monospace}$',
    r'$\mathcal{CALLIGRAPHIC}$',
    r'$\mathbb{Blackboard \pi}$',
    r'$\mathrm{\mathbb{Blackboard \pi}}$',
    r'$\mathbf{\mathbb{Blackboard \pi}}$',
    r'$\mathfrak{Fraktur} \mathbf{\mathfrak{Fraktur}}$',
    r'$\mathscr{Script}$']

if sys.maxunicode > 0xffff:
    s = r'Direct Unicode: $\u23ce \mathrm{\ue0f2 \U0001D538}$'


def doall():
    tests = stests

    plt.figure(figsize=(8, (len(tests) * 1) + 2))
    plt.plot([0, 0], 'r')
    plt.grid(False)
    plt.axis([0, 3, -len(tests), 0])
    plt.yticks(np.arange(len(tests)) * -1)
    for i, s in enumerate(tests):
        plt.text(0.1, -i, s, fontsize=32)

    plt.savefig('stix_fonts_example')
    plt.show()


if '--latex' in sys.argv:
    fd = open("stix_fonts_examples.ltx", "w")
    fd.write("\\documentclass{article}\n")
    fd.write("\\begin{document}\n")
    fd.write("\\begin{enumerate}\n")

    for i, s in enumerate(stests):
        s = re.sub(r"(?<!\\)\$", "$$", s)
        fd.write("\\item %s\n" % s)

    fd.write("\\end{enumerate}\n")
    fd.write("\\end{document}\n")
```

```
    fd.close()

    os.system("pdflatex stix_fonts_examples.ltx")
else:
    doall()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.186 pylab_examples example code: stock_demo.py

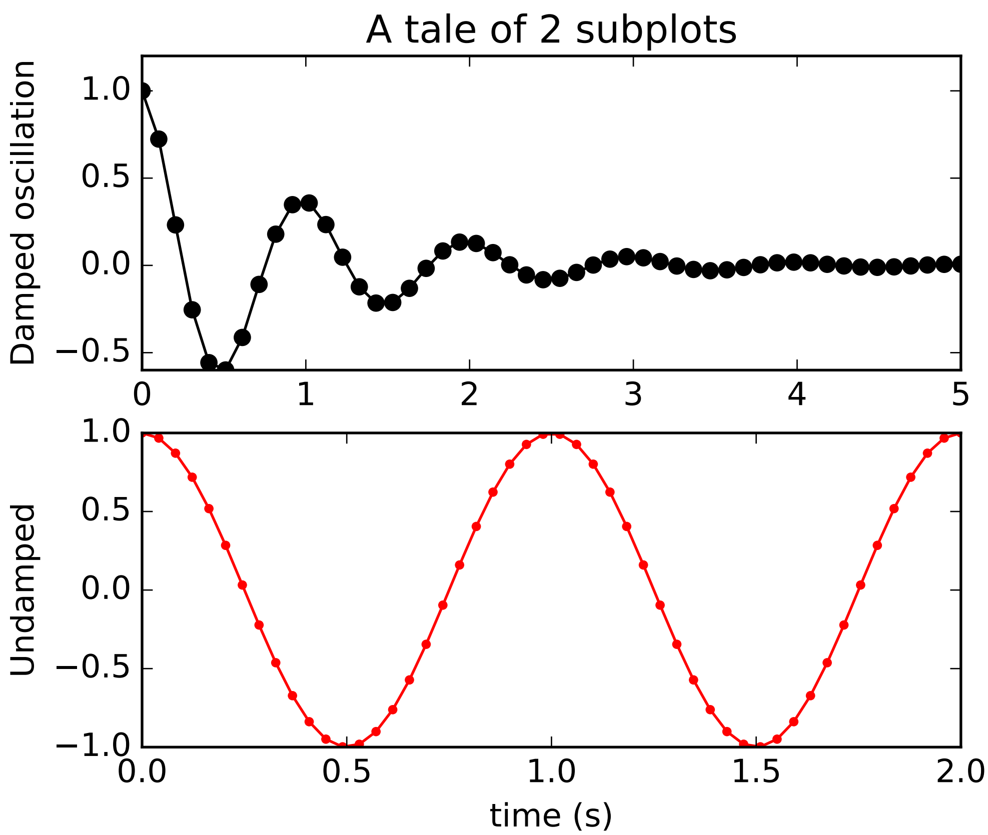



```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.ticker import MultipleLocator
from data_helper import get_two_stock_data

d1, p1, d2, p2 = get_two_stock_data()

fig, ax = plt.subplots()
lines = plt.plot(d1, p1, 'bs', d2, p2, 'go')
plt.xlabel('Days')
plt.ylabel('Normalized price')
```

```
plt.xlim(0, 3)
ax.xaxis.set_major_locator(MultipleLocator(1))

plt.title('INTC vs AAPL')
plt.legend(('INTC', 'AAPL'))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.187 pylab_examples example code: subplot_demo.py

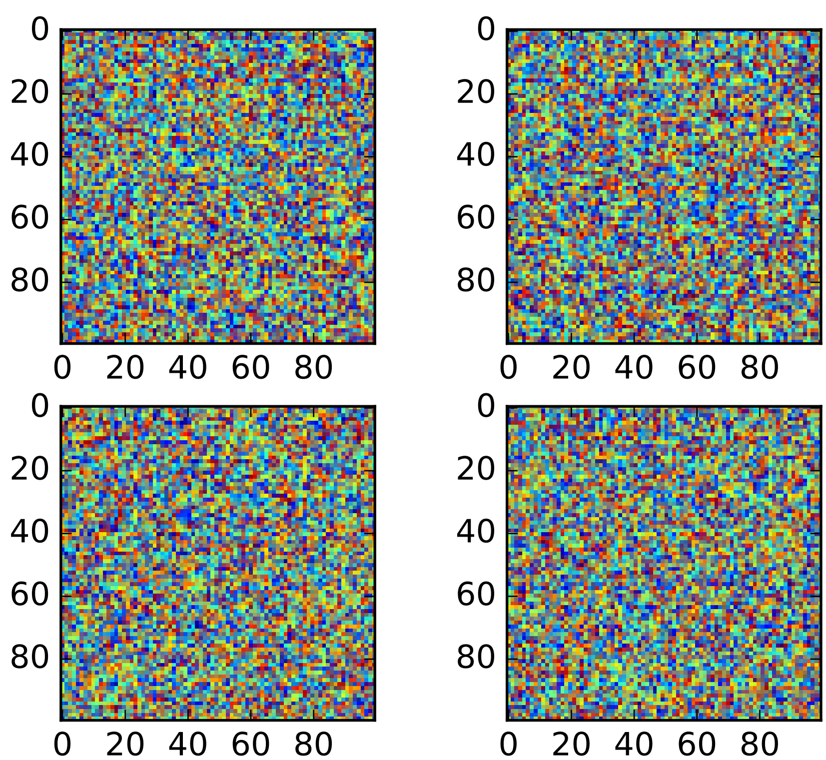


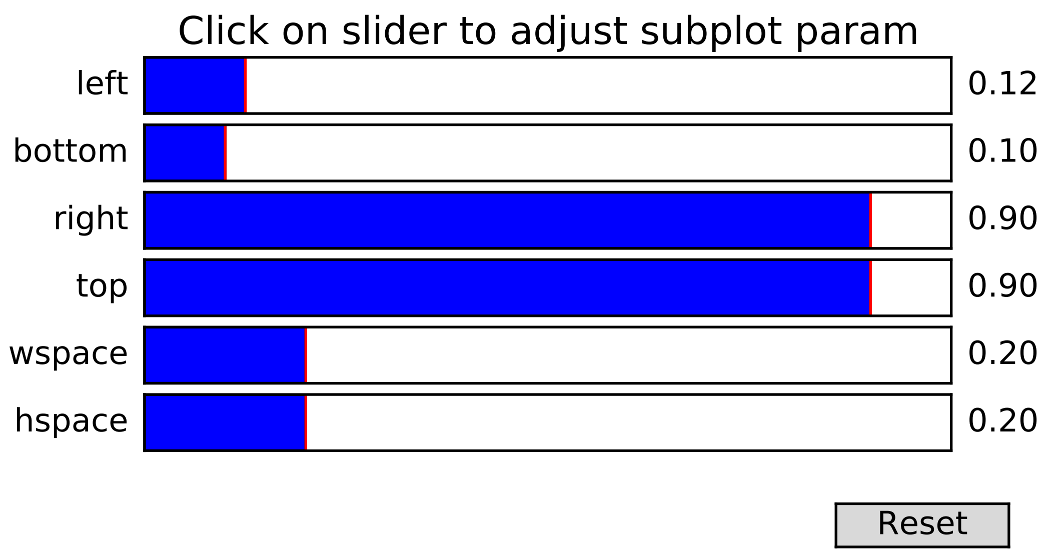

```
"""
Simple demo with multiple subplots.
"""
import numpy as np
import matplotlib.pyplot as plt


x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)
```

```
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'ko-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.188 pylab_examples example code: subplot_toolbar.py

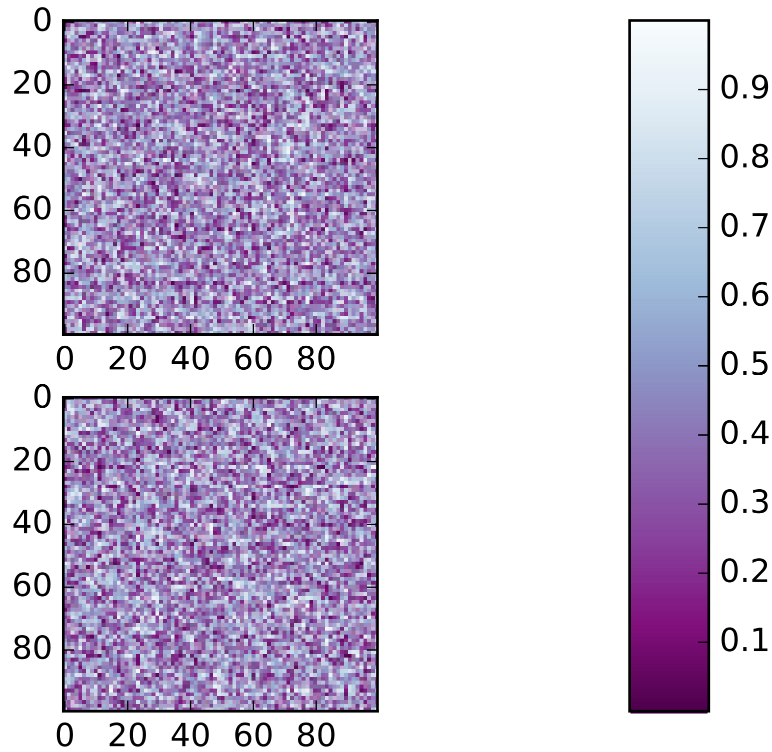





```
import matplotlib.pyplot as plt
import numpy.random as rnd

fig = plt.figure()
plt.subplot(221)
plt.imshow(rnd.random((100, 100)))
plt.subplot(222)
plt.imshow(rnd.random((100, 100)))
plt.subplot(223)
plt.imshow(rnd.random((100, 100)))
plt.subplot(224)
plt.imshow(rnd.random((100, 100)))

plt.subplot_tool()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

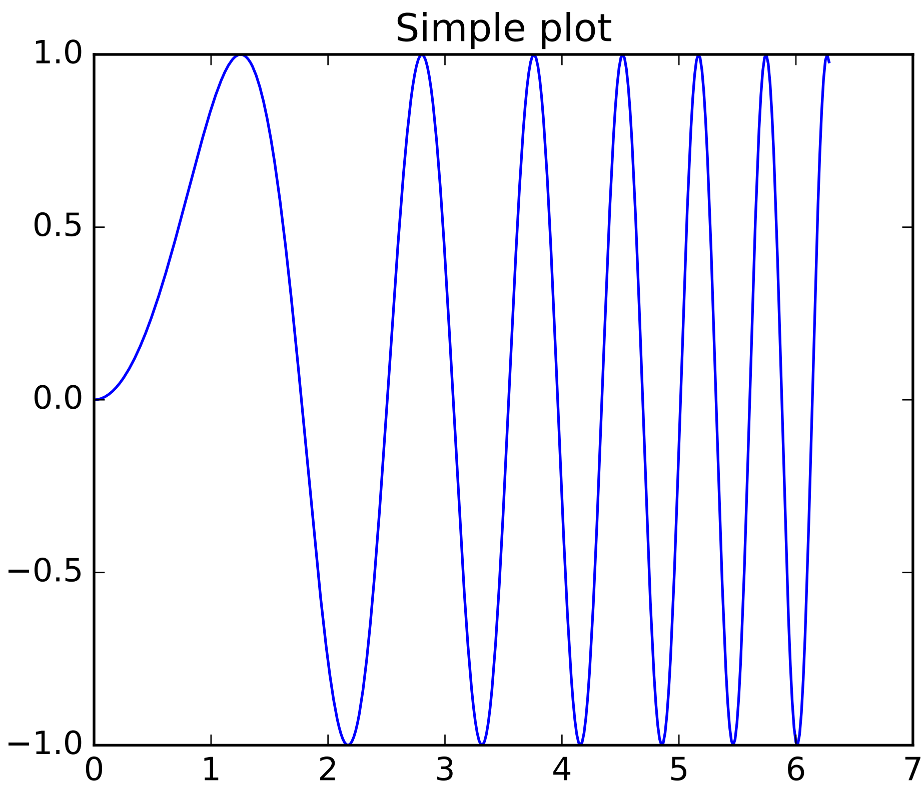
90.189 pylab_examples example code: subplots_adjust.py

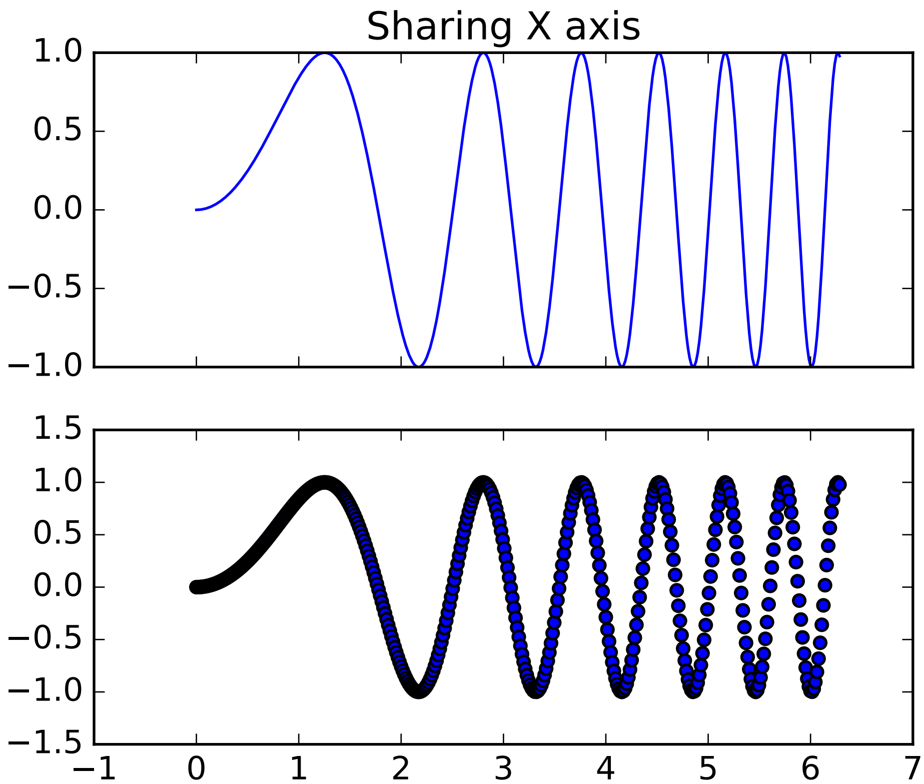


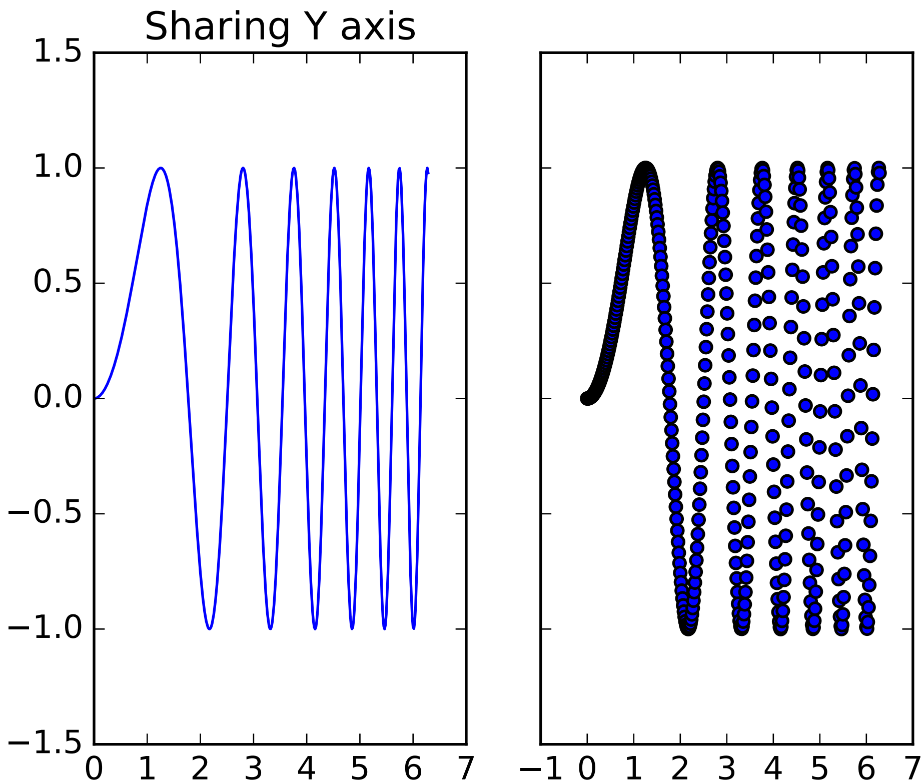

```
import matplotlib.pyplot as plt
import numpy as np

plt.subplot(211)
plt.imshow(np.random.random((100, 100)), cmap=plt.cm.BuPu_r)
plt.subplot(212)
plt.imshow(np.random.random((100, 100)), cmap=plt.cm.BuPu_r)

plt.subplots_adjust(bottom=0.1, right=0.8, top=0.9)
cax = plt.axes([0.85, 0.1, 0.075, 0.8])
plt.colorbar(cax=cax)
plt.show()
```

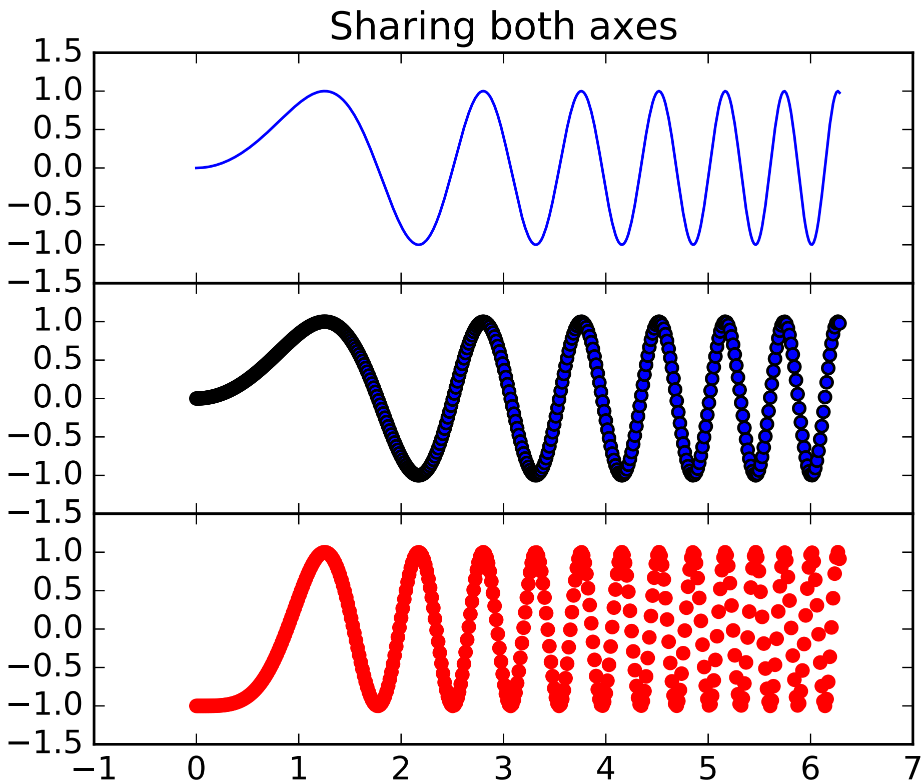
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

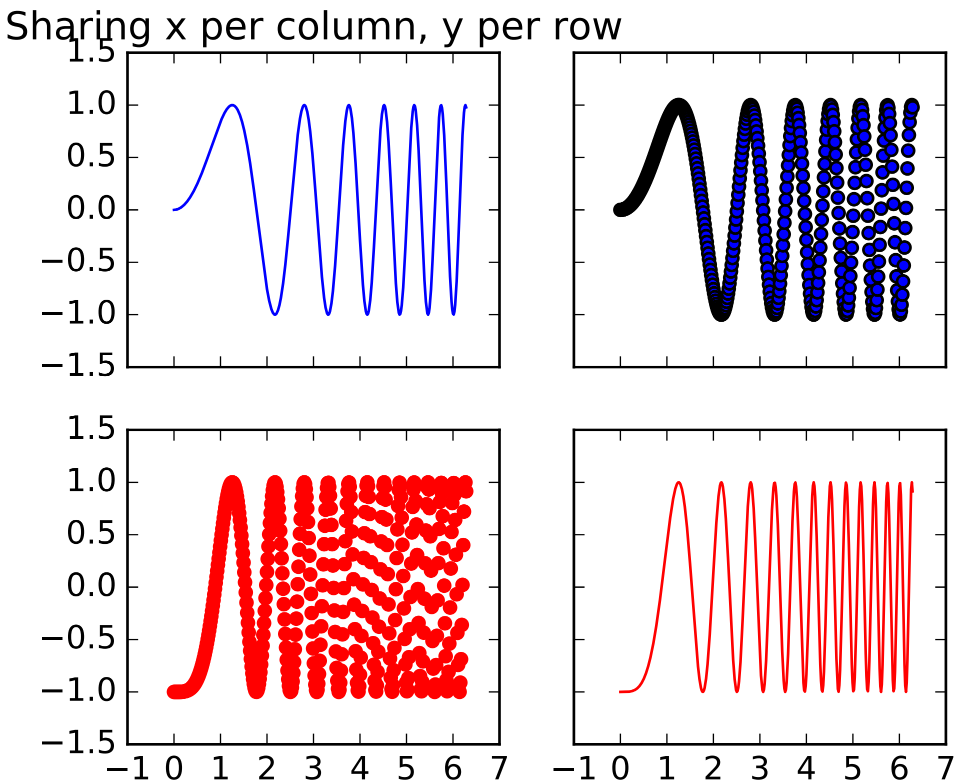
90.190 pylab_examples example code: subplots_demo.py

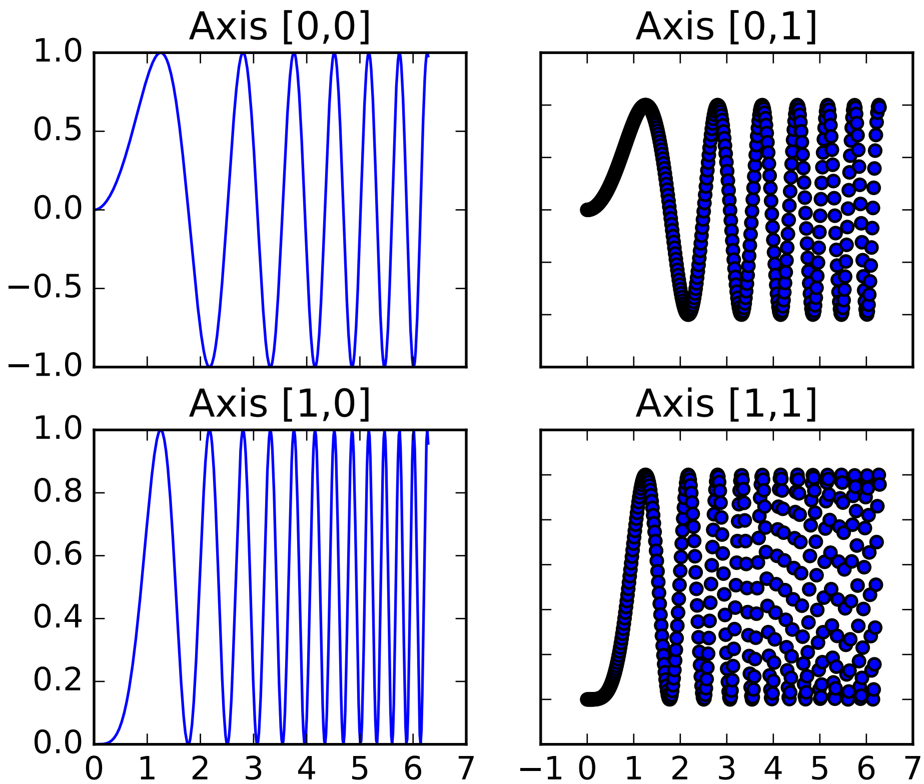


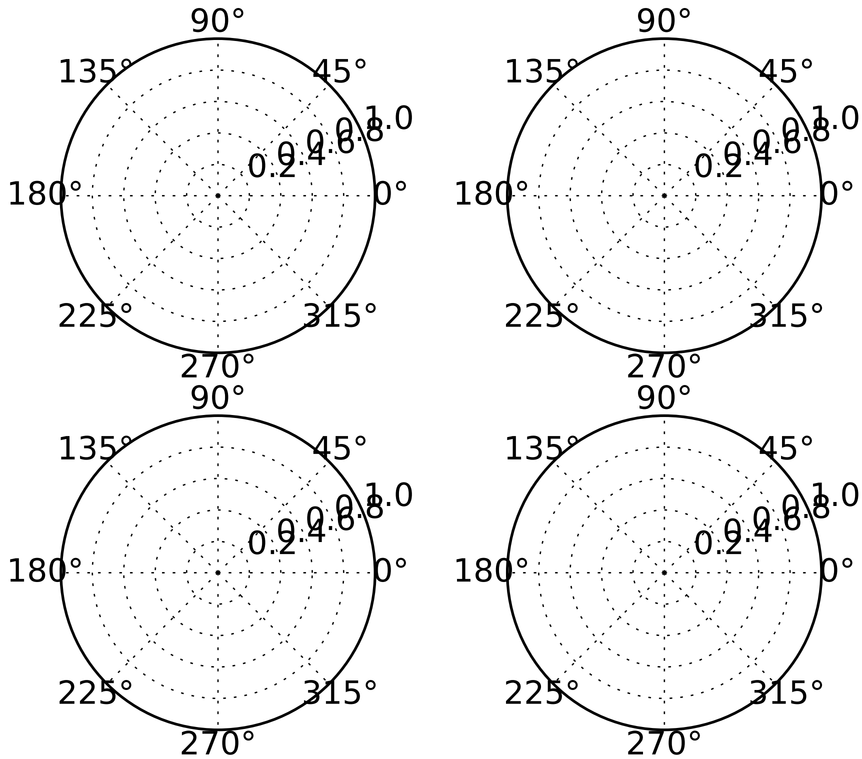

Sharing X axis
1.0
0.5
0.0
−0.5
−1.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1
0
1
2
3
4
5
6
7


Sharing Y axis
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1
2
3
4
5
6
7
−1
0
1
2
3
4
5
6
7


Sharing both axes
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1
0
1
2
3
4
5
6
7


Sharing x per column, y per row
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1 0 1 2 3 4 5 6 7
−1 0 1 2 3 4 5 6 7


Axis [0,0]
Axis [0,1]
Axis [1,0]
Axis [1,1]
1.0
0.5
0.0
−0.5
−1.0
1.0
0.8
0.6
0.4
0.2
0.0
0 1 2 3 4 5 6 7
−1 0 1 2 3 4 5 6 7

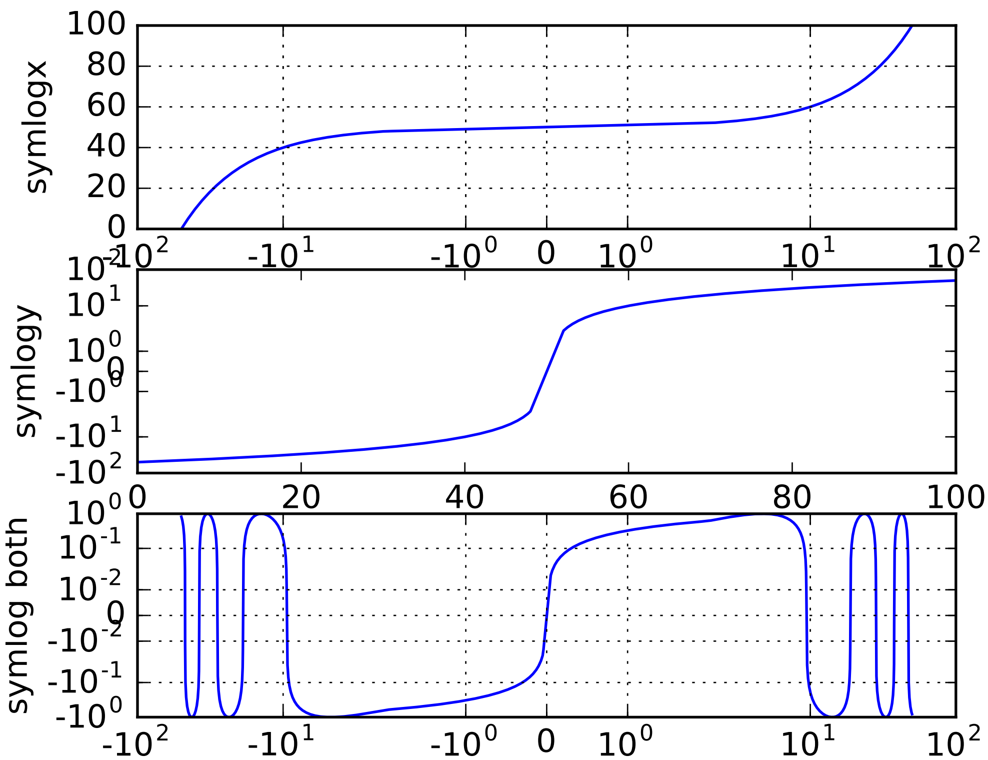



```
"""Examples illustrating the use of plt.subplots().

This function creates a figure and a grid of subplots with a single call, while
providing reasonable control over how the individual plots are created.  For
very refined tuning of subplot creation, you can still use add_subplot()
directly on a new figure.
"""

import matplotlib.pyplot as plt
import numpy as np

# Simple data to display in various forms
x = np.linspace(0, 2 * np.pi, 400)
y = np.sin(x ** 2)

plt.close('all')

# Just a figure and one subplot
f, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Two subplots, the axes array is 1-d
f, axarr = plt.subplots(2, sharex=True)
axarr[0].plot(x, y)
axarr[0].set_title('Sharing X axis')
```

```
axarr[1].scatter(x, y)

# Two subplots, unpack the axes array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Three subplots sharing both x/y axes
f, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing both axes')
ax2.scatter(x, y)
ax3.scatter(x, 2 * y ** 2 - 1, color='r')
# Fine-tune figure; make subplots close to each other and hide x ticks for
# all but bottom plot.
f.subplots_adjust(hspace=0)
plt.setp([a.get_xticklabels() for a in f.axes[:-1]], visible=False)

# row and column sharing
f, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex='col', sharey='row')
ax1.plot(x, y)
ax1.set_title('Sharing x per column, y per row')
ax2.scatter(x, y)
ax3.scatter(x, 2 * y ** 2 - 1, color='r')
ax4.plot(x, 2 * y ** 2 - 1, color='r')

# Four axes, returned as a 2-d array
f, axarr = plt.subplots(2, 2)
axarr[0, 0].plot(x, y)
axarr[0, 0].set_title('Axis [0,0]')
axarr[0, 1].scatter(x, y)
axarr[0, 1].set_title('Axis [0,1]')
axarr[1, 0].plot(x, y ** 2)
axarr[1, 0].set_title('Axis [1,0]')
axarr[1, 1].scatter(x, y ** 2)
axarr[1, 1].set_title('Axis [1,1]')
# Fine-tune figure; hide x ticks for top plots and y ticks for right plots
plt.setp([a.get_xticklabels() for a in axarr[0, :]], visible=False)
plt.setp([a.get_yticklabels() for a in axarr[:, 1]], visible=False)

# Four polar axes
plt.subplots(2, 2, subplot_kw=dict(projection='polar'))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.191 pylab_examples example code: symlog_demo.py

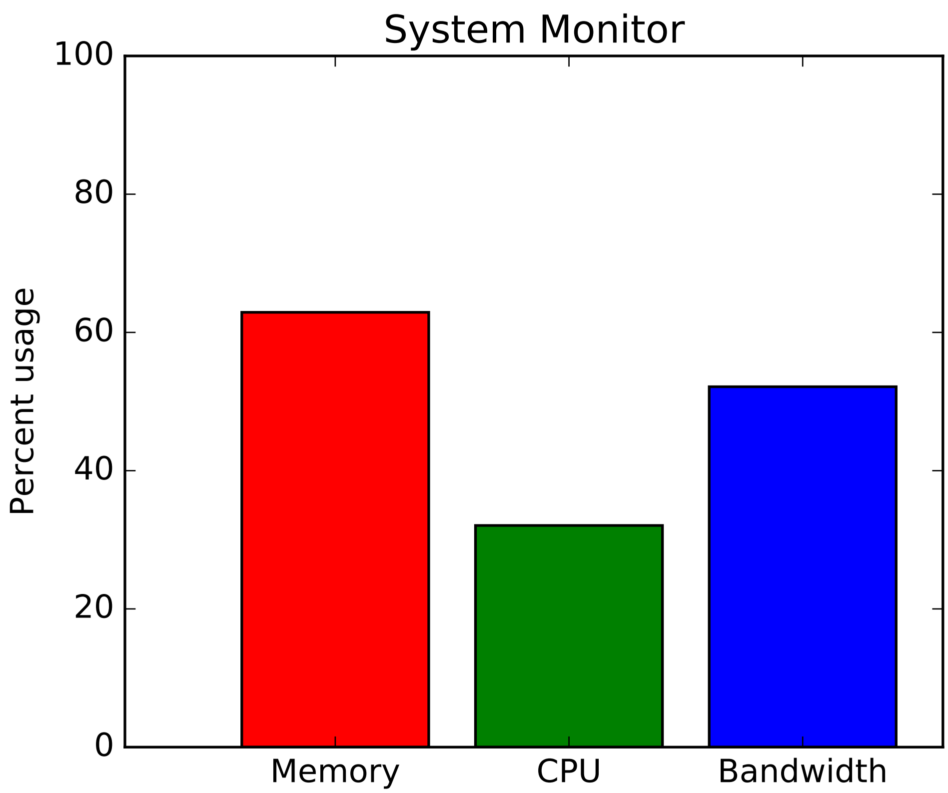



```
#!/usr/bin/env python

import matplotlib.pyplot as plt
import numpy as np

dt = 0.01
x = np.arange(-50.0, 50.0, dt)
y = np.arange(0, 100.0, dt)

plt.subplot(311)
plt.plot(x, y)
plt.xscale('symlog')
plt.ylabel('symlogx')
plt.grid(True)
plt.gca().xaxis.grid(True, which='minor')  # minor grid on too

plt.subplot(312)
plt.plot(y, x)
plt.yscale('symlog')
plt.ylabel('symlogy')

plt.subplot(313)
```

```
plt.plot(x, np.sin(x / 3.0))
plt.xscale('symlog')
plt.yscale('symlog', linthreshy=0.015)
plt.grid(True)
plt.ylabel('symlog both')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.192 pylab_examples example code: system_monitor.py



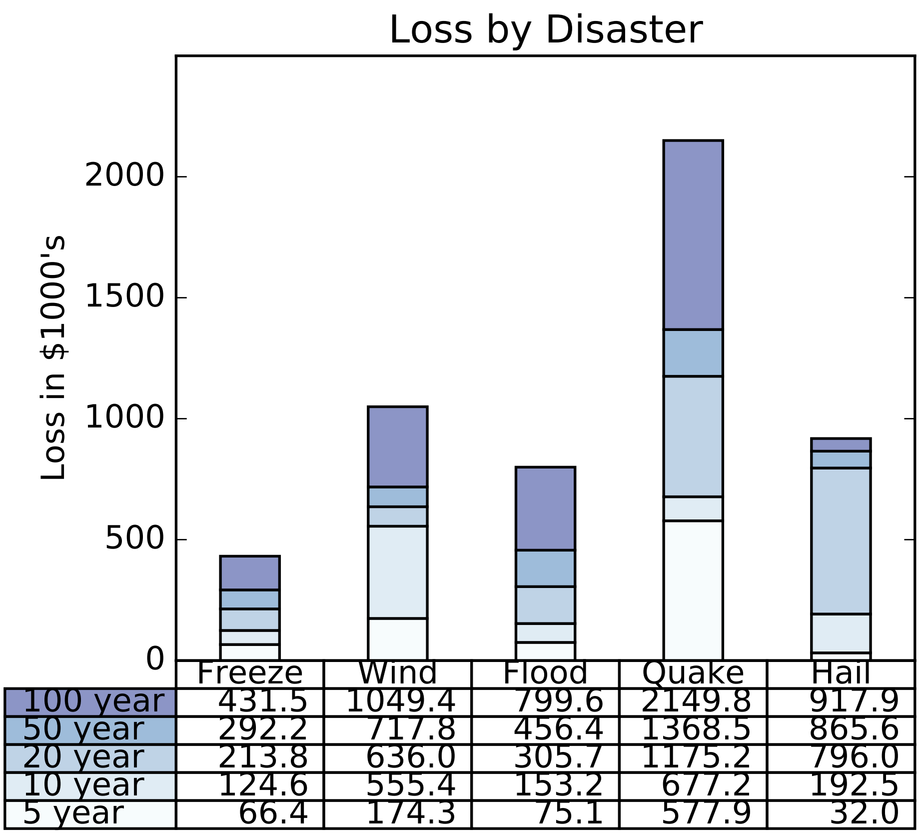

```
import time
import matplotlib.pyplot as plt
import numpy as np


def get_memory():
    "Simulate a function that returns system memory"
    return 100*(0.5 + 0.5*np.sin(0.5*np.pi*time.time()))
```

```
def get_cpu():
    "Simulate a function that returns cpu usage"
    return 100*(0.5 + 0.5*np.sin(0.2*np.pi*(time.time() - 0.25)))


def get_net():
    "Simulate a function that returns network bandwidth"
    return 100*(0.5 + 0.5*np.sin(0.7*np.pi*(time.time() - 0.1)))


def get_stats():
    return get_memory(), get_cpu(), get_net()

fig, ax = plt.subplots()
ind = np.arange(1, 4)

# show the figure, but do not block
plt.show(block=False)


pm, pc, pn = plt.bar(ind, get_stats())
centers = ind + 0.5*pm.get_width()
pm.set_facecolor('r')
pc.set_facecolor('g')
pn.set_facecolor('b')
ax.set_xlim([0.5, 4])
ax.set_xticks(centers)
ax.set_ylim([0, 100])
ax.set_xticklabels(['Memory', 'CPU', 'Bandwidth'])
ax.set_ylabel('Percent usage')
ax.set_title('System Monitor')

start = time.time()
for i in range(200):  # run for a little while
    m, c, n = get_stats()

    # update the animated artists
    pm.set_height(m)
    pc.set_height(c)
    pn.set_height(n)

    # ask the canvas to re-draw itself the next time it
    # has a chance.
    # For most of the GUI backends this adds an event to the queue
    # of the GUI frameworks event loop.
    fig.canvas.draw_idle()
    try:
        # make sure that the GUI framework has a chance to run its event loop
        # and clear any GUI events.  This needs to be in a try/except block
        # because the default implemenation of this method is to raise
        # NotImplementedError
        fig.canvas.flush_events()
    except NotImplementedError:
```

```
        pass

stop = time.time()
print("{fps:.1f} frames per second".format(fps=200 / (stop - start)))
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.193 pylab_examples example code: table_demo.py

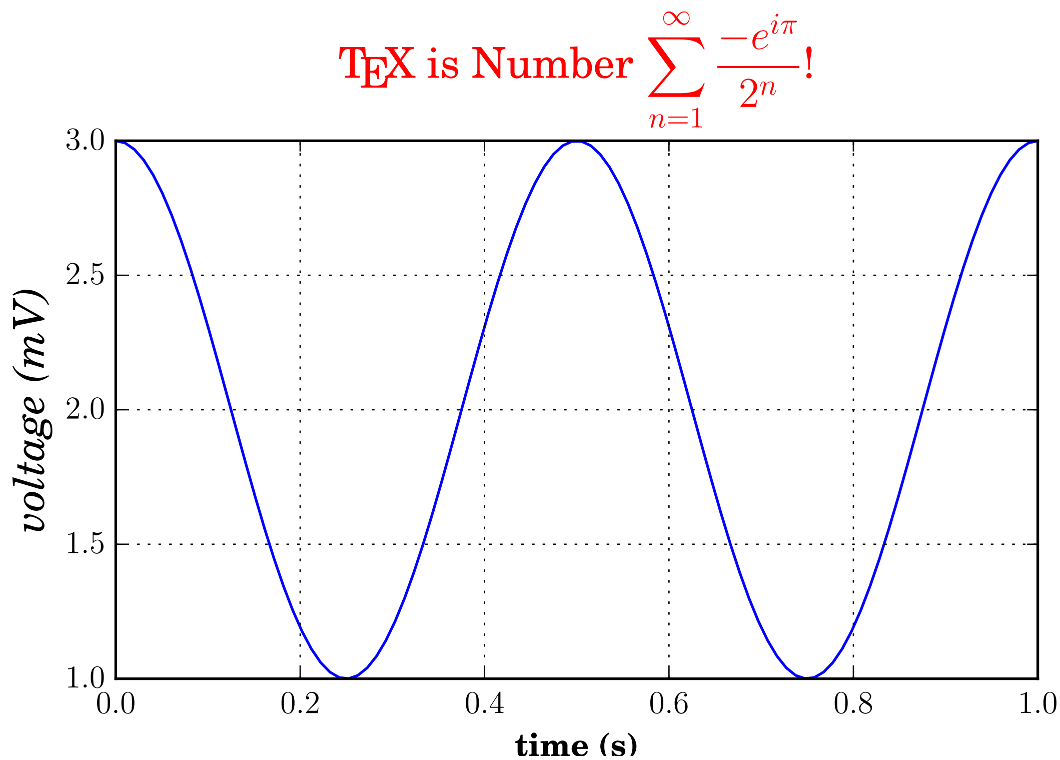



```
"""
Demo of table function to display a table within a plot.
"""
import numpy as np
import matplotlib.pyplot as plt


data = [[  66386,  174296,   75131,  577908,   32015],
        [  58230,  381139,   78045,   99308,  160454],
        [  89135,   80552,  152558,  497981,  603535],
        [  78415,   81858,  150656,  193263,   69638],
        [ 139361,  331509,  343164,  781380,   52269]]
```

```
columns = ('Freeze', 'Wind', 'Flood', 'Quake', 'Hail')
rows = ['%d year' % x for x in (100, 50, 20, 10, 5)]

values = np.arange(0, 2500, 500)
value_increment = 1000

# Get some pastel shades for the colors
colors = plt.cm.BuPu(np.linspace(0, 0.5, len(rows)))
n_rows = len(data)

index = np.arange(len(columns)) + 0.3
bar_width = 0.4

# Initialize the vertical-offset for the stacked bar chart.
y_offset = np.array([0.0] * len(columns))

# Plot bars and create text labels for the table
cell_text = []
for row in range(n_rows):
    plt.bar(index, data[row], bar_width, bottom=y_offset, color=colors[row])
    y_offset = y_offset + data[row]
    cell_text.append(['%1.1f' % (x/1000.0) for x in y_offset])
# Reverse colors and text labels to display the last value at the top.
colors = colors[::-1]
cell_text.reverse()

# Add a table at the bottom of the axes
the_table = plt.table(cellText=cell_text,
                      rowLabels=rows,
                      rowColours=colors,
                      colLabels=columns,
                      loc='bottom')

# Adjust layout to make room for the table:
plt.subplots_adjust(left=0.2, bottom=0.2)

plt.ylabel("Loss in ${0}'s".format(value_increment))
plt.yticks(values * value_increment, ['%d' % val for val in values])
plt.xticks([])
plt.title('Loss by Disaster')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.194 pylab_examples example code: tex_demo.py

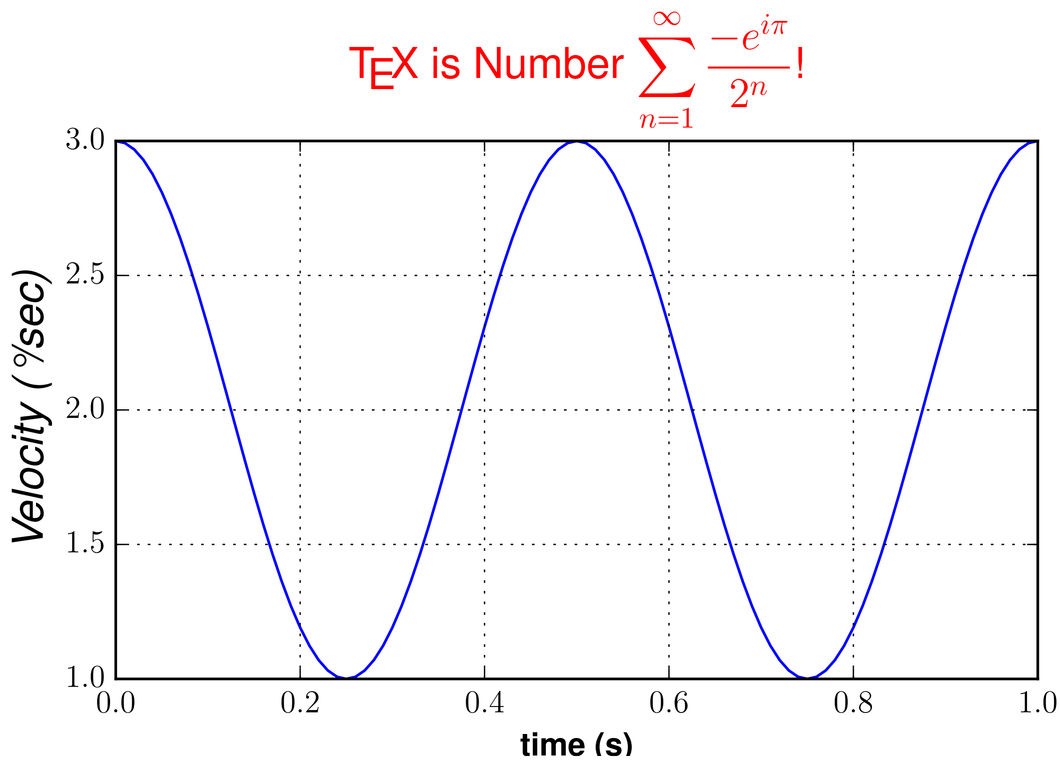



```
"""
You can use TeX to render all of your matplotlib text if the rc
parameter text.usetex is set.  This works currently on the agg and ps
backends, and requires that you have tex and the other dependencies
described at http://matplotlib.org/users/usetex.html
properly installed on your system.  The first time you run a script
you will see a lot of output from tex and associated tools.  The next
time, the run may be silent, as a lot of the information is cached in
~/.tex.cache

"""
import numpy as np
import matplotlib.pyplot as plt


plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.figure(1, figsize=(6, 4))
ax = plt.axes([0.1, 0.1, 0.8, 0.7])
t = np.linspace(0.0, 1.0, 100)
s = np.cos(4 * np.pi * t) + 2
plt.plot(t, s)

plt.xlabel(r'\textbf{time (s)}')
plt.ylabel(r'\textit{voltage (mV)}', fontsize=16)
```

```
plt.title(r"\TeX\ is Number $\displaystyle\sum_{n=1}^\infty"
          r"\frac{-e^{i\pi}}{2^n}$!", fontsize=16, color='r')
plt.grid(True)
plt.savefig('tex_demo')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.195 pylab_examples example code: tex_unicode_demo.py

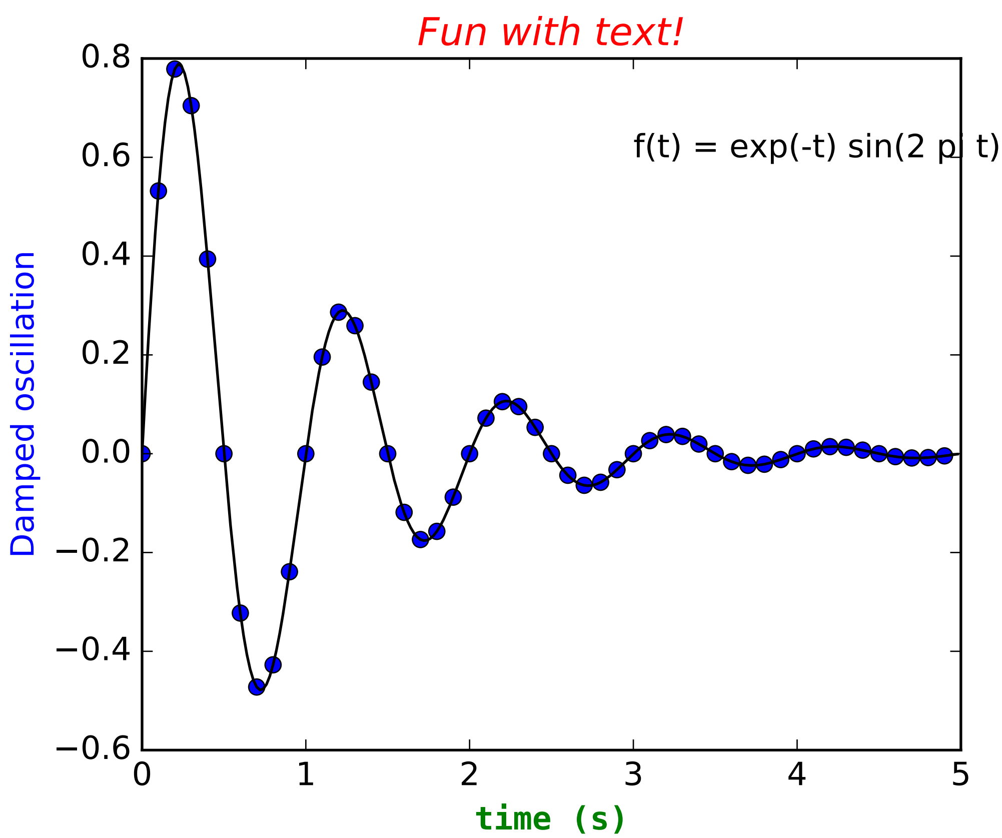



```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
This demo is tex_demo.py modified to have unicode. See that file for
more information.
"""

from __future__ import unicode_literals
import numpy as np
import matplotlib
matplotlib.rcParams['text.usetex'] = True
matplotlib.rcParams['text.latex.unicode'] = True
import matplotlib.pyplot as plt

plt.figure(1, figsize=(6, 4))
```

```
ax = plt.axes([0.1, 0.1, 0.8, 0.7])
t = np.arange(0.0, 1.0 + 0.01, 0.01)
s = np.cos(2*2*np.pi*t) + 2
plt.plot(t, s)

plt.xlabel(r'\textbf{time (s)}')
plt.ylabel('\\textit{Velocity (\u00B0/sec)}', fontsize=16)
plt.title(r'\TeX\ is Number $\displaystyle\sum_{n=1}^\infty'
          r'\frac{-e^{i\pi}}{2^n}$!', fontsize=16, color='r')
plt.grid(True)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.196 pylab_examples example code: text_handles.py

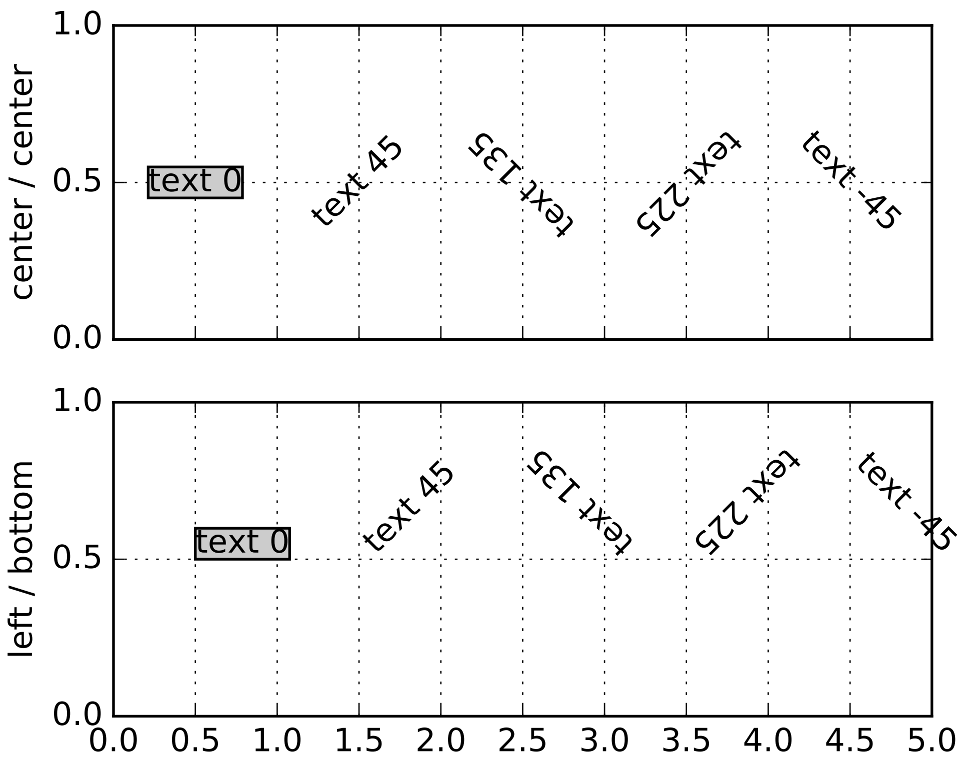



```
#!/usr/bin/env python
# Controlling the properties of axis text using handles

# See examples/text_themes.py for a more elegant, pythonic way to control
# fonts.  After all, if we were slaves to MATLAB , we wouldn't be
# using python!
```

```
import matplotlib.pyplot as plt
import numpy as np


def f(t):
    s1 = np.sin(2*np.pi*t)
    e1 = np.exp(-t)
    return np.multiply(s1, e1)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

fig, ax = plt.subplots()
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.text(3.0, 0.6, 'f(t) = exp(-t) sin(2 pi t)')
ttext = plt.title('Fun with text!')
ytext = plt.ylabel('Damped oscillation')
xtext = plt.xlabel('time (s)')

plt.setp(ttext, size='large', color='r', style='italic')
plt.setp(xtext, size='medium', name=['Courier', 'Bitstream Vera Sans Mono'],
     weight='bold', color='g')
plt.setp(ytext, size='medium', name=['Helvetica', 'Bitstream Vera Sans'],
     weight='light', color='b')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.197 pylab_examples example code: text_rotation.py

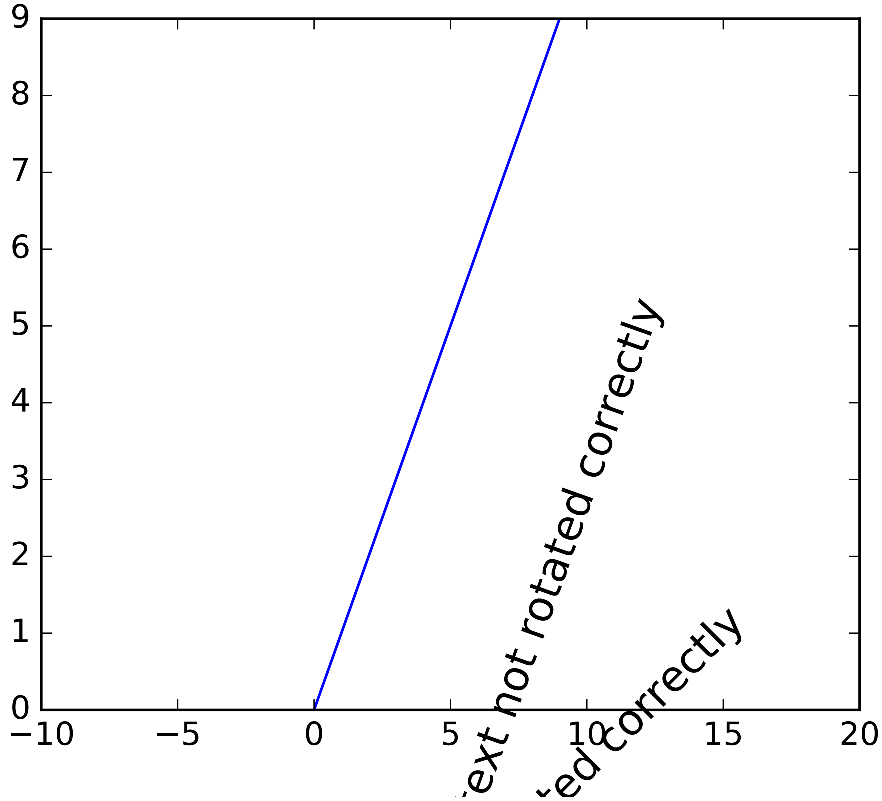



```
"""
The way matplotlib does text layout is counter-intuitive to some, so
this example is designed to make it a little clearer.  The text is
aligned by it's bounding box (the rectangular box that surrounds the
ink rectangle).  The order of operations is basically rotation then
alignment, rather than alignment then rotation.  Basically, the text
is centered at your x,y location, rotated around this point, and then
aligned according to the bounding box of the rotated text.

So if you specify left, bottom alignment, the bottom left of the
bounding box of the rotated text will be at the x,y coordinate of the
text.

But a picture is worth a thousand words!
"""

import matplotlib.pyplot as plt
import numpy as np


def addtext(props):
    plt.text(0.5, 0.5, 'text 0', props, rotation=0)
```

```
    plt.text(1.5, 0.5, 'text 45', props, rotation=45)
    plt.text(2.5, 0.5, 'text 135', props, rotation=135)
    plt.text(3.5, 0.5, 'text 225', props, rotation=225)
    plt.text(4.5, 0.5, 'text -45', props, rotation=-45)
    plt.yticks([0, .5, 1])
    plt.grid(True)

# the text bounding box
bbox = {'fc': '0.8', 'pad': 0}

plt.subplot(211)
addtext({'ha': 'center', 'va': 'center', 'bbox': bbox})
plt.xlim(0, 5)
plt.xticks(np.arange(0, 5.1, 0.5), [])
plt.ylabel('center / center')

plt.subplot(212)
addtext({'ha': 'left', 'va': 'bottom', 'bbox': bbox})
plt.xlim(0, 5)
plt.xticks(np.arange(0, 5.1, 0.5))
plt.ylabel('left / bottom')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.198 pylab_examples example code: text_rotation_relative_to_line.py

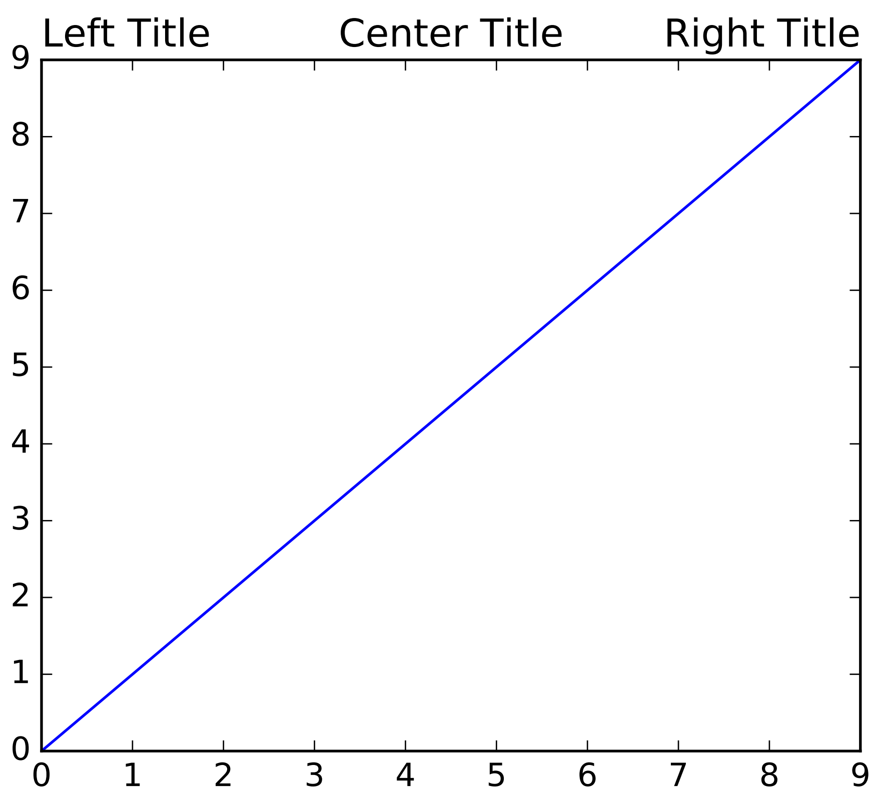



```
#!/usr/bin/env python
"""
Text objects in matplotlib are normally rotated with respect to the
screen coordinate system (i.e., 45 degrees rotation plots text along a
line that is in between horizontal and vertical no matter how the axes
are changed).  However, at times one wants to rotate text with respect
to something on the plot.  In this case, the correct angle won't be
the angle of that object in the plot coordinate system, but the angle
that that object APPEARS in the screen coordinate system.  This angle
is found by transforming the angle from the plot to the screen
coordinate system, as shown in the example below.
"""

import matplotlib.pyplot as plt
import numpy as np

# Plot diagonal line (45 degrees)
h = plt.plot(np.arange(0, 10), np.arange(0, 10))

# set limits so that it no longer looks on screen to be 45 degrees
plt.xlim([-10, 20])
```

```
# Locations to plot text
l1 = np.array((1, 1))
l2 = np.array((5, 5))

# Rotate angle
angle = 45
trans_angle = plt.gca().transData.transform_angles(np.array((45,)),
                                                   l2.reshape((1, 2)))[0]

# Plot text
th1 = plt.text(l1[0], l1[1], 'text not rotated correctly', fontsize=16,
           rotation=angle)
th2 = plt.text(l2[0], l2[1], 'text not rotated correctly', fontsize=16,
           rotation=trans_angle)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

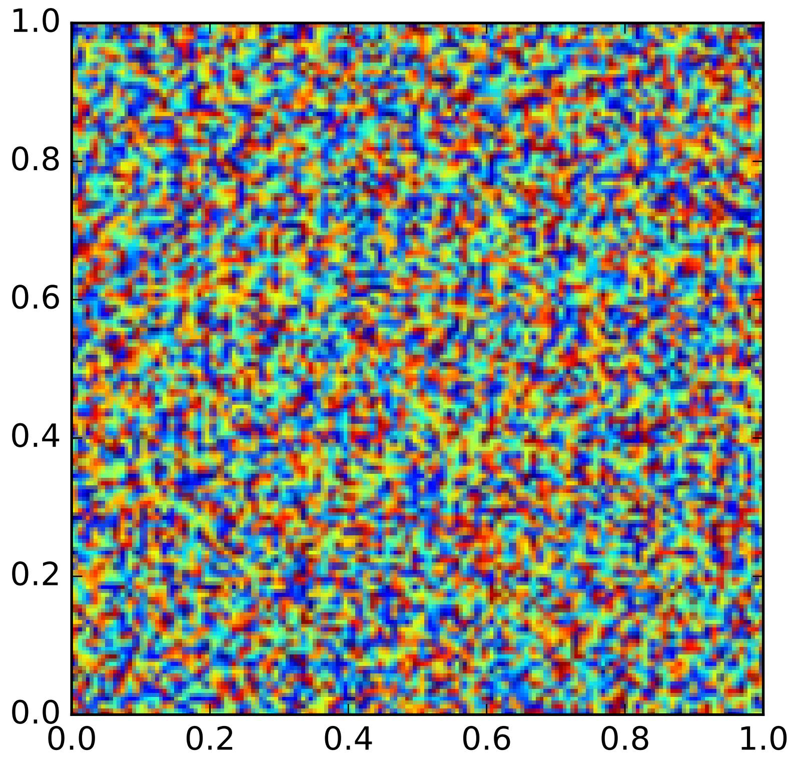
90.199 pylab_examples example code: titles_demo.py



```
#!/usr/bin/env python
"""
matplotlib can display plot titles centered, flush with the left side of
a set of axes, and flush with the right side of a set of axes.

"""
import matplotlib.pyplot as plt

plt.plot(range(10))

plt.title('Center Title')
plt.title('Left Title', loc='left')
plt.title('Right Title', loc='right')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.200 pylab_examples example code: toggle_images.py

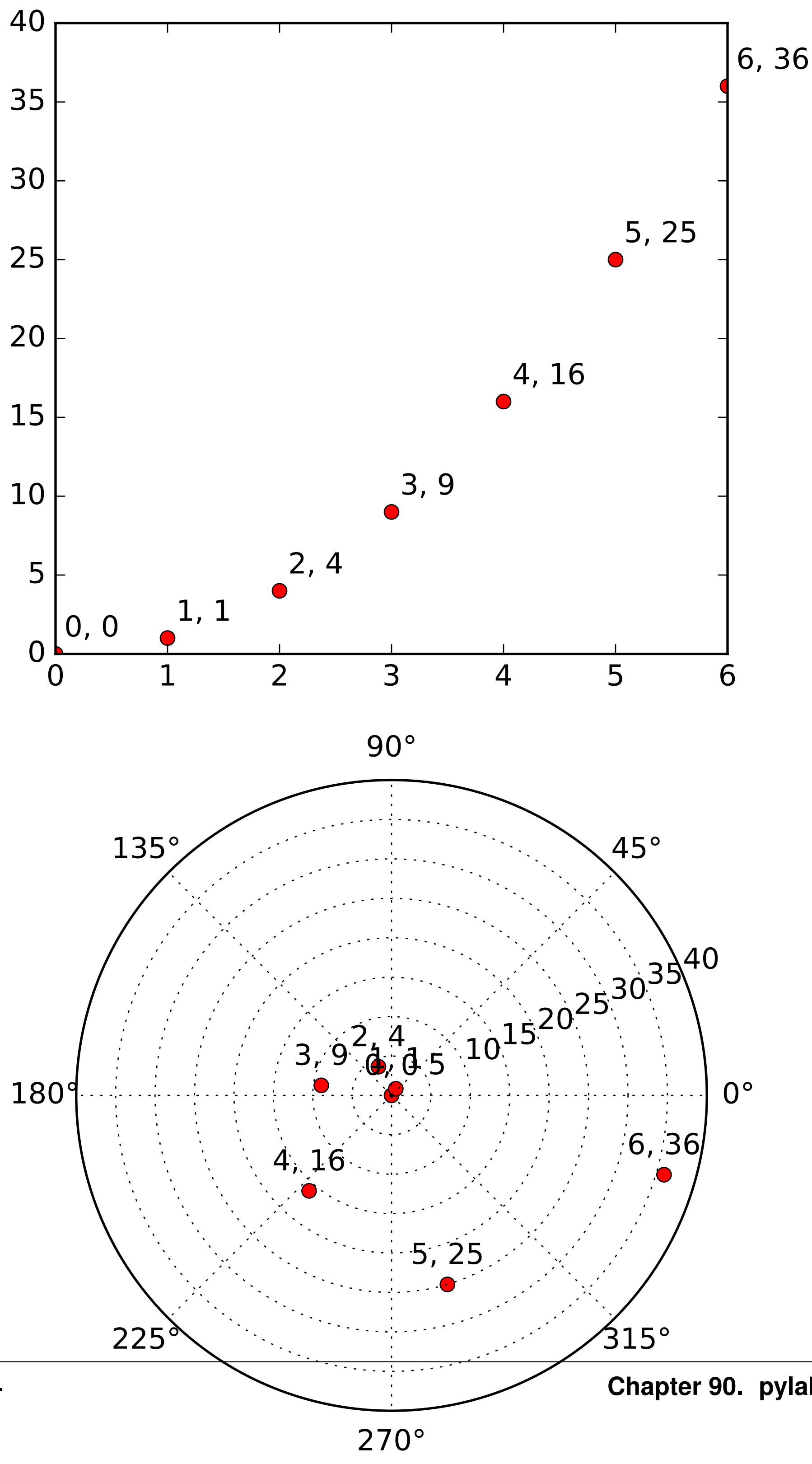



```
#!/usr/bin/env python
""" toggle between two images by pressing "t"
```

```
The basic idea is to load two images (they can be different shapes) and plot
them to the same axes with hold "on".  Then, toggle the visible property of
them using keypress event handling

If you want two images with different shapes to be plotted with the same
extent, they must have the same "extent" property

As usual, we'll define some random images for demo.  Real data is much more
exciting!

Note, on the wx backend on some platforms (e.g., linux), you have to
first click on the figure before the keypress events are activated.
If you know how to fix this, please email us!

"""

import matplotlib.pyplot as plt
import numpy as np

# two images x1 is initially visible, x2 is not
x1 = np.random.random((100, 100))
x2 = np.random.random((150, 175))

# arbitrary extent - both images must have same extent if you want
# them to be resampled into the same axes space
extent = (0, 1, 0, 1)
im1 = plt.imshow(x1, extent=extent)
im2 = plt.imshow(x2, extent=extent, hold=True)
im2.set_visible(False)


def toggle_images(event):
    'toggle the visible state of the two images'
    if event.key != 't':
        return
    b1 = im1.get_visible()
    b2 = im2.get_visible()
    im1.set_visible(not b1)
    im2.set_visible(not b2)
    plt.draw()

plt.connect('key_press_event', toggle_images)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.201 pylab_examples example code: transoffset.py

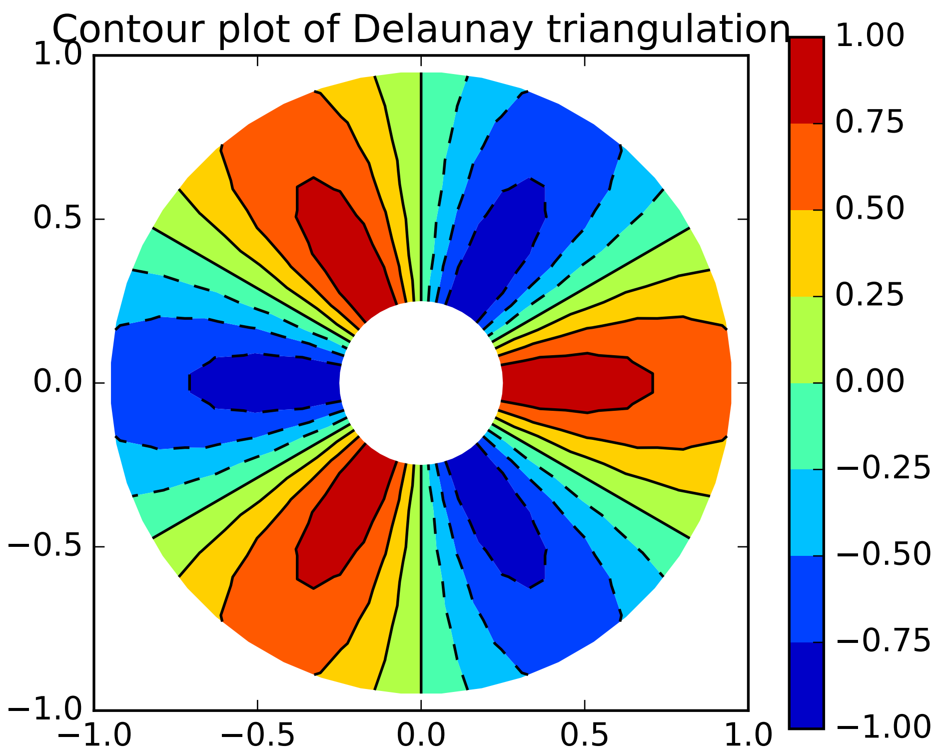


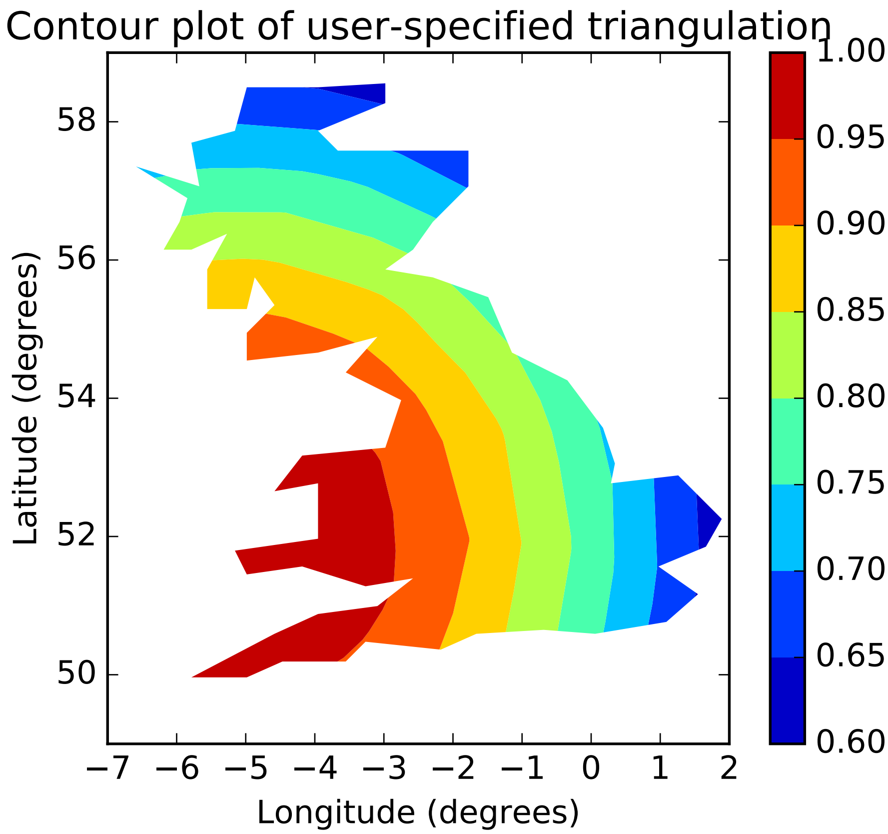
```
'''
This illustrates the use of transforms.offset_copy to
make a transform that positions a drawing element such as
a text string at a specified offset in screen coordinates
(dots or inches) relative to a location given in any
coordinates.

Every Artist--the mpl class from which classes such as
Text and Line are derived--has a transform that can be
set when the Artist is created, such as by the corresponding
pyplot command.  By default this is usually the Axes.transData
transform, going from data units to screen dots.  We can
use the offset_copy function to make a modified copy of
this transform, where the modification consists of an
offset.
'''

import matplotlib.pyplot as plt
import matplotlib.transforms as mtrans
import numpy as np

from matplotlib.transforms import offset_copy

xs = np.arange(7)
ys = xs**2

fig = plt.figure(figsize=(5, 10))
ax = plt.subplot(2, 1, 1)

# If we want the same offset for each text instance,
# we only need to make one transform.  To get the
# transform argument to offset_copy, we need to make the axes
# first; the subplot command above is one way to do this.
trans_offset = mtrans.offset_copy(ax.transData, fig=fig,
                                  x=0.05, y=0.10, units='inches')

for x, y in zip(xs, ys):
    plt.plot((x,), (y,), 'ro')
    plt.text(x, y, '%d, %d' % (int(x), int(y)), transform=trans_offset)


# offset_copy works for polar plots also.
ax = plt.subplot(2, 1, 2, projection='polar')

trans_offset = mtrans.offset_copy(ax.transData, fig=fig, y=6, units='dots')

for x, y in zip(xs, ys):
    plt.polar((x,), (y,), 'ro')
    plt.text(x, y, '%d, %d' % (int(x), int(y)),
             transform=trans_offset,
             horizontalalignment='center',
             verticalalignment='bottom')
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.202 pylab_examples example code: tricontour_demo.py





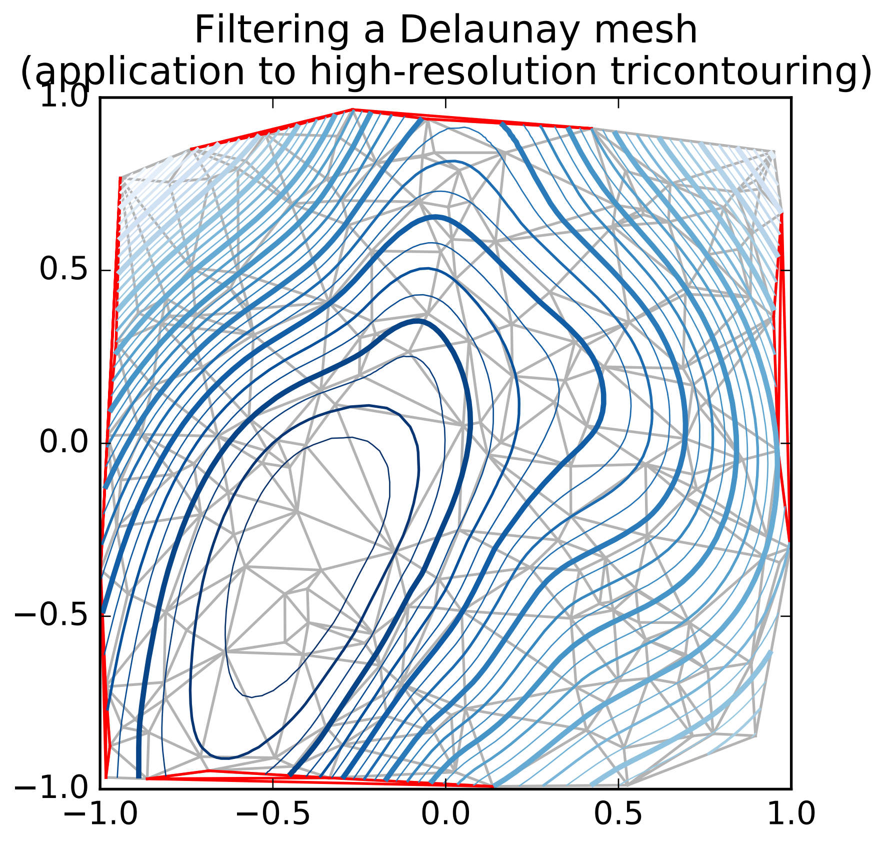

```
"""
Contour plots of unstructured triangular grids.
"""
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
import math

# Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.

# First create the x and y coordinates of the points.
n_angles = 48
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
```

```
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# pcolor plot.
plt.figure()
plt.gca().set_aspect('equal')
plt.tricontourf(triang, z)
plt.colorbar()
plt.tricontour(triang, z, colors='k')
plt.title('Contour plot of Delaunay triangulation')

# You can specify your own triangulation rather than perform a Delaunay
# triangulation of the points, where each triangle is given by the indices of
# the three points that make up the triangle, ordered in either a clockwise or
# anticlockwise manner.

xy = np.asarray([
    [-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
    [-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
    [-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.080, 0.919],
    [-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
    [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
    [-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
    [-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
    [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
    [-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
    [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
    [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
    [-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
    [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
    [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
    [ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
    [ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
    [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
    [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
    [-0.077, 0.990], [-0.059, 0.993]])
x = np.degrees(xy[:, 0])
y = np.degrees(xy[:, 1])
x0 = -5
y0 = 52
z = np.exp(-0.01*((x - x0)*(x - x0) + (y - y0)*(y - y0)))

triangles = np.asarray([
    [67, 66,  1], [65,  2, 66], [ 1, 66,  2], [64,  2, 65], [63,  3, 64],
    [60, 59, 57], [ 2, 64,  3], [ 3, 63,  4], [ 0, 67,  1], [62,  4, 63],
    [57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [ 4, 62, 68],
    [ 6,  5,  9], [61, 68, 62], [69, 68, 61], [ 9,  5, 70], [ 6,  8,  7],
    [ 4, 70,  5], [ 8,  6,  9], [56, 69, 57], [69, 56, 52], [70, 10,  9],
```

```
    [54, 53, 55], [56, 55, 53], [68, 70,  4], [52, 56, 53], [11, 10, 12],
    [69, 71, 68], [68, 13, 70], [10, 70, 13], [51, 50, 52], [13, 68, 71],
    [52, 71, 69], [12, 10, 13], [71, 52, 50], [71, 14, 13], [50, 49, 71],
    [49, 48, 71], [14, 16, 15], [14, 71, 48], [17, 19, 18], [17, 20, 19],
    [48, 16, 14], [48, 47, 16], [47, 46, 16], [16, 46, 45], [23, 22, 24],
    [21, 24, 22], [17, 16, 45], [20, 17, 45], [21, 25, 24], [27, 26, 28],
    [20, 72, 21], [25, 21, 72], [45, 72, 20], [25, 28, 26], [44, 73, 45],
    [72, 45, 73], [28, 25, 29], [29, 25, 31], [43, 73, 44], [73, 43, 40],
    [72, 73, 39], [72, 31, 25], [42, 40, 43], [31, 30, 29], [39, 73, 40],
    [42, 41, 40], [72, 33, 31], [32, 31, 33], [39, 38, 72], [33, 72, 38],
    [33, 38, 34], [37, 35, 38], [34, 38, 35], [35, 37, 36]])

# Rather than create a Triangulation object, can simply pass x, y and triangles
# arrays to tripcolor directly.  It would be better to use a Triangulation
# object if the same triangulation was to be used more than once to save
# duplicated calculations.
plt.figure()
plt.gca().set_aspect('equal')
plt.tricontourf(x, y, triangles, z)
plt.colorbar()
plt.title('Contour plot of user-specified triangulation')
plt.xlabel('Longitude (degrees)')
plt.ylabel('Latitude (degrees)')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.203 pylab_examples example code: tricontour_smooth_delaunay.py




```
"""
Demonstrates high-resolution tricontouring of a random set of points ;
a matplotlib.tri.TriAnalyzer is used to improve the plot quality.

The initial data points and triangular grid for this demo are:
    - a set of random points is instantiated, inside [-1, 1] x [-1, 1] square
    - A Delaunay triangulation of these points is then computed, of which a
    random subset of triangles is masked out by the user (based on
    *init_mask_frac* parameter). This simulates invalidated data.

The proposed generic procedure to obtain a high resolution contouring of such
a data set is the following:
    1) Compute an extended mask with a matplotlib.tri.TriAnalyzer, which will
    exclude badly shaped (flat) triangles from the border of the
    triangulation. Apply the mask to the triangulation (using set_mask).
    2) Refine and interpolate the data using a
    matplotlib.tri.UniformTriRefiner.
    3) Plot the refined data with tricontour.

"""
from matplotlib.tri import Triangulation, TriAnalyzer, UniformTriRefiner
import matplotlib.pyplot as plt
```

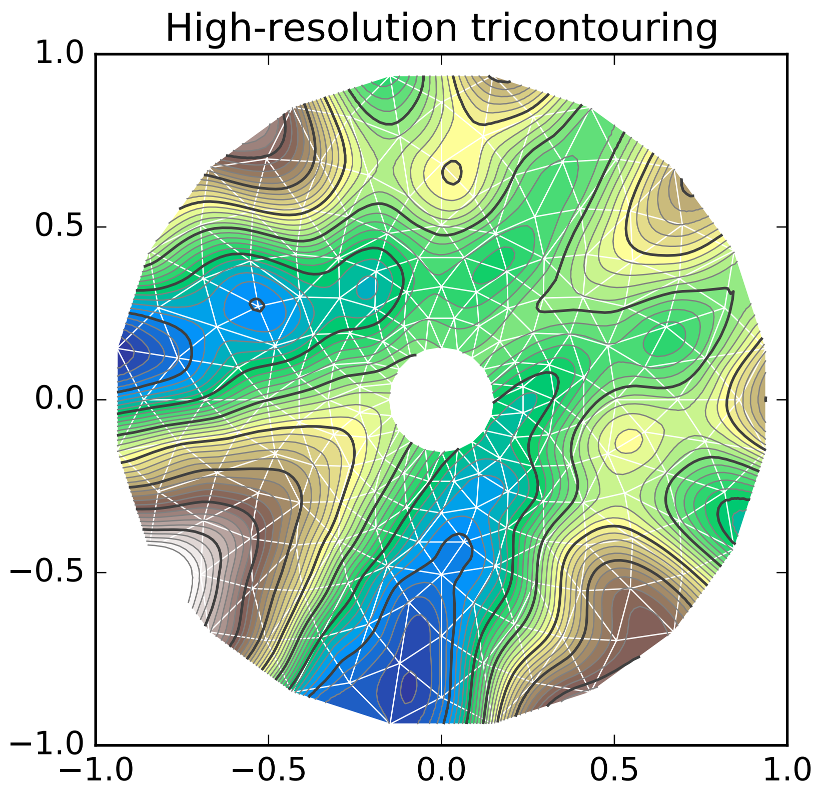
```
import matplotlib.cm as cm
import numpy as np


#-----------------------------------------------------------------------------
# Analytical test function
#-----------------------------------------------------------------------------
def experiment_res(x, y):
    """ An analytic function representing experiment results """
    x = 2.*x
    r1 = np.sqrt((0.5 - x)**2 + (0.5 - y)**2)
    theta1 = np.arctan2(0.5 - x, 0.5 - y)
    r2 = np.sqrt((-x - 0.2)**2 + (-y - 0.2)**2)
    theta2 = np.arctan2(-x - 0.2, -y - 0.2)
    z = (4*(np.exp((r1/10)**2) - 1)*30. * np.cos(3*theta1) +
         (np.exp((r2/10)**2) - 1)*30. * np.cos(5*theta2) +
         2*(x**2 + y**2))
    return (np.max(z) - z)/(np.max(z) - np.min(z))

#-----------------------------------------------------------------------------
# Generating the initial data test points and triangulation for the demo
#-----------------------------------------------------------------------------
# User parameters for data test points
n_test = 200  # Number of test data points, tested from 3 to 5000 for subdiv=3

subdiv = 3  # Number of recursive subdivisions of the initial mesh for smooth
            # plots. Values >3 might result in a very high number of triangles
            # for the refine mesh: new triangles numbering = (4**subdiv)*ntri

init_mask_frac = 0.0    # Float > 0. adjusting the proportion of
                        # (invalid) initial triangles which will be masked
                        # out. Enter 0 for no mask.

min_circle_ratio = .01  # Minimum circle ratio - border triangles with circle
                        # ratio below this will be masked if they touch a
                        # border. Suggested value 0.01 ; Use -1 to keep
                        # all triangles.

# Random points
random_gen = np.random.mtrand.RandomState(seed=127260)
x_test = random_gen.uniform(-1., 1., size=n_test)
y_test = random_gen.uniform(-1., 1., size=n_test)
z_test = experiment_res(x_test, y_test)

# meshing with Delaunay triangulation
tri = Triangulation(x_test, y_test)
ntri = tri.triangles.shape[0]

# Some invalid data are masked out
mask_init = np.zeros(ntri, dtype=np.bool)
masked_tri = random_gen.randint(0, ntri, int(ntri*init_mask_frac))
mask_init[masked_tri] = True
tri.set_mask(mask_init)
```

```
#-----------------------------------------------------------------------------
# Improving the triangulation before high-res plots: removing flat triangles
#-----------------------------------------------------------------------------
# masking badly shaped triangles at the border of the triangular mesh.
mask = TriAnalyzer(tri).get_flat_tri_mask(min_circle_ratio)
tri.set_mask(mask)

# refining the data
refiner = UniformTriRefiner(tri)
tri_refi, z_test_refi = refiner.refine_field(z_test, subdiv=subdiv)

# analytical 'results' for comparison
z_expected = experiment_res(tri_refi.x, tri_refi.y)

# for the demo: loading the 'flat' triangles for plot
flat_tri = Triangulation(x_test, y_test)
flat_tri.set_mask(~mask)


#-----------------------------------------------------------------------------
# Now the plots
#-----------------------------------------------------------------------------
# User options for plots
plot_tri = True          # plot of base triangulation
plot_masked_tri = True   # plot of excessively flat excluded triangles
plot_refi_tri = False    # plot of refined triangulation
plot_expected = False    # plot of analytical function values for comparison


# Graphical options for tricontouring
levels = np.arange(0., 1., 0.025)
cmap = cm.get_cmap(name='Blues', lut=None)

plt.figure()
plt.gca().set_aspect('equal')
plt.title("Filtering a Delaunay mesh\n" +
          "(application to high-resolution tricontouring)")

# 1) plot of the refined (computed) data countours:
plt.tricontour(tri_refi, z_test_refi, levels=levels, cmap=cmap,
               linewidths=[2.0, 0.5, 1.0, 0.5])
# 2) plot of the expected (analytical) data countours (dashed):
if plot_expected:
    plt.tricontour(tri_refi, z_expected, levels=levels, cmap=cmap,
                   linestyles='--')
# 3) plot of the fine mesh on which interpolation was done:
if plot_refi_tri:
    plt.triplot(tri_refi, color='0.97')
# 4) plot of the initial 'coarse' mesh:
if plot_tri:
    plt.triplot(tri, color='0.7')
```

```
# 4) plot of the unvalidated triangles from naive Delaunay Triangulation:
if plot_masked_tri:
    plt.triplot(flat_tri, color='red')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.204 pylab_examples example code: tricontour_smooth_user.py



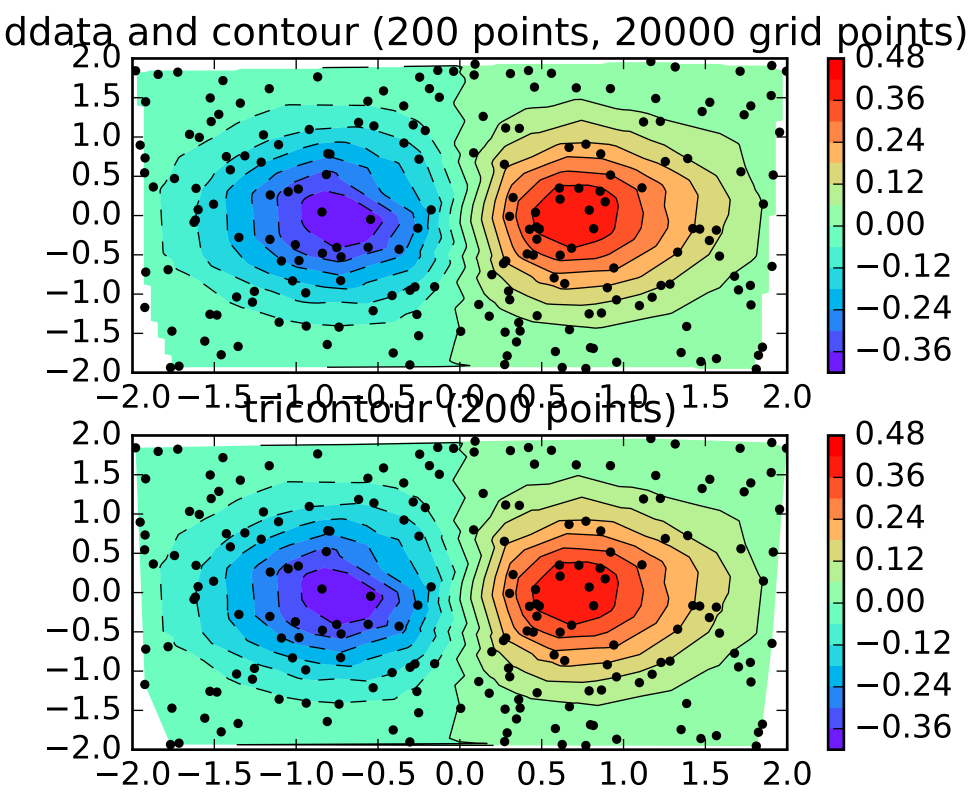

```
"""
Demonstrates high-resolution tricontouring on user-defined triangular grids
with matplotlib.tri.UniformTriRefiner
"""
import matplotlib.tri as tri
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import math


#-----------------------------------------------------------------------------
```

```
# Analytical test function
#-----------------------------------------------------------------------------
def function_z(x, y):
    """ A function of 2 variables """
    r1 = np.sqrt((0.5 - x)**2 + (0.5 - y)**2)
    theta1 = np.arctan2(0.5 - x, 0.5 - y)
    r2 = np.sqrt((-x - 0.2)**2 + (-y - 0.2)**2)
    theta2 = np.arctan2(-x - 0.2, -y - 0.2)
    z = -(2*(np.exp((r1/10)**2) - 1)*30. * np.cos(7.*theta1) +
          (np.exp((r2/10)**2) - 1)*30. * np.cos(11.*theta2) +
          0.7*(x**2 + y**2))
    return (np.max(z) - z)/(np.max(z) - np.min(z))

#-----------------------------------------------------------------------------
# Creating a Triangulation
#-----------------------------------------------------------------------------
# First create the x and y coordinates of the points.
n_angles = 20
n_radii = 10
min_radius = 0.15
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = function_z(x, y)

# Now create the Triangulation.
# (Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.)
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

#-----------------------------------------------------------------------------
# Refine data
#-----------------------------------------------------------------------------
refiner = tri.UniformTriRefiner(triang)
tri_refi, z_test_refi = refiner.refine_field(z, subdiv=3)

#-----------------------------------------------------------------------------
# Plot the triangulation and the high-res iso-contours
#-----------------------------------------------------------------------------
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(triang, lw=0.5, color='white')
```

```
levels = np.arange(0., 1., 0.025)
cmap = cm.get_cmap(name='terrain', lut=None)
plt.tricontourf(tri_refi, z_test_refi, levels=levels, cmap=cmap)
plt.tricontour(tri_refi, z_test_refi, levels=levels,
               colors=['0.25', '0.5', '0.5', '0.5', '0.5'],
               linewidths=[1.0, 0.5, 0.5, 0.5, 0.5])

plt.title("High-resolution tricontouring")

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.205 pylab_examples example code: tricontour_vs_griddata.py

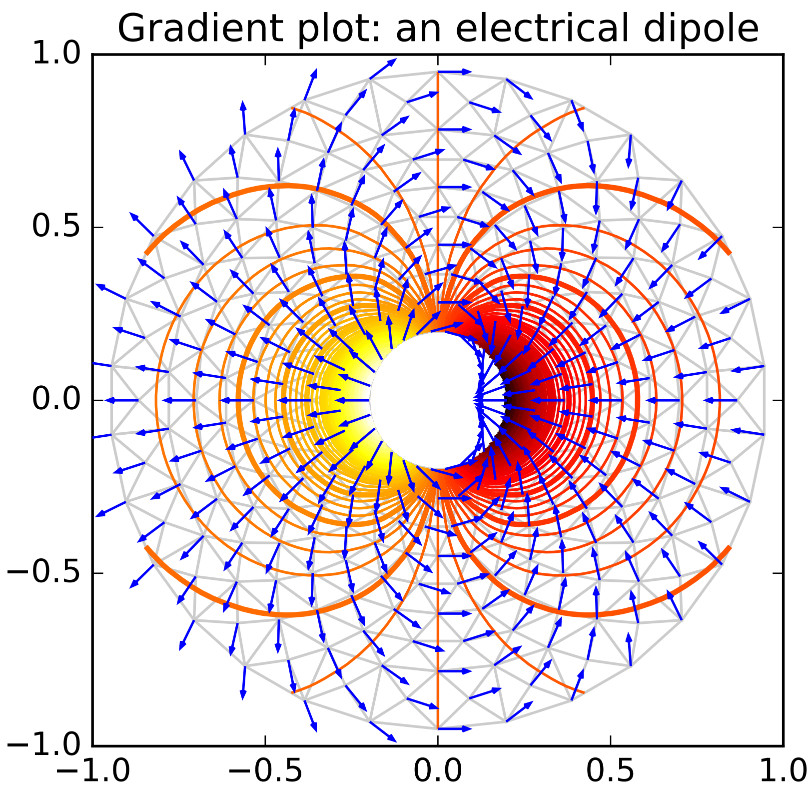



```
"""
Comparison of griddata and tricontour for an unstructured triangular grid.
"""
from __future__ import print_function
import matplotlib.pyplot as plt
import matplotlib.tri as tri
```

```
import numpy as np
import numpy.random as rnd
import matplotlib.mlab as mlab
import time

rnd.seed(0)
npts = 200
ngridx = 100
ngridy = 200
x = rnd.uniform(-2, 2, npts)
y = rnd.uniform(-2, 2, npts)
z = x*np.exp(-x**2 - y**2)

# griddata and contour.
start = time.clock()
plt.subplot(211)
xi = np.linspace(-2.1, 2.1, ngridx)
yi = np.linspace(-2.1, 2.1, ngridy)
zi = mlab.griddata(x, y, z, xi, yi, interp='linear')
plt.contour(xi, yi, zi, 15, linewidths=0.5, colors='k')
plt.contourf(xi, yi, zi, 15, cmap=plt.cm.rainbow,
             norm=plt.Normalize(vmax=abs(zi).max(), vmin=-abs(zi).max()))
plt.colorbar()  # draw colorbar
plt.plot(x, y, 'ko', ms=3)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('griddata and contour (%d points, %d grid points)' %
          (npts, ngridx*ngridy))
print('griddata and contour seconds: %f' % (time.clock() - start))

# tricontour.
start = time.clock()
plt.subplot(212)
triang = tri.Triangulation(x, y)
plt.tricontour(x, y, z, 15, linewidths=0.5, colors='k')
plt.tricontourf(x, y, z, 15, cmap=plt.cm.rainbow,
                norm=plt.Normalize(vmax=abs(zi).max(), vmin=-abs(zi).max()))
plt.colorbar()
plt.plot(x, y, 'ko', ms=3)
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('tricontour (%d points)' % npts)
print('tricontour seconds: %f' % (time.clock() - start))

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.206 pylab_examples example code: trigradient_demo.py



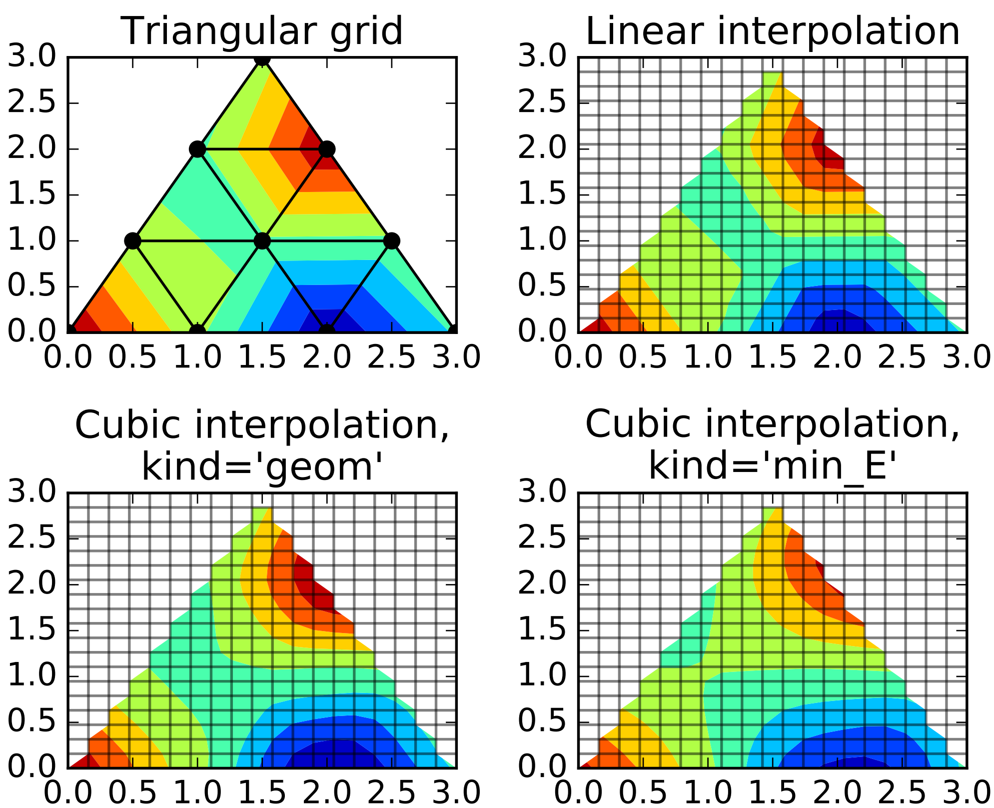

```
"""
Demonstrates computation of gradient with matplotlib.tri.CubicTriInterpolator.
"""
from matplotlib.tri import Triangulation, UniformTriRefiner,\
    CubicTriInterpolator
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import math


#-----------------------------------------------------------------------------
# Electrical potential of a dipole
#-----------------------------------------------------------------------------
def dipole_potential(x, y):
    """ The electric dipole potential V """
    r_sq = x**2 + y**2
    theta = np.arctan2(y, x)
    z = np.cos(theta)/r_sq
    return (np.max(z) - z) / (np.max(z) - np.min(z))
```

```
#-----------------------------------------------------------------------------
# Creating a Triangulation
#-----------------------------------------------------------------------------
# First create the x and y coordinates of the points.
n_angles = 30
n_radii = 10
min_radius = 0.2
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
V = dipole_potential(x, y)

# Create the Triangulation; no triangles specified so Delaunay triangulation
# created.
triang = Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

#-----------------------------------------------------------------------------
# Refine data - interpolates the electrical potential V
#-----------------------------------------------------------------------------
refiner = UniformTriRefiner(triang)
tri_refi, z_test_refi = refiner.refine_field(V, subdiv=3)

#-----------------------------------------------------------------------------
# Computes the electrical field (Ex, Ey) as gradient of electrical potential
#-----------------------------------------------------------------------------
tci = CubicTriInterpolator(triang, -V)
# Gradient requested here at the mesh nodes but could be anywhere else:
(Ex, Ey) = tci.gradient(triang.x, triang.y)
E_norm = np.sqrt(Ex**2 + Ey**2)

#-----------------------------------------------------------------------------
# Plot the triangulation, the potential iso-contours and the vector field
#-----------------------------------------------------------------------------
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(triang, color='0.8')

levels = np.arange(0., 1., 0.01)
cmap = cm.get_cmap(name='hot', lut=None)
plt.tricontour(tri_refi, z_test_refi, levels=levels, cmap=cmap,
               linewidths=[2.0, 1.0, 1.0, 1.0])
# Plots direction of the electrical vector field
```

```
plt.quiver(triang.x, triang.y, Ex/E_norm, Ey/E_norm,
           units='xy', scale=10., zorder=3, color='blue',
           width=0.007, headwidth=3., headlength=4.)

plt.title('Gradient plot: an electrical dipole')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.207 pylab_examples example code: triinterp_demo.py

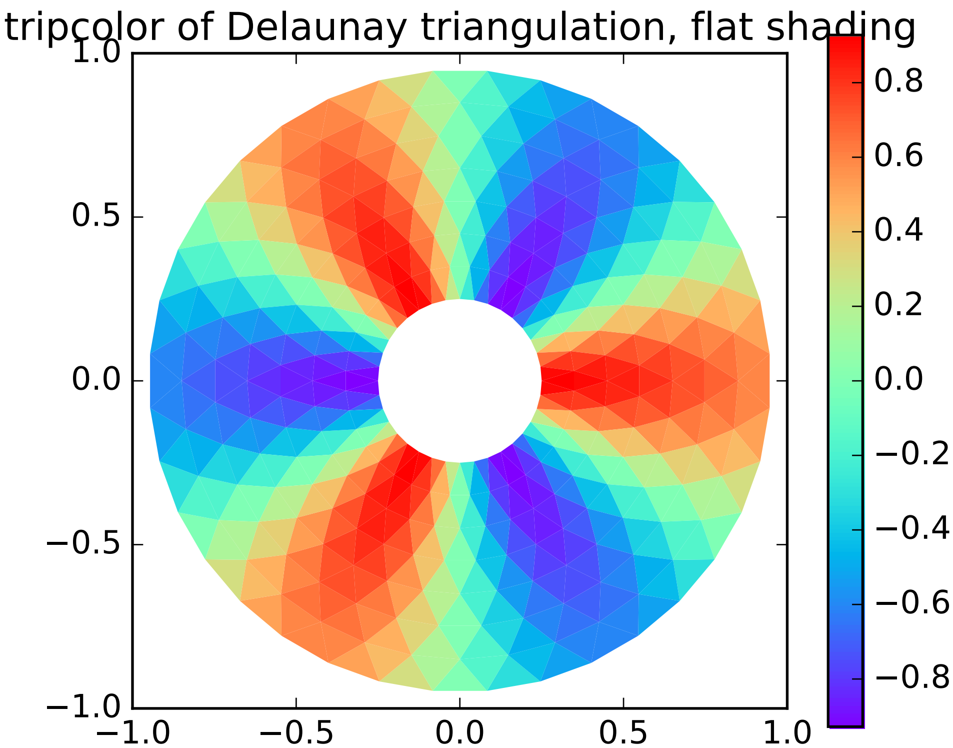


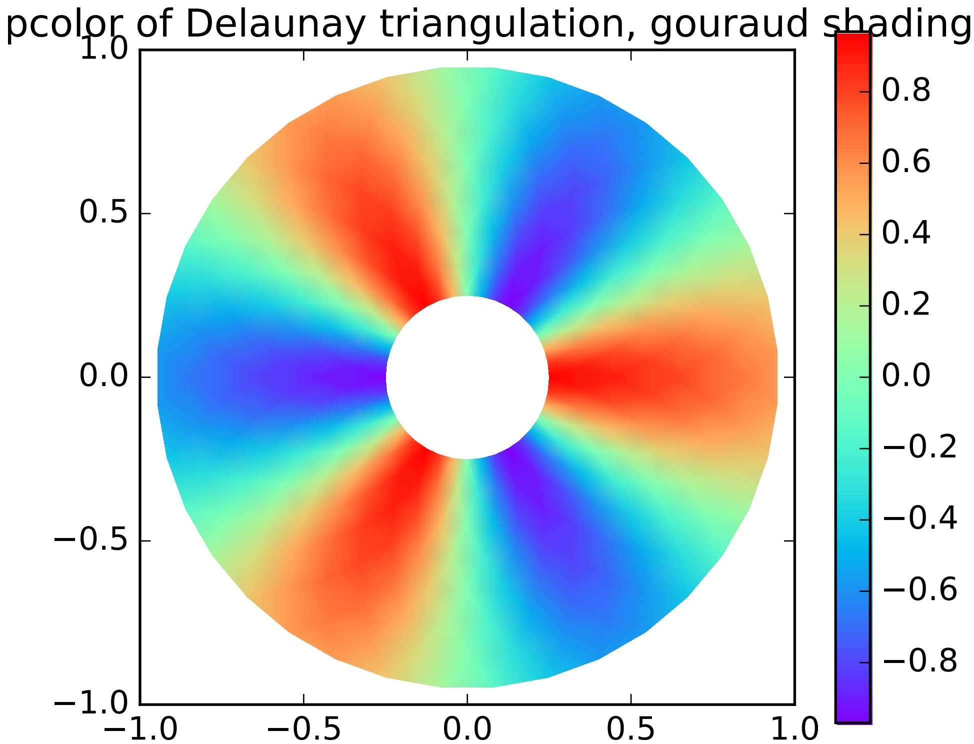

```
"""
Interpolation from triangular grid to quad grid.
"""
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
import numpy as np

# Create triangulation.
x = np.asarray([0, 1, 2, 3, 0.5, 1.5, 2.5, 1, 2, 1.5])
y = np.asarray([0, 0, 0, 0, 1.0, 1.0, 1.0, 2, 2, 3.0])
triangles = [[0, 1, 4], [1, 2, 5], [2, 3, 6], [1, 5, 4], [2, 6, 5], [4, 5, 7],
```

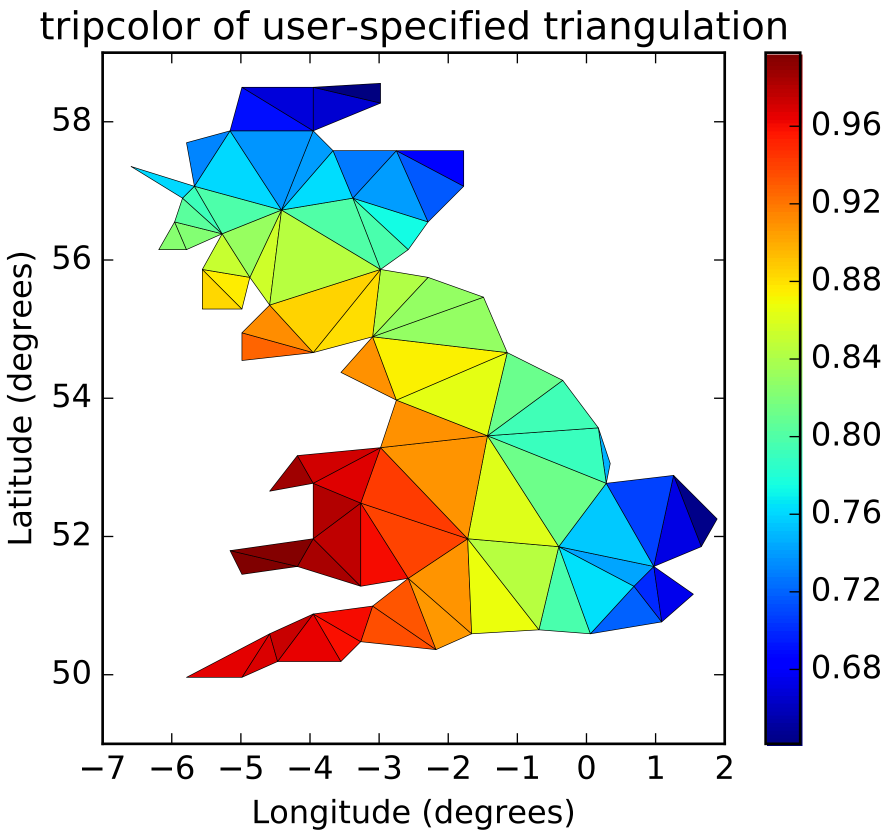
```
              [5, 6, 8], [5, 8, 7], [7, 8, 9]]
triang = mtri.Triangulation(x, y, triangles)

# Interpolate to regularly-spaced quad grid.
z = np.cos(1.5*x)*np.cos(1.5*y)
xi, yi = np.meshgrid(np.linspace(0, 3, 20), np.linspace(0, 3, 20))

interp_lin = mtri.LinearTriInterpolator(triang, z)
zi_lin = interp_lin(xi, yi)

interp_cubic_geom = mtri.CubicTriInterpolator(triang, z, kind='geom')
zi_cubic_geom = interp_cubic_geom(xi, yi)

interp_cubic_min_E = mtri.CubicTriInterpolator(triang, z, kind='min_E')
zi_cubic_min_E = interp_cubic_min_E(xi, yi)


# Plot the triangulation.
plt.subplot(221)
plt.tricontourf(triang, z)
plt.triplot(triang, 'ko-')
plt.title('Triangular grid')

# Plot linear interpolation to quad grid.
plt.subplot(222)
plt.contourf(xi, yi, zi_lin)
plt.plot(xi, yi, 'k-', alpha=0.5)
plt.plot(xi.T, yi.T, 'k-', alpha=0.5)
plt.title("Linear interpolation")

# Plot cubic interpolation to quad grid, kind=geom
plt.subplot(223)
plt.contourf(xi, yi, zi_cubic_geom)
plt.plot(xi, yi, 'k-', alpha=0.5)
plt.plot(xi.T, yi.T, 'k-', alpha=0.5)
plt.title("Cubic interpolation,\nkind='geom'")

# Plot cubic interpolation to quad grid, kind=min_E
plt.subplot(224)
plt.contourf(xi, yi, zi_cubic_min_E)
plt.plot(xi, yi, 'k-', alpha=0.5)
plt.plot(xi.T, yi.T, 'k-', alpha=0.5)
plt.title("Cubic interpolation,\nkind='min_E'")

plt.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.208 pylab_examples example code: tripcolor_demo.py




pcolor of Delaunay triangulation, gouraud shading
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8



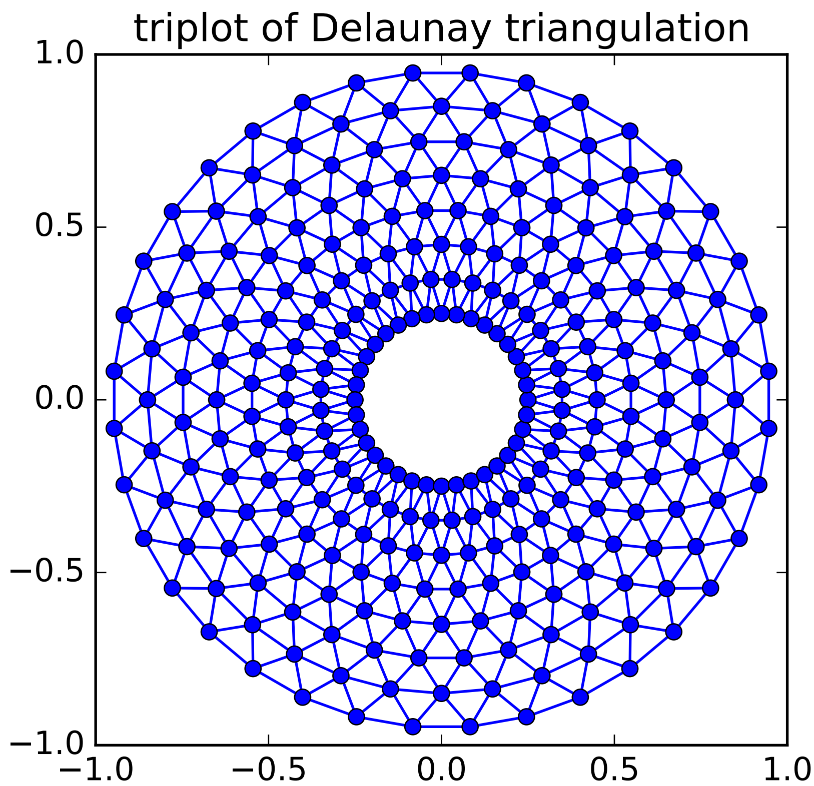

```
"""
Pseudocolor plots of unstructured triangular grids.
"""
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
import math

# Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.

# First create the x and y coordinates of the points.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
```

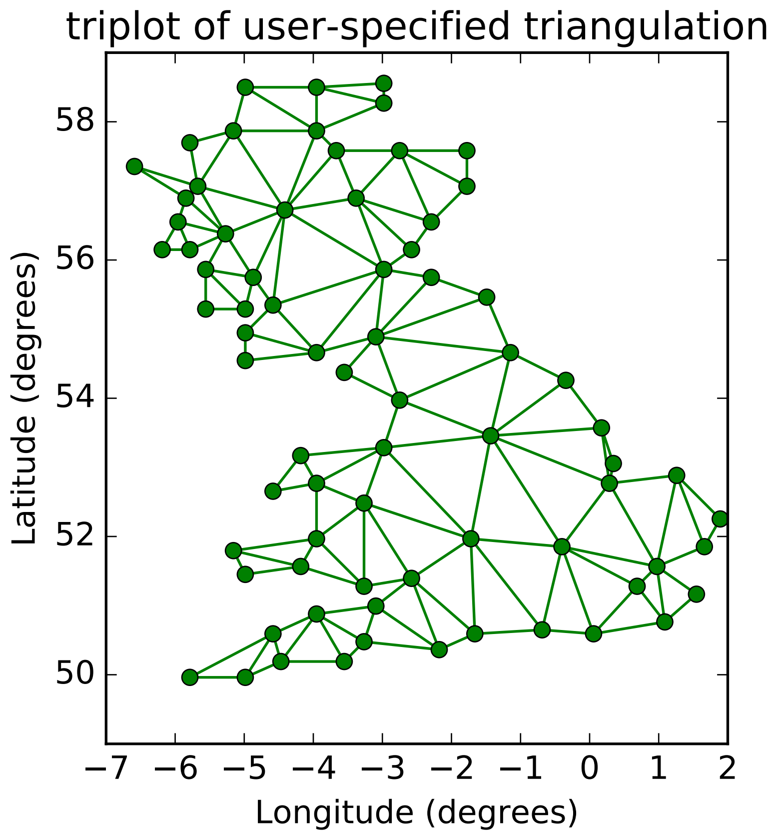
```
triang = tri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# tripcolor plot.
plt.figure()
plt.gca().set_aspect('equal')
plt.tripcolor(triang, z, shading='flat', cmap=plt.cm.rainbow)
plt.colorbar()
plt.title('tripcolor of Delaunay triangulation, flat shading')

# Illustrate Gouraud shading.
plt.figure()
plt.gca().set_aspect('equal')
plt.tripcolor(triang, z, shading='gouraud', cmap=plt.cm.rainbow)
plt.colorbar()
plt.title('tripcolor of Delaunay triangulation, gouraud shading')


# You can specify your own triangulation rather than perform a Delaunay
# triangulation of the points, where each triangle is given by the indices of
# the three points that make up the triangle, ordered in either a clockwise or
# anticlockwise manner.

xy = np.asarray([
    [-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
    [-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
    [-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.080, 0.919],
    [-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
    [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
    [-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
    [-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
    [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
    [-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
    [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
    [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
    [-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
    [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
    [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
    [ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
    [ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
    [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
    [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
    [-0.077, 0.990], [-0.059, 0.993]])
x = xy[:, 0]*180/3.14159
y = xy[:, 1]*180/3.14159

triangles = np.asarray([
    [67, 66,  1], [65,  2, 66], [ 1, 66,  2], [64,  2, 65], [63,  3, 64],
```

```
    [60, 59, 57], [ 2, 64,  3], [ 3, 63,  4], [ 0, 67,  1], [62,  4, 63],
    [57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [ 4, 62, 68],
    [ 6,  5,  9], [61, 68, 62], [69, 68, 61], [ 9,  5, 70], [ 6,  8,  7],
    [ 4, 70,  5], [ 8,  6,  9], [56, 69, 57], [69, 56, 52], [70, 10,  9],
    [54, 53, 55], [56, 55, 53], [68, 70,  4], [52, 56, 53], [11, 10, 12],
    [69, 71, 68], [68, 13, 70], [10, 70, 13], [51, 50, 52], [13, 68, 71],
    [52, 71, 69], [12, 10, 13], [71, 52, 50], [71, 14, 13], [50, 49, 71],
    [49, 48, 71], [14, 16, 15], [14, 71, 48], [17, 19, 18], [17, 20, 19],
    [48, 16, 14], [48, 47, 16], [47, 46, 16], [16, 46, 45], [23, 22, 24],
    [21, 24, 22], [17, 16, 45], [20, 17, 45], [21, 25, 24], [27, 26, 28],
    [20, 72, 21], [25, 21, 72], [45, 72, 20], [25, 28, 26], [44, 73, 45],
    [72, 45, 73], [28, 25, 29], [29, 25, 31], [43, 73, 44], [73, 43, 40],
    [72, 73, 39], [72, 31, 25], [42, 40, 43], [31, 30, 29], [39, 73, 40],
    [42, 41, 40], [72, 33, 31], [32, 31, 33], [39, 38, 72], [33, 72, 38],
    [33, 38, 34], [37, 35, 38], [34, 38, 35], [35, 37, 36]])

xmid = x[triangles].mean(axis=1)
ymid = y[triangles].mean(axis=1)
x0 = -5
y0 = 52
zfaces = np.exp(-0.01*((xmid - x0)*(xmid - x0) + (ymid - y0)*(ymid - y0)))

# Rather than create a Triangulation object, can simply pass x, y and triangles
# arrays to tripcolor directly.  It would be better to use a Triangulation
# object if the same triangulation was to be used more than once to save
# duplicated calculations.
# Can specify one color value per face rather than one per point by using the
# facecolors kwarg.
plt.figure()
plt.gca().set_aspect('equal')
plt.tripcolor(x, y, triangles, facecolors=zfaces, edgecolors='k')
plt.colorbar()
plt.title('tripcolor of user-specified triangulation')
plt.xlabel('Longitude (degrees)')
plt.ylabel('Latitude (degrees)')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.209 pylab_examples example code: triplot_demo.py





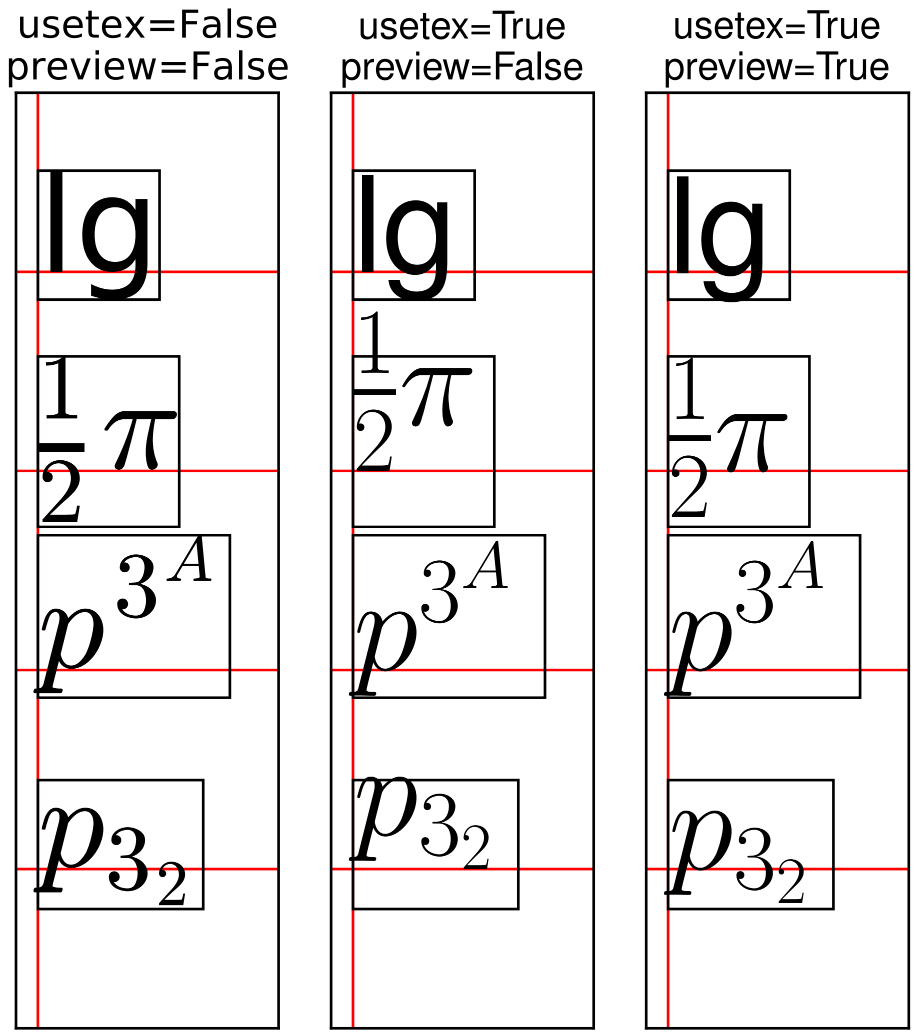

```
"""
Creating and plotting unstructured triangular grids.
"""
import matplotlib.pyplot as plt
import matplotlib.tri as tri
import numpy as np
import math

# Creating a Triangulation without specifying the triangles results in the
# Delaunay triangulation of the points.

# First create the x and y coordinates of the points.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*math.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += math.pi/n_angles

x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = tri.Triangulation(x, y)
```

```
# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid*xmid + ymid*ymid < min_radius*min_radius, 1, 0)
triang.set_mask(mask)

# Plot the triangulation.
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(triang, 'bo-')
plt.title('triplot of Delaunay triangulation')


# You can specify your own triangulation rather than perform a Delaunay
# triangulation of the points, where each triangle is given by the indices of
# the three points that make up the triangle, ordered in either a clockwise or
# anticlockwise manner.

xy = np.asarray([
    [-0.101, 0.872], [-0.080, 0.883], [-0.069, 0.888], [-0.054, 0.890],
    [-0.045, 0.897], [-0.057, 0.895], [-0.073, 0.900], [-0.087, 0.898],
    [-0.090, 0.904], [-0.069, 0.907], [-0.069, 0.921], [-0.080, 0.919],
    [-0.073, 0.928], [-0.052, 0.930], [-0.048, 0.942], [-0.062, 0.949],
    [-0.054, 0.958], [-0.069, 0.954], [-0.087, 0.952], [-0.087, 0.959],
    [-0.080, 0.966], [-0.085, 0.973], [-0.087, 0.965], [-0.097, 0.965],
    [-0.097, 0.975], [-0.092, 0.984], [-0.101, 0.980], [-0.108, 0.980],
    [-0.104, 0.987], [-0.102, 0.993], [-0.115, 1.001], [-0.099, 0.996],
    [-0.101, 1.007], [-0.090, 1.010], [-0.087, 1.021], [-0.069, 1.021],
    [-0.052, 1.022], [-0.052, 1.017], [-0.069, 1.010], [-0.064, 1.005],
    [-0.048, 1.005], [-0.031, 1.005], [-0.031, 0.996], [-0.040, 0.987],
    [-0.045, 0.980], [-0.052, 0.975], [-0.040, 0.973], [-0.026, 0.968],
    [-0.020, 0.954], [-0.006, 0.947], [ 0.003, 0.935], [ 0.006, 0.926],
    [ 0.005, 0.921], [ 0.022, 0.923], [ 0.033, 0.912], [ 0.029, 0.905],
    [ 0.017, 0.900], [ 0.012, 0.895], [ 0.027, 0.893], [ 0.019, 0.886],
    [ 0.001, 0.883], [-0.012, 0.884], [-0.029, 0.883], [-0.038, 0.879],
    [-0.057, 0.881], [-0.062, 0.876], [-0.078, 0.876], [-0.087, 0.872],
    [-0.030, 0.907], [-0.007, 0.905], [-0.057, 0.916], [-0.025, 0.933],
    [-0.077, 0.990], [-0.059, 0.993]])
x = np.degrees(xy[:, 0])
y = np.degrees(xy[:, 1])

triangles = np.asarray([
    [67, 66,  1], [65,  2, 66], [ 1, 66,  2], [64,  2, 65], [63,  3, 64],
    [60, 59, 57], [ 2, 64,  3], [ 3, 63,  4], [ 0, 67,  1], [62,  4, 63],
    [57, 59, 56], [59, 58, 56], [61, 60, 69], [57, 69, 60], [ 4, 62, 68],
    [ 6,  5,  9], [61, 68, 62], [69, 68, 61], [ 9,  5, 70], [ 6,  8,  7],
    [ 4, 70,  5], [ 8,  6,  9], [56, 69, 57], [69, 56, 52], [70, 10,  9],
    [54, 53, 55], [56, 55, 53], [68, 70,  4], [52, 56, 53], [11, 10, 12],
    [69, 71, 68], [68, 13, 70], [10, 70, 13], [51, 50, 52], [13, 68, 71],
    [52, 71, 69], [12, 10, 13], [71, 52, 50], [71, 14, 13], [50, 49, 71],
    [49, 48, 71], [14, 16, 15], [14, 71, 48], [17, 19, 18], [17, 20, 19],
    [48, 16, 14], [48, 47, 16], [47, 46, 16], [16, 46, 45], [23, 22, 24],
```

```
    [21, 24, 22], [17, 16, 45], [20, 17, 45], [21, 25, 24], [27, 26, 28],
    [20, 72, 21], [25, 21, 72], [45, 72, 20], [25, 28, 26], [44, 73, 45],
    [72, 45, 73], [28, 25, 29], [29, 25, 31], [43, 73, 44], [73, 43, 40],
    [72, 73, 39], [72, 31, 25], [42, 40, 43], [31, 30, 29], [39, 73, 40],
    [42, 41, 40], [72, 33, 31], [32, 31, 33], [39, 38, 72], [33, 72, 38],
    [33, 38, 34], [37, 35, 38], [34, 38, 35], [35, 37, 36]])

# Rather than create a Triangulation object, can simply pass x, y and triangles
# arrays to triplot directly.  It would be better to use a Triangulation object
# if the same triangulation was to be used more than once to save duplicated
# calculations.
plt.figure()
plt.gca().set_aspect('equal')
plt.triplot(x, y, triangles, 'go-')
plt.title('triplot of user-specified triangulation')
plt.xlabel('Longitude (degrees)')
plt.ylabel('Latitude (degrees)')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.210 pylab_examples example code: usetex_baseline_test.py



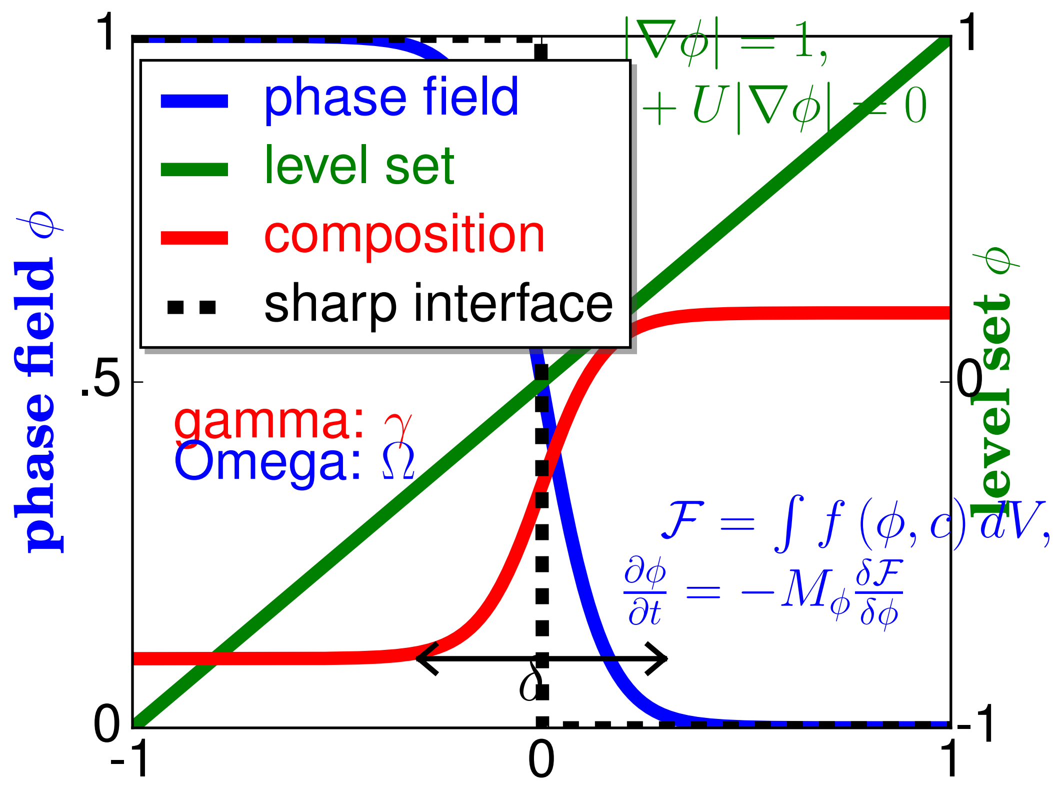

```
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.axes as maxes

from matplotlib import rcParams
rcParams['text.usetex'] = True
rcParams['text.latex.unicode'] = True


class Axes(maxes.Axes):
```

```
    """
    A hackish way to simultaneously draw texts w/ usetex=True and
    usetex=False in the same figure. It does not work in the ps backend.
    """

    def __init__(self, *kl, **kw):
        self.usetex = kw.pop("usetex", "False")
        self.preview = kw.pop("preview", "False")

        maxes.Axes.__init__(self, *kl, **kw)

    def draw(self, renderer):
        usetex = plt.rcParams["text.usetex"]
        preview = plt.rcParams["text.latex.preview"]
        plt.rcParams["text.usetex"] = self.usetex
        plt.rcParams["text.latex.preview"] = self.preview

        maxes.Axes.draw(self, renderer)

        plt.rcParams["text.usetex"] = usetex
        plt.rcParams["text.latex.preview"] = preview

subplot = maxes.subplot_class_factory(Axes)


def test_window_extent(ax, usetex, preview):

    va = "baseline"
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)

    #t = ax.text(0., 0., r"mlp", va="baseline", size=150)
    text_kw = dict(va=va,
                   size=50,
                   bbox=dict(pad=0., ec="k", fc="none"))

    test_strings = ["lg", r"$\frac{1}{2}\pi$",
                    r"$p^{3^A}$", r"$p_{3_2}$"]

    ax.axvline(0, color="r")

    for i, s in enumerate(test_strings):

        ax.axhline(i, color="r")
        ax.text(0., 3 - i, s, **text_kw)

    ax.set_xlim(-0.1, 1.1)
    ax.set_ylim(-.8, 3.9)

    ax.set_title("usetex=%s\npreview=%s" % (str(usetex), str(preview)))


fig = plt.figure(figsize=(2.*3, 6.5))
```

```
for i, usetex, preview in [[0, False, False],
                           [1, True, False],
                           [2, True, True]]:
    ax = subplot(fig, 1, 3, i + 1, usetex=usetex, preview=preview)
    fig.add_subplot(ax)
    fig.subplots_adjust(top=0.85)

    test_window_extent(ax, usetex=usetex, preview=preview)


plt.draw()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.211 pylab_examples example code: usetex_demo.py



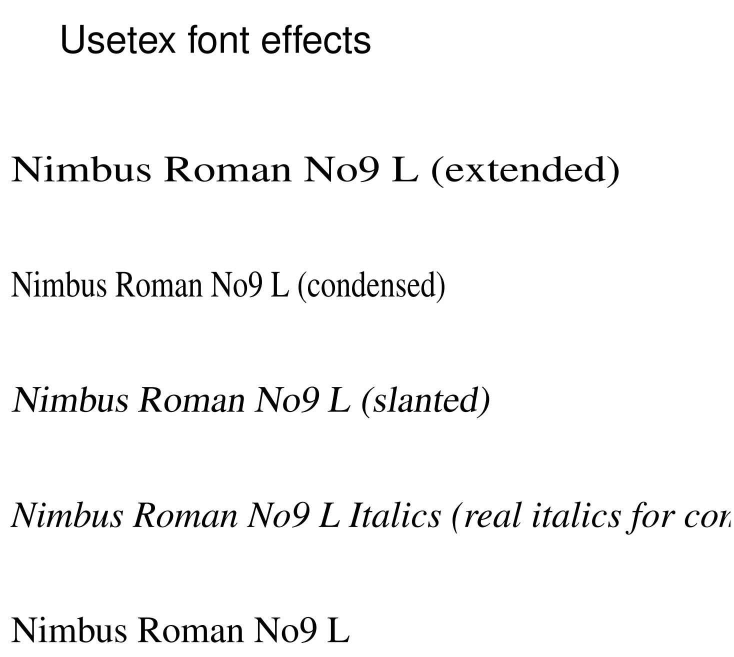

```
import matplotlib
matplotlib.rc('text', usetex=True)
import matplotlib.pyplot as plt
import numpy as np
```

```
# interface tracking profiles
N = 500
delta = 0.6
X = np.linspace(-1, 1, N)
plt.plot(X, (1 - np.tanh(4.*X/delta))/2,    # phase field tanh profiles
         X, (X + 1)/2,                        # level set distance function
         X, (1.4 + np.tanh(4.*X/delta))/4,  # composition profile
         X, X < 0, 'k--',                     # sharp interface
         linewidth=5)

# legend
plt.legend(('phase field', 'level set', 'composition', 'sharp interface'), shadow=True,␣
 ↪loc=(0.01, 0.55))

ltext = plt.gca().get_legend().get_texts()
plt.setp(ltext[0], fontsize=20, color='b')
plt.setp(ltext[1], fontsize=20, color='g')
plt.setp(ltext[2], fontsize=20, color='r')
plt.setp(ltext[3], fontsize=20, color='k')

# the arrow
height = 0.1
offset = 0.02
plt.plot((-delta / 2., delta / 2), (height, height), 'k', linewidth=2)
plt.plot((-delta / 2, -delta / 2 + offset * 2), (height, height - offset), 'k',␣
 ↪linewidth=2)
plt.plot((-delta / 2, -delta / 2 + offset * 2), (height, height + offset), 'k',␣
 ↪linewidth=2)
plt.plot((delta / 2, delta / 2 - offset * 2), (height, height - offset), 'k',␣
 ↪linewidth=2)
plt.plot((delta / 2, delta / 2 - offset * 2), (height, height + offset), 'k',␣
 ↪linewidth=2)
plt.text(-0.06, height - 0.06, r'$\delta$', {'color': 'k', 'fontsize': 24})

# X-axis label
plt.xticks((-1, 0, 1), ('-1', '0', '1'), color='k', size=20)

# Left Y-axis labels
plt.ylabel(r'\bf{phase field} $\phi$', {'color': 'b',
                                       'fontsize': 20})
plt.yticks((0, 0.5, 1), ('0', '.5', '1'), color='k', size=20)

# Right Y-axis labels
plt.text(1.05, 0.5, r"\bf{level set} $\phi$", {'color': 'g', 'fontsize': 20},
         horizontalalignment='left',
         verticalalignment='center',
         rotation=90,
         clip_on=False)
plt.text(1.01, -0.02, "-1", {'color': 'k', 'fontsize': 20})
plt.text(1.01, 0.98, "1", {'color': 'k', 'fontsize': 20})
plt.text(1.01, 0.48, "0", {'color': 'k', 'fontsize': 20})
```

```
# level set equations
plt.text(0.1, 0.85,
         r'$|\nabla\phi| = 1,$ \newline $ \frac{\partial \phi}{\partial t}'
         r'+ U|\nabla \phi| = 0$',
         {'color': 'g', 'fontsize': 20})

# phase field equations
plt.text(0.2, 0.15,
         r'$\mathcal{F} = \int f\left( \phi, c \right) dV,$ \newline '
         r'$ \frac{ \partial \phi } { \partial t } = -M_{ \phi } '
         r'\frac{ \delta \mathcal{F} } { \delta \phi }$',
         {'color': 'b', 'fontsize': 20})

# these went wrong in pdf in a previous version
plt.text(-.9, .42, r'gamma: $\gamma$', {'color': 'r', 'fontsize': 20})
plt.text(-.9, .36, r'Omega: $\Omega$', {'color': 'b', 'fontsize': 20})

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

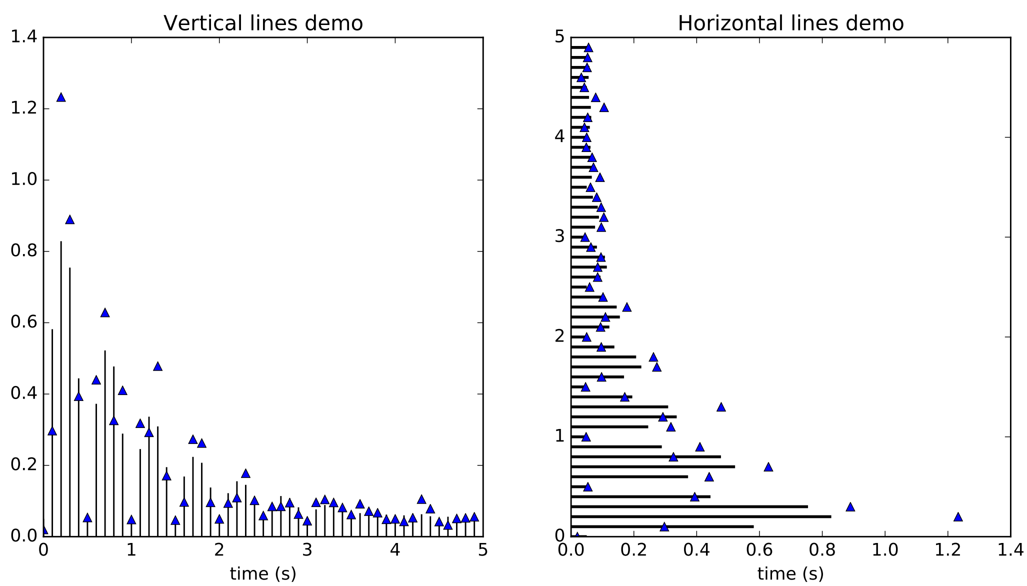
90.212 pylab_examples example code: usetex_fonteffects.py



```
# This script demonstrates that font effects specified in your pdftex.map
# are now supported in pdf usetex.

import matplotlib
import matplotlib.pyplot as plt
matplotlib.rc('text', usetex=True)


def setfont(font):
    return r'\font\a %s at 14pt\a ' % font

for y, font, text in zip(range(5),
                         ['ptmr8r', 'ptmri8r', 'ptmro8r', 'ptmr8rn', 'ptmrr8re'],
                         ['Nimbus Roman No9 L ' + x for x in
                          ['', 'Italics (real italics for comparison)',
                           '(slanted)', '(condensed)', '(extended)']]):
    plt.text(0, y, setfont(font) + text)

plt.ylim(-1, 5)
plt.xlim(-0.2, 0.6)
plt.setp(plt.gca(), frame_on=False, xticks=(), yticks=())
plt.title('Usetex font effects')
plt.savefig('usetex_fonteffects.pdf')
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.213 pylab_examples example code: vline_hline_demo.py




```
"""
Small demonstration of the hlines and vlines plots.
"""
```

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rnd


def f(t):
    s1 = np.sin(2 * np.pi * t)
    e1 = np.exp(-t)
    return np.absolute((s1 * e1)) + .05

t = np.arange(0.0, 5.0, 0.1)
s = f(t)
nse = rnd.normal(0.0, 0.3, t.shape) * s

fig = plt.figure(figsize=(12, 6))
vax = fig.add_subplot(121)
hax = fig.add_subplot(122)

vax.plot(t, s + nse, 'b^')
vax.vlines(t, [0], s)
vax.set_xlabel('time (s)')
vax.set_title('Vertical lines demo')

hax.plot(s + nse, t, 'b^')
hax.hlines(t, [0], s, lw=2)
hax.set_xlabel('time (s)')
hax.set_title('Horizontal lines demo')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.214 pylab_examples example code: webapp_demo.py

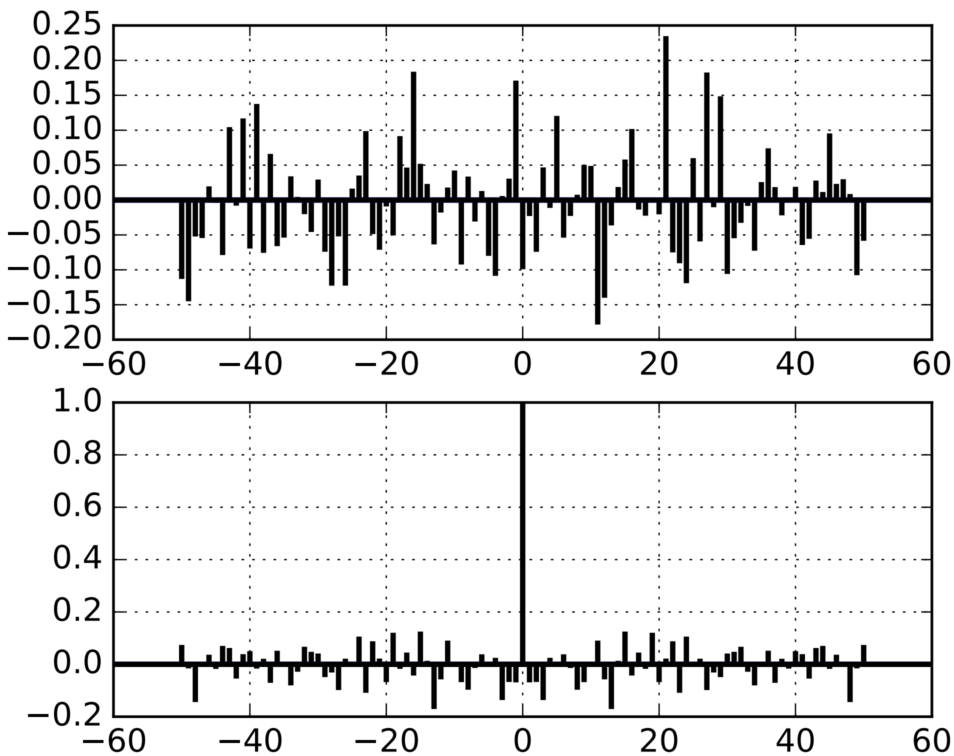
[source code]

```
#!/usr/bin/env python
# -*- noplot -*-
"""
This example shows how to use the agg backend directly to create
images, which may be of use to web application developers who want
full control over their code without using the pyplot interface to
manage figures, figure closing etc.

.. note::

    It is not necessary to avoid using the pyplot interface in order to
    create figures without a graphical front-end - simply setting
    the backend to "Agg" would be sufficient.

```

```
It is also worth noting that, because matplotlib can save figures to file-like
object, matplotlib can also be used inside a cgi-script *without* needing to
write a figure to disk.

"""

from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure
import numpy as np


def make_fig():
    """
    Make a figure and save it to "webagg.png".

    """
    fig = Figure()
    ax = fig.add_subplot(1, 1, 1)

    ax.plot([1, 2, 3], 'ro--', markersize=12, markerfacecolor='g')

    # make a translucent scatter collection
    x = np.random.rand(100)
    y = np.random.rand(100)
    area = np.pi * (10 * np.random.rand(100)) ** 2  # 0 to 10 point radiuses
    c = ax.scatter(x, y, area)
    c.set_alpha(0.5)

    # add some text decoration
    ax.set_title('My first image')
    ax.set_ylabel('Some numbers')
    ax.set_xticks((.2, .4, .6, .8))
    labels = ax.set_xticklabels(('Bill', 'Fred', 'Ted', 'Ed'))

    # To set object properties, you can either iterate over the
    # objects manually, or define you own set command, as in setapi
    # above.
    for label in labels:
        label.set_rotation(45)
        label.set_fontsize(12)

    FigureCanvasAgg(fig).print_png('webapp.png', dpi=150)

make_fig()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

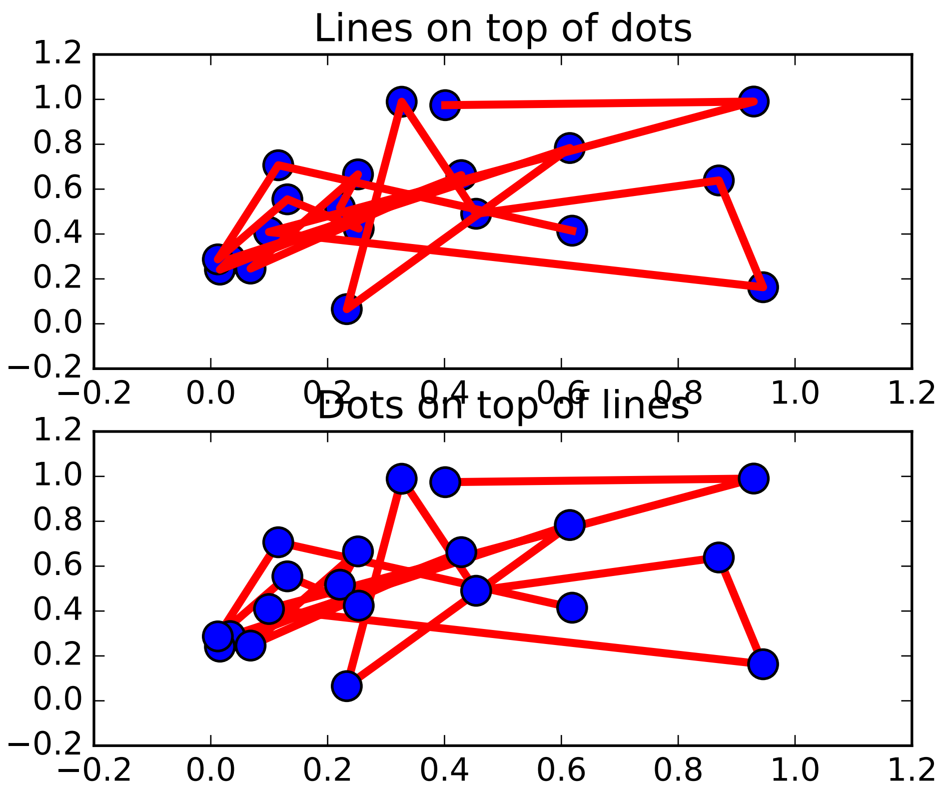
90.215 pylab_examples example code: xcorr_demo.py

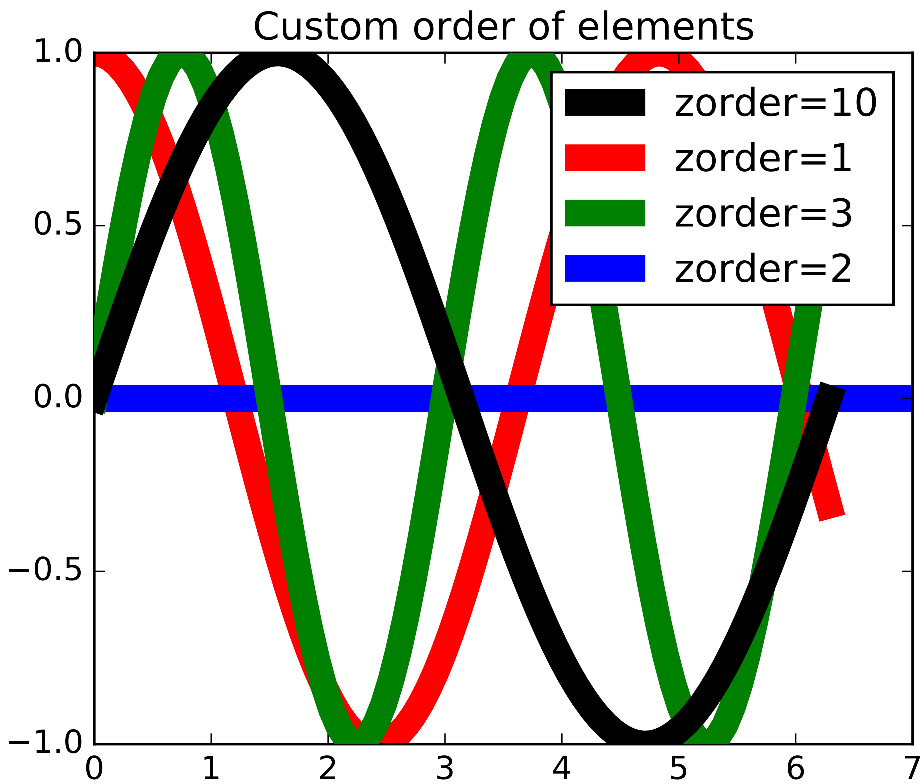



```
import matplotlib.pyplot as plt
import numpy as np

x, y = np.random.randn(2, 100)
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax1.xcorr(x, y, usevlines=True, maxlags=50, normed=True, lw=2)
ax1.grid(True)
ax1.axhline(0, color='black', lw=2)

ax2 = fig.add_subplot(212, sharex=ax1)
ax2.acorr(x, usevlines=True, normed=True, maxlags=50, lw=2)
ax2.grid(True)
ax2.axhline(0, color='black', lw=2)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

90.216 pylab_examples example code: zorder_demo.py






```
#!/usr/bin/env python
"""
The default drawing order for axes is patches, lines, text.  This
order is determined by the zorder attribute.  The following defaults
are set

Artist                        Z-order
Patch / PatchCollection         1
Line2D / LineCollection         2
Text                            3

You can change the order for individual artists by setting the zorder.  Any
individual plot() call can set a value for the zorder of that particular item.

In the fist subplot below, the lines are drawn above the patch
collection from the scatter, which is the default.

In the subplot below, the order is reversed.

The second figure shows how to control the zorder of individual lines.
"""

import matplotlib.pyplot as plt
import numpy as np

x = np.random.random(20)
```

```
y = np.random.random(20)

# Lines on top of scatter
plt.figure()
plt.subplot(211)
plt.plot(x, y, 'r', lw=3)
plt.scatter(x, y, s=120)
plt.title('Lines on top of dots')

# Scatter plot on top of lines
plt.subplot(212)
plt.plot(x, y, 'r', zorder=1, lw=3)
plt.scatter(x, y, s=120, zorder=2)
plt.title('Dots on top of lines')

# A new figure, with individually ordered items
x = np.linspace(0, 2*np.pi, 100)
plt.figure()
plt.plot(x, np.sin(x), linewidth=10, color='black', label='zorder=10', zorder=10)  # on
 ↪top
plt.plot(x, np.cos(1.3*x), linewidth=10, color='red', label='zorder=1', zorder=1)  #
 ↪bottom
plt.plot(x, np.sin(2.1*x), linewidth=10, color='green', label='zorder=3', zorder=3)
plt.axhline(0, linewidth=10, color='blue', label='zorder=2', zorder=2)
plt.title('Custom order of elements')
l = plt.legend()
l.set_zorder(20)  # put the legend on top
plt.show()
```

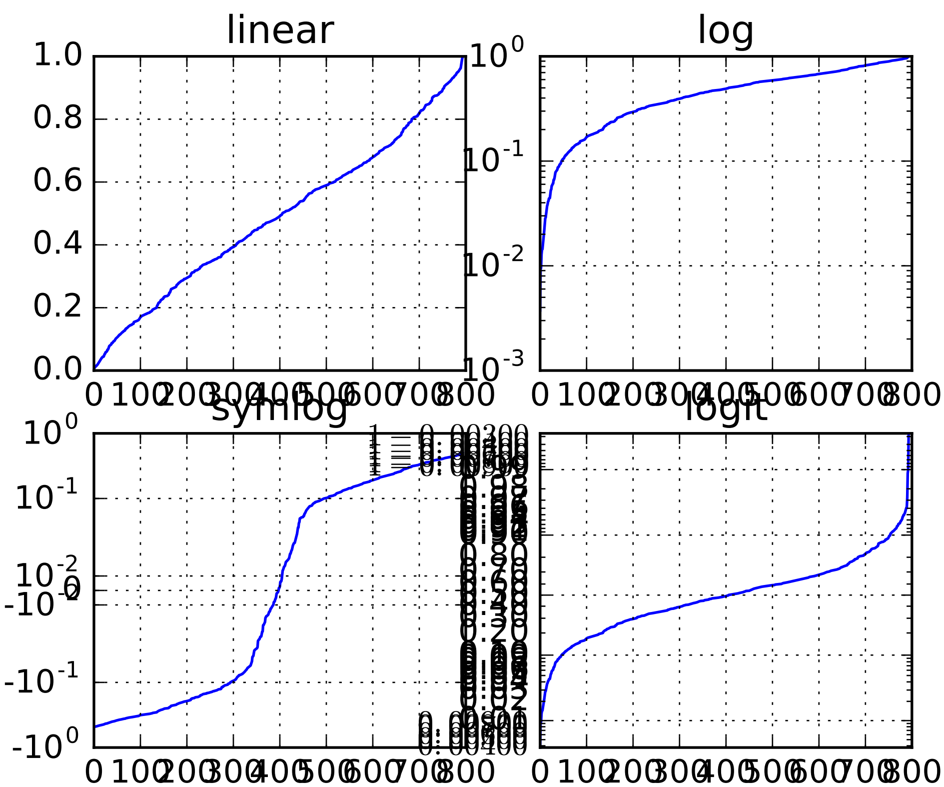
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

NINETYONE

SCALES EXAMPLES

91.1 scales example code: scales.py




```
"""
Illustrate the scale transformations applied to axes, e.g. log, symlog, logit.
"""
import numpy as np
import matplotlib.pyplot as plt

# make up some data in the interval ]0, 1[
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
```

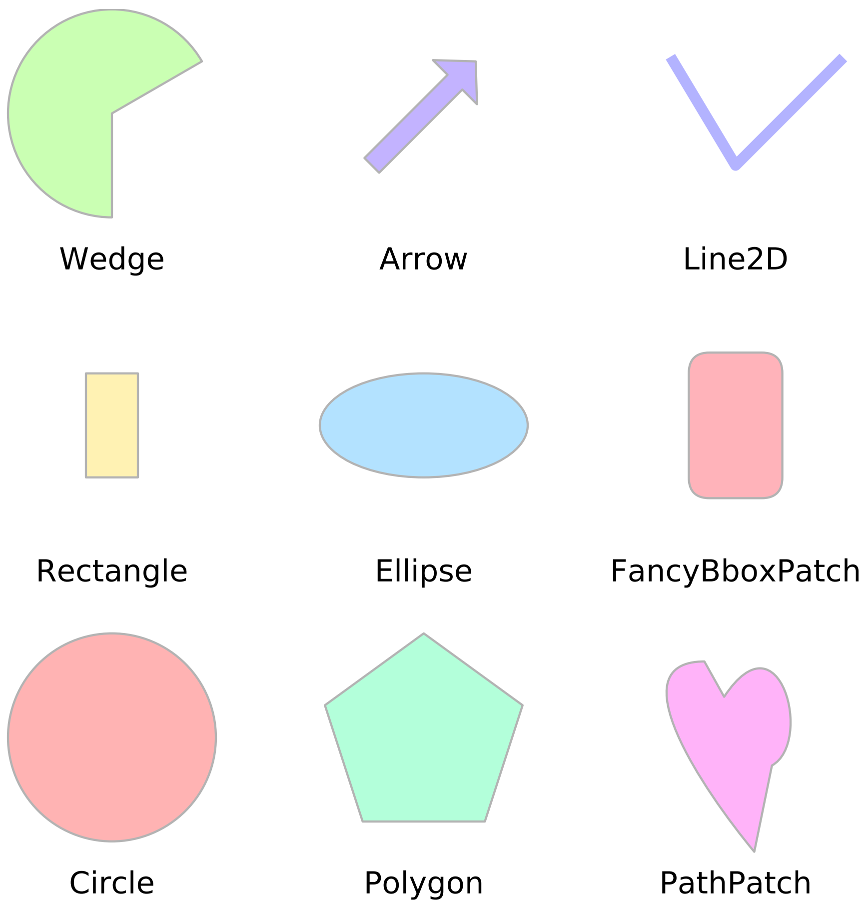
```
y.sort()
x = np.arange(len(y))

# plot with various axes scales
fig, axs = plt.subplots(2, 2)

# linear
ax = axs[0, 0]
ax.plot(x, y)
ax.set_yscale('linear')
ax.set_title('linear')
ax.grid(True)


# log
ax = axs[0, 1]
ax.plot(x, y)
ax.set_yscale('log')
ax.set_title('log')
ax.grid(True)


# symmetric log
ax = axs[1, 0]
ax.plot(x, y - y.mean())
ax.set_yscale('symlog', linthreshy=0.05)
ax.set_title('symlog')
ax.grid(True)

# logit
ax = axs[1, 1]
ax.plot(x, y)
ax.set_yscale('logit')
ax.set_title('logit')
ax.grid(True)


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER NINETYTWO

SHAPES_AND_COLLECTIONS EXAMPLES

92.1 shapes_and_collections example code: artist_reference.py



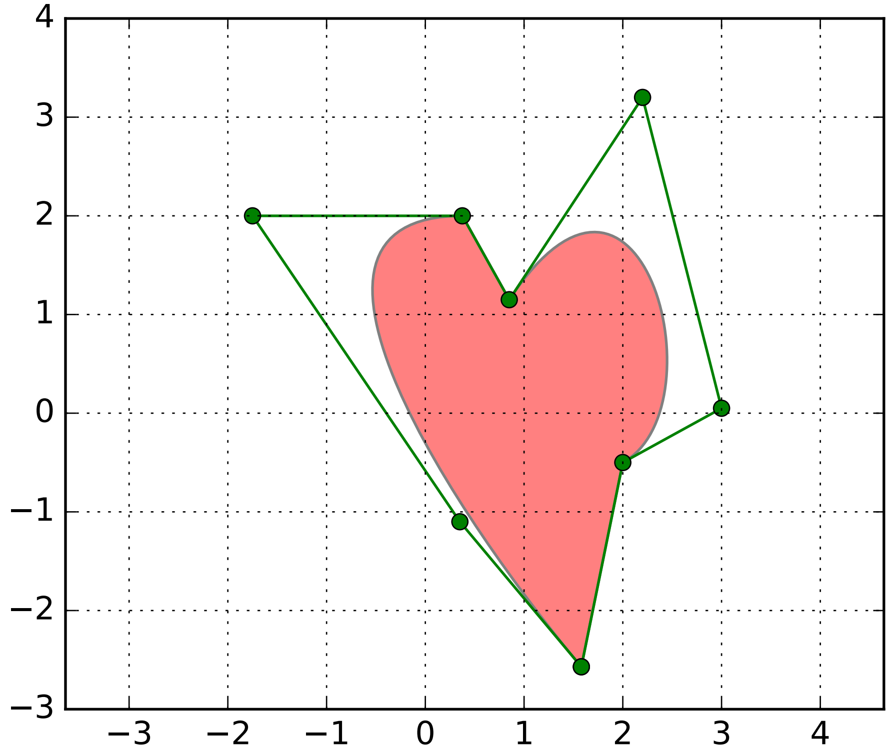

```
"""
Reference for matplotlib artists

This example displays several of matplotlib's graphics primitives (artists)
drawn using matplotlib API. A full list of artists and the documentation is
available at http://matplotlib.org/api/artist_api.html.

```

```
Copyright (c) 2010, Bartosz Telenczuk
BSD License
"""
import matplotlib.pyplot as plt
plt.rcdefaults()

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.path as mpath
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
from matplotlib.collections import PatchCollection


def label(xy, text):
    y = xy[1] - 0.15  # shift y-value for label so that it's below the artist
    plt.text(xy[0], y, text, ha="center", family='sans-serif', size=14)


fig, ax = plt.subplots()
# create 3x3 grid to plot the artists
grid = np.mgrid[0.2:0.8:3j, 0.2:0.8:3j].reshape(2, -1).T

patches = []

# add a circle
circle = mpatches.Circle(grid[0], 0.1, ec="none")
patches.append(circle)
label(grid[0], "Circle")

# add a rectangle
rect = mpatches.Rectangle(grid[1] - [0.025, 0.05], 0.05, 0.1, ec="none")
patches.append(rect)
label(grid[1], "Rectangle")

# add a wedge
wedge = mpatches.Wedge(grid[2], 0.1, 30, 270, ec="none")
patches.append(wedge)
label(grid[2], "Wedge")

# add a Polygon
polygon = mpatches.RegularPolygon(grid[3], 5, 0.1)
patches.append(polygon)
label(grid[3], "Polygon")

# add an ellipse
ellipse = mpatches.Ellipse(grid[4], 0.2, 0.1)
patches.append(ellipse)
label(grid[4], "Ellipse")

# add an arrow
arrow = mpatches.Arrow(grid[5, 0] - 0.05, grid[5, 1] - 0.05, 0.1, 0.1, width=0.1)
patches.append(arrow)
```

```
label(grid[5], "Arrow")

# add a path patch
Path = mpath.Path
path_data = [
    (Path.MOVETO, [0.018, -0.11]),
    (Path.CURVE4, [-0.031, -0.051]),
    (Path.CURVE4, [-0.115,  0.073]),
    (Path.CURVE4, [-0.03 ,  0.073]),
    (Path.LINETO, [-0.011,  0.039]),
    (Path.CURVE4, [0.043,  0.121]),
    (Path.CURVE4, [0.075, -0.005]),
    (Path.CURVE4, [0.035, -0.027]),
    (Path.CLOSEPOLY, [0.018, -0.11])
    ]
codes, verts = zip(*path_data)
path = mpath.Path(verts + grid[6], codes)
patch = mpatches.PathPatch(path)
patches.append(patch)
label(grid[6], "PathPatch")

# add a fancy box
fancybox = mpatches.FancyBboxPatch(
    grid[7] - [0.025, 0.05], 0.05, 0.1,
    boxstyle=mpatches.BoxStyle("Round", pad=0.02))
patches.append(fancybox)
label(grid[7], "FancyBboxPatch")

# add a line
x, y = np.array([[-0.06, 0.0, 0.1], [0.05, -0.05, 0.05]])
line = mlines.Line2D(x + grid[8, 0], y + grid[8, 1], lw=5., alpha=0.3)
label(grid[8], "Line2D")

colors = np.linspace(0, 1, len(patches))
collection = PatchCollection(patches, cmap=plt.cm.hsv, alpha=0.3)
collection.set_array(np.array(colors))
ax.add_collection(collection)
ax.add_line(line)

plt.subplots_adjust(left=0, right=1, bottom=0, top=1)
plt.axis('equal')
plt.axis('off')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

92.2 shapes_and_collections example code: path_patch_demo.py

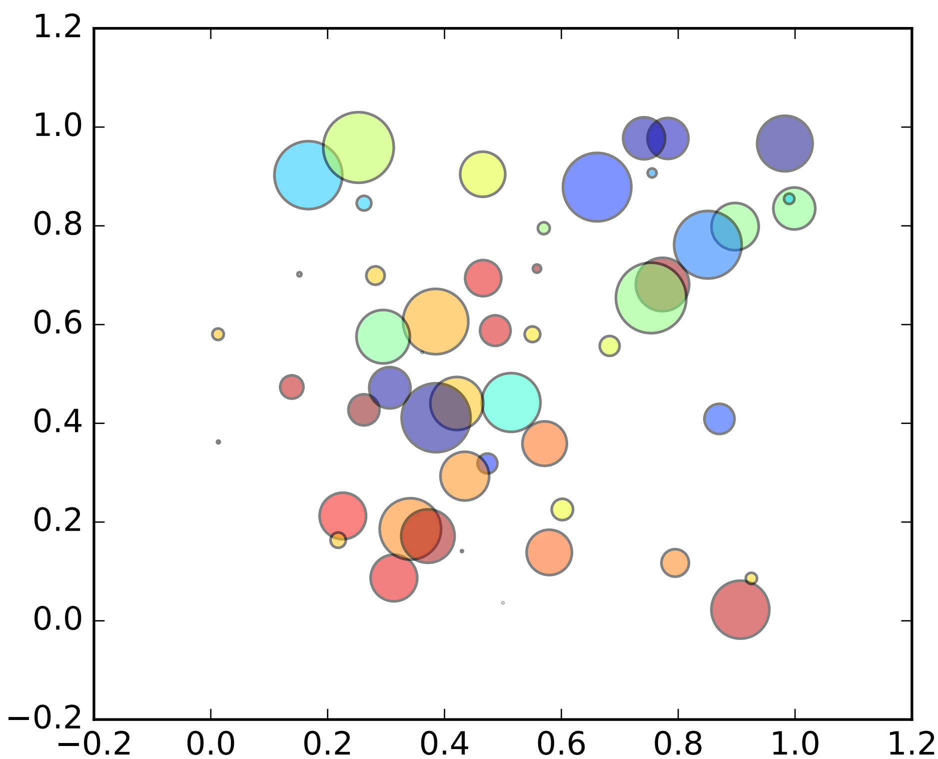



```
"""
Demo of a PathPatch object.
"""
import matplotlib.path as mpath
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt


fig, ax = plt.subplots()

Path = mpath.Path
path_data = [
    (Path.MOVETO, (1.58, -2.57)),
    (Path.CURVE4, (0.35, -1.1)),
    (Path.CURVE4, (-1.75, 2.0)),
    (Path.CURVE4, (0.375, 2.0)),
    (Path.LINETO, (0.85, 1.15)),
    (Path.CURVE4, (2.2, 3.2)),
    (Path.CURVE4, (3, 0.05)),
    (Path.CURVE4, (2.0, -0.5)),
    (Path.CLOSEPOLY, (1.58, -2.57)),
    ]
```

```
codes, verts = zip(*path_data)
path = mpath.Path(verts, codes)
patch = mpatches.PathPatch(path, facecolor='r', alpha=0.5)
ax.add_patch(patch)

# plot control points and connecting lines
x, y = zip(*path.vertices)
line, = ax.plot(x, y, 'go-')

ax.grid()
ax.axis('equal')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

92.3 shapes_and_collections example code: scatter_demo.py




```
"""
Simple demo of a scatter plot.
"""
import numpy as np
import matplotlib.pyplot as plt
```

```
N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radiuses

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```

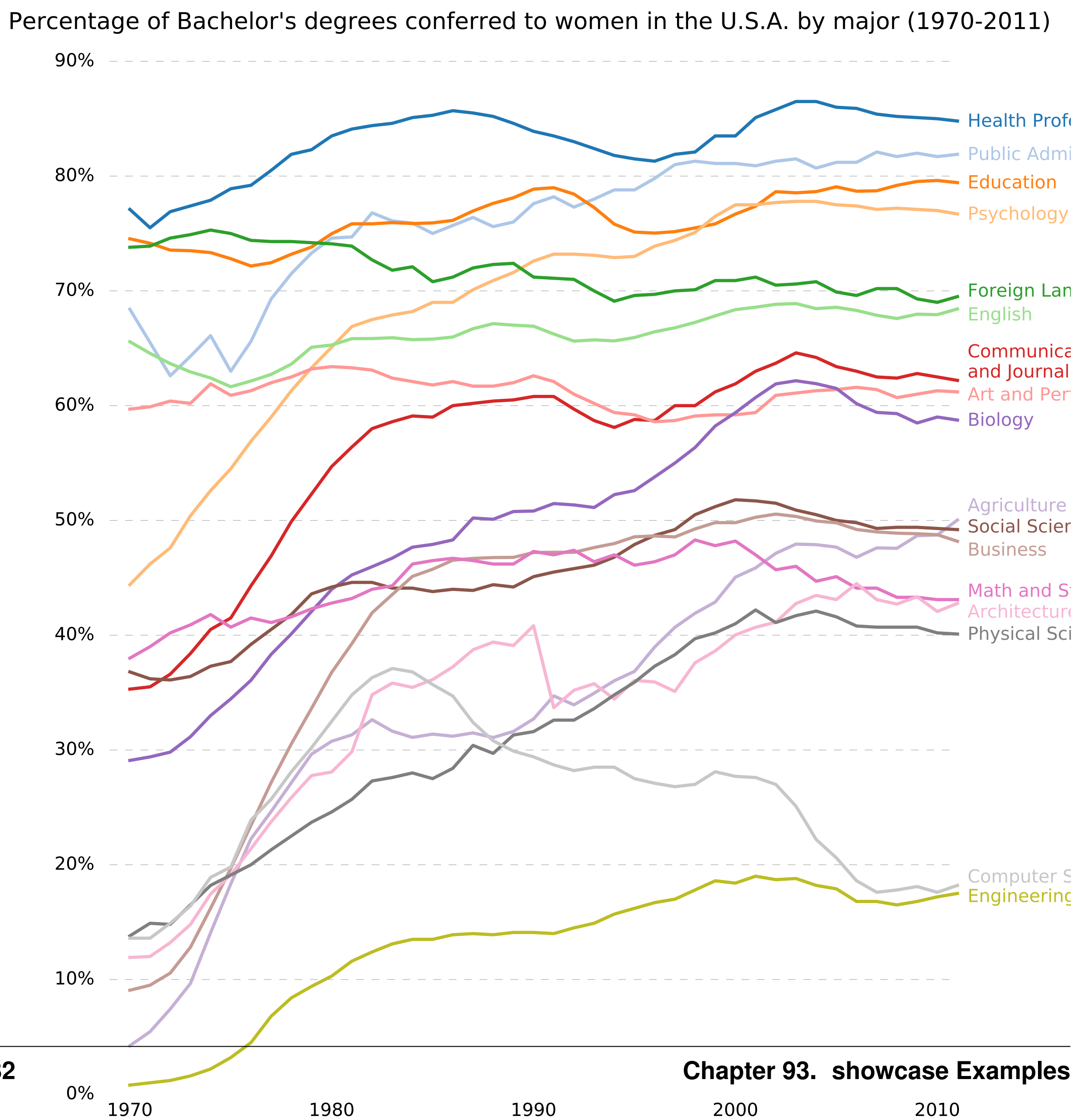
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

NINETYTHREE

SHOWCASE EXAMPLES

93.1 showcase example code: bachelors_degrees_by_gender.py

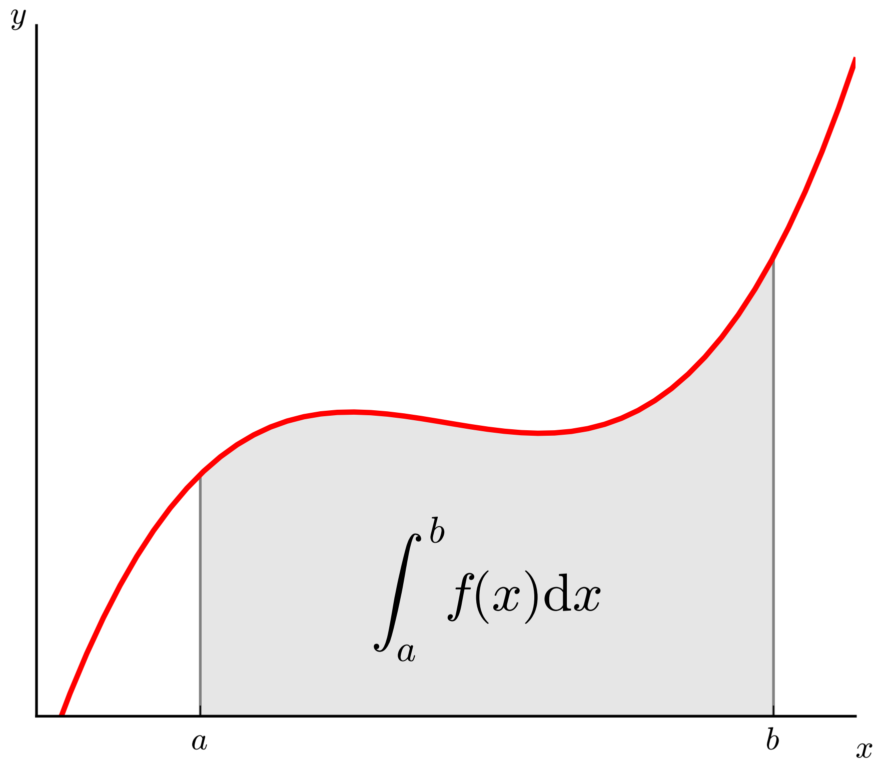


```
import matplotlib.pyplot as plt
from matplotlib.mlab import csv2rec
from matplotlib.cbook import get_sample_data

fname = get_sample_data('percent_bachelors_degrees_women_usa.csv')
gender_degree_data = csv2rec(fname)

# These are the colors that will be used in the plot
color_sequence = ['#1f77b4', '#aec7e8', '#ff7f0e', '#ffbb78', '#2ca02c',
                  '#98df8a', '#d62728', '#ff9896', '#9467bd', '#c5b0d5',
                  '#8c564b', '#c49c94', '#e377c2', '#f7b6d2', '#7f7f7f',
                  '#c7c7c7', '#bcbd22', '#dbdb8d', '#17becf', '#9edae5']

# You typically want your plot to be ~1.33x wider than tall. This plot
# is a rare exception because of the number of lines being plotted on it.
# Common sizes: (10, 7.5) and (12, 9)
fig, ax = plt.subplots(1, 1, figsize=(12, 14))

# Remove the plot frame lines. They are unnecessary here.
ax.spines['top'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_visible(False)

# Ensure that the axis ticks only show up on the bottom and left of the plot.
# Ticks on the right and top of the plot are generally unnecessary.
ax.get_xaxis().tick_bottom()
ax.get_yaxis().tick_left()

# Limit the range of the plot to only where the data is.
# Avoid unnecessary whitespace.
plt.xlim(1968.5, 2011.1)
plt.ylim(-0.25, 90)

# Make sure your axis ticks are large enough to be easily read.
# You don't want your viewers squinting to read your plot.
plt.xticks(range(1970, 2011, 10), fontsize=14)
plt.yticks(range(0, 91, 10), ['{0}%'.format(x)
                              for x in range(0, 91, 10)], fontsize=14)

# Provide tick lines across the plot to help your viewers trace along
# the axis ticks. Make sure that the lines are light and small so they
# don't obscure the primary data lines.
for y in range(10, 91, 10):
    plt.plot(range(1969, 2012), [y] * len(range(1969, 2012)), '--',
             lw=0.5, color='black', alpha=0.3)

# Remove the tick marks; they are unnecessary with the tick lines we just
# plotted.
plt.tick_params(axis='both', which='both', bottom='off', top='off',
                labelbottom='on', left='off', right='off', labelleft='on')

# Now that the plot is prepared, it's time to actually plot the data!
```

```
# Note that I plotted the majors in order of the highest % in the final year.
majors = ['Health Professions', 'Public Administration', 'Education',
          'Psychology', 'Foreign Languages', 'English',
          'Communications\nand Journalism', 'Art and Performance', 'Biology',
          'Agriculture', 'Social Sciences and History', 'Business',
          'Math and Statistics', 'Architecture', 'Physical Sciences',
          'Computer Science', 'Engineering']

y_offsets = {'Foreign Languages': 0.5, 'English': -0.5,
             'Communications\nand Journalism': 0.75,
             'Art and Performance': -0.25, 'Agriculture': 1.25,
             'Social Sciences and History': 0.25, 'Business': -0.75,
             'Math and Statistics': 0.75, 'Architecture': -0.75,
             'Computer Science': 0.75, 'Engineering': -0.25}

for rank, column in enumerate(majors):
    # Plot each line separately with its own color.
    column_rec_name = column.replace('\n', '_').replace(' ', '_').lower()

    line = plt.plot(gender_degree_data.year,
                    gender_degree_data[column_rec_name],
                    lw=2.5,
                    color=color_sequence[rank])

    # Add a text label to the right end of every line. Most of the code below
    # is adding specific offsets y position because some labels overlapped.
    y_pos = gender_degree_data[column_rec_name][-1] - 0.5

    if column in y_offsets:
        y_pos += y_offsets[column]

    # Again, make sure that all labels are large enough to be easily read
    # by the viewer.
    plt.text(2011.5, y_pos, column, fontsize=14, color=color_sequence[rank])

# Make the title big enough so it spans the entire plot, but don't make it
# so big that it requires two lines to show.

# Note that if the title is descriptive enough, it is unnecessary to include
# axis labels; they are self-evident, in this plot's case.
plt.title('Percentage of Bachelor\'s degrees conferred to women in '
          'the U.S.A. by major (1970-2011)\n', fontsize=18, ha='center')

# Finally, save the figure as a PNG.
# You can also save it as a PDF, JPEG, etc.
# Just change the file extension in this call.
plt.savefig('percent-bachelors-degrees-women-usa.png', bbox_inches='tight')
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

93.2 showcase example code: integral_demo.py

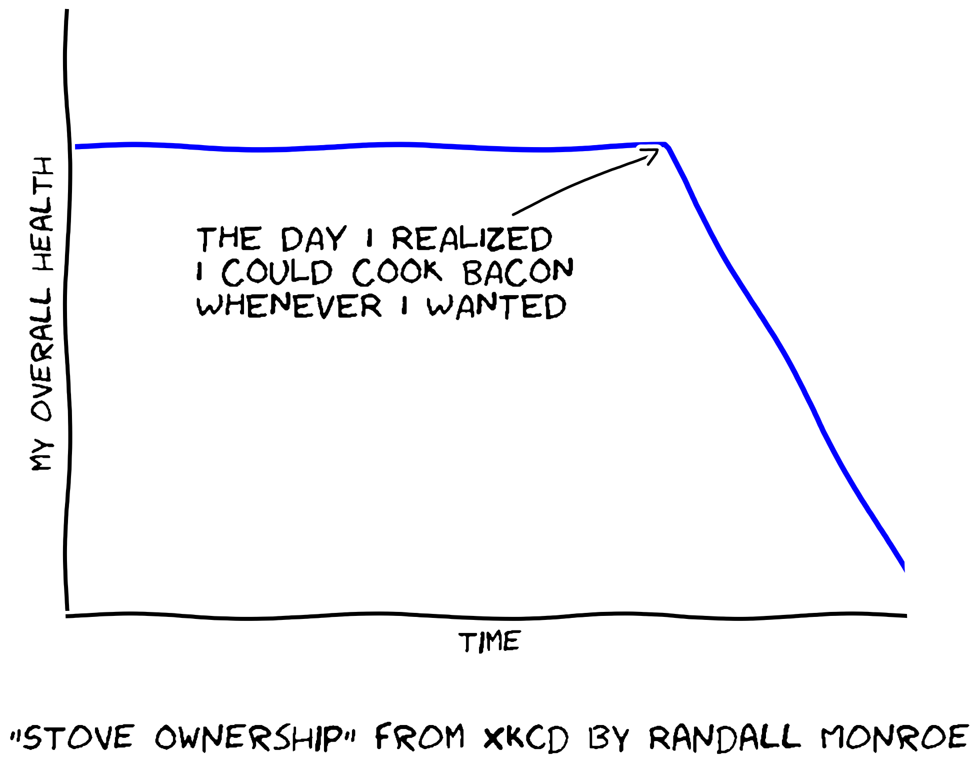


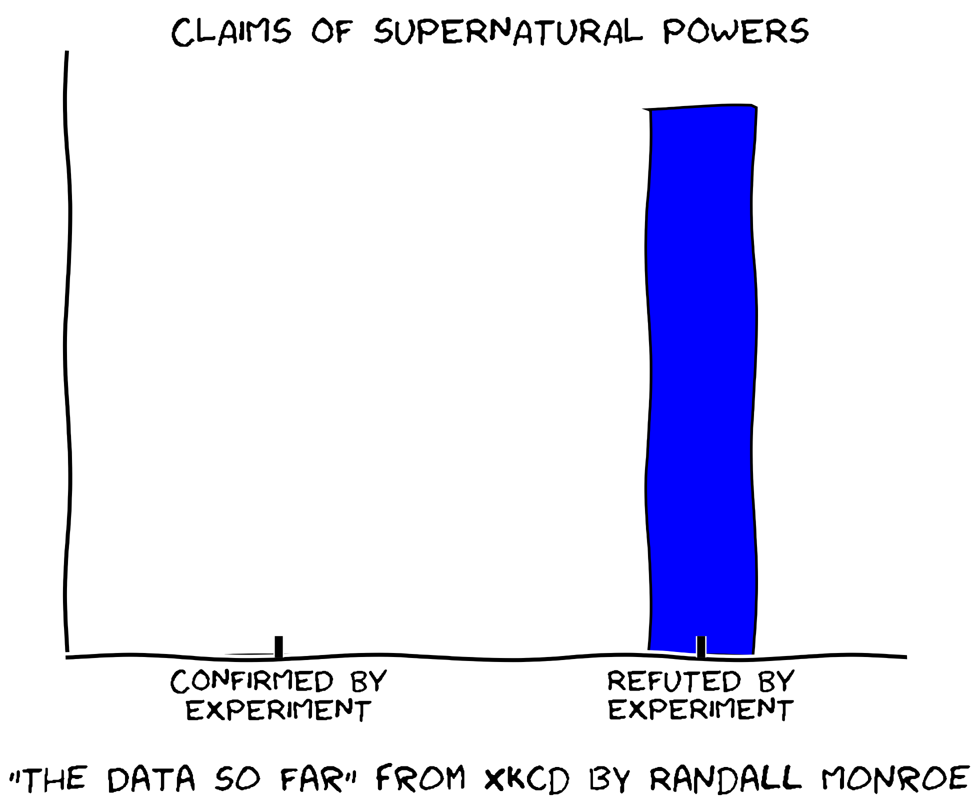

```
"""
Plot demonstrating the integral as the area under a curve.

Although this is a simple example, it demonstrates some important tweaks:

    * A simple line plot with custom color and line width.
    * A shaded region created using a Polygon patch.
    * A text label with mathtext rendering.
    * figtext calls to label the x- and y-axes.
    * Use of axis spines to hide the top and right spines.
    * Custom tick placement and labels.
"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon


def func(x):
    return (x - 3) * (x - 5) * (x - 7) + 85


a, b = 2, 9  # integral limits
```

```
x = np.linspace(0, 10)
y = func(x)

fig, ax = plt.subplots()
plt.plot(x, y, 'r', linewidth=2)
plt.ylim(ymin=0)

# Make the shaded region
ix = np.linspace(a, b)
iy = func(ix)
verts = [(a, 0)] + list(zip(ix, iy)) + [(b, 0)]
poly = Polygon(verts, facecolor='0.9', edgecolor='0.5')
ax.add_patch(poly)

plt.text(0.5 * (a + b), 30, r"$\int_a^b f(x)\mathrm{d}x$",
         horizontalalignment='center', fontsize=20)

plt.figtext(0.9, 0.05, '$x$')
plt.figtext(0.1, 0.9, '$y$')

ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.xaxis.set_ticks_position('bottom')

ax.set_xticks((a, b))
ax.set_xticklabels(('$a$', '$b$'))
ax.set_yticks([])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

93.3 showcase example code: xkcd.py






```
import matplotlib.pyplot as plt
import numpy as np

with plt.xkcd():
    # Based on "Stove Ownership" from XKCD by Randall Monroe
    # http://xkcd.com/418/

    fig = plt.figure()
    ax = fig.add_axes((0.1, 0.2, 0.8, 0.7))
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    plt.xticks([])
    plt.yticks([])
    ax.set_ylim([-30, 10])

    data = np.ones(100)
    data[70:] -= np.arange(30)

    plt.annotate(
        'THE DAY I REALIZED\nI COULD COOK BACON\nWHENEVER I WANTED',
        xy=(70, 1), arrowprops=dict(arrowstyle='->'), xytext=(15, -10))

    plt.plot(data)

    plt.xlabel('time')
    plt.ylabel('my overall health')
```

```
    fig.text(
        0.5, 0.05,
        '"Stove Ownership" from xkcd by Randall Monroe',
        ha='center')

    # Based on "The Data So Far" from XKCD by Randall Monroe
    # http://xkcd.com/373/

    fig = plt.figure()
    ax = fig.add_axes((0.1, 0.2, 0.8, 0.7))
    ax.bar([-0.125, 1.0 - 0.125], [0, 100], 0.25)
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.set_xticks([0, 1])
    ax.set_xlim([-0.5, 1.5])
    ax.set_ylim([0, 110])
    ax.set_xticklabels(['CONFIRMED BY\nEXPERIMENT', 'REFUTED BY\nEXPERIMENT'])
    plt.yticks([])

    plt.title("CLAIMS OF SUPERNATURAL POWERS")

    fig.text(
        0.5, 0.05,
        '"The Data So Far" from xkcd by Randall Monroe',
        ha='center')

plt.show()
```

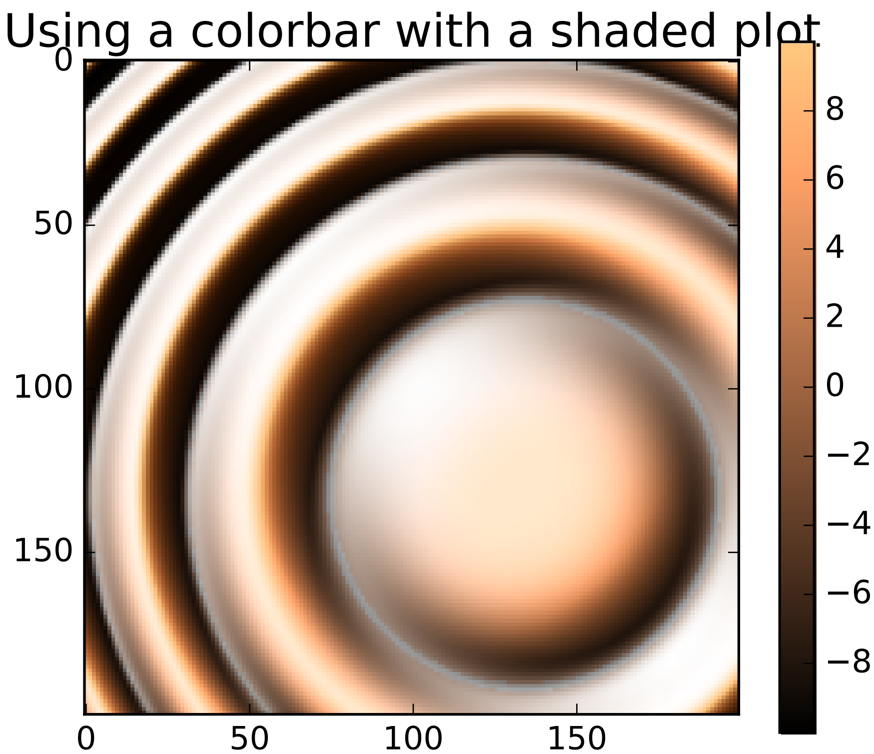
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

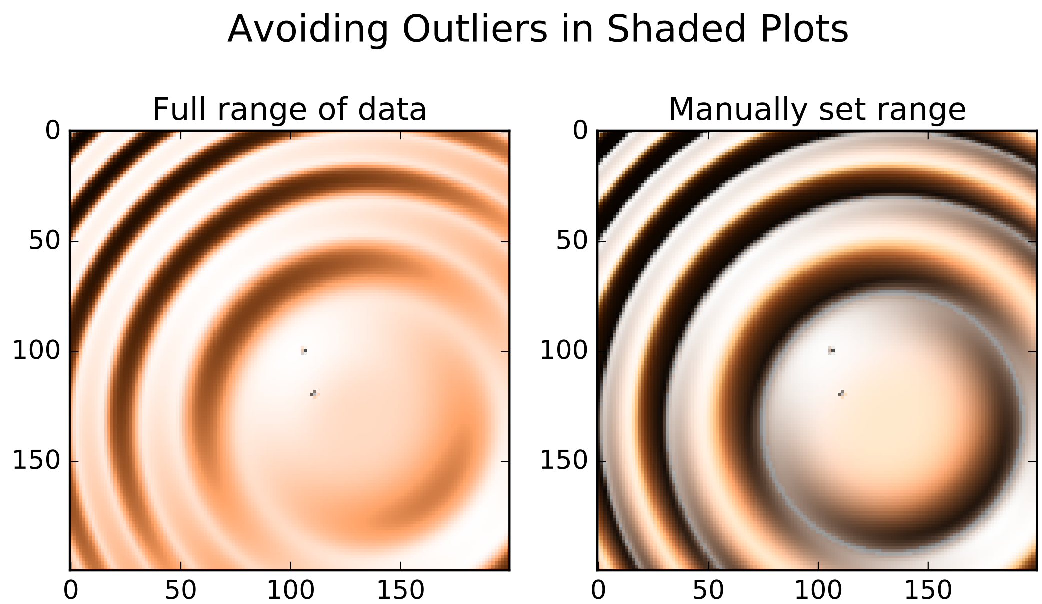
CHAPTER

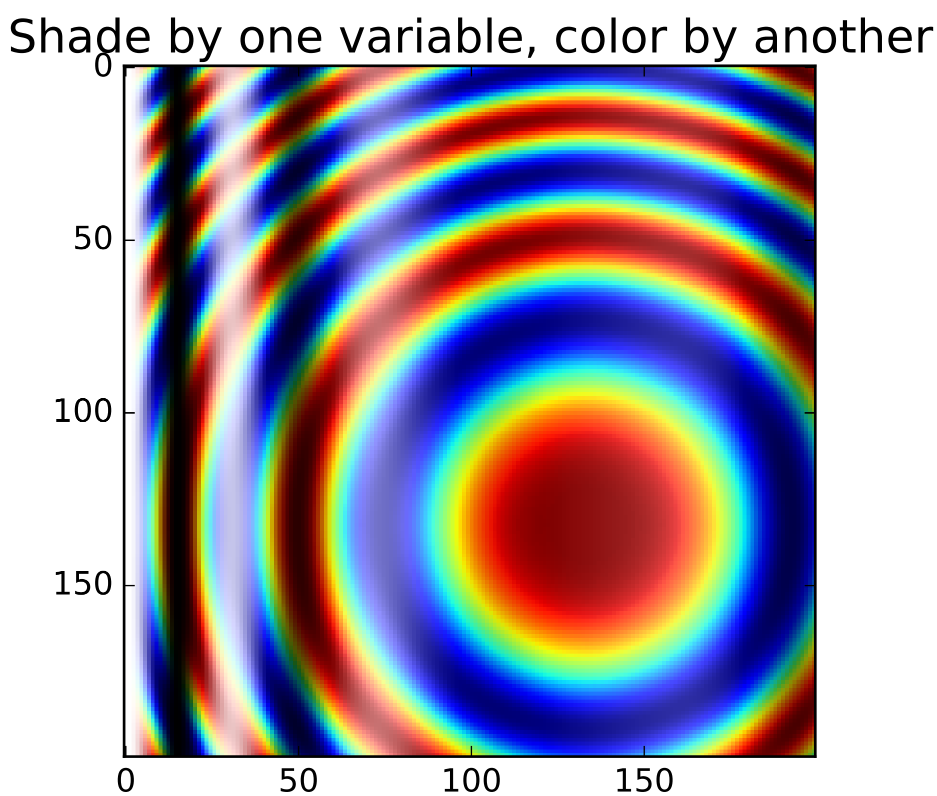
NINETYFOUR

SPECIALTY_PLOTS EXAMPLES

94.1 specialty_plots example code: advanced_hillshading.py







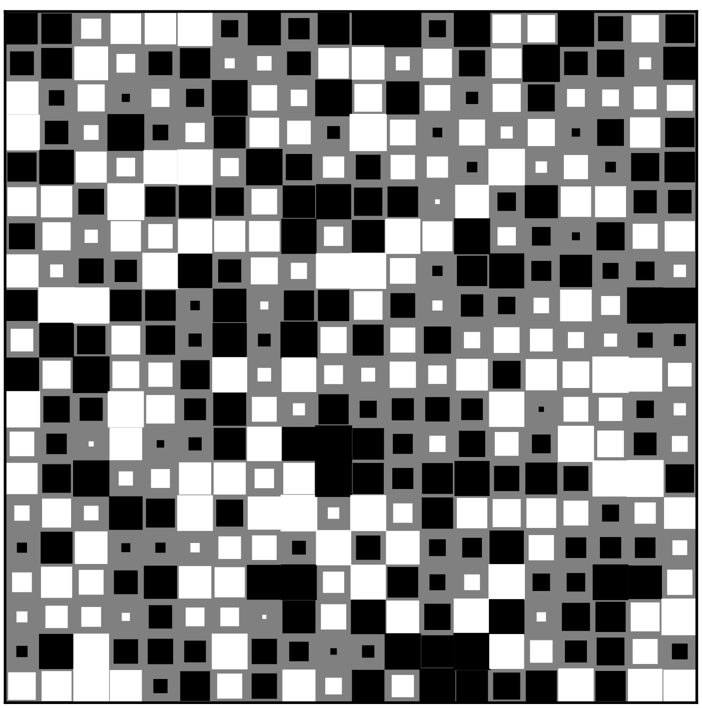

```python
"""
Demonstrates a few common tricks with shaded plots.
"""
```

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LightSource, Normalize


def display_colorbar():
    """Display a correct numeric colorbar for a shaded plot."""
    y, x = np.mgrid[-4:2:200j, -4:2:200j]
    z = 10 * np.cos(x**2 + y**2)

    cmap = plt.cm.copper
    ls = LightSource(315, 45)
    rgb = ls.shade(z, cmap)

    fig, ax = plt.subplots()
    ax.imshow(rgb)

    # Use a proxy artist for the colorbar...
    im = ax.imshow(z, cmap=cmap)
    im.remove()
    fig.colorbar(im)

    ax.set_title('Using a colorbar with a shaded plot', size='x-large')


def avoid_outliers():
    """Use a custom norm to control the displayed z-range of a shaded plot."""
    y, x = np.mgrid[-4:2:200j, -4:2:200j]
    z = 10 * np.cos(x**2 + y**2)

    # Add some outliers...
    z[100, 105] = 2000
    z[120, 110] = -9000

    ls = LightSource(315, 45)
    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 4.5))

    rgb = ls.shade(z, plt.cm.copper)
    ax1.imshow(rgb)
    ax1.set_title('Full range of data')

    rgb = ls.shade(z, plt.cm.copper, vmin=-10, vmax=10)
    ax2.imshow(rgb)
    ax2.set_title('Manually set range')

    fig.suptitle('Avoiding Outliers in Shaded Plots', size='x-large')


def shade_other_data():
    """Demonstrates displaying different variables through shade and color."""
    y, x = np.mgrid[-4:2:200j, -4:2:200j]
    z1 = np.sin(x**2)  # Data to hillshade
    z2 = np.cos(x**2 + y**2)  # Data to color
```

```
    norm = Normalize(z2.min(), z2.max())
    cmap = plt.cm.jet

    ls = LightSource(315, 45)
    rgb = ls.shade_rgb(cmap(norm(z2)), z1)

    fig, ax = plt.subplots()
    ax.imshow(rgb)
    ax.set_title('Shade by one variable, color by another', size='x-large')

display_colorbar()
avoid_outliers()
shade_other_data()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

94.2 specialty_plots example code: hinton_demo.py

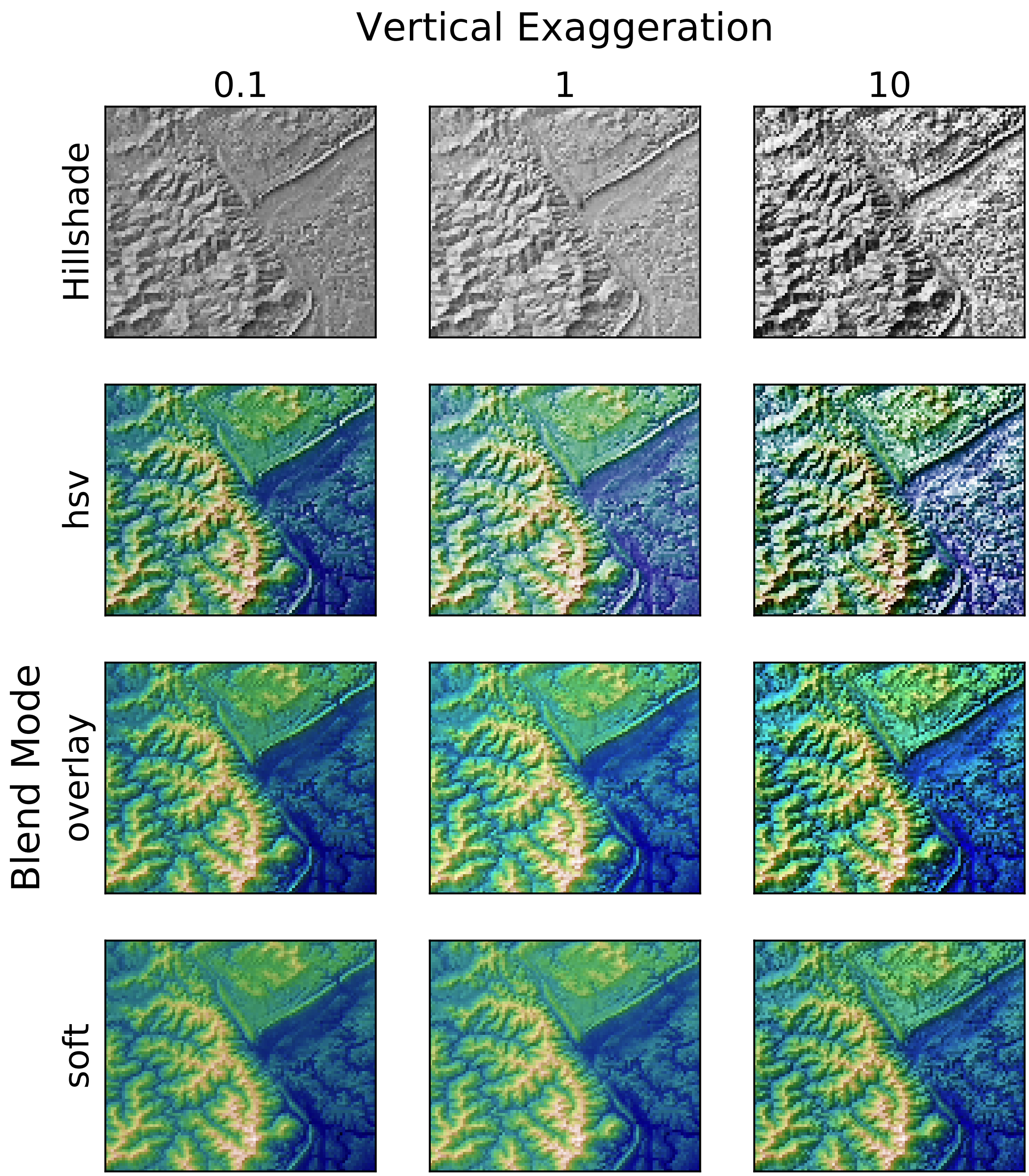


```
"""
Demo of a function to create Hinton diagrams.
```

```
Hinton diagrams are useful for visualizing the values of a 2D array (e.g.
a weight matrix): Positive and negative values are represented by white and
black squares, respectively, and the size of each square represents the
magnitude of each value.

Initial idea from David Warde-Farley on the SciPy Cookbook
"""
import numpy as np
import matplotlib.pyplot as plt


def hinton(matrix, max_weight=None, ax=None):
    """Draw Hinton diagram for visualizing a weight matrix."""
    ax = ax if ax is not None else plt.gca()

    if not max_weight:
        max_weight = 2 ** np.ceil(np.log(np.abs(matrix).max()) / np.log(2))

    ax.patch.set_facecolor('gray')
    ax.set_aspect('equal', 'box')
    ax.xaxis.set_major_locator(plt.NullLocator())
    ax.yaxis.set_major_locator(plt.NullLocator())

    for (x, y), w in np.ndenumerate(matrix):
        color = 'white' if w > 0 else 'black'
        size = np.sqrt(np.abs(w) / max_weight)
        rect = plt.Rectangle([x - size / 2, y - size / 2], size, size,
                             facecolor=color, edgecolor=color)
        ax.add_patch(rect)

    ax.autoscale_view()
    ax.invert_yaxis()


if __name__ == '__main__':
    hinton(np.random.rand(20, 20) - 0.5)
    plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

94.3 specialty_plots example code: topographic_hillshading.py




```
"""
Demonstrates the visual effect of varying blend mode and vertical exaggeration
on "hillshaded" plots.

Note that the "overlay" and "soft" blend modes work well for complex surfaces
such as this example, while the default "hsv" blend mode works best for smooth
```

```
surfaces such as many mathematical functions.

In most cases, hillshading is used purely for visual purposes, and *dx*/*dy*
can be safely ignored. In that case, you can tweak *vert_exag* (vertical
exaggeration) by trial and error to give the desired visual effect. However,
this example demonstrates how to use the *dx* and *dy* kwargs to ensure that
the *vert_exag* parameter is the true vertical exaggeration.
"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.cbook import get_sample_data
from matplotlib.colors import LightSource

dem = np.load(get_sample_data('jacksboro_fault_dem.npz'))
z = dem['elevation']

#-- Optional dx and dy for accurate vertical exaggeration --------------------
# If you need topographically accurate vertical exaggeration, or you don't want
# to guess at what *vert_exag* should be, you'll need to specify the cellsize
# of the grid (i.e. the *dx* and *dy* parameters).  Otherwise, any *vert_exag*
# value you specify will be realitive to the grid spacing of your input data
# (in other words, *dx* and *dy* default to 1.0, and *vert_exag* is calculated
# relative to those parameters).  Similarly, *dx* and *dy* are assumed to be in
# the same units as your input z-values.  Therefore, we'll need to convert the
# given dx and dy from decimal degrees to meters.
dx, dy = dem['dx'], dem['dy']
dy = 111200 * dy
dx = 111200 * dx * np.cos(np.radians(dem['ymin']))
#-----------------------------------------------------------------------------

# Shade from the northwest, with the sun 45 degrees from horizontal
ls = LightSource(azdeg=315, altdeg=45)
cmap = plt.cm.gist_earth

fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(8, 9))
plt.setp(axes.flat, xticks=[], yticks=[])

# Vary vertical exaggeration and blend mode and plot all combinations
for col, ve in zip(axes.T, [0.1, 1, 10]):
    # Show the hillshade intensity image in the first row
    col[0].imshow(ls.hillshade(z, vert_exag=ve, dx=dx, dy=dy), cmap='gray')

    # Place hillshaded plots with different blend modes in the rest of the rows
    for ax, mode in zip(col[1:], ['hsv', 'overlay', 'soft']):
        rgb = ls.shade(z, cmap=cmap, blend_mode=mode,
                       vert_exag=ve, dx=dx, dy=dy)
        ax.imshow(rgb)

# Label rows and columns
for ax, ve in zip(axes[0], [0.1, 1, 10]):
    ax.set_title('{0}'.format(ve), size=18)
for ax, mode in zip(axes[:, 0], ['Hillshade', 'hsv', 'overlay', 'soft']):
    ax.set_ylabel(mode, size=18)
```

```
# Group labels...
axes[0, 1].annotate('Vertical Exaggeration', (0.5, 1), xytext=(0, 30),
                    textcoords='offset points', xycoords='axes fraction',
                    ha='center', va='bottom', size=20)
axes[2, 0].annotate('Blend Mode', (0, 0.5), xytext=(-30, 0),
                    textcoords='offset points', xycoords='axes fraction',
                    ha='right', va='center', size=20, rotation=90)
fig.subplots_adjust(bottom=0.05, right=0.95)

plt.show()
```

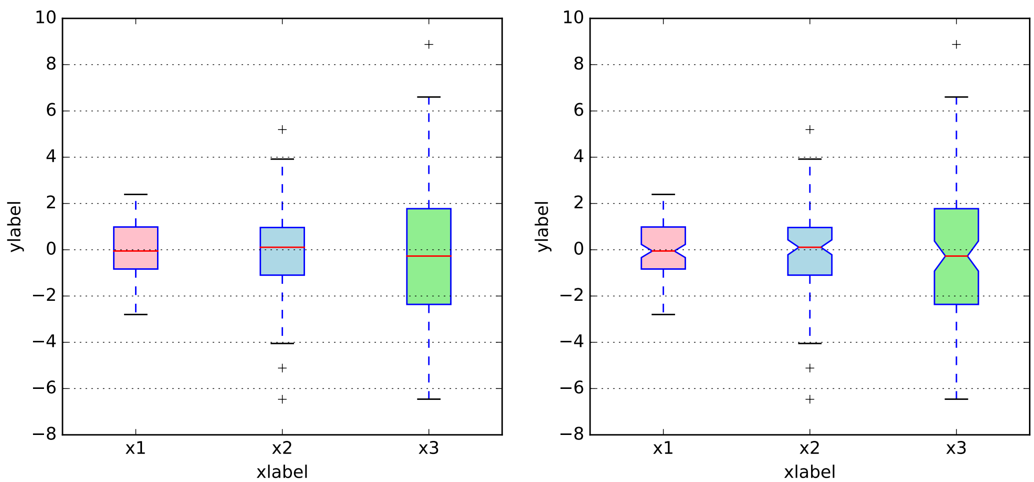
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

NINETYFIVE

STATISTICS EXAMPLES

95.1 statistics example code: boxplot_color_demo.py

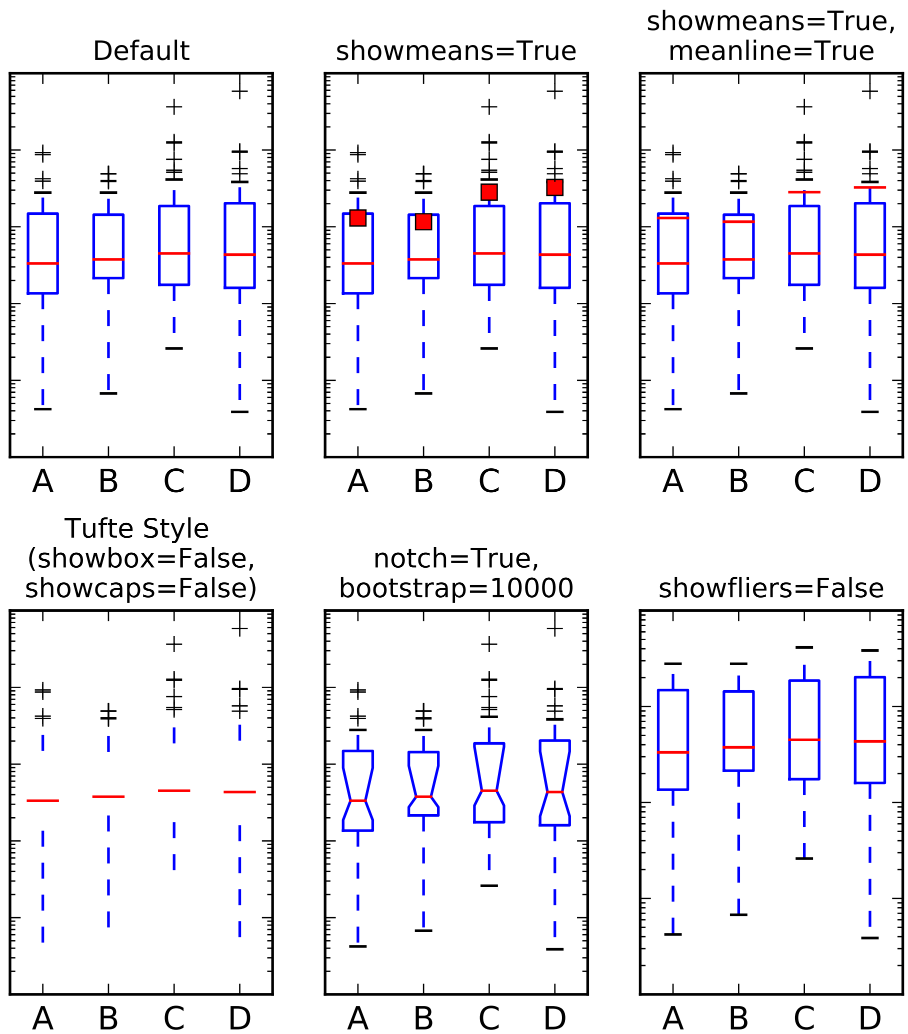


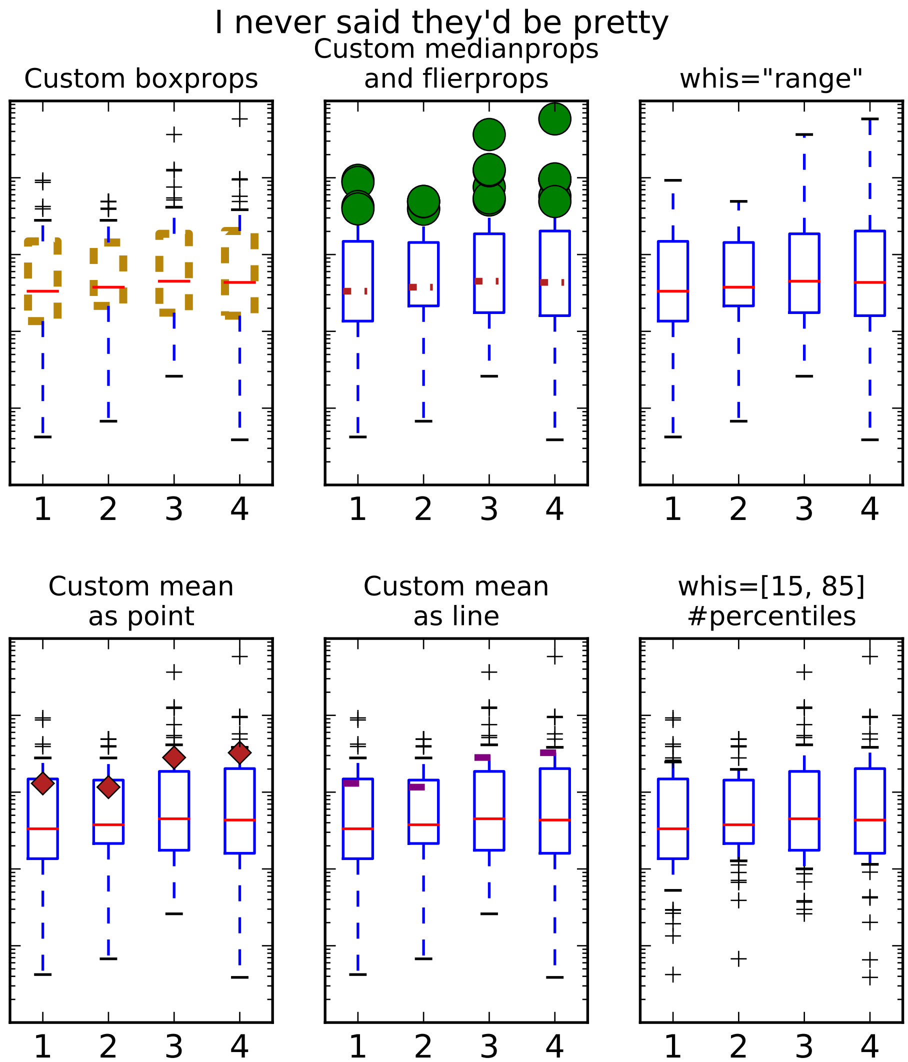

```
# Box plots with custom fill colors

import matplotlib.pyplot as plt
import numpy as np

# Random test data
np.random.seed(123)
all_data = [np.random.normal(0, std, 100) for std in range(1, 4)]

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# rectangular box plot
bplot1 = axes[0].boxplot(all_data,
                         vert=True,   # vertical box alignment
                         patch_artist=True)   # fill with color

# notch shape box plot
bplot2 = axes[1].boxplot(all_data,
                         notch=True,  # notch shape
                         vert=True,   # vertical box alignment
```

```
                            patch_artist=True)   # fill with color

# fill with colors
colors = ['pink', 'lightblue', 'lightgreen']
for bplot in (bplot1, bplot2):
    for patch, color in zip(bplot['boxes'], colors):
        patch.set_facecolor(color)

# adding horizontal grid lines
for ax in axes:
    ax.yaxis.grid(True)
    ax.set_xticks([y+1 for y in range(len(all_data))], )
    ax.set_xlabel('xlabel')
    ax.set_ylabel('ylabel')

# add x-tick labels
plt.setp(axes, xticks=[y+1 for y in range(len(all_data))],
         xticklabels=['x1', 'x2', 'x3', 'x4'])

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.2 statistics example code: boxplot_demo.py





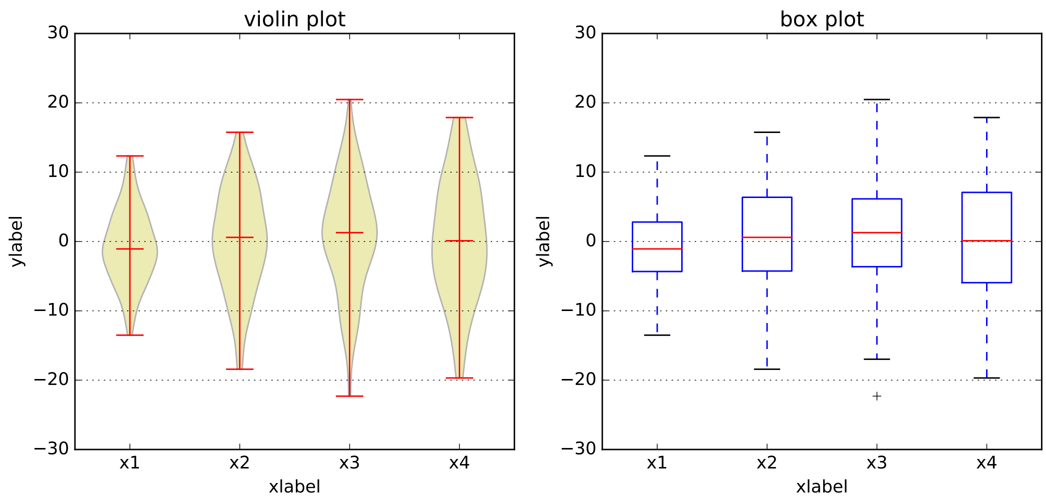

```
"""
Demo of the new boxplot functionality
"""

import numpy as np
import matplotlib.pyplot as plt

# fake data
np.random.seed(937)
data = np.random.lognormal(size=(37, 4), mean=1.5, sigma=1.75)
labels = list('ABCD')
fs = 10  # fontsize

# demonstrate how to toggle the display of different elements:
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6, 6))
axes[0, 0].boxplot(data, labels=labels)
```

```
axes[0, 0].set_title('Default', fontsize=fs)

axes[0, 1].boxplot(data, labels=labels, showmeans=True)
axes[0, 1].set_title('showmeans=True', fontsize=fs)

axes[0, 2].boxplot(data, labels=labels, showmeans=True, meanline=True)
axes[0, 2].set_title('showmeans=True,\nmeanline=True', fontsize=fs)

axes[1, 0].boxplot(data, labels=labels, showbox=False, showcaps=False)
axes[1, 0].set_title('Tufte Style \n(showbox=False,\nshowcaps=False)', fontsize=fs)

axes[1, 1].boxplot(data, labels=labels, notch=True, bootstrap=10000)
axes[1, 1].set_title('notch=True,\nbootstrap=10000', fontsize=fs)

axes[1, 2].boxplot(data, labels=labels, showfliers=False)
axes[1, 2].set_title('showfliers=False', fontsize=fs)

for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.subplots_adjust(hspace=0.4)
plt.show()


# demonstrate how to customize the display different elements:
boxprops = dict(linestyle='--', linewidth=3, color='darkgoldenrod')
flierprops = dict(marker='o', markerfacecolor='green', markersize=12,
                  linestyle='none')
medianprops = dict(linestyle='-.', linewidth=2.5, color='firebrick')
meanpointprops = dict(marker='D', markeredgecolor='black',
                      markerfacecolor='firebrick')
meanlineprops = dict(linestyle='--', linewidth=2.5, color='purple')

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6, 6))
axes[0, 0].boxplot(data, boxprops=boxprops)
axes[0, 0].set_title('Custom boxprops', fontsize=fs)

axes[0, 1].boxplot(data, flierprops=flierprops, medianprops=medianprops)
axes[0, 1].set_title('Custom medianprops\nand flierprops', fontsize=fs)

axes[0, 2].boxplot(data, whis='range')
axes[0, 2].set_title('whis="range"', fontsize=fs)

axes[1, 0].boxplot(data, meanprops=meanpointprops, meanline=False,
                   showmeans=True)
axes[1, 0].set_title('Custom mean\nas point', fontsize=fs)

axes[1, 1].boxplot(data, meanprops=meanlineprops, meanline=True, showmeans=True)
axes[1, 1].set_title('Custom mean\nas line', fontsize=fs)

axes[1, 2].boxplot(data, whis=[15, 85])
axes[1, 2].set_title('whis=[15, 85]\n#percentiles', fontsize=fs)
```

```
for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.suptitle("I never said they'd be pretty")
fig.subplots_adjust(hspace=0.4)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.3 statistics example code: boxplot_vs_violin_demo.py

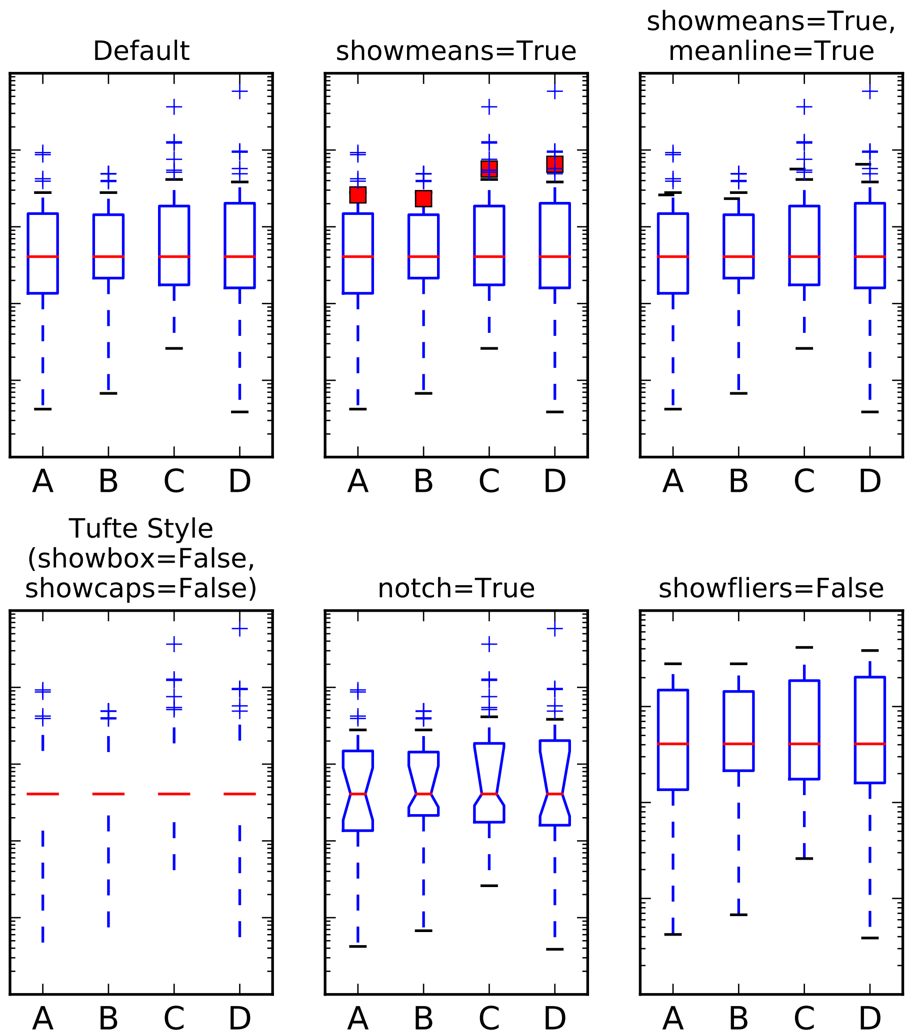


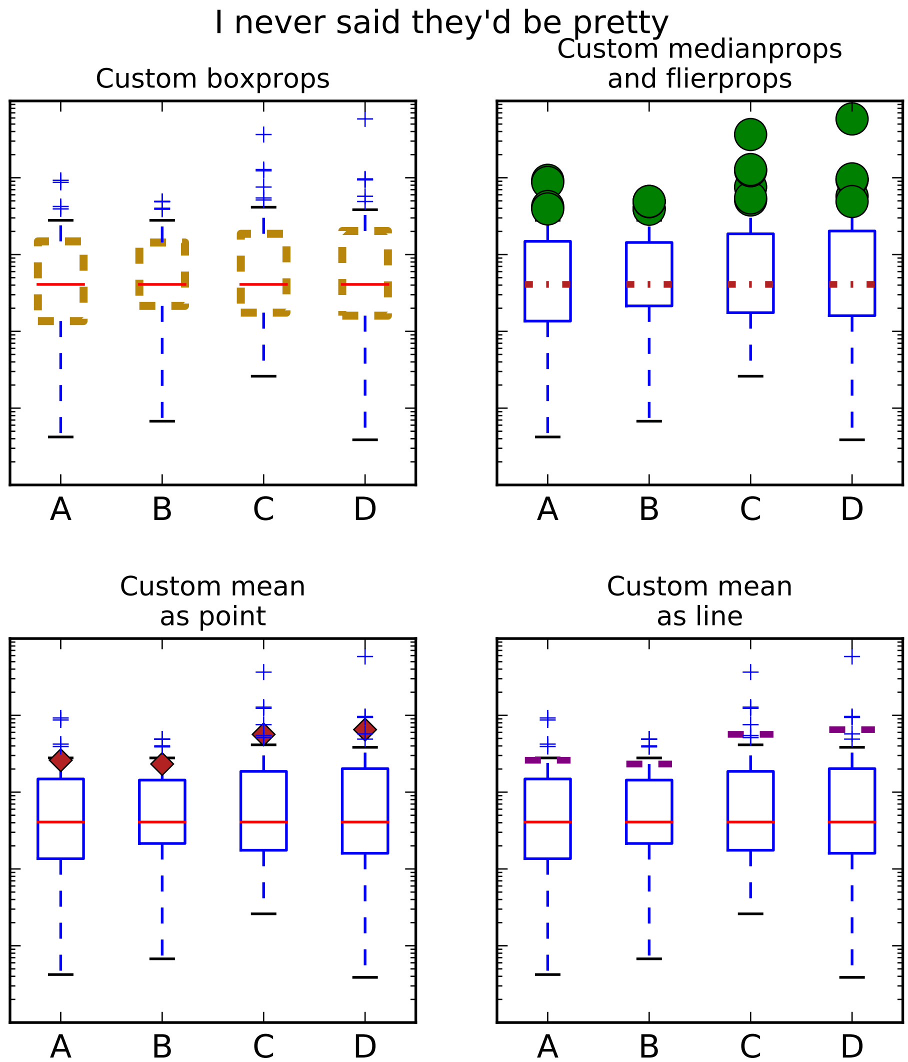

```
# Box plot - violin plot comparison
#
# Note that although violin plots are closely related to Tukey's (1977) box plots,
# they add useful information such as the distribution of the sample data (density␣
↪trace).
#
# By default, box plots show data points outside 1.5 x the inter-quartile range as␣
↪outliers
# above or below the whiskers wheras violin plots show the whole range of the data.
#
# Violin plots require matplotlib >= 1.4.

import matplotlib.pyplot as plt
import numpy as np

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# generate some random test data
all_data = [np.random.normal(0, std, 100) for std in range(6, 10)]

# plot violin plot
```

```
axes[0].violinplot(all_data,
                   showmeans=False,
                   showmedians=True)
axes[0].set_title('violin plot')

# plot box plot
axes[1].boxplot(all_data)
axes[1].set_title('box plot')

# adding horizontal grid lines
for ax in axes:
    ax.yaxis.grid(True)
    ax.set_xticks([y+1 for y in range(len(all_data))])
    ax.set_xlabel('xlabel')
    ax.set_ylabel('ylabel')

# add x-tick labels
plt.setp(axes, xticks=[y+1 for y in range(len(all_data))],
         xticklabels=['x1', 'x2', 'x3', 'x4'])
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.4 statistics example code: bxp_demo.py





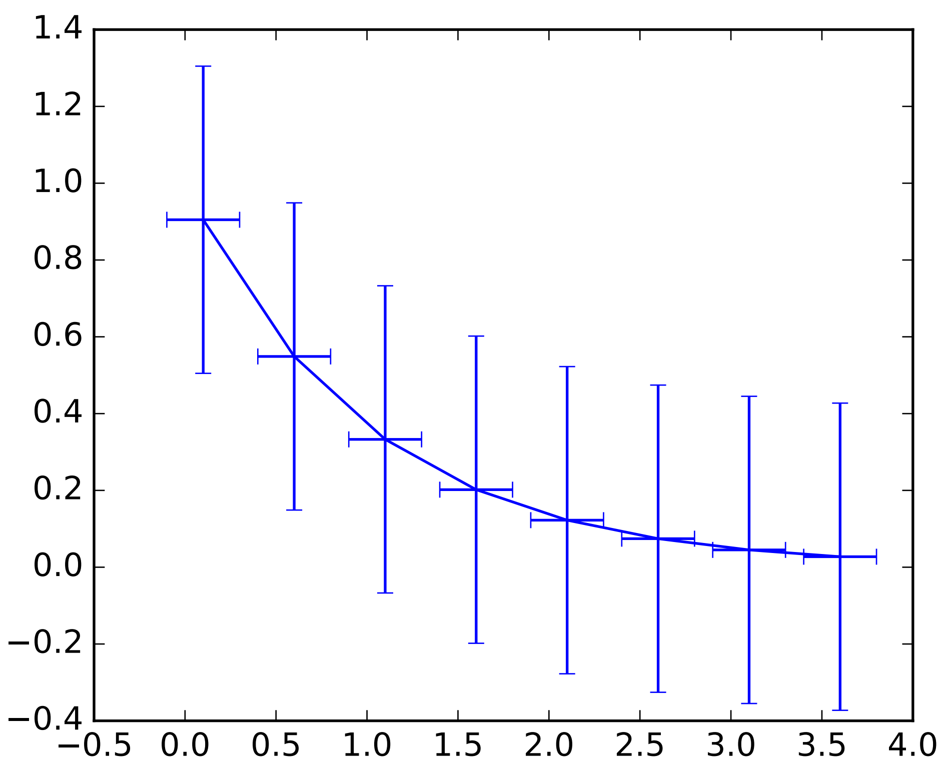

```
"""
Demo of the new boxplot drawer function
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

# fake data
np.random.seed(937)
data = np.random.lognormal(size=(37, 4), mean=1.5, sigma=1.75)
labels = list('ABCD')

# compute the boxplot stats
stats = cbook.boxplot_stats(data, labels=labels, bootstrap=10000)
# After we've computed the stats, we can go through and change anything.
```

```python
# Just to prove it, I'll set the median of each set to the median of all
# the data, and double the means
for n in range(len(stats)):
    stats[n]['med'] = np.median(data)
    stats[n]['mean'] *= 2

print(stats[0].keys())
fs = 10  # fontsize

# demonstrate how to toggle the display of different elements:
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6, 6))
axes[0, 0].bxp(stats)
axes[0, 0].set_title('Default', fontsize=fs)

axes[0, 1].bxp(stats, showmeans=True)
axes[0, 1].set_title('showmeans=True', fontsize=fs)

axes[0, 2].bxp(stats, showmeans=True, meanline=True)
axes[0, 2].set_title('showmeans=True,\nmeanline=True', fontsize=fs)

axes[1, 0].bxp(stats, showbox=False, showcaps=False)
axes[1, 0].set_title('Tufte Style\n(showbox=False,\nshowcaps=False)', fontsize=fs)

axes[1, 1].bxp(stats, shownotches=True)
axes[1, 1].set_title('notch=True', fontsize=fs)

axes[1, 2].bxp(stats, showfliers=False)
axes[1, 2].set_title('showfliers=False', fontsize=fs)

for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.subplots_adjust(hspace=0.4)
plt.show()


# demonstrate how to customize the display different elements:
boxprops = dict(linestyle='--', linewidth=3, color='darkgoldenrod')
flierprops = dict(marker='o', markerfacecolor='green', markersize=12,
                  linestyle='none')
medianprops = dict(linestyle='-.', linewidth=2.5, color='firebrick')
meanpointprops = dict(marker='D', markeredgecolor='black',
                      markerfacecolor='firebrick')
meanlineprops = dict(linestyle='--', linewidth=2.5, color='purple')

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(6, 6))
axes[0, 0].bxp(stats, boxprops=boxprops)
axes[0, 0].set_title('Custom boxprops', fontsize=fs)

axes[0, 1].bxp(stats, flierprops=flierprops, medianprops=medianprops)
axes[0, 1].set_title('Custom medianprops\nand flierprops', fontsize=fs)
```

```
axes[1, 0].bxp(stats, meanprops=meanpointprops, meanline=False,
               showmeans=True)
axes[1, 0].set_title('Custom mean\nas point', fontsize=fs)

axes[1, 1].bxp(stats, meanprops=meanlineprops, meanline=True, showmeans=True)
axes[1, 1].set_title('Custom mean\nas line', fontsize=fs)

for ax in axes.flatten():
    ax.set_yscale('log')
    ax.set_yticklabels([])

fig.suptitle("I never said they'd be pretty")
fig.subplots_adjust(hspace=0.4)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.5 statistics example code: errorbar_demo.py

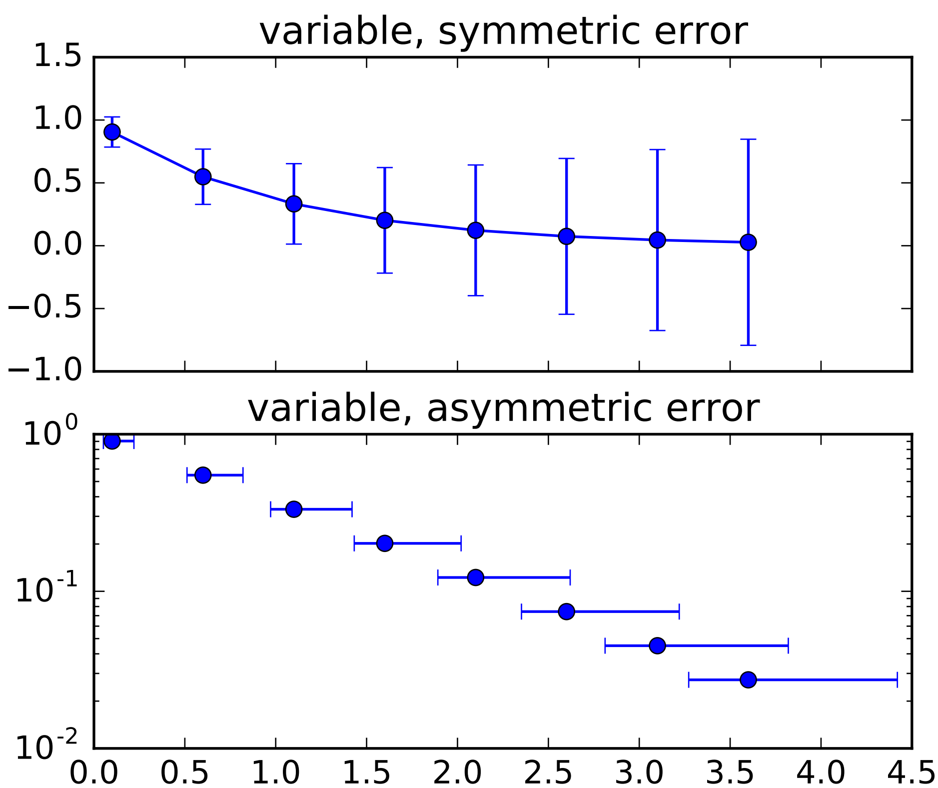



```
"""
Demo of the errorbar function.
"""
```

```
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)

plt.errorbar(x, y, xerr=0.2, yerr=0.4)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.6 statistics example code: errorbar_demo_features.py

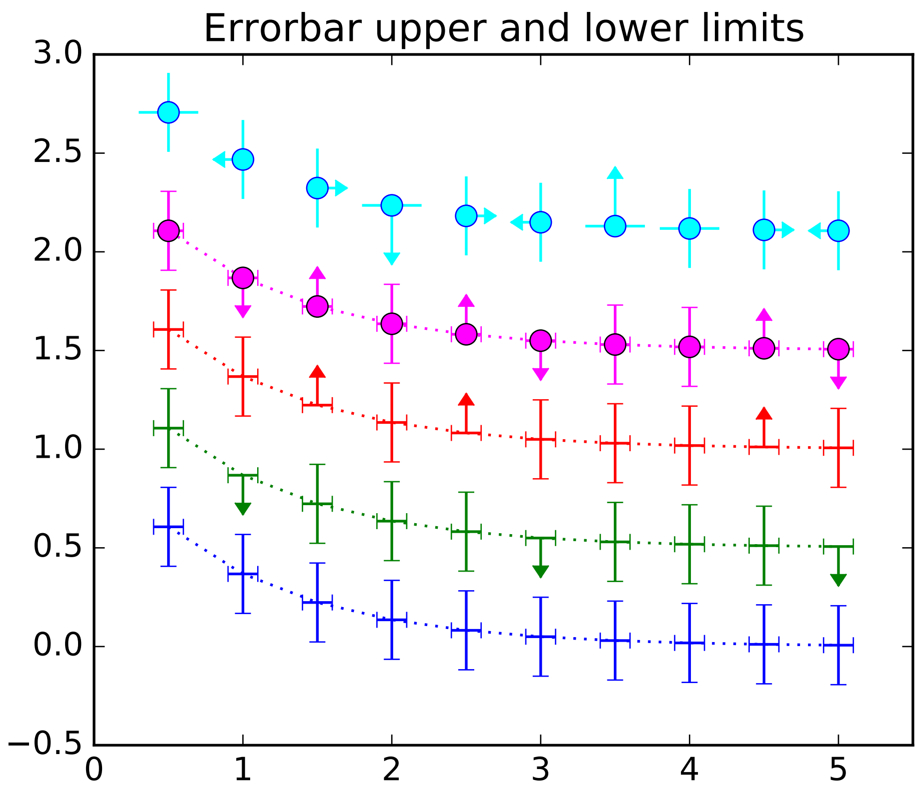



```
"""
Demo of errorbar function with different ways of specifying error bars.

Errors can be specified as a constant value (as shown in `errorbar_demo.py`),
or as demonstrated in this example, they can be specified by an N x 1 or 2 x N,
where N is the number of data points.

N x 1:
```

```
    Error varies for each point, but the error values are symmetric (i.e. the
    lower and upper values are equal).

2 x N:
    Error varies for each point, and the lower and upper limits (in that order)
    are different (asymmetric case)

In addition, this example demonstrates how to use log scale with errorbar.
"""
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)
# example error bar values that vary with x-position
error = 0.1 + 0.2 * x
# error bar values w/ different -/+ errors
lower_error = 0.4 * error
upper_error = error
asymmetric_error = [lower_error, upper_error]

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True)
ax0.errorbar(x, y, yerr=error, fmt='-o')
ax0.set_title('variable, symmetric error')

ax1.errorbar(x, y, xerr=asymmetric_error, fmt='o')
ax1.set_title('variable, asymmetric error')
ax1.set_yscale('log')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.7 statistics example code: errorbar_limits.py

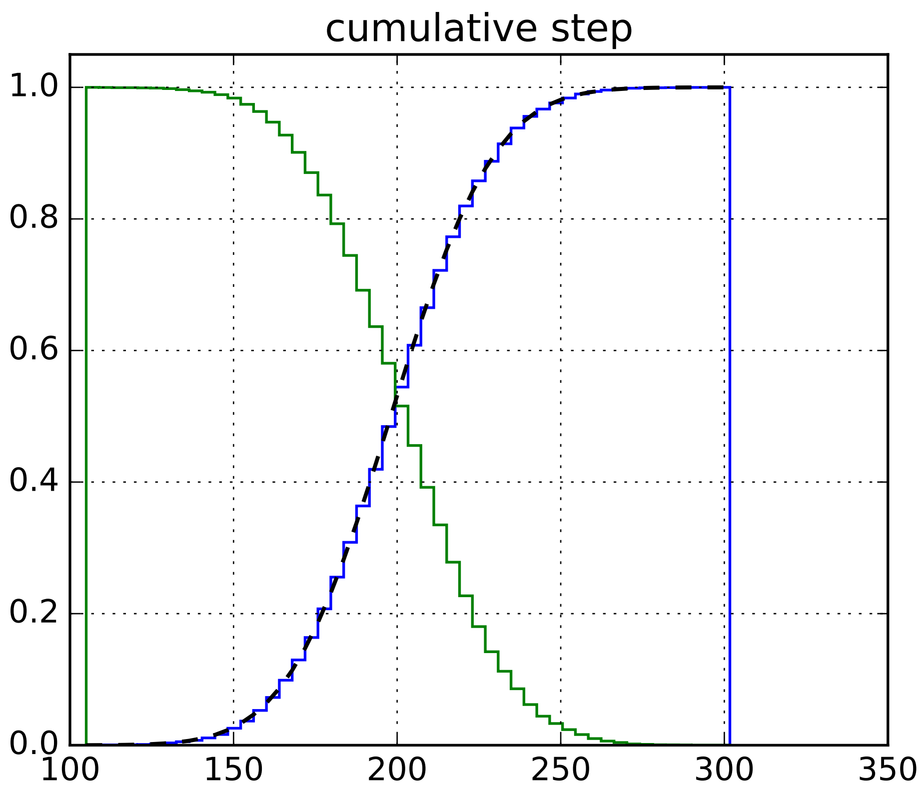



```
"""
Demo of the errorbar function, including upper and lower limits
"""
import numpy as np
import matplotlib.pyplot as plt

# example data
x = np.arange(0.5, 5.5, 0.5)
y = np.exp(-x)
xerr = 0.1
yerr = 0.2
ls = 'dotted'

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

# standard error bars
plt.errorbar(x, y, xerr=xerr, yerr=yerr, ls=ls, color='blue')

# including upper limits
uplims = np.zeros(x.shape)
uplims[[1, 5, 9]] = True
```

```
plt.errorbar(x, y + 0.5, xerr=xerr, yerr=yerr, uplims=uplims, ls=ls,
             color='green')

# including lower limits
lolims = np.zeros(x.shape)
lolims[[2, 4, 8]] = True
plt.errorbar(x, y + 1.0, xerr=xerr, yerr=yerr, lolims=lolims, ls=ls,
             color='red')

# including upper and lower limits
plt.errorbar(x, y + 1.5, marker='o', ms=8, xerr=xerr, yerr=yerr,
             lolims=lolims, uplims=uplims, ls=ls, color='magenta')

# including xlower and xupper limits
xerr = 0.2
yerr = np.zeros(x.shape) + 0.2
yerr[[3, 6]] = 0.3
xlolims = lolims
xuplims = uplims
lolims = np.zeros(x.shape)
uplims = np.zeros(x.shape)
lolims[[6]] = True
uplims[[3]] = True
plt.errorbar(x, y + 2.1, marker='o', ms=8, xerr=xerr, yerr=yerr,
             xlolims=xlolims, xuplims=xuplims, uplims=uplims, lolims=lolims,
             ls='none', mec='blue', capsize=0, color='cyan')

ax.set_xlim((0, 5.5))
ax.set_title('Errorbar upper and lower limits')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.8 statistics example code: histogram_demo_cumulative.py

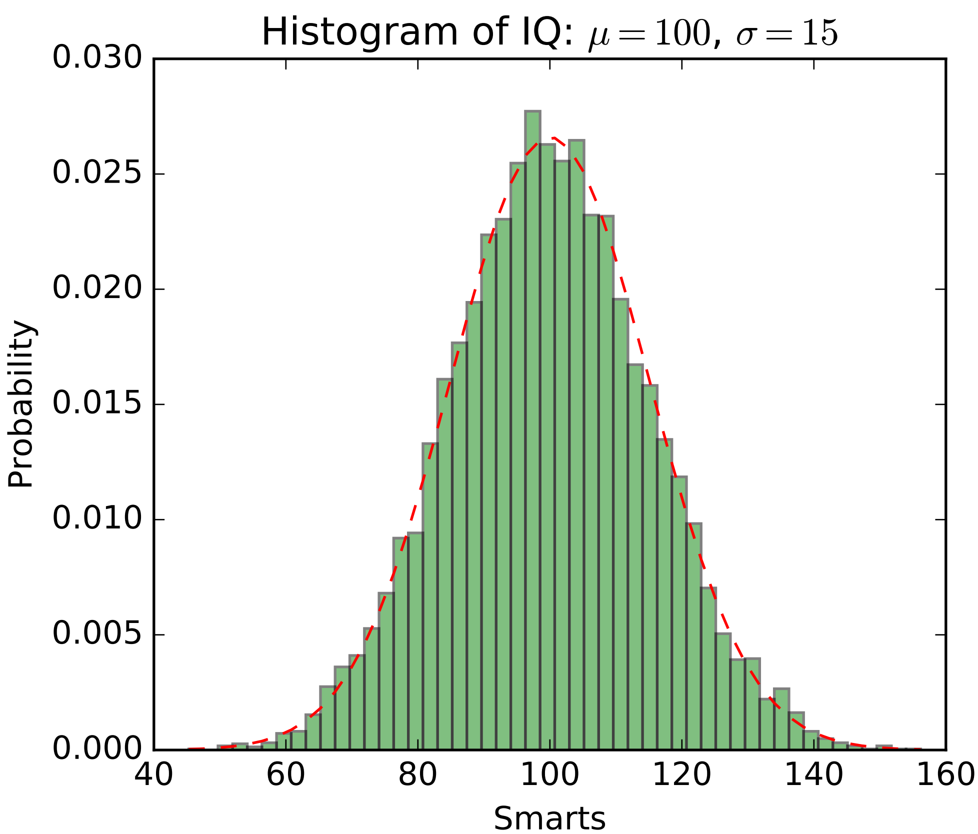



```
"""
Demo of the histogram (hist) function used to plot a cumulative distribution.

"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import mlab


mu = 200
sigma = 25
n_bins = 50
x = mu + sigma*np.random.randn(10000)

n, bins, patches = plt.hist(x, n_bins, normed=1,
                            histtype='step', cumulative=True)

# Add a line showing the expected distribution.
y = mlab.normpdf(bins, mu, sigma).cumsum()
y /= y[-1]
plt.plot(bins, y, 'k--', linewidth=1.5)
```

```
# Overlay a reversed cumulative histogram.
plt.hist(x, bins=bins, normed=1, histtype='step', cumulative=-1)

plt.grid(True)
plt.ylim(0, 1.05)
plt.title('cumulative step')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.9 statistics example code: histogram_demo_features.py

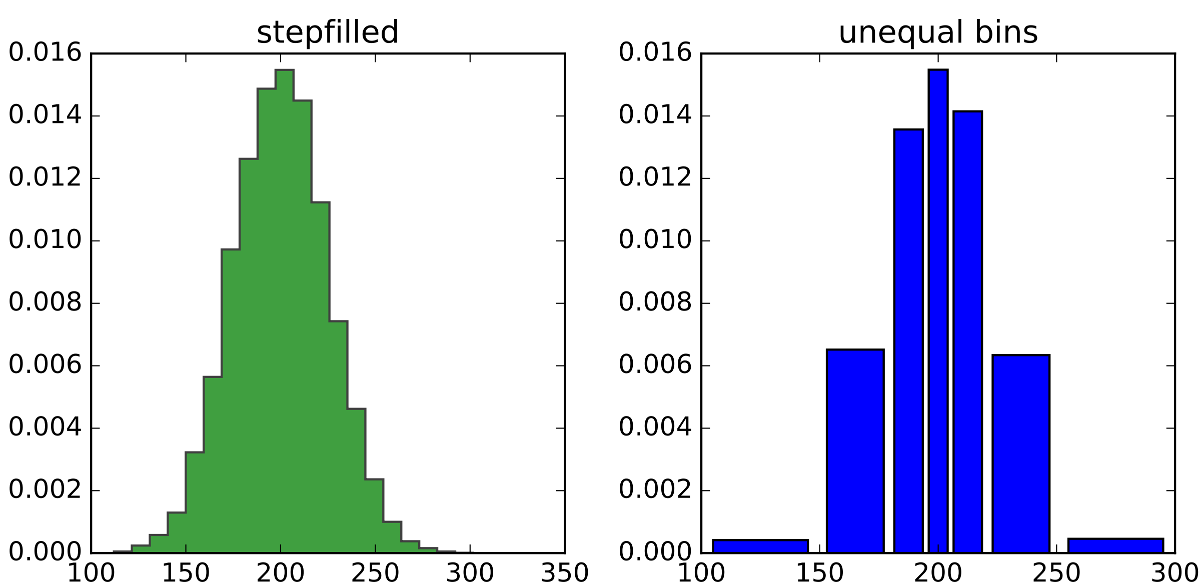



```
"""
Demo of the histogram (hist) function with a few features.

In addition to the basic histogram, this demo shows a few optional features:

    * Setting the number of data bins
    * The ``normed`` flag, which normalizes bin heights so that the integral of
      the histogram is 1. The resulting histogram is a probability density.
    * Setting the face color of the bars
```

```
    * Setting the opacity (alpha value).

"""
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt


# example data
mu = 100  # mean of distribution
sigma = 15  # standard deviation of distribution
x = mu + sigma * np.random.randn(10000)

num_bins = 50
# the histogram of the data
n, bins, patches = plt.hist(x, num_bins, normed=1, facecolor='green', alpha=0.5)
# add a 'best fit' line
y = mlab.normpdf(bins, mu, sigma)
plt.plot(bins, y, 'r--')
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'Histogram of IQ: $\mu=100$, $\sigma=15$')

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

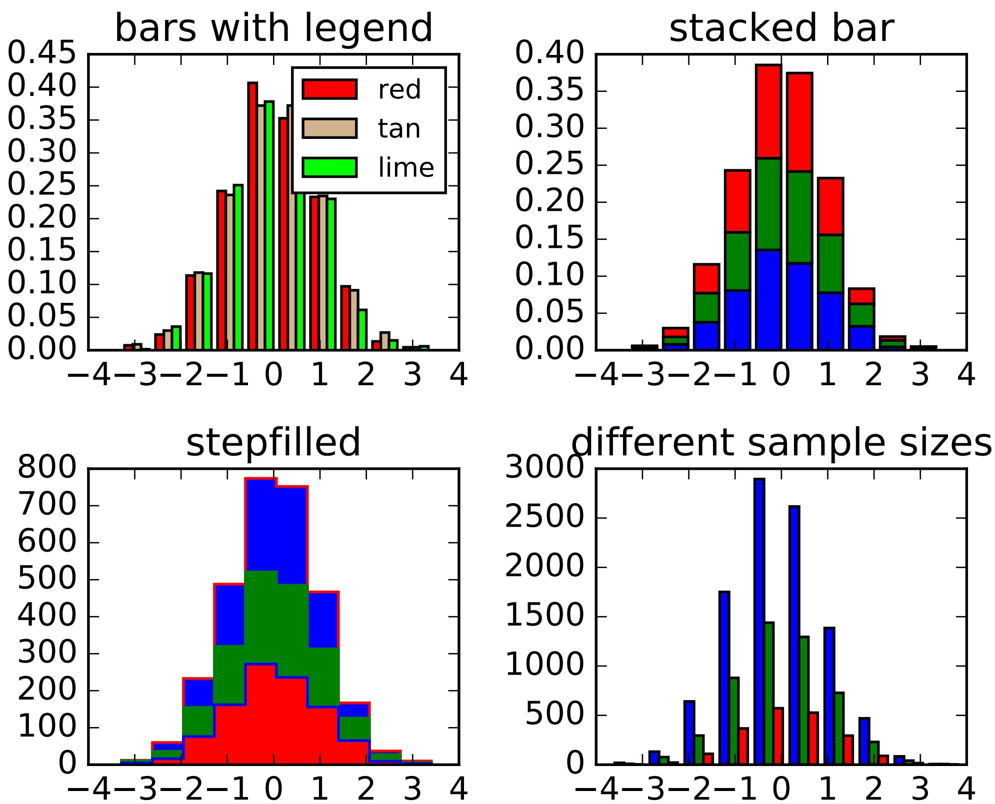
95.10 statistics example code: histogram_demo_histtypes.py



```
"""
Demo of the histogram (hist) function with different ``histtype`` settings.

* Histogram with step curve that has a color fill.
* Histogram with with unequal bin widths.

"""
import numpy as np
import matplotlib.pyplot as plt


mu = 200
sigma = 25
x = mu + sigma*np.random.randn(10000)

fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(8, 4))

ax0.hist(x, 20, normed=1, histtype='stepfilled', facecolor='g', alpha=0.75)
ax0.set_title('stepfilled')

# Create a histogram by providing the bin edges (unequally spaced).
bins = [100, 150, 180, 195, 205, 220, 250, 300]
ax1.hist(x, bins, normed=1, histtype='bar', rwidth=0.8)
ax1.set_title('unequal bins')

plt.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.11 statistics example code: histogram_demo_multihist.py

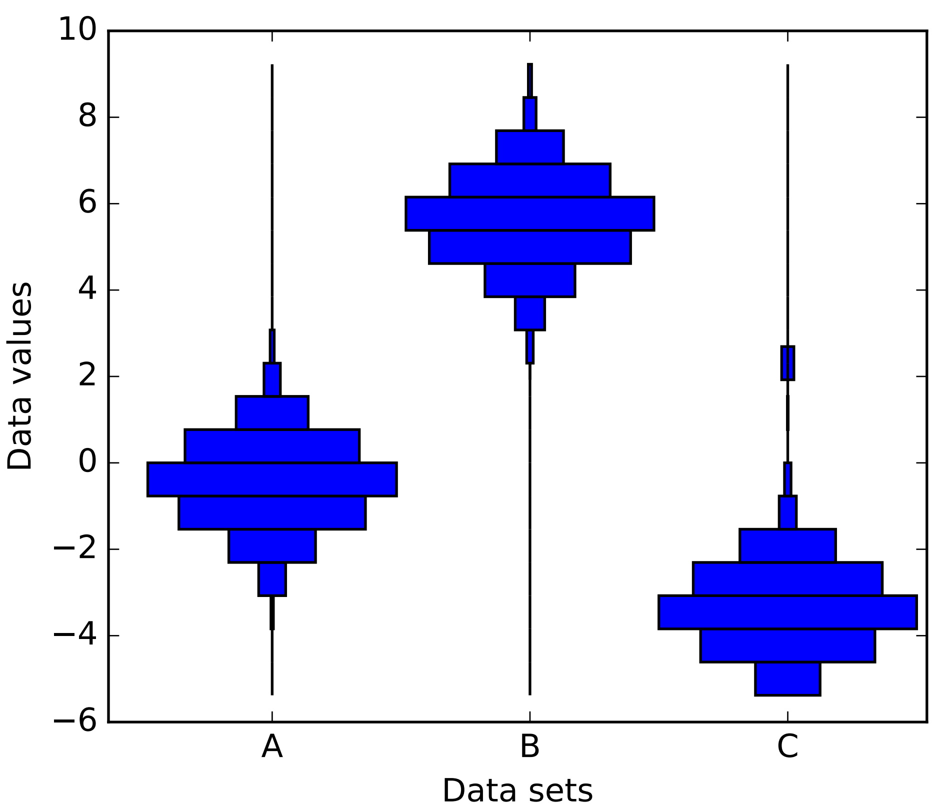



```
"""
Demo of the histogram (hist) function with multiple data sets.

Plot histogram with multiple sample sets and demonstrate:

    * Use of legend with multiple sample sets
    * Stacked bars
    * Step curve with a color fill
    * Data sets of different sample sizes
"""
import numpy as np
import matplotlib.pyplot as plt


n_bins = 10
x = np.random.randn(1000, 3)

fig, axes = plt.subplots(nrows=2, ncols=2)
ax0, ax1, ax2, ax3 = axes.flat

colors = ['red', 'tan', 'lime']
ax0.hist(x, n_bins, normed=1, histtype='bar', color=colors, label=colors)
```

```
ax0.legend(prop={'size': 10})
ax0.set_title('bars with legend')

ax1.hist(x, n_bins, normed=1, histtype='bar', stacked=True)
ax1.set_title('stacked bar')

ax2.hist(x, n_bins, histtype='step', stacked=True, fill=True)
ax2.set_title('stepfilled')

# Make a multiple-histogram of data-sets with different length.
x_multi = [np.random.randn(n) for n in [10000, 5000, 2000]]
ax3.hist(x_multi, n_bins, histtype='bar')
ax3.set_title('different sample sizes')

plt.tight_layout()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

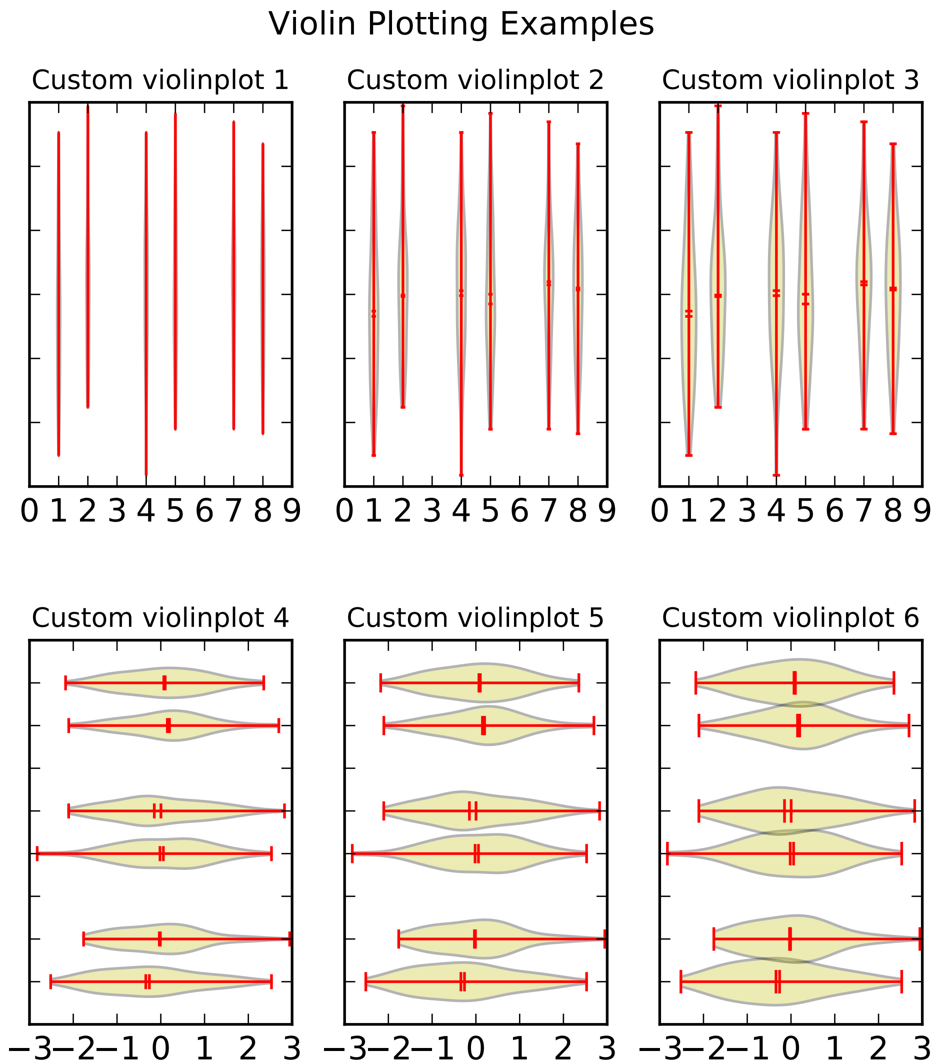
95.12 statistics example code: multiple_histograms_side_by_side.py



```
"""
Demo of how to produce multiple histograms side by side
"""

import numpy as np
import matplotlib.pyplot as plt

number_of_bins = 20

# An example of three data sets to compare
number_of_data_points = 1000
labels = ["A", "B", "C"]
data_sets = [np.random.normal(0, 1, number_of_data_points),
             np.random.normal(6, 1, number_of_data_points),
             np.random.normal(-3, 1, number_of_data_points)]

# Computed quantities to aid plotting
hist_range = (np.min(data_sets), np.max(data_sets))
binned_data_sets = [np.histogram(d, range=hist_range, bins=number_of_bins)[0]
                    for d in data_sets]
binned_maximums = np.max(binned_data_sets, axis=1)
x_locations = np.arange(0, sum(binned_maximums), np.max(binned_maximums))

# The bin_edges are the same for all of the histograms
bin_edges = np.linspace(hist_range[0], hist_range[1], number_of_bins + 1)
centers = .5 * (bin_edges + np.roll(bin_edges, 1))[:-1]
heights = np.diff(bin_edges)

# Cycle through and plot each histogram
ax = plt.subplot(111)
for x_loc, binned_data in zip(x_locations, binned_data_sets):
    lefts = x_loc - .5 * binned_data
    ax.barh(centers, binned_data, height=heights, left=lefts)

ax.set_xticks(x_locations)
ax.set_xticklabels(labels)

ax.set_ylabel("Data values")
ax.set_xlabel("Data sets")

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

95.13 statistics example code: violinplot_demo.py




```
"""
Demo of the new violinplot functionality
"""

import random
import numpy as np
import matplotlib.pyplot as plt

# fake data
fs = 10  # fontsize
pos = [1, 2, 4, 5, 7, 8]
data = [np.random.normal(size=100) for i in pos]
```

```
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(6, 6))

axes[0, 0].violinplot(data, pos, points=20, widths=0.1,
                      showmeans=True, showextrema=True, showmedians=True)
axes[0, 0].set_title('Custom violinplot 1', fontsize=fs)

axes[0, 1].violinplot(data, pos, points=40, widths=0.3,
                      showmeans=True, showextrema=True, showmedians=True,
                      bw_method='silverman')
axes[0, 1].set_title('Custom violinplot 2', fontsize=fs)

axes[0, 2].violinplot(data, pos, points=60, widths=0.5, showmeans=True,
                      showextrema=True, showmedians=True, bw_method=0.5)
axes[0, 2].set_title('Custom violinplot 3', fontsize=fs)

axes[1, 0].violinplot(data, pos, points=80, vert=False, widths=0.7,
                      showmeans=True, showextrema=True, showmedians=True)
axes[1, 0].set_title('Custom violinplot 4', fontsize=fs)

axes[1, 1].violinplot(data, pos, points=100, vert=False, widths=0.9,
                      showmeans=True, showextrema=True, showmedians=True,
                      bw_method='silverman')
axes[1, 1].set_title('Custom violinplot 5', fontsize=fs)

axes[1, 2].violinplot(data, pos, points=200, vert=False, widths=1.1,
                      showmeans=True, showextrema=True, showmedians=True,
                      bw_method=0.5)
axes[1, 2].set_title('Custom violinplot 6', fontsize=fs)

for ax in axes.flatten():
    ax.set_yticklabels([])

fig.suptitle("Violin Plotting Examples")
fig.subplots_adjust(hspace=0.4)
plt.show()
```

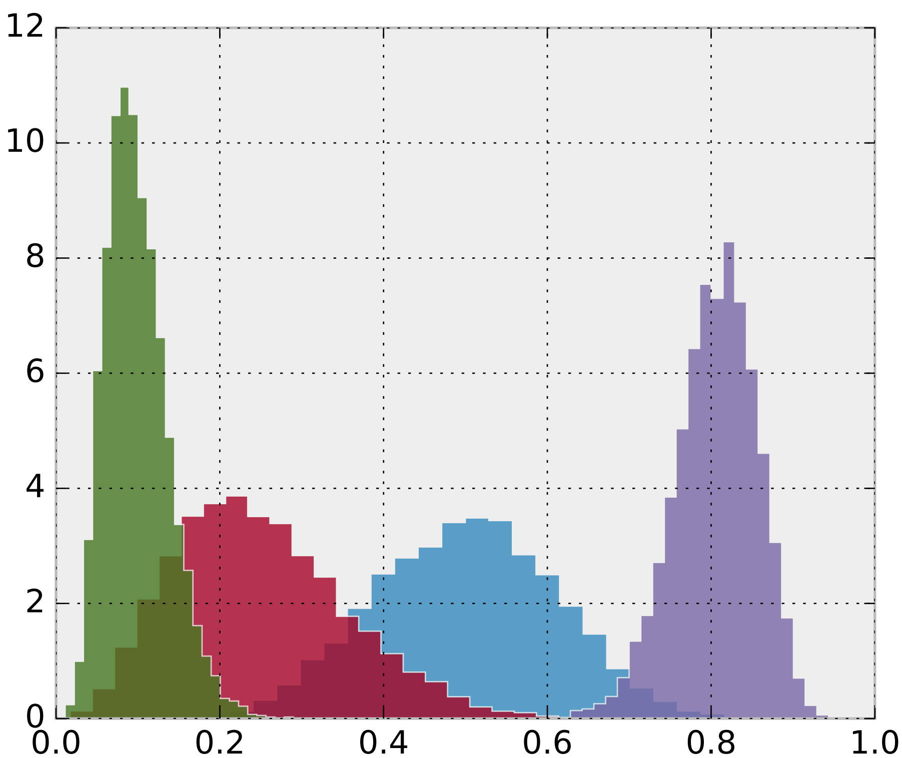
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

NINETYSIX

STYLE_SHEETS EXAMPLES

96.1 style_sheets example code: plot_bmh.py

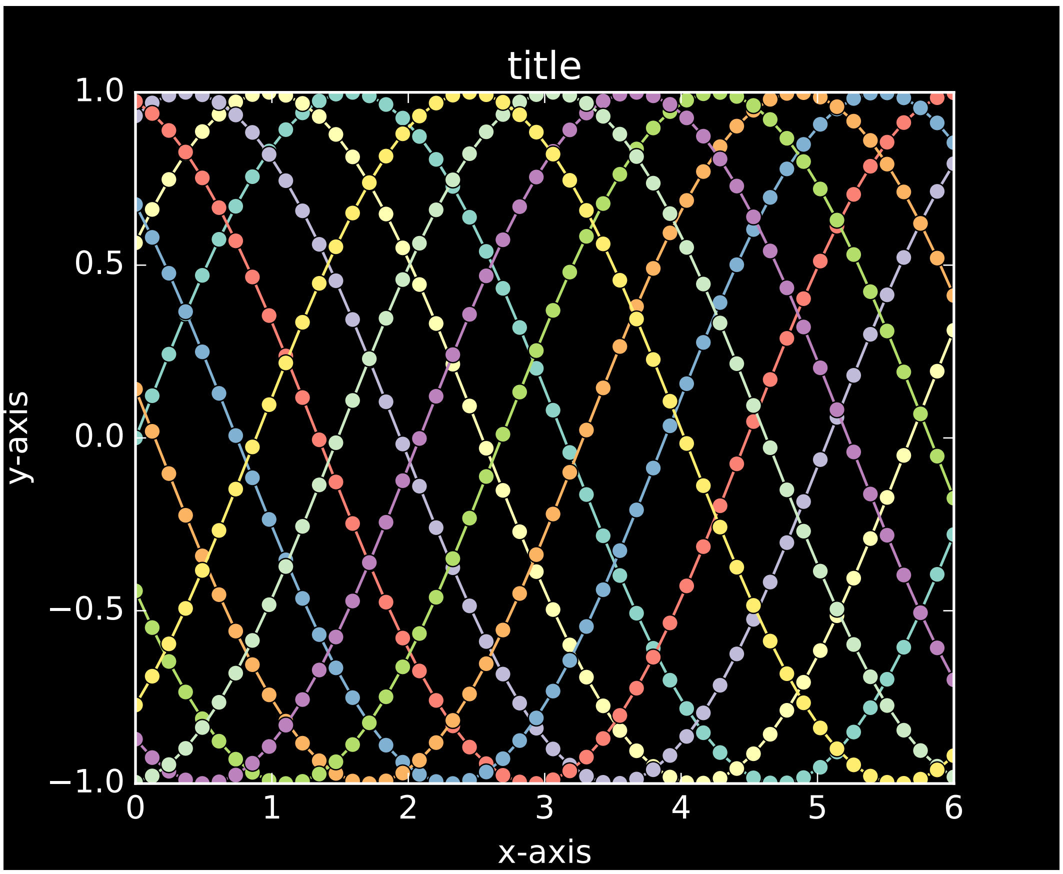



```
"""
This example demonstrates the "bmh" style, which is the design used in the
Bayesian Methods for Hackers online book.
"""
from numpy.random import beta
import matplotlib.pyplot as plt

plt.style.use('bmh')
```

```python
def plot_beta_hist(a, b):
    plt.hist(beta(a, b, size=10000), histtype="stepfilled",
             bins=25, alpha=0.8, normed=True)
    return

plot_beta_hist(10, 10)
plot_beta_hist(4, 12)
plot_beta_hist(50, 12)
plot_beta_hist(6, 55)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

96.2 style_sheets example code: plot_dark_background.py

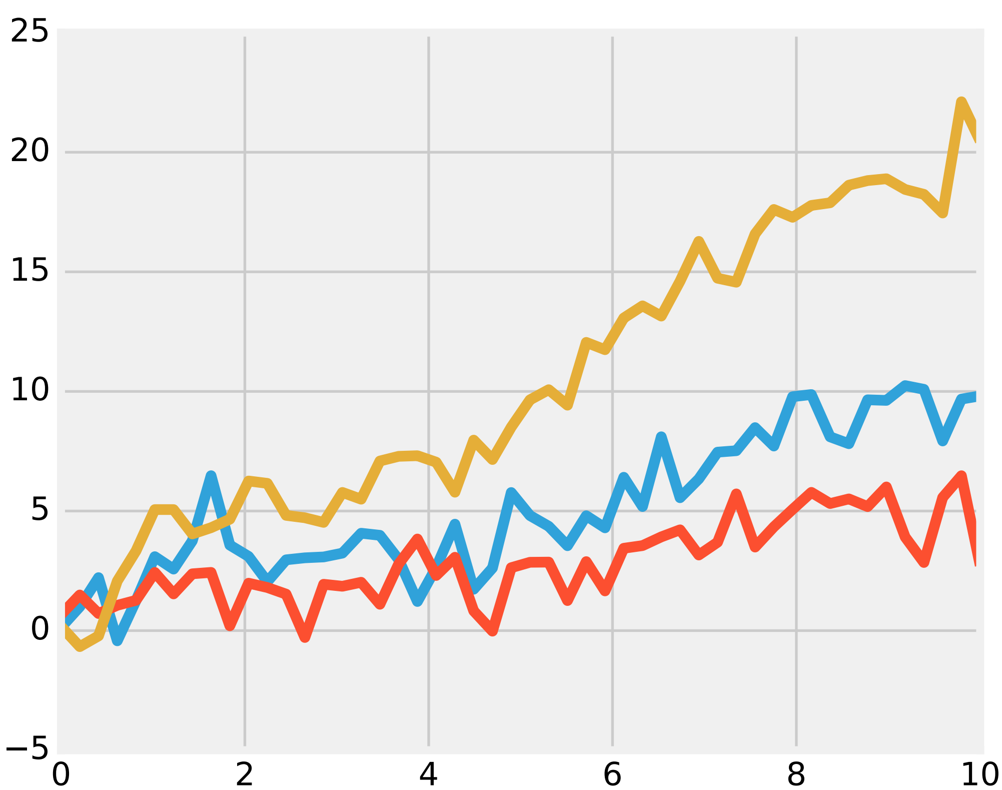



```python
"""
This example demonstrates the "dark_background" style, which uses white for
elements that are typically black (text, borders, etc). Note, however, that not
all plot elements default to colors defined by an rc parameter.
```

```
"""
import numpy as np
import matplotlib.pyplot as plt


plt.style.use('dark_background')

L = 6
x = np.linspace(0, L)
ncolors = len(plt.rcParams['axes.color_cycle'])
shift = np.linspace(0, L, ncolors, endpoint=False)
for s in shift:
    plt.plot(x, np.sin(x + s), 'o-')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('title')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

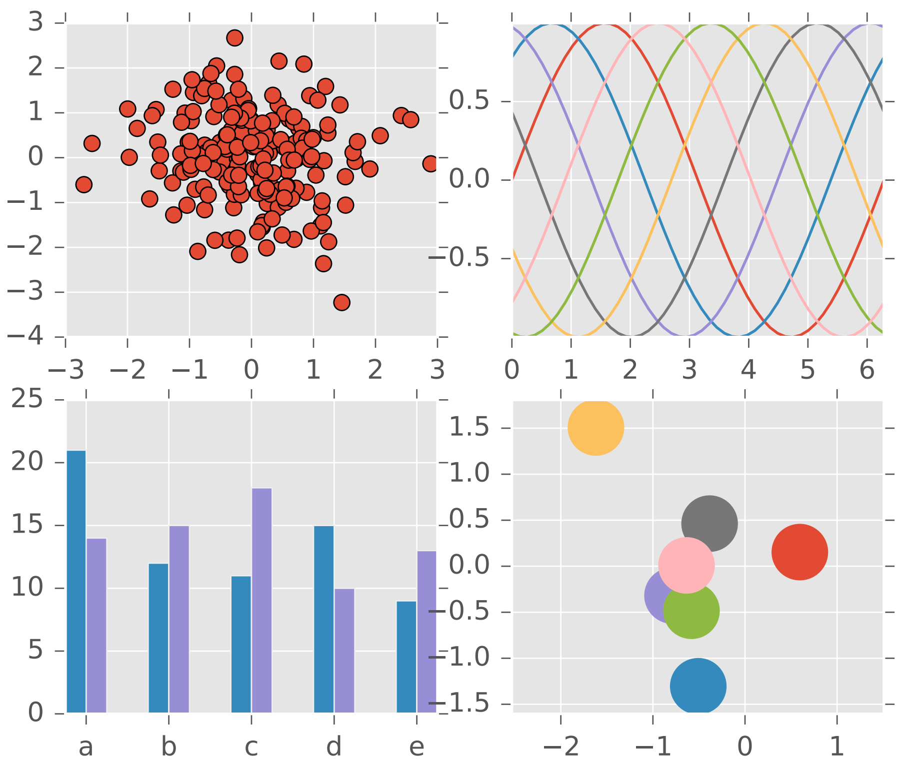
96.3 style_sheets example code: plot_fivethirtyeight.py



```
"""
This shows an example of the "fivethirtyeight" styling, which
tries to replicate the styles from FiveThirtyEight.com.
"""


from matplotlib import pyplot as plt
import numpy as np

x = np.linspace(0, 10)

with plt.style.context('fivethirtyeight'):
    plt.plot(x, np.sin(x) + x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 0.5 * x + np.random.randn(50))
    plt.plot(x, np.sin(x) + 2 * x + np.random.randn(50))


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

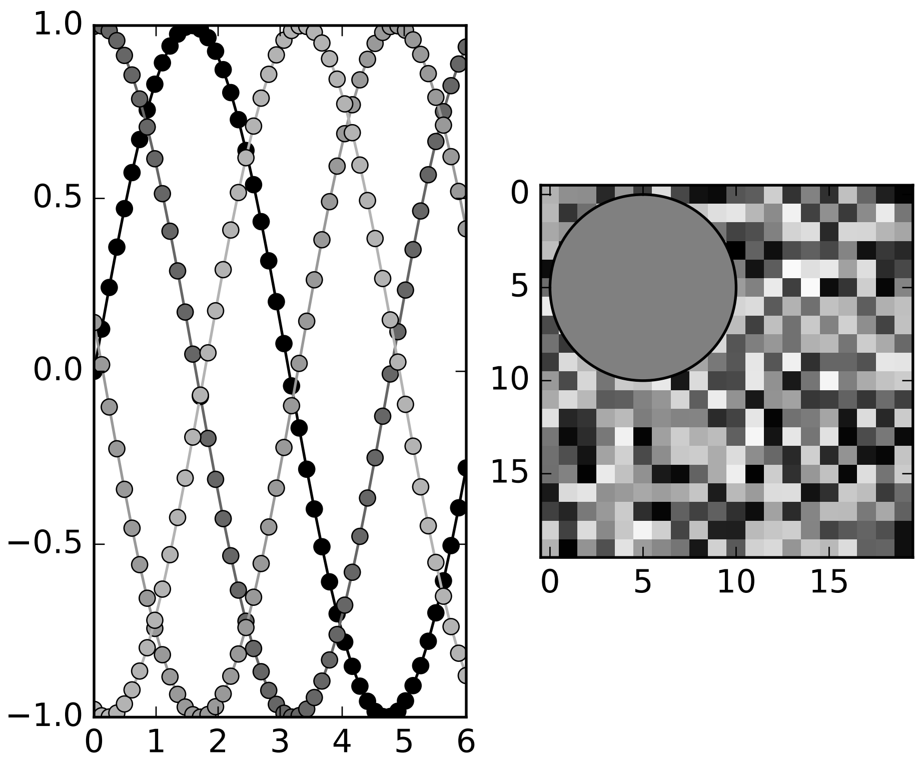
96.4 style_sheets example code: plot_ggplot.py



```
"""
This example demonstrates the "ggplot" style, which adjusts the style to
emulate ggplot_ (a popular plotting package for R_).

These settings were shamelessly stolen from [1]_ (with permission).

.. [1] http://www.huyng.com/posts/sane-color-scheme-for-matplotlib/

.. _ggplot: http://ggplot2.org/
.. _R: https://www.r-project.org/

"""
import numpy as np
import matplotlib.pyplot as plt

plt.style.use('ggplot')

fig, axes = plt.subplots(ncols=2, nrows=2)
ax1, ax2, ax3, ax4 = axes.ravel()

# scatter plot (Note: `plt.scatter` doesn't use default colors)
x, y = np.random.normal(size=(2, 200))
ax1.plot(x, y, 'o')

# sinusoidal lines with colors from default color cycle
L = 2*np.pi
x = np.linspace(0, L)
ncolors = len(plt.rcParams['axes.color_cycle'])
shift = np.linspace(0, L, ncolors, endpoint=False)
for s in shift:
    ax2.plot(x, np.sin(x + s), '-')
ax2.margins(0)

# bar graphs
x = np.arange(5)
y1, y2 = np.random.randint(1, 25, size=(2, 5))
width = 0.25
ax3.bar(x, y1, width)
ax3.bar(x + width, y2, width, color=plt.rcParams['axes.color_cycle'][2])
ax3.set_xticks(x + width)
ax3.set_xticklabels(['a', 'b', 'c', 'd', 'e'])

# circles with colors from default color cycle
for i, color in enumerate(plt.rcParams['axes.color_cycle']):
    xy = np.random.normal(size=2)
    ax4.add_patch(plt.Circle(xy, radius=0.3, color=color))
ax4.axis('equal')
ax4.margins(0)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

96.5 style_sheets example code: plot_grayscale.py




```
"""
This example demonstrates the "grayscale" style sheet, which changes all colors
that are defined as rc parameters to grayscale. Note, however, that not all
plot elements default to colors defined by an rc parameter.

"""
import numpy as np
import matplotlib.pyplot as plt


def color_cycle_example(ax):
    L = 6
    x = np.linspace(0, L)
    ncolors = len(plt.rcParams['axes.color_cycle'])
    shift = np.linspace(0, L, ncolors, endpoint=False)
    for s in shift:
        ax.plot(x, np.sin(x + s), 'o-')


def image_and_patch_example(ax):
    ax.imshow(np.random.random(size=(20, 20)), interpolation='none')
    c = plt.Circle((5, 5), radius=5, label='patch')
```

```
    ax.add_patch(c)


plt.style.use('grayscale')

fig, (ax1, ax2) = plt.subplots(ncols=2)

color_cycle_example(ax1)
image_and_patch_example(ax2)

plt.show()
```

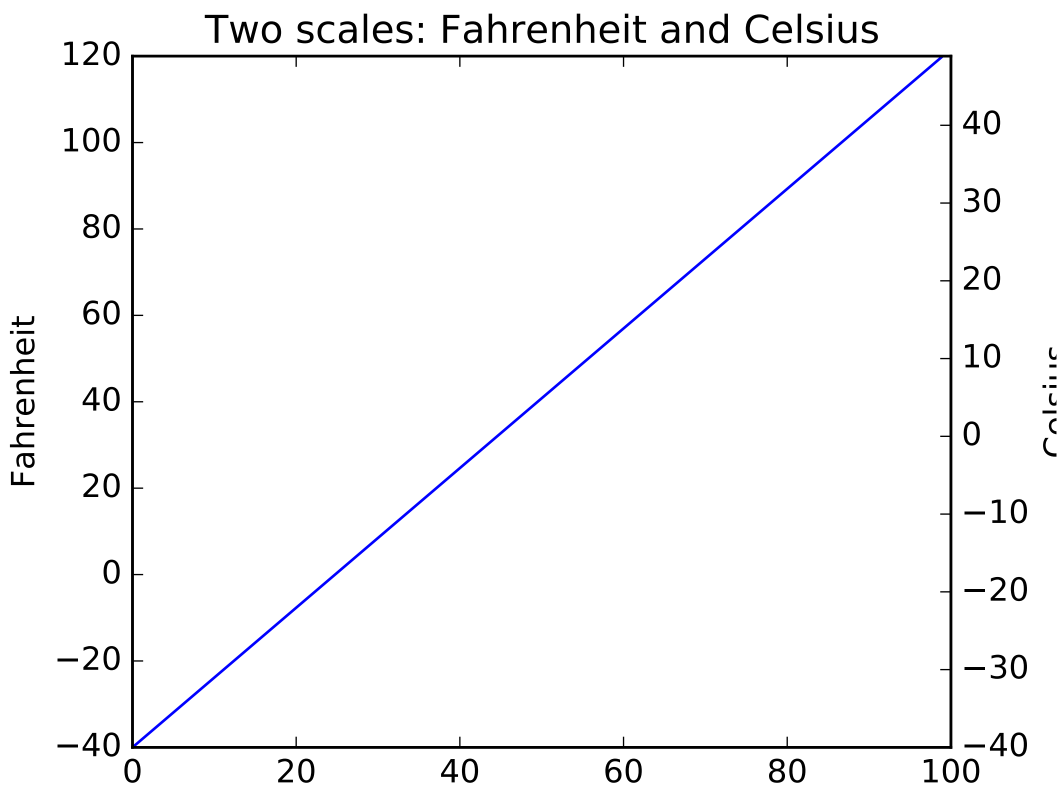
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

NINETYSEVEN

SUBPLOTS_AXES_AND_FIGURES EXAMPLES

97.1 subplots_axes_and_figures example code: fahrenheit_celsius_scales.py

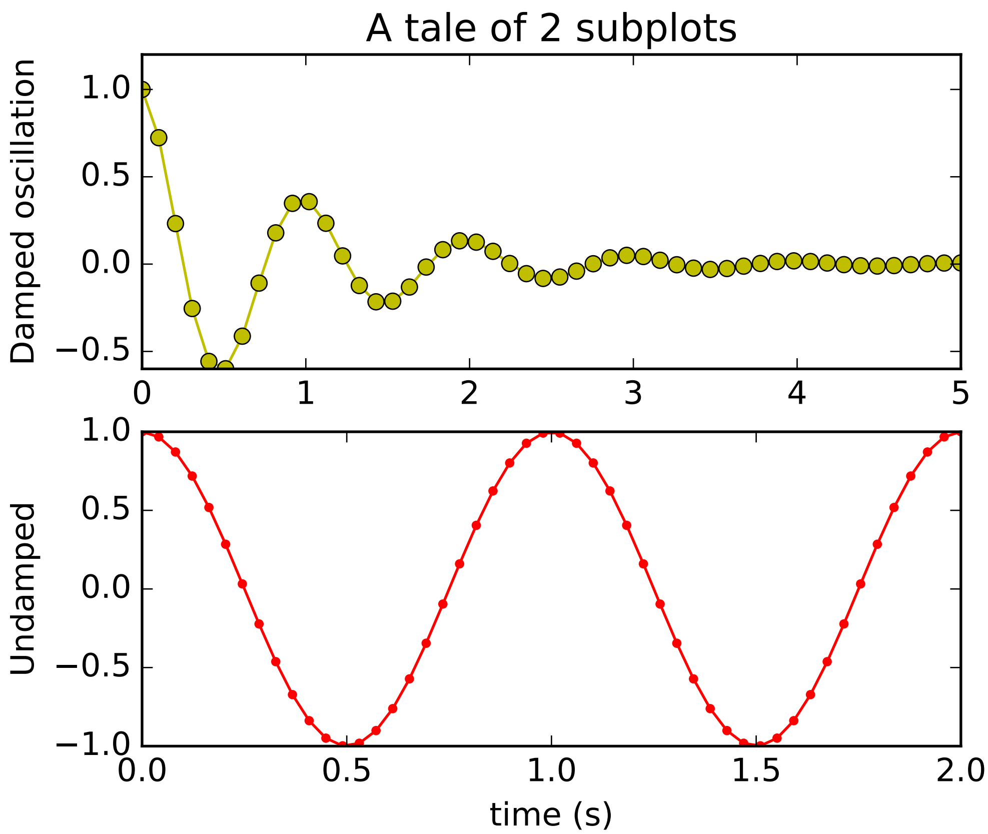



```
"""
Demo of how to display two scales on the left and right y axis.

This example uses the Fahrenheit and Celsius scales.
"""
import matplotlib.pyplot as plt
import numpy as np
```

```
def fahrenheit2celsius(temp):
    """
    Returns temperature in Celsius.
    """
    return (5. / 9.) * (temp - 32)


def convert_ax_c_to_celsius(ax_f):
    """
    Update second axis according with first axis.
    """
    y1, y2 = ax_f.get_ylim()
    ax_c.set_ylim(fahrenheit2celsius(y1), fahrenheit2celsius(y2))
    ax_c.figure.canvas.draw()

fig, ax_f = plt.subplots()
ax_c = ax_f.twinx()

# automatically update ylim of ax2 when ylim of ax1 changes.
ax_f.callbacks.connect("ylim_changed", convert_ax_c_to_celsius)
ax_f.plot(np.linspace(-40, 120, 100))
ax_f.set_xlim(0, 100)

ax_f.set_title('Two scales: Fahrenheit and Celsius')
ax_f.set_ylabel('Fahrenheit')
ax_c.set_ylabel('Celsius')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

97.2 subplots_axes_and_figures example code: subplot_demo.py




```
"""
Simple demo with multiple subplots.
"""
import numpy as np
import matplotlib.pyplot as plt


x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'yo-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER NINETYEIGHT

TESTS EXAMPLES

98.1 tests example code: backend_driver.py

[source code]

```
#!/usr/bin/env python

from __future__ import print_function, division
"""
This is used to drive many of the examples across the backends, for
regression testing, and comparing backend efficiency.

You can specify the backends to be tested either via the --backends
switch, which takes a comma-separated list, or as separate arguments,
e.g.

    python backend_driver.py agg ps

would test the agg and ps backends. If no arguments are given, a
default list of backends will be tested.

Interspersed with the backend arguments can be switches for the Python
interpreter executing the tests. If entering such arguments causes an
option parsing error with the driver script, separate them from driver
switches with a --.
"""

import os
import time
import sys
import glob
from optparse import OptionParser

import matplotlib.rcsetup as rcsetup
from matplotlib.cbook import Bunch, dedent


all_backends = list(rcsetup.all_backends)  # to leave the original list alone

# actual physical directory for each dir
```

```
dirs = dict(files=os.path.join('..', 'lines_bars_and_markers'),
            shapes=os.path.join('..', 'shapes_and_collections'),
            images=os.path.join('..', 'images_contours_and_fields'),
            pie=os.path.join('..', 'pie_and_polar_charts'),
            text=os.path.join('..', 'text_labels_and_annotations'),
            ticks=os.path.join('..', 'ticks_and_spines'),
            subplots=os.path.join('..', 'subplots_axes_and_figures'),
            specialty=os.path.join('..', 'specialty_plots'),
            showcase=os.path.join('..', 'showcase'),
            pylab=os.path.join('..', 'pylab_examples'),
            api=os.path.join('..', 'api'),
            units=os.path.join('..', 'units'),
            mplot3d=os.path.join('..', 'mplot3d'))


# files in each dir
files = dict()

files['lines'] = [
    'barh_demo.py',
    'fill_demo.py',
    'fill_demo_features.py',
    'line_demo_dash_control.py',
    'line_styles_reference.py',
    'scatter_with_legend.py'
    ]

files['shapes'] = [
    'path_patch_demo.py',
    'scatter_demo.py',
    ]

files['colors'] = [
    'color_cycle_demo.py',
    ]

files['images'] = [
    'image_demo.py',
    'contourf_log.py',
    ]

files['statistics'] = [
    'errorbar_demo.py',
    'errorbar_demo_features.py',
    'histogram_demo_cumulative.py',
    'histogram_demo_features.py',
    'histogram_demo_histtypes.py',
    'histogram_demo_multihist.py',
    ]

files['pie'] = [
    'pie_demo.py',
    'polar_bar_demo.py',
```

```
    'polar_scatter_demo.py',
    ]

files['text_labels_and_annotations'] = [
    'text_demo_fontdict.py',
    'unicode_demo.py',
    ]

files['ticks_and_spines'] = [
    'spines_demo_bounds.py',
    'ticklabels_demo_rotation.py',
    ]

files['subplots_axes_and_figures'] = [
    'subplot_demo.py',
    ]

files['showcase'] = [
    'integral_demo.py',
    ]

files['pylab'] = [
    'accented_text.py',
    'alignment_test.py',
    'annotation_demo.py',
    'annotation_demo.py',
    'annotation_demo2.py',
    'annotation_demo2.py',
    'anscombe.py',
    'arctest.py',
    'arrow_demo.py',
    'axes_demo.py',
    'axes_props.py',
    'axhspan_demo.py',
    'axis_equal_demo.py',
    'bar_stacked.py',
    'barb_demo.py',
    'barchart_demo.py',
    'barcode_demo.py',
    'boxplot_demo.py',
    'broken_barh.py',
    'cohere_demo.py',
    'color_by_yvalue.py',
    'color_demo.py',
    'colorbar_tick_labelling_demo.py',
    'contour_demo.py',
    'contour_image.py',
    'contour_label_demo.py',
    'contourf_demo.py',
    'coords_demo.py',
    'coords_report.py',
    'csd_demo.py',
    'cursor_demo.py',
```

```
    'custom_cmap.py',
    'custom_figure_class.py',
    'custom_ticker1.py',
    'customize_rc.py',
    'dashpointlabel.py',
    'date_demo1.py',
    'date_demo2.py',
    'date_demo_convert.py',
    'date_demo_rrule.py',
    'date_index_formatter.py',
    'dolphin.py',
    'ellipse_collection.py',
    'ellipse_demo.py',
    'ellipse_rotated.py',
    'equal_aspect_ratio.py',
    'errorbar_limits.py',
    'fancyarrow_demo.py',
    'fancybox_demo.py',
    'fancybox_demo2.py',
    'fancytextbox_demo.py',
    'figimage_demo.py',
    'figlegend_demo.py',
    'figure_title.py',
    'fill_between_demo.py',
    'fill_spiral.py',
    'finance_demo.py',
    'findobj_demo.py',
    'fonts_demo.py',
    'fonts_demo_kw.py',
    'ganged_plots.py',
    'geo_demo.py',
    'gradient_bar.py',
    'griddata_demo.py',
    'hatch_demo.py',
    'hexbin_demo.py',
    'hexbin_demo2.py',
    'hist_colormapped.py',
    'vline_hline_demo.py',

    'image_clip_path.py',
    'image_demo.py',
    'image_demo2.py',
    'image_interp.py',
    'image_masked.py',
    'image_nonuniform.py',
    'image_origin.py',
    'image_slices_viewer.py',
    'interp_demo.py',
    'invert_axes.py',
    'layer_images.py',
    'legend_demo2.py',
    'legend_demo3.py',
    'line_collection.py',
```

```
    'line_collection2.py',
    'log_bar.py',
    'log_demo.py',
    'log_test.py',
    'major_minor_demo1.py',
    'major_minor_demo2.py',
    'manual_axis.py',
    'masked_demo.py',
    'mathtext_demo.py',
    'mathtext_examples.py',
    'matplotlib_icon.py',
    'matshow.py',
    'mri_demo.py',
    'mri_with_eeg.py',
    'multi_image.py',
    'multiline.py',
    'multiple_figs_demo.py',
    'nan_test.py',
    'newscalarformatter_demo.py',
    'pcolor_demo.py',
    'pcolor_log.py',
    'pcolor_small.py',
    'pie_demo2.py',
    'plotfile_demo.py',
    'polar_demo.py',
    'polar_legend.py',
    'psd_demo.py',
    'psd_demo2.py',
    'psd_demo3.py',
    'quadmesh_demo.py',
    'quiver_demo.py',
    'scatter_custom_symbol.py',
    'scatter_demo2.py',
    'scatter_masked.py',
    'scatter_profile.py',
    'scatter_star_poly.py',
    #'set_and_get.py',
    'shared_axis_across_figures.py',
    'shared_axis_demo.py',
    'simple_plot.py',
    'specgram_demo.py',
    'spine_placement_demo.py',
    'spy_demos.py',
    'stem_plot.py',
    'step_demo.py',
    'stix_fonts_demo.py',
    'stock_demo.py',
    'subplots_adjust.py',
    'symlog_demo.py',
    'table_demo.py',
    'text_handles.py',
    'text_rotation.py',
    'text_rotation_relative_to_line.py',
```

```
    'transoffset.py',
    'xcorr_demo.py',
    'zorder_demo.py',
    ]


files['api'] = [
    'agg_oo.py',
    'barchart_demo.py',
    'bbox_intersect.py',
    'collections_demo.py',
    'colorbar_only.py',
    'custom_projection_example.py',
    'custom_scale_example.py',
    'date_demo.py',
    'date_index_formatter.py',
    'donut_demo.py',
    'font_family_rc.py',
    'image_zcoord.py',
    'joinstyle.py',
    'legend_demo.py',
    'line_with_text.py',
    'logo2.py',
    'mathtext_asarray.py',
    'patch_collection.py',
    'quad_bezier.py',
    'scatter_piecharts.py',
    'span_regions.py',
    'two_scales.py',
    'unicode_minus.py',
    'watermark_image.py',
    'watermark_text.py',
]

files['units'] = [
    'annotate_with_units.py',
    #'artist_tests.py',  # broken, fixme
    'bar_demo2.py',
    #'bar_unit_demo.py', # broken, fixme
    #'ellipse_with_units.py',  # broken, fixme
    'radian_demo.py',
    'units_sample.py',
    #'units_scatter.py', # broken, fixme

    ]

files['mplot3d'] = [
    '2dcollections3d_demo.py',
    'bars3d_demo.py',
    'contour3d_demo.py',
    'contour3d_demo2.py',
    'contourf3d_demo.py',
    'lines3d_demo.py',
```

```
    'polys3d_demo.py',
    'scatter3d_demo.py',
    'surface3d_demo.py',
    'surface3d_demo2.py',
    'text3d_demo.py',
    'wire3d_demo.py',
    ]

# dict from dir to files we know we don't want to test (e.g., examples
# not using pyplot, examples requiring user input, animation examples,
# examples that may only work in certain environs (usetex examples?),
# examples that generate multiple figures

excluded = {
    'pylab': ['__init__.py', 'toggle_images.py', ],
    'units': ['__init__.py', 'date_support.py', ],
}


def report_missing(dir, flist):
    'report the py files in dir that are not in flist'
    globstr = os.path.join(dir, '*.py')
    fnames = glob.glob(globstr)

    pyfiles = set([os.path.split(fullpath)[-1] for fullpath in set(fnames)])

    exclude = set(excluded.get(dir, []))
    flist = set(flist)
    missing = list(pyfiles - flist - exclude)
    missing.sort()
    if missing:
        print('%s files not tested: %s' % (dir, ', '.join(missing)))


def report_all_missing(directories):
    for f in directories:
        report_missing(dirs[f], files[f])


# tests known to fail on a given backend

failbackend = dict(
    svg=('tex_demo.py', ),
    agg=('hyperlinks.py', ),
    pdf=('hyperlinks.py', ),
    ps=('hyperlinks.py', ),
    )


try:
    import subprocess

    def run(arglist):
```

```
        try:
            ret = subprocess.call(arglist)
        except KeyboardInterrupt:
            sys.exit()
        else:
            return ret
except ImportError:
    def run(arglist):
        os.system(' '.join(arglist))


def drive(backend, directories, python=['python'], switches=[]):
    exclude = failbackend.get(backend, [])

    # Clear the destination directory for the examples
    path = backend
    if os.path.exists(path):
        import glob
        for fname in os.listdir(path):
            os.unlink(os.path.join(path, fname))
    else:
        os.mkdir(backend)
    failures = []

    testcases = [os.path.join(dirs[d], fname)
                 for d in directories
                 for fname in files[d]]

    for fullpath in testcases:
        print('\tdriving %-40s' % (fullpath))
        sys.stdout.flush()
        fpath, fname = os.path.split(fullpath)

        if fname in exclude:
            print('\tSkipping %s, known to fail on backend: %s' % backend)
            continue

        basename, ext = os.path.splitext(fname)
        outfile = os.path.join(path, basename)
        tmpfile_name = '_tmp_%s.py' % basename
        tmpfile = open(tmpfile_name, 'w')

        future_imports = 'from __future__ import division, print_function'
        for line in open(fullpath):
            line_lstrip = line.lstrip()
            if line_lstrip.startswith("#"):
                tmpfile.write(line)
            elif 'unicode_literals' in line:
                future_imports = future_imports + ', unicode_literals'

        tmpfile.writelines((
            future_imports + '\n',
            'import sys\n',
```

```
            'sys.path.append("%s")\n' % fpath.replace('\\', '\\\\'),
            'import matplotlib\n',
            'matplotlib.use("%s")\n' % backend,
            'from pylab import savefig\n',
            'import numpy\n',
            'numpy.seterr(invalid="ignore")\n',
            ))
        for line in open(fullpath):
            line_lstrip = line.lstrip()
            if (line_lstrip.startswith('from __future__ import') or
                    line_lstrip.startswith('matplotlib.use') or
                    line_lstrip.startswith('savefig') or
                    line_lstrip.startswith('show')):
                continue
            tmpfile.write(line)
        if backend in rcsetup.interactive_bk:
            tmpfile.write('show()')
        else:
            tmpfile.write('\nsavefig(r"%s", dpi=150)' % outfile)

        tmpfile.close()
        start_time = time.time()
        program = [x % {'name': basename} for x in python]
        ret = run(program + [tmpfile_name] + switches)
        end_time = time.time()
        print("%s %s" % ((end_time - start_time), ret))
        #os.system('%s %s %s' % (python, tmpfile_name, ' '.join(switches)))
        os.remove(tmpfile_name)
        if ret:
            failures.append(fullpath)
    return failures


def parse_options():
    doc = (__doc__ and __doc__.split('\n\n')) or "  "
    op = OptionParser(description=doc[0].strip(),
                      usage='%prog [options] [--] [backends and switches]',
                      #epilog='\n'.join(doc[1:])  # epilog not supported on my python2.4␣
 ↪machine: JDH
                      )
    op.disable_interspersed_args()
    op.set_defaults(dirs='pylab,api,units,mplot3d',
                    clean=False, coverage=False, valgrind=False)
    op.add_option('-d', '--dirs', '--directories', type='string',
                  dest='dirs', help=dedent('''
      Run only the tests in these directories; comma-separated list of
      one or more of: pylab (or pylab_examples), api, units, mplot3d'''))
    op.add_option('-b', '--backends', type='string', dest='backends',
                  help=dedent('''
      Run tests only for these backends; comma-separated list of
      one or more of: agg, ps, svg, pdf, template, cairo,
      Default is everything except cairo.'''))
    op.add_option('--clean', action='store_true', dest='clean',
```

```
                  help='Remove result directories, run no tests')
    op.add_option('-c', '--coverage', action='store_true', dest='coverage',
                  help='Run in coverage.py')
    op.add_option('-v', '--valgrind', action='store_true', dest='valgrind',
                  help='Run in valgrind')

    options, args = op.parse_args()
    switches = [x for x in args if x.startswith('--')]
    backends = [x.lower() for x in args if not x.startswith('--')]
    if options.backends:
        backends += [be.lower() for be in options.backends.split(',')]

    result = Bunch(
        dirs=options.dirs.split(','),
        backends=backends or ['agg', 'ps', 'svg', 'pdf', 'template'],
        clean=options.clean,
        coverage=options.coverage,
        valgrind=options.valgrind,
        switches=switches)
    if 'pylab_examples' in result.dirs:
        result.dirs[result.dirs.index('pylab_examples')] = 'pylab'
    #print(result)
    return (result)

if __name__ == '__main__':
    times = {}
    failures = {}
    options = parse_options()

    if options.clean:
        localdirs = [d for d in glob.glob('*') if os.path.isdir(d)]
        all_backends_set = set(all_backends)
        for d in localdirs:
            if d.lower() not in all_backends_set:
                continue
            print('removing %s' % d)
            for fname in glob.glob(os.path.join(d, '*')):
                os.remove(fname)
            os.rmdir(d)
        for fname in glob.glob('_tmp*.py'):
            os.remove(fname)

        print('all clean...')
        raise SystemExit
    if options.coverage:
        python = ['coverage.py', '-x']
    elif options.valgrind:
        python = ['valgrind', '--tool=memcheck', '--leak-check=yes',
                  '--log-file=%(name)s', sys.executable]
    elif sys.platform == 'win32':
        python = [sys.executable]
    else:
        python = [sys.executable]
```

```
    report_all_missing(options.dirs)
    for backend in options.backends:
        print('testing %s %s' % (backend, ' '.join(options.switches)))
        t0 = time.time()
        failures[backend] = \
            drive(backend, options.dirs, python, options.switches)
        t1 = time.time()
        times[backend] = (t1 - t0)/60.0

    #print(times)
    for backend, elapsed in times.items():
        print('Backend %s took %1.2f minutes to complete' % (backend, elapsed))
        failed = failures[backend]
        if failed:
            print('  Failures: %s' % failed)
        if 'template' in times:
            print('\ttemplate ratio %1.3f, template residual %1.3f' % (
                elapsed/times['template'], elapsed - times['template']))
```

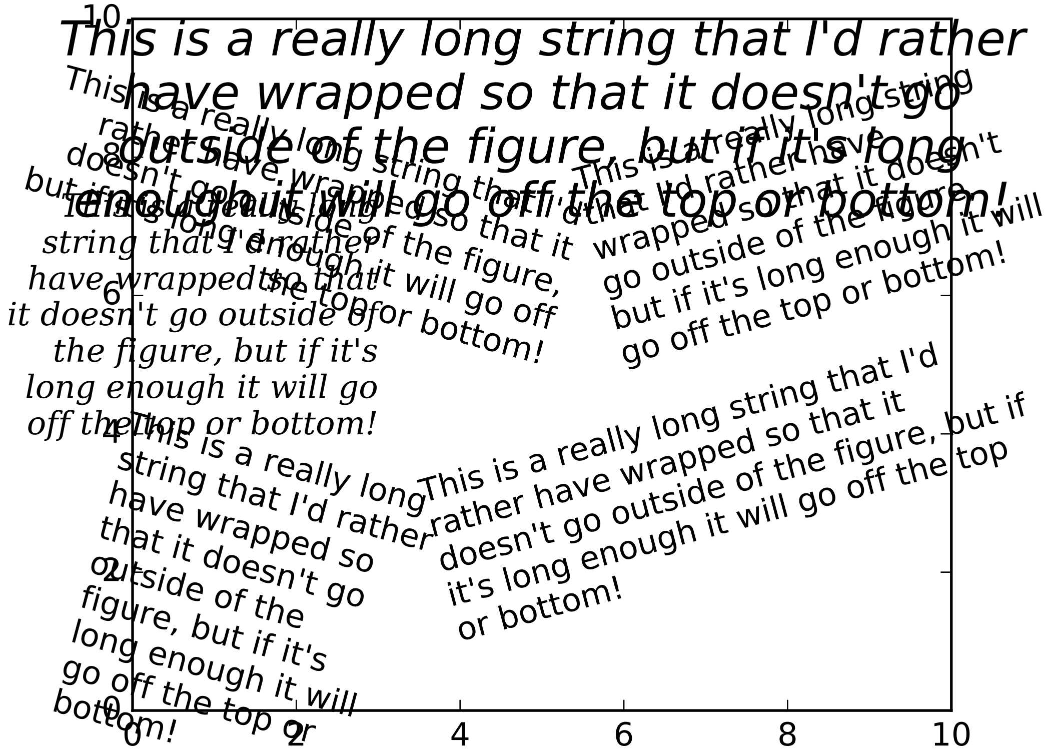
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

NINETYNINE

TEXT_LABELS_AND_ANNOTATIONS EXAMPLES

99.1 text_labels_and_annotations example code: autowrap_demo.py

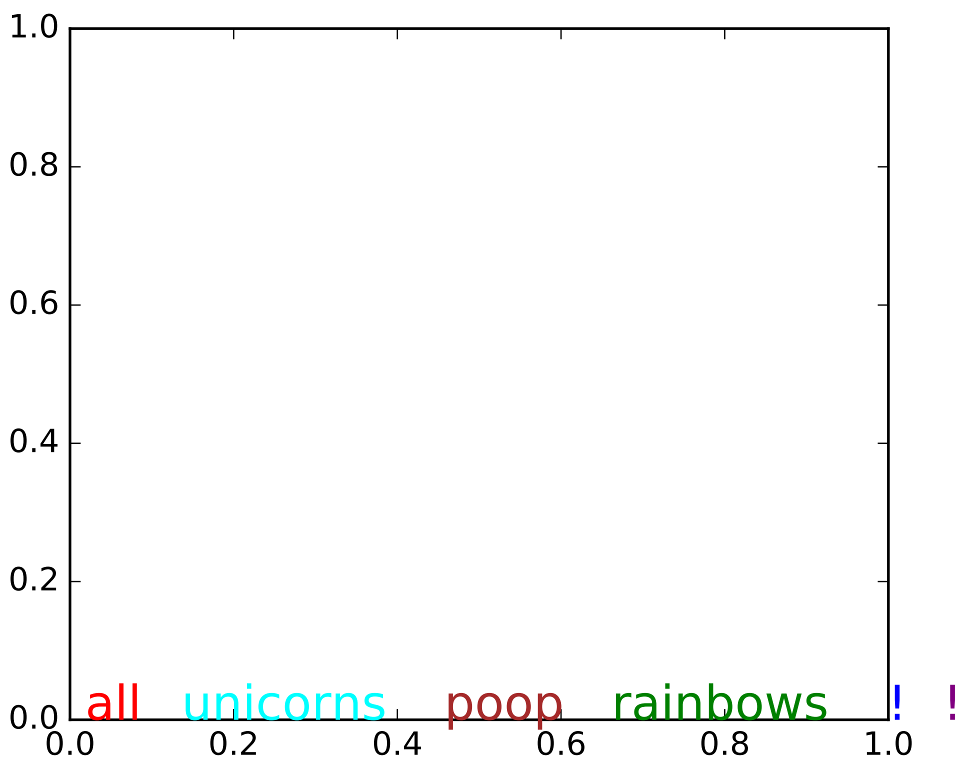



```
"""
Auto-wrapping text demo.
"""
import matplotlib.pyplot as plt

fig = plt.figure()
plt.axis([0, 10, 0, 10])
t = "This is a really long string that I'd rather have wrapped so that it"\
    " doesn't go outside of the figure, but if it's long enough it will go"\
```

```
    " off the top or bottom!"
plt.text(4, 1, t, ha='left', rotation=15, wrap=True)
plt.text(6, 5, t, ha='left', rotation=15, wrap=True)
plt.text(5, 5, t, ha='right', rotation=-15, wrap=True)
plt.text(5, 10, t, fontsize=18, style='oblique', ha='center',
        va='top', wrap=True)
plt.text(3, 4, t, family='serif', style='italic', ha='right', wrap=True)
plt.text(-1, 0, t, ha='left', rotation=-15, wrap=True)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

99.2 text_labels_and_annotations example code: rainbow_text.py



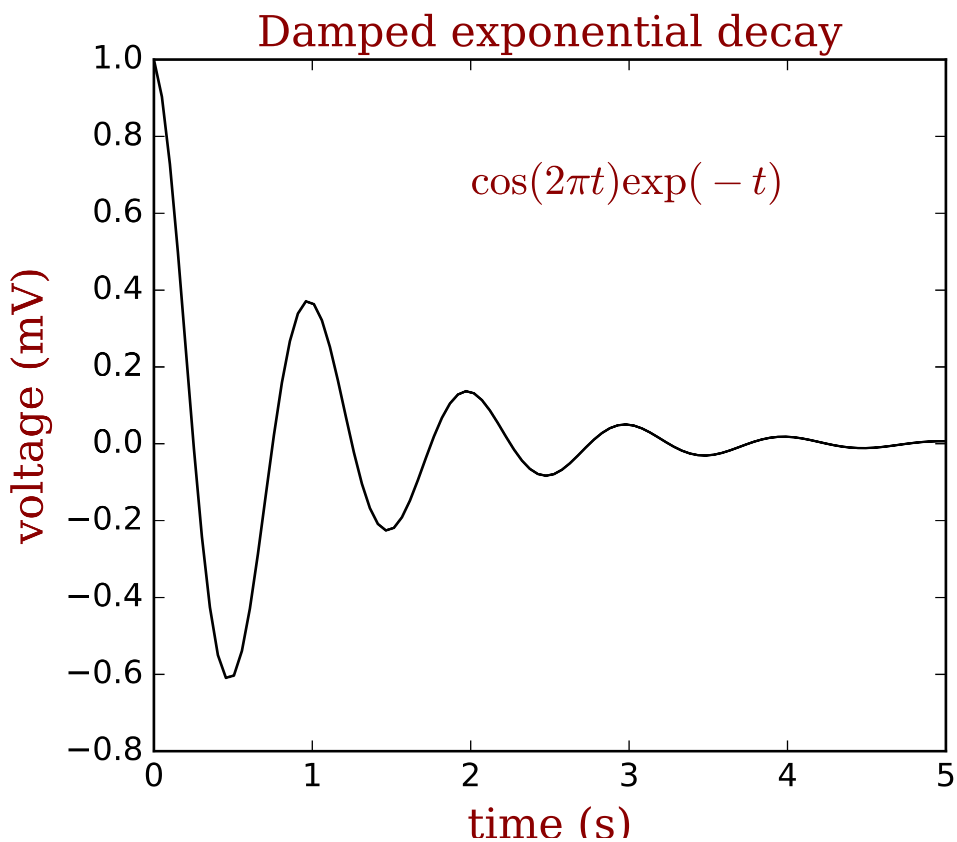

```
# -*- coding: utf-8 -*-
"""
The example shows how to string together several text objects.

HISTORY
-------
On the matplotlib-users list back in February 2012, Gökhan Sever asked the
```

```
following question:

    Is there a way in matplotlib to partially specify the color of a string?

    Example:

    plt.ylabel("Today is cloudy.")
    How can I show "today" as red, "is" as green and "cloudy." as blue?

    Thanks.

Paul Ivanov responded with this answer:
"""
import matplotlib.pyplot as plt
from matplotlib import transforms


def rainbow_text(x, y, strings, colors, ax=None, **kw):
    """
    Take a list of ``strings`` and ``colors`` and place them next to each
    other, with text strings[i] being shown in colors[i].

    This example shows how to do both vertical and horizontal text, and will
    pass all keyword arguments to plt.text, so you can set the font size,
    family, etc.

    The text will get added to the ``ax`` axes, if provided, otherwise the
    currently active axes will be used.
    """
    if ax is None:
        ax = plt.gca()
    t = ax.transData
    canvas = ax.figure.canvas

    # horizontal version
    for s, c in zip(strings, colors):
        text = ax.text(x, y, " " + s + " ", color=c, transform=t, **kw)
        text.draw(canvas.get_renderer())
        ex = text.get_window_extent()
        t = transforms.offset_copy(text._transform, x=ex.width, units='dots')

    # vertical version
    for s, c in zip(strings, colors):
        text = ax.text(x, y, " " + s + " ", color=c, transform=t,
                       rotation=90, va='bottom', ha='center', **kw)
        text.draw(canvas.get_renderer())
        ex = text.get_window_extent()
        t = transforms.offset_copy(text._transform, y=ex.height, units='dots')


rainbow_text(0, 0, "all unicorns poop rainbows ! ! !".split(),
             ['red', 'cyan', 'brown', 'green', 'blue', 'purple', 'black'],
             size=18)
```

```
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

99.3 text_labels_and_annotations example code: text_demo_fontdict.py

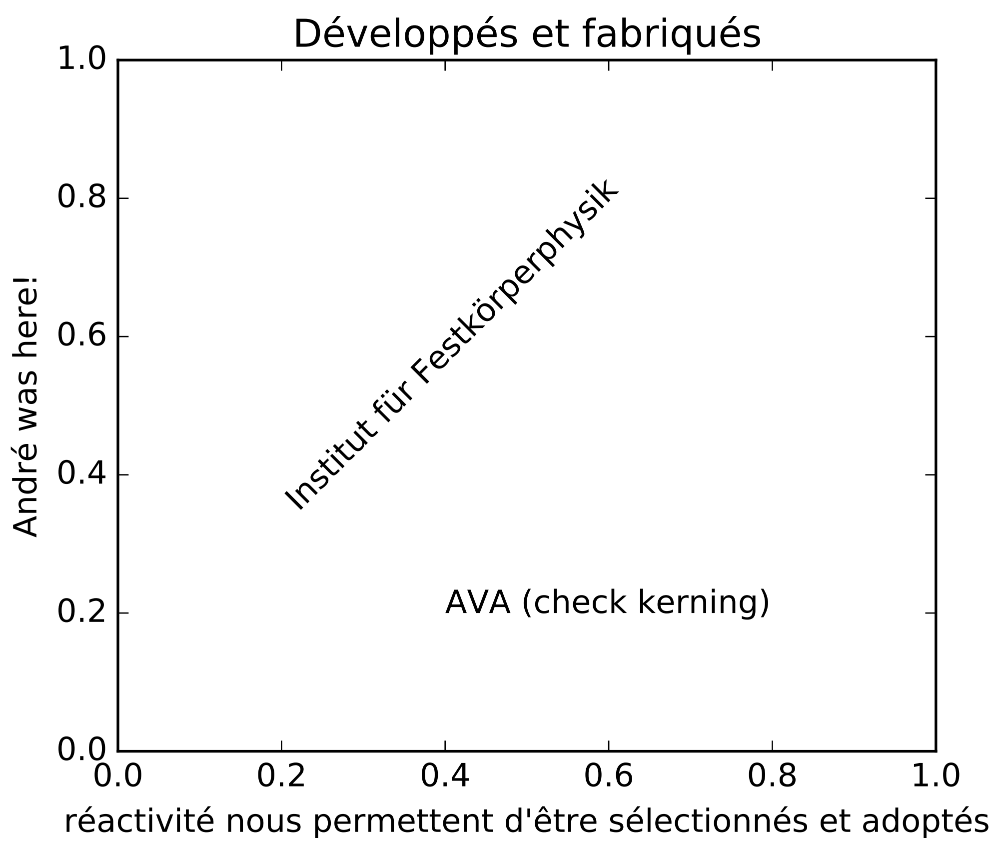



```
"""
Demo using fontdict to control style of text and labels.
"""
import numpy as np
import matplotlib.pyplot as plt


font = {'family': 'serif',
        'color':  'darkred',
        'weight': 'normal',
        'size': 16,
        }

x = np.linspace(0.0, 5.0, 100)
y = np.cos(2*np.pi*x) * np.exp(-x)
```

```
plt.plot(x, y, 'k')
plt.title('Damped exponential decay', fontdict=font)
plt.text(2, 0.65, r'$\cos(2 \pi t) \exp(-t)$', fontdict=font)
plt.xlabel('time (s)', fontdict=font)
plt.ylabel('voltage (mV)', fontdict=font)

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

99.4 text_labels_and_annotations example code: unicode_demo.py




```
# -*- coding: utf-8 -*-
"""
Demo of unicode support in text and labels.
"""
from __future__ import unicode_literals

import matplotlib.pyplot as plt
```

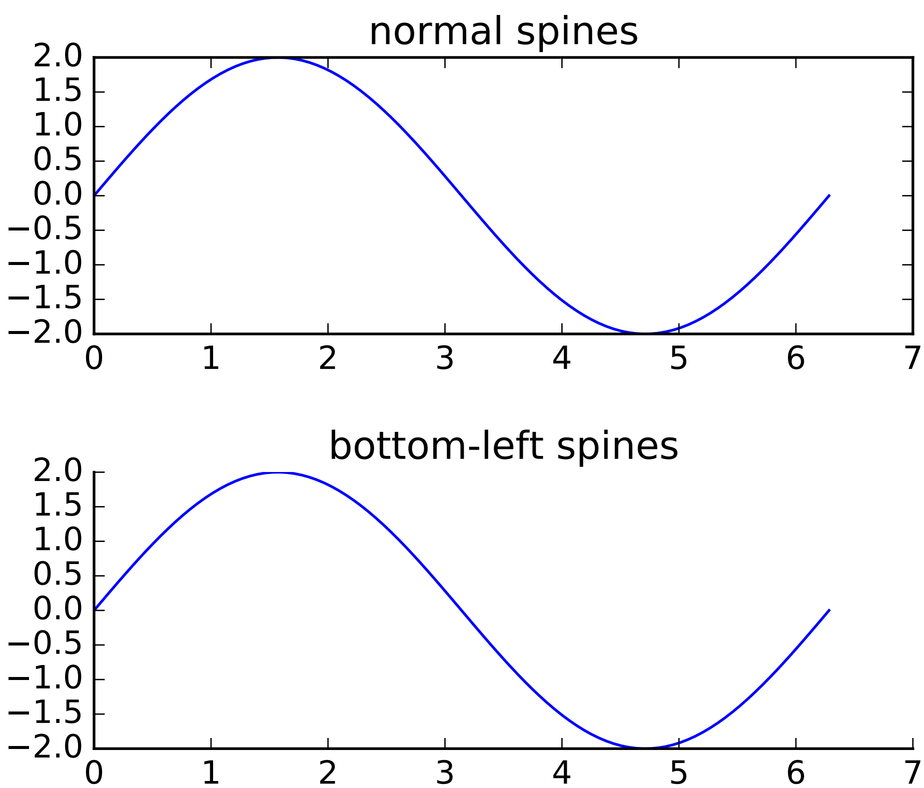
```
plt.title('Développés et fabriqués')
plt.xlabel("réactivité nous permettent d'être sélectionnés et adoptés")
plt.ylabel('André was here!')
plt.text(0.2, 0.8, 'Institut für Festkörperphysik', rotation=45)
plt.text(0.4, 0.2, 'AVA (check kerning)')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

TICKS_AND_SPINES EXAMPLES

100.1 ticks_and_spines example code: spines_demo.py

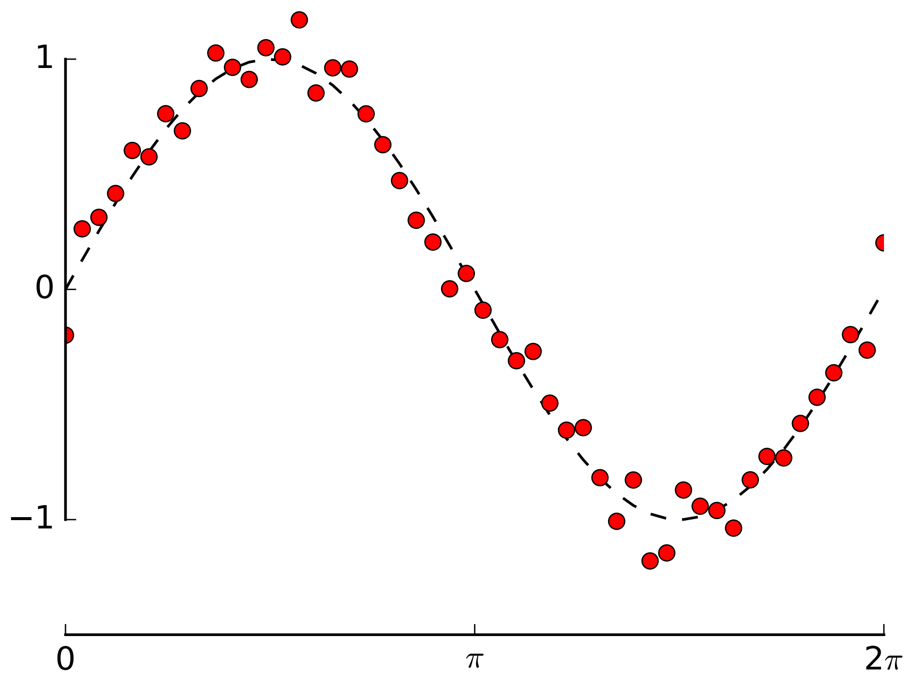



```
"""
Basic demo of axis spines.

This demo compares a normal axes, with spines on all four sides, and an axes
with spines only on the left and bottom.
"""
import numpy as np
import matplotlib.pyplot as plt


x = np.linspace(0, 2 * np.pi, 100)
```

```
y = 2 * np.sin(x)

fig, (ax0, ax1) = plt.subplots(nrows=2)

ax0.plot(x, y)
ax0.set_title('normal spines')

ax1.plot(x, y)
ax1.set_title('bottom-left spines')

# Hide the right and top spines
ax1.spines['right'].set_visible(False)
ax1.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax1.yaxis.set_ticks_position('left')
ax1.xaxis.set_ticks_position('bottom')

# Tweak spacing between subplots to prevent labels from overlapping
plt.subplots_adjust(hspace=0.5)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

100.2 ticks_and_spines example code: spines_demo_bounds.py

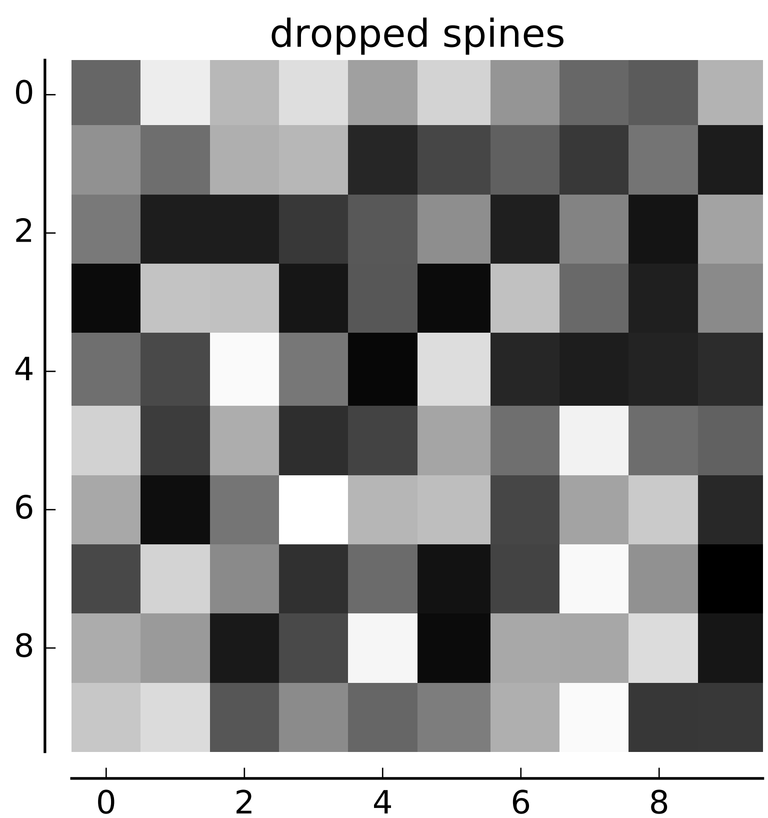



```
"""
Demo of spines using custom bounds to limit the extent of the spine.
"""
import numpy as np
import matplotlib.pyplot as plt


x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = y + 0.1 * np.random.normal(size=x.shape)

fig, ax = plt.subplots()
ax.plot(x, y, 'k--')
ax.plot(x, y2, 'ro')

# set ticks and tick labels
ax.set_xlim((0, 2*np.pi))
ax.set_xticks([0, np.pi, 2*np.pi])
ax.set_xticklabels(['0', '$\pi$', '2$\pi$'])
ax.set_ylim((-1.5, 1.5))
ax.set_yticks([-1, 0, 1])
```

```
# Only draw spine between the y-ticks
ax.spines['left'].set_bounds(-1, 1)
# Hide the right and top spines
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

100.3 ticks_and_spines example code: spines_demo_dropped.py

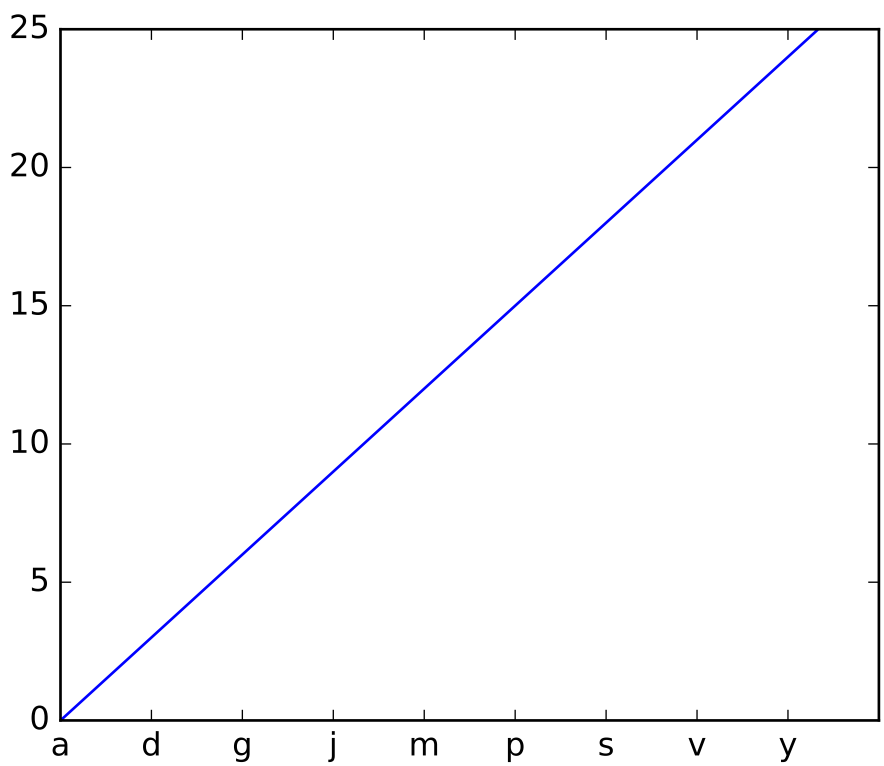



```
"""
Demo of spines offset from the axes (a.k.a. "dropped spines").
"""
import numpy as np
import matplotlib.pyplot as plt
```

```
fig, ax = plt.subplots()

image = np.random.uniform(size=(10, 10))
ax.imshow(image, cmap=plt.cm.gray, interpolation='nearest')
ax.set_title('dropped spines')

# Move left and bottom spines outward by 10 points
ax.spines['left'].set_position(('outward', 10))
ax.spines['bottom'].set_position(('outward', 10))
# Hide the right and top spines
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

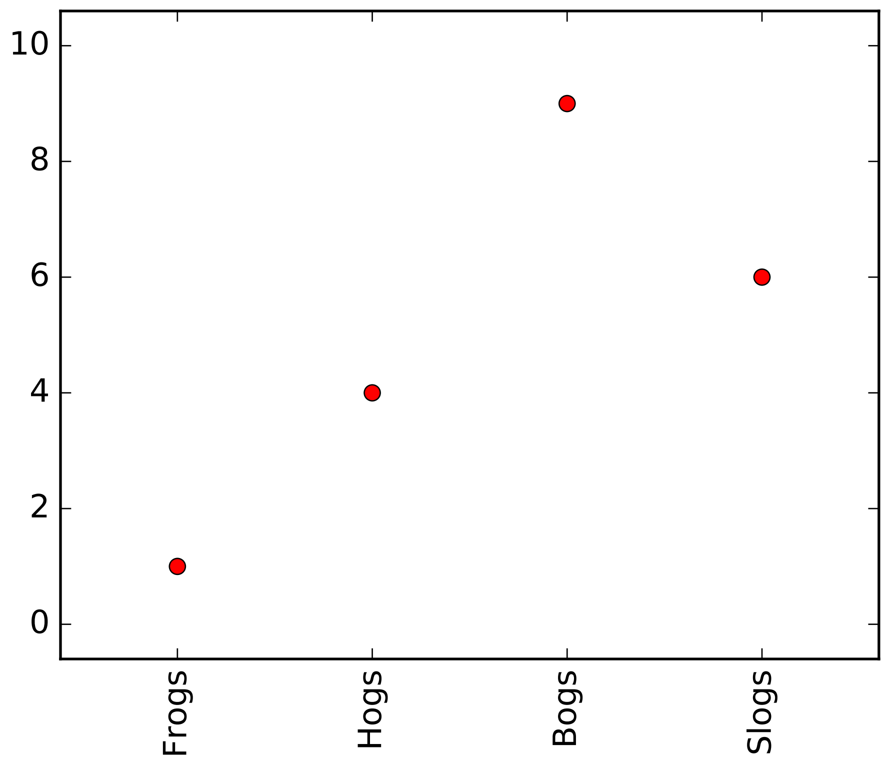
100.4 ticks_and_spines example code: tick_labels_from_values.py



```
"""

Basic demo showing how to set tick labels to values of a series.

Using ax.set_xticks causes the tick labels to be set on the currently
chosen ticks. However, you may want to allow matplotlib to dynamically
choose the number of ticks and their spacing.

In this case it may be better to determine the tick label from the
value at the tick. The following example shows how to do this.

NB: The MaxNLocator is used here to ensure that the tick values
take integer values.

"""

import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter, MaxNLocator
fig = plt.figure()
ax = fig.add_subplot(111)
xs = range(26)
ys = range(26)
labels = list('abcdefghijklmnopqrstuvwxyz')


def format_fn(tick_val, tick_pos):
    if int(tick_val) in xs:
        return labels[int(tick_val)]
    else:
        return ''
ax.xaxis.set_major_formatter(FuncFormatter(format_fn))
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
ax.plot(xs, ys)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

100.5 ticks_and_spines example code: ticklabels_demo_rotation.py




```
"""
Demo of custom tick-labels with user-defined rotation.
"""
import matplotlib.pyplot as plt


x = [1, 2, 3, 4]
y = [1, 4, 9, 6]
labels = ['Frogs', 'Hogs', 'Bogs', 'Slogs']

plt.plot(x, y, 'ro')
# You can specify a rotation for the tick labels in degrees or with keywords.
plt.xticks(x, labels, rotation='vertical')
# Pad margins so that markers don't get clipped by the axes
plt.margins(0.2)
# Tweak spacing to prevent clipping of tick-labels
plt.subplots_adjust(bottom=0.15)
plt.show()
```

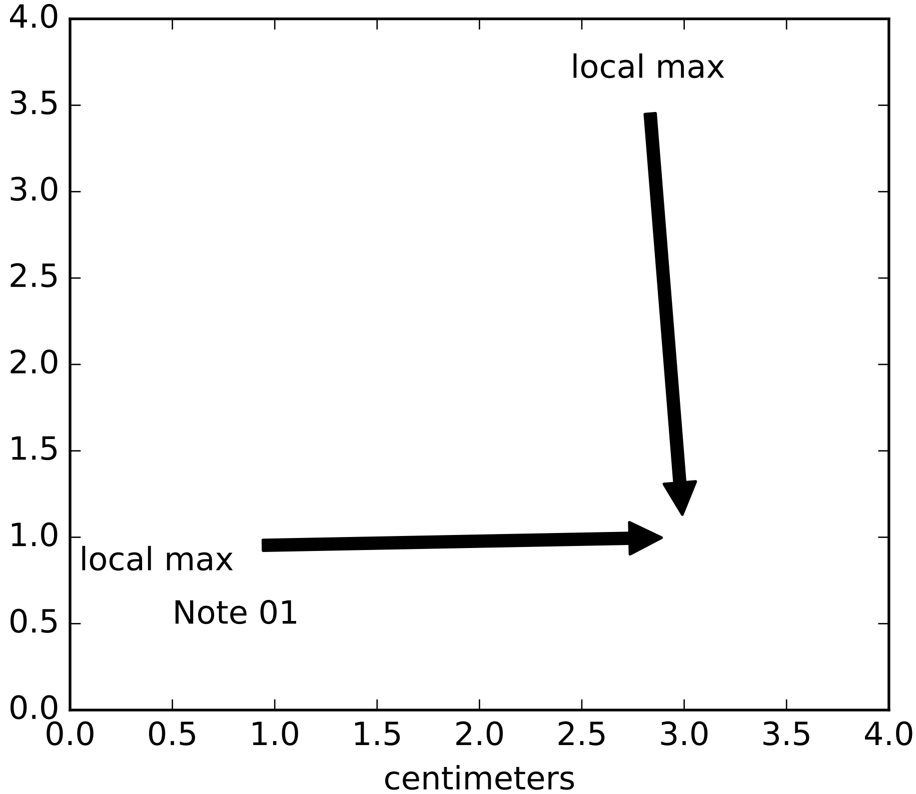
Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

ONE

UNITS EXAMPLES

101.1 units example code: annotate_with_units.py

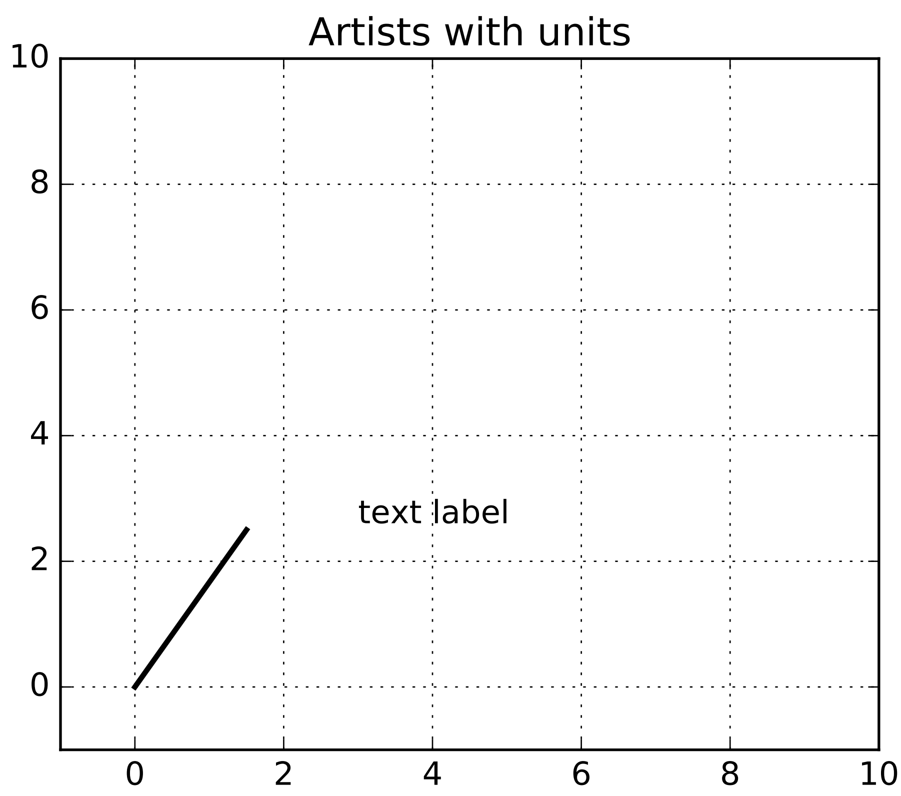



```
import matplotlib.pyplot as plt
from basic_units import cm

fig, ax = plt.subplots()

ax.annotate("Note 01", [0.5*cm, 0.5*cm])

# xy and text both unitized
ax.annotate('local max', xy=(3*cm, 1*cm), xycoords='data',
```

```
            xytext=(0.8*cm, 0.95*cm), textcoords='data',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')

# mixing units w/ nonunits
ax.annotate('local max', xy=(3*cm, 1*cm), xycoords='data',
            xytext=(0.8, 0.95), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top')


ax.set_xlim(0*cm, 4*cm)
ax.set_ylim(0*cm, 4*cm)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

101.2 units example code: artist_tests.py

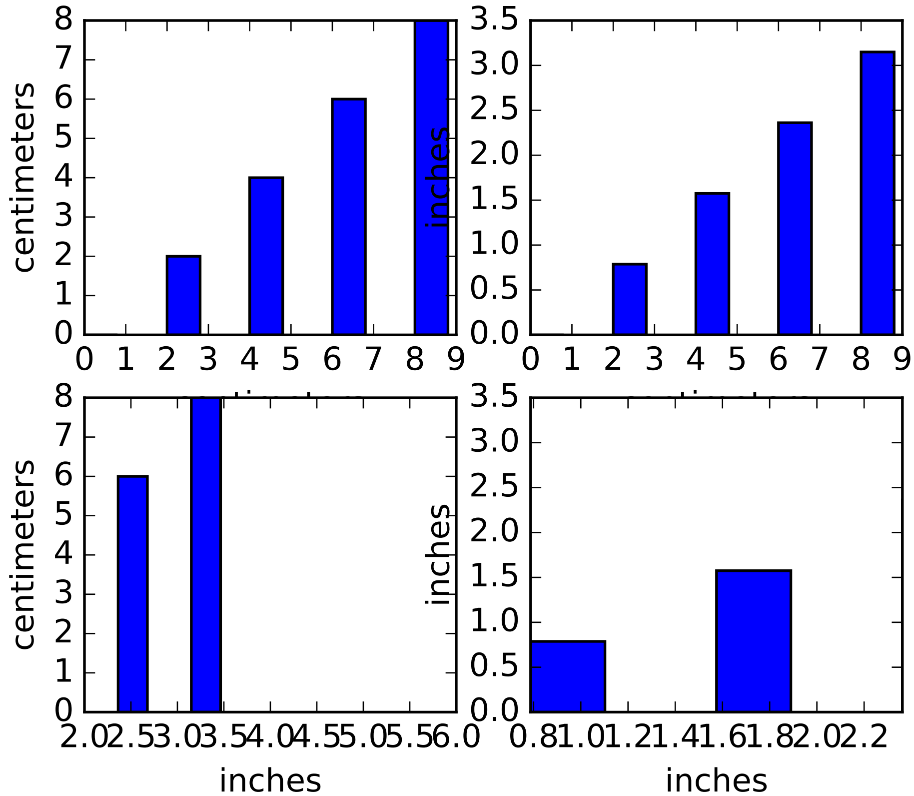



```
"""
Test unit support with each of the matplotlib primitive artist types
```

```
The axes handles unit conversions and the artists keep a pointer to
their axes parent, so you must init the artists with the axes instance
if you want to initialize them with unit data, or else they will not
know how to convert the units to scalars
"""
import random
import matplotlib.lines as lines
import matplotlib.patches as patches
import matplotlib.text as text
import matplotlib.collections as collections

from basic_units import cm, inch
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.xaxis.set_units(cm)
ax.yaxis.set_units(cm)

if 0:
    # test a line collection
    # Not supported at present.
    verts = []
    for i in range(10):
        # a random line segment in inches
        verts.append(zip(*inch*10*np.random.rand(2, random.randint(2, 15))))
    lc = collections.LineCollection(verts, axes=ax)
    ax.add_collection(lc)

# test a plain-ol-line
line = lines.Line2D([0*cm, 1.5*cm], [0*cm, 2.5*cm], lw=2, color='black', axes=ax)
ax.add_line(line)

if 0:
    # test a patch
    # Not supported at present.
    rect = patches.Rectangle((1*cm, 1*cm), width=5*cm, height=2*cm, alpha=0.2, axes=ax)
    ax.add_patch(rect)


t = text.Text(3*cm, 2.5*cm, 'text label', ha='left', va='bottom', axes=ax)
ax.add_artist(t)

ax.set_xlim(-1*cm, 10*cm)
ax.set_ylim(-1*cm, 10*cm)
#ax.xaxis.set_units(inch)
ax.grid(True)
ax.set_title("Artists with units")
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

101.3 units example code: bar_demo2.py

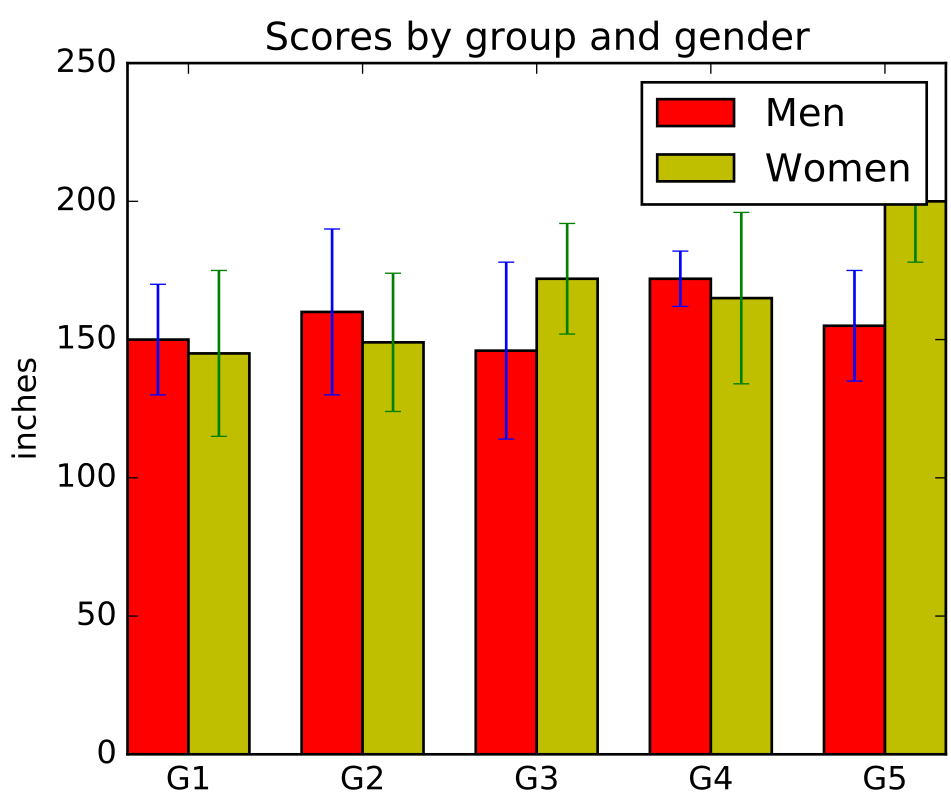



```python
"""
plot using a variety of cm vs inches conversions.  The example shows
how default unit instrospection works (ax1), how various keywords can
be used to set the x and y units to override the defaults (ax2, ax3,
ax4) and how one can set the xlimits using scalars (ax3, current units
assumed) or units (conversions applied to get the numbers to current
units)

"""
import numpy as np
from basic_units import cm, inch
import matplotlib.pyplot as plt


cms = cm * np.arange(0, 10, 2)
bottom = 0*cm
width = 0.8*cm

fig = plt.figure()

ax1 = fig.add_subplot(2, 2, 1)
ax1.bar(cms, cms, bottom=bottom)
```

```
ax2 = fig.add_subplot(2, 2, 2)
ax2.bar(cms, cms, bottom=bottom, width=width, xunits=cm, yunits=inch)

ax3 = fig.add_subplot(2, 2, 3)
ax3.bar(cms, cms, bottom=bottom, width=width, xunits=inch, yunits=cm)
ax3.set_xlim(2, 6)  # scalars are interpreted in current units

ax4 = fig.add_subplot(2, 2, 4)
ax4.bar(cms, cms, bottom=bottom, width=width, xunits=inch, yunits=inch)
#fig.savefig('simple_conversion_plot.png')
ax4.set_xlim(2*cm, 6*cm)  # cm are converted to inches

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

101.4 units example code: bar_unit_demo.py




```
#!/usr/bin/env python
import numpy as np
from basic_units import cm, inch
```

```
import matplotlib.pyplot as plt


N = 5
menMeans = (150*cm, 160*cm, 146*cm, 172*cm, 155*cm)
menStd = (20*cm, 30*cm, 32*cm, 10*cm, 20*cm)

fig, ax = plt.subplots()

ind = np.arange(N)    # the x locations for the groups
width = 0.35         # the width of the bars
p1 = ax.bar(ind, menMeans, width, color='r', bottom=0*cm, yerr=menStd)


womenMeans = (145*cm, 149*cm, 172*cm, 165*cm, 200*cm)
womenStd = (30*cm, 25*cm, 20*cm, 31*cm, 22*cm)
p2 = ax.bar(ind + width, womenMeans, width, color='y', bottom=0*cm, yerr=womenStd)

ax.set_title('Scores by group and gender')
ax.set_xticks(ind + width)
ax.set_xticklabels(('G1', 'G2', 'G3', 'G4', 'G5'))

ax.legend((p1[0], p2[0]), ('Men', 'Women'))
ax.yaxis.set_units(inch)
ax.autoscale_view()

#plt.savefig('barchart_demo')
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

101.5 units example code: basic_units.py

```
import math

import numpy as np

import matplotlib.units as units
import matplotlib.ticker as ticker
from matplotlib.axes import Axes
from matplotlib.cbook import iterable


class ProxyDelegate(object):
    def __init__(self, fn_name, proxy_type):
        self.proxy_type = proxy_type
        self.fn_name = fn_name

    def __get__(self, obj, objtype=None):
        return self.proxy_type(self.fn_name, obj)
```

```
class TaggedValueMeta (type):
    def __init__(cls, name, bases, dict):
        for fn_name in cls._proxies.keys():
            try:
                dummy = getattr(cls, fn_name)
            except AttributeError:
                setattr(cls, fn_name,
                        ProxyDelegate(fn_name, cls._proxies[fn_name]))


class PassThroughProxy(object):
    def __init__(self, fn_name, obj):
        self.fn_name = fn_name
        self.target = obj.proxy_target

    def __call__(self, *args):
        fn = getattr(self.target, self.fn_name)
        ret = fn(*args)
        return ret


class ConvertArgsProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        PassThroughProxy.__init__(self, fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        converted_args = []
        for a in args:
            try:
                converted_args.append(a.convert_to(self.unit))
            except AttributeError:
                converted_args.append(TaggedValue(a, self.unit))
        converted_args = tuple([c.get_value() for c in converted_args])
        return PassThroughProxy.__call__(self, *converted_args)


class ConvertReturnProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        PassThroughProxy.__init__(self, fn_name, obj)
        self.unit = obj.unit

    def __call__(self, *args):
        ret = PassThroughProxy.__call__(self, *args)
        if (type(ret) == type(NotImplemented)):
            return NotImplemented
        return TaggedValue(ret, self.unit)


class ConvertAllProxy(PassThroughProxy):
    def __init__(self, fn_name, obj):
        PassThroughProxy.__init__(self, fn_name, obj)
```

```
        self.unit = obj.unit

    def __call__(self, *args):
        converted_args = []
        arg_units = [self.unit]
        for a in args:
            if hasattr(a, 'get_unit') and not hasattr(a, 'convert_to'):
                # if this arg has a unit type but no conversion ability,
                # this operation is prohibited
                return NotImplemented

            if hasattr(a, 'convert_to'):
                try:
                    a = a.convert_to(self.unit)
                except:
                    pass
                arg_units.append(a.get_unit())
                converted_args.append(a.get_value())
            else:
                converted_args.append(a)
                if hasattr(a, 'get_unit'):
                    arg_units.append(a.get_unit())
                else:
                    arg_units.append(None)
        converted_args = tuple(converted_args)
        ret = PassThroughProxy.__call__(self, *converted_args)
        if (type(ret) == type(NotImplemented)):
            return NotImplemented
        ret_unit = unit_resolver(self.fn_name, arg_units)
        if (ret_unit == NotImplemented):
            return NotImplemented
        return TaggedValue(ret, ret_unit)


class _TaggedValue(object):

    _proxies = {'__add__': ConvertAllProxy,
                '__sub__': ConvertAllProxy,
                '__mul__': ConvertAllProxy,
                '__rmul__': ConvertAllProxy,
                '__cmp__': ConvertAllProxy,
                '__lt__': ConvertAllProxy,
                '__gt__': ConvertAllProxy,
                '__len__': PassThroughProxy}

    def __new__(cls, value, unit):
        # generate a new subclass for value
        value_class = type(value)
        try:
            subcls = type('TaggedValue_of_%s' % (value_class.__name__),
                          tuple([cls, value_class]),
                          {})
            if subcls not in units.registry:
```

```
                units.registry[subcls] = basicConverter
            return object.__new__(subcls)
        except TypeError:
            if cls not in units.registry:
                units.registry[cls] = basicConverter
            return object.__new__(cls)

    def __init__(self, value, unit):
        self.value = value
        self.unit = unit
        self.proxy_target = self.value

    def __getattribute__(self, name):
        if (name.startswith('__')):
            return object.__getattribute__(self, name)
        variable = object.__getattribute__(self, 'value')
        if (hasattr(variable, name) and name not in self.__class__.__dict__):
            return getattr(variable, name)
        return object.__getattribute__(self, name)

    def __array__(self, t=None, context=None):
        if t is not None:
            return np.asarray(self.value).astype(t)
        else:
            return np.asarray(self.value, 'O')

    def __array_wrap__(self, array, context):
        return TaggedValue(array, self.unit)

    def __repr__(self):
        return 'TaggedValue(' + repr(self.value) + ', ' + repr(self.unit) + ')'

    def __str__(self):
        return str(self.value) + ' in ' + str(self.unit)

    def __len__(self):
        return len(self.value)

    def __iter__(self):
        class IteratorProxy(object):
            def __init__(self, iter, unit):
                self.iter = iter
                self.unit = unit

            def __next__(self):
                value = next(self.iter)
                return TaggedValue(value, self.unit)
            next = __next__  # for Python 2
        return IteratorProxy(iter(self.value), self.unit)

    def get_compressed_copy(self, mask):
        new_value = np.ma.masked_array(self.value, mask=mask).compressed()
        return TaggedValue(new_value, self.unit)
```

```
    def convert_to(self, unit):
        if (unit == self.unit or not unit):
            return self
        new_value = self.unit.convert_value_to(self.value, unit)
        return TaggedValue(new_value, unit)

    def get_value(self):
        return self.value

    def get_unit(self):
        return self.unit


TaggedValue = TaggedValueMeta('TaggedValue', (_TaggedValue, ), {})


class BasicUnit(object):
    def __init__(self, name, fullname=None):
        self.name = name
        if fullname is None:
            fullname = name
        self.fullname = fullname
        self.conversions = dict()

    def __repr__(self):
        return 'BasicUnit(%s)' % self.name

    def __str__(self):
        return self.fullname

    def __call__(self, value):
        return TaggedValue(value, self)

    def __mul__(self, rhs):
        value = rhs
        unit = self
        if hasattr(rhs, 'get_unit'):
            value = rhs.get_value()
            unit = rhs.get_unit()
            unit = unit_resolver('__mul__', (self, unit))
        if (unit == NotImplemented):
            return NotImplemented
        return TaggedValue(value, unit)

    def __rmul__(self, lhs):
        return self*lhs

    def __array_wrap__(self, array, context):
        return TaggedValue(array, self)

    def __array__(self, t=None, context=None):
        ret = np.array([1])
```

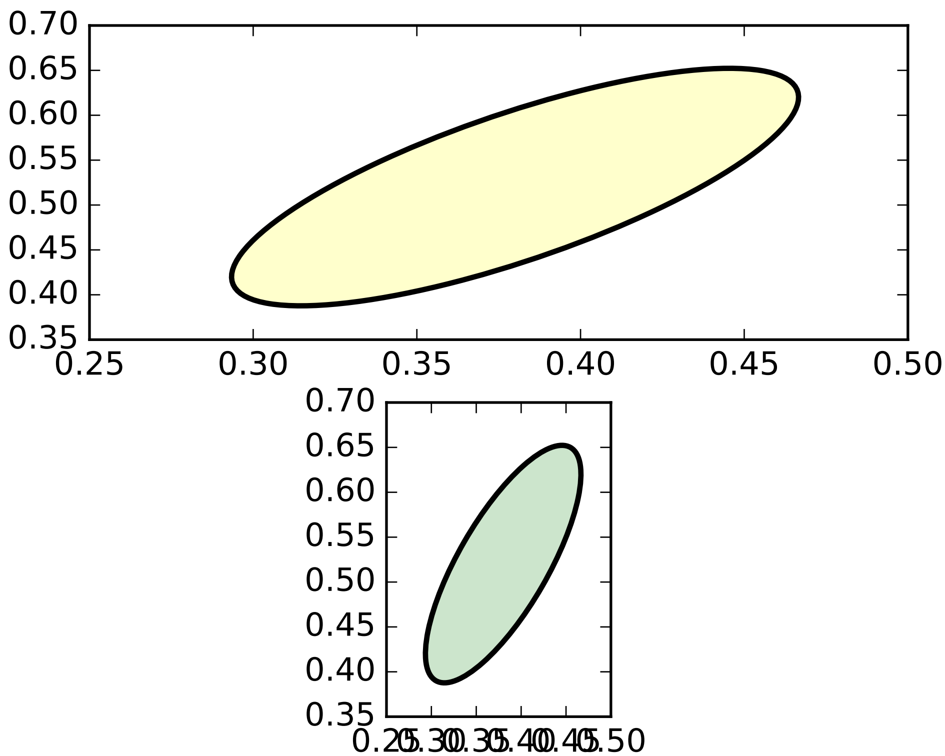
```
        if t is not None:
            return ret.astype(t)
        else:
            return ret

    def add_conversion_factor(self, unit, factor):
        def convert(x):
            return x*factor
        self.conversions[unit] = convert

    def add_conversion_fn(self, unit, fn):
        self.conversions[unit] = fn

    def get_conversion_fn(self, unit):
        return self.conversions[unit]

    def convert_value_to(self, value, unit):
        conversion_fn = self.conversions[unit]
        ret = conversion_fn(value)
        return ret

    def get_unit(self):
        return self


class UnitResolver(object):
    def addition_rule(self, units):
        for unit_1, unit_2 in zip(units[:-1], units[1:]):
            if (unit_1 != unit_2):
                return NotImplemented
        return units[0]

    def multiplication_rule(self, units):
        non_null = [u for u in units if u]
        if (len(non_null) > 1):
            return NotImplemented
        return non_null[0]

    op_dict = {
        '__mul__': multiplication_rule,
        '__rmul__': multiplication_rule,
        '__add__': addition_rule,
        '__radd__': addition_rule,
        '__sub__': addition_rule,
        '__rsub__': addition_rule}

    def __call__(self, operation, units):
        if (operation not in self.op_dict):
            return NotImplemented

        return self.op_dict[operation](self, units)
```

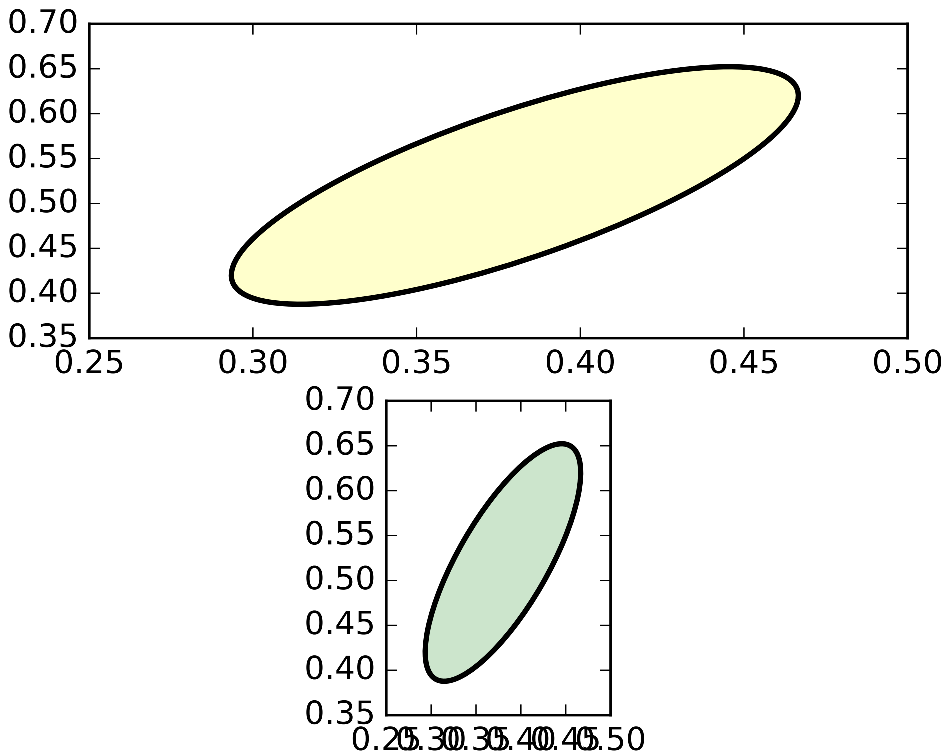
```
unit_resolver = UnitResolver()

cm = BasicUnit('cm', 'centimeters')
inch = BasicUnit('inch', 'inches')
inch.add_conversion_factor(cm, 2.54)
cm.add_conversion_factor(inch, 1/2.54)

radians = BasicUnit('rad', 'radians')
degrees = BasicUnit('deg', 'degrees')
radians.add_conversion_factor(degrees, 180.0/np.pi)
degrees.add_conversion_factor(radians, np.pi/180.0)

secs = BasicUnit('s', 'seconds')
hertz = BasicUnit('Hz', 'Hertz')
minutes = BasicUnit('min', 'minutes')

secs.add_conversion_fn(hertz, lambda x: 1./x)
secs.add_conversion_factor(minutes, 1/60.0)


# radians formatting
def rad_fn(x, pos=None):
    n = int((x / np.pi) * 2.0 + 0.25)
    if n == 0:
        return '0'
    elif n == 1:
        return r'$\pi/2$'
    elif n == 2:
        return r'$\pi$'
    elif n % 2 == 0:
        return r'$%s\pi$' % (n//2,)
    else:
        return r'$%s\pi/2$' % (n,)


class BasicUnitConverter(units.ConversionInterface):
    @staticmethod
    def axisinfo(unit, axis):
        'return AxisInfo instance for x and unit'

        if unit == radians:
            return units.AxisInfo(
                majloc=ticker.MultipleLocator(base=np.pi/2),
                majfmt=ticker.FuncFormatter(rad_fn),
                label=unit.fullname,
            )
        elif unit == degrees:
            return units.AxisInfo(
                majloc=ticker.AutoLocator(),
                majfmt=ticker.FormatStrFormatter(r'$%i^\circ$'),
                label=unit.fullname,
            )
        elif unit is not None:
```

```
            if hasattr(unit, 'fullname'):
                return units.AxisInfo(label=unit.fullname)
            elif hasattr(unit, 'unit'):
                return units.AxisInfo(label=unit.unit.fullname)
        return None

    @staticmethod
    def convert(val, unit, axis):
        if units.ConversionInterface.is_numlike(val):
            return val
        if iterable(val):
            return [thisval.convert_to(unit).get_value() for thisval in val]
        else:
            return val.convert_to(unit).get_value()

    @staticmethod
    def default_units(x, axis):
        'return the default unit for x or None'
        if iterable(x):
            for thisx in x:
                return thisx.unit
        return x.unit


def cos(x):
    if iterable(x):
        return [math.cos(val.convert_to(radians).get_value()) for val in x]
    else:
        return math.cos(x.convert_to(radians).get_value())


basicConverter = BasicUnitConverter()
units.registry[BasicUnit] = basicConverter
units.registry[TaggedValue] = basicConverter
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

101.6 units example code: ellipse_with_units.py



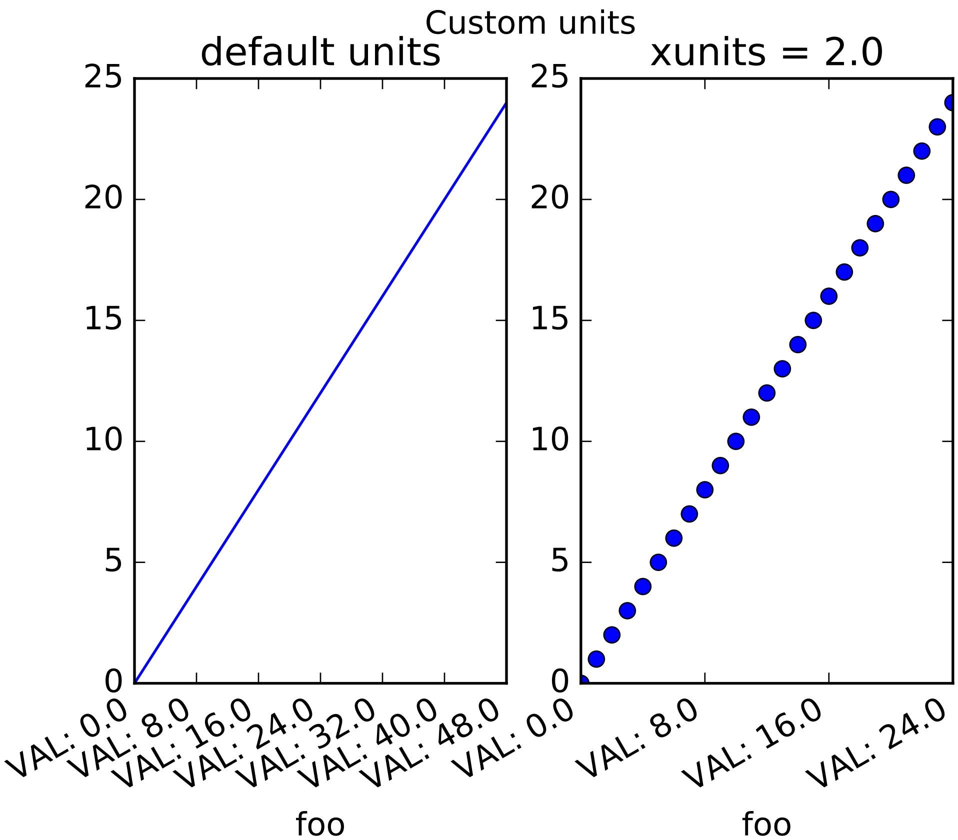



```
"""
Compare the ellipse generated with arcs versus a polygonal approximation
"""
from basic_units import cm
import numpy as np
from matplotlib import patches
import matplotlib.pyplot as plt


xcenter, ycenter = 0.38*cm, 0.52*cm
#xcenter, ycenter = 0., 0.
width, height = 1e-1*cm, 3e-1*cm
angle = -30

theta = np.arange(0.0, 360.0, 1.0)*np.pi/180.0
x = 0.5 * width * np.cos(theta)
y = 0.5 * height * np.sin(theta)

rtheta = np.radians(angle)
R = np.array([
    [np.cos(rtheta), -np.sin(rtheta)],
    [np.sin(rtheta),  np.cos(rtheta)],
    ])


x, y = np.dot(R, np.array([x, y]))
```

```
x += xcenter
y += ycenter

fig = plt.figure()
ax = fig.add_subplot(211, aspect='auto')
ax.fill(x, y, alpha=0.2, facecolor='yellow', edgecolor='yellow', linewidth=1, zorder=1)

e1 = patches.Ellipse((xcenter, ycenter), width, height,
                     angle=angle, linewidth=2, fill=False, zorder=2)

ax.add_patch(e1)

ax = fig.add_subplot(212, aspect='equal')
ax.fill(x, y, alpha=0.2, facecolor='green', edgecolor='green', zorder=1)
e2 = patches.Ellipse((xcenter, ycenter), width, height,
                     angle=angle, linewidth=2, fill=False, zorder=2)


ax.add_patch(e2)

#fig.savefig('ellipse_compare.png')
fig.savefig('ellipse_compare')

fig = plt.figure()
ax = fig.add_subplot(211, aspect='auto')
ax.fill(x, y, alpha=0.2, facecolor='yellow', edgecolor='yellow', linewidth=1, zorder=1)

e1 = patches.Arc((xcenter, ycenter), width, height,
                 angle=angle, linewidth=2, fill=False, zorder=2)

ax.add_patch(e1)

ax = fig.add_subplot(212, aspect='equal')
ax.fill(x, y, alpha=0.2, facecolor='green', edgecolor='green', zorder=1)
e2 = patches.Arc((xcenter, ycenter), width, height,
                 angle=angle, linewidth=2, fill=False, zorder=2)


ax.add_patch(e2)

#fig.savefig('arc_compare.png')
fig.savefig('arc_compare')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

101.7 units example code: evans_test.py



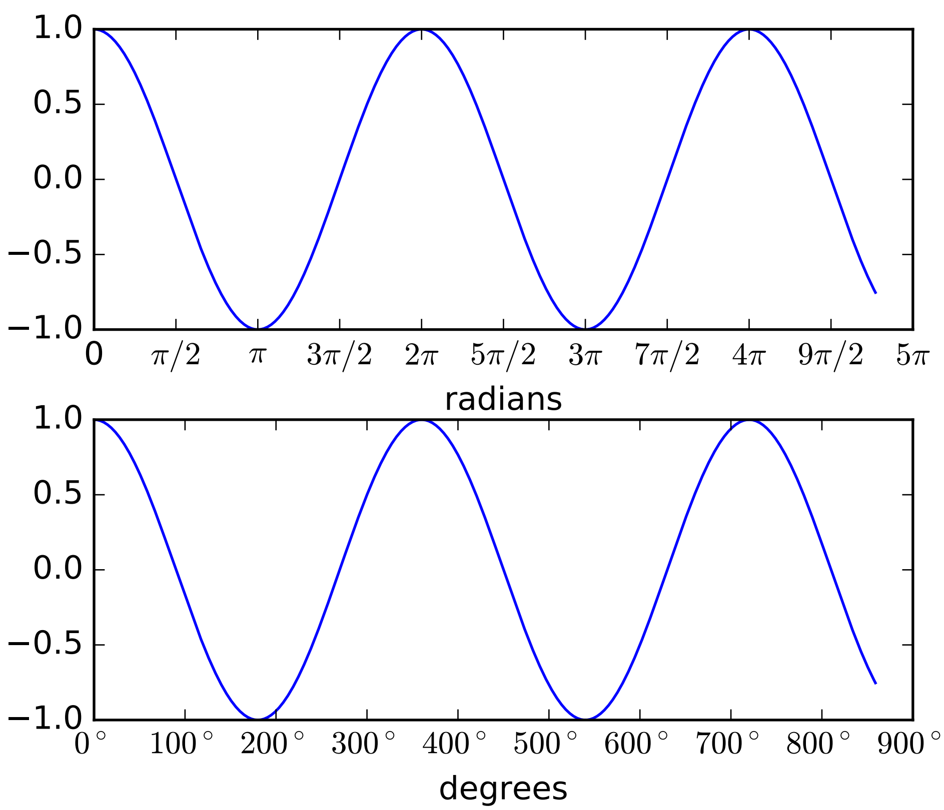

```python
"""
A mockup "Foo" units class which supports
conversion and different tick formatting depending on the "unit".
Here the "unit" is just a scalar conversion factor, but this example shows mpl is
entirely agnostic to what kind of units client packages use
"""
from matplotlib.cbook import iterable
import matplotlib.units as units
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt


class Foo(object):
    def __init__(self, val, unit=1.0):
        self.unit = unit
        self._val = val * unit

    def value(self, unit):
        if unit is None:
            unit = self.unit
        return self._val / unit
```

```
class FooConverter(object):
    @staticmethod
    def axisinfo(unit, axis):
        'return the Foo AxisInfo'
        if unit == 1.0 or unit == 2.0:
            return units.AxisInfo(
                majloc=ticker.IndexLocator(8, 0),
                majfmt=ticker.FormatStrFormatter("VAL: %s"),
                label='foo',
                )

        else:
            return None

    @staticmethod
    def convert(obj, unit, axis):
        """
        convert obj using unit.  If obj is a sequence, return the
        converted sequence
        """
        if units.ConversionInterface.is_numlike(obj):
            return obj

        if iterable(obj):
            return [o.value(unit) for o in obj]
        else:
            return obj.value(unit)

    @staticmethod
    def default_units(x, axis):
        'return the default unit for x or None'
        if iterable(x):
            for thisx in x:
                return thisx.unit
        else:
            return x.unit

units.registry[Foo] = FooConverter()

# create some Foos
x = []
for val in range(0, 50, 2):
    x.append(Foo(val, 1.0))

# and some arbitrary y data
y = [i for i in range(len(x))]


# plot specifying units
fig = plt.figure()
fig.suptitle("Custom units")
fig.subplots_adjust(bottom=0.2)
```

```
ax = fig.add_subplot(1, 2, 2)
ax.plot(x, y, 'o', xunits=2.0)
for label in ax.get_xticklabels():
    label.set_rotation(30)
    label.set_ha('right')
ax.set_title("xunits = 2.0")


# plot without specifying units; will use the None branch for axisinfo
ax = fig.add_subplot(1, 2, 1)
ax.plot(x, y)  # uses default units
ax.set_title('default units')
for label in ax.get_xticklabels():
    label.set_rotation(30)
    label.set_ha('right')

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

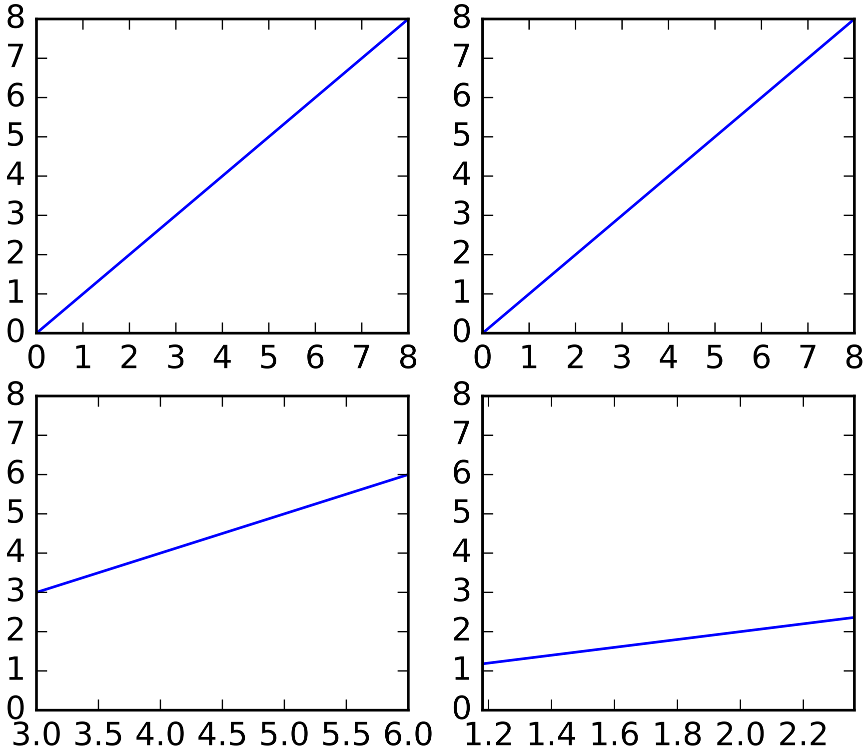
101.8 units example code: radian_demo.py



```
"""
Plot with radians from the basic_units mockup example package
This example shows how the unit class can determine the tick locating,
formatting and axis labeling.
"""
import numpy as np
from basic_units import radians, degrees, cos
from matplotlib.pyplot import figure, show

x = [val*radians for val in np.arange(0, 15, 0.01)]

fig = figure()
fig.subplots_adjust(hspace=0.3)

ax = fig.add_subplot(211)
line1, = ax.plot(x, cos(x), xunits=radians)

ax = fig.add_subplot(212)
line2, = ax.plot(x, cos(x), xunits=degrees)

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

101.9 units example code: units_sample.py

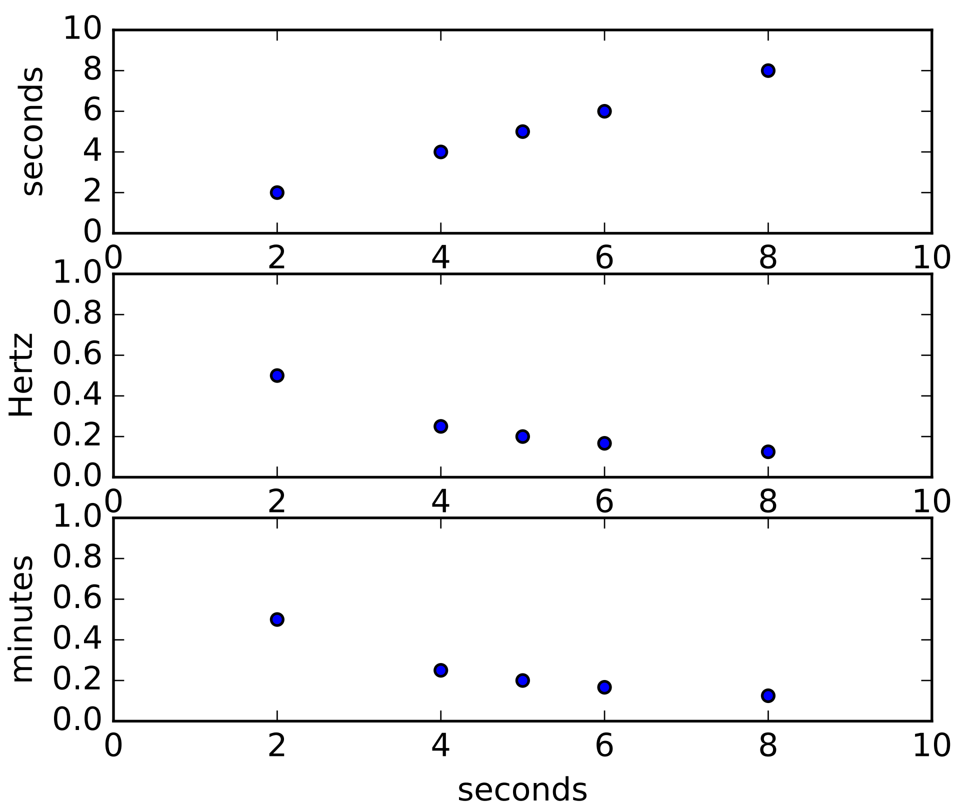



```
"""
plot using a variety of cm vs inches conversions.  The example shows
how default unit instrospection works (ax1), how various keywords can
be used to set the x and y units to override the defaults (ax2, ax3,
ax4) and how one can set the xlimits using scalars (ax3, current units
assumed) or units (conversions applied to get the numbers to current
units)

"""
from basic_units import cm, inch
import matplotlib.pyplot as plt
import numpy

cms = cm * numpy.arange(0, 10, 2)

fig = plt.figure()

ax1 = fig.add_subplot(2, 2, 1)
ax1.plot(cms, cms)

ax2 = fig.add_subplot(2, 2, 2)
ax2.plot(cms, cms, xunits=cm, yunits=inch)
```

```
ax3 = fig.add_subplot(2, 2, 3)
ax3.plot(cms, cms, xunits=inch, yunits=cm)
ax3.set_xlim(3, 6)  # scalars are interpreted in current units

ax4 = fig.add_subplot(2, 2, 4)
ax4.plot(cms, cms, xunits=inch, yunits=inch)
#fig.savefig('simple_conversion_plot.png')
ax4.set_xlim(3*cm, 6*cm)  # cm are converted to inches

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

101.10 units example code: units_scatter.py




```
"""
Demonstrate unit handling

basic_units is a mockup of a true units package used for testing
purposed, which illustrates the basic interface that a units package
must provide to matplotlib.
```

```
The example below shows support for unit conversions over masked
arrays.
"""
import numpy as np
from basic_units import secs, hertz, minutes
from matplotlib.pylab import figure, show

# create masked array


xsecs = secs*np.ma.MaskedArray((1, 2, 3, 4, 5, 6, 7, 8), (1, 0, 1, 0, 0, 0, 1, 0), np.
↪float)
#xsecs = secs*np.arange(1,10.)

fig = figure()
ax1 = fig.add_subplot(3, 1, 1)
ax1.scatter(xsecs, xsecs)
#ax1.set_ylabel('seconds')
ax1.axis([0, 10, 0, 10])

ax2 = fig.add_subplot(3, 1, 2, sharex=ax1)
ax2.scatter(xsecs, xsecs, yunits=hertz)
ax2.axis([0, 10, 0, 1])

ax3 = fig.add_subplot(3, 1, 3, sharex=ax1)
ax3.scatter(xsecs, xsecs, yunits=hertz)
ax3.yaxis.set_units(minutes)
ax3.axis([0, 10, 0, 1])

show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

TWO

USER_INTERFACES EXAMPLES

102.1 user_interfaces example code: embedding_in_gtk.py

[source code]

```
#!/usr/bin/env python
"""
show how to add a matplotlib FigureCanvasGTK or FigureCanvasGTKAgg widget to a
gtk.Window
"""

import gtk

from matplotlib.figure import Figure
from numpy import arange, sin, pi

# uncomment to select /GTK/GTKAgg/GTKCairo
#from matplotlib.backends.backend_gtk import FigureCanvasGTK as FigureCanvas
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
#from matplotlib.backends.backend_gtkcairo import FigureCanvasGTKCairo as FigureCanvas


win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400, 300)
win.set_title("Embedding in GTK")

f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)
a.plot(t, s)

canvas = FigureCanvas(f)  # a gtk.DrawingArea
win.add(canvas)

win.show_all()
gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.2 user_interfaces example code: embedding_in_gtk2.py

[source code]

```
#!/usr/bin/env python
"""
show how to add a matplotlib FigureCanvasGTK or FigureCanvasGTKAgg widget and
a toolbar to a gtk.Window
"""
import gtk

from matplotlib.figure import Figure
from numpy import arange, sin, pi

# uncomment to select /GTK/GTKAgg/GTKCairo
#from matplotlib.backends.backend_gtk import FigureCanvasGTK as FigureCanvas
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
#from matplotlib.backends.backend_gtkcairo import FigureCanvasGTKCairo as FigureCanvas

# or NavigationToolbar for classic
#from matplotlib.backends.backend_gtk import NavigationToolbar2GTK as NavigationToolbar
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as␣
↪NavigationToolbar

# implement the default mpl key bindings
from matplotlib.backend_bases import key_press_handler

win = gtk.Window()
win.connect("destroy", lambda x: gtk.main_quit())
win.set_default_size(400, 300)
win.set_title("Embedding in GTK")

vbox = gtk.VBox()
win.add(vbox)

fig = Figure(figsize=(5, 4), dpi=100)
ax = fig.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)

ax.plot(t, s)


canvas = FigureCanvas(fig)  # a gtk.DrawingArea
vbox.pack_start(canvas)
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False)


def on_key_event(event):
    print('you pressed %s' % event.key)
    key_press_handler(event, canvas, toolbar)
```

```
canvas.mpl_connect('key_press_event', on_key_event)

win.show_all()
gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.3 user_interfaces example code: embedding_in_gtk3.py

[source code]

```
#!/usr/bin/env python
"""
demonstrate adding a FigureCanvasGTK3Agg widget to a Gtk.ScrolledWindow
using GTK3 accessed via pygobject
"""

from gi.repository import Gtk

from matplotlib.figure import Figure
from numpy import arange, sin, pi
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas

win = Gtk.Window()
win.connect("delete-event", Gtk.main_quit)
win.set_default_size(400, 300)
win.set_title("Embedding in GTK")

f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)
a.plot(t, s)

sw = Gtk.ScrolledWindow()
win.add(sw)
# A scrolled window border goes outside the scrollbars and viewport
sw.set_border_width(10)

canvas = FigureCanvas(f)  # a Gtk.DrawingArea
canvas.set_size_request(800, 600)
sw.add_with_viewport(canvas)

win.show_all()
Gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.4 user_interfaces example code: embedding_in_gtk3_panzoom.py

[source code]

```
#!/usr/bin/env python
"""
demonstrate NavigationToolbar with GTK3 accessed via pygobject
"""

from gi.repository import Gtk

from matplotlib.figure import Figure
from numpy import arange, sin, pi
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas
from matplotlib.backends.backend_gtk3 import NavigationToolbar2GTK3 as NavigationToolbar

win = Gtk.Window()
win.connect("delete-event", Gtk.main_quit)
win.set_default_size(400, 300)
win.set_title("Embedding in GTK")

f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(1, 1, 1)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)
a.plot(t, s)

vbox = Gtk.VBox()
win.add(vbox)

# Add canvas to vbox
canvas = FigureCanvas(f)  # a Gtk.DrawingArea
vbox.pack_start(canvas, True, True, 0)

# Create toolbar
toolbar = NavigationToolbar(canvas, win)
vbox.pack_start(toolbar, False, False, 0)

win.show_all()
Gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.5 user_interfaces example code: embedding_in_qt4.py

[source code]

```
#!/usr/bin/env python

# embedding_in_qt4.py --- Simple Qt4 application embedding matplotlib canvases
#
```

```
# Copyright (C) 2005 Florent Rougon
#               2006 Darren Dale
#
# This file is an example program for matplotlib. It may be used and
# modified with no restriction; raw copies as well as modified versions
# may be distributed without limitation.

from __future__ import unicode_literals
import sys
import os
import random
from matplotlib.backends import qt_compat
use_pyside = qt_compat.QT_API == qt_compat.QT_API_PYSIDE
if use_pyside:
    from PySide import QtGui, QtCore
else:
    from PyQt4 import QtGui, QtCore

from numpy import arange, sin, pi
from matplotlib.backends.backend_qt4agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure

progname = os.path.basename(sys.argv[0])
progversion = "0.1"


class MyMplCanvas(FigureCanvas):
    """Ultimately, this is a QWidget (as well as a FigureCanvasAgg, etc.)."""

    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        # We want the axes cleared every time plot() is called
        self.axes.hold(False)

        self.compute_initial_figure()

        #
        FigureCanvas.__init__(self, fig)
        self.setParent(parent)

        FigureCanvas.setSizePolicy(self,
                                   QtGui.QSizePolicy.Expanding,
                                   QtGui.QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)

    def compute_initial_figure(self):
        pass


class MyStaticMplCanvas(MyMplCanvas):
    """Simple canvas with a sine plot."""
```

```
    def compute_initial_figure(self):
        t = arange(0.0, 3.0, 0.01)
        s = sin(2*pi*t)
        self.axes.plot(t, s)


class MyDynamicMplCanvas(MyMplCanvas):
    """A canvas that updates itself every second with a new plot."""

    def __init__(self, *args, **kwargs):
        MyMplCanvas.__init__(self, *args, **kwargs)
        timer = QtCore.QTimer(self)
        timer.timeout.connect(self.update_figure)
        timer.start(1000)

    def compute_initial_figure(self):
        self.axes.plot([0, 1, 2, 3], [1, 2, 0, 4], 'r')

    def update_figure(self):
        # Build a list of 4 random integers between 0 and 10 (both inclusive)
        l = [random.randint(0, 10) for i in range(4)]

        self.axes.plot([0, 1, 2, 3], l, 'r')
        self.draw()


class ApplicationWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.setAttribute(QtCore.Qt.WA_DeleteOnClose)
        self.setWindowTitle("application main window")

        self.file_menu = QtGui.QMenu('&File', self)
        self.file_menu.addAction('&Quit', self.fileQuit,
                                 QtCore.Qt.CTRL + QtCore.Qt.Key_Q)
        self.menuBar().addMenu(self.file_menu)

        self.help_menu = QtGui.QMenu('&Help', self)
        self.menuBar().addSeparator()
        self.menuBar().addMenu(self.help_menu)

        self.help_menu.addAction('&About', self.about)

        self.main_widget = QtGui.QWidget(self)

        l = QtGui.QVBoxLayout(self.main_widget)
        sc = MyStaticMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        dc = MyDynamicMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        l.addWidget(sc)
        l.addWidget(dc)

        self.main_widget.setFocus()
        self.setCentralWidget(self.main_widget)
```

```
        self.statusBar().showMessage("All hail matplotlib!", 2000)

    def fileQuit(self):
        self.close()

    def closeEvent(self, ce):
        self.fileQuit()

    def about(self):
        QtGui.QMessageBox.about(self, "About",
                                """embedding_in_qt4.py example
Copyright 2005 Florent Rougon, 2006 Darren Dale

This program is a simple example of a Qt4 application embedding matplotlib
canvases.

It may be used and modified with no restriction; raw copies as well as
modified versions may be distributed without limitation."""
                                )


qApp = QtGui.QApplication(sys.argv)

aw = ApplicationWindow()
aw.setWindowTitle("%s" % progname)
aw.show()
sys.exit(qApp.exec_())
#qApp.exec_()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.6 user_interfaces example code: embedding_in_qt4_wtoolbar.py

[source code]

```
from __future__ import print_function

import sys

import numpy as np
from matplotlib.figure import Figure
from matplotlib.backend_bases import key_press_handler
from matplotlib.backends.backend_qt4agg import (
    FigureCanvasQTAgg as FigureCanvas,
    NavigationToolbar2QT as NavigationToolbar)
from matplotlib.backends import qt4_compat
use_pyside = qt4_compat.QT_API == qt4_compat.QT_API_PYSIDE

if use_pyside:
    from PySide.QtCore import *
```

```
    from PySide.QtGui import *
else:
    from PyQt4.QtCore import *
    from PyQt4.QtGui import *


class AppForm(QMainWindow):
    def __init__(self, parent=None):
        QMainWindow.__init__(self, parent)
        #self.x, self.y = self.get_data()
        self.data = self.get_data2()
        self.create_main_frame()
        self.on_draw()

    def create_main_frame(self):
        self.main_frame = QWidget()

        self.fig = Figure((5.0, 4.0), dpi=100)
        self.canvas = FigureCanvas(self.fig)
        self.canvas.setParent(self.main_frame)
        self.canvas.setFocusPolicy(Qt.StrongFocus)
        self.canvas.setFocus()

        self.mpl_toolbar = NavigationToolbar(self.canvas, self.main_frame)

        self.canvas.mpl_connect('key_press_event', self.on_key_press)

        vbox = QVBoxLayout()
        vbox.addWidget(self.canvas)  # the matplotlib canvas
        vbox.addWidget(self.mpl_toolbar)
        self.main_frame.setLayout(vbox)
        self.setCentralWidget(self.main_frame)

    def get_data2(self):
        return np.arange(20).reshape([4, 5]).copy()

    def on_draw(self):
        self.fig.clear()
        self.axes = self.fig.add_subplot(111)
        #self.axes.plot(self.x, self.y, 'ro')
        self.axes.imshow(self.data, interpolation='nearest')
        #self.axes.plot([1,2,3])
        self.canvas.draw()

    def on_key_press(self, event):
        print('you pressed', event.key)
        # implement the default mpl key press events described at
        # http://matplotlib.org/users/navigation_toolbar.html#navigation-keyboard-
↪shortcuts
        key_press_handler(event, self.canvas, self.mpl_toolbar)


def main():
```

```
    app = QApplication(sys.argv)
    form = AppForm()
    form.show()
    app.exec_()

if __name__ == "__main__":
    main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.7 user_interfaces example code: embedding_in_qt5.py

[source code]

```
#!/usr/bin/env python

# embedding_in_qt5.py --- Simple Qt5 application embedding matplotlib canvases
#
# Copyright (C) 2005 Florent Rougon
#               2006 Darren Dale
#               2015 Jens H Nielsen
#
# This file is an example program for matplotlib. It may be used and
# modified with no restriction; raw copies as well as modified versions
# may be distributed without limitation.

from __future__ import unicode_literals
import sys
import os
import random
import matplotlib
# Make sure that we are using QT5
matplotlib.use('Qt5Agg')
from PyQt5 import QtCore, QtWidgets

from numpy import arange, sin, pi
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure

progname = os.path.basename(sys.argv[0])
progversion = "0.1"


class MyMplCanvas(FigureCanvas):
    """Ultimately, this is a QWidget (as well as a FigureCanvasAgg, etc.)."""

    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        # We want the axes cleared every time plot() is called
        self.axes.hold(False)
```

```
        self.compute_initial_figure()

        #
        FigureCanvas.__init__(self, fig)
        self.setParent(parent)

        FigureCanvas.setSizePolicy(self,
                                   QtWidgets.QSizePolicy.Expanding,
                                   QtWidgets.QSizePolicy.Expanding)
        FigureCanvas.updateGeometry(self)

    def compute_initial_figure(self):
        pass


class MyStaticMplCanvas(MyMplCanvas):
    """Simple canvas with a sine plot."""

    def compute_initial_figure(self):
        t = arange(0.0, 3.0, 0.01)
        s = sin(2*pi*t)
        self.axes.plot(t, s)


class MyDynamicMplCanvas(MyMplCanvas):
    """A canvas that updates itself every second with a new plot."""

    def __init__(self, *args, **kwargs):
        MyMplCanvas.__init__(self, *args, **kwargs)
        timer = QtCore.QTimer(self)
        timer.timeout.connect(self.update_figure)
        timer.start(1000)

    def compute_initial_figure(self):
        self.axes.plot([0, 1, 2, 3], [1, 2, 0, 4], 'r')

    def update_figure(self):
        # Build a list of 4 random integers between 0 and 10 (both inclusive)
        l = [random.randint(0, 10) for i in range(4)]

        self.axes.plot([0, 1, 2, 3], l, 'r')
        self.draw()


class ApplicationWindow(QtWidgets.QMainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        self.setAttribute(QtCore.Qt.WA_DeleteOnClose)
        self.setWindowTitle("application main window")

        self.file_menu = QtWidgets.QMenu('&File', self)
        self.file_menu.addAction('&Quit', self.fileQuit,
```

```
                                 QtCore.Qt.CTRL + QtCore.Qt.Key_Q)
        self.menuBar().addMenu(self.file_menu)

        self.help_menu = QtWidgets.QMenu('&Help', self)
        self.menuBar().addSeparator()
        self.menuBar().addMenu(self.help_menu)

        self.help_menu.addAction('&About', self.about)

        self.main_widget = QtWidgets.QWidget(self)

        l = QtWidgets.QVBoxLayout(self.main_widget)
        sc = MyStaticMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        dc = MyDynamicMplCanvas(self.main_widget, width=5, height=4, dpi=100)
        l.addWidget(sc)
        l.addWidget(dc)

        self.main_widget.setFocus()
        self.setCentralWidget(self.main_widget)

        self.statusBar().showMessage("All hail matplotlib!", 2000)

    def fileQuit(self):
        self.close()

    def closeEvent(self, ce):
        self.fileQuit()

    def about(self):
        QtWidgets.QMessageBox.about(self, "About",
                                    """embedding_in_qt5.py example
Copyright 2005 Florent Rougon, 2006 Darren Dale, 2015 Jens H Nielsen

This program is a simple example of a Qt5 application embedding matplotlib
canvases.

It may be used and modified with no restriction; raw copies as well as
modified versions may be distributed without limitation.

This is modified from the embedding in qt4 example to show the difference
between qt4 and qt5"""
                                )


qApp = QtWidgets.QApplication(sys.argv)

aw = ApplicationWindow()
aw.setWindowTitle("%s" % progname)
aw.show()
sys.exit(qApp.exec_())
#qApp.exec_()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.8 user_interfaces example code: embedding_in_tk.py

[source code]

```
#!/usr/bin/env python

import matplotlib
matplotlib.use('TkAgg')

from numpy import arange, sin, pi
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg
# implement the default mpl key bindings
from matplotlib.backend_bases import key_press_handler


from matplotlib.figure import Figure

import sys
if sys.version_info[0] < 3:
    import Tkinter as Tk
else:
    import tkinter as Tk

root = Tk.Tk()
root.wm_title("Embedding in TK")


f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)

a.plot(t, s)


# a tk.DrawingArea
canvas = FigureCanvasTkAgg(f, master=root)
canvas.show()
canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

toolbar = NavigationToolbar2TkAgg(canvas, root)
toolbar.update()
canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)


def on_key_event(event):
    print('you pressed %s' % event.key)
    key_press_handler(event, canvas, toolbar)

canvas.mpl_connect('key_press_event', on_key_event)


def _quit():
```

```
    root.quit()     # stops mainloop
    root.destroy()  # this is necessary on Windows to prevent
                    # Fatal Python Error: PyEval_RestoreThread: NULL tstate

button = Tk.Button(master=root, text='Quit', command=_quit)
button.pack(side=Tk.BOTTOM)

Tk.mainloop()
# If you put root.destroy() here, it will cause an error if
# the window is closed with the window manager.
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.9 user_interfaces example code: embedding_in_tk2.py

[source code]

```
#!/usr/bin/env python
import matplotlib
matplotlib.use('TkAgg')

from numpy import arange, sin, pi
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure

import sys
if sys.version_info[0] < 3:
    import Tkinter as Tk
else:
    import tkinter as Tk


def destroy(e):
    sys.exit()

root = Tk.Tk()
root.wm_title("Embedding in TK")


f = Figure(figsize=(5, 4), dpi=100)
a = f.add_subplot(111)
t = arange(0.0, 3.0, 0.01)
s = sin(2*pi*t)

a.plot(t, s)
a.set_title('Tk embedding')
a.set_xlabel('X axis label')
a.set_ylabel('Y label')


# a tk.DrawingArea
```

```
canvas = FigureCanvasTkAgg(f, master=root)
canvas.show()
canvas.get_tk_widget().pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

canvas._tkcanvas.pack(side=Tk.TOP, fill=Tk.BOTH, expand=1)

button = Tk.Button(master=root, text='Quit', command=sys.exit)
button.pack(side=Tk.BOTTOM)

Tk.mainloop()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.10 user_interfaces example code: embedding_in_tk_canvas.py

[source code]

```
#!/usr/bin/env python
# -*- noplot -*-

import matplotlib as mpl
import numpy as np
import sys
if sys.version_info[0] < 3:
    import Tkinter as tk
else:
    import tkinter as tk
import matplotlib.backends.tkagg as tkagg
from matplotlib.backends.backend_agg import FigureCanvasAgg


def draw_figure(canvas, figure, loc=(0, 0)):
    """ Draw a matplotlib figure onto a Tk canvas

    loc: location of top-left corner of figure on canvas in pixels.

    Inspired by matplotlib source: lib/matplotlib/backends/backend_tkagg.py
    """
    figure_canvas_agg = FigureCanvasAgg(figure)
    figure_canvas_agg.draw()
    figure_x, figure_y, figure_w, figure_h = figure.bbox.bounds
    figure_w, figure_h = int(figure_w), int(figure_h)
    photo = tk.PhotoImage(master=canvas, width=figure_w, height=figure_h)

    # Position: convert from top-left anchor to center anchor
    canvas.create_image(loc[0] + figure_w/2, loc[1] + figure_h/2, image=photo)

    # Unfortunatly, there's no accessor for the pointer to the native renderer
    tkagg.blit(photo, figure_canvas_agg.get_renderer()._renderer, colormode=2)

    # Return a handle which contains a reference to the photo object
```

```
    # which must be kept live or else the picture disappears
    return photo

# Create a canvas
w, h = 300, 200
window = tk.Tk()
window.title("A figure in a canvas")
canvas = tk.Canvas(window, width=w, height=h)
canvas.pack()

# Generate some example data
X = np.linspace(0, 2.0*3.14, 50)
Y = np.sin(X)

# Create the figure we desire to add to an existing canvas
fig = mpl.figure.Figure(figsize=(2, 1))
ax = fig.add_axes([0, 0, 1, 1])
ax.plot(X, Y)

# Keep this handle alive, or else figure will disappear
fig_x, fig_y = 100, 100
fig_photo = draw_figure(canvas, fig, loc=(fig_x, fig_y))
fig_w, fig_h = fig_photo.width(), fig_photo.height()

# Add more elements to the canvas, potentially on top of the figure
canvas.create_line(200, 50, fig_x + fig_w / 2, fig_y + fig_h / 2)
canvas.create_text(200, 50, text="Zero-crossing", anchor="s")

# Let Tk take over
tk.mainloop()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.11 user_interfaces example code: embedding_in_wx2.py

[source code]

```
#!/usr/bin/env python
"""
An example of how to use wx or wxagg in an application with the new
toolbar - comment out the setA_toolbar line for no toolbar
"""

# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
#import wxversion
#wxversion.ensureMinimal('2.8')

from numpy import arange, sin, pi
```

```
import matplotlib

# uncomment the following to use wx rather than wxagg
#matplotlib.use('WX')
#from matplotlib.backends.backend_wx import FigureCanvasWx as FigureCanvas

# comment out the following to use wx rather than wxagg
matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas

from matplotlib.backends.backend_wx import NavigationToolbar2Wx

from matplotlib.figure import Figure

import wx
import wx.lib.mixins.inspection as WIT


class CanvasFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          'CanvasFrame', size=(550, 350))

        self.figure = Figure()
        self.axes = self.figure.add_subplot(111)
        t = arange(0.0, 3.0, 0.01)
        s = sin(2 * pi * t)

        self.axes.plot(t, s)
        self.canvas = FigureCanvas(self, -1, self.figure)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.canvas, 1, wx.LEFT | wx.TOP | wx.EXPAND)
        self.SetSizer(self.sizer)
        self.Fit()

        self.add_toolbar()  # comment this out for no toolbar

    def add_toolbar(self):
        self.toolbar = NavigationToolbar2Wx(self.canvas)
        self.toolbar.Realize()
        # By adding toolbar in sizer, we are able to put it at the bottom
        # of the frame - so appearance is closer to GTK version.
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        # update the axes menu on the toolbar
        self.toolbar.update()


# alternatively you could use
#class App(wx.App):
class App(WIT.InspectableApp):
    def OnInit(self):
        'Create the main window and insert the custom frame'
```

```
        self.Init()
        frame = CanvasFrame()
        frame.Show(True)

        return True

app = App(0)
app.MainLoop()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.12 user_interfaces example code: embedding_in_wx3.py

[source code]

```
#!/usr/bin/env python

"""
Copyright (C) 2003-2004 Andrew Straw, Jeremy O'Donoghue and others

License: This work is licensed under the PSF. A copy should be included
with this source code, and is also available at
http://www.python.org/psf/license.html

This is yet another example of using matplotlib with wx.  Hopefully
this is pretty full-featured:

  - both matplotlib toolbar and WX buttons manipulate plot
  - full wxApp framework, including widget interaction
  - XRC (XML wxWidgets resource) file to create GUI (made with XRCed)

This was derived from embedding_in_wx and dynamic_image_wxagg.

Thanks to matplotlib and wx teams for creating such great software!

"""
from __future__ import print_function

# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
#import wxversion
#wxversion.ensureMinimal('2.8')

import sys
import time
import os
import gc
import matplotlib
matplotlib.use('WXAgg')
import matplotlib.cm as cm
```

```python
import matplotlib.cbook as cbook
from matplotlib.backends.backend_wxagg import Toolbar, FigureCanvasWxAgg
from matplotlib.figure import Figure
import numpy as np

import wx
import wx.xrc as xrc

ERR_TOL = 1e-5  # floating point slop for peak-detection


matplotlib.rc('image', origin='lower')


class PlotPanel(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent, -1)

        self.fig = Figure((5, 4), 75)
        self.canvas = FigureCanvasWxAgg(self, -1, self.fig)
        self.toolbar = Toolbar(self.canvas)  # matplotlib toolbar
        self.toolbar.Realize()
        # self.toolbar.set_active([0,1])

        # Now put all into a sizer
        sizer = wx.BoxSizer(wx.VERTICAL)
        # This way of adding to sizer allows resizing
        sizer.Add(self.canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
        # Best to allow the toolbar to resize!
        sizer.Add(self.toolbar, 0, wx.GROW)
        self.SetSizer(sizer)
        self.Fit()

    def init_plot_data(self):
        a = self.fig.add_subplot(111)

        x = np.arange(120.0) * 2 * np.pi / 60.0
        y = np.arange(100.0) * 2 * np.pi / 50.0
        self.x, self.y = np.meshgrid(x, y)
        z = np.sin(self.x) + np.cos(self.y)
        self.im = a.imshow(z, cmap=cm.jet)  # , interpolation='nearest')

        zmax = np.amax(z) - ERR_TOL
        ymax_i, xmax_i = np.nonzero(z >= zmax)
        if self.im.origin == 'upper':
            ymax_i = z.shape[0] - ymax_i
        self.lines = a.plot(xmax_i, ymax_i, 'ko')

        self.toolbar.update()  # Not sure why this is needed - ADS

    def GetToolBar(self):
        # You will need to override GetToolBar if you are using an
        # unmanaged toolbar in your frame
```

```python
        return self.toolbar

    def OnWhiz(self, evt):
        self.x += np.pi / 15
        self.y += np.pi / 20
        z = np.sin(self.x) + np.cos(self.y)
        self.im.set_array(z)

        zmax = np.amax(z) - ERR_TOL
        ymax_i, xmax_i = np.nonzero(z >= zmax)
        if self.im.origin == 'upper':
            ymax_i = z.shape[0] - ymax_i
        self.lines[0].set_data(xmax_i, ymax_i)

        self.canvas.draw()

    def onEraseBackground(self, evt):
        # this is supposed to prevent redraw flicker on some X servers...
        pass


class MyApp(wx.App):
    def OnInit(self):
        xrcfile = cbook.get_sample_data('embedding_in_wx3.xrc',
                                        asfileobj=False)
        print('loading', xrcfile)

        self.res = xrc.XmlResource(xrcfile)

        # main frame and panel ---------

        self.frame = self.res.LoadFrame(None, "MainFrame")
        self.panel = xrc.XRCCTRL(self.frame, "MainPanel")

        # matplotlib panel -------------

        # container for matplotlib panel (I like to make a container
        # panel for our panel so I know where it'll go when in XRCed.)
        plot_container = xrc.XRCCTRL(self.frame, "plot_container_panel")
        sizer = wx.BoxSizer(wx.VERTICAL)

        # matplotlib panel itself
        self.plotpanel = PlotPanel(plot_container)
        self.plotpanel.init_plot_data()

        # wx boilerplate
        sizer.Add(self.plotpanel, 1, wx.EXPAND)
        plot_container.SetSizer(sizer)

        # whiz button ------------------
        whiz_button = xrc.XRCCTRL(self.frame, "whiz_button")
        whiz_button.Bind(wx.EVT_BUTTON, self.plotpanel.OnWhiz)
```

```
        # bang button ------------------
        bang_button = xrc.XRCCTRL(self.frame, "bang_button")
        bang_button.Bind(wx.EVT_BUTTON, self.OnBang)

        # final setup ------------------
        sizer = self.panel.GetSizer()
        self.frame.Show(1)

        self.SetTopWindow(self.frame)

        return True

    def OnBang(self, event):
        bang_count = xrc.XRCCTRL(self.frame, "bang_count")
        bangs = bang_count.GetValue()
        bangs = int(bangs) + 1
        bang_count.SetValue(str(bangs))

if __name__ == '__main__':
    app = MyApp(0)
    app.MainLoop()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.13 user_interfaces example code: embedding_in_wx4.py

[source code]

```
#!/usr/bin/env python
"""
An example of how to use wx or wxagg in an application with a custom
toolbar
"""

# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
#import wxversion
#wxversion.ensureMinimal('2.8')

from numpy import arange, sin, pi

import matplotlib

matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.backends.backend_wxagg import NavigationToolbar2WxAgg

from matplotlib.backends.backend_wx import _load_bitmap
from matplotlib.figure import Figure
from numpy.random import rand
```

```
import wx


class MyNavigationToolbar(NavigationToolbar2WxAgg):
    """
    Extend the default wx toolbar with your own event handlers
    """
    ON_CUSTOM = wx.NewId()

    def __init__(self, canvas, cankill):
        NavigationToolbar2WxAgg.__init__(self, canvas)

        # for simplicity I'm going to reuse a bitmap from wx, you'll
        # probably want to add your own.
        if 'phoenix' in wx.PlatformInfo:
            self.AddTool(self.ON_CUSTOM, 'Click me',
                         _load_bitmap('stock_left.xpm'),
                         'Activate custom contol')
            self.Bind(wx.EVT_TOOL, self._on_custom, id=self.ON_CUSTOM)
        else:
            self.AddSimpleTool(self.ON_CUSTOM, _load_bitmap('stock_left.xpm'),
                               'Click me', 'Activate custom contol')
            self.Bind(wx.EVT_TOOL, self._on_custom, id=self.ON_CUSTOM)

    def _on_custom(self, evt):
        # add some text to the axes in a random location in axes (0,1)
        # coords) with a random color

        # get the axes
        ax = self.canvas.figure.axes[0]

        # generate a random location can color
        x, y = tuple(rand(2))
        rgb = tuple(rand(3))

        # add the text and draw
        ax.text(x, y, 'You clicked me',
                transform=ax.transAxes,
                color=rgb)
        self.canvas.draw()
        evt.Skip()


class CanvasFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1,
                          'CanvasFrame', size=(550, 350))

        self.figure = Figure(figsize=(5, 4), dpi=100)
        self.axes = self.figure.add_subplot(111)
        t = arange(0.0, 3.0, 0.01)
        s = sin(2 * pi * t)
```

```
        self.axes.plot(t, s)

        self.canvas = FigureCanvas(self, -1, self.figure)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.canvas, 1, wx.TOP | wx.LEFT | wx.EXPAND)
        # Capture the paint message
        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.toolbar = MyNavigationToolbar(self.canvas, True)
        self.toolbar.Realize()
        # By adding toolbar in sizer, we are able to put it at the bottom
        # of the frame - so appearance is closer to GTK version.
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)

        # update the axes menu on the toolbar
        self.toolbar.update()
        self.SetSizer(self.sizer)
        self.Fit()

    def OnPaint(self, event):
        self.canvas.draw()
        event.Skip()


class App(wx.App):
    def OnInit(self):
        'Create the main window and insert the custom frame'
        frame = CanvasFrame()
        frame.Show(True)

        return True

app = App(0)
app.MainLoop()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.14 user_interfaces example code: embedding_in_wx5.py

[source code]

```
#!/usr/bin/env python

# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
#import wxversion
#wxversion.ensureMinimal('2.8')
```

```
import wx
import wx.lib.mixins.inspection as wit

if 'phoenix' in wx.PlatformInfo:
    import wx.lib.agw.aui as aui
else:
    import wx.aui as aui

import matplotlib as mpl
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as Canvas
from matplotlib.backends.backend_wxagg import NavigationToolbar2Wx as Toolbar


class Plot(wx.Panel):
    def __init__(self, parent, id=-1, dpi=None, **kwargs):
        wx.Panel.__init__(self, parent, id=id, **kwargs)
        self.figure = mpl.figure.Figure(dpi=dpi, figsize=(2, 2))
        self.canvas = Canvas(self, -1, self.figure)
        self.toolbar = Toolbar(self.canvas)
        self.toolbar.Realize()

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.canvas, 1, wx.EXPAND)
        sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        self.SetSizer(sizer)


class PlotNotebook(wx.Panel):
    def __init__(self, parent, id=-1):
        wx.Panel.__init__(self, parent, id=id)
        self.nb = aui.AuiNotebook(self)
        sizer = wx.BoxSizer()
        sizer.Add(self.nb, 1, wx.EXPAND)
        self.SetSizer(sizer)

    def add(self, name="plot"):
        page = Plot(self.nb)
        self.nb.AddPage(page, name)
        return page.figure


def demo():
    # alternatively you could use
    #app = wx.App()
    # InspectableApp is a great debug tool, see:
    # http://wiki.wxpython.org/Widget%20Inspection%20Tool
    app = wit.InspectableApp()
    frame = wx.Frame(None, -1, 'Plotter')
    plotter = PlotNotebook(frame)
    axes1 = plotter.add('figure 1').gca()
    axes1.plot([1, 2, 3], [2, 1, 4])
    axes2 = plotter.add('figure 2').gca()
    axes2.plot([1, 2, 3, 4, 5], [2, 1, 4, 2, 3])
```

```
    frame.Show()
    app.MainLoop()

if __name__ == "__main__":
    demo()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.15 user_interfaces example code: embedding_webagg.py

[source code]

```
"""
This example demonstrates how to embed matplotlib WebAgg interactive
plotting in your own web application and framework.  It is not
necessary to do all this if you merely want to display a plot in a
browser or use matplotlib's built-in Tornado-based server "on the
side".

The framework being used must support web sockets.
"""

import io

try:
    import tornado
except ImportError:
    raise RuntimeError("This example requires tornado.")
import tornado.web
import tornado.httpserver
import tornado.ioloop
import tornado.websocket


from matplotlib.backends.backend_webagg_core import (
    FigureManagerWebAgg, new_figure_manager_given_figure)
from matplotlib.figure import Figure

import numpy as np

import json


def create_figure():
    """
    Creates a simple example figure.
    """
    fig = Figure()
    a = fig.add_subplot(111)
    t = np.arange(0.0, 3.0, 0.01)
    s = np.sin(2 * np.pi * t)
```

```
    a.plot(t, s)
    return fig


# The following is the content of the web page.  You would normally
# generate this using some sort of template facility in your web
# framework, but here we just use Python string formatting.
html_content = """
<html>
  <head>
    <!-- TODO: There should be a way to include all of the required javascript
               and CSS so matplotlib can add to the set in the future if it
               needs to. -->
    <link rel="stylesheet" href="_static/css/page.css" type="text/css">
    <link rel="stylesheet" href="_static/css/boilerplate.css" type="text/css" />
    <link rel="stylesheet" href="_static/css/fbm.css" type="text/css" />
    <link rel="stylesheet" href="_static/jquery/css/themes/base/jquery-ui.min.css" >
    <script src="_static/jquery/js/jquery-1.11.3.min.js"></script>
    <script src="_static/jquery/js/jquery-ui.min.js"></script>
    <script src="mpl.js"></script>

    <script>
      /* This is a callback that is called when the user saves
         (downloads) a file.  Its purpose is really to map from a
         figure and file format to a url in the application. */
      function ondownload(figure, format) {
        window.open('download.' + format, '_blank');
      };

      $(document).ready(
        function() {
          /* It is up to the application to provide a websocket that the figure
             will use to communicate to the server.  This websocket object can
             also be a "fake" websocket that underneath multiplexes messages
             from multiple figures, if necessary. */
          var websocket_type = mpl.get_websocket_type();
          var websocket = new websocket_type("%(ws_uri)sws");

          // mpl.figure creates a new figure on the webpage.
          var fig = new mpl.figure(
              // A unique numeric identifier for the figure
              %(fig_id)s,
              // A websocket object (or something that behaves like one)
              websocket,
              // A function called when a file type is selected for download
              ondownload,
              // The HTML element in which to place the figure
              $('div#figure'));
        }
      );
    </script>

    <title>matplotlib</title>
```

```
  </head>

  <body>
    <div id="figure">
    </div>
  </body>
</html>
"""


class MyApplication(tornado.web.Application):
    class MainPage(tornado.web.RequestHandler):
        """
        Serves the main HTML page.
        """

        def get(self):
            manager = self.application.manager
            ws_uri = "ws://{req.host}/".format(req=self.request)
            content = html_content % {
                "ws_uri": ws_uri, "fig_id": manager.num}
            self.write(content)

    class MplJs(tornado.web.RequestHandler):
        """
        Serves the generated matplotlib javascript file.  The content
        is dynamically generated based on which toolbar functions the
        user has defined.  Call `FigureManagerWebAgg` to get its
        content.
        """

        def get(self):
            self.set_header('Content-Type', 'application/javascript')
            js_content = FigureManagerWebAgg.get_javascript()

            self.write(js_content)

    class Download(tornado.web.RequestHandler):
        """
        Handles downloading of the figure in various file formats.
        """

        def get(self, fmt):
            manager = self.application.manager

            mimetypes = {
                'ps': 'application/postscript',
                'eps': 'application/postscript',
                'pdf': 'application/pdf',
                'svg': 'image/svg+xml',
                'png': 'image/png',
                'jpeg': 'image/jpeg',
                'tif': 'image/tiff',
```

```
            'emf': 'application/emf'
        }

        self.set_header('Content-Type', mimetypes.get(fmt, 'binary'))

        buff = io.BytesIO()
        manager.canvas.print_figure(buff, format=fmt)
        self.write(buff.getvalue())

class WebSocket(tornado.websocket.WebSocketHandler):
    """
    A websocket for interactive communication between the plot in
    the browser and the server.

    In addition to the methods required by tornado, it is required to
    have two callback methods:

        - ``send_json(json_content)`` is called by matplotlib when
          it needs to send json to the browser.  `json_content` is
          a JSON tree (Python dictionary), and it is the responsibility
          of this implementation to encode it as a string to send over
          the socket.

        - ``send_binary(blob)`` is called to send binary image data
          to the browser.
    """
    supports_binary = True

    def open(self):
        # Register the websocket with the FigureManager.
        manager = self.application.manager
        manager.add_web_socket(self)
        if hasattr(self, 'set_nodelay'):
            self.set_nodelay(True)

    def on_close(self):
        # When the socket is closed, deregister the websocket with
        # the FigureManager.
        manager = self.application.manager
        manager.remove_web_socket(self)

    def on_message(self, message):
        # The 'supports_binary' message is relevant to the
        # websocket itself.  The other messages get passed along
        # to matplotlib as-is.

        # Every message has a "type" and a "figure_id".
        message = json.loads(message)
        if message['type'] == 'supports_binary':
            self.supports_binary = message['value']
        else:
            manager = self.application.manager
            manager.handle_json(message)
```

```
        def send_json(self, content):
            self.write_message(json.dumps(content))

        def send_binary(self, blob):
            if self.supports_binary:
                self.write_message(blob, binary=True)
            else:
                data_uri = "data:image/png;base64,{0}".format(
                    blob.encode('base64').replace('\n', ''))
                self.write_message(data_uri)

    def __init__(self, figure):
        self.figure = figure
        self.manager = new_figure_manager_given_figure(
            id(figure), figure)

        super(MyApplication, self).__init__([
            # Static files for the CSS and JS
            (r'/_static/(.*)',
             tornado.web.StaticFileHandler,
             {'path': FigureManagerWebAgg.get_static_file_path()}),

            # The page that contains all of the pieces
            ('/', self.MainPage),

            ('/mpl.js', self.MplJs),

            # Sends images and events to the browser, and receives
            # events from the browser
            ('/ws', self.WebSocket),

            # Handles the downloading (i.e., saving) of static images
            (r'/download.([a-z0-9.]+)', self.Download),
        ])


if __name__ == "__main__":
    figure = create_figure()
    application = MyApplication(figure)

    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(8080)

    print("http://127.0.0.1:8080/")
    print("Press Ctrl+C to quit")

    tornado.ioloop.IOLoop.instance().start()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.16 user_interfaces example code: fourier_demo_wx.py

[source code]

```
#!/usr/bin/env python
import numpy as np

# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
#import wxversion
#wxversion.ensureMinimal('2.8')

import wx
import matplotlib
matplotlib.interactive(False)
matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg
from matplotlib.figure import Figure
from matplotlib.pyplot import gcf, setp


class Knob(object):
    """
    Knob - simple class with a "setKnob" method.
    A Knob instance is attached to a Param instance, e.g., param.attach(knob)
    Base class is for documentation purposes.
    """

    def setKnob(self, value):
        pass


class Param(object):
    """
    The idea of the "Param" class is that some parameter in the GUI may have
    several knobs that both control it and reflect the parameter's state, e.g.
    a slider, text, and dragging can all change the value of the frequency in
    the waveform of this example.
    The class allows a cleaner way to update/"feedback" to the other knobs when
    one is being changed.  Also, this class handles min/max constraints for all
    the knobs.
    Idea - knob list - in "set" method, knob object is passed as well
      - the other knobs in the knob list have a "set" method which gets
        called for the others.
    """

    def __init__(self, initialValue=None, minimum=0., maximum=1.):
        self.minimum = minimum
        self.maximum = maximum
        if initialValue != self.constrain(initialValue):
            raise ValueError('illegal initial value')
        self.value = initialValue
```

```
        self.knobs = []

    def attach(self, knob):
        self.knobs += [knob]

    def set(self, value, knob=None):
        self.value = value
        self.value = self.constrain(value)
        for feedbackKnob in self.knobs:
            if feedbackKnob != knob:
                feedbackKnob.setKnob(self.value)
        return self.value

    def constrain(self, value):
        if value <= self.minimum:
            value = self.minimum
        if value >= self.maximum:
            value = self.maximum
        return value


class SliderGroup(Knob):
    def __init__(self, parent, label, param):
        self.sliderLabel = wx.StaticText(parent, label=label)
        self.sliderText = wx.TextCtrl(parent, -1, style=wx.TE_PROCESS_ENTER)
        self.slider = wx.Slider(parent, -1)
        # self.slider.SetMax(param.maximum*1000)
        self.slider.SetRange(0, param.maximum * 1000)
        self.setKnob(param.value)

        sizer = wx.BoxSizer(wx.HORIZONTAL)
        sizer.Add(self.sliderLabel, 0,
                  wx.EXPAND | wx.ALIGN_CENTER | wx.ALL,
                  border=2)
        sizer.Add(self.sliderText, 0,
                  wx.EXPAND | wx.ALIGN_CENTER | wx.ALL,
                  border=2)
        sizer.Add(self.slider, 1, wx.EXPAND)
        self.sizer = sizer

        self.slider.Bind(wx.EVT_SLIDER, self.sliderHandler)
        self.sliderText.Bind(wx.EVT_TEXT_ENTER, self.sliderTextHandler)

        self.param = param
        self.param.attach(self)

    def sliderHandler(self, evt):
        value = evt.GetInt() / 1000.
        self.param.set(value)

    def sliderTextHandler(self, evt):
        value = float(self.sliderText.GetValue())
        self.param.set(value)
```

```
    def setKnob(self, value):
        self.sliderText.SetValue('%g' % value)
        self.slider.SetValue(value * 1000)


class FourierDemoFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)

        self.fourierDemoWindow = FourierDemoWindow(self)
        self.frequencySliderGroup = SliderGroup(
            self,
            label='Frequency f0:',
            param=self.fourierDemoWindow.f0)
        self.amplitudeSliderGroup = SliderGroup(self, label=' Amplitude a:',
                                                param=self.fourierDemoWindow.A)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.fourierDemoWindow, 1, wx.EXPAND)
        sizer.Add(self.frequencySliderGroup.sizer, 0,
                  wx.EXPAND | wx.ALIGN_CENTER | wx.ALL, border=5)
        sizer.Add(self.amplitudeSliderGroup.sizer, 0,
                  wx.EXPAND | wx.ALIGN_CENTER | wx.ALL, border=5)
        self.SetSizer(sizer)


class FourierDemoWindow(wx.Window, Knob):
    def __init__(self, *args, **kwargs):
        wx.Window.__init__(self, *args, **kwargs)
        self.lines = []
        self.figure = Figure()
        self.canvas = FigureCanvasWxAgg(self, -1, self.figure)
        self.canvas.callbacks.connect('button_press_event', self.mouseDown)
        self.canvas.callbacks.connect('motion_notify_event', self.mouseMotion)
        self.canvas.callbacks.connect('button_release_event', self.mouseUp)
        self.state = ''
        self.mouseInfo = (None, None, None, None)
        self.f0 = Param(2., minimum=0., maximum=6.)
        self.A = Param(1., minimum=0.01, maximum=2.)
        self.draw()

        # Not sure I like having two params attached to the same Knob,
        # but that is what we have here... it works but feels kludgy -
        # although maybe it's not too bad since the knob changes both params
        # at the same time (both f0 and A are affected during a drag)
        self.f0.attach(self)
        self.A.attach(self)
        self.Bind(wx.EVT_SIZE, self.sizeHandler)

        self.Bind(wx.EVT_PAINT, self.OnPaint)

    def OnPaint(self, event):
```

```
        self.canvas.draw()
        event.Skip()

    def sizeHandler(self, *args, **kwargs):
        self.canvas.SetSize(self.GetSize())

    def mouseDown(self, evt):
        if self.lines[0] in self.figure.hitlist(evt):
            self.state = 'frequency'
        elif self.lines[1] in self.figure.hitlist(evt):
            self.state = 'time'
        else:
            self.state = ''
        self.mouseInfo = (evt.xdata, evt.ydata,
                          max(self.f0.value, .1),
                          self.A.value)

    def mouseMotion(self, evt):
        if self.state == '':
            return
        x, y = evt.xdata, evt.ydata
        if x is None:  # outside the axes
            return
        x0, y0, f0Init, AInit = self.mouseInfo
        self.A.set(AInit + (AInit * (y - y0) / y0), self)
        if self.state == 'frequency':
            self.f0.set(f0Init + (f0Init * (x - x0) / x0))
        elif self.state == 'time':
            if (x - x0) / x0 != -1.:
                self.f0.set(1. / (1. / f0Init + (1. / f0Init * (x - x0) / x0)))

    def mouseUp(self, evt):
        self.state = ''

    def draw(self):
        if not hasattr(self, 'subplot1'):
            self.subplot1 = self.figure.add_subplot(211)
            self.subplot2 = self.figure.add_subplot(212)
        x1, y1, x2, y2 = self.compute(self.f0.value, self.A.value)
        color = (1., 0., 0.)
        self.lines += self.subplot1.plot(x1, y1, color=color, linewidth=2)
        self.lines += self.subplot2.plot(x2, y2, color=color, linewidth=2)
        # Set some plot attributes
        self.subplot1.set_title(
            "Click and drag waveforms to change frequency and amplitude",
            fontsize=12)
        self.subplot1.set_ylabel("Frequency Domain Waveform X(f)", fontsize=8)
        self.subplot1.set_xlabel("frequency f", fontsize=8)
        self.subplot2.set_ylabel("Time Domain Waveform x(t)", fontsize=8)
        self.subplot2.set_xlabel("time t", fontsize=8)
        self.subplot1.set_xlim([-6, 6])
        self.subplot1.set_ylim([0, 1])
        self.subplot2.set_xlim([-2, 2])
```

```
        self.subplot2.set_ylim([-2, 2])
        self.subplot1.text(0.05, .95,
                           r'$X(f) = \mathcal{F}\{x(t)\}$',
                           verticalalignment='top',
                           transform=self.subplot1.transAxes)
        self.subplot2.text(0.05, .95,
                           r'$x(t) = a \cdot \cos(2\pi f_0 t) e^{-\pi t^2}$',
                           verticalalignment='top',
                           transform=self.subplot2.transAxes)

    def compute(self, f0, A):
        f = np.arange(-6., 6., 0.02)
        t = np.arange(-2., 2., 0.01)
        x = A * np.cos(2 * np.pi * f0 * t) * np.exp(-np.pi * t ** 2)
        X = A / 2 * \
            (np.exp(-np.pi * (f - f0) ** 2) + np.exp(-np.pi * (f + f0) ** 2))
        return f, X, t, x

    def repaint(self):
        self.canvas.draw()

    def setKnob(self, value):
        # Note, we ignore value arg here and just go by state of the params
        x1, y1, x2, y2 = self.compute(self.f0.value, self.A.value)
        setp(self.lines[0], xdata=x1, ydata=y1)
        setp(self.lines[1], xdata=x2, ydata=y2)
        self.repaint()


class App(wx.App):
    def OnInit(self):
        self.frame1 = FourierDemoFrame(parent=None, title="Fourier Demo",
                                       size=(640, 480))
        self.frame1.Show()
        return True

app = App()
app.MainLoop()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.17 user_interfaces example code: gtk_spreadsheet.py

[source code]

```
#!/usr/bin/env python
"""
Example of embedding matplotlib in an application and interacting with
a treeview to store data.  Double click on an entry to update plot
data

```

```
"""
import pygtk
pygtk.require('2.0')
import gtk
from gtk import gdk

import matplotlib
matplotlib.use('GTKAgg')  # or 'GTK'
from matplotlib.backends.backend_gtk import FigureCanvasGTK as FigureCanvas

from numpy.random import random
from matplotlib.figure import Figure


class DataManager(gtk.Window):
    numRows, numCols = 20, 10

    data = random((numRows, numCols))

    def __init__(self):
        gtk.Window.__init__(self)
        self.set_default_size(600, 600)
        self.connect('destroy', lambda win: gtk.main_quit())

        self.set_title('GtkListStore demo')
        self.set_border_width(8)

        vbox = gtk.VBox(False, 8)
        self.add(vbox)

        label = gtk.Label('Double click a row to plot the data')

        vbox.pack_start(label, False, False)

        sw = gtk.ScrolledWindow()
        sw.set_shadow_type(gtk.SHADOW_ETCHED_IN)
        sw.set_policy(gtk.POLICY_NEVER,
                      gtk.POLICY_AUTOMATIC)
        vbox.pack_start(sw, True, True)

        model = self.create_model()

        self.treeview = gtk.TreeView(model)
        self.treeview.set_rules_hint(True)

        # matplotlib stuff
        fig = Figure(figsize=(6, 4))

        self.canvas = FigureCanvas(fig)  # a gtk.DrawingArea
        vbox.pack_start(self.canvas, True, True)
        ax = fig.add_subplot(111)
        self.line, = ax.plot(self.data[0, :], 'go')  # plot the first row
```

```
        self.treeview.connect('row-activated', self.plot_row)
        sw.add(self.treeview)

        self.add_columns()

        self.add_events(gdk.BUTTON_PRESS_MASK |
                        gdk.KEY_PRESS_MASK |
                        gdk.KEY_RELEASE_MASK)

    def plot_row(self, treeview, path, view_column):
        ind, = path  # get the index into data
        points = self.data[ind, :]
        self.line.set_ydata(points)
        self.canvas.draw()

    def add_columns(self):
        for i in range(self.numCols):
            column = gtk.TreeViewColumn('%d' % i, gtk.CellRendererText(), text=i)
            self.treeview.append_column(column)

    def create_model(self):
        types = [float]*self.numCols
        store = gtk.ListStore(*types)

        for row in self.data:
            store.append(row)
        return store


manager = DataManager()
manager.show_all()
gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.18 user_interfaces example code: histogram_demo_canvasagg.py

[source code]

```
#!/usr/bin/env python
"""
This is an example that shows you how to work directly with the agg
figure canvas to create a figure using the pythonic API.

In this example, the contents of the agg canvas are extracted to a
string, which can in turn be passed off to PIL or put in a numeric
array


"""
from matplotlib.backends.backend_agg import FigureCanvasAgg
```

```python
from matplotlib.figure import Figure
from matplotlib.mlab import normpdf
from numpy.random import randn
import numpy

fig = Figure(figsize=(5, 4), dpi=100)
ax = fig.add_subplot(111)

canvas = FigureCanvasAgg(fig)

mu, sigma = 100, 15
x = mu + sigma*randn(10000)

# the histogram of the data
n, bins, patches = ax.hist(x, 50, normed=1)

# add a 'best fit' line
y = normpdf(bins, mu, sigma)
line, = ax.plot(bins, y, 'r--')
line.set_linewidth(1)

ax.set_xlabel('Smarts')
ax.set_ylabel('Probability')
ax.set_title(r'$\mathrm{Histogram of IQ: }\mu=100, \sigma=15$')

ax.set_xlim((40, 160))
ax.set_ylim((0, 0.03))

canvas.draw()

s = canvas.tostring_rgb()  # save this and convert to bitmap as needed

# get the figure dimensions for creating bitmaps or numpy arrays,
# etc.
l, b, w, h = fig.bbox.bounds
w, h = int(w), int(h)

if 0:
    # convert to a numpy array
    X = numpy.fromstring(s, numpy.uint8)
    X.shape = h, w, 3

if 0:
    # pass off to PIL
    from PIL import Image
    im = Image.fromstring("RGB", (w, h), s)
    im.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.19 user_interfaces example code: interactive.py

[source code]

```
#!/usr/bin/env python

"""Multithreaded interactive interpreter with GTK and Matplotlib support.

WARNING:
As of 2010/06/25, this is not working, at least on Linux.
I have disabled it as a runnable script. - EF


Usage:

  pyint-gtk.py -> starts shell with gtk thread running separately

  pyint-gtk.py -pylab [filename] -> initializes matplotlib, optionally running
  the named file.  The shell starts after the file is executed.

Threading code taken from:
http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/65109, by Brian
McErlean and John Finlay.

Matplotlib support taken from interactive.py in the matplotlib distribution.

Also borrows liberally from code.py in the Python standard library."""

from __future__ import print_function

__author__ = "Fernando Perez <Fernando.Perez@colorado.edu>"

import sys
import code
import threading

import gobject
import gtk

try:
    import readline
except ImportError:
    has_readline = False
else:
    has_readline = True


class MTConsole(code.InteractiveConsole):
    """Simple multi-threaded shell"""

    def __init__(self, on_kill=None, *args, **kw):
        code.InteractiveConsole.__init__(self, *args, **kw)
        self.code_to_run = None
```

```
        self.ready = threading.Condition()
        self._kill = False
        if on_kill is None:
            on_kill = []
        # Check that all things to kill are callable:
        for _ in on_kill:
            if not callable(_):
                raise TypeError('on_kill must be a list of callables')
        self.on_kill = on_kill
        # Set up tab-completer
        if has_readline:
            import rlcompleter
            try:  # this form only works with python 2.3
                self.completer = rlcompleter.Completer(self.locals)
            except:  # simpler for py2.2
                self.completer = rlcompleter.Completer()

            readline.set_completer(self.completer.complete)
            # Use tab for completions
            readline.parse_and_bind('tab: complete')
            # This forces readline to automatically print the above list when tab
            # completion is set to 'complete'.
            readline.parse_and_bind('set show-all-if-ambiguous on')
            # Bindings for incremental searches in the history. These searches
            # use the string typed so far on the command line and search
            # anything in the previous input history containing them.
            readline.parse_and_bind('"\C-r": reverse-search-history')
            readline.parse_and_bind('"\C-s": forward-search-history')

    def runsource(self, source, filename="<input>", symbol="single"):
        """Compile and run some source in the interpreter.

        Arguments are as for compile_command().

        One several things can happen:

        1) The input is incorrect; compile_command() raised an
        exception (SyntaxError or OverflowError).  A syntax traceback
        will be printed by calling the showsyntaxerror() method.

        2) The input is incomplete, and more input is required;
        compile_command() returned None.  Nothing happens.

        3) The input is complete; compile_command() returned a code
        object.  The code is executed by calling self.runcode() (which
        also handles run-time exceptions, except for SystemExit).

        The return value is True in case 2, False in the other cases (unless
        an exception is raised).  The return value can be used to
        decide whether to use sys.ps1 or sys.ps2 to prompt the next
        line.
        """
        try:
```

```
            code = self.compile(source, filename, symbol)
        except (OverflowError, SyntaxError, ValueError):
            # Case 1
            self.showsyntaxerror(filename)
            return False

        if code is None:
            # Case 2
            return True

        # Case 3
        # Store code in self, so the execution thread can handle it
        self.ready.acquire()
        self.code_to_run = code
        self.ready.wait()  # Wait until processed in timeout interval
        self.ready.release()

        return False

    def runcode(self):
        """Execute a code object.

        When an exception occurs, self.showtraceback() is called to display a
        traceback."""

        self.ready.acquire()
        if self._kill:
            print('Closing threads...')
            sys.stdout.flush()
            for tokill in self.on_kill:
                tokill()
            print('Done.')

        if self.code_to_run is not None:
            self.ready.notify()
            code.InteractiveConsole.runcode(self, self.code_to_run)

        self.code_to_run = None
        self.ready.release()
        return True

    def kill(self):
        """Kill the thread, returning when it has been shut down."""
        self.ready.acquire()
        self._kill = True
        self.ready.release()


class GTKInterpreter(threading.Thread):
    """Run gtk.main in the main thread and a python interpreter in a
    separate thread.
    Python commands can be passed to the thread where they will be executed.
    This is implemented by periodically checking for passed code using a
```

```
    GTK timeout callback.
    """
    TIMEOUT = 100  # Millisecond interval between timeouts.

    def __init__(self, banner=None):
        threading.Thread.__init__(self)
        self.banner = banner
        self.shell = MTConsole(on_kill=[gtk.main_quit])

    def run(self):
        self.pre_interact()
        self.shell.interact(self.banner)
        self.shell.kill()

    def mainloop(self):
        self.start()
        gobject.timeout_add(self.TIMEOUT, self.shell.runcode)
        try:
            if gtk.gtk_version[0] >= 2:
                gtk.gdk.threads_init()
        except AttributeError:
            pass
        gtk.main()
        self.join()

    def pre_interact(self):
        """This method should be overridden by subclasses.

        It gets called right before interact(), but after the thread starts.
        Typically used to push initialization code into the interpreter"""

        pass


class MatplotLibInterpreter(GTKInterpreter):
    """Threaded interpreter with matplotlib support.

    Note that this explicitly sets GTKAgg as the backend, since it has
    specific GTK hooks in it."""

    def __init__(self, banner=None):
        banner = """\nWelcome to matplotlib, a MATLAB-like python environment.
    help(matlab)   -> help on matlab compatible commands from matplotlib.
    help(plotting) -> help on plotting commands.
    """
        GTKInterpreter.__init__(self, banner)

    def pre_interact(self):
        """Initialize matplotlib before user interaction begins"""

        push = self.shell.push
        # Code to execute in user's namespace
        lines = ["import matplotlib",
```

```
                 "matplotlib.use('GTKAgg')",
                 "matplotlib.interactive(1)",
                 "import matplotlib.pylab as pylab",
                 "from matplotlib.pylab import *\n"]

        map(push, lines)

        # Execute file if given.
        if len(sys.argv) > 1:
            import matplotlib
            matplotlib.interactive(0)  # turn off interaction
            fname = sys.argv[1]
            try:
                inFile = file(fname, 'r')
            except IOError:
                print('*** ERROR *** Could not read file <%s>' % fname)
            else:
                print('*** Executing file <%s>:' % fname)
                for line in inFile:
                    if line.lstrip().find('show()') == 0:
                        continue
                    print('>>', line)
                    push(line)
                inFile.close()
            matplotlib.interactive(1)   # turn on interaction

if __name__ == '__main__':
    print("This demo is not presently functional, so running")
    print("it as a script has been disabled.")
    sys.exit()
    # Quick sys.argv hack to extract the option and leave filenames in sys.argv.
    # For real option handling, use optparse or getopt.
    if len(sys.argv) > 1 and sys.argv[1] == '-pylab':
        sys.argv = [sys.argv[0]] + sys.argv[2:]
        MatplotLibInterpreter().mainloop()
    else:
        GTKInterpreter().mainloop()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.20 user_interfaces example code: interactive2.py

[source code]

```
#!/usr/bin/env python

from __future__ import print_function

#  GTK Interactive Console
#  (C) 2003, Jon Anderson
#  See www.python.org/2.2/license.html for
```

```
#  license details.
#
import gtk
import gtk.gdk

import code
import os
import sys
import pango

import __builtin__
import __main__


banner = """GTK Interactive Python Console
Thanks to Jon Anderson
%s
""" % sys.version

banner += """

Welcome to matplotlib.

    help(matplotlib) -- some general information about matplotlib
    help(plotting) -- shows a list of plot specific commands

"""


class Completer(object):
    """
    Taken from rlcompleter, with readline references stripped, and a local dictionary to␣
↪use.
    """

    def __init__(self, locals):
        self.locals = locals

    def complete(self, text, state):
        """Return the next possible completion for 'text'.
        This is called successively with state == 0, 1, 2, ... until it
        returns None.  The completion should begin with 'text'.

        """
        if state == 0:
            if "." in text:
                self.matches = self.attr_matches(text)
            else:
                self.matches = self.global_matches(text)
        try:
            return self.matches[state]
        except IndexError:
            return None
```

```
    def global_matches(self, text):
        """Compute matches when text is a simple name.

        Return a list of all keywords, built-in functions and names
        currently defines in __main__ that match.

        """
        import keyword
        matches = []
        n = len(text)
        for list in [keyword.kwlist, __builtin__.__dict__.keys(), __main__.__dict__.
↪keys(), self.locals.keys()]:
            for word in list:
                if word[:n] == text and word != "__builtins__":
                    matches.append(word)
        return matches

    def attr_matches(self, text):
        """Compute matches when text contains a dot.

        Assuming the text is of the form NAME.NAME....[NAME], and is
        evaluatable in the globals of __main__, it will be evaluated
        and its attributes (as revealed by dir()) are used as possible
        completions.  (For class instances, class members are are also
        considered.)

        WARNING: this can still invoke arbitrary C code, if an object
        with a __getattr__ hook is evaluated.

        """
        import re
        m = re.match(r"(\w+(\.\w+)*)\.(\w*)", text)
        if not m:
            return
        expr, attr = m.group(1, 3)
        object = eval(expr, __main__.__dict__, self.locals)
        words = dir(object)
        if hasattr(object, '__class__'):
            words.append('__class__')
            words = words + get_class_members(object.__class__)
        matches = []
        n = len(attr)
        for word in words:
            if word[:n] == attr and word != "__builtins__":
                matches.append("%s.%s" % (expr, word))
        return matches


def get_class_members(klass):
    ret = dir(klass)
    if hasattr(klass, '__bases__'):
        for base in klass.__bases__:
```

```
            ret = ret + get_class_members(base)
    return ret


class OutputStream(object):
    """
    A Multiplexing output stream.
    It can replace another stream, and tee output to the original stream and too
    a GTK textview.
    """

    def __init__(self, view, old_out, style):
        self.view = view
        self.buffer = view.get_buffer()
        self.mark = self.buffer.create_mark("End", self.buffer.get_end_iter(), False)
        self.out = old_out
        self.style = style
        self.tee = 1

    def write(self, text):
        if self.tee:
            self.out.write(text)

        end = self.buffer.get_end_iter()

        if self.view is not None:
            self.view.scroll_to_mark(self.mark, 0, True, 1, 1)

        self.buffer.insert_with_tags(end, text, self.style)


class GTKInterpreterConsole(gtk.ScrolledWindow):
    """
    An InteractiveConsole for GTK. It's an actual widget,
    so it can be dropped in just about anywhere.
    """

    def __init__(self):
        gtk.ScrolledWindow.__init__(self)
        self.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)

        self.text = gtk.TextView()
        self.text.set_wrap_mode(True)

        self.interpreter = code.InteractiveInterpreter()

        self.completer = Completer(self.interpreter.locals)
        self.buffer = []
        self.history = []
        self.banner = banner
        self.ps1 = ">>> "
        self.ps2 = "... "
```

```
        self.text.add_events(gtk.gdk.KEY_PRESS_MASK)
        self.text.connect("key_press_event", self.key_pressed)

        self.current_history = -1

        self.mark = self.text.get_buffer().create_mark("End", self.text.get_buffer().get_
↪end_iter(), False)

        # setup colors
        self.style_banner = gtk.TextTag("banner")
        self.style_banner.set_property("foreground", "saddle brown")

        self.style_ps1 = gtk.TextTag("ps1")
        self.style_ps1.set_property("foreground", "DarkOrchid4")
        self.style_ps1.set_property("editable", False)
        self.style_ps1.set_property("font", "courier")

        self.style_ps2 = gtk.TextTag("ps2")
        self.style_ps2.set_property("foreground", "DarkOliveGreen")
        self.style_ps2.set_property("editable", False)
        self.style_ps2.set_property("font", "courier")

        self.style_out = gtk.TextTag("stdout")
        self.style_out.set_property("foreground", "midnight blue")
        self.style_err = gtk.TextTag("stderr")
        self.style_err.set_property("style", pango.STYLE_ITALIC)
        self.style_err.set_property("foreground", "red")

        self.text.get_buffer().get_tag_table().add(self.style_banner)
        self.text.get_buffer().get_tag_table().add(self.style_ps1)
        self.text.get_buffer().get_tag_table().add(self.style_ps2)
        self.text.get_buffer().get_tag_table().add(self.style_out)
        self.text.get_buffer().get_tag_table().add(self.style_err)

        self.stdout = OutputStream(self.text, sys.stdout, self.style_out)
        self.stderr = OutputStream(self.text, sys.stderr, self.style_err)

        sys.stderr = self.stderr
        sys.stdout = self.stdout

        self.current_prompt = None

        self.write_line(self.banner, self.style_banner)
        self.prompt_ps1()

        self.add(self.text)
        self.text.show()

    def reset_history(self):
        self.history = []

    def reset_buffer(self):
        self.buffer = []
```

```
    def prompt_ps1(self):
        self.current_prompt = self.prompt_ps1
        self.write_line(self.ps1, self.style_ps1)

    def prompt_ps2(self):
        self.current_prompt = self.prompt_ps2
        self.write_line(self.ps2, self.style_ps2)

    def write_line(self, text, style=None):
        start, end = self.text.get_buffer().get_bounds()
        if style is None:
            self.text.get_buffer().insert(end, text)
        else:
            self.text.get_buffer().insert_with_tags(end, text, style)

        self.text.scroll_to_mark(self.mark, 0, True, 1, 1)

    def push(self, line):

        self.buffer.append(line)
        if len(line) > 0:
            self.history.append(line)

        source = "\n".join(self.buffer)

        more = self.interpreter.runsource(source, "<<console>>")

        if not more:
            self.reset_buffer()

        return more

    def key_pressed(self, widget, event):
        if event.keyval == gtk.gdk.keyval_from_name('Return'):
            return self.execute_line()

        if event.keyval == gtk.gdk.keyval_from_name('Up'):
            self.current_history = self.current_history - 1
            if self.current_history < - len(self.history):
                self.current_history = - len(self.history)
            return self.show_history()
        elif event.keyval == gtk.gdk.keyval_from_name('Down'):
            self.current_history = self.current_history + 1
            if self.current_history > 0:
                self.current_history = 0
            return self.show_history()
        elif event.keyval == gtk.gdk.keyval_from_name('Home'):
            l = self.text.get_buffer().get_line_count() - 1
            start = self.text.get_buffer().get_iter_at_line_offset(l, 4)
            self.text.get_buffer().place_cursor(start)
            return True
        elif event.keyval == gtk.gdk.keyval_from_name('space') and event.state & gtk.gdk.
↪CONTROL_MASK:
```

```
            return self.complete_line()
        return False

    def show_history(self):
        if self.current_history == 0:
            return True
        else:
            self.replace_line(self.history[self.current_history])
            return True

    def current_line(self):
        start, end = self.current_line_bounds()
        return self.text.get_buffer().get_text(start, end, True)

    def current_line_bounds(self):
        txt_buffer = self.text.get_buffer()
        l = txt_buffer.get_line_count() - 1

        start = txt_buffer.get_iter_at_line(l)
        if start.get_chars_in_line() >= 4:
            start.forward_chars(4)
        end = txt_buffer.get_end_iter()
        return start, end

    def replace_line(self, txt):
        start, end = self.current_line_bounds()
        self.text.get_buffer().delete(start, end)
        self.write_line(txt)

    def execute_line(self, line=None):
        if line is None:
            line = self.current_line()
            self.write_line("\n")
        else:
            self.write_line(line + "\n")

        more = self.push(line)

        self.text.get_buffer().place_cursor(self.text.get_buffer().get_end_iter())

        if more:
            self.prompt_ps2()
        else:
            self.prompt_ps1()

        self.current_history = 0

        self.window.raise_()

        return True

    def complete_line(self):
        line = self.current_line()
```

```
        tokens = line.split()
        token = tokens[-1]

        completions = []
        p = self.completer.complete(token, len(completions))
        while p is not None:
            completions.append(p)
            p = self.completer.complete(token, len(completions))

        if len(completions) != 1:
            self.write_line("\n")
            self.write_line("\n".join(completions), self.style_ps1)
            self.write_line("\n")
            self.current_prompt()
            self.write_line(line)
        else:
            i = line.rfind(token)
            line = line[0:i] + completions[0]
            self.replace_line(line)

        return True


def main():
    w = gtk.Window()
    console = GTKInterpreterConsole()
    console.set_size_request(640, 480)
    w.add(console)

    def destroy(arg=None):
        gtk.main_quit()

    def key_event(widget, event):
        if gtk.gdk.keyval_name(event.keyval) == 'd' and \
                event.state & gtk.gdk.CONTROL_MASK:
            destroy()
        return False

    w.connect("destroy", destroy)

    w.add_events(gtk.gdk.KEY_PRESS_MASK)
    w.connect('key_press_event', key_event)
    w.show_all()

    console.execute_line('import matplotlib')
    console.execute_line("matplotlib.use('GTKAgg')")
    console.execute_line('matplotlib.interactive(1)')
    console.execute_line('from pylab import *')

    if len(sys.argv) > 1:
        fname = sys.argv[1]
        if not os.path.exists(fname):
            print('%s does not exist' % fname)
```

```
        for line in file(fname):
            line = line.strip()

            console.execute_line(line)
    gtk.main()

if __name__ == '__main__':
    main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.21 user_interfaces example code: lineprops_dialog_gtk.py

[source code]

```
import matplotlib
matplotlib.use('GTKAgg')
from matplotlib.backends.backend_gtk import DialogLineprops

import numpy as np
import matplotlib.pyplot as plt


def f(t):
    s1 = np.cos(2*np.pi*t)
    e1 = np.exp(-t)
    return np.multiply(s1, e1)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
t3 = np.arange(0.0, 2.0, 0.01)

fig, ax = plt.subplots()
l1, = ax.plot(t1, f(t1), 'bo', label='line 1')
l2, = ax.plot(t2, f(t2), 'k--', label='line 2')

dlg = DialogLineprops([l1, l2])
dlg.show()
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.22 user_interfaces example code: mathtext_wx.py

[source code]

```
"""
Demonstrates how to convert mathtext to a wx.Bitmap for display in various
controls on wxPython.
```

```
"""

import matplotlib
matplotlib.use("WxAgg")
from numpy import arange, sin, pi, cos, log
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.backends.backend_wx import NavigationToolbar2Wx, wxc
from matplotlib.figure import Figure

import wx

IS_GTK = 'wxGTK' in wx.PlatformInfo
IS_WIN = 'wxMSW' in wx.PlatformInfo

############################################################
# This is where the "magic" happens.
from matplotlib.mathtext import MathTextParser
mathtext_parser = MathTextParser("Bitmap")


def mathtext_to_wxbitmap(s):
    ftimage, depth = mathtext_parser.parse(s, 150)
    return wxc.BitmapFromBuffer(
        ftimage.get_width(), ftimage.get_height(),
        ftimage.as_rgba_str())
############################################################

functions = [
    (r'$\sin(2 \pi x)$', lambda x: sin(2*pi*x)),
    (r'$\frac{4}{3}\pi x^3$', lambda x: (4.0/3.0)*pi*x**3),
    (r'$\cos(2 \pi x)$', lambda x: cos(2*pi*x)),
    (r'$\log(x)$', lambda x: log(x))
]


class CanvasFrame(wx.Frame):
    def __init__(self, parent, title):
        wx.Frame.__init__(self, parent, -1, title, size=(550, 350))
        self.SetBackgroundColour(wxc.NamedColour("WHITE"))

        self.figure = Figure()
        self.axes = self.figure.add_subplot(111)

        self.canvas = FigureCanvas(self, -1, self.figure)

        self.change_plot(0)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.add_buttonbar()
        self.sizer.Add(self.canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
        self.add_toolbar()  # comment this out for no toolbar

        menuBar = wx.MenuBar()
```

```
        # File Menu
        menu = wx.Menu()
        menu.Append(wx.ID_EXIT, "E&xit\tAlt-X", "Exit this simple sample")
        menuBar.Append(menu, "&File")

        if IS_GTK or IS_WIN:
            # Equation Menu
            menu = wx.Menu()
            for i, (mt, func) in enumerate(functions):
                bm = mathtext_to_wxbitmap(mt)
                item = wx.MenuItem(menu, 1000 + i, " ")
                item.SetBitmap(bm)
                menu.AppendItem(item)
                self.Bind(wx.EVT_MENU, self.OnChangePlot, item)
            menuBar.Append(menu, "&Functions")

        self.SetMenuBar(menuBar)

        self.SetSizer(self.sizer)
        self.Fit()

    def add_buttonbar(self):
        self.button_bar = wx.Panel(self)
        self.button_bar_sizer = wx.BoxSizer(wx.HORIZONTAL)
        self.sizer.Add(self.button_bar, 0, wx.LEFT | wx.TOP | wx.GROW)

        for i, (mt, func) in enumerate(functions):
            bm = mathtext_to_wxbitmap(mt)
            button = wx.BitmapButton(self.button_bar, 1000 + i, bm)
            self.button_bar_sizer.Add(button, 1, wx.GROW)
            self.Bind(wx.EVT_BUTTON, self.OnChangePlot, button)

        self.button_bar.SetSizer(self.button_bar_sizer)

    def add_toolbar(self):
        """Copied verbatim from embedding_wx2.py"""
        self.toolbar = NavigationToolbar2Wx(self.canvas)
        self.toolbar.Realize()
        # By adding toolbar in sizer, we are able to put it at the bottom
        # of the frame - so appearance is closer to GTK version.
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        # update the axes menu on the toolbar
        self.toolbar.update()

    def OnChangePlot(self, event):
        self.change_plot(event.GetId() - 1000)

    def change_plot(self, plot_number):
        t = arange(1.0, 3.0, 0.01)
        s = functions[plot_number][1](t)
        self.axes.clear()
        self.axes.plot(t, s)
```

```
        self.canvas.draw()


class MyApp(wx.App):
    def OnInit(self):
        frame = CanvasFrame(None, "wxPython mathtext demo app")
        self.SetTopWindow(frame)
        frame.Show(True)
        return True

app = MyApp()
app.MainLoop()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.23 user_interfaces example code: mpl_with_glade.py

[source code]

```
#!/usr/bin/env python

from __future__ import print_function
import matplotlib
matplotlib.use('GTK')

from matplotlib.figure import Figure
from matplotlib.axes import Subplot
from matplotlib.backends.backend_gtkagg import FigureCanvasGTKAgg as FigureCanvas
from matplotlib.backends.backend_gtkagg import NavigationToolbar2GTKAgg as␣
↪NavigationToolbar
from matplotlib.widgets import SpanSelector

from numpy import arange, sin, pi
import gtk
import gtk.glade


def simple_msg(msg, parent=None, title=None):
    dialog = gtk.MessageDialog(
        parent=None,
        type=gtk.MESSAGE_INFO,
        buttons=gtk.BUTTONS_OK,
        message_format=msg)
    if parent is not None:
        dialog.set_transient_for(parent)
    if title is not None:
        dialog.set_title(title)
    dialog.show()
    dialog.run()
    dialog.destroy()
    return None
```

```
class GladeHandlers(object):
    def on_buttonClickMe_clicked(event):
        simple_msg('Nothing to say, really',
                   parent=widgets['windowMain'],
                   title='Thanks!')


class WidgetsWrapper(object):
    def __init__(self):
        self.widgets = gtk.glade.XML('mpl_with_glade.glade')
        self.widgets.signal_autoconnect(GladeHandlers.__dict__)

        self['windowMain'].connect('destroy', lambda x: gtk.main_quit())
        self['windowMain'].move(10, 10)
        self.figure = Figure(figsize=(8, 6), dpi=72)
        self.axis = self.figure.add_subplot(111)

        t = arange(0.0, 3.0, 0.01)
        s = sin(2*pi*t)
        self.axis.plot(t, s)
        self.axis.set_xlabel('time (s)')
        self.axis.set_ylabel('voltage')

        self.canvas = FigureCanvas(self.figure)  # a gtk.DrawingArea
        self.canvas.show()
        self.canvas.set_size_request(600, 400)
        self.canvas.set_events(
            gtk.gdk.BUTTON_PRESS_MASK |
            gtk.gdk.KEY_PRESS_MASK |
            gtk.gdk.KEY_RELEASE_MASK
            )
        self.canvas.set_flags(gtk.HAS_FOCUS | gtk.CAN_FOCUS)
        self.canvas.grab_focus()

        def keypress(widget, event):
            print('key press')

        def buttonpress(widget, event):
            print('button press')

        self.canvas.connect('key_press_event', keypress)
        self.canvas.connect('button_press_event', buttonpress)

        def onselect(xmin, xmax):
            print(xmin, xmax)

        span = SpanSelector(self.axis, onselect, 'horizontal', useblit=False,
                            rectprops=dict(alpha=0.5, facecolor='red'))

        self['vboxMain'].pack_start(self.canvas, True, True)
        self['vboxMain'].show()
```

```
        # below is optional if you want the navigation toolbar
        self.navToolbar = NavigationToolbar(self.canvas, self['windowMain'])
        self.navToolbar.lastDir = '/var/tmp/'
        self['vboxMain'].pack_start(self.navToolbar)
        self.navToolbar.show()

        sep = gtk.HSeparator()
        sep.show()
        self['vboxMain'].pack_start(sep, True, True)

        self['vboxMain'].reorder_child(self['buttonClickMe'], -1)

    def __getitem__(self, key):
        return self.widgets.get_widget(key)

widgets = WidgetsWrapper()
gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.24 user_interfaces example code: mpl_with_glade_316.py

[source code]

```
#!/usr/bin/env python3

from gi.repository import Gtk

from matplotlib.figure import Figure
from matplotlib.axes import Subplot
from numpy import arange, sin, pi
from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as FigureCanvas


class Window1Signals(object):
    def on_window1_destroy(self, widget):
        Gtk.main_quit()


def main():
    builder = Gtk.Builder()
    builder.add_objects_from_file("mpl_with_glade_316.glade", ("window1", ""))
    builder.connect_signals(Window1Signals())
    window = builder.get_object("window1")
    sw = builder.get_object("scrolledwindow1")

    # Start of Matplotlib specific code
    figure = Figure(figsize=(8, 6), dpi=71)
    axis = figure.add_subplot(111)
    t = arange(0.0, 3.0, 0.01)
```

```
    s = sin(2*pi*t)
    axis.plot(t, s)

    axis.set_xlabel('time [s]')
    axis.set_ylabel('voltage [V]')

    canvas = FigureCanvas(figure)  # a Gtk.DrawingArea
    canvas.set_size_request(800, 600)
    sw.add_with_viewport(canvas)
    # End of Matplotlib specific code

    window.show_all()
    Gtk.main()

if __name__ == "__main__":
    main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.25 user_interfaces example code: pylab_with_gtk.py

[source code]

```
"""
An example of how to use pylab to manage your figure windows, but
modify the GUI by accessing the underlying gtk widgets
"""
from __future__ import print_function
import matplotlib
matplotlib.use('GTKAgg')
import matplotlib.pyplot as plt


fig, ax = plt.subplots()
plt.plot([1, 2, 3], 'ro-', label='easy as 1 2 3')
plt.plot([1, 4, 9], 'gs--', label='easy as 1 2 3 squared')
plt.legend()


manager = plt.get_current_fig_manager()
# you can also access the window or vbox attributes this way
toolbar = manager.toolbar

# now let's add a button to the toolbar
import gtk
next = 8  # where to insert this in the mpl toolbar
button = gtk.Button('Click me')
button.show()


def clicked(button):
```

```
    print('hi mom')
button.connect('clicked', clicked)

toolitem = gtk.ToolItem()
toolitem.show()
toolitem.set_tooltip(
    toolbar.tooltips,
    'Click me for fun and profit')

toolitem.add(button)
toolbar.insert(toolitem, next)
next += 1

# now let's add a widget to the vbox
label = gtk.Label()
label.set_markup('Drag mouse over axes for position')
label.show()
vbox = manager.vbox
vbox.pack_start(label, False, False)
vbox.reorder_child(manager.toolbar, -1)


def update(event):
    if event.xdata is None:
        label.set_markup('Drag mouse over axes for position')
    else:
        label.set_markup('<span color="#ef0000">x,y=(%f, %f)</span>' % (event.xdata,␣
↪event.ydata))

plt.connect('motion_notify_event', update)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.26 user_interfaces example code: rec_edit_gtk_custom.py

[source code]

```
"""
generate an editable gtk treeview widget for record arrays with custom
formatting of the cells and show how to limit string entries to a list
of strings
"""
from __future__ import print_function
import gtk
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import mpl_toolkits.gtktools as gtktools
```

```
datafile = cbook.get_sample_data('demodata.csv', asfileobj=False)
r = mlab.csv2rec(datafile, converterd={'weekdays': str})


formatd = mlab.get_formatd(r)
formatd['date'] = mlab.FormatDate('%Y-%m-%d')
formatd['prices'] = mlab.FormatMillions(precision=1)
formatd['gain'] = mlab.FormatPercent(precision=2)

# use a drop down combo for weekdays
stringd = dict(weekdays=['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'])
constant = ['clientid']   # block editing of this field


liststore = gtktools.RecListStore(r, formatd=formatd, stringd=stringd)
treeview = gtktools.RecTreeView(liststore, constant=constant)


def mycallback(liststore, rownum, colname, oldval, newval):
    print('verify: old=%s, new=%s, rec=%s' % (oldval, newval, liststore.
↪r[rownum][colname]))

liststore.callbacks.connect('cell_changed', mycallback)

win = gtk.Window()
win.set_title('click to edit')
win.add(treeview)
win.show_all()
win.connect('delete-event', lambda *args: gtk.main_quit())
gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.27 user_interfaces example code: rec_edit_gtk_simple.py

[source code]

```
"""
Load a CSV file into a record array and edit it in a gtk treeview
"""

import gtk
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.cbook as cbook
import mpl_toolkits.gtktools as gtktools

datafile = cbook.get_sample_data('demodata.csv', asfileobj=False)
r = mlab.csv2rec(datafile, converterd={'weekdays': str})
```

```
liststore, treeview, win = gtktools.edit_recarray(r)
win.set_title('click to edit')
win.connect('delete-event', lambda *args: gtk.main_quit())
gtk.main()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.28 user_interfaces example code: svg_histogram.py

[source code]

```
#!/usr/bin/env python
#-*- encoding:utf-8 -*-

"""
Demonstrate how to create an interactive histogram, in which bars
are hidden or shown by cliking on legend markers.

The interactivity is encoded in ecmascript (javascript) and inserted in
the SVG code in a post-processing step. To render the image, open it in
a web browser. SVG is supported in most web browsers used by Linux and
OSX users. Windows IE9 supports SVG, but earlier versions do not.

Notes
-----
The matplotlib backend lets us assign ids to each object. This is the
mechanism used here to relate matplotlib objects created in python and
the corresponding SVG constructs that are parsed in the second step.
While flexible, ids are cumbersome to use for large collection of
objects. Two mechanisms could be used to simplify things:
 * systematic grouping of objects into SVG <g> tags,
 * assingning classes to each SVG object according to its origin.

For example, instead of modifying the properties of each individual bar,
the bars from the `hist` function could either be grouped in
a PatchCollection, or be assigned a class="hist_##" attribute.

CSS could also be used more extensively to replace repetitive markup
troughout the generated SVG.

__author__="david.huard@gmail.com"

"""


import numpy as np
import matplotlib.pyplot as plt
import xml.etree.ElementTree as ET
from io import BytesIO
import json
```

```python
plt.rcParams['svg.embed_char_paths'] = 'none'

# Apparently, this `register_namespace` method works only with
# python 2.7 and up and is necessary to avoid garbling the XML name
# space with ns0.
ET.register_namespace("", "http://www.w3.org/2000/svg")


# --- Create histogram, legend and title ---
plt.figure()
r = np.random.randn(100)
r1 = r + 1
labels = ['Rabbits', 'Frogs']
H = plt.hist([r, r1], label=labels)
containers = H[-1]
leg = plt.legend(frameon=False)
plt.title("From a web browser, click on the legend\n"
          "marker to toggle the corresponding histogram.")


# --- Add ids to the svg objects we'll modify

hist_patches = {}
for ic, c in enumerate(containers):
    hist_patches['hist_%d' % ic] = []
    for il, element in enumerate(c):
        element.set_gid('hist_%d_patch_%d' % (ic, il))
        hist_patches['hist_%d' % ic].append('hist_%d_patch_%d' % (ic, il))

# Set ids for the legend patches
for i, t in enumerate(leg.get_patches()):
    t.set_gid('leg_patch_%d' % i)

# Set ids for the text patches
for i, t in enumerate(leg.get_texts()):
    t.set_gid('leg_text_%d' % i)

# Save SVG in a fake file object.
f = BytesIO()
plt.savefig(f, format="svg")

# Create XML tree from the SVG file.
tree, xmlid = ET.XMLID(f.getvalue())


# --- Add interactivity ---

# Add attributes to the patch objects.
for i, t in enumerate(leg.get_patches()):
    el = xmlid['leg_patch_%d' % i]
    el.set('cursor', 'pointer')
    el.set('onclick', "toggle_hist(this)")
```

```
# Add attributes to the text objects.
for i, t in enumerate(leg.get_texts()):
    el = xmlid['leg_text_%d' % i]
    el.set('cursor', 'pointer')
    el.set('onclick', "toggle_hist(this)")

# Create script defining the function `toggle_hist`.
# We create a global variable `container` that stores the patches id
# belonging to each histogram. Then a function "toggle_element" sets the
# visibility attribute of all patches of each histogram and the opacity
# of the marker itself.

script = """
<script type="text/ecmascript">
<![CDATA[
var container = %s

function toggle(oid, attribute, values) {
    /* Toggle the style attribute of an object between two values.

    Parameters
    ----------
    oid : str
      Object identifier.
    attribute : str
      Name of syle attribute.
    values : [on state, off state]
      The two values that are switched between.
    */
    var obj = document.getElementById(oid);
    var a = obj.style[attribute];

    a = (a == values[0] || a == "") ? values[1] : values[0];
    obj.style[attribute] = a;
    }

function toggle_hist(obj) {

    var num = obj.id.slice(-1);

    toggle('leg_patch_' + num, 'opacity', [1, 0.3]);
    toggle('leg_text_' + num, 'opacity', [1, 0.5]);

    var names = container['hist_'+num]

    for (var i=0; i < names.length; i++) {
        toggle(names[i], 'opacity', [1,0])
    };
    }
]]>
</script>
""" % json.dumps(hist_patches)
```

```
# Add a transition effect
css = tree.getchildren()[0][0]
css.text = css.text + "g {-webkit-transition:opacity 0.4s ease-out;" + \
    "-moz-transition:opacity 0.4s ease-out;}"

# Insert the script and save to file.
tree.insert(0, ET.XML(script))

ET.ElementTree(tree).write("svg_histogram.svg")
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.29 user_interfaces example code: svg_tooltip.py

[source code]

```
"""
SVG tooltip example
===================

This example shows how to create a tooltip that will show up when
hovering over a matplotlib patch.

Although it is possible to create the tooltip from CSS or javascript,
here we create it in matplotlib and simply toggle its visibility on
when hovering over the patch. This approach provides total control over
the tooltip placement and appearance, at the expense of more code up
front.

The alternative approach would be to put the tooltip content in `title`
atttributes of SVG objects. Then, using an existing js/CSS library, it
would be relatively straightforward to create the tooltip in the
browser. The content would be dictated by the `title` attribute, and
the appearance by the CSS.


:author: David Huard
"""


import matplotlib.pyplot as plt
import xml.etree.ElementTree as ET
from io import BytesIO

ET.register_namespace("", "http://www.w3.org/2000/svg")

fig, ax = plt.subplots()

# Create patches to which tooltips will be assigned.
circle = plt.Circle((0, 0), 5, fc='blue')
rect = plt.Rectangle((-5, 10), 10, 5, fc='green')
```

```
ax.add_patch(circle)
ax.add_patch(rect)

# Create the tooltips
circle_tip = ax.annotate(
    'This is a blue circle.',
    xy=(0, 0),
    xytext=(30, -30),
    textcoords='offset points',
    color='w',
    ha='left',
    bbox=dict(boxstyle='round,pad=.5', fc=(.1, .1, .1, .92),
              ec=(1., 1., 1.), lw=1, zorder=1))

rect_tip = ax.annotate(
    'This is a green rectangle.',
    xy=(-5, 10),
    xytext=(30, 40),
    textcoords='offset points',
    color='w',
    ha='left',
    bbox=dict(boxstyle='round,pad=.5', fc=(.1, .1, .1, .92),
              ec=(1., 1., 1.), lw=1, zorder=1))

# Set id for the patches
for i, t in enumerate(ax.patches):
    t.set_gid('patch_%d' % i)

# Set id for the annotations
for i, t in enumerate(ax.texts):
    t.set_gid('tooltip_%d' % i)


# Save the figure in a fake file object
ax.set_xlim(-30, 30)
ax.set_ylim(-30, 30)
ax.set_aspect('equal')

f = BytesIO()
plt.savefig(f, format="svg")

# --- Add interactivity ---

# Create XML tree from the SVG file.
tree, xmlid = ET.XMLID(f.getvalue())
tree.set('onload', 'init(evt)')

# Hide the tooltips
for i, t in enumerate(ax.texts):
    el = xmlid['tooltip_%d' % i]
    el.set('visibility', 'hidden')
```

```
# Assign onmouseover and onmouseout callbacks to patches.
for i, t in enumerate(ax.patches):
    el = xmlid['patch_%d' % i]
    el.set('onmouseover', "ShowTooltip(this)")
    el.set('onmouseout', "HideTooltip(this)")

# This is the script defining the ShowTooltip and HideTooltip functions.
script = """
    <script type="text/ecmascript">
    <![CDATA[

    function init(evt) {
        if ( window.svgDocument == null ) {
            svgDocument = evt.target.ownerDocument;
            }
        }

    function ShowTooltip(obj) {
        var cur = obj.id.slice(-1);

        var tip = svgDocument.getElementById('tooltip_' + cur);
        tip.setAttribute('visibility',"visible")
        }

    function HideTooltip(obj) {
        var cur = obj.id.slice(-1);
        var tip = svgDocument.getElementById('tooltip_' + cur);
        tip.setAttribute('visibility',"hidden")
        }

    ]]>
    </script>
    """

# Insert the script at the top of the file and save it.
tree.insert(0, ET.XML(script))
ET.ElementTree(tree).write('svg_tooltip.svg')
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.30 user_interfaces example code: toolmanager.py

[source code]

```
'''This example demonstrates how to:
* Modify the Toolbar
* Create tools
* Add tools
* Remove tools
Using `matplotlib.backend_managers.ToolManager`
'''
```

```
from __future__ import print_function
import matplotlib
matplotlib.use('GTK3Cairo')
matplotlib.rcParams['toolbar'] = 'toolmanager'
import matplotlib.pyplot as plt
from matplotlib.backend_tools import ToolBase, ToolToggleBase
from gi.repository import Gtk, Gdk


class ListTools(ToolBase):
    '''List all the tools controlled by the `ToolManager`'''
    # keyboard shortcut
    default_keymap = 'm'
    description = 'List Tools'

    def trigger(self, *args, **kwargs):
        print('_' * 80)
        print("{0:12} {1:45} {2}".format('Name (id)',
                                         'Tool description',
                                         'Keymap'))
        print('-' * 80)
        tools = self.toolmanager.tools
        for name in sorted(tools.keys()):
            if not tools[name].description:
                continue
            keys = ', '.join(sorted(self.toolmanager.get_tool_keymap(name)))
            print("{0:12} {1:45} {2}".format(name,
                                             tools[name].description,
                                             keys))
        print('_' * 80)
        print("Active Toggle tools")
        print("{0:12} {1:45}".format("Group", "Active"))
        print('-' * 80)
        for group, active in self.toolmanager.active_toggle.items():
            print("{0:12} {1:45}".format(group, active))


class GroupHideTool(ToolToggleBase):
    '''Hide lines with a given gid'''
    default_keymap = 'G'
    description = 'Hide by gid'

    def __init__(self, *args, **kwargs):
        self.gid = kwargs.pop('gid')
        ToolToggleBase.__init__(self, *args, **kwargs)

    def enable(self, *args):
        self.set_lines_visibility(False)

    def disable(self, *args):
        self.set_lines_visibility(True)
```

```
    def set_lines_visibility(self, state):
        gr_lines = []
        for ax in self.figure.get_axes():
            for line in ax.get_lines():
                if line.get_gid() == self.gid:
                    line.set_visible(state)
        self.figure.canvas.draw()


fig = plt.figure()
plt.plot([1, 2, 3], gid='mygroup')
plt.plot([2, 3, 4], gid='unknown')
plt.plot([3, 2, 1], gid='mygroup')

# Add the custom tools that we created
fig.canvas.manager.toolmanager.add_tool('List', ListTools)
fig.canvas.manager.toolmanager.add_tool('Hide', GroupHideTool, gid='mygroup')


# Add an existing tool to new group `foo`.
# It can be added as many times as we want
fig.canvas.manager.toolbar.add_tool('zoom', 'foo')

# Remove the forward button
fig.canvas.manager.toolmanager.remove_tool('forward')

# To add a custom tool to the toolbar at specific location inside
# the navigation group
fig.canvas.manager.toolbar.add_tool('Hide', 'navigation', 1)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

102.31 user_interfaces example code: wxcursor_demo.py

[source code]

```
"""
Example to draw a cursor and report the data coords in wx
"""
# matplotlib requires wxPython 2.8+
# set the wxPython version in lib\site-packages\wx.pth file
# or if you have wxversion installed un-comment the lines below
#import wxversion
#wxversion.ensureMinimal('2.8')

import matplotlib
matplotlib.use('WXAgg')
```

```
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.backends.backend_wx import NavigationToolbar2Wx, wxc
from matplotlib.figure import Figure
from numpy import arange, sin, pi

import wx


class CanvasFrame(wx.Frame):
    def __init__(self, ):
        wx.Frame.__init__(self, None, -1,
                          'CanvasFrame', size=(550, 350))

        self.SetBackgroundColour(wxc.NamedColour("WHITE"))

        self.figure = Figure()
        self.axes = self.figure.add_subplot(111)
        t = arange(0.0, 3.0, 0.01)
        s = sin(2*pi*t)

        self.axes.plot(t, s)
        self.axes.set_xlabel('t')
        self.axes.set_ylabel('sin(t)')
        self.figure_canvas = FigureCanvas(self, -1, self.figure)

        # Note that event is a MplEvent
        self.figure_canvas.mpl_connect('motion_notify_event', self.UpdateStatusBar)
        self.figure_canvas.Bind(wx.EVT_ENTER_WINDOW, self.ChangeCursor)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.figure_canvas, 1, wx.LEFT | wx.TOP | wx.GROW)
        self.SetSizer(self.sizer)
        self.Fit()

        self.statusBar = wx.StatusBar(self, -1)
        self.SetStatusBar(self.statusBar)

        self.toolbar = NavigationToolbar2Wx(self.figure_canvas)
        self.sizer.Add(self.toolbar, 0, wx.LEFT | wx.EXPAND)
        self.toolbar.Show()

    def ChangeCursor(self, event):
        self.figure_canvas.SetCursor(wxc.StockCursor(wx.CURSOR_BULLSEYE))

    def UpdateStatusBar(self, event):
        if event.inaxes:
            x, y = event.xdata, event.ydata
            self.statusBar.SetStatusText(("x= " + str(x) +
                                          "  y=" + str(y)),
                                         0)


class App(wx.App):
```

```
    def OnInit(self):
        'Create the main window and insert the custom frame'
        frame = CanvasFrame()
        self.SetTopWindow(frame)
        frame.Show(True)
        return True

if __name__ == '__main__':
    app = App(0)
    app.MainLoop()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

CHAPTER

THREE

WIDGETS EXAMPLES

103.1 widgets example code: buttons.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Button

freqs = np.arange(2, 20, 3)

fig, ax = plt.subplots()
plt.subplots_adjust(bottom=0.2)
t = np.arange(0.0, 1.0, 0.001)
s = np.sin(2*np.pi*freqs[0]*t)
l, = plt.plot(t, s, lw=2)


class Index(object):
    ind = 0

    def next(self, event):
        self.ind += 1
        i = self.ind % len(freqs)
        ydata = np.sin(2*np.pi*freqs[i]*t)
        l.set_ydata(ydata)
        plt.draw()

    def prev(self, event):
        self.ind -= 1
        i = self.ind % len(freqs)
        ydata = np.sin(2*np.pi*freqs[i]*t)
        l.set_ydata(ydata)
        plt.draw()

callback = Index()
axprev = plt.axes([0.7, 0.05, 0.1, 0.075])
axnext = plt.axes([0.81, 0.05, 0.1, 0.075])
bnext = Button(axnext, 'Next')
bnext.on_clicked(callback.next)
```

```
bprev = Button(axprev, 'Previous')
bprev.on_clicked(callback.prev)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

103.2 widgets example code: check_buttons.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import CheckButtons

t = np.arange(0.0, 2.0, 0.01)
s0 = np.sin(2*np.pi*t)
s1 = np.sin(4*np.pi*t)
s2 = np.sin(6*np.pi*t)

fig, ax = plt.subplots()
l0, = ax.plot(t, s0, visible=False, lw=2)
l1, = ax.plot(t, s1, lw=2)
l2, = ax.plot(t, s2, lw=2)
plt.subplots_adjust(left=0.2)

rax = plt.axes([0.05, 0.4, 0.1, 0.15])
check = CheckButtons(rax, ('2 Hz', '4 Hz', '6 Hz'), (False, True, True))


def func(label):
    if label == '2 Hz':
        l0.set_visible(not l0.get_visible())
    elif label == '4 Hz':
        l1.set_visible(not l1.get_visible())
    elif label == '6 Hz':
        l2.set_visible(not l2.get_visible())
    plt.draw()
check.on_clicked(func)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

103.3 widgets example code: cursor.py

[source code]

```
#!/usr/bin/env python

from matplotlib.widgets import Cursor
import numpy as np
import matplotlib.pyplot as plt


fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, axisbg='#FFFFCC')

x, y = 4*(np.random.rand(2, 100) - .5)
ax.plot(x, y, 'o')
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)

# set useblit = True on gtkagg for enhanced performance
cursor = Cursor(ax, useblit=True, color='red', linewidth=2)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

103.4 widgets example code: lasso_selector_demo.py

[source code]

```
from __future__ import print_function

import numpy as np

from matplotlib.widgets import LassoSelector
from matplotlib.path import Path

try:
    raw_input
except NameError:
    # Python 3
    raw_input = input


class SelectFromCollection(object):
    """Select indices from a matplotlib collection using `LassoSelector`.

    Selected indices are saved in the `ind` attribute. This tool highlights
    selected points by fading them out (i.e., reducing their alpha values).
    If your collection has alpha < 1, this tool will permanently alter them.

    Note that this tool selects collection objects based on their *origins*
    (i.e., `offsets`).

    Parameters
```

```
    ----------
    ax : :class:`~matplotlib.axes.Axes`
        Axes to interact with.

    collection : :class:`matplotlib.collections.Collection` subclass
        Collection you want to select from.

    alpha_other : 0 <= float <= 1
        To highlight a selection, this tool sets all selected points to an
        alpha value of 1 and non-selected points to `alpha_other`.
    """

    def __init__(self, ax, collection, alpha_other=0.3):
        self.canvas = ax.figure.canvas
        self.collection = collection
        self.alpha_other = alpha_other

        self.xys = collection.get_offsets()
        self.Npts = len(self.xys)

        # Ensure that we have separate colors for each object
        self.fc = collection.get_facecolors()
        if len(self.fc) == 0:
            raise ValueError('Collection must have a facecolor')
        elif len(self.fc) == 1:
            self.fc = np.tile(self.fc, self.Npts).reshape(self.Npts, -1)

        self.lasso = LassoSelector(ax, onselect=self.onselect)
        self.ind = []

    def onselect(self, verts):
        path = Path(verts)
        self.ind = np.nonzero([path.contains_point(xy) for xy in self.xys])[0]
        self.fc[:, -1] = self.alpha_other
        self.fc[self.ind, -1] = 1
        self.collection.set_facecolors(self.fc)
        self.canvas.draw_idle()

    def disconnect(self):
        self.lasso.disconnect_events()
        self.fc[:, -1] = 1
        self.collection.set_facecolors(self.fc)
        self.canvas.draw_idle()


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    plt.ion()
    data = np.random.rand(100, 2)

    subplot_kw = dict(xlim=(0, 1), ylim=(0, 1), autoscale_on=False)
    fig, ax = plt.subplots(subplot_kw=subplot_kw)
```

```
    pts = ax.scatter(data[:, 0], data[:, 1], s=80)
    selector = SelectFromCollection(ax, pts)

    plt.draw()
    raw_input('Press any key to accept selected points')
    print("Selected points:")
    print(selector.xys[selector.ind])
    selector.disconnect()

    # Block end of script so you can check that the lasso is disconnected.
    raw_input('Press any key to quit')
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

103.5 widgets example code: menu.py

[source code]

```
from __future__ import division, print_function
import numpy as np
import matplotlib
import matplotlib.colors as colors
import matplotlib.patches as patches
import matplotlib.mathtext as mathtext
import matplotlib.pyplot as plt
import matplotlib.artist as artist
import matplotlib.image as image


class ItemProperties(object):
    def __init__(self, fontsize=14, labelcolor='black', bgcolor='yellow',
                 alpha=1.0):
        self.fontsize = fontsize
        self.labelcolor = labelcolor
        self.bgcolor = bgcolor
        self.alpha = alpha

        self.labelcolor_rgb = colors.colorConverter.to_rgb(labelcolor)
        self.bgcolor_rgb = colors.colorConverter.to_rgb(bgcolor)


class MenuItem(artist.Artist):
    parser = mathtext.MathTextParser("Bitmap")
    padx = 5
    pady = 5

    def __init__(self, fig, labelstr, props=None, hoverprops=None,
                 on_select=None):
        artist.Artist.__init__(self)
```

```
        self.set_figure(fig)
        self.labelstr = labelstr

        if props is None:
            props = ItemProperties()

        if hoverprops is None:
            hoverprops = ItemProperties()

        self.props = props
        self.hoverprops = hoverprops

        self.on_select = on_select

        x, self.depth = self.parser.to_mask(
            labelstr, fontsize=props.fontsize, dpi=fig.dpi)

        if props.fontsize != hoverprops.fontsize:
            raise NotImplementedError(
                'support for different font sizes not implemented')

        self.labelwidth = x.shape[1]
        self.labelheight = x.shape[0]

        self.labelArray = np.zeros((x.shape[0], x.shape[1], 4))
        self.labelArray[:, :, -1] = x/255.

        self.label = image.FigureImage(fig, origin='upper')
        self.label.set_array(self.labelArray)

        # we'll update these later
        self.rect = patches.Rectangle((0, 0), 1, 1)

        self.set_hover_props(False)

        fig.canvas.mpl_connect('button_release_event', self.check_select)

    def check_select(self, event):
        over, junk = self.rect.contains(event)
        if not over:
            return

        if self.on_select is not None:
            self.on_select(self)

    def set_extent(self, x, y, w, h):
        print(x, y, w, h)
        self.rect.set_x(x)
        self.rect.set_y(y)
        self.rect.set_width(w)
        self.rect.set_height(h)

        self.label.ox = x + self.padx
```

```
        self.label.oy = y - self.depth + self.pady/2.

        self.rect._update_patch_transform()
        self.hover = False

    def draw(self, renderer):
        self.rect.draw(renderer)
        self.label.draw(renderer)

    def set_hover_props(self, b):
        if b:
            props = self.hoverprops
        else:
            props = self.props

        r, g, b = props.labelcolor_rgb
        self.labelArray[:, :, 0] = r
        self.labelArray[:, :, 1] = g
        self.labelArray[:, :, 2] = b
        self.label.set_array(self.labelArray)
        self.rect.set(facecolor=props.bgcolor, alpha=props.alpha)

    def set_hover(self, event):
        'check the hover status of event and return true if status is changed'
        b, junk = self.rect.contains(event)

        changed = (b != self.hover)

        if changed:
            self.set_hover_props(b)

        self.hover = b
        return changed


class Menu(object):
    def __init__(self, fig, menuitems):
        self.figure = fig
        fig.suppressComposite = True

        self.menuitems = menuitems
        self.numitems = len(menuitems)

        maxw = max([item.labelwidth for item in menuitems])
        maxh = max([item.labelheight for item in menuitems])

        totalh = self.numitems*maxh + (self.numitems + 1)*2*MenuItem.pady

        x0 = 100
        y0 = 400

        width = maxw + 2*MenuItem.padx
        height = maxh + MenuItem.pady
```

```
        for item in menuitems:
            left = x0
            bottom = y0 - maxh - MenuItem.pady

            item.set_extent(left, bottom, width, height)

            fig.artists.append(item)
            y0 -= maxh + MenuItem.pady

        fig.canvas.mpl_connect('motion_notify_event', self.on_move)

    def on_move(self, event):
        draw = False
        for item in self.menuitems:
            draw = item.set_hover(event)
            if draw:
                self.figure.canvas.draw()
                break


fig = plt.figure()
fig.subplots_adjust(left=0.3)
props = ItemProperties(labelcolor='black', bgcolor='yellow',
                       fontsize=15, alpha=0.2)
hoverprops = ItemProperties(labelcolor='white', bgcolor='blue',
                            fontsize=15, alpha=0.2)

menuitems = []
for label in ('open', 'close', 'save', 'save as', 'quit'):
    def on_select(item):
        print('you selected %s' % item.labelstr)
    item = MenuItem(fig, label, props=props, hoverprops=hoverprops,
                    on_select=on_select)
    menuitems.append(item)

menu = Menu(fig, menuitems)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

103.6 widgets example code: multicursor.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import MultiCursor

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
```

```
s2 = np.sin(4*np.pi*t)
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax1.plot(t, s1)


ax2 = fig.add_subplot(212, sharex=ax1)
ax2.plot(t, s2)

multi = MultiCursor(fig.canvas, (ax1, ax2), color='r', lw=1)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

103.7 widgets example code: radio_buttons.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import RadioButtons

t = np.arange(0.0, 2.0, 0.01)
s0 = np.sin(2*np.pi*t)
s1 = np.sin(4*np.pi*t)
s2 = np.sin(8*np.pi*t)

fig, ax = plt.subplots()
l, = ax.plot(t, s0, lw=2, color='red')
plt.subplots_adjust(left=0.3)

axcolor = 'lightgoldenrodyellow'
rax = plt.axes([0.05, 0.7, 0.15, 0.15], axisbg=axcolor)
radio = RadioButtons(rax, ('2 Hz', '4 Hz', '8 Hz'))


def hzfunc(label):
    hzdict = {'2 Hz': s0, '4 Hz': s1, '8 Hz': s2}
    ydata = hzdict[label]
    l.set_ydata(ydata)
    plt.draw()
radio.on_clicked(hzfunc)

rax = plt.axes([0.05, 0.4, 0.15, 0.15], axisbg=axcolor)
radio2 = RadioButtons(rax, ('red', 'blue', 'green'))


def colorfunc(label):
    l.set_color(label)
    plt.draw()
radio2.on_clicked(colorfunc)
```

```
rax = plt.axes([0.05, 0.1, 0.15, 0.15], axisbg=axcolor)
radio3 = RadioButtons(rax, ('-', '--', '-.', 'steps', ':'))


def stylefunc(label):
    l.set_linestyle(label)
    plt.draw()
radio3.on_clicked(stylefunc)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

103.8 widgets example code: rectangle_selector.py

[source code]

```
from __future__ import print_function
"""
Do a mouseclick somewhere, move the mouse to some destination, release
the button.  This class gives click- and release-events and also draws
a line or a box from the click-point to the actual mouseposition
(within the same axes) until the button is released.  Within the
method 'self.ignore()' it is checked wether the button from eventpress
and eventrelease are the same.

"""
from matplotlib.widgets import RectangleSelector
import numpy as np
import matplotlib.pyplot as plt


def line_select_callback(eclick, erelease):
    'eclick and erelease are the press and release events'
    x1, y1 = eclick.xdata, eclick.ydata
    x2, y2 = erelease.xdata, erelease.ydata
    print("(%3.2f, %3.2f) --> (%3.2f, %3.2f)" % (x1, y1, x2, y2))
    print(" The button you used were: %s %s" % (eclick.button, erelease.button))


def toggle_selector(event):
    print(' Key pressed.')
    if event.key in ['Q', 'q'] and toggle_selector.RS.active:
        print(' RectangleSelector deactivated.')
        toggle_selector.RS.set_active(False)
    if event.key in ['A', 'a'] and not toggle_selector.RS.active:
        print(' RectangleSelector activated.')
        toggle_selector.RS.set_active(True)
```

```
fig, current_ax = plt.subplots()                    # make a new plotingrange
N = 100000                                          # If N is large one can see
x = np.linspace(0.0, 10.0, N)                       # improvement by use blitting!

plt.plot(x, +np.sin(.2*np.pi*x), lw=3.5, c='b', alpha=.7)  # plot something
plt.plot(x, +np.cos(.2*np.pi*x), lw=3.5, c='r', alpha=.5)
plt.plot(x, -np.sin(.2*np.pi*x), lw=3.5, c='g', alpha=.3)

print("\n      click  -->  release")

# drawtype is 'box' or 'line' or 'none'
toggle_selector.RS = RectangleSelector(current_ax, line_select_callback,
                                       drawtype='box', useblit=True,
                                       button=[1, 3],  # don't use middle button
                                       minspanx=5, minspany=5,
                                       spancoords='pixels',
                                       interactive=True)
plt.connect('key_press_event', toggle_selector)
plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

103.9 widgets example code: slider_demo.py

[source code]

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider, Button, RadioButtons

fig, ax = plt.subplots()
plt.subplots_adjust(left=0.25, bottom=0.25)
t = np.arange(0.0, 1.0, 0.001)
a0 = 5
f0 = 3
s = a0*np.sin(2*np.pi*f0*t)
l, = plt.plot(t, s, lw=2, color='red')
plt.axis([0, 1, -10, 10])

axcolor = 'lightgoldenrodyellow'
axfreq = plt.axes([0.25, 0.1, 0.65, 0.03], axisbg=axcolor)
axamp = plt.axes([0.25, 0.15, 0.65, 0.03], axisbg=axcolor)

sfreq = Slider(axfreq, 'Freq', 0.1, 30.0, valinit=f0)
samp = Slider(axamp, 'Amp', 0.1, 10.0, valinit=a0)


def update(val):
    amp = samp.val
    freq = sfreq.val
    l.set_ydata(amp*np.sin(2*np.pi*freq*t))
```

```
    fig.canvas.draw_idle()
sfreq.on_changed(update)
samp.on_changed(update)

resetax = plt.axes([0.8, 0.025, 0.1, 0.04])
button = Button(resetax, 'Reset', color=axcolor, hovercolor='0.975')


def reset(event):
    sfreq.reset()
    samp.reset()
button.on_clicked(reset)

rax = plt.axes([0.025, 0.5, 0.15, 0.15], axisbg=axcolor)
radio = RadioButtons(rax, ('red', 'blue', 'green'), active=0)


def colorfunc(label):
    l.set_color(label)
    fig.canvas.draw_idle()
radio.on_clicked(colorfunc)

plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

103.10 widgets example code: span_selector.py

[source code]

```
#!/usr/bin/env python
"""
The SpanSelector is a mouse widget to select a xmin/xmax range and plot the
detail view of the selected region in the lower axes
"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import SpanSelector

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(211, axisbg='#FFFFCC')

x = np.arange(0.0, 5.0, 0.01)
y = np.sin(2*np.pi*x) + 0.5*np.random.randn(len(x))

ax.plot(x, y, '-')
ax.set_ylim(-2, 2)
ax.set_title('Press left mouse button and drag to test')

ax2 = fig.add_subplot(212, axisbg='#FFFFCC')
line2, = ax2.plot(x, y, '-')
```

```python
def onselect(xmin, xmax):
    indmin, indmax = np.searchsorted(x, (xmin, xmax))
    indmax = min(len(x) - 1, indmax)

    thisx = x[indmin:indmax]
    thisy = y[indmin:indmax]
    line2.set_data(thisx, thisy)
    ax2.set_xlim(thisx[0], thisx[-1])
    ax2.set_ylim(thisy.min(), thisy.max())
    fig.canvas.draw()

# set useblit True on gtkagg for enhanced performance
span = SpanSelector(ax, onselect, 'horizontal', useblit=True,
                    rectprops=dict(alpha=0.5, facecolor='red'))


plt.show()
```

Keywords: python, matplotlib, pylab, example, codex (see *Search examples*)

Part XI

Glossary

AGG The Anti-Grain Geometry (Agg) rendering engine, capable of rendering high-quality images

Cairo The Cairo graphics engine

dateutil The dateutil library provides extensions to the standard datetime module

EPS Encapsulated Postscript (EPS)

freetype freetype is a font rasterization library used by matplotlib which supports TrueType, Type 1, and OpenType fonts.

GDK The Gimp Drawing Kit for GTK+

GTK The GIMP Toolkit (GTK) graphical user interface library

JPG The Joint Photographic Experts Group (JPEG) compression method and file format for photographic images

numpy numpy is the standard numerical array library for python, the successor to Numeric and numarray. numpy provides fast operations for homogeneous data sets and common mathematical operations like correlations, standard deviation, fourier transforms, and convolutions.

PDF Adobe's Portable Document Format (PDF)

PNG Portable Network Graphics (PNG), a raster graphics format that employs lossless data compression which is more suitable for line art than the lossy jpg format. Unlike the gif format, png is not encumbered by requirements for a patent license.

PS Postscript (PS) is a vector graphics ASCII text language widely used in printers and publishing. Postscript was developed by adobe systems and is starting to show its age: for example is does not have an alpha channel. PDF was designed in part as a next-generation document format to replace postscript

pygtk pygtk provides python wrappers for the *GTK* widgets library for use with the GTK or GTKAgg backend. Widely used on linux, and is often packages as 'python-gtk2'

pyqt pyqt provides python wrappers for the *Qt* widgets library and is required by the matplotlib Qt5Agg and Qt4Agg backends. Widely used on linux and windows; many linux distributions package this as 'python-qt5' or 'python-qt4'.

python python is an object oriented interpreted language widely used for scripting, application development, web application servers, scientific computing and more.

pytz pytz provides the Olson tz database in Python. it allows accurate and cross platform timezone calculations and solves the issue of ambiguous times at the end of daylight savings

Qt Qt is a cross-platform application framework for desktop and embedded development.

Qt4 Qt4 is the previous, but most widely used, version of Qt cross-platform application framework for desktop and embedded development.

Qt5 Qt5 is the current version of Qt cross-platform application framework for desktop and embedded development.

raster graphics Raster graphics, or bitmaps, represent an image as an array of pixels which is resolution dependent. Raster graphics are generally most practical for photo-realistic images, but do not scale easily without loss of quality.

SVG The Scalable Vector Graphics format (SVG). An XML based vector graphics format supported by many web browsers.

TIFF Tagged Image File Format (TIFF) is a file format for storing images, including photographs and line art.

Tk Tk is a graphical user interface for Tcl and many other dynamic languages. It can produce rich, native applications that run unchanged across Windows, Mac OS X, Linux and more.

vector graphics vector graphics use geometrical primitives based upon mathematical equations to represent images in computer graphics. Primitives can include points, lines, curves, and shapes or polygons. Vector graphics are scalable, which means that they can be resized without suffering from issues related to inherent resolution like are seen in raster graphics. Vector graphics are generally most practical for typesetting and graphic design applications.

wxpython wxpython provides python wrappers for the *wxWidgets* library for use with the WX and WX-Agg backends. Widely used on linux, OS-X and windows, it is often packaged by linux distributions as 'python-wxgtk'

wxWidgets WX is cross-platform GUI and tools library for GTK, MS Windows, and MacOS. It uses native widgets for each operating system, so applications will have the look-and-feel that users on that operating system expect.

[Ware] http://ccom.unh.edu/sites/default/files/publications/Ware_1988_CGA_Color_sequences_univariate_maps.pdf

[Moreland] http://www.sandia.gov/~kmorel/documents/ColorMaps/ColorMapsExpanded.pdf

[list-colormaps] https://gist.github.com/endolith/2719900#id7

[mycarta-banding] http://mycarta.wordpress.com/2012/10/14/the-rainbow-is-deadlong-live-the-rainbow-part-4-cie-lab-he

[mycarta-jet] http://mycarta.wordpress.com/2012/10/06/the-rainbow-is-deadlong-live-the-rainbow-part-3/

[mycarta-lablinear] http://mycarta.wordpress.com/2012/12/06/the-rainbow-is-deadlong-live-the-rainbow-part-5-cie-lab-li

[mycarta-cubelaw] http://mycarta.wordpress.com/2013/02/21/perceptual-rainbow-palette-the-method/

[bw] http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/

[colorblindness] http://aspnetresources.com/tools/colorBlindness

[asp] http://aspnetresources.com/tools/colorBlindness

[IBM] http://www.research.ibm.com/people/l/lloydt/color/color.HTM

[R1] Michel Bernadou, Kamal Hassan, "Basis functions for general Hsieh-Clough-Tocher triangles, complete or reduced.", International Journal for Numerical Methods in Engineering, 17(5):784 - 789. 2.01.

[R2] C.T. Kelley, "Iterative Methods for Optimization".

m

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z