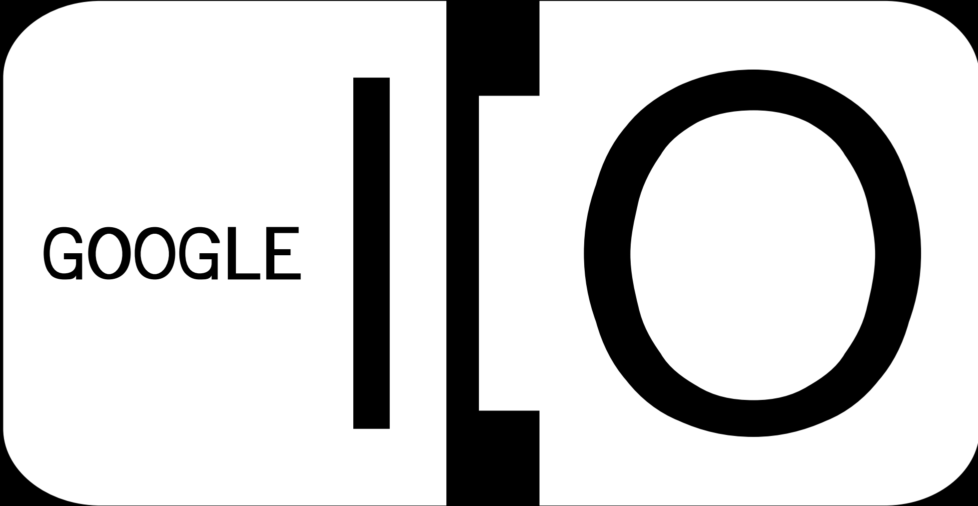
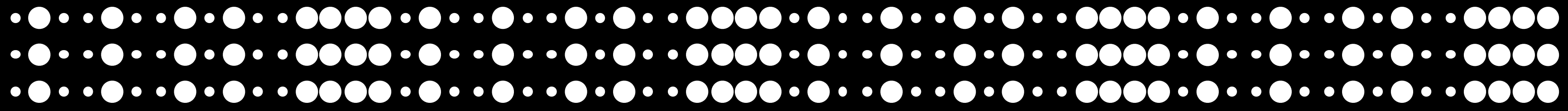
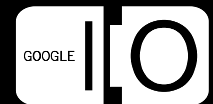

GOOGLE
IO



Painless Python for Proficient Programmers

Alex Martelli, aleax@google.com



Audience for this talk

- experienced programmers (typically Java, C++, C, ...)
- no Python knowledge necessary
  - a little can't hurt -- but...:
  - if you know Python well, you could get bored!
- advance apologies for...:
  - too-succint code (short names, compact formatting...) -- sorry, but, it does have to fit in a slide!-)
  - a few verbose and elementary slides (for completeness!)
  - some "out of order" code examples (to keep things more interesting...;-)

Python's key aspects

- Very High Level Language (VHLL)
  - clean, spare syntax
  - simple, regular semantics
  - object-oriented, but multi-paradigm
  - the key goal is enhancing developer productivity through: modularity, uniformity, simplicity, pragmatism
- rich standard library
- thousands of third-party extensions and tools
- several good implementations, all open-source:
  - CPython 2.5, IronPython 1.1 [.NET], Jython 2.2 [JVM], pypy 1.0
  - we'll focus on CPython

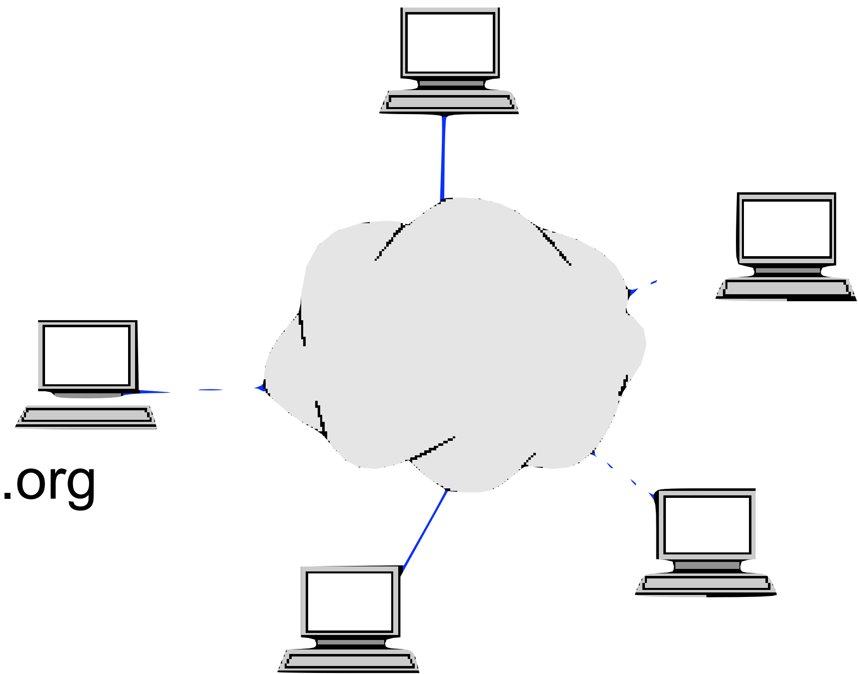
Lots of excellent books (some are a bit dated...)

My personal few top book recommendations

- Introductory (but not simplistic!)
    - Python for Dummies
    - Core Python Programming
    - Beginning Python
- More advanced
    - Python Cookbook (actually wide-spectrum) [bias alert!-)]
    - Python in a Nutshell [bias alert!-)]
    - Python Essential Reference

Online resources

- www.python.org
- www.diveintopython.org
- (Usenet) comp.lang.python
- (Mailing list) help @ python.org
- any good search engine!



Python: some basics

- interactive shell (>>> is the primary prompt, `...` means "continue and complete the current statement")
  - only to try things out, check expressions' values
- for actual programming, write source files, e.g. `foo.py`
  - auto-compiled to `foo.pyc` upon import
- simple assignment:

  <name> = <expression>
- binds the name to the expression's value
  - names aren't "declared" (there **are** no "declarations" at all!)
  - names have no "type" (*objects* do!)
- None: placeholder for "there's nothing here"

GOOGLE I/O

Seeing results

- the interactive shell displays non-None expression results
- the `print` statement does elementary I/O to standard output
  - use comma to separate the values you want to emit
  - `print` uses space separators and newline at the end
- in real programs, you'll use very different tools:
  - the `logging` package for logging purposes
  - add strings to a HTML response object
  - GUI widgets and their methods
  - ...

Some trivial examples

```
print 'Hello, world!' # classic!-)

x = 23       # name x now means 23
print x      # emits 23
x = 'foo'    # now x means 'foo' instead
print x      # emits foo
del x        # name x reverts to undefined
print x      # raises an error ("exception")

y = None     # name y gets bound to None
print y      # emits None
```

Similarities with Java

- both typically compiled → bytecode
  - but: Python compiles implicitly (like a built-in `make`!)
- "everything" inherits from `object`
  - but: in Python, also numbers, functions, classes, ...
  - "everything is a first-class object", no "boxing/unboxing"
- uniform *object-reference* semantics
  - assignment, argument-passing, return
- garbage collection (but: CPython also uses RC...)
- key "marginalia": vast standard library (+ third party packages & tools), introspection, serialization, threads, ...

GOOGLE I/O

Similarities with C++

- multi-paradigm (not just "forced" OOP;-)
  - OOP, procedural, generic, a little FP
- multiple inheritance
- operator overloading
  - but: not for simple assignment (not an operator!)
- "signature-based" polymorphism
  - as if "everything was a template" (but: nicer syntax;-)
- abundant choices for most all "side issues"
  - GUI, Web & other networking, DB, IPC, distributed processing...

Deep connection with C

"Spirit of C" @87% (more than Java or C++ despite surface syntax...), according to ISO C Standard "Rationale":

1. trust the programmer
2. don't stop the programmer from doing what needs to be done
3. keep the language small and simple
4. provide only one way to perform an operation
5. (be fast, even if not guaranteed to be portable)

   (this is the one point out of five that's not at 100% in Python:-)

The Zen of Python

```
>>> import this
```

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

[...]

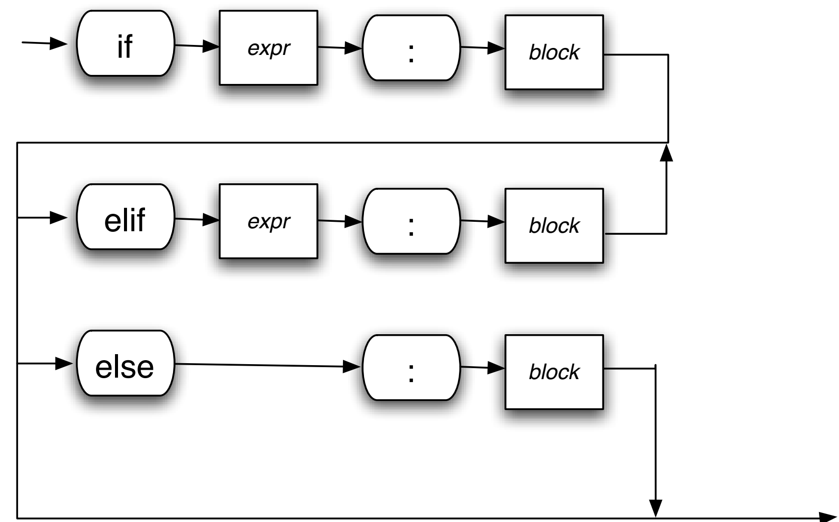
In the face of ambiguity, refuse the temptation to guess.

There should be one, preferably only one, obvious way to do it.

GOOGLE I/O

Python vs Java/C++/C: syntax

- spare syntax, no ornamentation
  - "no wasted pixels" (Tufte)
    - no { } around blocks, just indent
    - no ( ) around conditions in `if` and `while`
    - generally little punctuation



GOOGLE I/O

Python vs Java/C++/C: semantics

- types: strong but dynamic
  - *names* have no types (*objects* have types)
  - "duck typing" (if it walks like a duck, &c)
- no "declarations", just *statements*
- "everything" is a first-class object
  - classes, functions, methods, modules, packages, ...
- the focus is on high and very high levels
  - though lower levels are decently supported too

Duck Typing

Takes some work to teach the ducks to type, but the returns in productivity are quite worthwhile;-)

GOOGLE IO

Flow control

- if <expr>: <nested block>
  - 0+ elif <expr>: <nested block>
  - optionally: else: <nested block>
- while <expr>: <nested block>
  - block can contain break, continue
  - optionally: else: <nested block>
    - executes if no break terminated the loop
- for <name> in <iterable>:
  - break, continue, else: like in while
- with <expr> as <name>: <nested block>
  - in 2.5, requires: from __future__ import with_statement

Built-in simple types (all *immutable*)

- numbers: `int`, `long`, `float`, `complex`
  - `23 943721743892819 0x17  2.3  4.5+6.7j`
  - operators: `+ - * ** / // % ~ & | ^ << >>`
  - built-in functions: `abs min max pow round sum`
- strings: plain and Unicode
  - `'single' "double" r'aw' u"nicode" \n &c`
  - operators: + (cat), * (rep), % (format)
    - rich "format language" (akin to C's `printf`)
  - built-in functions: `chr ord unichr`
  - are immutable sequences: `len`, `[]` (index/slice), `for`
    - each item is a "character" (actually: a length-1 string)
  - *tons and tons* of methods: `capitalize`, `center`, ...

Discovering methods

```
>>> dir('anystring')
['__add__', '__class__', '__contains__',
'__delattr__', '__doc__', '__eq__', '__ge__',
'__getattribute__', '__getitem__',
'__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__',
'__str__', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs',
'find', 'index', 'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'partition',
'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

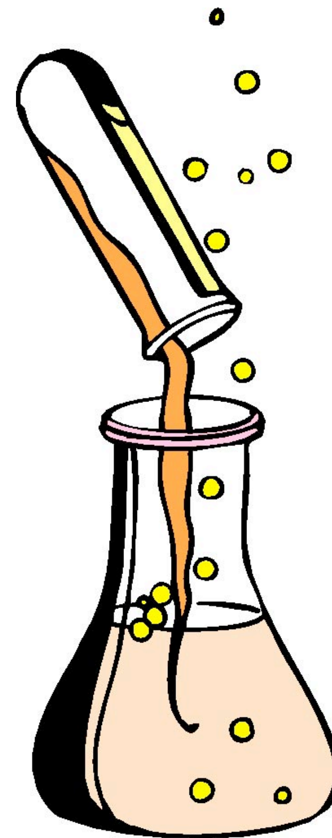
GOOGLE I/O

Getting help about a method

```
>>> help(''.center)
Help on builtin function center:

center(...)
  S.center(width[, fillchar]) -> string

  Returns S centered in a string of length
  width. Padding is done using the
  specified fill character (default is
  a space)
(END)
>>>
```

...and then, experiment!

```
>>> 'ciao'.center(10, '+')
'+++ciao+++'
>>> 'ciao'.center(3, '+')
'ciao'
>>> 'ciao'.center(9, '+')
'+++ciao++'
```



GOOGLE IO

Built-in container types

- `tuple`: immutable sequence
  - `() (23,) (23, 45) tuple('ciao')`
- `list`: mutable sequence (a "vector")
  - `[] [23] [23, 45] list('ciao')`
- `set`, `frozenzet`: simple hashtables
  - `set() set((23,)) set('ciao')`
- `dict`: map key→value by hashtable
  - `{} {2:3} {4:5, 6:7} dict(ci='ao')`
- all containers support: `len(c)`, looping (`for x in c`), membership testing (`if x in c`)
  - and, generally, many methods (sets also support operators)

Sequences

- strings, tuples and lists are sequences (other sequence types are in the standard library)
- repetition (`c*N`), catenation (`c1+c2`)
- indexing: `c[i]`, slicing: `c[i:j]` and `c[i:j:k]`:
  - `'ciao'[2]=='a', 'ciao'[3:1:-1]=='oa'`
  - always: start included, end excluded (per Koenig...)
- lists are mutable sequences (can assign to item or slice)
  - assigning to a slice can change the length
- dicts and sets are *not* sequences (though they are iterables)
  - ordering "arbitrary" (as befits hash tables), no slicing

More about containers

- mutating vs non-mutating methods
  - most mutating methods return None
  - e.g: `alist.append(...)`, `aset.add(...)`, `adict.update(...)`
- a few mutating methods return non-None:
  - `alist.pop(...)`, `adict.pop(...)`, `adict.setdefault(...)`
- `set` has an immutable variant (`frozenset`)
- `dict` has a variant `defaultdict` (in module `collections`)
- also in `collections`: deque (sequence with added methods ...`left`, `clear`, `rotate`; no slicing, no sort)
- no heap type, but rather heap functions on lists (module `heapq`)

GOOGLE I/O

Example: making a heap (priority-queue) **type**

```
import heapq
class heap(object):
    def __init__(self, content=()):
        self._L = list(content)
        heapq.heapify(self._L)
    def push(self, item): heapq.heappush(self._L, item)
    def pop(self): return heapq.heappop(self._L)
    def replace(self, item):
        return heapq.heapreplace(self._L, item)
    def __len__(self): return len(self._L)
    def __iter__(self): return iter(self._L)
    def __repr__(self): return 'heap(%r)' % self._L
```

GOOGLE I/O

Comparisons, tests, truth (or, rather, "truthiness")

- equality, identity: `==` `!=` `is` `is not`
- order: `<` `>` `<=` `>=`
- membership: `in` `not in`
- "chaining": `3 <= x < 9`
- falsy: numbers == `0`, `""`, None, empty containers, `False` (aka `bool(0)`)
- truthy: anything else, `True` (aka `bool(1)`)
- `not x == not bool(x)` for any x
- `and`, `or` "short-circuit" (-> return an operand)
- same for built-ins `any`, `all` (-> return a `bool`)

Exceptions

- Errors (and other "anomalies") "raise exceptions" (instances of `Exception` or any subtype of `Exception`)
- Statement `raise` explicitly raises an exception
- Exceptions propagate "along the call stack", terminating functions along the way, until they're "caught"
- If uncaught, an exception terminates the program
- Statement `try/except` can catch exceptions (also: `try/finally`, and elegant `with` to implement "RAAI" [in 2.5, needs `from __future__ import with_statement` at start of module])

Some RAAI examples

```
from __future__ import with_statement

with open('afile.txt') as f:
    for aline in f: print aline[3],

with some_lock do: a_critical_section()

import contextlib, mysqldb
with contextlib.closing(mysqldb.connect()) as db:
  with contextlib.closing(db.cursor()) as c:
    c.execute("SELECT * FROM t WHERE t.zap<23")
    for record in c.fetchall(): ...
```

iterators and for loops

```
for i in c: <body>

    ===>

_t = iter(c)
while True:
  try: i = _t.next()
  except StopIteration: break
  <body>
```

also:

`(<exp> for i in c <opt.clauses>)` ("genexp")

`[<exp> for i in c <opt.clauses>]` ("list comprehension")

Some examples of genexp use (w/"accumulators")

```
def countall(it): return sum(1 for x in it)
print countall(x for x in range(99) if x%5==x%7) # 15


if any(x>5 for x in xs): ...


if all(x>y>z for x, y, z in zip(xs, ys, zs)): ...
```

or, better (since zip makes a list...):

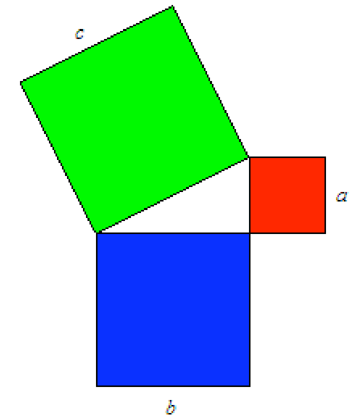
```
from itertools import izip
if all(x>y>z for x, y, z in izip(xs, ys, zs)): ...


print max(i for i, x in enumerate(xs) if x>0)
```

PYTHON

function definition and calling

- `def <name>(<parms>): <body>`
  - <body> compiled, not executed
- call `name(<args>)` executes the body
- last 0+ parms may have "default values", `<name>=<expr>` (`expr` evaluates **once**, at def time), then the args are *optional* in the call
- last 0+ args may be "named", `<name>=<expr>`
- <parms> may end with `*<name>` (`tuple` of 0+ positional args) and/or `**<name>` (`dict` of 0+ named args)

Example: sum of squares




```
def sumsq(a, b): return a*a+b*b
print sumsq(23, 45)
```

More general, higher-abstraction:

```
def sumsq(*a): return sum(x*x for x in a)
```

Lower-abstraction, slower but still OK:

```
def sumsq(*a):
    total = 0
    for x in a: total += x*x
    return total
```

Generators

- functions with `yield` instead of `return`
- each call builds and returns an *iterator* (object with a `next` method, suitable for iterating-on e.g in a `for` loop)
- end of function raises `StopIteration`

```
def enumerate(seq):   # actually a built-in...
  n = 0
  for item in seq:
    yield n, item
    n += 1

def enumerate2(seq): # higher-abstraction variant:
  from itertools import izip, count
  return izip(count(), seq)
```

GOOGLE IO

An endless generator

```
def fibonacci():
  i = j = 1
  while True:
    r, i, j = i, j, i + j
    yield r

for rabbits in fibonacci():
  print rabbits,
  if rabbits > 100: break
1 1 2 3 5 8 13 21 34 55 89 144
```

GOOGLE IO

Closures

- `def` is an executable statement (each execution creates a new function object), and scoping is lexical, so...:

```
def makeAdder(addend):
  def adder(augend):
    return augend + addend
  return adder

a23 = makeAdder(23)
a42 = makeAdder(42)
print a23(100), a42(100), a23(a42(100))
123 142 165
```

Decorators

```
@<decorator>
def <name> etc, etc
```

- is like:

```
def <name> etc, etc
<name> = <decorator>(<name>)
```

- Handy syntax to apply a Higher-Order-Function (HOF); <decorator> may be a name or a call (HOF^2...!)

Classes ("new-style")

```
class <name>(<bases>):
  <body>
```

- <body> is usually a series of `def` and assignment statements; names defined or assigned become attributes of new class object <name> (functions become "methods")

- attributes of any base are also attributes of the new class, unless "overridden" (assigned or defined in the body)

Instantiating a class

To create an instance, just call the class:

```
class eg(object):
  cla = []                  # class attribute
  def __init__(self):       # initializer
    self.ins = {}           # instance attribute
  def meth1(self, x):       # a method
    self.cla.append(x)
  def meth2(self, y, z):    # another method
    self.ins[y] = z
es1 = eg()
es2 = eg()
```

Classes and instances

```
print es1.cla, es2.cla, es1.ins, es2.ins
[] [] {} {}

es1.meth1(1); es1.meth2(2, 3)
es2.meth1(4); es2.meth2(5, 6)
print es1.cla, es2.cla, es1.ins, es2.ins
[1, 4] [1, 4] {2: 3} {5: 6}
print es1.cla is es2.cla
True
print es1.ins is es2.ins
False
```

Look-up mechanics

```
inst.method(arg1, arg2)

    ==>

type(inst).method(inst, arg1, arg2)
```

`inst.name` [[whether it later gets called, or not...!]]

==> ("descriptors" can change this behavior...)

1. first try `inst.__dict__['name']`
2. else try `type(inst).__dict__['name']`
3. else try each of `type(inst).__bases__`
4. finally try `type(inst).__getattr__(inst, 'name')`
5. if all else fails, `raise AttributeError`

Subclassing

```
class sub(eg):
  def meth2(self, x, y=1):     # override
    eg.meth2(self, x, y)       # super-call
    # or: super(sub, self).meth2(x, y)

class repeater(list):
  def append(self, x):
    for i in 1, 2:
      list.append(self, x)

class data_overrider(sub):
  cla = repeater()
```

GOOGLE I/O

Properties

```
class blah(object):
  def getter(self):
    return ...
  def setter(self, value): ...
  name = property(getter, setter)
inst = blah()
```

Now...:

```
print inst.name  # like inst.getter()
inst.name = 23   # like inst.setter(23)
```

GOOGLE I/O

Why properties matter:

- **never** "hide" attributes behind getters/setters "for flexibility"
- instead, expose interesting attributes directly
- if/when in a future release you need a getter and/or a setter...:
  - write the new needed methods,
  - wrap them up into a `property`,
  - and all client-code of the class need NOT change!
- down with boilerplate! **NEVER** code like:

```
def getFoo(self): return self._foo
def setFoo(self, foo): self._foo = foo
```

just name the attribute `foo` (not `_foo`) so it's directly available!

Operator overloading

- "special methods" have names starting and ending with __ (double-underscore AKA "dunder"):

```
__new__ __init__ __del__        # ctor, init, finalize
__repr__ __str__ __int__        # convert
__lt__ __gt__ __eq__ ...        # compare
__add__ __sub__ __mul__ ...     # arithmetic
__call__ __hash__ __nonzero_ ...
__getattr__ __setattr__ __delattr__
__getitem__ __setitem__ __delitem__
__len__ __iter__ __contains__
__get__ __set__ __enter__ __exit__ ...
```

- Python calls a type's special methods appropriately when you perform operations on instances of the type

An endless iterator

```
class Fibonacci(object):
  def __init__(self): self.i = self.j = 1
  def __iter__(self): return self
  def next(self):
    r, self.i = self.i, self.j
    self.j += r
    return r
for rabbits in Fibonacci():
  print rabbits,
  if rabbits > 100: break
1 1 2 3 5 8 13 21 34 55 89 144
```

Built-in functions

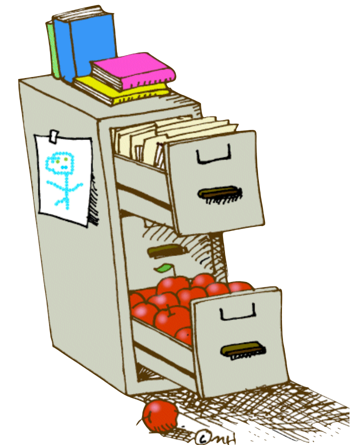
- don't call special methods directly: let built-in functions and operators do it right!
  - e.g.: `abs(x)`, *never* `x.__abs__()`
- many interesting built-in functions:

```
abs any all chr cmp compile dir enumerate eval
getattr hasattr hex id intern isinstance iter len
max min oct open ord pow range repr reversed round
setattr sorted sum unichr xrange zip
```

- many, MANY other important types and functions in standard library modules (`array`, `bisect`, `cmath`, `collections`, `contextlib`, `copy`, `functools`, `heapq`, `inspect`, `itertools`, `operator`, `pprint`, `Queue`, `random`, `re`, `StringIO`, `struct`, `weakref`, ...)

Example: indexing a textfile

```
# Build map word → [list of line #s containing it]
indx = {}
with open(filename) as f:
  for n, line in enumerate(f):
    for word in line.split():
      indx.setdefault(word, []).append(n)
# emit it in alphabetical order
for word in sorted(indx):
  print "%s:" % word,
  for n in indx[word]: print n,
  print
```



GOOGLE I/O

Or, a bit simpler, with a little stdlib help...:

```
import collections
indx = collections.defaultdict(list)
with open(filename) as f:
  for n, line in enumerate(f):
    for word in line.split():
      indx[word].append(n)
for word in sorted(indx):
  print "%s:" % word,
  for n in indx[word]: print n,
  print
```

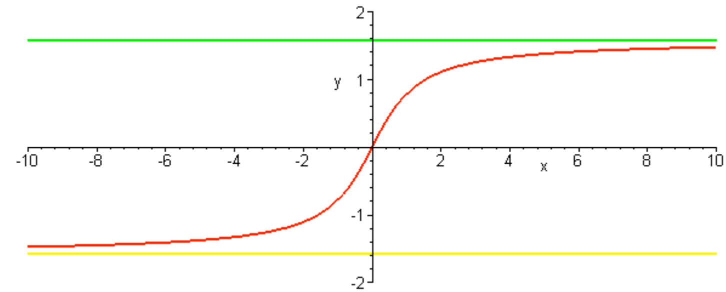
Other possibilities given the indx map

```
# the 7 most popular words in the textfile
import heapq
for w in heapq.nlargest(7, indx, key=indx.get):
  print w

# lines that contain N given words
def enwhack(*words):
  words = list(words)
  r = set(indx[words.pop()])
  for w in words: r &= set(indx[w])
  return sorted(r)
```

Importing modules

- `import modulename`
- `from some.given.package import modulename`
  - then, in either case, use `modulename.blah`
  - some abbreviations (not necessarily recommended...):
    - alias module name with an 'as' clause:
    - `import thisnameisfartoolong as z`
      - then use `z.blah`
    - `from thisnameisfartoolong import blah`
    - `from thisnameisfartoolong import *` # **not** recommended

Import example




```
import math
print math.atan2(1, 3)
# emits 0.321750554397
print atan2(1, 3)
# raises a NameError exception
from math import atan2
# injects atan2 into current namespace
# handy for interactive use, often confusing in "real" programs
# and, always avoid (except for interactive use)...:
from math import *
```

GOOGLE IO

Coding your own modules

- any Python source `wot.py` **is** a module
- just `import wot`
  - file must be in a directory (or zip file) listed in `sys.path`
  - also importable: the bytecode file (`wot.pyc`) which the Python compiler creates the first time you import the source
  - also importable: binary native-code extension files (`wot.pyd`) coded in C and using Python's C API (or other compiled-to-native-code languages and tools: pyrex, Cython, SWIG, ...)
- **note**: Google App Engine only supports `.py` modules!

What's a module?

- a module is a simple object with attributes -- the attributes are the module's "top-level" names
- bound by assignment, or assigning-statements: class, def, import, from
- a module's attributes are also known as "global variables" <u>of the module</u> (there are **no** "global" globals!-)
- a module's attributes can also be bound/unbound/rebound "from the outside" (also known as "monkey patching")
  - not good practice, but sometimes useful for testing with *Mock* and similar design patterns
  - do however consider the *Dependency Injection* DP alternative, cfr e.g. http://www.aleax.it/yt_pydi.pdf

Modules are *singletons*

- the most natural and Pythonic form (automatic singletons)
- the first `import` loads the module (compiling if needed) and executes it, every other import just accesses it, specifically by using `sys.modules[modulename]`
- if you *really* need a class (for overloading, getattr, &c)...:

```
import sys
class _hidden(object): ...
sys.modules[__name__] = _hidden()
```

Packages

- a *package* is a module containing other modules (and maybe sub-packages &c)
- lives in a directory containing a file named `__init__.py`:
  - `__init__.py` is the "module body"
  - often empty (just "tagging" the directory as a package)
  - a package's module are the dir's `.py` files
  - subpackages are subdirs w/`__init__.py`
- the *parent*-directory must be on `sys.path`
- `import foo.bar` or `from foo import bar`

GOOGLE I/O

"Batteries included"

- The standard Python library has > 200 modules (plus many, many more for unit-tests, encoding/decoding, demos)
- some are pure Python modules, some are C-coded ones
  - the App Engine supports ANY pure-Python module,
  - and MOST standard library C-coded modules
    - (with very few specific exceptions, such as: sockets, threads)
  - plus, some specific APIs (Datastore, users, urlfetch, mail)
  - also, any pure-Python module you include w/your app

A peculiar consequence...:

- time needed for an expert programmer to learn well...:
  - Python itself (the Python language): 1-3 days
  - built-ins, special methods, metaprogramming, &c: 2-4 days
  - standard library (absolutely-crucial modules): 10-15 days
  - all the standard library: 30-50 days
  - all third-party offerings: ...!-)

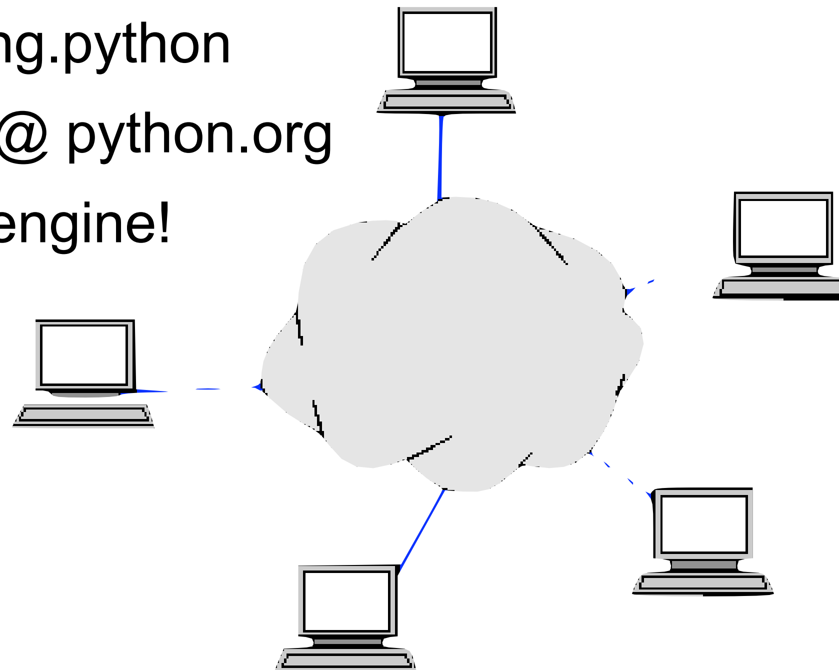
To find more nifty modules...:

- http://cheeseshop.python.org/pypi : 4000+ packages, and counting!
- it's always fun to roll your own, but...;-)
- time to learn all about 4000+ packages: well, hmmm...

GOOGLE I O

Online resources (to get started)...:

- www.python.org
- www.diveintopython.org
- (Usenet) comp.lang.python
- (Mailing list) help @ python.org
- any good search engine!

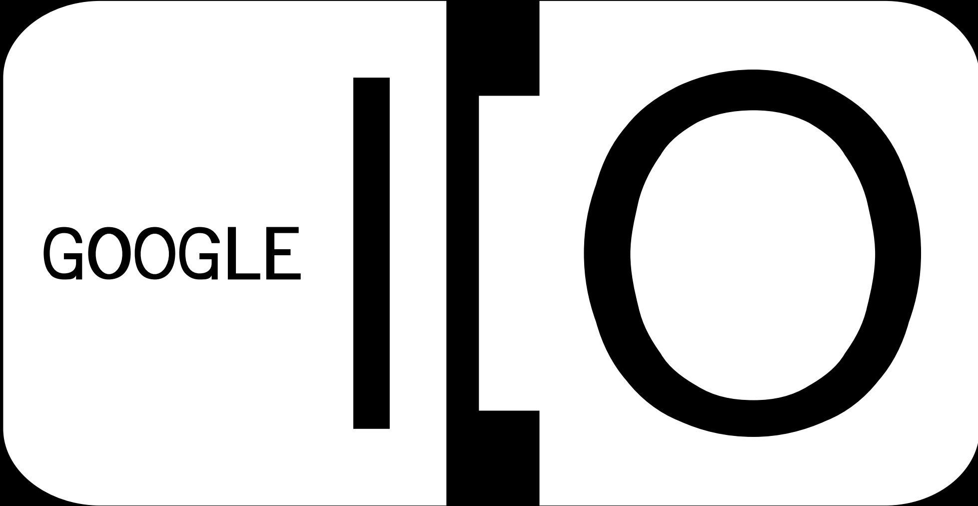


GOOGLE I/O

Questions & Answers

Q?

A!


GOOGLE I O