Sage Reference Manual: The Sage Command Line

Release 8.0

The Sage Development Team

Jul 23, 2017

CONTENTS

The Sage Read-Eval-Print-Loop (REPL) is based on IPython. In this document, you'll find how the IPython integration works. You should also be familiar with the documentation for IPython.

For more details about using the Sage command line, see the Sage tutorial.

RUNNING SAGE

1.1 Invoking Sage

To run Sage, you basically just need to type `sage` from the command-line prompt to start the Sage interpreter. See the Sage Installation Guide for information about making sure your `$PATH` is set correctly, etc.

1.1.1 Command-line options for Sage

Running Sage, the most common options

- `file.[sage|py|spyx]` – run the given .sage, .py or .spyx files (as in `sage my_file.sage`)
- `-h`, `-?`, `--help` – print a short help message
- `-v`, `--version` – print the Sage version
- `--advanced` – print (essentially this) list of Sage options
- `-c cmd` – evaluate `cmd` as sage code. For example, `sage -c 'print(factor(35))'` will print "5 * 7".

Running Sage, other options

- `--preparse file.sage` – preparse `file.sage`, a file of Sage code, and produce the corresponding Python file `file.sage.py`. See the Sage tutorial for more about preparsing and the differences between Sage and Python.
- `-q` – quiet; start with no banner
- `--grep [options] <string>` – grep through all the Sage library code for `string`. Any options will get passed to the "grep" command; for example, `sage --grep -i epstein` will search for `epstein`, and the `-i` flag tells grep to ignore case when searching. Note that while running Sage, you can also use the function `search_src` to accomplish the same thing.
- `--grepdoc [options] <string>` – grep through all the Sage documentation for `string`. Note that while running Sage, you can also use the function `search_doc` to accomplish the same thing.
- `--min [...]` – do not populate global namespace (must be first option)
- `-gthread`, `-qthread`, `-q4thread`, `-wthread`, `-pylab` – pass the option through to IPython
- `--nodotsage` – run Sage without using the user's `.sage` directory: create and use a temporary `.sage` directory instead. Warning: notebooks are stored in the `.sage` directory, so any notebooks created while running with `--nodotsage` will be temporary also.

Running the notebook

- `-n [...]`, `--notebook=[...]` – start the notebook, valid options are `default`, `sagenb`, `jupyter` and `export` (see the output of `sage --notebook --help` for more details and examples of how to pass optional arguments)

- `-bn [...]`, `--build-and-notebook [...]` – build the Sage library (as by running `sage -b`) then start the Sage notebook

- `--inotebook [...]` – start the *insecure* Sage notebook

Running external programs and utilities

- `--cython [...]` – run Cython with the given arguments

- `--ecl [...]`, `--lisp [...]` – run Sage's copy of ECL (Embeddable Common Lisp) with the given arguments

- `--gap [...]` – run Sage's Gap with the given arguments

- `--git [...]` – run Sage's Git with the given arguments

- `--gp [...]` – run Sage's PARI/GP calculator with the given arguments

- `--ipython [...]` – run Sage's IPython using the default environment (not Sage), passing additional options to IPython

- `--kash [...]` – run Sage's Kash with the given arguments

- `--M2 [...]` – run Sage's Macaulay2 with the given arguments

- `--maxima [...]` – run Sage's Maxima with the given arguments

- `--mwrank [...]` – run Sage's mwrank with the given arguments

- `--python [...]`, `--python2 [...]` – run the Python 2 interpreter

- `--python3 [...]` – run the Python 3 interpreter

- `-R [...]` – run Sage's R with the given arguments

- `--scons [...]` – run Sage's scons

- `--singular [...]` – run Sage's singular with the given arguments

- `--twistd [...]` – run Twisted server

- `--sh [...]` – run a shell with Sage environment variables set

- `--gdb` – run Sage under the control of gdb

- `--gdb-ipython` – run Sage's IPython under the control of gdb

- `--cleaner` – run the Sage cleaner. This cleans up after Sage, removing temporary directories and spawned processes. (This gets run by Sage automatically, so it is usually not necessary to run it separately.)

Installing packages and upgrading

- `-i [options] [packages]` – install the given Sage packages (unless they are already installed); if no packages are given, print a list of all installed packages. Options:

    - `-c` – run the packages' test suites, overriding the settings of `SAGE_CHECK` and `SAGE_CHECK_PACKAGES`.

- – `-f` – force build: install the packages even if they are already installed.

- – `-s` – do not delete the `spkg/build` directories after a successful build – useful for debugging.

- `-f [options] [packages]` – shortcut for `-i -f`: force build of the given Sage packages.

- `--info [packages]` – display the `SPKG.txt` file of the given Sage packages.

- `--standard` – list all standard packages that can be installed

- `--optional` – list all optional packages that can be installed

- `--experimental` – list all experimental packages that can be installed

- `--upgrade [url]` – download, build and install standard packages from given url. If url not given, automatically selects a suitable mirror. If url='ask', it lets you select the mirror.

Building and testing the Sage library

- `--root` – print the Sage root directory

- `-b` – build Sage library – do this if you have modified any source code files in `$SAGE_ROOT/src/sage/`.

- `-ba` – same as `-b`, but rebuild *all* Cython code. This could take a while, so you will be asked if you want to proceed.

- `-ba-force` – same as `-ba`, but don't query before rebuilding

- `--br` – build and run Sage

- `-t [options] <files|dir>` – test examples in .py, .pyx, .sage or .tex files. Options:

  - – `--long` – include lines with the phrase 'long time'

  - – `--verbose` – print debugging output during the test

  - – `--optional` – also test all examples labeled `# optional`

  - – `--only-optional[=tags]` – if no `tags` are specified, only run blocks of tests containing a line labeled `# optional`. If a comma separated list of tags is specified, only run blocks containing a line labeled `# optional tag` for any of the tags given and in these blocks only run the lines which are unlabeled or labeled `#optional` or labeled `#optional tag` for any of the tags given.

  - – `--randorder[=seed]` – randomize order of tests

- `-tnew [...]` – like `-t` above, but only tests files modified since last commit

- `-tp <N> [...]` – like `-t` above, but tests in parallel using `N` threads with 0 interpreted as `minimum(8, cpu_count())`

- `--testall [options]` – test all source files, docs, and examples; options are the same as for `-t`.

- `-bt [...]` – build and test, options like `-t` above

- `-btp <N> [...]` – build and test in parallel, options like `-tp` above

- `-btnew [...]` – build and test modified files, options like `-tnew`

- `--fixdoctests file.py [output_file] [--long]` – writes a new version of `file.py` to `output_file` (default: `file.py.out`) that will pass the doctests. With the optional `--long` argument the long time tests are also checked. A patch for the new file is printed to stdout.

- `--startuptime [module]` – display how long each component of Sage takes to start up. Optionally specify a module (e.g., "sage.rings.qqbar") to get more details about that particular module.

- `--coverage <files>` – give information about doctest coverage of files

- `--coverageall` – give summary info about doctest coverage of all files in the Sage library

Documentation

- `--docbuild [options] document (format | command)` – build or return information about the Sage documentation.

  - `document` – name of the document to build
  - `format` – document output format
  - `command` – document-specific command

  A `document` and either a `format` or a `command` are required, unless a list of one or more of these is requested.

  Options:

  - `help`, `-h`, `--help` – print a help message
  - `-H`, `--help-all` – print an extended help message, including the output from the options `-h`, `-D`, `-F`, `-C all`, and a short list of examples.
  - `-D`, `--documents` – list all available documents
  - `-F`, `--formats` – list all output formats
  - `-C DOC`, `--commands=DOC` – list all commands for document `DOC`; use `-C all` to list all
  - `-i`, `--inherited` – include inherited members in reference manual; may be slow, may fail for PDF output
  - `-u`, `--underscore` – include variables prefixed with _ in reference manual; may be slow, may fail for PDF output
  - `-j`, `--jsmath` – render math using jsMath; formats: `html`, `json`, `pickle`, `web`
  - `--no-pdf-links` – do not include PDF links in document `website`; formats: `html`, `json`, `pickle`, `web`
  - `--check-nested` – check picklability of nested classes in document `reference`
  - `-N`, `--no-colors` – do not color output; does not affect children
  - `-q`, `--quiet` – work quietly; same as `--verbose=0`
  - `-v LEVEL`, `--verbose=LEVEL` – report progress at level 0 (quiet), 1 (normal), 2 (info), or 3 (debug); does not affect children

  Advanced – use these options with care:

  - `-S OPTS`, `--sphinx-opts=OPTS` – pass comma-separated `OPTS` to sphinx-build
  - `-U`, `--update-mtimes` – before building reference manual, update modification times for auto-generated ReST files

Making Sage packages or distributions

- `--pkg dir` – create the Sage package `dir.spkg` from the directory `dir`
- `--pkg_nc dir` – as `--pkg`, but do not compress the package
- `--merge` – run Sage's automatic merge and test script

- `--sdist` – build a source distribution of Sage

Valgrind memory debugging

- `--cachegrind` – run Sage using Valgrind's cachegrind tool

- `--callgrind` – run Sage using Valgrind's callgrind tool

- `--massif` – run Sage using Valgrind's massif tool

- `--memcheck` – run Sage using Valgrind's memcheck tool

- `--omega` – run Sage using Valgrind's omega tool

- `--valgrind` – this is an alias for `--memcheck`

1.2 Sage startup scripts

There are two kinds of startup scripts that Sage reads when starting:

1.2.1 The sagerc shell script

The *bash shell script* `$DOT_SAGE/sagerc` (with the default value of `DOT_SAGE`, this is `~/.sage/sagerc`) is read by `$SAGE_ROOT/spkg/bin/sage-env` after Sage has set its environment variables. It can be used to override some of the environment variables determined by Sage, or it can contain other shell commands like creating directories. This script is sourced not only when running Sage itself, but also when running any of the subcommands (like `sage --python`, `sage -b` or `sage -i <package>`). In particular, setting `PS1` here overrides the default prompt for the Sage shell `sage --sh`.

Note: This script is run with the Sage directories in its `PATH`, so executing `git` for example will run the Git inside Sage.

The default location of this file can be changed using the environment variable `SAGE_RC_FILE`.

1.2.2 The init.sage script

The *Sage script* `$DOT_SAGE/init.sage` (with the default value of `DOT_SAGE`, this is `~/.sage/init.sage`) contains Sage commands to be executed every time Sage starts. If you want symbolic variables `y` and `z` in every Sage session, you could put

```
var('y, z')
```

in this file.

The default location of this file can be changed using the environment variable `SAGE_STARTUP_FILE`.

1.3 Environment variables used by Sage

Sage uses several environment variables when running. These all have sensible default values, so many users won't need to set any of these. (There are also variables used to compile Sage; see the Sage Installation Guide for more about those.)

- `DOT_SAGE` – this is the directory, to which the user has read and write access, where Sage stores a number of files. The default location is `~/.sage/`, but you can change that by setting this variable.

- `SAGE_RC_FILE` – a shell script which is sourced after Sage has determined its environment variables. This script is executed before starting Sage or any of its subcommands (like `sage -i <package>`). The default value is `$DOT_SAGE/sagerc`.

- `SAGE_STARTUP_FILE` – a file including commands to be executed every time Sage starts. The default value is `$DOT_SAGE/init.sage`.

- `SAGE_SERVER` – only used for installing packages. Alternative mirror from which to download sources, see the Installation Guide for details.

- `SAGE_PATH` – a colon-separated list of directories which Sage searches when trying to locate Python libraries.

- `BROWSER` – on most platforms, Sage will detect the command to run a web browser, but if this doesn't seem to work on your machine, set this variable to the appropriate command.

1.4 Relevant environment variables for other packages

This is a non-exhaustive list of environment variables which influence some package contained within the SageMath distribution.

In many cases, SageMath uses a custom default value if the variable is not set, which is not the same default that the system-wide package would use. So, if you would like to use your system-wide configuration, you need to explicitly set the environment variable to the system-wide default.

- `IPYTHONDIR` – directory where the configuration of IPython is stored. By default, this is some directory inside `DOT_SAGE`. See http://ipython.readthedocs.io/en/stable/development/config.html for more information.

- `JUPYTER_CONFIG_DIR` – directory where the configuration of Jupyter is stored. By default, this is some directory inside `DOT_SAGE`. See http://jupyter.readthedocs.io/en/latest/projects/jupyter-directories.html for more information.

1.5 Interactively tracing execution of a command

sage.misc.trace.**trace**(*code*, *preparse=True*)

Evaluate Sage code using the interactive tracer and return the result. The string `code` must be a valid expression enclosed in quotes (no assignments - the result of the expression is returned). In the Sage notebook this just raises a NotImplementedException.

INPUT:

- `code` - str

- `preparse` - bool (default: True); if True, run expression through the Sage preparser.

REMARKS: This function is extremely powerful! For example, if you want to step through each line of execution of, e.g., `factor(100)`, type

```
sage: trace("factor(100)")            # not tested
```

then at the (Pdb) prompt type `s` (or `step`), then press return over and over to step through every line of Python that is called in the course of the above computation. Type `?` at any time for help on how to use the debugger (e.g., `l` lists 11 lines around the current line; `bt` gives a back trace, etc.).

Setting a break point: If you have some code in a file and would like to drop into the debugger at a given point, put the following code at that point in the file:

```
import pdb; pdb.set_trace()
```

For an article on how to use the Python debugger, see http://www.onlamp.com/pub/a/python/2005/09/01/debugger.html

PREPARSING

Sage commands are "preparsed" to valid Python syntax. This allows for example to support the `R.<x> = QQ[]` syntax.

2.1 The Sage Preparser

AUTHORS:

- William Stein (2006-02-19)

    - Fixed bug when loading .py files.

- William Stein (2006-03-09)

    - Fixed crash in parsing exponentials.

    - Precision of real literals now determined by digits of input (like Mathematica).

- Joe Wetherell (2006-04-14)

    - Added MAGMA-style constructor preparsing.

- Bobby Moretti (2007-01-25)

    - Added preliminary function assignment notation.

- Robert Bradshaw (2007-09-19)

    - Added strip_string_literals, containing_block utility functions. Arrr!

    - Added [1,2,..,n] notation.

- Robert Bradshaw (2008-01-04)

    - Implicit multiplication (off by default).

- Robert Bradshaw (2008-09-23)

    - Factor out constants.

- Robert Bradshaw (2000-01)

    - Simplify preparser by making it modular and using regular expressions.

    - Bug fixes, complex numbers, and binary input.

EXAMPLES:

Preparsing:

```
sage: preparse('2/3')
'Integer(2)/Integer(3)'
sage: preparse('2.5')
"RealNumber('2.5')"
sage: preparse('2^3')
'Integer(2)**Integer(3)'
sage: preparse('a^b')              # exponent
'a**b'
sage: preparse('a**b')
'a**b'
sage: preparse('G.0')              # generator
'G.gen(0)'
sage: preparse('a = 939393R')      # raw
'a = 939393'
sage: implicit_multiplication(True)
sage: preparse('a b c in L')       # implicit multiplication
'a*b*c in L'
sage: preparse('2e3x + 3exp(y)')
"RealNumber('2e3')*x + Integer(3)*exp(y)"
```

A string with escaped quotes in it (the point here is that the preparser doesn't get confused by the internal quotes):

```
sage: ""Yes," he said."
'"Yes," he said.'
sage: s = "\"; s
'\'
```

A hex literal:

```
sage: preparse('0x2e3')
'Integer(0x2e3)'
sage: 0xA
10
sage: 0xe
14
```

Raw and hex work correctly:

```
sage: type(0xa1)
<type 'sage.rings.integer.Integer'>
sage: type(0xa1r)
<... 'int'>
sage: type(0Xa1R)
<... 'int'>
```

In Sage, methods can also be called on integer and real literals (note that in pure Python this would be a syntax error):

```
sage: 16.sqrt()
4
sage: 87.factor()
3 * 29
sage: 15.10.sqrt()
3.88587184554509
sage: preparse('87.sqrt()')
'Integer(87).sqrt()'
sage: preparse('15.10.sqrt()')
"RealNumber('15.10').sqrt()"
```

Note that calling methods on int literals in pure Python is a syntax error, but Sage allows this for Sage integers and reals, because users frequently request it:

```
sage: eval('4.__add__(3)')
Traceback (most recent call last):
...
SyntaxError: invalid syntax
```

Symbolic functional notation:

```
sage: a=10; f(theta, beta) = theta + beta; b = x^2 + theta
sage: f
(theta, beta) |--> beta + theta
sage: a
10
sage: b
x^2 + theta
sage: f(theta,theta)
2*theta

sage: a = 5; f(x,y) = x*y*sqrt(a)
sage: f
(x, y) |--> sqrt(5)*x*y
```

This involves an =-, but should still be turned into a symbolic expression:

```
sage: preparse('a(x) =- 5')
'__tmp__=var("x"); a = symbolic_expression(- Integer(5)).function(x)'
sage: f(x)=-x
sage: f(10)
-10
```

This involves -=, which should not be turned into a symbolic expression (of course a(x) isn't an identifier, so this will never be valid):

```
sage: preparse('a(x) -= 5')
'a(x) -= Integer(5)'
```

Raw literals:

Raw literals are not preparsed, which can be useful from an efficiency point of view. Just like Python ints are denoted by an L, in Sage raw integer and floating literals are followed by an"r" (or "R") for raw, meaning not preparsed.

We create a raw integer:

```
sage: a = 393939r
sage: a
393939
sage: type(a)
<... 'int'>
```

We create a raw float:

```
sage: z = 1.5949r
sage: z
1.5949
sage: type(z)
<... 'float'>
```

You can also use an upper case letter:

```
sage: z = 3.1415R
sage: z
3.1415
sage: type(z)
<... 'float'>
```

This next example illustrates how raw literals can be very useful in certain cases. We make a list of even integers up to 10000:

```
sage: v = [ 2*i for i in range(10000)]
```

This takes a noticeable fraction of a second (e.g., 0.25 seconds). After preparsing, what Python is really executing is the following:

```
sage: preparse('v = [ 2*i for i in range(10000)]')
'v = [ Integer(2)*i for i in range(Integer(10000))]'
```

If instead we use a raw 2 we get execution that is *instant* (0.00 seconds):

```
sage: v = [ 2r * i for i in range(10000r)]
```

Behind the scenes what happens is the following:

```
sage: preparse('v = [ 2r * i for i in range(10000r)]')
'v = [ 2 * i for i in range(10000)]'
```

> **Warning:** The results of the above two expressions are different. The first one computes a list of Sage integers, whereas the second creates a list of Python integers. Python integers are typically much more efficient than Sage integers when they are very small; large Sage integers are much more efficient than Python integers, since they are implemented using the GMP C library.

sage.repl.preparse.**containing_block**(*code, ix, delimiters=['()', '[]', '{}'], require_delim=True*)

Returns the smallest range (start,end) such that code[start,end] is delimited by balanced delimiters (e.g., parentheses, brackets, and braces).

INPUT:

- `code` - a string

- `ix` - an integer; a starting position

- `delimiters` - a list of strings (default: ['()', '[]', '{}']); the delimiters to balance

- `require_delim` - a boolean (default: True); whether to raise a SyntaxError if delimiters are unbalanced

OUTPUT:

- a 2-tuple of integers

EXAMPLES:

```
sage: from sage.repl.preparse import containing_block
sage: s = "factor(next_prime(L[5]+1))"
sage: s[22]
'+'
sage: start, end = containing_block(s, 22)
```

```
sage: start, end
(17, 25)
sage: s[start:end]
'(L[5]+1)'
sage: s[20]
'5'
sage: start, end = containing_block(s, 20); s[start:end]
'[5]'
sage: start, end = containing_block(s, 20, delimiters=['()']); s[start:end]
'(L[5]+1)'
sage: start, end = containing_block(s, 10); s[start:end]
'(next_prime(L[5]+1))'
```

sage.repl.preparse.**extract_numeric_literals**(*code*)

Pulls out numeric literals and assigns them to global variables. This eliminates the need to re-parse and create the literals, e.g., during every iteration of a loop.

INPUT:

- •code - a string; a block of code

OUTPUT:

- •a (string, string:string dictionary) 2-tuple; the block with literals replaced by variable names and a mapping from names to the new variables

EXAMPLES:

```
sage: from sage.repl.preparse import extract_numeric_literals
sage: code, nums = extract_numeric_literals("1.2 + 5")
sage: print(code)
_sage_const_1p2  + _sage_const_5
sage: print(nums)
{'_sage_const_1p2': "RealNumber('1.2')", '_sage_const_5': 'Integer(5)'}

sage: extract_numeric_literals("[1, 1.1, 1e1, -1e-1, 1.]")[0]
'[_sage_const_1 , _sage_const_1p1 , _sage_const_1e1 , -_sage_const_1en1 , _sage_
→const_1p ]'

sage: extract_numeric_literals("[1.sqrt(), 1.2.sqrt(), 1r, 1.2r, R.1, R0.1, (1..
→5)]")[0]
'[_sage_const_1 .sqrt(), _sage_const_1p2 .sqrt(), 1 , 1.2 , R.1, R0.1, (_sage_
→const_1 .._sage_const_5 )]'
```

sage.repl.preparse.**handle_encoding_declaration**(*contents*, *out*)

Find a PEP 263-style Python encoding declaration in the first or second line of `contents`. If found, output it to `out` and return `contents` without the encoding line; otherwise output a default UTF-8 declaration and return `contents`.

EXAMPLES:

```
sage: from sage.repl.preparse import handle_encoding_declaration
sage: import sys
sage: c1='# -*- coding: latin-1 -*-\nimport os, sys\n...'
sage: c2='# -*- coding: iso-8859-15 -*-\nimport os, sys\n...'
sage: c3='# -*- coding: ascii -*-\nimport os, sys\n...'
sage: c4='import os, sys\n...'
sage: handle_encoding_declaration(c1, sys.stdout)
# -*- coding: latin-1 -*-
'import os, sys\n...'
```

```
sage: handle_encoding_declaration(c2, sys.stdout)
# -*- coding: iso-8859-15 -*-
'import os, sys\n...'
sage: handle_encoding_declaration(c3, sys.stdout)
# -*- coding: ascii -*-
'import os, sys\n...'
sage: handle_encoding_declaration(c4, sys.stdout)
# -*- coding: utf-8 -*-
'import os, sys\n...'
```

NOTES:

- PEP 263: http://www.python.org/dev/peps/pep-0263/

- PEP 263 says that Python will interpret a UTF-8 byte order mark as a declaration of UTF-8 encoding, but I don't think we do that; this function only sees a Python string so it can't account for a BOM.

- We default to UTF-8 encoding even though PEP 263 says that Python files should default to ASCII.

- Also see http://docs.python.org/ref/encodings.html.

AUTHORS:

- Lars Fischer

- Dan Drake (2010-12-08, rewrite for trac ticket #10440)

sage.repl.preparse.**implicit_mul**(*code*, *level=5*)

Inserts *'s to make implicit multiplication explicit.

INPUT:

- `code` – a string; the code with missing *'s

- `level` – an integer (default: 5); how aggressive to be in placing *'s

  - 0 - Do nothing

  - 1 - Numeric followed by alphanumeric

  - 2 - Closing parentheses followed by alphanumeric

  - 3 - Spaces between alphanumeric

  - 10 - Adjacent parentheses (may mangle call statements)

OUTPUT:

- a string

EXAMPLES:

```
sage: from sage.repl.preparse import implicit_mul
sage: implicit_mul('(2x^2-4x+3)a0')
'(2*x^2-4*x+3)*a0'
sage: implicit_mul('a b c in L')
'a*b*c in L'
sage: implicit_mul('1r + 1e3 + 5exp(2)')
'1r + 1e3 + 5*exp(2)'
sage: implicit_mul('f(a)(b)', level=10)
'f(a)*(b)'
```

sage.repl.preparse.**implicit_multiplication**(*level=None*)

Turns implicit multiplication on or off, optionally setting a specific `level`. Returns the current `level` if no argument is given.

INPUT:

> - `level` - an integer (default: None); see *implicit_mul()* for a list

EXAMPLES:

```
sage: implicit_multiplication(True)
sage: implicit_multiplication()
5
sage: preparse('2x')
'Integer(2)*x'
sage: implicit_multiplication(False)
sage: preparse('2x')
'2x'
```

`sage.repl.preparse.`**`in_quote`**`()`

`sage.repl.preparse.`**`isalphadigit_`**`(s)`
> Return `True` if `s` is a non-empty string of alphabetic characters or a non-empty string of digits or just a single `_`

> EXAMPLES:

```
sage: from sage.repl.preparse import isalphadigit_
sage: isalphadigit_('abc')
True
sage: isalphadigit_('123')
True
sage: isalphadigit_('_')
True
sage: isalphadigit_('a123')
False
```

`sage.repl.preparse.`**`parse_ellipsis`**`(code, preparse_step=True)`
> Preparses [0,2,..,n] notation.

> INPUT:

> - `code` - a string

> - `preparse_step` - a boolean (default: True)

> OUTPUT:

> - a string

> EXAMPLES:

```
sage: from sage.repl.preparse import parse_ellipsis
sage: parse_ellipsis("[1,2,..,n]")
'(ellipsis_range(1,2,Ellipsis,n))'
sage: parse_ellipsis("for i in (f(x) .. L[10]):")
'for i in (ellipsis_iter(f(x) ,Ellipsis, L[10])):'
sage: [1.0..2.0]
[1.00000000000000, 2.00000000000000]
```

`sage.repl.preparse.`**`preparse`**`(line, reset=True, do_time=False, ignore_prompts=False, nu-`
> `meric_literals=True)`
> Preparses a line of input.

> INPUT:

> - `line` - a string

- •`reset` - a boolean (default: True)

- •`do_time` - a boolean (default: False)

- •`ignore_prompts` - a boolean (default: False)

- •`numeric_literals` - a boolean (default: True)

OUTPUT:

- •a string

EXAMPLES:

```
sage: preparse("ZZ.<x> = ZZ['x']")
"ZZ = ZZ['x']; (x,) = ZZ._first_ngens(1)"
sage: preparse("ZZ.<x> = ZZ['y']")
"ZZ = ZZ['y']; (x,) = ZZ._first_ngens(1)"
sage: preparse("ZZ.<x,y> = ZZ[]")
"ZZ = ZZ['x, y']; (x, y,) = ZZ._first_ngens(2)"
sage: preparse("ZZ.<x,y> = ZZ['u,v']")
"ZZ = ZZ['u,v']; (x, y,) = ZZ._first_ngens(2)"
sage: preparse("ZZ.<x> = QQ[2^(1/3)]")
'ZZ = QQ[Integer(2)**(Integer(1)/Integer(3))]; (x,) = ZZ._first_ngens(1)'
sage: QQ[2^(1/3)]
Number Field in a with defining polynomial x^3 - 2

sage: preparse("a^b")
'a**b'
sage: preparse("a^^b")
'a^b'
sage: 8^1
8
sage: 8^^1
9
sage: 9^^1
8

sage: preparse("A \ B")
'A  * BackslashOperator() * B'
sage: preparse("A^2 \ B + C")
'A**Integer(2)  * BackslashOperator() * B + C'
sage: preparse("a \\ b \\") # There is really only one backslash here, it's just␣
↪being escaped.
'a  * BackslashOperator() * b \\'

sage: preparse("time R.<x> = ZZ[]", do_time=True)
'__time__=misc.cputime(); __wall__=misc.walltime(); R = ZZ[\'x\']; print("Time:␣
↪CPU %.2f s, Wall: %.2f s"%(misc.cputime(__time__), misc.walltime(__wall__)));␣
↪(x,) = R._first_ngens(1)'
```

`sage.repl.preparse.`**`preparse_calculus`**(*code*)

Supports calculus-like function assignment, e.g., transforms:

```
f(x,y,z) = sin(x^3 - 4*y) + y^x
```

into:

```
__tmp__=var("x,y,z")
f = symbolic_expression(sin(x**3 - 4*y) + y**x).function(x,y,z)
```

AUTHORS:

- Bobby Moretti

  - Initial version - 02/2007

- William Stein

  - Make variables become defined if they aren't already defined.

- Robert Bradshaw

  - Rewrite using regular expressions (01/2009)

EXAMPLES:

```
sage: preparse("f(x) = x^3-x")
'__tmp__=var("x"); f = symbolic_expression(x**Integer(3)-x).function(x)'
sage: preparse("f(u,v) = u - v")
'__tmp__=var("u,v"); f = symbolic_expression(u - v).function(u,v)'
sage: preparse("f(x) =-5")
'__tmp__=var("x"); f = symbolic_expression(-Integer(5)).function(x)'
sage: preparse("f(x) -= 5")
'f(x) -= Integer(5)'
sage: preparse("f(x_1, x_2) = x_1^2 - x_2^2")
'__tmp__=var("x_1,x_2"); f = symbolic_expression(x_1**Integer(2) - x_
↪2**Integer(2)).function(x_1,x_2)'
```

For simplicity, this function assumes all statements begin and end with a semicolon:

```
sage: from sage.repl.preparse import preparse_calculus
sage: preparse_calculus(";f(t,s)=t^2;")
';__tmp__=var("t,s"); f = symbolic_expression(t^2).function(t,s);'
sage: preparse_calculus(";f( t , s ) = t^2;")
';__tmp__=var("t,s"); f = symbolic_expression(t^2).function(t,s);'
```

sage.repl.preparse.**preparse_file**(*contents*, *globals=None*, *numeric_literals=True*)
Preparses input, attending to numeric literals and load/attach file directives.

Note: Temporarily, if @parallel is in the input, then numeric_literals is always set to False.

INPUT:

- `contents` - a string

- `globals` - dict or None (default: None); if given, then arguments to load/attach are evaluated in the namespace of this dict.

- `numeric_literals` - bool (default: True), whether to factor out wrapping of integers and floats, so they don't get created repeatedly inside loops

OUTPUT:

- a string

sage.repl.preparse.**preparse_file_named**(*name*)
Preparse file named code{name} (presumably a .sage file), outputting to a temporary file. Returns name of temporary file.

sage.repl.preparse.**preparse_file_named_to_stream**(*name*, *out*)
Preparse file named code{name} (presumably a .sage file), outputting to stream code{out}.

---

sage.repl.preparse.**preparse_generators**(*code*)

Parses generator syntax, converting:

```
obj.<gen0,gen1,...,genN> = objConstructor(...)
```

into:

```
obj = objConstructor(..., names=("gen0", "gen1", ..., "genN"))
(gen0, gen1, ..., genN,) = obj.gens()
```

and:

```
obj.<gen0,gen1,...,genN> = R[interior]
```

into:

```
obj = R[interior]; (gen0, gen1, ..., genN,) = obj.gens()
```

INPUT:

- `code` - a string

OUTPUT:

- a string

LIMITATIONS:

- The entire constructor *must* be on one line.

AUTHORS:

- 2006-04-14: Joe Wetherell (jlwether@alum.mit.edu)

    – Initial version.

- 2006-04-17: William Stein

    – Improvements to allow multiple statements.

- 2006-05-01: William

    – Fix bug that Joe found.

- 2006-10-31: William

    – Fix so obj doesn't have to be mutated.

- 2009-01-27: Robert Bradshaw

    – Rewrite using regular expressions

sage.repl.preparse.**preparse_numeric_literals**(*code*, *extract=False*)

This preparses numerical literals into their Sage counterparts, e.g. Integer, RealNumber, and ComplexNumber.

INPUT:

- `code` - a string; a code block to preparse

- `extract` - a boolean (default: False); whether to create names for the literals and return a dictionary of name-construction pairs

OUTPUT:

- a string or (string, string:string dictionary) 2-tuple; the preparsed block and, if `extract` is True, the name-construction mapping

EXAMPLES:

```
sage: from sage.repl.preparse import preparse_numeric_literals
sage: preparse_numeric_literals("5")
'Integer(5)'
sage: preparse_numeric_literals("5j")
"ComplexNumber(0, '5')"
sage: preparse_numeric_literals("5jr")
'5J'
sage: preparse_numeric_literals("5l")
'5l'
sage: preparse_numeric_literals("5L")
'5L'
sage: preparse_numeric_literals("1.5")
"RealNumber('1.5')"
sage: preparse_numeric_literals("1.5j")
"ComplexNumber(0, '1.5')"
sage: preparse_numeric_literals(".5j")
"ComplexNumber(0, '.5')"
sage: preparse_numeric_literals("5e9j")
"ComplexNumber(0, '5e9')"
sage: preparse_numeric_literals("5.")
"RealNumber('5.')"
sage: preparse_numeric_literals("5.j")
"ComplexNumber(0, '5.')"
sage: preparse_numeric_literals("5.foo()")
'Integer(5).foo()'
sage: preparse_numeric_literals("5.5.foo()")
"RealNumber('5.5').foo()"
sage: preparse_numeric_literals("5.5j.foo()")
"ComplexNumber(0, '5.5').foo()"
sage: preparse_numeric_literals("5j.foo()")
"ComplexNumber(0, '5').foo()"
sage: preparse_numeric_literals("1.exp()")
'Integer(1).exp()'
sage: preparse_numeric_literals("1e+10")
"RealNumber('1e+10')"
sage: preparse_numeric_literals("0x0af")
'Integer(0x0af)'
sage: preparse_numeric_literals("0x10.sqrt()")
'Integer(0x10).sqrt()'
sage: preparse_numeric_literals('0o100')
'Integer(0o100)'
sage: preparse_numeric_literals('0b111001')
'Integer(0b111001)'
sage: preparse_numeric_literals('0xe')
'Integer(0xe)'
sage: preparse_numeric_literals('0xEAR')
'0xEA'
sage: preparse_numeric_literals('0x1012Fae')
'Integer(0x1012Fae)'
```

sage.repl.preparse.**strip_prompts**(*line*)

    Removes leading sage: and >>> prompts so that pasting of examples from the documentation works.

    INPUT:

        •line - a string to process

    OUTPUT:

> •a string stripped of leading prompts

EXAMPLES:

```
sage: from sage.repl.preparse import strip_prompts
sage: strip_prompts("sage: 2 + 2")
'2 + 2'
sage: strip_prompts(">>>   3 + 2")
'3 + 2'
sage: strip_prompts("  2 + 4")
'  2 + 4'
```

sage.repl.preparse.**strip_string_literals**(*code*, *state=None*)

> Returns a string with all literal quotes replaced with labels and a dictionary of labels for re-substitution. This makes parsing easier.

> INPUT:

> > •`code` - a string; the input

> > •`state` - a 2-tuple (default: None); state with which to continue processing, e.g., across multiple calls to this function

> OUTPUT:

> > •a 3-tuple of the processed code, the dictionary of labels, and any accumulated state

> EXAMPLES:

```
sage: from sage.repl.preparse import strip_string_literals
sage: s, literals, state = strip_string_literals(r'''['a', "b", 'c', "d\""]''')
sage: s
'[%(L1)s, %(L2)s, %(L3)s, %(L4)s]'
sage: literals
{'L1': "'a'", 'L2': '"b"', 'L3': "'c'", 'L4': '"d\\""'}
sage: print(s % literals)
['a', "b", 'c', "d\""]
sage: print(strip_string_literals(r'-"\\""-"\\"-')[0])
-%(L1)s-%(L2)s-
```

> Triple-quotes are handled as well:

```
sage: s, literals, state = strip_string_literals("[a, '''b''', c, '']")
sage: s
'[a, %(L1)s, c, %(L2)s]'
sage: print(s % literals)
[a, '''b''', c, '']
```

> Comments are substitute too:

```
sage: s, literals, state = strip_string_literals("code '#' # ccc 't'"); s
'code %(L1)s #%(L2)s'
sage: s % literals
"code '#' # ccc 't'"
```

> A state is returned so one can break strings across multiple calls to this function:

```
sage: s, literals, state = strip_string_literals('s = "some'); s
's = %(L1)s'
sage: s, literals, state = strip_string_literals('thing" * 5', state); s
'%(L1)s * 5'
```

LOADING AND ATTACHING FILES

Sage or Python files can be loaded (similar to Python's `execfile`) in a Sage session. Attaching is similar, except that the attached file is reloaded whenever it is changed.

3.1 Load Python, Sage, Cython, Fortran and Magma files in Sage

sage.repl.load.**is_loadable_filename**(*filename*)

Returns whether a file can be loaded into Sage. This checks only whether its name ends in one of the supported extensions `.py`, `.pyx`, `.sage`, `.spyx`, `.f`, `.f90` and `.m`. Note: *load()* assumes the latter signifies a Magma file.

INPUT:

- `filename` - a string

OUTPUT:

- a boolean

EXAMPLES:

```
sage: sage.repl.load.is_loadable_filename('foo.bar')
False
sage: sage.repl.load.is_loadable_filename('foo.c')
False
sage: sage.repl.load.is_loadable_filename('foo.sage')
True
sage: sage.repl.load.is_loadable_filename('FOO.F90')
True
sage: sage.repl.load.is_loadable_filename('foo.m')
True
```

sage.repl.load.**load**(*filename*, *globals*, *attach=False*)

Executes a file in the scope given by `globals`. If the name starts with `http://`, it is treated as a URL and downloaded.

Note: For Cython files, the situation is more complicated – the module is first compiled to a temporary module `t` and executed via:

```
from t import *
```

INPUT:

- `filename` – a string denoting a filename or URL.

- `globals` – a string:object dictionary; the context in which to execute the file contents.

- `attach` – a boolean (default: False); whether to add the file to the list of attached files.

EXAMPLES:

Note that `.py` files are *not* preparsed:

```
sage: t = tmp_filename(ext='.py')
sage: _ = open(t,'w').write("print 'hi', 2/3; z = -2/7")
sage: z = 1
sage: sage.repl.load.load(t, globals())  # optional - python2
hi 0
sage: z  # optional - python2
-1
```

A `.sage` file *is* preparsed:

```
sage: t = tmp_filename(ext='.sage')
sage: _ = open(t,'w').write("print 'hi', 2/3; z = -2/7")
sage: z = 1
sage: sage.repl.load.load(t, globals())
hi 2/3
sage: z
-2/7
```

Cython files are *not* preparsed:

```
sage: t = tmp_filename(ext='.pyx')
sage: _ = open(t,'w').write("print 'hi', 2/3; z = -2/7")
sage: z = 1
sage: sage.repl.load.load(t, globals())
Compiling ...
hi 0
sage: z
-1
```

If the file isn't a Cython, Python, or a Sage file, a ValueError is raised:

```
sage: sage.repl.load.load(tmp_filename(ext=".foo"), globals())
Traceback (most recent call last):
...
ValueError: unknown file extension '.foo' for load or attach (supported␣
→extensions: .py, .pyx, .sage, .spyx, .f, .f90, .m)
```

We load a file given at a remote URL:

```
sage: sage.repl.load.load('http://wstein.org/loadtest.py', globals())  # optional␣
→- internet
hi from the net
5
```

We can load files using secure http (https):

```
sage: sage.repl.load.load('https://github.com/jasongrout/minimum_rank/raw/minimum_
→rank_1_0_0/minrank.py', globals())  # optional - internet
```

We attach a file:

```
sage: t = tmp_filename(ext='.py')
sage: _ = open(t,'w').write("print 'hello world'")
sage: sage.repl.load.load(t, globals(), attach=True)
hello world
sage: t in attached_files()
True
```

You can't attach remote URLs (yet):

```
sage: sage.repl.load.load('http://wstein.org/loadtest.py', globals(),
→attach=True)  # optional - internet
Traceback (most recent call last):
...
NotImplementedError: you can't attach a URL
```

The default search path for loading and attaching files is the current working directory, i.e., `'.'`. But you can modify the path with *load_attach_path()*:

```
sage: sage.repl.attach.reset(); reset_load_attach_path()
sage: load_attach_path()
['.']
sage: t_dir = tmp_dir()
sage: fullpath = os.path.join(t_dir, 'test.py')
sage: _ = open(fullpath, 'w').write("print 37 * 3")
sage: load_attach_path(t_dir)
sage: attach('test.py')
111
sage: sage.repl.attach.reset(); reset_load_attach_path() # clean up
```

or by setting the environment variable SAGE_LOAD_ATTACH_PATH to a colon-separated list before starting Sage:

```
$ export SAGE_LOAD_ATTACH_PATH="/path/to/my/library:/path/to/utils"
$ sage
sage: load_attach_path()          # not tested
['.', '/path/to/my/library', '/path/to/utils']
```

sage.repl.load.**load_cython**(*name*)

Helper function to load a Cython file.

INPUT:

- name – filename of the Cython file

OUTPUT:

- A string with Python code to import the names from the compiled module.

sage.repl.load.**load_wrap**(*filename*, *attach=False*)

Encodes a load or attach command as valid Python code.

INPUT:

- filename - a string; the argument to the load or attach command

- attach - a boolean (default: False); whether to attach filename, instead of loading it

OUTPUT:

- a string

EXAMPLES:

```
sage: sage.repl.load.load_wrap('foo.py', True)
'sage.repl.load.load(sage.repl.load.base64.b64decode("Zm9vLnB5"),globals(),True)'
sage: sage.repl.load.load_wrap('foo.sage')
'sage.repl.load.load(sage.repl.load.base64.b64decode("Zm9vLnNhZ2U="),globals(),
↪False)'
sage: sage.repl.load.base64.b64decode("Zm9vLnNhZ2U=")
'foo.sage'
```

3.2 Keep track of attached files

sage.repl.attach.**add_attached_file**(*filename*)
Add to the list of attached files

This is a callback to be used from *load()* after evaluating the attached file the first time.

INPUT:

- `filename` – string, the fully qualified file name.

EXAMPLES:

```
sage: import sage.repl.attach as af
sage: af.reset()
sage: t = tmp_filename(ext='.py')
sage: af.add_attached_file(t)
sage: af.attached_files()
['/.../tmp_....py']
sage: af.detach(t)
sage: af.attached_files()
[]
```

sage.repl.attach.**attach**(**files*)
Attach a file or files to a running instance of Sage and also load that file.

Note: Attaching files uses the Python inputhook, which will conflict with other inputhook users. This generally includes GUI main loop integrations, for example tkinter. So you can only use tkinter or attach, but not both at the same time.

INPUT:

- `files` – a list of filenames (strings) to attach.

OUTPUT:

Each file is read in and added to an internal list of watched files. The meaning of reading in a file depends on the file type:

- `.py` files are read in with no preparsing (so, e.g., `2^3` is 2 bit-xor 3);

- `.sage` files are preparsed, then the result is read in;

- **.pyx files are *not* preparsed, but rather are compiled to a** module `m` and then `from m import *` is executed.

The contents of the file are then loaded, which means they are read into the running Sage session. For example, if `foo.sage` contains `x=5`, after attaching `foo.sage` the variable `x` will be set to 5. Moreover, any time

---

you change `foo.sage`, before you execute a command, the attached file will be re-read automatically (with no intervention on your part).

See also:

load() is the same as *attach()*, but doesn't automatically reload a file when it changes.

EXAMPLES:

You attach a file, e.g., `foo.sage` or `foo.py` or `foo.pyx`, to a running Sage session by typing:

```
sage: attach('foo.sage')    # not tested
```

Here we test attaching multiple files at once:

```
sage: sage.repl.attach.reset()
sage: t1 = tmp_filename(ext='.py')
sage: _ = open(t1,'w').write("print('hello world')")
sage: t2 = tmp_filename(ext='.py')
sage: _ = open(t2,'w').write("print('hi there xxx')")
sage: attach(t1, t2)
hello world
hi there xxx
sage: set(attached_files()) == set([t1,t2])
True
```

See also:

- *attached_files()* returns a list of all currently attached files.

- *detach()* instructs Sage to remove a file from the internal list of watched files.

- *load_attach_path()* allows you to get or modify the current search path for loading and attaching files.

sage.repl.attach.**attached_files**()
    Returns a list of all files attached to the current session with *attach()*.

    OUTPUT:

    The filenames in a sorted list of strings.

    EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: t = tmp_filename(ext='.py')
sage: _ = open(t,'w').write("print('hello world')")
sage: attach(t)
hello world
sage: attached_files()
['/....py']
sage: attached_files() == [t]
True
```

sage.repl.attach.**detach**(*filename*)
    Detach a file.

    This is the counterpart to *attach()*.

    INPUT:

    - `filename` – a string, or a list of strings, or a tuple of strings.

EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: t = tmp_filename(ext='.py')
sage: _ = open(t,'w').write("print('hello world')")
sage: attach(t)
hello world
sage: attached_files() == [t]
True
sage: detach(t)
sage: attached_files()
[]

sage: sage.repl.attach.reset(); reset_load_attach_path()
sage: load_attach_path()
['.']
sage: t_dir = tmp_dir()
sage: fullpath = os.path.join(t_dir, 'test.py')
sage: _ = open(fullpath, 'w').write("print(37 * 3)")
sage: load_attach_path(t_dir)
sage: attach('test.py')
111
sage: attached_files() == [os.path.normpath(fullpath)]
True
sage: detach('test.py')
sage: attached_files()
[]
sage: attach('test.py')
111
sage: fullpath = os.path.join(t_dir, 'test2.py')
sage: _ = open(fullpath, 'w').write("print(3)")
sage: attach('test2.py')
3
sage: detach(attached_files())
sage: attached_files()
[]
```

sage.repl.attach.**load_attach_mode**(*load_debug=None*, *attach_debug=None*)
   Get or modify the current debug mode for the behavior of load() and *attach()* on .sage files.

   In debug mode, loaded or attached .sage files are preparsed through a file to make their tracebacks more informative. If not in debug mode, then .sage files are preparsed in memory only for performance.

   At startup, debug mode is True for attaching and False for loading.

   Note: This function should really be deprecated and code executed from memory should raise proper tracebacks.

   INPUT:

   • load_debug – boolean or None (default); if not None, then set a new value for the debug mode for loading files.

   • attach_debug – boolean or None (default); same as load_debug, but for attaching files.

   OUTPUT:

   If all input values are None, returns a tuple giving the current modes for loading and attaching.

EXAMPLES:

```
sage: load_attach_mode()
(False, True)
sage: load_attach_mode(attach_debug=False)
sage: load_attach_mode()
(False, False)
sage: load_attach_mode(load_debug=True)
sage: load_attach_mode()
(True, False)
sage: load_attach_mode(load_debug=False, attach_debug=True)
```

sage.repl.attach.**load_attach_path**(*path=None*, *replace=False*)
Get or modify the current search path for load() and *attach()*.

INPUT:

- path – string or list of strings (default: None); path(s) to append to or replace the current path.

- replace – boolean (default: False); if path is not None, whether to *replace* the search path instead of *appending* to it.

OUTPUT:

None or a *reference* to the current search paths.

EXAMPLES:

First, we extend the example given in load()'s docstring:

```
sage: sage.repl.attach.reset(); reset_load_attach_path()
sage: load_attach_path()
['.']
sage: t_dir = tmp_dir()
sage: fullpath = os.path.join(t_dir, 'test.py')
sage: _ = open(fullpath, 'w').write("print(37 * 3)")
sage: attach('test.py')
Traceback (most recent call last):
...
IOError: did not find file 'test.py' to load or attach
sage: load_attach_path(t_dir)
sage: attach('test.py')
111
sage: attached_files() == [fullpath]
True
sage: sage.repl.attach.reset(); reset_load_attach_path()
sage: load_attach_path() == ['.']
True
sage: load('test.py')
Traceback (most recent call last):
...
IOError: did not find file 'test.py' to load or attach
```

The function returns a reference to the path list:

```
sage: reset_load_attach_path(); load_attach_path()
['.']
sage: load_attach_path('/path/to/my/sage/scripts'); load_attach_path()
['.', '/path/to/my/sage/scripts']
sage: load_attach_path(['good', 'bad', 'ugly'], replace=True)
sage: load_attach_path()
```

```
['good', 'bad', 'ugly']
sage: p = load_attach_path(); p.pop()
'ugly'
sage: p[0] = 'weird'; load_attach_path()
['weird', 'bad']
sage: reset_load_attach_path(); load_attach_path()
['.']
```

sage.repl.attach.**modified_file_iterator**()
> Iterate over the changed files

> As a side effect the stored time stamps are updated with the actual time stamps. So if you iterate over the attached files in order to reload them and you hit an error then the subsequent files are not marked as read.

> Files that are in the process of being saved are excluded.

> EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: t = tmp_filename(ext='.py')
sage: attach(t)
sage: from sage.repl.attach import modified_file_iterator
sage: list(modified_file_iterator())
[]
sage: sleep(1)    # filesystem mtime granularity
sage: _ = open(t, 'w').write('1')
sage: list(modified_file_iterator())
[('/.../tmp_....py', time.struct_time(...))]
```

sage.repl.attach.**reload_attached_files_if_modified**()
> Reload attached files that have been modified

> This is the internal implementation of the attach mechanism.

> EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: tmp = tmp_filename(ext='.py')
sage: _ = open(tmp, 'w').write('a = 2\n')
sage: shell.run_cell('attach({0})'.format(repr(tmp)))
sage: shell.run_cell('a')
2
sage: sleep(1)    # filesystem mtime granularity
sage: _ = open(tmp, 'w').write('a = 3\n')
```

> Note that the doctests are never really at the command prompt where the automatic reload is triggered. So we have to do it manually:

```
sage: shell.run_cell('from sage.repl.attach import reload_attached_files_if_
↪modified')
sage: shell.run_cell('reload_attached_files_if_modified()')
### reloading attached file tmp_....py modified at ... ###

sage: shell.run_cell('a')
3
sage: shell.run_cell('detach({0})'.format(repr(tmp)))
sage: shell.run_cell('attached_files()')
```

```
[]
sage: shell.quit()
```

sage.repl.attach.**reset**()

Remove all the attached files from the list of attached files.

EXAMPLES:

```
sage: sage.repl.attach.reset()
sage: t = tmp_filename(ext='.py')
sage: _ = open(t,'w').write("print('hello world')")
sage: attach(t)
hello world
sage: attached_files() == [t]
True
sage: sage.repl.attach.reset()
sage: attached_files()
[]
```

sage.repl.attach.**reset_load_attach_path**()

Resets the current search path for load() and *attach()*.

The default path is '.' plus any paths specified in the environment variable SAGE_LOAD_ATTACH_PATH.

EXAMPLES:

```
sage: load_attach_path()
['.']
sage: t_dir = tmp_dir()
sage: load_attach_path(t_dir)
sage: t_dir in load_attach_path()
True
sage: reset_load_attach_path(); load_attach_path()
['.']
```

At startup, Sage adds colon-separated paths in the environment variable SAGE_LOAD_ATTACH_PATH:

```
sage: reset_load_attach_path(); load_attach_path()
['.']
sage: os.environ['SAGE_LOAD_ATTACH_PATH'] = '/veni/vidi:vici:'
sage: import imp
sage: imp.reload(sage.repl.attach)     # Simulate startup
<module 'sage.repl.attach' from '...'>
sage: load_attach_path()
['.', '/veni/vidi', 'vici']
sage: del os.environ['SAGE_LOAD_ATTACH_PATH']
sage: imp.reload(sage.repl.preparse)     # Simulate startup
<module 'sage.repl.preparse' from '...'>
sage: reset_load_attach_path(); load_attach_path()
['.']
```

PRETTY PRINTING

In addition to making input nicer, we also modify how results are printed. This again builds on how IPython formats output. Technically, this works using a modified displayhook in Python.

4.1 IPython Displayhook Formatters

The classes in this module can be used as IPython displayhook formatters. It has two main features, by default the displayhook contains a new facility for displaying lists of matrices in an easier to read format:

```
sage: [identity_matrix(i) for i in range(2,5)]
[
                [1 0 0 0]
       [1 0 0]  [0 1 0 0]
[1 0]  [0 1 0]  [0 0 1 0]
[0 1], [0 0 1], [0 0 0 1]
]
```

This facility uses _repr_() (and a simple string) to try do a nice read format (see sage.structure.parent. Parent._repr_option() for details).

With this displayhook there exists an other way for displaying object and more generally, all sage expression as an ASCII art object:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%display ascii_art')
sage: shell.run_cell('integral(x^2/pi^x, x)')
 / 2    2                      \  -x*log(pi)
-\x *log (pi) + 2*x*log(pi) + 2/*e
---------------------------------------------
                    3
               log (pi)
sage: shell.run_cell("i = var('i')")
sage: shell.run_cell('sum(i*x^i, i, 0, 10)')
    10       9      8      7      6      5      4      3      2
10*x   + 9*x  + 8*x  + 7*x  + 6*x  + 5*x  + 4*x  + 3*x  + 2*x  + x
sage: shell.run_cell('StandardTableaux(4).list()')
[
[                                                                  1  4
[                 1  3  4    1  2  4    1  2  3    1  3    1  2    2
[   1  2  3  4,   2      ,   3      ,   4      ,   2  4,   3  4,   3     ,

                 1 ]
```

```
   1 3    1 2     2 ]
   2       3      3 ]
   4    ,  4    ,  4 ]
sage: shell.run_cell('%display default')
sage: shell.quit()
```

This other facility uses a simple `AsciiArt` object (see and `sage.structure.sage_object.SageObject._ascii_art_()`).

class `sage.repl.display.formatter.`**`SageDisplayFormatter`**(**args*, ***kwds*)

   Bases: `IPython.core.formatters.DisplayFormatter`

   This is where the Sage rich objects are translated to IPython

   INPUT/OUTPUT:

   See the IPython documentation.

   EXAMPLES:

   This is part of how Sage works with the IPython output system. It cannot be used in doctests:

```
sage: from sage.repl.display.formatter import SageDisplayFormatter
sage: fmt = SageDisplayFormatter()
Traceback (most recent call last):
...
RuntimeError: check failed: current backend is invalid
```

   `format`(*obj*, *include=None*, *exclude=None*)

      Use the Sage rich output instead of IPython

      INPUT/OUTPUT:

      See the IPython documentation.

      EXAMPLES:

```
sage: [identity_matrix(i) for i in range(3,7)]
[
                                    [1 0 0 0 0 0]
                    [1 0 0 0 0]     [0 1 0 0 0 0]
          [1 0 0 0] [0 1 0 0 0]     [0 0 1 0 0 0]
[1 0 0]   [0 1 0 0] [0 0 1 0 0]     [0 0 0 1 0 0]
[0 1 0]   [0 0 1 0] [0 0 0 1 0]     [0 0 0 0 1 0]
[0 0 1], [0 0 0 1], [0 0 0 0 1],    [0 0 0 0 0 1]
]
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%display ascii_art')    # indirect doctest
sage: shell.run_cell("i = var('i')")
sage: shell.run_cell('sum(i*x^i, i, 0, 10)')
     10      9      8      7      6      5      4      3      2
10*x    + 9*x  + 8*x  + 7*x  + 6*x  + 5*x  + 4*x  + 3*x  + 2*x  + x
sage: shell.run_cell('%display default')
sage: shell.quit()
```

class `sage.repl.display.formatter.`**`SagePlainTextFormatter`**(**args*, ***kwds*)

   Bases: `IPython.core.formatters.PlainTextFormatter`

   Improved plain text IPython formatter.

In particular, it correctly print lists of matrices or other objects (see `sage.structure.parent.Parent._repr_option()`).

> **Warning:** This IPython formatter is NOT used. You could use it to enable Sage formatting in IPython, but Sage uses its own rich output system that is more flexible and supports different backends.

INPUT/OUTPUT:

See the IPython documentation.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.display_formatter.formatters['text/plain']
<IPython.core.formatters.PlainTextFormatter object at 0x...>
sage: shell.quit()
```

4.2 The Sage pretty printer

Any transformation to a string goes through here. In other words, the `SagePlainTextFormatter` is entirely implemented via *SagePrettyPrinter*. Other formatters may or may not use *SagePrettyPrinter* to generate text output.

AUTHORS:

- Bill Cauchois (2009): initial version
- Jean-Baptiste Priez <jbp@kerios.fr> (2013): ASCII art
- Volker Braun (2013): refactored into DisplayHookBase

class `sage.repl.display.pretty_print.`**`SagePrettyPrinter`**(*output*, *max_width*, *newline*, *max_seq_length=None*)

Bases: `IPython.lib.pretty.PrettyPrinter`

Pretty print Sage objects for the commandline

INPUT:

See IPython documentation.

EXAMPLES:

```
sage: 123
123
```

IPython pretty printers:

```
sage: set({1, 2, 3})
{1, 2, 3}
sage: dict(zzz=123, aaa=99, xab=10)    # sorted by keys
{'aaa': 99, 'xab': 10, 'zzz': 123}
```

These are overridden in IPython in a way that we feel is somewhat confusing, and we prefer to print them like plain Python which is more informative. See trac ticket #14466

```
sage: 'this is a string'
'this is a string'
sage: type(123)
<type 'sage.rings.integer.Integer'>
sage: type
<... 'type'>
sage: import types
sage: type('name', (), {})
<class '__main__.name'>
sage: types.BuiltinFunctionType
<type 'builtin_function_or_method'>

sage: def foo(): pass
sage: foo
<function foo at 0x...>
```

pretty(*obj*)

Pretty print obj

This is the only method that outside code should invoke.

INPUT:

- obj – anything.

OUTPUT:

String representation for object.

EXAMPLES:

```
sage: from sage.repl.display.pretty_print import SagePrettyPrinter
sage: from six import StringIO
sage: stream = StringIO()
sage: SagePrettyPrinter(stream, 78, '\n').pretty([type, 123, 'foo'])
sage: stream.getvalue()
"[<... 'type'>,"
```

toplevel()

Return whether we are currently at the top level.

OUTPUT:

Boolean. Whether we are currently pretty-printing an object at the outermost level (True), or whether the object is inside a container (False).

EXAMPLES:

```
sage: from sage.repl.display.pretty_print import SagePrettyPrinter
sage: from six import StringIO
sage: stream = StringIO()
sage: spp = SagePrettyPrinter(stream, 78, '\n')
sage: spp.toplevel()
True
```

4.3 Representations of objects.

class sage.repl.display.fancy_repr.**LargeMatrixHelpRepr**

Bases: *sage.repl.display.fancy_repr.ObjectReprABC*

Representation including help for large Sage matrices

__call__(*obj*, *p*, *cycle*)

Format matrix.

INPUT:

- •obj – anything. Object to format.

- •p – PrettyPrinter instance.

- •cycle – boolean. Whether there is a cycle.

OUTPUT:

Boolean. Whether the representer is applicable to obj. If True, the string representation is appended to p.

EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import LargeMatrixHelpRepr
sage: M = identity_matrix(40)
sage: pp = LargeMatrixHelpRepr()
sage: pp.format_string(M)
"40 x 40 dense matrix over Integer Ring (use the '.str()' method to see the
↪entries)"
sage: pp.format_string([M, M])
'--- object not handled by representer ---'
```

Leads to:

```
sage: M
40 x 40 dense matrix over Integer Ring (use the '.str()' method to see the
↪entries)
sage: [M, M]
[40 x 40 dense matrix over Integer Ring,
 40 x 40 dense matrix over Integer Ring]
```

class sage.repl.display.fancy_repr.**ObjectReprABC**

Bases: object

The abstract base class of an object representer.

__call__(*obj*, *p*, *cycle*)

Format object.

INPUT:

- •obj – anything. Object to format.

- •p – PrettyPrinter instance.

- •cycle – boolean. Whether there is a cycle.

OUTPUT:

Boolean. Whether the representer is applicable to obj. If True, the string representation is appended to p.

EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import ObjectReprABC
sage: ObjectReprABC().format_string(123)    # indirect doctest
'Error: ObjectReprABC.__call__ is abstract'
```

format_string(*obj*)

For doctesting only: Directly return string.

INPUT:

- •obj – anything. Object to format.

OUTPUT:

String.

EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import ObjectReprABC
sage: ObjectReprABC().format_string(123)
'Error: ObjectReprABC.__call__ is abstract'
```

class sage.repl.display.fancy_repr.**PlainPythonRepr**

Bases: *sage.repl.display.fancy_repr.ObjectReprABC*

The ordinary Python representation

__call__(*obj*, *p*, *cycle*)

Format matrix.

INPUT:

- •obj – anything. Object to format.

- •p – PrettyPrinter instance.

- •cycle – boolean. Whether there is a cycle.

OUTPUT:

Boolean. Whether the representer is applicable to obj. If True, the string representation is appended to p.

EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import PlainPythonRepr
sage: pp = PlainPythonRepr()
sage: pp.format_string(type(1))
"<type 'sage.rings.integer.Integer'>"
```

Do not swallow a trailing newline at the end of the output of a custom representer. Note that it is undesirable to have a trailing newline, and if we don't display it you can't fix it:

```
sage: class Newline(object):
....:     def __repr__(self):
....:         return 'newline\n'
sage: n = Newline()
sage: pp.format_string(n)
'newline\n'
sage: pp.format_string([n, n, n])
'[newline\n, newline\n, newline\n]'
sage: [n, n, n]
```

```
[newline
, newline
, newline
]
```

class sage.repl.display.fancy_repr.**SomeIPythonRepr**

    Bases: *sage.repl.display.fancy_repr.ObjectReprABC*

    Some selected representers from IPython

    EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import SomeIPythonRepr
sage: SomeIPythonRepr()
SomeIPythonRepr pretty printer
```

    __call__(*obj*, *p*, *cycle*)

        Format object.

        INPUT:

            •obj – anything. Object to format.

            •p – PrettyPrinter instance.

            •cycle – boolean. Whether there is a cycle.

        OUTPUT:

        Boolean. Whether the representer is applicable to obj. If True, the string representation is appended to p.

        EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import SomeIPythonRepr
sage: pp = SomeIPythonRepr()
sage: pp.format_string(set([1, 2, 3]))
'{1, 2, 3}'
```

class sage.repl.display.fancy_repr.**TallListRepr**

    Bases: *sage.repl.display.fancy_repr.ObjectReprABC*

    Special representation for lists with tall entries (e.g. matrices)

    __call__(*obj*, *p*, *cycle*)

        Format list/tuple.

        INPUT:

            •obj – anything. Object to format.

            •p – PrettyPrinter instance.

            •cycle – boolean. Whether there is a cycle.

        OUTPUT:

        Boolean. Whether the representer is applicable to obj. If True, the string representation is appended to p.

        EXAMPLES:

```
sage: from sage.repl.display.fancy_repr import TallListRepr
sage: format_list = TallListRepr().format_string
sage: format_list([1, 2, identity_matrix(2)])
'[\n     [1 0]\n1, 2, [0 1]\n]'
```

Check that trac ticket #18743 is fixed:

```
sage: class Foo(object):
....:     def __repr__(self):
....:         return '''BBB    AA    RRR
....: B B   A  A   R  R
....: BBB    AAAA  RRR
....: B B   A  A   R  R
....: BBB   A  A   R   R'''
....:     def _repr_option(self, key):
....:         return key == 'ascii_art'
sage: F = Foo()
sage: [F, F]
[
BBB     AA    RRR     BBB     AA    RRR
B B    A  A   R  R    B B    A  A   R  R
BBB    AAAA   RRR     BBB    AAAA   RRR
B B    A  A   R  R    B B    A  A   R  R
BBB    A  A   R   R, BBB    A  A   R   R
]
```

4.4 Utility functions for pretty-printing

These utility functions are used in the implementations of _repr_ methods elsewhere.

class sage.repl.display.util.**TallListFormatter**
    Bases: object

    Special representation for lists with tall entries (e.g. matrices)

    __call__(*the_list*)
        Return "tall list" string representation.

        See also *try_format()*.

        INPUT:

        • the_list – list or tuple.

        OUTPUT:

        String.

        EXAMPLES:

```
sage: from sage.repl.display.util import format_list
sage: format_list(['not', 'tall'])
"['not', 'tall']"
```

    try_format(*the_list*)
        First check whether a list is "tall" – whether the reprs of the elements of the list will span multiple lines and cause the list to be printed awkwardly. If not, this function returns None and does nothing; you should revert back to the normal method for printing an object (its repr). If so, return the string in the

special format. Note that the special format isn't just for matrices. Any object with a multiline repr will be formatted.

INPUT:

- `the_list` - The list (or a tuple).

OUTPUT:

String or `None`. The latter is returned if the list is not deemed to be tall enough and another formatter should be used.

FIVE

DISPLAY BACKEND INFRASTRUCTURE

5.1 Display Manager

This is the heart of the rich output system, the display manager arbitrates between

- Backend capabilities: what can be displayed

- Backend preferences: what gives good quality on the backend

- Sage capabilities: every Sage object can only generate certain representations, and

- User preferences: typeset vs. plain text vs. ascii art, etc.

The display manager is a singleton class, Sage always has exactly one instance of it. Use *get_display_manager()* to obtain it.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager();  dm
The Sage display manager using the doctest backend
```

exception sage.repl.rich_output.display_manager.**DisplayException**

　　Bases: exceptions.Exception

　　Base exception for all rich output-related exceptions.

　　EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import DisplayException
sage: raise DisplayException('foo')
Traceback (most recent call last):
...
DisplayException: foo
```

class sage.repl.rich_output.display_manager.**DisplayManager**

　　Bases: sage.structure.sage_object.SageObject

　　The Display Manager

　　Used to decide what kind of rich output is best.

　　EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: get_display_manager()
The Sage display manager using the doctest backend
```

check_backend_class(*backend_class*)

    Check that the current backend is an instance of `backend_class`.

    This is, for example, used by the Sage IPython display formatter to ensure that the IPython backend is in use.

    INPUT:

        •`backend_class` – type of a backend class.

    OUTPUT:

    This method returns nothing. A `RuntimeError` is raised if `backend_class` is not the type of the current backend.

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.check_backend_class(BackendSimple)
Traceback (most recent call last):
...
RuntimeError: check failed: current backend is invalid
```

display_immediately(*obj*, ***rich_repr_kwds*)

    Show output without going back to the command line prompt.

    This method must be called to create rich output from an object when we are not returning to the command line prompt, for example during program execution. Typically, it is being called by `sage.plot.graphics.Graphics.show()`.

    INPUT:

        •`obj` – anything. The object to be shown.

        •`rich_repr_kwds` – optional keyword arguments that are passed through to `obj._rich_repr_`.

    EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.display_immediately(1/2)
1/2
```

displayhook(*obj*)

    Implementation of the displayhook

    Every backend must pass the value of the last statement of a line / cell to this method. See also *display_immediately()* if you want do display rich output while a program is running.

    INPUT:

        •`obj` – anything. The object to be shown.

    OUTPUT:

    Returns whatever the backend's displayhook method returned.

    EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
```

```
sage: dm.displayhook(1/2)
1/2
```

classmethod get_instance()
> Get the singleton instance.
>
> This class method is equivalent to *get_display_manager()*.
>
> OUTPUT:
>
> The display manager singleton.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import DisplayManager
sage: DisplayManager.get_instance()
The Sage display manager using the doctest backend
```

graphics_from_save(*save_function*, *save_kwds*, *file_extension*, *output_container*, *figsize=None*, *dpi=None*)
> Helper to construct graphics.
>
> This method can be used to simplify the implementation of a _rich_repr_ method of a graphics object if there is already a function to save graphics to a file.
>
> INPUT:
>
> • `save_function` – callable that can save graphics to a file and accepts options like `sage.plot.graphics.Graphics.save()`.
>
> • `save_kwds` – dictionary. Keyword arguments that are passed to the save function.
>
> • `file_extension` – string starting with `'.'`. The file extension of the graphics file.
>
> • `output_container` – subclass of *sage.repl.rich_output.output_basic.OutputBase*. The output container to use. Must be one of the types in *supported_output()*.
>
> • `figsize` – pair of integers (optional). The desired graphics size in pixels. Suggested, but need not be respected by the output.
>
> • `dpi` – integer (optional). The desired resolution in dots per inch. Suggested, but need not be respected by the output.
>
> OUTPUT:
>
> Return an instance of `output_container`.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: plt = plot(sin)
sage: out = dm.graphics_from_save(plt.save, dict(), '.png', dm.types.
↪OutputImagePng)
sage: out
OutputImagePng container
sage: out.png.get().startswith('\x89PNG')
True
sage: out.png.filename()    # random
'/home/user/.sage/temp/localhost.localdomain/23903/tmp_pu5woK.png'
```

is_in_terminal()
> Test whether the UI is meant to run in a terminal

When this method returns `True`, you can assume that it is possible to use `raw_input` or launch external programs that take over the input.

Otherwise, you should assume that the backend runs remotely or in a pty controlled by another program. Then you should not launch external programs with a (text or graphical) UI.

This is used to enable/disable interpreter consoles.

OUTPUT:

Boolean.

preferences
>   Return the preferences.
>
>   OUTPUT:
>
>   The display preferences as instance of *DisplayPreferences*.
>
>   EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.preferences
Display preferences:
* graphics is not specified
* supplemental_plot = never
* text is not specified
```

supported_output()
>   Return the output container classes that can be used.
>
>   OUTPUT:
>
>   Frozen set of subclasses of *OutputBase*. If the backend defines derived container classes, this method will always return their base classes.
>
>   EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.types.OutputPlainText in dm.supported_output()
True
sage: type(dm.supported_output())
<... 'frozenset'>
```

switch_backend(*backend*, ***kwds*)
>   Switch to a new backend
>
>   INPUT:
>
>   • `backend` – instance of *BackendBase*.
>
>   • `kwds` – optional keyword arguments that are passed on to the *install()* method.
>
>   OUTPUT:
>
>   The previous backend.
>
>   EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: simple = BackendSimple()
sage: from sage.repl.rich_output import get_display_manager
```

```
sage: dm = get_display_manager();   dm
The Sage display manager using the doctest backend

sage: previous = dm.switch_backend(simple)
sage: dm
The Sage display manager using the simple backend
```

Restore the doctest backend:

```
sage: dm.switch_backend(previous) is simple
True
```

threejs_scripts(*online*)

Return Three.js script tags for the current backend.

INPUT:

- online – Boolean determining script usage context

OUTPUT:

String containing script tags

Note: This base method handles online=True case only, serving CDN script tags. Location of scripts for offline usage is backend-specific.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: get_display_manager().threejs_scripts(online=True)
'...<script src="https://cdn.rawgit.com/mrdoob/three.js/...'
sage: get_display_manager().threejs_scripts(online=False)
Traceback (most recent call last):
...
ValueError: current backend does not support
offline threejs graphics
```

types

Catalog of all output container types.

Note that every output type must be registered in *sage.repl.rich_output.output_catalog*.

OUTPUT:

Returns the *sage.repl.rich_output.output_catalog* module.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.types.OutputPlainText
<class 'sage.repl.rich_output.output_basic.OutputPlainText'>
```

exception sage.repl.rich_output.display_manager.**OutputTypeException**

Bases: *sage.repl.rich_output.display_manager.DisplayException*

Wrong Output container.

The output containers are the subclasses of *OutputBase* that contain the entire output. The display backends must create output containers of a suitable type depending on the displayed Python object. This exception indicates that there is a mistake in the backend and it returned the wrong type of output container.

EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import OutputTypeException
sage: raise OutputTypeException('foo')
Traceback (most recent call last):
...
OutputTypeException: foo
```

exception `sage.repl.rich_output.display_manager.`**`RichReprWarning`**
    Bases: `exceptions.UserWarning`

Warning that is throws if a call to _rich_repr_ fails.

If an object implements `_rich_repr_` then it must return a value, possibly `None` to indicate that no rich output can be generated. But it may not raise an exception as it is very confusing for the user if the displayhook fails.

EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import RichReprWarning
sage: raise RichReprWarning('foo')
Traceback (most recent call last):
...
RichReprWarning: foo
```

`sage.repl.rich_output.display_manager.`**`get_display_manager = <bound method type.get_instance of <class '`**
    Get the singleton instance.

    This class method is equivalent to *get_display_manager()*.

    OUTPUT:

    The display manager singleton.

    EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import DisplayManager
sage: DisplayManager.get_instance()
The Sage display manager using the doctest backend
```

class `sage.repl.rich_output.display_manager.`**`restricted_output`**(*display_manager*,
                                                                              output_classes)
    Bases: `object`

Context manager to temporarily restrict the accepted output types

In the context, the output is restricted to the output container types listed in `output_classes`. Additionally, display preferences are changed not not show graphics.

INPUT:

- `display_manager` – the display manager.

- `output_classes` – iterable of output container types.

EXAMPLES:

```
sage: from sage.repl.rich_output.display_manager import (
....:     get_display_manager, restricted_output)
sage: dm = get_display_manager()
sage: restricted_output(dm, [dm.types.OutputPlainText])
<sage.repl.rich_output.display_manager.restricted_output object at 0x...>
```

5.2 Display Preferences

This class is used to express display preferences that are not simply a choice of a particular output format. For example, whether to prefer vector over raster graphics. By convention, the value `None` is always a valid value for a preference and means no particular preference.

EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import DisplayPreferences
sage: prefs = DisplayPreferences()
sage: prefs.available_options()
(graphics, supplemental_plot, text)
sage: prefs.text is None
True
sage: prefs.text = 'ascii_art'
sage: prefs.text
'ascii_art'
sage: prefs
Display preferences:
* graphics is not specified
* supplemental_plot is not specified
* text = ascii_art
```

Properties can be unset by deleting them or by assigning `None`:

```
sage: prefs.text = 'ascii_art'
sage: del prefs.text
sage: prefs.text is None
True

sage: prefs.text = 'ascii_art'
sage: prefs.text = None
sage: prefs.text is None
True
```

Properties have documentation attached:

```
sage: import pydoc
sage: doc = pydoc.render_doc(prefs)
sage: assert ' graphics' in doc
sage: assert '    Preferred graphics format' in doc
sage: assert ' text' in doc
sage: assert '    Which textual representation is preferred' in doc
```

Values can also be specified as keyword arguments to the constructor:

```
sage: DisplayPreferences(text='latex')
Display preferences:
* graphics is not specified
```

```
* supplemental_plot is not specified
* text = latex
```

Todo

A value-checking preference system should be used elsewhere in Sage, too. The class here is just a simple implementation, a proper implementation would use a metaclass to construct the preference items.

---

class sage.repl.rich_output.preferences.**DisplayPreferences**(*args*, ***kwds*)

    Bases: *sage.repl.rich_output.preferences.PreferencesABC*

    Preferences for displaying graphics

    These can be preferences expressed by the user or by the display backend. They are specified as keyword arguments.

    INPUT:

        •*args* – positional arguments are preferences instances. The property values will be inherited from left to right, that is, later parents override values from earlier parents.

        •***kwds* – keyword arguments. Will be used to initialize properties, and override inherited values if necessary.

    EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import DisplayPreferences
sage: p1 = DisplayPreferences(graphics='vector')
sage: p2 = DisplayPreferences(graphics='raster')
sage: DisplayPreferences(p1, p2)
Display preferences:
* graphics = raster
* supplemental_plot is not specified
* text is not specified
```

    If specified in the opposite order, the setting from p1 is inherited:

```
sage: DisplayPreferences(p2, p1)
Display preferences:
* graphics = vector
* supplemental_plot is not specified
* text is not specified
```

    Further keywords override:

```
sage: DisplayPreferences(p2, p1, graphics='disable')
Display preferences:
* graphics = disable
* supplemental_plot is not specified
* text is not specified
```

    graphics

        Preferred graphics format

        Allowed values:

            •None (default): no preference

            •'disable'

- •'vector'

- •'raster'

supplemental_plot

    Whether to graphically display graphs and other graph-like objects that implement rich output. When not specified small objects are show graphically and large objects as textual overview.

    Allowed values:

- •`None` (default): no preference

- •'always'

- •'never'

text

    Which textual representation is preferred

    Allowed values:

- •`None` (default): no preference

- •'plain'

- •'ascii_art'

- •'unicode_art'

- •'latex'

class `sage.repl.rich_output.preferences.`**`PreferencesABC`**(*args*, ***kwds*)

    Bases: `sage.structure.sage_object.SageObject`

    Preferences for displaying graphics

    These can be preferences expressed by the user or by the display backend. They are specified as keyword arguments.

    INPUT:

- •`*args*` – positional arguments are preferences instances. The property values will be inherited from left to right, that is, later parents override values from earlier parents.

- •`**kwds` – keyword arguments. Will be used to initialize properties, and override inherited values if necessary.

    EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import DisplayPreferences
sage: p1 = DisplayPreferences(graphics='vector')
sage: p2 = DisplayPreferences(graphics='raster')
sage: DisplayPreferences(p1, p2)
Display preferences:
* graphics = raster
* supplemental_plot is not specified
* text is not specified
```

    If specified in the opposite order, the setting from `p1` is inherited:

```
sage: DisplayPreferences(p2, p1)
Display preferences:
* graphics = vector
* supplemental_plot is not specified
* text is not specified
```

Further keywords override:

```
sage: DisplayPreferences(p2, p1, graphics='disable')
Display preferences:
* graphics = disable
* supplemental_plot is not specified
* text is not specified
```

available_options()

> Return the available options

> OUTPUT:

> Tuple of the preference items as instances of *Property*.

> EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import DisplayPreferences
sage: DisplayPreferences().available_options()
(graphics, supplemental_plot, text)
```

class sage.repl.rich_output.preferences.**Property**(*name*, *allowed_values*, *doc=None*)

> Bases: property

> Preference item

> INPUT:

> • name – string. The name of the property.

> • allowed_values – list/tuple/iterable of allowed values.

> • doc – string (optional). The docstring of the property.

> EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import Property
sage: prop = Property('foo', [0, 1, 2], 'The Foo Property')
sage: prop.__doc__
'The Foo Property\n\nAllowed values:\n\n* ``None`` (default): no preference\n\n*␣
↪0\n\n* 1\n\n* 2'
sage: prop.allowed_values
(0, 1, 2)
```

deleter(*prefs*)

> Delete the current value of the property

> INPUT:

> > • prefs – the *PreferencesABC* instance that the property is bound to.

> EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import Property, PreferencesABC
sage: prop = Property('foo', [0, 1, 2], 'The Foo Property')
sage: prefs = PreferencesABC()
sage: prop.getter(prefs) is None
True
sage: prop.setter(prefs, 1)
sage: prop.deleter(prefs)
sage: prop.getter(prefs) is None
True
```

getter(*prefs*)

>   Get the current value of the property

>   INPUT:

>> •`prefs` – the *PreferencesABC* instance that the property is bound to.

>   OUTPUT:

>   One of the allowed values or `None` if not set.

>   EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import Property, PreferencesABC
sage: prop = Property('foo', [0, 1, 2], 'The Foo Property')
sage: prefs = PreferencesABC()
sage: prop.getter(prefs) is None
True
sage: prop.setter(prefs, 1)
sage: prop.getter(prefs)
1
```

setter(*prefs*, *value*)

>   Get the current value of the property

>   INPUT:

>> •`prefs` – the *PreferencesABC* instance that the property is bound to.

>> •`value` – anything. The new value of the property. Setting a property to `None` is equivalent to deleting the value.

>   OUTPUT:

>   This method does not return anything. A `ValueError` is raised if the given `value` is not one of the allowed values.

>   EXAMPLES:

```
sage: from sage.repl.rich_output.preferences import Property, PreferencesABC
sage: prop = Property('foo', [0, 1, 2], 'The Foo Property')
sage: prefs = PreferencesABC()
sage: prop.getter(prefs) is None
True
sage: prop.setter(prefs, 1)
sage: prop.getter(prefs)
1

sage: prop.setter(prefs, None)
sage: prop.getter(prefs) is None
True
```

5.3 Output Buffer

This is the fundamental unit of rich output, a single immutable buffer (either in-memory or as a file). Rich output always consists of one or more buffers. Ideally, the Sage library always uses the buffer object as an in-memory buffer. But you can also ask it for a filename, and it will save the data to a file if necessary. Either way, the buffer object presents the same interface for getting the content of an in-memory buffer or a temporary file. So any rich output backends do not need to know where the buffer content is actually stored.

EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: buf = OutputBuffer('this is the buffer content');  buf
buffer containing 26 bytes
sage: buf.get()
'this is the buffer content'
```

class sage.repl.rich_output.buffer.**OutputBuffer**(*data*)

> Bases: sage.structure.sage_object.SageObject

> Data stored either in memory or as a file

> This class is an abstraction for "files", in that they can either be defined by a bytes array (Python 3) or string (Python 2) or by a file (see *from_file()*).

> INPUT:

>> •data – bytes. The data that is stored in the buffer.

> EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: buf = OutputBuffer('this is the buffer content');  buf
buffer containing 26 bytes

sage: buf2 = OutputBuffer(buf);  buf2
buffer containing 26 bytes

sage: buf.get()
'this is the buffer content'
sage: buf.filename(ext='.txt')
'/....txt'
```

> **filename**(*ext=None*)

>> Return the filename.

>> INPUT:

>>> •ext – string. The file extension.

>> OUTPUT:

>> Name of a file, most likely a temporary file. If ext is specified, the filename will have that extension.

>> You must not modify the returned file. Its permissions are set to readonly to help with that.

>> EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: buf = OutputBuffer('test')
sage: buf.filename()             # random output
'/home/user/.sage/temp/hostname/26085/tmp_RNSfAc'

sage: os.path.isfile(buf.filename())
True
sage: buf.filename(ext='txt')   # random output
'/home/user/.sage/temp/hostname/26085/tmp_Rjjp4V.txt'
sage: buf.filename(ext='txt').endswith('.txt')
True
```

classmethod **from_file**(*filename*)
  Construct buffer from data in file.

> **Warning:** The buffer assumes that the file content remains the same during the lifetime of the Sage session. To communicate this to the user, the file permissions will be changed to read only.

  INPUT:

  • `filename` – string. The filename under which the data is stored.

  OUTPUT:

  String containing the buffer data.

  EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: name = sage.misc.temporary_file.tmp_filename()
sage: with open(name, 'w') as f:
....:     _ = f.write('file content')
sage: buf = OutputBuffer.from_file(name);  buf
buffer containing 12 bytes

sage: buf.filename() == name
True
sage: buf.get()
'file content'
```

get()
  Return the buffer content

  OUTPUT:

  Bytes. A string in Python 2.x.

  EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: OutputBuffer('test1234').get()
'test1234'
```

get_unicode()
  Return the buffer content as string

  OUTPUT:

  String. Unicode in Python 2.x. Raises a `UnicodeEncodeError` if the data is not valid utf-8.

  EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: OutputBuffer('test1234').get()
'test1234'
sage: OutputBuffer('test1234').get_unicode()
u'test1234'
```

save_as(*filename*)
  Save a copy of the buffer content.

  You may edit the returned file, unlike the file returned by `filename()`.

INPUT:

- `filename` – string. The file name to save under.

EXAMPLES:

```
sage: from sage.repl.rich_output.buffer import OutputBuffer
sage: buf = OutputBuffer('test')
sage: buf.filename(ext='txt')    # random output
sage: tmp = tmp_dir()
sage: filename = os.path.join(tmp, 'foo.txt')
sage: buf.save_as(filename)
sage: with open(filename, 'r') as f:
....:     f.read()
'test'
```

5.4 Basic Output Types

The Sage rich representation system requires a special container class to hold the data for each type of rich output. They all inherit from *OutputBase*, though a more typical example is *OutputPlainText*. Some output classes consist of more than one data buffer, for example jmol or certain animation formats. The output class is independent of user preferences and of the display backend.

The display backends can define derived classes to attach backend-specific display functionality to, for example how to launch a viewer. But they must not change how the output container is created. To enforce this, the Sage _rich_repr_ magic method will only ever see the output class defined here. The display manager will promote it to a backend-specific subclass if necessary prior to displaying it.

To create new types of output, you must create your own subclass of *OutputBase* and register it in *sage.repl. rich_output.output_catalog*.

> **Warning:** All rich output data in sublasses of *OutputBase* must be contained in *OutputBuffer* instances. You must never reference any files on the local file system, as there is no guarantee that the notebook server and the worker process are on the same computer. Or even share a common file system.

class sage.repl.rich_output.output_basic.**OutputAsciiArt**(*ascii_art*)

Bases: *sage.repl.rich_output.output_basic.OutputBase*

ASCII Art Output

INPUT:

- `ascii_art` – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. Ascii art rendered into a string.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputAsciiArt
sage: OutputAsciiArt(':-}')
OutputAsciiArt container
```

classmethod example()

Construct a sample ascii art output container

This static method is meant for doctests, so they can easily construct an example.

OUTPUT:

An instance of *OutputAsciiArt*.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputAsciiArt
sage: OutputAsciiArt.example()
OutputAsciiArt container
sage: OutputAsciiArt.example().ascii_art.get()
'[                        *   *   *    * ]\n[      **   **   *    *   *   *  *␣
↪    *   ]\n[ ***, * , *  , **, ** , *, * , * , *    ]'
```

print_to_stdout()
> Write the data to stdout.
>
> This is just a convenience method to help with debugging.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.output_catalog import OutputAsciiArt
> sage: ascii_art = OutputAsciiArt.example()
> sage: ascii_art.print_to_stdout()
> [                        *   *   *    * ]
> [      **   **   *    *   *   *  *    * ]
> [ ***, * , *  , **, ** , *, * , * , *    ]
> ```

class sage.repl.rich_output.output_basic.**OutputBase**
> Bases: sage.structure.sage_object.SageObject

Base class for all rich output containers.

classmethod example()
> Construct a sample instance
>
> This static method is meant for doctests, so they can easily construct an example.
>
> OUTPUT:
>
> An instance of the *OutputBase* subclass.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.output_basic import OutputBase
> sage: OutputBase.example()
> Traceback (most recent call last):
> ...
> NotImplementedError: derived classes must implement this class method
> ```

class sage.repl.rich_output.output_basic.**OutputLatex**(*latex*)
> Bases: *sage.repl.rich_output.output_basic.OutputBase*

LaTeX Output

Note: The LaTeX commands will only use a subset of LaTeX that can be displayed by MathJax.

INPUT:

- latex – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. String containing the latex equation code. Excludes the surrounding dollar signs / LaTeX equation environment. Also excludes the surrounding MathJax <html> tag.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: OutputLatex('<html><script type="math/tex; mode=display">1</script></html>')
OutputLatex container
```

display_equation()
> Return the LaTeX code for a display equation

> OUTPUT:

> String.

> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: rich_output = OutputLatex('1')
sage: rich_output.latex
buffer containing 1 bytes
sage: rich_output.latex.get()
'1'
sage: rich_output.display_equation()
'\\begin{equation}\n1\n\\end{equation}'
```

classmethod **example**()
> Construct a sample LaTeX output container

> This static method is meant for doctests, so they can easily construct an example.

> OUTPUT:

> An instance of *OutputLatex*.

> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: OutputLatex.example()
OutputLatex container
sage: OutputLatex.example().latex.get()
'\\newcommand{\\Bold}[1]{\\mathbf{#1}}\\int \\sin\\left(x\\right)\\,{d x}'
```

inline_equation()
> Return the LaTeX code for an inline equation

> OUTPUT:

> String.

> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: rich_output = OutputLatex('1')
sage: rich_output.latex
buffer containing 1 bytes
sage: rich_output.latex.get()
'1'
sage: rich_output.inline_equation()
'\\begin{math}\n1\n\\end{math}'
```

mathjax(*display=True*)
> Return the LaTeX with a surrounding MathJax HTML code.

> INPUT:

•`display` – boolean. Whether to return display (as opposed to inline) TeX.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: rich_output = OutputLatex('1')
sage: rich_output.latex
buffer containing 1 bytes
sage: rich_output.latex.get()
'1'
sage: rich_output.mathjax()
'<html><script type="math/tex; mode=display">1</script></html>'
sage: rich_output.mathjax(display=False)
'<html><script type="math/tex">1</script></html>'
```

print_to_stdout()
Write the data to stdout.

This is just a convenience method to help with debugging.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputLatex
sage: rich_output = OutputLatex.example()
sage: rich_output.print_to_stdout()
\newcommand{\Bold}[1]{\mathbf{#1}}\int \sin\left(x\right)\,{d x}
```

class sage.repl.rich_output.output_basic.**OutputPlainText**(*plain_text*)
Bases: *sage.repl.rich_output.output_basic.OutputBase*

Plain Text Output

INPUT:

•`plain_text` – *OutputBuffer*. Alternatively, a bytes (string in Python 2.x) or string (unicode in Python 2.x) can be passed directly which will then be converted into an *OutputBuffer*. The plain text output.

This should always be exactly the same as the (non-rich) output from the `_repr_` method. Every backend object must support plain text output as fallback.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputPlainText
sage: OutputPlainText('foo')
OutputPlainText container
```

classmethod example()
Construct a sample plain text output container

This static method is meant for doctests, so they can easily construct an example.

OUTPUT:

An instance of *OutputPlainText*.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputPlainText
sage: OutputPlainText.example()
OutputPlainText container
```

```
sage: OutputPlainText.example().text.get()
'Example plain text output'
```

print_to_stdout()
>   Write the data to stdout.
>
>   This is just a convenience method to help with debugging.
>
>   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: plain_text.print_to_stdout()
Example plain text output
```

class sage.repl.rich_output.output_basic.**OutputUnicodeArt**(*unicode_art*)
>   Bases: *sage.repl.rich_output.output_basic.OutputBase*
>
>   Unicode Art Output
>
>   Similar to *OutputAsciiArt* but using the entire unicode range.
>
>   INPUT:
>
>   •unicode_art – *OutputBuffer*. Alternatively, a string (unicode in Python 2.x) can be passed directly
>   which will then be converted into an *OutputBuffer*. Unicode art rendered into a string.
>
>   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputUnicodeArt
sage: OutputUnicodeArt(u':-}')
OutputUnicodeArt container
```

>   **classmethod example**()
>   >   Construct a sample unicode art output container
>   >
>   >   This static method is meant for doctests, so they can easily construct an example.
>   >
>   >   OUTPUT:
>   >
>   >   An instance of *OutputUnicodeArt*.
>   >
>   >   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputUnicodeArt
sage: OutputUnicodeArt.example()
OutputUnicodeArt container
sage: OutputUnicodeArt.example().unicode_art.get()
'\xe2\x8e\x9b-11   0   1\xe2\x8e\x9e\n\xe2\x8e\x9c  3  -1   ␣
↪0\xe2\x8e\x9f\n\xe2\x8e\x9d -1  -1   0\xe2\x8e\xa0'
```

>   **print_to_stdout**()
>   >   Write the data to stdout.
>   >
>   >   This is just a convenience method to help with debugging.
>   >
>   >   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputUnicodeArt
sage: unicode_art = OutputUnicodeArt.example()
sage: unicode_art.print_to_stdout()
⎛-11   0   1⎞
```

```
⎡  3  -1   0⎤
⎣ -1  -1   0⎦
```

5.5 Graphics Output Types

This module defines the rich output types for 2-d images, both vector and raster graphics.

class sage.repl.rich_output.output_graphics.**OutputImageDvi**(*dvi*)

Bases: *sage.repl.rich_output.output_basic.OutputBase*

DVI Image

INPUT:

- dvi – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The DVI data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageDvi
sage: OutputImageDvi.example()       # indirect doctest
OutputImageDvi container
```

classmethod example()

Construct a sample DVI output container

This static method is meant for doctests, so they can easily construct an example.

OUTPUT:

An instance of *OutputImageDvi*.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageDvi
sage: OutputImageDvi.example()
OutputImageDvi container
sage: OutputImageDvi.example().dvi
buffer containing 212 bytes
sage: 'TeX output' in OutputImageDvi.example().dvi.get()
True
```

class sage.repl.rich_output.output_graphics.**OutputImageGif**(*gif*)

Bases: *sage.repl.rich_output.output_basic.OutputBase*

GIF Image (possibly animated)

INPUT:

- gif – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The GIF image data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageGif
sage: OutputImageGif.example()    # indirect doctest
OutputImageGif container
```

classmethod **example**()
>   Construct a sample GIF output container
>
>   This static method is meant for doctests, so they can easily construct an example.
>
>   OUTPUT:
>
>   An instance of *OutputImageGif*.
>
>   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageGif
sage: OutputImageGif.example()
OutputImageGif container
sage: OutputImageGif.example().gif
buffer containing 408 bytes
sage: OutputImageGif.example().gif.get().startswith('GIF89a')
True
```

html_fragment()
>   Return a self-contained HTML fragment displaying the image
>
>   This is a workaround for the Jupyter notebook which doesn't support GIF directly.
>
>   OUTPUT:
>
>   String. HTML fragment for displaying the GIF image.
>
>   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageGif
sage: OutputImageGif.example().html_fragment()
'<img src="data:image/gif;base64,R0lGODl...zd3t/g4eLj5OVDQQA7"/>'
```

class sage.repl.rich_output.output_graphics.**OutputImageJpg**(*jpg*)
>   Bases: *sage.repl.rich_output.output_basic.OutputBase*
>
>   JPEG Image
>
>   INPUT:
>
>   •jpg – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The JPEG image data.
>
>   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageJpg
sage: OutputImageJpg.example()      # indirect doctest
OutputImageJpg container
```

classmethod **example**()
>   Construct a sample JPEG output container
>
>   This static method is meant for doctests, so they can easily construct an example.
>
>   OUTPUT:
>
>   An instance of *OutputImageJpg*.
>
>   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageJpg
sage: OutputImageJpg.example()
OutputImageJpg container
```

```
sage: OutputImageJpg.example().jpg
buffer containing 978 bytes
sage: OutputImageJpg.example().jpg.get().startswith(
↪'\xff\xd8\xff\xe0\x00\x10JFIF')
True
```

class sage.repl.rich_output.output_graphics.**OutputImagePdf**(*pdf*)

Bases: *sage.repl.rich_output.output_basic.OutputBase*

PDF Image

INPUT:

- pdf – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The PDF data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImagePdf
sage: OutputImagePdf.example()    # indirect doctest
OutputImagePdf container
```

classmethod example()

Construct a sample PDF output container

This static method is meant for doctests, so they can easily construct an example.

OUTPUT:

An instance of *OutputImagePdf*.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImagePdf
sage: OutputImagePdf.example()
OutputImagePdf container
sage: OutputImagePdf.example().pdf
buffer containing 4285 bytes
sage: OutputImagePdf.example().pdf.get().startswith('%PDF-1.4')
True
```

class sage.repl.rich_output.output_graphics.**OutputImagePng**(*png*)

Bases: *sage.repl.rich_output.output_basic.OutputBase*

PNG Image

Note: Every backend that is capable of displaying any kind of graphics is supposed to support the PNG format at least.

INPUT:

- png – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The PNG image data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImagePng
sage: OutputImagePng.example()    # indirect doctest
OutputImagePng container
```

---

classmethod **example**()
> Construct a sample PNG output container

> This static method is meant for doctests, so they can easily construct an example.

> OUTPUT:

> An instance of *OutputImagePng*.

> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImagePng
sage: OutputImagePng.example()
OutputImagePng container
sage: OutputImagePng.example().png
buffer containing 608 bytes
sage: OutputImagePng.example().png.get().startswith('\x89PNG')
True
```

class sage.repl.rich_output.output_graphics.**OutputImageSvg**(*svg*)
> Bases: *sage.repl.rich_output.output_basic.OutputBase*

SVG Image

INPUT:

- svg – *OutputBuffer*. Alternatively, a string (bytes) can be passed directly which will then be converted into an *OutputBuffer*. The SVG image data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageSvg
sage: OutputImageSvg.example()    # indirect doctest
OutputImageSvg container
```

classmethod **example**()
> Construct a sample SVG output container

> This static method is meant for doctests, so they can easily construct an example.

> OUTPUT:

> An instance of *OutputImageSvg*.

> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputImageSvg
sage: OutputImageSvg.example()
OutputImageSvg container
sage: OutputImageSvg.example().svg
buffer containing 1422 bytes
sage: '</svg>' in OutputImageSvg.example().svg.get()
True
```

5.6 Three-Dimensional Graphics Output Types

This module defines the rich output types for 3-d scenes.

class sage.repl.rich_output.output_graphics3d.**OutputSceneCanvas3d**(*canvas3d*)
> Bases: *sage.repl.rich_output.output_basic.OutputBase*

Canvas3d Scene

INPUT:

- `canvas3d` – string/bytes. The canvas3d data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneCanvas3d
sage: OutputSceneCanvas3d.example()
OutputSceneCanvas3d container
```

classmethod example()

Construct a sample Canvas3D output container

This static method is meant for doctests, so they can easily construct an example.

OUTPUT:

An instance of *OutputSceneCanvas3d*.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneCanvas3d
sage: rich_output = OutputSceneCanvas3d.example();  rich_output
OutputSceneCanvas3d container

sage: rich_output.canvas3d
buffer containing 829 bytes
sage: rich_output.canvas3d.get()
'[{"vertices":[{"x":1,"y":1,"z":1},...{"x":1,"y":-1,"z":-1}],"faces":[[0,1,2,
→3]],"color":"008000"}]'
```

class sage.repl.rich_output.output_graphics3d.**OutputSceneJmol**(*scene_zip*, *preview_png*)

Bases: *sage.repl.rich_output.output_basic.OutputBase*

JMol Scene

By our (Sage) convention, the actual scene is called SCENE inside the zip archive.

INPUT:

- `scene_zip` – string/bytes. The jmol scene (a zip archive).

- `preview_png` – string/bytes. Preview as png file.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneJmol
sage: OutputSceneJmol.example()
OutputSceneJmol container
```

classmethod example()

Construct a sample Jmol output container

This static method is meant for doctests, so they can easily construct an example.

OUTPUT:

An instance of *OutputSceneJmol*.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneJmol
sage: rich_output = OutputSceneJmol.example();  rich_output
OutputSceneJmol container

sage: rich_output.scene_zip
buffer containing 654 bytes
sage: rich_output.scene_zip.get().startswith('PK')
True

sage: rich_output.preview_png
buffer containing 608 bytes
sage: rich_output.preview_png.get().startswith('\x89PNG')
True
```

launch_script_filename()

> Return a launch script suitable to display the scene.
>
> This method saves the scene to disk and creates a launch script. The latter contains an absolute path to the scene file. The launch script is often necessary to make jmol render the 3d scene.
>
> OUTPUT:
>
> String. The file name of a suitable launch script.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneJmol
sage: rich_output = OutputSceneJmol.example();  rich_output
OutputSceneJmol container
sage: filename = rich_output.launch_script_filename();  filename
'/.../scene.spt'
sage: print(open(filename).read())
set defaultdirectory "/.../scene.spt.zip"
script SCRIPT
```

class sage.repl.rich_output.output_graphics3d.**OutputSceneThreejs**(*html*)

> Bases: *sage.repl.rich_output.output_basic.OutputBase*

Three.js Scene

INPUT:

- html – string/bytes. The Three.js HTML data.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneThreejs
sage: OutputSceneThreejs('<html></html>')
OutputSceneThreejs container
```

class sage.repl.rich_output.output_graphics3d.**OutputSceneWavefront**(*obj*, *mtl*)

> Bases: *sage.repl.rich_output.output_basic.OutputBase*

Wavefront $*.obj$ Scene

The Wavefront format consists of two files, an .obj file defining the geometry data (mesh points, normal vectors, ...) together with a .mtl file defining texture data.

INPUT:

- obj – bytes. The Wavefront obj file format describing the mesh shape.

•`mtl` – bytes. The Wavefront mtl file format describing textures.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneWavefront
sage: OutputSceneWavefront.example()
OutputSceneWavefront container
```

classmethod example()

> Construct a sample Canvas3D output container
>
> This static method is meant for doctests, so they can easily construct an example.
>
> OUTPUT:
>
> An instance of *OutputSceneCanvas3d*.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneWavefront
sage: rich_output = OutputSceneWavefront.example();  rich_output
OutputSceneWavefront container

sage: rich_output.obj
buffer containing 227 bytes
sage: rich_output.obj.get()
'mtllib scene.mtl\ng obj_1\n...\nf 1 5 6 2\nf 1 4 7 5\nf 6 5 7 8\nf 7 4 3␣
→8\nf 3 2 6 8\n'

sage: rich_output.mtl
buffer containing 80 bytes
sage: rich_output.mtl.get()
'newmtl texture177\nKa 0.2 0.2 0.5\nKd 0.4 0.4 1.0\nKs 0.0 0.0 0.0\nillum␣
→1\nNs 1\nd 1\n'
```

mtllib()

> Return the `mtllib` filename
>
> The `mtllib` line in the Wavefront file format (`*.obj`) is the name of the separate texture file.
>
> OUTPUT:
>
> String. The filename under which `mtl` is supposed to be saved.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneWavefront
sage: rich_output = OutputSceneWavefront.example()
sage: rich_output.mtllib()
'scene.mtl'
```

obj_filename()

> Return the file name of the `.obj` file
>
> This method saves the object and texture to separate files in a temporary directory and returns the object file name. This is often used to launch a 3d viewer.
>
> OUTPUT:
>
> String. The file name (absolute path) of the saved obj file.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputSceneWavefront
sage: rich_output = OutputSceneWavefront.example();  rich_output
OutputSceneWavefront container
sage: obj = rich_output.obj_filename();  obj
'/.../scene.obj'
sage: print(open(obj).read())
mtllib scene.mtl
g obj_1
...
f 3 2 6 8

sage: path = os.path.dirname(obj)
sage: mtl = os.path.join(path, 'scene.mtl');  mtl
'/.../scene.mtl'
sage: os.path.exists(mtl)
True
sage: os.path.dirname(obj) == os.path.dirname(mtl)
True
sage: print(open(mtl).read())
newmtl texture177
Ka 0.2 0.2 0.5
...
d 1
```

5.7 Video Output Types

This module defines the rich output types for video formats.

class sage.repl.rich_output.output_video.**OutputVideoAvi**(*video*, *loop=True*)

Bases: *sage.repl.rich_output.output_video.OutputVideoBase*

AVI video

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoAvi
sage: OutputVideoAvi.example()
OutputVideoAvi container
```

class sage.repl.rich_output.output_video.**OutputVideoBase**(*video*, *loop=True*)

Bases: *sage.repl.rich_output.output_basic.OutputBase*

Abstract base class for rich video output

INPUT:

- video – *OutputBuffer*. The video data.

- loop – boolean. Whether to repeat the video in an endless loop.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoOgg
sage: OutputVideoOgg.example()  # indirect doctest
OutputVideoOgg container
```

classmethod example()

Construct a sample video output container

This static method is meant for doctests, so they can easily construct an example. The method is implemented in the abstract *OutputVideoBase* class, but should get invoked on a concrete subclass for which an actual example can exist.

OUTPUT:

An instance of the class on which this method is called.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoOgg
sage: OutputVideoOgg.example()
OutputVideoOgg container
sage: OutputVideoOgg.example().video
buffer containing 5612 bytes
sage: OutputVideoOgg.example().ext
'.ogv'
sage: OutputVideoOgg.example().mimetype
'video/ogg'
```

html_fragment (*url*, *link_attrs=''*)
Construct a HTML fragment for embedding this video

INPUT:

- url – string. The URL where the data of this video can be found.

- link_attrs – string. Can be used to style the fallback link which is presented to the user if the video is not supported.

EXAMPLES:: sage: from sage.repl.rich_output.output_catalog import OutputVideoOgg sage: print(OutputVideoOgg.example().html_fragment ....:   ('foo', 'class="bar"').replace('><','>n<')) <video autoplay="autoplay" controls="controls" loop="loop"> <source src="foo" type="video/ogg" /> <p> Download video/ogg video </p> </video>

class sage.repl.rich_output.output_video.**OutputVideoFlash** (*video*, *loop=True*)
Bases: *sage.repl.rich_output.output_video.OutputVideoBase*

Flash video

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoFlash
sage: OutputVideoFlash.example()
OutputVideoFlash container
```

class sage.repl.rich_output.output_video.**OutputVideoMatroska** (*video*, *loop=True*)
Bases: *sage.repl.rich_output.output_video.OutputVideoBase*

Matroska Video

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoMatroska
sage: OutputVideoMatroska.example()
OutputVideoMatroska container
```

class sage.repl.rich_output.output_video.**OutputVideoMp4** (*video*, *loop=True*)
Bases: *sage.repl.rich_output.output_video.OutputVideoBase*

MPEG 4 video

EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoMp4
sage: OutputVideoMp4.example()
OutputVideoMp4 container
```

class sage.repl.rich_output.output_video.**OutputVideoOgg**(*video*, *loop=True*)

   Bases: *sage.repl.rich_output.output_video.OutputVideoBase*

   Ogg video, Ogg Theora in particular

   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoOgg
sage: OutputVideoOgg.example()
OutputVideoOgg container
```

class sage.repl.rich_output.output_video.**OutputVideoQuicktime**(*video*, *loop=True*)

   Bases: *sage.repl.rich_output.output_video.OutputVideoBase*

   Quicktime video

   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoQuicktime
sage: OutputVideoQuicktime.example()
OutputVideoQuicktime container
```

class sage.repl.rich_output.output_video.**OutputVideoWebM**(*video*, *loop=True*)

   Bases: *sage.repl.rich_output.output_video.OutputVideoBase*

   WebM video

   The video can be encoded using VP8, VP9 or an even more recent codec.

   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoWebM
sage: OutputVideoWebM.example()
OutputVideoWebM container
```

class sage.repl.rich_output.output_video.**OutputVideoWmv**(*video*, *loop=True*)

   Bases: *sage.repl.rich_output.output_video.OutputVideoBase*

   Windows Media Video

   EXAMPLES:

```
sage: from sage.repl.rich_output.output_catalog import OutputVideoWmv
sage: OutputVideoWmv.example()
OutputVideoWmv container
```

5.8 Catalog of all available output container types.

If you define another output type then you must add it to the imports here.

5.9 Base Class for Backends

The display backends are the commandline, the SageNB notebook, the IPython notebook, the Emacs sage mode, the Sage doctester, ... All of these have different capabilities for what they can display.

To implement a new display backend, you need to subclass *BackendBase*. All backend-specific handlig of rich output should be in *displayhook()* and *display_immediately()*. See *BackendSimple* for an absolutely minimal example of a functioning backend.

You declare the types of rich output that your backend can handle in *supported_output()*. There are two ways to then display specific output types in your own backend.

- Directly use one of the existing output containers listed in *sage.repl.rich_output.output_catalog*. That is, test for the rich output type in *display_immediately()* and handle it.

- Subclass the rich output container to attach your backend-specific functionality. Then *display_immediately()* will receive instances of your subclass. See *BackendTest* for an example of how this is done.

You can also mix both ways of implementing different rich output types.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: backend = BackendSimple()
sage: plain_text = backend.plain_text_formatter(list(range(10)));  plain_text
OutputPlainText container
sage: backend.displayhook(plain_text, plain_text)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

class sage.repl.rich_output.backend_base.**BackendBase**
    Bases: sage.structure.sage_object.SageObject

    ascii_art_formatter(*obj*, ***kwds*)
        Hook to override how ascii art is being formatted.

        INPUT:

            •obj – anything.

            •**kwds – optional keyword arguments to control the formatting. Supported are:

                –concatenate – boolean (default: False). If True, the argument obj must be iterable and its entries will be concatenated. There is a single whitespace between entries.

        OUTPUT:

        Instance of *OutputAsciiArt* containing the ascii art string representation of the object.

        EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: out = backend.ascii_art_formatter(list(range(30)))
sage: out
OutputAsciiArt container
sage: out.ascii_art
buffer containing 114 bytes
sage: print(out.ascii_art.get())
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,␣
→21,
```

```
 22, 23, 24, 25, 26, 27, 28, 29 ]
sage: backend.ascii_art_formatter([1,2,3], concatenate=False).ascii_art.get()
'[ 1, 2, 3 ]'
sage: backend.ascii_art_formatter([1,2,3], concatenate=True ).ascii_art.get()
'1 2 3'
```

default_preferences()

> Return the backend's display preferences
>
> Override this method to change the default preferences when using your backend.
>
> OUTPUT:
>
> Instance of *DisplayPreferences*.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.default_preferences()
Display preferences:
* graphics is not specified
* supplemental_plot is not specified
* text is not specified
```

display_immediately(*plain_text*, *rich_output*)

> Show output without going back to the command line prompt.
>
> This method is similar to the rich output *displayhook()*, except that it can be invoked at any time. Typically, it ends up being called by sage.plot.graphics.Graphics.show().
>
> Derived classes must implement this method.
>
> INPUT:
>
> Same as *displayhook()*.
>
> OUTPUT:
>
> This method may return something so you can implement *displayhook()* by calling this method. However, when called by the display manager any potential return value is discarded: There is no way to return anything without returning to the command prompt.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.display_immediately(plain_text, plain_text)
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

displayhook(*plain_text*, *rich_output*)

> Backend implementation of the displayhook
>
> The value of the last statement on a REPL input line or notebook cell are usually handed to the Python displayhook and shown on screen. By overriding this method you define how your backend handles output. The difference to the usual displayhook is that Sage already converted the value to the most suitable rich output container.

Derived classes must implement this method.

INPUT:

- • plain_text – instance of *OutputPlainText*. The plain text version of the output.

- • rich_output – instance of an output container class (subclass of *OutputBase*). Guaranteed to be one of the output containers returned from *supported_output()*, possibly the same as plain_text.

OUTPUT:

This method may return something, which is then returned from the display manager's *displayhook()* method.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.displayhook(plain_text, plain_text)
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

get_display_manager()
Return the display manager singleton

This is a convenience method to access the display manager singleton.

OUTPUT:

The unique *DisplayManager* instance.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.get_display_manager()
The Sage display manager using the doctest backend
```

install(********kwds*)
Hook that will be called once before the backend is used for the first time.

The default implementation does nothing.

INPUT:

- • kwds – optional keyword arguments that are passed through by the *switch_backend()* method.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.install()
```

is_in_terminal()
Test whether the UI is meant to run in a terminal

See *sage.repl.rich_output.display_manager.DisplayManager.is_in_terminal()* for details.

OUTPUT:

Defaults to `False`.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.is_in_terminal()
False
```

latex_formatter(*obj*, ***kwds*)

Hook to override how Latex is being formatted.

INPUT:

- `obj` – anything.

- `**kwds` – optional keyword arguments to control the formatting. Supported are:

    - `concatenate` – boolean (default: `False`). If `True`, the argument `obj` must be iterable and its entries will be concatenated. There is a single whitespace between entries.

OUTPUT:

Instance of *OutputLatex* containing the latex string representation of the object.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: out = backend.latex_formatter(1/2)
sage: out
OutputLatex container
sage: out.latex
buffer containing 45 bytes
sage: out.latex.get()
'\\newcommand{\\Bold}[1]{\\mathbf{#1}}\\frac{1}{2}'
sage: out.mathjax()
'<html><script type="math/tex; mode=display">\\newcommand{\\Bold}[1]{\\mathbf{
↪#1}}\\frac{1}{2}</script></html>'

sage: out = backend.latex_formatter([1/2, x, 3/4, ZZ], concatenate=False)
sage: out.latex.get()
'\\newcommand{\\Bold}[1]{\\mathbf{#1}}\\left[\\frac{1}{2}, x, \\frac{3}{4},␣
↪\\Bold{Z}\\right]'
sage: out = backend.latex_formatter([1/2, x, 3/4, ZZ], concatenate=True)
sage: out.latex.get()
'\\newcommand{\\Bold}[1]{\\mathbf{#1}}\\frac{1}{2} x \\frac{3}{4} \\Bold{Z}'
```

max_width()

Return the number of characters that fit into one output line

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.max_width()
79
```

newline()

Return the newline string.

OUTPUT:

String for starting a new line of output.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.newline()
'\n'
```

plain_text_formatter(*obj*, ***kwds*)

Hook to override how plain text is being formatted.

If the object does not have a _rich_repr_ method, or if it does not return a rich output object (*OutputBase*), then this method is used to generate plain text output.

INPUT:

- obj – anything.

- **kwds – optional keyword arguments to control the formatting. Supported are:

    – concatenate – boolean (default: False). If True, the argument obj must be iterable and its entries will be concatenated. There is a single whitespace between entries.

OUTPUT:

Instance of *OutputPlainText* containing the string representation of the object.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: out = backend.plain_text_formatter(list(range(30)))
sage: out
OutputPlainText container
sage: out.text
buffer containing 139 bytes
sage: out.text.get()
'[0,\n 1,\n 2,\n 3,\n 4,\n 5,\n 6,\n 7,\n 8,\n 9,\n
10,\n 11,\n 12,\n 13,\n 14,\n 15,\n 16,\n 17,\n 18,\n
19,\n 20,\n 21,\n 22,\n 23,\n 24,\n 25,\n 26,\n 27,\n
28,\n 29]'

sage: out = backend.plain_text_formatter(list(range(20)), concatenate=True)
sage: out.text.get()
'0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19'
```

set_underscore_variable(*obj*)

Set the _ builtin variable.

By default, this sets the special _ variable. Backends that organize the history differently (e.g. IPython) can override this method.

INPUT:

- obj – result of the most recent evaluation.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.set_underscore_variable(123)
sage: _
123

sage: 'foo'
'foo'
sage: _        # indirect doctest
'foo'
```

supported_output()
> Return the outputs that are supported by the backend.

> Subclasses must implement this method.

> OUTPUT:

> Iterable of output container classes, that is, subclass of *OutputBase*). May be a list/tuple/set/frozenset. The order is ignored. Only used internally by the display manager.

> You may return backend-specific subclasses of existing output containers. This allows you to attach backend-specifc functionality to the output container.

> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: backend.supported_output()
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

unicode_art_formatter(*obj*, ***kwds*)
> Hook to override how unicode art is being formatted.

> INPUT:

> > •obj – anything.

> > •**kwds – optional keyword arguments to control the formatting. Supported are:

> > > –concatenate – boolean (default: False). If True, the argument obj must be iterable and its entries will be concatenated. There is a single whitespace between entries.

> OUTPUT:

> Instance of *OutputUnicodeArt* containing the unicode art string representation of the object.

> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendBase
sage: backend = BackendBase()
sage: out = backend.unicode_art_formatter(list(range(30)))
sage: out
OutputUnicodeArt container
sage: out.unicode_art
buffer containing 114 bytes
sage: print(out.unicode_art.get())
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,␣
↪21,
```

```
22, 23, 24, 25, 26, 27, 28, 29 ]

sage: backend.unicode_art_formatter([1,2,3], concatenate=False).unicode_art.
→get()
'[ 1, 2, 3 ]'
sage: backend.unicode_art_formatter([1,2,3], concatenate=True ).unicode_art.
→get()
'1 2 3'
```

uninstall()

> Hook that will be called once right before the backend is removed.
>
> The default implementation does nothing.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.backend_base import BackendBase
> sage: backend = BackendBase()
> sage: backend.uninstall()
> ```

class sage.repl.rich_output.backend_base.**BackendSimple**

> Bases: *sage.repl.rich_output.backend_base.BackendBase*
>
> Simple Backend
>
> This backend only supports plain text.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.backend_base import BackendSimple
> sage: BackendSimple()
> simple
> ```

display_immediately(*plain_text*, *rich_output*)

> Show output without going back to the command line prompt.
>
> INPUT:
>
> Same as *displayhook()*.
>
> OUTPUT:
>
> This backend returns nothing, it just prints to stdout.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.output_basic import OutputPlainText
> sage: plain_text = OutputPlainText.example()
> sage: from sage.repl.rich_output.backend_base import BackendSimple
> sage: backend = BackendSimple()
> sage: backend.display_immediately(plain_text, plain_text)
> Example plain text output
> ```

supported_output()

> Return the outputs that are supported by the backend.
>
> OUTPUT:
>
> Iterable of output container classes, that is, subclass of *OutputBase*). This backend only supports the plain text output container.
>
> EXAMPLES:

```
sage: from sage.repl.rich_output.backend_base import BackendSimple
sage: backend = BackendSimple()
sage: backend.supported_output()
{<class 'sage.repl.rich_output.output_basic.OutputPlainText'>}
```

5.10 Test Backend

This backend is only for doctesting purposes.

EXAMPLES:

We switch to the test backend for the remainder of this file:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: from sage.repl.rich_output.backend_test import BackendTest, TestObject
sage: doctest_backend = dm.switch_backend(BackendTest())
sage: dm
The Sage display manager using the test backend

sage: dm._output_promotions
{<class 'sage.repl.rich_output.output_basic.OutputPlainText'>:
 <class 'sage.repl.rich_output.backend_test.TestOutputPlainText'>}
sage: dm.displayhook(1/2)
1/2 [TestOutputPlainText]
TestOutputPlainText container

sage: test = TestObject()
sage: test
called the _repr_ method
sage: dm.displayhook(test)
called the _rich_repr_ method [TestOutputPlainText]
TestOutputPlainText container
```

class sage.repl.rich_output.backend_test.**BackendTest**
> Bases: *sage.repl.rich_output.backend_base.BackendBase*

> **display_immediately**(*plain_text*, *rich_output*)
> > Show output without going back to the command line prompt.

> > INPUT:

> > Same as displayhook().

> > OUTPUT:

> > This method returns the rich output for doctesting convenience. The actual display framework ignores the return value.

> > EXAMPLES:

> > ```
> > sage: from sage.repl.rich_output.output_basic import OutputPlainText
> > sage: plain_text = OutputPlainText.example()
> > sage: from sage.repl.rich_output.backend_test import BackendTest
> > sage: backend = BackendTest()
> > sage: backend.display_immediately(plain_text, plain_text)
> > Example plain text output
> > OutputPlainText container
> > ```

supported_output()
> Return the outputs that are supported by the backend.
>
> OUTPUT:
>
> Iterable of output container classes. Only the `TestOutputPlainText` output container is supported by the test backend.
>
> EXAMPLES:
>
> ```
> sage: display_manager = sage.repl.rich_output.get_display_manager()
> sage: backend = display_manager._backend
> sage: backend.supported_output()
> set([<class 'sage.repl.rich_output.backend_test.TestOutputPlainText'>])
> ```
>
> The output of this method is used by the display manager to set up the actual supported outputs. Compare:
>
> ```
> sage: display_manager.supported_output()
> frozenset([<class 'sage.repl.rich_output.output_basic.OutputPlainText'>])
> ```

class sage.repl.rich_output.backend_test.**TestObject**
> Bases: sage.structure.sage_object.SageObject
>
> Test object with both _repr_() and _rich_repr_()

class sage.repl.rich_output.backend_test.**TestOutputPlainText**(*args*, ***kwds*)
> Bases: *sage.repl.rich_output.output_basic.OutputPlainText*
>
> Backend-specific subclass of the plain text output container.
>
> Backends must not influence how the display system constucts output containers, they can only control how the output container is displayed. In particular, we cannot override the constructor (only the *OutputPlainText* constructor is used).
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output.backend_test import TestOutputPlainText
> sage: TestOutputPlainText()
> Traceback (most recent call last):
> ...
> AssertionError: cannot override constructor
> ```

print_to_stdout()
> Write the data to stdout.
>
> This is just a convenience method to help with debugging.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.rich_output import get_display_manager
> sage: dm = get_display_manager()
> sage: test_output = dm.displayhook(123)
> 123 [TestOutputPlainText]
> sage: type(test_output)
> <class 'sage.repl.rich_output.backend_test.TestOutputPlainText'>
> sage: test_output.print_to_stdout()
> 123 [TestOutputPlainText]
> ```

5.11 The backend used for doctests

This backend is active during doctests. It should mimic the behavior of the IPython command line as close as possible. Without actually launching image viewers, of course.

EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: get_display_manager()
The Sage display manager using the doctest backend
```

class sage.repl.rich_output.backend_doctest.**BackendDoctest**

Bases: *sage.repl.rich_output.backend_base.BackendBase*

default_preferences()

Return the backend's display preferences

Matches the IPython command line display preferences to keep the differences between that and the doctests to a minimum.

OUTPUT:

Instance of *DisplayPreferences*.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.default_preferences()
Display preferences:
* graphics is not specified
* supplemental_plot = never
* text is not specified
```

display_immediately(*plain_text*, *rich_output*)

Display object immediately

INPUT:

Same as *displayhook()*.

EXAMPLES:

The following example does not call the displayhook. More precisely, the show() method returns None which is ignored by the displayhook. When running the example on a Sage display backend capable of displaying graphics outside of the displayhook, the plot is still shown. Nothing is shown during doctests:

```
sage: plt = plot(sin)
sage: plt
Graphics object consisting of 1 graphics primitive
sage: plt.show()

sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.display_immediately(plt)   # indirect doctest
```

displayhook(*plain_text*, *rich_output*)

Display object from displayhook

INPUT:

- plain_text – instance of *OutputPlainText*. The plain text version of the output.

- rich_output – instance of an output container class (subclass of *OutputBase*). Guaranteed to be one of the output containers returned from *supported_output()*, possibly the same as plain_text.

EXAMPLES:

This ends up calling the displayhook:

```
sage: plt = plot(sin)
sage: plt
Graphics object consisting of 1 graphics primitive
sage: plt.show()

sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: dm.displayhook(plt)        # indirect doctest
Graphics object consisting of 1 graphics primitive
```

install(***kwds*)

Switch to the doctest backend.

This method is being called from within *switch_backend()*. You should never call it by hand.

INPUT:

None of the optional keyword arguments are used in the doctest backend.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_doctest import BackendDoctest
sage: backend = BackendDoctest()
sage: backend.install()
sage: backend.uninstall()
```

supported_output()

Return the supported output types

OUTPUT:

Set of subclasses of *OutputBase*, the supported output container types.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_doctest import BackendDoctest
sage: from sage.repl.rich_output.output_catalog import *
sage: backend = BackendDoctest()
sage: OutputPlainText in backend.supported_output()
True
sage: OutputSceneJmol in backend.supported_output()
True
```

uninstall()

Switch away from the doctest backend

This method is being called from within *switch_backend()*. You should never call it by hand.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_doctest import BackendDoctest
sage: backend = BackendDoctest()
```

5.11. The backend used for doctests   **81**

```
sage: backend.install()
sage: backend.uninstall()
```

validate(*rich_output*)

    Perform checks on `rich_output`

    INPUT:

        •`rich_output` – instance of a subclass of *OutputBase*.

    OUTPUT:

    An assertion is triggered if `rich_output` is invalid.

    EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: invalid = dm.types.OutputImagePng('invalid')
sage: backend = dm._backend;  backend
doctest
sage: backend.validate(invalid)
Traceback (most recent call last):
...
AssertionError
sage: backend.validate(dm.types.OutputPlainText.example())
sage: backend.validate(dm.types.OutputAsciiArt.example())
sage: backend.validate(dm.types.OutputLatex.example())
sage: backend.validate(dm.types.OutputImagePng.example())
sage: backend.validate(dm.types.OutputImageGif.example())
sage: backend.validate(dm.types.OutputImageJpg.example())
sage: backend.validate(dm.types.OutputImageSvg.example())
sage: backend.validate(dm.types.OutputImagePdf.example())
sage: backend.validate(dm.types.OutputImageDvi.example())
sage: backend.validate(dm.types.OutputSceneJmol.example())
sage: backend.validate(dm.types.OutputSceneWavefront.example())
sage: backend.validate(dm.types.OutputSceneCanvas3d.example())
sage: backend.validate(dm.types.OutputVideoOgg.example())
sage: backend.validate(dm.types.OutputVideoWebM.example())
sage: backend.validate(dm.types.OutputVideoMp4.example())
sage: backend.validate(dm.types.OutputVideoFlash.example())
sage: backend.validate(dm.types.OutputVideoMatroska.example())
sage: backend.validate(dm.types.OutputVideoAvi.example())
sage: backend.validate(dm.types.OutputVideoWmv.example())
sage: backend.validate(dm.types.OutputVideoQuicktime.example())
```

5.12 IPython Backend for the Sage Rich Output System

This module defines the IPython backends for `sage.repl.rich_output`.

class `sage.repl.rich_output.backend_ipython.`**BackendIPython**

    Bases: *sage.repl.rich_output.backend_base.BackendBase*

    Common base for the IPython UIs

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPython
sage: BackendIPython()._repr_()
Traceback (most recent call last):
...
NotImplementedError: derived classes must implement this method
```

display_immediately(*plain_text*, *rich_output*)

Show output immediately.

This method is similar to the rich output `displayhook()`, except that it can be invoked at any time.

INPUT:

Same as `displayhook()`.

OUTPUT:

This method does not return anything.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: backend = BackendIPythonNotebook()
sage: _ = backend.display_immediately(plain_text, plain_text)
Example plain text output
```

install(***kwds*)

Switch the Sage rich output to the IPython backend

INPUT:

- `shell` – keyword argument. The IPython shell.

No tests since switching away from the doctest rich output backend will break the doctests.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: from sage.repl.rich_output.backend_ipython import BackendIPython
sage: backend = BackendIPython()
sage: shell = get_test_shell();
sage: backend.install(shell=shell)
sage: shell.run_cell('1+1')
2
```

set_underscore_variable(*obj*)

Set the _ builtin variable.

Since IPython handles the history itself, this does nothing.

INPUT:

- `obj` – anything.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: from sage.repl.rich_output.backend_ipython import BackendIPython
sage: backend = BackendIPython()
sage: backend.set_underscore_variable(123)
sage: _
0
```

---

class sage.repl.rich_output.backend_ipython.**BackendIPythonCommandline**

    Bases: *sage.repl.rich_output.backend_ipython.BackendIPython*

    Backend for the IPython Command Line

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonCommandline
sage: BackendIPythonCommandline()
IPython command line
```

    default_preferences()

        Return the backend's display preferences

        The default for the commandline is to not plot graphs since the launching of an external viewer is considered too disruptive.

        OUTPUT:

        Instance of *DisplayPreferences*.

        EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
 ↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.default_preferences()
Display preferences:
* graphics is not specified
* supplemental_plot = never
* text is not specified
```

    display_immediately(*plain_text*, *rich_output*)

        Show output without going back to the command line prompt.

        This method is similar to the rich output *displayhook()*, except that it can be invoked at any time. On the Sage command line it launches viewers just like *displayhook()*.

        INPUT:

        Same as *displayhook()*.

        OUTPUT:

        This method does not return anything.

        EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_ipython import␣
 ↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.display_immediately(plain_text, plain_text)
Example plain text output
```

    displayhook(*plain_text*, *rich_output*)

        Backend implementation of the displayhook

        INPUT:

- plain_text – instance of *OutputPlainText*. The plain text version of the output.

- rich_output – instance of an output container class (subclass of *OutputBase*). Guaranteed to be one of the output containers returned from *supported_output()*, possibly the same as plain_text.

OUTPUT:

The IPython commandline display hook returns the IPython display data, a pair of dictionaries. The first dictionary contains mime types as keys and the respective output as value. The second dictionary is metadata.

EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_ipython import
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.displayhook(plain_text, plain_text)
({u'text/plain': u'Example plain text output'}, {})
```

is_in_terminal()
    Test whether the UI is meant to run in a terminal

    See   *sage.repl.rich_output.display_manager.DisplayManager.
    is_in_terminal()* for details.

    OUTPUT:

    True for the IPython commandline.

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.is_in_terminal()
True
```

launch_jmol(*output_jmol*, *plain_text*)
    Launch the stand-alone jmol viewer

    INPUT:

    - output_jmol – *OutputSceneJmol*. The scene to launch Jmol with.

    - plain_text – string. The plain text representation.

    OUTPUT:

    String. Human-readable message indicating that the viewer was launched.

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: backend.launch_jmol(OutputSceneJmol.example(), 'Graphics3d object')
'Launched jmol viewer for Graphics3d object'
```

launch_viewer(*image_file*, *plain_text*)
    Launch external viewer for the graphics file.

    INPUT:

    - image_file – string. File name of the image file.

    - plain_text – string. The plain text representation of the image file.

    OUTPUT:

    String. Human-readable message indicating whether the viewer was launched successfully.

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.launch_viewer('/path/to/foo.bar', 'Graphics object')
'Launched bar viewer for Graphics object'
```

supported_output()
    Return the outputs that are supported by the IPython commandline backend.

    OUTPUT:

    Iterable of output container classes, that is, subclass of *OutputBase*). The order is ignored.

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: supp = backend.supported_output();  supp      # random output
set([<class 'sage.repl.rich_output.output_graphics.OutputImageGif'>,
     ...,
     <class 'sage.repl.rich_output.output_graphics.OutputImagePng'>])
sage: from sage.repl.rich_output.output_basic import OutputLatex
sage: OutputLatex in supp
True
```

threejs_offline_scripts()
    Three.js scripts for the IPython command line

    OUTPUT:

    String containing script tags

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import␣
↪BackendIPythonCommandline
sage: backend = BackendIPythonCommandline()
sage: backend.threejs_offline_scripts()
'...<script ...</script>...'
```

class sage.repl.rich_output.backend_ipython.**BackendIPythonNotebook**
    Bases: *sage.repl.rich_output.backend_ipython.BackendIPython*

    Backend for the IPython Notebook

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: BackendIPythonNotebook()
IPython notebook
```

displayhook(*plain_text*, *rich_output*)

    Backend implementation of the displayhook

    INPUT:

        • `plain_text` – instance of *OutputPlainText*. The plain text version of the output.

        • `rich_output` – instance of an output container class (subclass of *OutputBase*). Guaranteed to be one of the output containers returned from *supported_output()*, possibly the same as `plain_text`.

    OUTPUT:

    The IPython notebook display hook returns the IPython display data, a pair of dictionaries. The first dictionary contains mime types as keys and the respective output as value. The second dictionary is metadata.

    EXAMPLES:

```
sage: from sage.repl.rich_output.output_basic import OutputPlainText
sage: plain_text = OutputPlainText.example()
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: backend = BackendIPythonNotebook()
sage: backend.displayhook(plain_text, plain_text)
({u'text/plain': u'Example plain text output'}, {})
```

supported_output()

    Return the outputs that are supported by the IPython notebook backend.

    OUTPUT:

    Iterable of output container classes, that is, subclass of *OutputBase*). The order is ignored.

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: backend = BackendIPythonNotebook()
sage: supp = backend.supported_output();  supp      # random output
set([<class 'sage.repl.rich_output.output_graphics.OutputPlainText'>,
     ...,
     <class 'sage.repl.rich_output.output_graphics.OutputImagePdf'>])
sage: from sage.repl.rich_output.output_basic import OutputLatex
sage: OutputLatex in supp
True
sage: from sage.repl.rich_output.output_graphics import OutputImageGif
sage: OutputImageGif in supp
True
```

threejs_offline_scripts()

    Three.js scripts for the IPython notebook

    OUTPUT:

    String containing script tags

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_ipython import BackendIPythonNotebook
sage: backend = BackendIPythonNotebook()
sage: backend.threejs_offline_scripts()
'...<script src="/nbextensions/threejs/three.min...<\/script>...'
```

5.13 SageNB Backend for the Sage Rich Output System

This module defines the IPython backends for `sage.repl.rich_output`.

EXAMPLES:

Install the SageNB displayhook while doctesting for the rest of this file. This somewhat odd incantation is how SageNB installed its displayhook before:

```
sage: from sage.misc.displayhook import DisplayHook
sage: import sys
sage: sys.displayhook = DisplayHook()

sage: from sage.repl.rich_output import get_display_manager
sage: get_display_manager()
The Sage display manager using the SageNB backend
```

We also enable the SageNB magic global variable:

```
sage: import sage.plot.plot
sage: sage.plot.plot.EMBEDDED_MODE = True
```

And switch to a temporary directory so our current directory will not get cluttered with temporary files:

```
sage: os.chdir(tmp_dir())
```

The SageNB notebook is based on saving data files with predictable filenames:

```
sage: os.path.exists('sage0.png')
False
sage: Graphics()
sage: os.path.exists('sage0.png')
True
sage: os.remove('sage0.png')
```

Tables are typeset as html in SageNB:

```
sage: table([1, 2, 3])
<html><div class="notruncate">
<table  class="table_form">
<tbody>
<tr class ="row-a">
<td><script type="math/tex">1</script></td>
<td><script type="math/tex">2</script></td>
<td><script type="math/tex">3</script></td>
</tr>
</tbody>
</table>
</div></html>
```

class sage.repl.rich_output.backend_sagenb.**BackendSageNB**

    Bases: *sage.repl.rich_output.backend_base.BackendBase*

    display_immediately(*plain_text*, *rich_output*)

        Show output without waiting for the prompt.

        INPUT:

- plain_text – instance of *OutputPlainText*. The plain text version of the output.

- rich_output – instance of an output container class (subclass of *OutputBase*). Guaranteed to be one of the output containers returned from *supported_output()*, possibly the same as plain_text.

        OUTPUT:

        This method does not return anything.

        EXAMPLES:

```
sage: import sage.repl.rich_output.output_catalog as catalog
sage: plain_text = catalog.OutputPlainText.example()
sage: from sage.repl.rich_output.backend_sagenb import BackendSageNB
sage: backend = BackendSageNB()
sage: backend.display_immediately(plain_text, plain_text)
Example plain text output
sage: latex = catalog.OutputLatex.example()
sage: backend.display_immediately(plain_text, latex)
<html><script type="math/tex; mode=display">\newcommand{\Bold}[1]{\mathbf{#1}}
↪\int \sin\left(x\right)\,{d x}</script></html>
```

    embed_image(*output_buffer*, *file_ext*)

        Embed Image in the SageNB worksheet

        SageNB scans per-cell directories for image files, so all we have to do here is to save the image at the right place.

        INPUT:

- output_buffer – Buffer. A buffer holding the image data.

- file_ext – string. The file extension to use for saving the image.

        OUTPUT:

        Nothing is returned. The image file is saved in the appropriate place for SageNB.

        EXAMPLES:

```
sage: from sage.repl.rich_output import get_display_manager
sage: dm = get_display_manager()
sage: rich_output = dm.types.OutputImagePng.example()
sage: os.path.exists('sage0.png')
False
sage: dm._backend.embed_image(rich_output.png, '.png')
sage: os.path.exists('sage0.png')
True
```

    embed_video(*video_output*)

    supported_output()

        Return the outputs that are supported by the SageNB backend.

        OUTPUT:

Iterable of output container classes, that is, subclass of *OutputBase*). The order is ignored.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_sagenb import BackendSageNB
sage: backend = BackendSageNB()
sage: supp = backend.supported_output();  supp      # random output
set([<class 'sage.repl.rich_output.output_graphics.OutputPlainText'>,
     ...,
     <class 'sage.repl.rich_output.output_graphics.OutputCanvas3d'>])
sage: from sage.repl.rich_output.output_basic import OutputLatex
sage: OutputLatex in supp
True
```

threejs_offline_scripts()
    Three.js scripts for the Sage notebook

    OUTPUT:

    String containing script tags

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_sagenb import BackendSageNB
sage: backend = BackendSageNB()
sage: backend.threejs_offline_scripts()
'...<script ...</script>...'
```

class sage.repl.rich_output.backend_sagenb.**SageNbOutputSceneJmol**(*scene_zip*, *preview_png*)
    Bases: *sage.repl.rich_output.output_graphics3d.OutputSceneJmol*

    Adapt Jmol rich output container for SageNB.

    For legacy reasons, SageNB expects Jmol files saved under strange names. This class takes care of that.

    EXAMPLES:

```
sage: from sage.repl.rich_output.backend_sagenb import SageNbOutputSceneJmol
sage: SageNbOutputSceneJmol.example()
SageNbOutputSceneJmol container
```

embed()
    Save all files necessary to embed jmol

    EXAMPLES:

    Switch to a new empty temporary directory:

```
sage: os.chdir(tmp_dir())
```

```
sage: from sage.repl.rich_output.backend_sagenb import SageNbOutputSceneJmol
sage: j = SageNbOutputSceneJmol.example()
sage: j.embed()
sage: sorted(os.listdir('.'))
['.jmol_images', 'sage0-size32-....jmol.zip', 'sage0-size32.jmol']
sage: sorted(os.listdir('.jmol_images'))
['sage0-size32.jmol.png']
```

preview_filename()
    Return the filename for the png preview

OUTPUT:

String.

EXAMPLES:

```
sage: from sage.repl.rich_output.backend_sagenb import SageNbOutputSceneJmol
sage: j = SageNbOutputSceneJmol.example()
sage: j.preview_filename()
'././.jmol_images/sage0-size32.jmol.png'
```

sagenb_launch_script_filename()
　　Return the launch script filename used by SageNB

　　OUTPUT:

　　String.

　　EXAMPLES:

```
sage: from sage.repl.rich_output.backend_sagenb import SageNbOutputSceneJmol
sage: j = SageNbOutputSceneJmol.example()
sage: j.sagenb_launch_script_filename()
'sage0-size32.jmol'
```

save_launch_script()
　　Save the Jmol launch script

　　See *sagenb_launch_script_filename()*.

　　EXAMPLES:

```
sage: from sage.repl.rich_output.backend_sagenb import SageNbOutputSceneJmol
sage: j = SageNbOutputSceneJmol.example()
sage: os.path.exists('sage0-size32.jmol')
False
sage: j.save_launch_script()
sage: os.path.exists('sage0-size32.jmol')
True
```

save_preview()
　　Save the preview PNG image

　　See *preview_filename()*.

　　EXAMPLES:

```
sage: from sage.repl.rich_output.backend_sagenb import SageNbOutputSceneJmol
sage: j = SageNbOutputSceneJmol.example()
sage: import shutil
sage: shutil.rmtree('.jmol_images', ignore_errors=True)
sage: j.save_preview()
sage: os.listdir('.jmol_images')
['sage1-size32.jmol.png']
```

scene_zip_filename()
　　Return the filename for the scene zip archive

　　OUTPUT:

　　String.

　　EXAMPLES:

```
sage: from sage.repl.rich_output.backend_sagenb import SageNbOutputSceneJmol
sage: j = SageNbOutputSceneJmol.example()
sage: j.scene_zip_filename()
'sage0-size32-....jmol.zip'
```

sage.repl.rich_output.backend_sagenb.**world_readable**(*filename*)

All SageNB temporary files must be world-readable.

Discussion of this design choice can be found at trac ticket #17743.

EXAMPLES:

```
sage: import os, stat
sage: f = tmp_filename()
```

At least on a sane system the temporary files are only readable by the user, but not by others in the group or total strangers:

```
sage: mode = os.stat(f).st_mode
sage: bool(mode & stat.S_IRUSR), bool(mode & stat.S_IRGRP), bool(mode & stat.S_
→IROTH)    # random output
(True, False, False)
```

This function disables that protection:

```
sage: from sage.repl.rich_output.backend_sagenb import world_readable
sage: world_readable(f)
sage: mode = os.stat(f).st_mode
sage: bool(mode & stat.S_IRUSR), bool(mode & stat.S_IRGRP), bool(mode & stat.S_
→IROTH)
(True, True, True)
```

MISCELLANEOUS

6.1 Extra Readline Commands

> **Warning:** The feature described here is no longer available in Sage, as IPython upon which Sage's command line interface is based adopted prompt_toolkit as a replacement of readline as of IPython version 5.0

The following extra readline commands are available in Sage:

- `operate-and-get-next`
- `history-search-backward-and-save`
- `history-search-forward-and-save`

The `operate-and-get-next` command accepts the input line and fetches the next line from the history. This is the same command with the same name in the Bash shell.

The `history-search-backward-and-save` command searches backward in the history for the string of characters from the start of the input line to the current cursor position, and fetches the first line found. If the cursor is at the start of the line, the previous line is fetched. The position of the fetched line is saved internally, and the next search begins at the saved position.

The `history-search-forward-and-save` command behaves similarly but forward.

The previous two commands is best used in tandem to fetch a block of lines from the history, by searching backward the first line of the block and then issuing the forward command as many times as needed. They are intended to replace the `history-search-backward` command and the `history-search-forward` command provided by the GNU readline library used in Sage.

To bind these commands with keys, insert the relevant lines into the IPython configuration file `$DOT_SAGE/ipython-*/profile_default/ipython_config.py`. Note that `$DOT_SAGE` is `$HOME/.sage` by default. For example,

```
c = get_config()

c.InteractiveShell.readline_parse_and_bind = [
    '"\C-o": operate-and-get-next',
    '"\e[A": history-search-backward-and-save',
    '"\e[B": history-search-forward-and-save'
    ]
```

binds the three commands with the control-o key, the up arrow key, and the down arrow key, respectively. *Warning:* Sometimes, these keys may be bound to do other actions by the terminal and does not reach to the readline properly (check this by running `stty -a` and reading the `cchars` section). Then you may need to turn off these bindings

before the new readline commands work fine . A prominent case is when control-o is bound to `discard` by the terminal. You can turn this off by running `stty discard undef`.

AUTHORS:

- Kwankyu Lee (2010-11-23): initial version

- Kwankyu Lee (2013-06-05): updated for the new IPython configuration format.

6.2 Sage's IPython Modifications

This module contains all of Sage's customizations to the IPython interpreter. These changes consist of the following major components:

- *SageTerminalApp*

- `SageInteractiveShell`

- *SageTerminalInteractiveShell*

- *interface_shell_embed()*

6.2.1 SageTerminalApp

This is the main application object. It is used by the `$SAGE_LOCAL/bin/sage-ipython` script to start the Sage command-line. It's primary purpose is to

- Initialize the *SageTerminalInteractiveShell*.

- Provide default configuration options for the shell, and its subcomponents. These work with (and can be over-ridden by) IPython's configuration system.

- Load the Sage ipython extension (which does things like preparsing, add magics, etc.).

- Provide a custom *SageCrashHandler* to give the user instructions on how to report the crash to the Sage support mailing list.

6.2.2 SageInteractiveShell

The `SageInteractiveShell` object is the object responsible for accepting input from the user and evaluating it. From the command-line, this object can be retrieved by running:

```
sage: shell = get_ipython()       # not tested
```

Any input is preprocessed and evaluated inside the `shell.run_cell` method. If the command line processing does not do what you want it to do, you can step through it in the debugger:

```
sage: %debug shell.run_cell('?')          # not tested
```

The `SageInteractiveShell` provides the following customizations:

- Modify the libraries before calling system commands. See `system_raw()`.

6.2.3 SageTerminalInteractiveShell

The *SageTerminalInteractiveShell* is a close relative of `SageInteractiveShell` that is specialized for running in a terminal. In particular, running commands like `!ls` will directly write to stdout. Technically, the `system` attribute will point to `system_raw` instead of `system_piped`.

6.2.4 Interface Shell

The function *interface_shell_embed()* takes a `Interface` object and returns an embeddable IPython shell which can be used to directly interact with that shell. The bulk of this functionality is provided through *InterfaceShellTransformer*.

class `sage.repl.interpreter.`**`InterfaceShellTransformer`**(**args*, ***kwds*)

    Bases: `IPython.core.prefilter.PrefilterTransformer`

    Initialize this class. All of the arguments get passed to `PrefilterTransformer.__init__()`.

    `temporary_objects`
        a list of hold onto interface objects and keep them from being garbage collected

    See also:

    interface_shell_embed()

    EXAMPLES:

```
sage: from sage.repl.interpreter import interface_shell_embed
sage: shell = interface_shell_embed(maxima)
sage: ift = shell.prefilter_manager.transformers[0]
sage: ift.temporary_objects
set()
sage: ift._sage_import_re.findall('sage(a) + maxima(b)')
['a', 'b']
```

    `preparse_imports_from_sage`(*line*)
        Finds occurrences of strings such as `sage(object)` in *line*, converts `object` to `shell.interface`, and replaces those strings with their identifier in the new system. This also works with strings such as `maxima(object)` if `shell.interface` is `maxima`.

        Parameters **`line`** (*string*) – the line to transform

> **Warning:** This does not parse nested parentheses correctly. Thus, lines like `sage(a.foo())` will not work correctly. This can't be done in generality with regular expressions.

        EXAMPLES:

```
sage: from sage.repl.interpreter import interface_shell_embed,
→InterfaceShellTransformer
sage: shell = interface_shell_embed(maxima)
sage: ift = InterfaceShellTransformer(shell=shell, config=shell.config,
→prefilter_manager=shell.prefilter_manager)
sage: ift.shell.ex('a = 3')
sage: ift.preparse_imports_from_sage('2 + sage(a)')
'2 + sage0 '
sage: maxima.eval('sage0')
'3'
sage: ift.preparse_imports_from_sage('2 + maxima(a)') # maxima calls set_seed
→on startup which is why 'sage0' will becomes 'sage4' and not just 'sage1'
```

```
'2 +  sage4 '
sage: ift.preparse_imports_from_sage('2 + gap(a)')
'2 + gap(a)'
```

transform(*line*, *continue_prompt*)

Evaluates *line* in `shell.interface` and returns a string representing the result of that evaluation.

>   **Parameters**
>
>   -   **line** (`string`) – the line to be transformed *and evaluated*
>
>   -   **continue_prompt** (`bool`) – is this line a continuation in a sequence of multiline input?

EXAMPLES:

```
sage: from sage.repl.interpreter import interface_shell_embed,
↪InterfaceShellTransformer
sage: shell = interface_shell_embed(maxima)
sage: ift = InterfaceShellTransformer(shell=shell, config=shell.config,
↪prefilter_manager=shell.prefilter_manager)
sage: ift.transform('2+2', False)   # note: output contains triple quotation
↪marks
'sage.misc.all.logstr("""4""")'
sage: ift.shell.ex('a = 4')
sage: ift.transform(r'sage(a)+4', False)
'sage.misc.all.logstr("""8""")'
sage: ift.temporary_objects
set()
sage: shell = interface_shell_embed(gap)
sage: ift = InterfaceShellTransformer(shell=shell, config=shell.config,
↪prefilter_manager=shell.prefilter_manager)
sage: ift.transform('2+2', False)
'sage.misc.all.logstr("""4""")'
```

class `sage.repl.interpreter.`**SageCrashHandler**(*app*)

Bases: `IPython.terminal.ipapp.IPAppCrashHandler`

A custom `CrashHandler` which gives the user instructions on how to post the problem to sage-support.

EXAMPLES:

```
sage: from sage.repl.interpreter import SageTerminalApp, SageCrashHandler
sage: app = SageTerminalApp.instance()
sage: sch = SageCrashHandler(app); sch
<sage.repl.interpreter.SageCrashHandler object at 0x...>
sage: sorted(sch.info.items())
[('app_name', u'Sage'),
 ('bug_tracker', 'http://trac.sagemath.org'),
 ('contact_email', 'sage-support@googlegroups.com'),
 ('contact_name', 'sage-support'),
 ('crash_report_fname', u'Crash_report_Sage.txt')]
```

class `sage.repl.interpreter.`**SageNotebookInteractiveShell**(*ipython_dir=None*, *profile_dir=None*, *user_module=None*, *user_ns=None*, *custom_exceptions=((), None)*, ***kwargs*)

Bases: *sage.repl.interpreter.SageShellOverride*, IPython.core.
interactiveshell.InteractiveShell

IPython Shell for the Sage IPython Notebook

The doctests are not tested since they would change the current rich output backend away from the doctest rich output backend.

EXAMPLES:

```
sage: from sage.repl.interpreter import SageNotebookInteractiveShell
sage: SageNotebookInteractiveShell()   # not tested
<sage.repl.interpreter.SageNotebookInteractiveShell object at 0x...>
```

init_display_formatter()
> Switch to the Sage IPython notebook rich output backend

> EXAMPLES:

> ```
> sage: from sage.repl.interpreter import SageNotebookInteractiveShell
> sage: SageNotebookInteractiveShell().init_display_formatter()   # not tested
> ```

sage.repl.interpreter.**SagePreparseTransformer**(***kwargs*)
> EXAMPLES:

```
sage: from sage.repl.interpreter import SagePreparseTransformer
sage: spt = SagePreparseTransformer()
sage: spt.push('1+1r+2.3^2.3r')
"Integer(1)+1+RealNumber('2.3')**2.3"
sage: preparser(False)
sage: spt.push('2.3^2')
'2.3^2'
```

sage.repl.interpreter.**SagePromptTransformer**(***kwargs*)
> Strip the sage:/....: prompts of Sage.

> EXAMPLES:

```
sage: from sage.repl.interpreter import SagePromptTransformer
sage: spt = SagePromptTransformer()
sage: spt.push("sage: 2 + 2")
'2 + 2'
sage: spt.push('')
''
sage: spt.push("....: 2+2")
'2+2'
```

This should strip multiple prompts: see trac ticket #16297:

```
sage: spt.push("sage:   sage: 2+2")
'2+2'
sage: spt.push("   sage: ....: 2+2")
'2+2'
```

The prompt contains a trailing space. Extra spaces between the last prompt and the remainder should not be stripped:

```
sage: spt.push("   sage: ....:   2+2")
'   2+2'
```

We test that the input transformer is enabled on the Sage command line:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('sage: a = 123')            # single line
sage: shell.run_cell('sage: a = [\n... 123]')    # old-style multi-line
sage: shell.run_cell('sage: a = [\n....: 123]')  # new-style multi-line
```

We test that trac ticket #16196 is resolved:

```
sage: shell.run_cell('    sage: 1+1')
2
sage: shell.quit()
```

class sage.repl.interpreter.**SageShellOverride**

    Bases: object

    Mixin to override methods in IPython's [Terminal]InteractiveShell classes.

    show_usage()

        Print the basic Sage usage.

        This method ends up being called when you enter ? and nothing else on the command line.

        EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('?')
Welcome to Sage ...
sage: shell.quit()
```

    system_raw(*cmd*)

        Run a system command.

        This is equivalent to the sage-native-execute shell script.

        EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.system_raw('false')
sage: shell.user_ns['_exit_code'] > 0
True
sage: shell.system_raw('true')
sage: shell.user_ns['_exit_code']
0
sage: shell.system_raw('R --version')
R version ...
sage: shell.user_ns['_exit_code']
0
sage: shell.quit()
```

class sage.repl.interpreter.**SageTerminalApp**(***kwargs*)

    Bases: IPython.terminal.ipapp.TerminalIPythonApp

    crash_handler_class

        alias of *SageCrashHandler*

    init_shell()

        Initialize the SageInteractiveShell instance.

Note: This code is based on `TerminalIPythonApp.init_shell()`.

EXAMPLES:

```
sage: from sage.repl.interpreter import SageTerminalApp
sage: app = SageTerminalApp.instance()
sage: app.shell
<sage.repl.interpreter.SageTestShell object at 0x...>
```

load_config_file(**args*, ***kwds*)
> Merges a config file with the default sage config.

Note: This code is based on `Application.update_config()`.

---

shell_class
> A trait whose value must be a subclass of a specified class.

test_shell
> A boolean (True, False) trait.

class sage.repl.interpreter.**SageTerminalInteractiveShell**(**args*, ***kwargs*)
> Bases: *sage.repl.interpreter.SageShellOverride*, IPython.terminal. interactiveshell.TerminalInteractiveShell

> IPython Shell for the Sage IPython Commandline Interface

> The doctests are not tested since they would change the current rich output backend away from the doctest rich output backend.

> EXAMPLES:

```
sage: from sage.repl.interpreter import SageTerminalInteractiveShell
sage: SageTerminalInteractiveShell()    # not tested
<sage.repl.interpreter.SageNotebookInteractiveShell object at 0x...>
```

> **init_display_formatter**()
> > Switch to the Sage IPython commandline rich output backend

> > EXAMPLES:

```
sage: from sage.repl.interpreter import SageTerminalInteractiveShell
sage: SageTerminalInteractiveShell().init_display_formatter()    # not tested
```

class sage.repl.interpreter.**SageTestShell**(**args*, ***kwargs*)
> Bases: *sage.repl.interpreter.SageShellOverride*, IPython.terminal. interactiveshell.TerminalInteractiveShell

> Test Shell

> Care must be taken in these doctests to quit the test shell in order to switch back the rich output display backend to the doctest backend.

> EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell();  shell
<sage.repl.interpreter.SageTestShell object at 0x...>
sage: shell.quit()
```

---

init_display_formatter()
> Switch to the Sage IPython commandline rich output backend

> EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell();  shell
<sage.repl.interpreter.SageTestShell object at 0x...>
sage: shell.quit()
sage: shell.init_display_formatter()
sage: shell.quit()
```

quit()
> Quit the test shell.

> To make the test shell as realistic as possible, we switch to the *BackendIPythonCommandline* display backend. This method restores the previous display backend, which is the *BackendDoctest* during doctests.

> EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: from sage.repl.rich_output import get_display_manager
sage: get_display_manager()
The Sage display manager using the doctest backend

sage: shell = get_test_shell()
sage: get_display_manager()
The Sage display manager using the IPython command line backend

sage: shell.quit()
sage: get_display_manager()
The Sage display manager using the doctest backend
```

run_cell(**args*, ***kwds*)
> Run IPython cell

> Starting with IPython-3.0, this returns an success/failure information. Since it is more convenient for doctests, we ignore it.

> EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: rc = shell.run_cell('1/0')
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
...
ZeroDivisionError: rational division by zero
sage: rc is None
True
sage: shell.quit()
```

sage.repl.interpreter.**embedded**()
> Returns True if Sage is being run from the notebook.

> EXAMPLES:

```
sage: from sage.repl.interpreter import embedded
sage: embedded()
False
```

sage.repl.interpreter.**get_test_shell**()

Returns a IPython shell that can be used in testing the functions in this module.

OUTPUT:

An IPython shell

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell(); shell
<sage.repl.interpreter.SageTestShell object at 0x...>
sage: shell.parent.shell_class
<class 'sage.repl.interpreter.SageTestShell'>
sage: shell.parent.test_shell
True
sage: shell.quit()
```

sage.repl.interpreter.**interface_shell_embed**(*interface*)

Returns an IPython shell which uses a Sage interface on the backend to perform the evaluations. It uses *InterfaceShellTransformer* to transform the input into the appropriate `interface.eval(...)` input.

INPUT:

- `interface` – A Sage `PExpect` interface instance.

EXAMPLES:

```
sage: from sage.repl.interpreter import interface_shell_embed
sage: shell = interface_shell_embed(gap)
sage: shell.run_cell('List( [1..10], IsPrime )')
[ false, true, true, false, true, false, true, false, false, false ]
<ExecutionResult object at ..., execution_count=None error_before_exec=None error_
→in_exec=None result=[ false, true, true, false, true, false, true, false, false,
→ false ]>
```

sage.repl.interpreter.**preparser**(*on=True*)

Turn on or off the Sage preparser.

> **Parameters** **on** (`bool`) – if True turn on preparsing; if False, turn it off.

EXAMPLES:

```
sage: 2/3
2/3
sage: preparser(False)
sage: 2/3  # not tested since doctests are always preparsed
0
sage: preparser(True)
sage: 2^3
8
```

6.3 Sage's IPython Extension

A Sage extension which adds sage-specific features:

- magics

    - `%crun`

    - `%runfile`

    - `%attach`

    - `%display`

    - `%mode` (like `%maxima`, etc.)

    - `%%cython`

    - `%%fortran`

- preparsing of input

- loading Sage library

- running init.sage

- changing prompt to Sage prompt

- Display hook

class `sage.repl.ipython_extension.`**`SageCustomizations`**(*shell=None*)

Bases: `object`

Initialize the Sage plugin.

static `all_globals`()

Return a Python module containing all globals which should be made available to the user.

EXAMPLES:

```
sage: from sage.repl.ipython_extension import SageCustomizations
sage: SageCustomizations.all_globals()
<module 'sage.all_cmdline' ...>
```

`init_environment`()

Set up Sage command-line environment

`init_inspector`()

`init_line_transforms`()

Set up transforms (like the preparser).

`register_interface_magics`()

Register magics for each of the Sage interfaces

`run_init`()

Run Sage's initial startup file.

`set_quit_hook`()

Set the exit hook to cleanly exit Sage.

class `sage.repl.ipython_extension.`**`SageJupyterCustomizations`**(*shell=None*)

Bases: *sage.repl.ipython_extension.SageCustomizations*

Initialize the Sage plugin.

static **all_globals**()
>   Return a Python module containing all globals which should be made available to the user when running the Jupyter notebook.
>
>   EXAMPLES:
>
>    ```
>    sage: from sage.repl.ipython_extension import SageJupyterCustomizations
>    sage: SageJupyterCustomizations.all_globals()
>    <module 'sage.repl.ipython_kernel.all_jupyter' ...>
>    ```

class sage.repl.ipython_extension.**SageMagics**(*shell=None*, ***kwargs*)
>   Bases: IPython.core.magic.Magics

>   **attach**(*s*)
>   >   Attach the code contained in the file `s`.
>   >
>   >   This is designed to be used from the command line as `%attach /path/to/file`.
>   >
>   >   •s – string. The file to be attached
>   >
>   >   EXAMPLES:
>   >
>    >    ```
>    >    sage: import os
>    >    sage: from sage.repl.interpreter import get_test_shell
>    >    sage: shell = get_test_shell()
>    >    sage: tmp = os.path.normpath(os.path.join(SAGE_TMP, 'run_cell.py'))
>    >    sage: f = open(tmp, 'w'); _ = f.write('a = 2\n'); f.close()
>    >    sage: shell.run_cell('%attach ' + tmp)
>    >    sage: shell.run_cell('a')
>    >    2
>    >    sage: sleep(1)  # filesystem timestamp granularity
>    >    sage: f = open(tmp, 'w'); _ = f.write('a = 3\n'); f.close()
>    >    ```
>   >
>   >   Note that the doctests are never really at the command prompt, so we call the input hook manually:
>   >
>    >    ```
>    >    sage: shell.run_cell('from sage.repl.attach import reload_attached_files_if_
>    >    ↪modified')
>    >    sage: shell.run_cell('reload_attached_files_if_modified()')
>    >    ### reloading attached file run_cell.py modified at ... ###
>   >
>   >   sage: shell.run_cell('a')
>   >   3
>   >   sage: shell.run_cell('detach(%r)'%tmp)
>   >   sage: shell.run_cell('attached_files()')
>   >   []
>   >   sage: os.remove(tmp)
>   >   sage: shell.quit()
>    >    ```

>   **crun**(*s*)
>   >   Profile C function calls
>   >
>   >   INPUT:
>   >
>   >   •s – string. Sage command to profile.
>   >
>   >   EXAMPLES:
>   >
>    >    ```
>    >    sage: from sage.repl.interpreter import get_test_shell
>    >    sage: shell = get_test_shell()
>    >    sage: shell.run_cell('%crun sum(1/(1+n^2) for n in range(100))')  # optional
>    >    ↪- gperftools
>    >    ```

```
PROFILE: interrupts/evictions/bytes = ...
Using local file ...
Using local file ...
sage: shell.quit()
```

cython (*line*, *cell*)

Cython cell magic

This is syntactic sugar on the `cython()` function.

INPUT:

- `line` – ignored.

- `cell` – string. The Cython source code to process.

OUTPUT:

None. The Cython code is compiled and loaded.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('''
....: %%cython
....: def f():
....:     print('test')
....: ''')
....: shell.run_cell('f()')
```

display (*args*)

A magic command to switch between simple display and ASCII art display.

- `args` – string. See `sage.misc.display_hook.DisplayHookBase.set_display()` for allowed values. If the mode is `ascii_art`, it can optionally be followed by a width.

How to use: if you want activate the ASCII art mod:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%display ascii_art')
```

That means you don't have to use `ascii_art()` to get an ASCII art output:

```
sage: shell.run_cell("i = var('i')")
sage: shell.run_cell('sum(i^2*x^i, i, 0, 10)')
    10       9      8      7      6      5      4      3      2
100*x   + 81*x  + 64*x  + 49*x  + 36*x  + 25*x  + 16*x  + 9*x  + 4*x   + x
```

Then when you want return in 'textual mode':

```
sage: shell.run_cell('%display text plain')
sage: shell.run_cell('%display plain')          # shortcut for "text plain"
sage: shell.run_cell('sum(i^2*x^i, i, 0, 10)')
100*x^10 + 81*x^9 + 64*x^8 + 49*x^7 + 36*x^6 + 25*x^5 + 16*x^4 + 9*x^3 + 4*x^
→2 + x
```

Sometime you could have to use a special output width and you could specify it:

```
sage: shell.run_cell('%display ascii_art')
sage: shell.run_cell('StandardTableaux(4).list()')
[
[                                                          1  4
[                   1  3  4    1  2  4    1  2  3    1  3    1  2    2
[   1  2  3  4,    2      ,   3      ,   4      ,    2  4,   3  4,   3     ,

                     1 ]
   1  3    1  2    2 ]
   2       3       3 ]
   4    ,  4    ,  4 ]
sage: shell.run_cell('%display ascii_art 50')
sage: shell.run_cell('StandardTableaux(4).list()')
[
[
[                   1  3  4    1  2  4    1  2  3
[   1  2  3  4,    2      ,   3      ,   4      ,

                                                1 ]
                   1  4    1  3    1  2    2 ]
   1  3    1  2    2       2       3       3 ]
   2  4,   3  4,   3    ,  4    ,  4    ,  4 ]
```

As yet another option, typeset mode. This is used in the emacs interface:

```
sage: shell.run_cell('%display text latex')
sage: shell.run_cell('1/2')
\newcommand{\Bold}[1]{\mathbf{#1}}\frac{1}{2}
```

Switch back:

```
sage: shell.run_cell('%display default')
```

Switch graphics to default to vector or raster graphics file formats:

```
sage: shell.run_cell('%display graphics vector')
```

fortran(*line*, *cell*)

Fortran cell magic.

This is syntactic sugar on the fortran() function.

INPUT:

- line – ignored.

- cell – string. The Cython source code to process.

OUTPUT:

None. The Fortran code is compiled and loaded.

EXAMPLES:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('''
....: %%fortran
....: C FILE: FIB1.F
....:       SUBROUTINE FIB(A,N)
```

```
....: C
....: C       CALCULATE FIRST N FIBONACCI NUMBERS
....: C
....:         INTEGER N
....:         REAL*8 A(N)
....:         DO I=1,N
....:             IF (I.EQ.1) THEN
....:                 A(I) = 0.0D0
....:             ELSEIF (I.EQ.2) THEN
....:                 A(I) = 1.0D0
....:             ELSE
....:                 A(I) = A(I-1) + A(I-2)
....:             ENDIF
....:         ENDDO
....:         END
....: C END FILE FIB1.F
....: ''')
sage: fib
<fortran object>
sage: from numpy import array
sage: a = array(range(10), dtype=float)
sage: fib(a, 10)
sage: a
array([ 0.,   1.,   1.,   2.,   3.,   5.,   8.,  13.,  21.,  34.])
```

iload(*args*)

A magic command to interactively load a file as in MAGMA.

- `args` – string. The file to be interactively loaded

Note: Currently, this cannot be completely doctested as it relies on `raw_input()`.

EXAMPLES:

```
sage: ip = get_ipython()              # not tested: works only in interactive
↪shell
sage: ip.magic_iload('/dev/null')  # not tested: works only in interactive
↪shell
Interactively loading "/dev/null"  # not tested: works only in interactive
↪shell
```

runfile(*s*)

Execute the code contained in the file s.

This is designed to be used from the command line as `%runfile /path/to/file`.

- `s` – string. The file to be loaded.

EXAMPLES:

```
sage: import os
sage: from sage.repl.interpreter import get_test_shell
sage: from sage.misc.all import tmp_dir
sage: shell = get_test_shell()
sage: tmp = os.path.join(tmp_dir(), 'run_cell.py')
sage: f = open(tmp, 'w'); _ = f.write('a = 2\n'); f.close()
sage: shell.run_cell('%runfile '+tmp)
sage: shell.run_cell('a')
```

```
2
sage: shell.quit()
```

sage.repl.ipython_extension.**load_ipython_extension**(**args*, ***kwargs*)
> Load the extension in IPython.

sage.repl.ipython_extension.**run_once**(*func*)
> Runs a function (successfully) only once.

> The running can be reset by setting the has_run attribute to False

6.4 Magics for each of the Sage interfaces

This module defines magic functions for interpreters. As an example, consider the GAP interpreter which can evaluate a gap command given as a string:

```
sage: gap('SymmetricGroup(4)')          # not tested
SymmetricGroup( [ 1 .. 4 ] )
```

Magics are syntactic sugar to avoid writing the Python string. They are either called as line magics:

```
sage: %gap SymmetricGroup(4)            # not tested
```

or as cell magics, that is, spanning multiple lines:

```
sage: %%gap                             # not tested
....: G := SymmetricGroup(4);
....: Display(G);
```

Note that the cell magic needs semicolons, this is required by the GAP language to separate multiple commands.

class sage.repl.interface_magic.**InterfaceMagic**(*name*, *interface*)
> Bases: object

> Interface Magic

> This class is a wrapper around interface objects to provide them with magics.

> INPUT:

> > •name – string. The interface name

> > •interface – sage.interfaces.expect.Expect. The interface to wrap.

> EXAMPLES:

```
sage: from sage.repl.interface_magic import InterfaceMagic
sage: InterfaceMagic.find('gap')
<sage.repl.interface_magic.InterfaceMagic object at 0x...>
```

> **classmethod all_iter**()
> > Iterate over the available interfaces

> > EXAMPLES:

```
sage: from sage.repl.interface_magic import InterfaceMagic
sage: next(InterfaceMagic.all_iter())
<sage.repl.interface_magic.InterfaceMagic object at 0x...>
```

cell_magic_factory()
   Factory for cell magic

   OUTPUT:

   A function suitable to be used as cell magic.

   EXAMPLES:

```
sage: from sage.repl.interface_magic import InterfaceMagic
sage: cell_magic = InterfaceMagic.find('gap').cell_magic_factory()
sage: output = cell_magic('', '1+1;')
2
sage: output is None
True
sage: cell_magic('foo', '1+1;')
Traceback (most recent call last):
...
SyntaxError: Interface magics have no options, got "foo"
```

   This is how the built cell magic is used in practice:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%%gap\nG:=SymmetricGroup(5);\n1+1;Order(G);')
Sym( [ 1 .. 5 ] )
2
120
sage: shell.run_cell('%%gap foo\n1+1;\n')
File "<string>", line unknown
SyntaxError: Interface magics have no options, got "foo"

sage: shell.run_cell('%%gap?')
Docstring:
Interact with gap

The cell magic %%gap sends multiple lines to the gap interface.
...
```

classmethod find(*name*)
   Find a particular magic by name

   This method is for doctesting purposes only.

   INPUT:

   •name – string. The name of the interface magic to search for.

   OUTPUT:

   The corresponding *InterfaceMagic* instance.

   EXAMPLES:

```
sage: from sage.repl.interface_magic import InterfaceMagic
sage: InterfaceMagic.find('gap')
<sage.repl.interface_magic.InterfaceMagic object at 0x...>
```

line_magic_factory()
   Factory for line magic

   OUTPUT:

A function suitable to be used as line magic.

EXAMPLES:

```
sage: from sage.repl.interface_magic import InterfaceMagic
sage: line_magic = InterfaceMagic.find('gap').line_magic_factory()
sage: output = line_magic('1+1')
sage: output
2
sage: type(output)
<class 'sage.interfaces.gap.GapElement'>
```

This is how the built line magic is used in practice:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('%gap 1+1')
2
sage: shell.run_cell('%gap?')
Docstring:
Interact with gap

The line magic %gap sends a single line to the gap interface.
...
```

classmethod **register_all**(*shell=None*)
    Register all available interfaces

EXAMPLES:

```
sage: class MockShell():
....:     magics = set()
....:     def register_magic_function(self, fn, magic_name, magic_kind):
....:         self.magics.add(magic_name)
....:         print(magic_name, magic_kind)
sage: from sage.repl.interface_magic import InterfaceMagic
sage: InterfaceMagic.register_all(MockShell())    # random output
('gp', 'line')
('gp', 'cell')
('mwrank', 'line')
('mwrank', 'cell')
...
('maxima', 'line')
('maxima', 'cell')
sage: 'gap' in MockShell.magics
True
sage: 'maxima' in MockShell.magics
True
```

6.5 Interacts for the Sage Jupyter notebook

This is mostly the same as the stock `ipywidgets.interact`, but with some customizations for Sage.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import interact
sage: @interact
```

```
....: def f(x=(0,10)):
....:      pass
Interactive function <function f ...> with 1 widget
  x: IntSlider(value=5, min=0, max=10, step=1, description=u'x')
sage: f.widget.children
(IntSlider(value=5, min=0, max=10, step=1, description=u'x'), Output())
```

class `sage.repl.ipython_kernel.interact.`**`sage_interactive`**(*args*, ***kwds*)

    Bases: `ipywidgets.widgets.interaction.interactive`

    Wrapper around the ipywidgets interactive which handles some SageNB specifics.

    EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import sage_interactive
sage: def myfunc(x=10, y="hello", z=None): pass
sage: sage_interactive(myfunc, x=(0,100), z=["one", "two", "three"])
Interactive function <function myfunc ...> with 3 widgets
  x: IntSlider(value=10, min=0, max=100, step=1, description=u'x')
  y: Text(value=u'hello', description=u'y')
  z: Dropdown(value='one', options=['one', 'two', 'three'], description=u'z')
```

    `signature`()

        Return the fixed signature of the interactive function (after a possible `auto_update` parameter was removed).

        EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import sage_interactive
sage: def myfunc(x=[1,2,3], auto_update=False): pass
sage: sage_interactive(myfunc).signature().parameters
OrderedDict([('x', <Parameter ... 'x'>)])
```

    classmethod **`widget_from_iterable`**(*abbrev*, **args*, ***kwds*)

        Convert an unspecified iterable to a widget.

        This behaves like in ipywidgets, except that an iterator (like a generator object) becomes a `SelectionSlider`.

        EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import sage_interactive
sage: sage_interactive.widget_from_iterable([1..5])
Dropdown(value=1, options=[1, 2, 3, 4, 5])
sage: sage_interactive.widget_from_iterable(iter([1..5]))
SelectionSlider(value=1, options=[1, 2, 3, 4, 5])
sage: sage_interactive.widget_from_iterable((1..5))
SelectionSlider(value=1, options=[1, 2, 3, 4, 5])
sage: sage_interactive.widget_from_iterable(x for x in [1..5])
SelectionSlider(value=1, options=[1, 2, 3, 4, 5])
sage: def gen():
....:      yield 1; yield 2; yield 3; yield 4; yield 5
sage: sage_interactive.widget_from_iterable(gen())
SelectionSlider(value=1, options=[1, 2, 3, 4, 5])
```

    classmethod **`widget_from_single_value`**(*abbrev*, **args*, ***kwds*)

        Convert a single value (i.e. a non-iterable) to a widget.

        This supports the Sage `Color` class. Any unknown type is changed to a string for evaluating. This is meant to support symbolic expressions like `sin(x)`.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import sage_interactive
sage: sage_interactive.widget_from_single_value("sin(x)")
Text(value=u'sin(x)')
sage: sage_interactive.widget_from_single_value(sin(x))
EvalText(value=u'sin(x)')
sage: from sage.plot.colors import Color
sage: sage_interactive.widget_from_single_value(Color('cornflowerblue'))
SageColorPicker(value='#6495ed')
```

classmethod **widget_from_tuple**(*abbrev*, **args*, ***kwds*)

Convert a tuple to a widget.

This supports two SageNB extensions: (description, abbrev) if description is a string and (default, abbrev) if abbrev is not a single value.

Symbolic expressions are changed to a floating-point number.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.interact import sage_interactive
sage: sage_interactive.widget_from_tuple( (0, 10) )
IntSlider(value=5, min=0, max=10, step=1)
sage: sage_interactive.widget_from_tuple( (3, (0, 10)) )
IntSlider(value=3, min=0, max=10, step=1)
sage: sage_interactive.widget_from_tuple( (2, dict(one=1, two=2, three=3)) )
Dropdown(value=2, options={'three': 3, 'two': 2, 'one': 1})
sage: sage_interactive.widget_from_tuple( (sqrt(2), pi) )
FloatSlider(value=2.277903107981444, min=1.4142135623730951, max=3.
↪141592653589793, step=0.1)
```

6.6 Widgets to be used for the Sage Jupyter notebook

These are all based on widgets from ipywidgets, changing or combining existing widgets.

class sage.repl.ipython_kernel.widgets.**EvalText**(**args*, ***kwds*)

Bases: *sage.repl.ipython_kernel.widgets.EvalWidget*, ipywidgets.widgets.widget_string.Text

A ipywidgets.Text widget which evaluates (using sage_eval()) its contents and applies an optional transformation.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import EvalText
sage: w = EvalText(value="pi", transform=lambda x: x^2)
sage: w
EvalText(value=u'pi')
sage: w.get_interact_value()
pi^2
```

class sage.repl.ipython_kernel.widgets.**EvalTextarea**(**args*, ***kwds*)

Bases: *sage.repl.ipython_kernel.widgets.EvalWidget*, ipywidgets.widgets.widget_string.Textarea

A ipywidgets.Textarea widget which evaluates (using sage_eval()) its contents and applies an optional transformation.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import EvalTextarea
sage: w = EvalTextarea(value="pi", transform=lambda x: x^2)
sage: w
EvalTextarea(value=u'pi')
sage: w.get_interact_value()
pi^2
```

class `sage.repl.ipython_kernel.widgets.`**EvalWidget**(**args*, ***kwds*)

    Bases: *sage.repl.ipython_kernel.widgets.TransformWidget*

    A mixin class for a widget to evaluate (using `sage_eval()`) the widget value and possibly transform it like *TransformWidget*.

    EXAMPLES:

```
sage: from ipywidgets import ToggleButtons
sage: from sage.repl.ipython_kernel.widgets import EvalWidget
sage: class EvalToggleButtons(EvalWidget, ToggleButtons): pass
sage: w = EvalToggleButtons(options=["pi", "e"], transform=lambda x: x+x)
sage: w
EvalToggleButtons(value='pi', options=['pi', 'e'])
sage: w.get_interact_value()
2*pi
```

    get_value()

        Evaluate the bare widget value using `sage_eval()`.

        EXAMPLES:

            sage: from ipywidgets import Dropdown sage: from sage.repl.ipython_kernel.widgets import EvalWidget sage: class EvalDropdown(EvalWidget, Dropdown): pass sage: w = EvalDropdown(options=["the_answer"], transform=RR) sage: w EvalDropdown(value='the_answer', options=['the_answer']) sage: the_answer = 42 sage: w.get_value() 42 sage: w.get_interact_value() 42.0000000000000

class `sage.repl.ipython_kernel.widgets.`**Grid**(*nrows*, *ncols*, *make_widget*, *description=u''*, *transform=None*)

    Bases: *sage.repl.ipython_kernel.widgets.TransformWidget*, `ipywidgets.widgets.widget_box.HBox`, `ipywidgets.widgets.valuewidget.ValueWidget`

    A square grid of widgets whose value is a list of lists of the values of the individual widgets.

    This is usually created using the `input_grid()` function.

    EXAMPLES:

```
sage: from ipywidgets import Text
sage: from sage.repl.ipython_kernel.widgets import Grid
sage: w = Grid(2, 2, lambda i,j: Text(value="%s,%s"%(i,j)))
sage: w
Grid(value=[[u'0,0', u'0,1'], [u'1,0', u'1,1']], children=(Label(value=u''),
↪VBox(children=(Text(value=u'0,0'), Text(value=u'1,0'))),
↪VBox(children=(Text(value=u'0,1'), Text(value=u'1,1')))))
sage: w.get_interact_value()
[[u'0,0', u'0,1'], [u'1,0', u'1,1']]
```

    description

        A trait for unicode strings.

value
>   An instance of a Python list.

class sage.repl.ipython_kernel.widgets.**HTMLText**(*value=None*, ***kwargs*)
>   Bases: ipywidgets.widgets.widget_string.HTML

>   A HTML widget whose description is always empty.

>   This is used to display arbitrary HTML text in interacts without a label. The text_control() function from SageNB is an alias of *HTMLText*.

>   EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import HTMLText
sage: w = HTMLText("Hello")
sage: w.description
u''
sage: w.description = "text"
sage: w.description
u''
```

>   **description**
>   >   Always return empty string.

>   >   EXAMPLES:

>    ```
>    sage: from sage.repl.ipython_kernel.widgets import HTMLText
>    sage: w = HTMLText("Hello")
>    sage: w.description
>    u''
>    ```

class sage.repl.ipython_kernel.widgets.**SageColorPicker**(***kwargs*)
>   Bases: ipywidgets.widgets.widget_color.ColorPicker

>   A color picker widget return a Sage Color.

>   EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import SageColorPicker
sage: SageColorPicker()
SageColorPicker(value='black')
```

>   **get_interact_value**()
>   >   Return a Sage Color corresponding to the value of this widget.

>   >   EXAMPLES:

>    ```
>    sage: from sage.repl.ipython_kernel.widgets import SageColorPicker
>    sage: SageColorPicker().get_interact_value()
>    RGB color (0.0, 0.0, 0.0)
>    ```

class sage.repl.ipython_kernel.widgets.**TransformFloatRangeSlider**(**args*, ***kwds*)
>   Bases: *sage.repl.ipython_kernel.widgets.TransformWidget*, ipywidgets.widgets.widget_float.FloatRangeSlider

>   An ipywidgets.FloatRangeSlider widget with an optional transformation.

>   EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import TransformFloatRangeSlider
sage: w = TransformFloatRangeSlider(min=0, max=100, value=(7,9), transform=lambda␣
↪x: x[1]-x[0])
```

```
sage: w
TransformFloatRangeSlider(value=(7.0, 9.0), min=0.0, max=100.0, step=1.0)
sage: w.get_interact_value()
2.0
```

class sage.repl.ipython_kernel.widgets.**TransformFloatSlider**(*args*, ***kwds*)

> Bases: *sage.repl.ipython_kernel.widgets.TransformWidget*, ipywidgets.widgets.
> widget_float.FloatSlider
>
> A ipywidgets.FloatSlider widget with an optional transformation.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.ipython_kernel.widgets import TransformFloatSlider
> sage: w = TransformFloatSlider(min=0, max=100, value=7, transform=lambda x:
> →sqrt(x))
> sage: w
> TransformFloatSlider(value=7.0, min=0.0, max=100.0, step=0.1)
> sage: w.get_interact_value()
> 2.6457513110645907
> ```

class sage.repl.ipython_kernel.widgets.**TransformIntRangeSlider**(*args*, ***kwds*)

> Bases: *sage.repl.ipython_kernel.widgets.TransformWidget*, ipywidgets.widgets.
> widget_int.IntRangeSlider
>
> An ipywidgets.IntRangeSlider widget with an optional transformation.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.ipython_kernel.widgets import TransformIntRangeSlider
> sage: w = TransformIntRangeSlider(min=0, max=100, value=(7,9), transform=lambda
> →x: x[1]-x[0])
> sage: w
> TransformIntRangeSlider(value=(7, 9), min=0, max=100, step=1)
> sage: w.get_interact_value()
> 2
> ```

class sage.repl.ipython_kernel.widgets.**TransformIntSlider**(*args*, ***kwds*)

> Bases: *sage.repl.ipython_kernel.widgets.TransformWidget*, ipywidgets.widgets.
> widget_int.IntSlider
>
> An ipywidgets.IntSlider widget with an optional transformation.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.ipython_kernel.widgets import TransformIntSlider
> sage: w = TransformIntSlider(min=0, max=100, value=7, transform=lambda x: x^2)
> sage: w
> TransformIntSlider(value=7, min=0, max=100, step=1)
> sage: w.get_interact_value()
> 49
> ```

class sage.repl.ipython_kernel.widgets.**TransformText**(*args*, ***kwds*)

> Bases: *sage.repl.ipython_kernel.widgets.TransformWidget*, ipywidgets.widgets.
> widget_string.Text
>
> A ipywidgets.Text widget with an optional transformation.
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import TransformText
sage: w = TransformText(value="hello", transform=lambda x: x+x)
sage: w
TransformText(value=u'hello')
sage: w.get_interact_value()
u'hellohello'
```

class sage.repl.ipython_kernel.widgets.**TransformTextarea**(**args*, ***kwds*)

    Bases: *sage.repl.ipython_kernel.widgets.TransformWidget*, ipywidgets.widgets.
widget_string.Textarea

    A ipywidgets.Textarea widget with an optional transformation.

    EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets import TransformTextarea
sage: w = TransformTextarea(value="hello", transform=lambda x: x+x)
sage: w
TransformTextarea(value=u'hello')
sage: w.get_interact_value()
u'hellohello'
```

class sage.repl.ipython_kernel.widgets.**TransformWidget**(**args*, ***kwds*)

    Bases: object

    A mixin class for a widget to transform the bare widget value for use in interactive functions.

    INPUT:

> • transform – a one-argument function which transforms the value of the widget for use by an interactive
> function.
>
> • other arguments are passed to the base class

    EXAMPLES:

```
sage: from ipywidgets import ToggleButtons
sage: from sage.repl.ipython_kernel.widgets import TransformWidget
sage: class TransformToggleButtons(TransformWidget, ToggleButtons): pass
sage: w = TransformToggleButtons(options=["pi", "e"], transform=lambda x: x+x)
sage: w
TransformToggleButtons(value='pi', options=['pi', 'e'])
sage: w.get_interact_value()
'pipi'
```

    get_interact_value()

        Return the transformed value of this widget, by calling the transform function.

        EXAMPLES:

> sage: from ipywidgets import Checkbox sage: from sage.repl.ipython_kernel.widgets import
> TransformWidget sage: class TransformCheckbox(TransformWidget, Checkbox): pass sage:
> w = TransformCheckbox(value=True, transform=int); w TransformCheckbox(value=True) sage:
> w.get_interact_value() 1

    get_value()

        Return self.value.

        This is meant to be overridden by sub-classes to change the input of the transform function.

        EXAMPLES:

```
sage: from ipywidgets import ColorPicker
sage: from sage.repl.ipython_kernel.widgets import TransformWidget
sage: class TransformColorPicker(TransformWidget, ColorPicker): pass
sage: TransformColorPicker(value="red").get_value()
'red'
```

6.7 Functions to construct widgets, based on the old SageNB interface.

These should ensure mostly backwards compatibility with SageNB.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.widgets_sagenb import text_control
sage: text_control("Hello World!")
HTMLText(value=u'Hello World!')
```

sage.repl.ipython_kernel.widgets_sagenb.**checkbox**(*default=True*, *label=None*)
 A checkbox widget.

 INPUT:

   •`default` – (boolean) initial value

   •`label` – optional label

 EXAMPLES:

```
sage: from sage.repl.ipython_kernel.all_jupyter import checkbox
sage: checkbox(label="toggle me")
Checkbox(value=True, description=u'toggle me')
sage: checkbox(default=0)
Checkbox(value=False)
```

sage.repl.ipython_kernel.widgets_sagenb.**color_selector**(*default=(0, 0, 1)*, *label=None*, *widget=None*, *hide_box=False*)

 A widget for choosing a color.

 INPUT:

   •`default` – initial value

   •`label` – optional label

   •`hide_box` – (boolean) if True, do not show the textbox

 EXAMPLES:

```
sage: from sage.repl.ipython_kernel.all_jupyter import color_selector
sage: w = color_selector("orange", label="color me"); w
SageColorPicker(value='#ffa500', description=u'color me')
sage: w.get_interact_value()
RGB color (1.0, 0.6470588235294118, 0.0)
sage: color_selector(Color(0.1, 0.2, 0.3))
SageColorPicker(value='#19334c')
```

`sage.repl.ipython_kernel.widgets_sagenb.`**`input_box`**(*default=None, label=None, type=None, width=80, height=1*)

A textbox widget.

INPUT:

- `default` – initial value

- `label` – optional label

- `type` – function of one variable or `None`. if `type` is `str`, the value of this widget for interactive functions is just the text as `str`. Otherwise, the text is evaluated using `sage_eval()`, `type` is called on it and the result is used as value. Except if `type` is `None`, then the evaluated text is used as value.

- `width` – width of the box

- `height` – if `height > 1`, create a textarea instead of a single-line textbox.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.all_jupyter import input_box
```

The most basic usage is when `type=str`:

```
sage: w = input_box("4+5", type=str, label="enter a string")
sage: w
TransformText(value=u'4+5', description=u'enter a string')
sage: w.get_interact_value()
'4+5'
```

Without `type`, the text is evaluated:

```
sage: w = input_box("4+5")
sage: w
EvalText(value=u'4+5')
sage: w.get_interact_value()
9
```

With a different `type`, the text is evaluated and `type` is called on it:

> sage: w = input_box("4+5",   type=float)   sage: w   EvalText(value=u'4+5')   sage: w.get_interact_value() 9.0

Despite the keyword name, `type` does not need to be a type:

```
sage: w = input_box("4+5", type=sqrt)
sage: w
EvalText(value=u'4+5')
sage: w.get_interact_value()
3
```

When `height > 1`, a textarea is returned:

```
sage: w = input_box("4+5", width=100, height=1)
sage: w
EvalText(value=u'4+5')
sage: w = input_box("4+5", width=100, height=5)
sage: w
EvalTextarea(value=u'4+5')
```

`sage.repl.ipython_kernel.widgets_sagenb.`**`input_grid`**(*nrows*, *ncols*, *default=None*, *label=None*, *to_value=None*, *width=4*)

A widget consisting of a grid (matrix) of textboxes.

The values entered in the textboxes are evaluated (using `sage_eval()`). These are stored as a list of lists on the `value` attribute. The value of this widget for an interactive function is the result of calling `to_value` on this list of lists.

INPUT:

- `nrows`, `ncols` – number of rows and columns in the grid

- `default` – initial value (given as a list of lists, a single constant value or a flat list)

- `label` – optional label

- `to_value` – function to be called to get the value for interactive functions

- `width` – width of each textbox

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.all_jupyter import input_grid
sage: input_grid(2, 2, default=42, label="answers")
Grid(value=[[42, 42], [42, 42]], description=u'answers', children=(Label(value=u
↪'answers'), VBox(children=(EvalText(value=u'42'), EvalText(value=u'42'))),␣
↪VBox(children=(EvalText(value=u'42'), EvalText(value=u'42')))))
sage: w = input_grid(2, 2, default=[[cos(x), sin(x)], [-sin(x), cos(x)]], to_
↪value=matrix); w
Grid(value=[[cos(x), sin(x)], [-sin(x), cos(x)]], children=(Label(value=u''),␣
↪VBox(children=(EvalText(value=u'cos(x)'), EvalText(value=u'-sin(x)'))),␣
↪VBox(children=(EvalText(value=u'sin(x)'), EvalText(value=u'cos(x)')))))
sage: w.get_interact_value()
[ cos(x)  sin(x)]
[-sin(x)  cos(x)]
sage: w = input_grid(2, 2, default=[1, x, x^2, x^3], to_value=lambda x: x[1][1]);␣
↪w
Grid(value=[[1, x], [x^2, x^3]], children=(Label(value=u''),␣
↪VBox(children=(EvalText(value=u'1'), EvalText(value=u'x^2'))),␣
↪VBox(children=(EvalText(value=u'x'), EvalText(value=u'x^3')))))
sage: w.get_interact_value()
x^3
```

`sage.repl.ipython_kernel.widgets_sagenb.`**`range_slider`**(**args*, ***kwds*)

A slider widget to select a range of values.

INPUT:

- `vmin`, `vmax` – minimum and maximum value

- `step_size` – the step size

- `default` – initial value, given as a 2-tuple

- `label` – optional label

- `display_value` – (boolean) if `True`, display the current value.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.all_jupyter import range_slider
sage: range_slider(5, label="slide me")
TransformIntRangeSlider(value=(28, 76), min=5, max=100, step=1, description=u
↪'slide me')
```

```
sage: range_slider(5, 20)
TransformIntRangeSlider(value=(8, 16), min=5, max=20, step=1)
sage: range_slider(5, 20, 0.5)
TransformFloatRangeSlider(value=(8.75, 16.25), min=5.0, max=20.0, step=0.5)
sage: range_slider(5, 20, default=(12,15))
TransformIntRangeSlider(value=(12, 15), min=5, max=20, step=1)
```

The parent of the inputs determines the parent of the value:

```
sage: w = range_slider(5); w
TransformIntRangeSlider(value=(28, 76), min=5, max=100, step=1)
sage: [parent(x) for x in w.get_interact_value()]
[Integer Ring, Integer Ring]
sage: w = range_slider(int(5)); w
IntRangeSlider(value=(28, 76), min=5, max=100, step=1)
sage: [parent(x) for x in w.get_interact_value()]
[<... 'int'>, <... 'int'>]
sage: w = range_slider(5, 20, step_size=RDF("0.1")); w
TransformFloatRangeSlider(value=(8.75, 16.25), min=5.0, max=20.0, step=0.1)
sage: [parent(x) for x in w.get_interact_value()]
[Real Double Field, Real Double Field]
```

Unfortunately, rational numbers are not supported:

```
sage: w = range_slider(5, 20, step_size=10/3); w
Traceback (most recent call last):
...
NotImplementedError: range_slider does not support rational numbers
```

sage.repl.ipython_kernel.widgets_sagenb.**selector**(*values*, *label=None*, *default=None*, *nrows=None*, *ncols=None*, *width=None*, *buttons=False*)

A widget to select a value from a given list of values.

This is rendered as a dropdown box (if `buttons` is False) or as a list of buttons (if `buttons` is True).

INPUT:

- `values` – a list of values to choose from (see examples below for the accepted formats for this)

- `label` – optional label

- `default` – initial value

- `buttons` – (boolean) if True, display buttons instead of a dropdown box

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.all_jupyter import selector
sage: selector(range(5), label="choose one")
Dropdown(value=0, options=[0, 1, 2, 3, 4], description=u'choose one')
sage: selector(range(5), buttons=True, default=4)
ToggleButtons(value=4, options=[0, 1, 2, 3, 4])
```

Apart from a simple list, `values` can be given as a list of 2-tuples (`value`, `label`):

```
sage: selector([(1,"one"), (2,"two"), (3,"three")])
Dropdown(value=1, options=[('one', 1), ('two', 2), ('three', 3)])
sage: selector([(1,"one"), (2,"two"), (3,"three")], buttons=True)
ToggleButtons(value=1, options=[('one', 1), ('two', 2), ('three', 3)])
```

A dict of `label:value` pairs is also allowed. Since a `dict` is not ordered, it is better to use an `OrderedDict`:

```
sage: from collections import OrderedDict
sage: selector(OrderedDict(one=1, two=2, three=3))
Dropdown(value=1, options=OrderedDict([('one', 1), ('three', 3), ('two', 2)]))
sage: selector(OrderedDict(one=1, two=2, three=3), buttons=True)
ToggleButtons(value=1, options=OrderedDict([('one', 1), ('three', 3), ('two',
→2)]))
```

The values can be any kind of object:

> sage: selector([sin(x^2), GF(29), EllipticCurve('37a1')]) Dropdown(value=sin(x^2), options=[sin(x^2), Finite Field of size 29, Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field])

`sage.repl.ipython_kernel.widgets_sagenb.`**`slider`**(*vmin*,   *vmax=None*,   *step_size=None*, *default=None*,   *label=None*,   *display_value=True*, *_range=False*)

A slider widget.

INPUT:

For a numeric slider (select a value from a range):

> •`vmin`, `vmax` – minimum and maximum value

> •`step_size` – the step size

For a selection slider (select a value from a list of values):

> •`vmin` – a list of possible values for the slider

For all sliders:

> •`default` – initial value

> •`label` – optional label

> •`display_value` – (boolean) if `True`, display the current value.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.all_jupyter import slider
sage: slider(5, label="slide me")
TransformIntSlider(value=5, min=5, max=100, step=1, description=u'slide me')
sage: slider(5, 20)
TransformIntSlider(value=5, min=5, max=20, step=1)
sage: slider(5, 20, 0.5)
TransformFloatSlider(value=5.0, min=5.0, max=20.0, step=0.5)
sage: slider(5, 20, default=12)
TransformIntSlider(value=12, min=5, max=20, step=1)
```

The parent of the inputs determines the parent of the value:

```
sage: w = slider(5); w
TransformIntSlider(value=5, min=5, max=100, step=1)
sage: parent(w.get_interact_value())
Integer Ring
sage: w = slider(int(5)); w
IntSlider(value=5, min=5, max=100, step=1)
sage: parent(w.get_interact_value())
<... 'int'>
```

```
sage: w = slider(5, 20, step_size=RDF("0.1")); w
TransformFloatSlider(value=5.0, min=5.0, max=20.0, step=0.1)
sage: parent(w.get_interact_value())
Real Double Field
sage: w = slider(5, 20, step_size=10/3); w
SelectionSlider(value=35/3, options=[5, 25/3, 35/3, 15, 55/3])
sage: parent(w.get_interact_value())
Rational Field
```

Symbolic input is evaluated numerically:

```
sage: w = slider(e, pi); w
TransformFloatSlider(value=2.718281828459045, min=2.718281828459045, max=3.
↪141592653589793, step=0.1)
sage: parent(w.get_interact_value())
Real Field with 53 bits of precision
```

For a selection slider, the default is adjusted to one of the possible values:

```
sage: slider(range(10), default=17/10)
SelectionSlider(value=2, options=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

6.8 Installing the SageMath Jupyter Kernel and Extensions

Kernels have to register themselves with Jupyter so that they appear in the Jupyter notebook's kernel drop-down. This is done by *SageKernelSpec*.

class sage.repl.ipython_kernel.install.**SageKernelSpec**

Bases: object

Utility to manage SageMath kernels and extensions

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec()
sage: spec._display_name    # random output
'SageMath 6.9'
```

classmethod identifier()

Internal identifier for the SageMath kernel

OUTPUT: the string "sagemath".

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: SageKernelSpec.identifier()
'sagemath'
```

kernel_spec()

Return the kernel spec as Python dictionary

OUTPUT:

A dictionary. See the Jupyter documentation for details.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec()
sage: spec.kernel_spec()
{'argv': ..., 'display_name': 'SageMath ...'}
```

symlink(*src*, *dst*)

Symlink `src` to `dst`

This is not an atomic operation.

Already-existing symlinks will be deleted, already existing non-empty directories will be kept.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec()
sage: path = tmp_dir()
sage: spec.symlink(os.path.join(path, 'a'), os.path.join(path, 'b'))
sage: os.listdir(path)
['b']
```

classmethod **update**()

Configure the Jupyter notebook for the SageMath kernel

This method does everything necessary to use the SageMath kernel, you should never need to call any of the other methods directly.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec()
sage: spec.update()  # not tested
```

use_local_jsmol()

Symlink jsmol to the Jupyter notebook.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec()
sage: spec.use_local_jsmol()
sage: jsmol = os.path.join(spec.nbextensions_dir, 'jsmol')
sage: os.path.isdir(jsmol)
True
```

use_local_mathjax()

Symlink SageMath's Mathjax install to the Jupyter notebook.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec()
sage: spec.use_local_mathjax()
sage: mathjax = os.path.join(spec.nbextensions_dir, 'mathjax')
sage: os.path.isdir(mathjax)
True
```

use_local_threejs()

Symlink threejs to the Jupyter notebook.

EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import SageKernelSpec
sage: spec = SageKernelSpec()
sage: spec.use_local_threejs()
sage: threejs = os.path.join(spec.nbextensions_dir, 'threejs')
sage: os.path.isdir(threejs)
True
```

sage.repl.ipython_kernel.install.**have_prerequisites**(*debug=True*)
    Check that we have all prerequisites to run the Jupyter notebook.

    In particular, the Jupyter notebook requires OpenSSL whether or not you are using https. See trac ticket #17318.

    INPUT:

    debug – boolean (default: `True`). Whether to print debug information in case that prerequisites are missing.

    OUTPUT:

    Boolean.

    EXAMPLES:

```
sage: from sage.repl.ipython_kernel.install import have_prerequisites
sage: have_prerequisites(debug=False) in [True, False]
True
```

6.9 The Sage ZMQ Kernel

Version of the Jupyter kernel when running Sage inside the Jupyter notebook or remote Jupyter sessions.

class sage.repl.ipython_kernel.kernel.**SageKernel**(***kwds*)
    Bases: ipykernel.ipkernel.IPythonKernel

    The Sage Jupyter Kernel

    INPUT:

    See the Jupyter documentation

    EXAMPLES:

```
sage: from sage.repl.ipython_kernel.kernel import SageKernel
sage: SageKernel.__new__(SageKernel)
<sage.repl.ipython_kernel.kernel.SageKernel object at 0x...>
```

    banner
        The Sage Banner

        The value of this property is displayed in the Jupyter notebook.

        OUTPUT:

        String.

        EXAMPLES:

```
sage: from sage.repl.ipython_kernel.kernel import SageKernel
sage: sk = SageKernel.__new__(SageKernel)
```

```
sage: sk.banner
'...SageMath version...'
```

help_links

> Help in the Jupyter Notebook
>
> OUTPUT:
>
> See the Jupyter documentation.
>
> ---
>
> **Note:** Urls starting with "kernelspecs" are prepended by the browser with the appropriate path.
>
> ---
>
> EXAMPLES:

```
sage: from sage.repl.ipython_kernel.kernel import SageKernel
sage: sk = SageKernel.__new__(SageKernel)
sage: sk.help_links
[{'text': 'Sage Documentation',
  'url': 'kernelspecs/sagemath/doc/index.html'},
 ...]
```

pre_handler_hook()

> Restore the signal handlers to their default values at Sage startup, saving the old handler at the `saved_sigint_handler` attribute. This is needed because Jupyter needs to change the `SIGINT` handler.
>
> See trac ticket #19135.

shell_class

> A trait whose value must be a subclass of a specified class.

class sage.repl.ipython_kernel.kernel.**SageZMQInteractiveShell**(*ipython_dir=None*, *profile_dir=None*, *user_module=None*, *user_ns=None*, *custom_exceptions=((), None)*, ***kwargs*)

> Bases: *sage.repl.interpreter.SageNotebookInteractiveShell*, ipykernel.zmqshell.ZMQInteractiveShell

6.10 Tests for the IPython integration

First, test the pinfo magic for Python code. This is what IPython calls when you ask for the single-questionmark help, like *foo*?

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(u'from sage.repl.ipython_tests import dummy')
sage: shell.run_cell(u'%pinfo dummy')
Signature:       dummy(argument, optional=None)
Docstring:
   Dummy Docstring Title

   Dummy docstring explanation.
```

```
   INPUT:

   * "argument" -- anything. Dummy argument.

   * "optional" -- anything (optional). Dummy optional.

   EXAMPLES:


...
Init docstring: x.__init__(...) initializes x; see help(type(x)) for signature
File:           .../sage/repl/ipython_tests.py
Type:           function
```

Next, test the pinfo magic for Cython code:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(u'from sage.tests.stl_vector import stl_int_vector')
sage: shell.run_cell(u'%pinfo stl_int_vector')
Docstring:
   Example class wrapping an STL vector

   EXAMPLES:


...
Init docstring: x.__init__(...) initializes x; see help(type(x)) for signature
File:           .../sage/tests/stl_vector.pyx
Type:           type
```

Next, test the pinfo2 magic for Python code. This is what IPython calls when you ask for the double-questionmark help, like foo??

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(u'from sage.repl.ipython_tests import dummy')
sage: shell.run_cell(u'%pinfo2 dummy')
Signature: dummy(argument, optional=None)
Source:
def dummy(argument, optional=None):
    """
    Dummy Docstring Title

    Dummy docstring explanation.

    INPUT:

    - ``argument`` -- anything. Dummy argument.

    - ``optional`` -- anything (optional). Dummy optional.

    EXAMPLES::


    ...
    """
    return 'Source code would be here'
File:      .../sage/repl/ipython_tests.py
Type:      function
```

Next, test the pinfo2 magic for Cython code:

```
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell(u'from sage.tests.stl_vector import stl_int_vector')
sage: shell.run_cell(u'%pinfo2 stl_int_vector')
Source:
cdef class stl_int_vector(SageObject):
    """
    Example class wrapping an STL vector

    EXAMPLES::

...
    """

    cdef vector[int] *data
    cdef string *name

    def __cinit__(self):
        """
        The Cython constructor.

        EXAMPLES::

...
File:   .../sage/tests/stl_vector.pyx
Type:   type
```

Test that there are no warnings being ignored internally:

```
sage: import warnings
sage: warnings.simplefilter('error');  get_test_shell()
<sage.repl.interpreter.SageTestShell object at 0x...>
```

sage.repl.ipython_tests.**dummy**(*argument*, *optional=None*)
> Dummy Docstring Title

> Dummy docstring explanation.

> INPUT:

>> •`argument` – anything. Dummy argument.

>> •`optional` – anything (optional). Dummy optional.

> EXAMPLES:

> ```
> sage: from sage.repl.ipython_tests import dummy
> sage: dummy(1)
> 'Source code would be here'
> ```

6.11 HTML Generator for JSmol

This is all an evil iframe hack to get JSmol to display 3-d graphics while separating JSmol's j2s machinery from your actual web page.

There are some caveats for how to load JSmol, in particular it cannot just load its code from a `file://` uri. To use a html file generated by this module, you need

- A web server,

- The JSmol directory tree must be served by your web server,

- The output of *JSMolHtml.inner_html()* or *JSMolHtml.outer_html()* must be served by the same web server.

See https://github.com/phetsims/molecule-polarity/issues/6 for a discussion of loading JSMol.

class sage.repl.display.jsmol_iframe.**JSMolHtml**(*jmol*, *path_to_jsmol=None*, *width='100%'*, *height='100%'*)

    Bases: sage.structure.sage_object.SageObject

    INPUT:

        •jmol – 3-d graphics or *sage.repl.rich_output.output_graphics3d.* *OutputSceneJmol* instance. The 3-d scene to show.

        •path_to_jsmol – string (optional, default is `'/nbextensions/jsmol'`). The path (relative or absolute) where `JSmol.min.js` is served on the web server.

        •width – integer or string (optional, default: `'100%'`). The width of the JSmol applet using CSS dimensions.

        •height – integer or string (optional, default: `'100%'`). The height of the JSmol applet using CSS dimensions.

    EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: JSMolHtml(sphere(), width=500, height=300)
JSmol Window 500x300
```

    iframe()

        Return HTML iframe

        OUTPUT:

        String.

        EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jmol = JSMolHtml(OutputSceneJmol.example())
sage: print(jmol.iframe())
<iframe srcdoc="
...
</iframe>
```

    inner_html()

        Return a HTML document containing a JSmol applet

        EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jmol = JSMolHtml(OutputSceneJmol.example(), width=500, height=300)
sage: print(jmol.inner_html())
<html>
```

```
<head>
  <style>
    * {
      margin: 0;
      padding: 0;
        ...
</html>
```

js_script()
> The *script()* as Javascript string.
>
> Since the many shortcomings of Javascript include multi-line strings, this actually returns Javascript code to reassemble the script from a list of strings.
>
> OUTPUT:
>
> String. Javascript code that evaluates to *script()* as Javascript string.
>
> EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jsmol = JSMolHtml(OutputSceneJmol.example(), width=500, height=300)
sage: print(jsmol.js_script())
[
  'data "model list"',
  ...
  'isosurface fullylit; pmesh o* fullylit; set antialiasdisplay on;',
].join('\n');
```

outer_html()
> Return a HTML document containing an iframe with a JSmol applet
>
> OUTPUT:
>
> String
>
> EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jmol = JSMolHtml(OutputSceneJmol.example(), width=500, height=300)
sage: print(jmol.outer_html())
<html>
<head>
  <title>JSmol 3D Scene</title>
</head>
</body>

<iframe srcdoc="
      ...
</html>
```

script()
> Return the JMol script file.
>
> This method extracts the Jmol script from the Jmol spt file (a zip archive) and inlines meshes.
>
> OUTPUT:
>
> String.

EXAMPLES:

```
sage: from sage.repl.display.jsmol_iframe import JSMolHtml
sage: from sage.repl.rich_output.output_graphics3d import OutputSceneJmol
sage: jsmol = JSMolHtml(OutputSceneJmol.example(), width=500, height=300)
sage: jsmol.script()
'data "model list"\n10\nempt...aliasdisplay on;\n'
```

6.12 Sage Wrapper for Bitmap Images

Some computations in Sage return bitmap images, for example matrices can be turned into bitmaps directly. Note that this is different from all plotting functionality, the latter can equally produce vector graphics. This module is about bitmaps only, and a shallow wrapper around `PIL.Image`. The only difference is that *Image* is displayed as graphics by the Sage if the UI can.

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('RGB', (256, 256), 'white')
sage: pixels = img.pixels()
sage: for x in range(img.width()):
....:     for y in range(img.height()):
....:         pixels[x, y] = (x, y, 100)
sage: img
256x256px 24-bit RGB image
sage: type(img)
<class 'sage.repl.image.Image'>
```

class `sage.repl.image.`**`Image`**(*mode*, *size*, *color='white'*)

Bases: `sage.structure.sage_object.SageObject`

Creates a new image with the given mode and size.

INPUT:

- mode – string. The mode to use for the new image. Valid options are:

  - `'1'` (1-bit pixels, black and white, stored with one pixel per byte)

  - `'L'` (8-bit pixels, black and white)

  - `'P'` (8-bit pixels, mapped to any other mode using a color palette)

  - `'RGB'` (3x8-bit pixels, true color)

  - `'RGBA'` (4x8-bit pixels, true color with transparency mask)

  - `'CMYK'` (4x8-bit pixels, color separation)

  - `'YCbCr'` (3x8-bit pixels, color video format)

  - `'LAB'` (3x8-bit pixels, the L*a*b color space)

  - `'HSV'` (3x8-bit pixels, Hue, Saturation, Value color space)

  - `'I'` (32-bit signed integer pixels)

  - `'F'` (32-bit floating point pixels)

- size – 2-tuple, containing (width, height) in pixels.

• `color` – string or tuple of numeric. What colour to use for the image. Default is black. If given, this should be a a tuple with one value per band. When creating RGB images, you can also use colour strings as supported by the ImageColor module. If the colour is None, the image is not initialised.

OUTPUT:

A new *Image* object.

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: Image('P', (16, 16), (13,))
16x16px 8-bit Color image
```

height()

> Return the vertical dimension in pixels
>
> OUTPUT:
>
> Integer.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.image import Image
> sage: img = Image('1', (12, 34), 'white')
> sage: img.width()
> 12
> sage: img.height()
> 34
> ```

mode()

> Return the color mode
>
> OUTPUT:
>
> String. As given when constructing the image.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.image import Image
> sage: img = Image('YCbCr', (16, 16), 'white')
> sage: img.mode()
> 'YCbCr'
> ```

pil

> Access the wrapped PIL(low) Image
>
> OUTPUT:
>
> The underlying `PIL.Image.Image` object.
>
> EXAMPLES:
>
> ```
> sage: from sage.repl.image import Image
> sage: img = Image('RGB', (16, 16), 'white')
> sage: img.pil
> <PIL.Image.Image image mode=RGB size=16x16 at 0x...>
> ```

pixels()

> Return the pixel map
>
> OUTPUT:

The PIL PixelAccess object that allows you to get/set the pixel data.

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('RGB', (16, 16), 'white')
sage: img.pixels()
<PixelAccess object at 0x...>
```

save (*filename*)

Save the bitmap image

INPUT:

- •`filename` – string. The filename to save as. The given extension automatically determines the image file type.

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('P', (12, 34), (13,))
sage: filename = tmp_filename(ext='.png')
sage: img.save(filename)
sage: open(filename).read().startswith('\x89PNG')
True
```

show ()

Show this image immediately.

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

OUTPUT:

This method does not return anything. Use *save()* if you want to save the figure as an image.

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('1', (12, 34), 'white')
sage: img.show()
```

width ()

Return the horizontal dimension in pixels

OUTPUT:

Integer.

EXAMPLES:

```
sage: from sage.repl.image import Image
sage: img = Image('1', (12, 34), 'white')
sage: img.width()
12
sage: img.height()
34
```

6.13 The Sage Input Hook

This lets us perform actions while IPython is sitting at the terminal input prompt. We use it to reload attached files if they have changed.

sage.repl.inputhook.**install**()

> Install the Sage input hook

> EXAMPLES:

```
sage: from sage.repl.inputhook import install
sage: install()
```

sage.repl.inputhook.**sage_inputhook**(*context*)

sage.repl.inputhook.**uninstall**()

> Uninstall the Sage input hook

> EXAMPLES:

```
sage: from sage.repl.inputhook import uninstall
sage: uninstall()
```

SEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

Symbols

A

B

C

F

G

H

M

N

O

text (sage.repl.rich_output.preferences.DisplayPreferences attribute), 51

threejs_offline_scripts() (sage.repl.rich_output.backend_ipython.BackendIPythonCommandline method), 86

threejs_offline_scripts() (sage.repl.rich_output.backend_ipython.BackendIPythonNotebook method), 87

threejs_offline_scripts() (sage.repl.rich_output.backend_sagenb.BackendSageNB method), 90

threejs_scripts() (sage.repl.rich_output.display_manager.DisplayManager method), 47

toplevel() (sage.repl.display.pretty_print.SagePrettyPrinter method), 36

trace() (in module sage.misc.trace), 8

transform() (sage.repl.interpreter.InterfaceShellTransformer method), 96

TransformFloatRangeSlider (class in sage.repl.ipython_kernel.widgets), 113

TransformFloatSlider (class in sage.repl.ipython_kernel.widgets), 114

TransformIntRangeSlider (class in sage.repl.ipython_kernel.widgets), 114

TransformIntSlider (class in sage.repl.ipython_kernel.widgets), 114

TransformText (class in sage.repl.ipython_kernel.widgets), 114

TransformTextarea (class in sage.repl.ipython_kernel.widgets), 115

TransformWidget (class in sage.repl.ipython_kernel.widgets), 115

try_format() (sage.repl.display.util.TallListFormatter method), 40

types (sage.repl.rich_output.display_manager.DisplayManager attribute), 47

U

unicode_art_formatter() (sage.repl.rich_output.backend_base.BackendBase method), 76

uninstall() (in module sage.repl.inputhook), 132

uninstall() (sage.repl.rich_output.backend_base.BackendBase method), 77

uninstall() (sage.repl.rich_output.backend_doctest.BackendDoctest method), 81

update() (sage.repl.ipython_kernel.install.SageKernelSpec class method), 122

use_local_jsmol() (sage.repl.ipython_kernel.install.SageKernelSpec method), 122

use_local_mathjax() (sage.repl.ipython_kernel.install.SageKernelSpec method), 122

use_local_threejs() (sage.repl.ipython_kernel.install.SageKernelSpec method), 122

V

validate() (sage.repl.rich_output.backend_doctest.BackendDoctest method), 82

value (sage.repl.ipython_kernel.widgets.Grid attribute), 112

W

widget_from_iterable() (sage.repl.ipython_kernel.interact.sage_interactive class method), 110

widget_from_single_value() (sage.repl.ipython_kernel.interact.sage_interactive class method), 110

widget_from_tuple() (sage.repl.ipython_kernel.interact.sage_interactive class method), 111

width() (sage.repl.image.Image method), 131

world_readable() (in module sage.repl.rich_output.backend_sagenb), 92