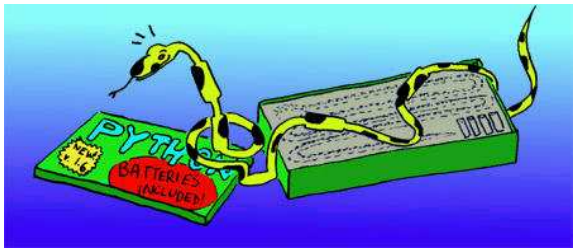
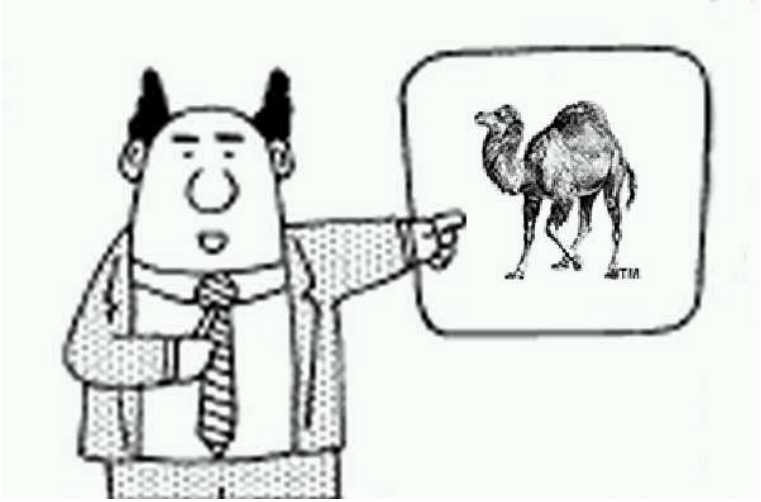
A Lightning Tour of Python



Mike Coleman
Math Dogs, LLC
mkc@mathdogs.com

Python Is Not...

- the choice of PHBs



Python Is...

- interpreted, interactive OO scripting language
- comes from Unix background, runs on Win, Mac, etc
- elegant notation
- readable, maintainable code
- small language--details pushed into libraries
- re Perl, less emphasis on text processing, more on large-scale, general-purpose programming
- introspective
- garbage collection by ref counting (cycle-breaking in v2.0)
- threading (native under Linux)

- my biases (favorite languages, minimalism)

Python Philosophy

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- There should be one--and preferably only one--obvious way to do it.

A Simple Program

In file 'hello.py':

```
#!/usr/bin/python

import sys

# a comment
for n in sys.argv:
    print 'Hello %s!' % n
```

which does this

```
$ ./hello.py Guido Larry
Hello Guido!
Hello Larry!
```

(note block indentation in for loop body)

Indentation

Python style:

```
if  x > 0  :
    if  y > 0  :
        blah()
        blah()

else:
    blah()
```

Indentation

C style:

```
if (x > 0) {
    if (y > 0) {
        blah();
        blah();
    }
}
else
    blah();
```

Interactive Use

The Python interpreter can be used interactively, typically to try things out:

```
$ python
Python 2.0 (#0, Feb 21 2001, 12:48:36)
[GCC 2.95.3 20010125 (prerelease)] on linux2
Type "copyright", "credits" or "license" for more
information.
>>> x = 1
>>> print x
1
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>>
```

Basic Types

- Numbers
  - Int (>= 32 bits)
  - Long (arbitrary precision)
  - Float (double-precision native)
  - Complex (pair of floats)
- None
- Sequences
  - String (plus Unicode strings in 2.0)
  - List (mutable)
  - Tuple (non-mutable)
- Mapping (hash on non-mutable keys)
- Callable types (e.g., functions, lambdas, class methods)
- File
- Module
- Class

Literals

```
int = 1
long = 1234123412341234123412341234L
float = 0.9e10
complex = 1.8 + 3.2J

string = "he said, \"hooray\""
string = 'line one\nline two'
string = """line one
line two
line three
"""
string = r'[a-zA-Z][\w-]*'
```

Literals (cont.)

```
list = []
list = [1, 2, 3]
list = [1, 2, ['foo', bar']]

tuple = ()
tuple = (1,)
tuple = (1, 2, 'baz', [5, 6])

dict = { 'one' : 1, 'two' : 2 }

func = lambda x, y: x + y
x = func(1, 2)    # x gets 3
```

Control Structures - Loops

```
while 1:          # loop forever
    pass          # do nothing

for c in "letters":
    handle_one_letter(c)

for n in [0, 1, 2, 3]:
    handle_one_number(n)

for n in range(4):              # or xrange
    handle_one_number(n)

for name, age in [("Homer", 37), ("Marge, 36)]:
    print name, age
```

Control Structures - Conditionals

```
if x > 0:
    foo(x)
elif x < 0:
    bar(x)
else:
    baz(x)
```

- no switch statement

```
assert x > 0, "uh-oh, x (= %s) not > 0!" % x
```

- assertion checking disabled by -O flag

Functions

```
def draw_square(width, center=(0,0)):
    "draw a square on the display"
    for side in range(4):
        draw_side(side, width, center)
    # ...
    return (time, cost)

#invoked like this:
draw_square(4, (5, 6))

mytime, mycost = draw_square(7)

draw_square(center=(4,2), width=7)
```

Control Structures - Exceptions

```
try:
    f = open('foo')
except IOError, e:
    print 'Unable to open 'foo': ', e
except TypeError, e:
    print "Shouldn't happen: ", e
else:
    # do this if there were no exceptions
    data = f.read()
    f.close()

f = open('foo')
try:
    # do something
finally:
    # do this even if above causes exception
    f.close()
```

- finally and except cannot appear together

Functions (cont.)

```
def printf(format, *args):
    # args is a tuple of all args except the first
    for a in args:
        # handle arg a

def foo(**keyargs):
    # keyargs is a dictionary of keyword->value
pairs
    for k in keyargs.keys():
        # handle keyword arg with
        # key k and arg keyargs[k]
```

- either or both of * and ** can appear in a parameter list

Operators

The usual:

```
+ - * / % ~ >> << & | ^ < > <= >= == !=

[]     # indexing
o.a    # object attribute
**     # power
not and or  # short circuit, 2 or 3 -> 2
x is y, x is not y   # object identity
x in y, x not in y   # contains

"abc" + "def" -> "abcdef"
"abc" * 3 -> "abcabcabc"
[1,2] * 3 -> [1,2,1,2,1,2]
"%d %s %f" % (1, 'foo', 3.5) -> "1 foo 3.500000"
```

- version 2.0 has augmented assignment ops like +=

References

in Python, everything is a reference; this mainly makes a difference only for mutable types (lists, dictionaries):

```
a = [1,2,3,4,5]
b = a
a[0] = 6          # b[0] == 6
c = a[:]          # shallow copy
a[0] = 7          # c[0] == 6
```

- library functions for shallow/deep copying

Operators - Slicing

```
a = [1,2,3,4,5]
a[1] = 6          # a == [1,6,3,4,5]
a[2:4] = [8,9]    # a == [1,6,8,9,5]
a[3:5] = [2,3,4]  # a == [1,6,8,2,3,4]
a[2:] = [0]       # a == [1,6,0]
a[:-1] = [2,3,4]  # a == [2,3,4,0]
del a[1]          # a == [2,4,0]
del a[-2:]        # a == [2]
```

Applicative Functions

```
a = [1,2,3,4,5]
b = [0,1,0,0,1]
def p(x,y):
    return x*y
c = map(p, a, b)
        # c == [0,2,0,0,5]
d = reduce(p, a)
        # d == 120
e = filter(lambda x: x > 3, a)
        # e == [4,5]
```

List Comprehensions (new in 2.0)

```
a = [1,2,3,4]
b = ['a', 'b', 'c']
c = [ (m, 2*n) for n in a for m in b if n > 1 ]
# c == [('a', 4), ('b', 4), ('c', 4),
#       ('a', 6), ('b', 6), ('c', 6),
#       ('a', 8), ('b', 8), ('c', 8)]

# Given the list L, make a list of all strings
# containing the substring S.
S, L = 'ed', ['walked', 'foo', 'barred']
sublist = filter(lambda s, substring=S:
                 string.find(s, substring) != -1,
                 L)

sublist = [ s for s in L if string.find(s, S) !=
-1 ]

# L == ['walked', 'barred']
```

File I/O

```
f = open("foo", "w")   # open for
writing
s = f.read(n)          # read n bytes
s = f.readline()       # read a line
l = f.readlines()      # snarf whole
file
f.write(s)
f.writelines(l)        # write list of
                       # strings
 # plus the other usual functions
```

- standard files are sys.stdin, sys.stdout, sys.stderr
- pickle and shelve serialize to file and dbm

Modules

- means for breaking up code into separate files

```
# file foo.py
x = 1
def bar(y):
    x = x + y

# in other code
import foo
foo.bar(3)
c = foo.x    # gets 4
```

- PYTHONPATH controls module search path
- also could be foo.so, a C extension module
- packages allow grouping of modules into directories

Classes

```
class Account:
    "a simple account class"
    acct_type = 'Basic'     # class variable
    def __init__(self, name):   # method
        self.name = name    # instance variable
        self.__balance = 0  # likewise, hidden
    def deposit(self, amount):
        self.__balance = self.__balance + amount

a = Account('a name')
a.deposit(20)
```

- no class methods
- multiple inheritance

Classes - Operator Overloading

```
class MyNumber:
    "my number class"
    def __init__(self, value):
        self.v = int(value)
    def __add__(self, value):
        return MyNumber(value + self.v)

n1 = MyNumber(3)
n2 = MyNumber(4)
n3 = n1 + n2
```

- __op__ methods for all ops
- other features for full support

Large-Scale Software Development

- classes
  - cooperative hiding
- modules
  - external modules
- packages
- assertions
- PyUnit
- doctest (in v2.0)
- distutils

Libraries

- maybe 150 (?) modules in the standard Python Library.
- for example:
  - math
  - re
  - string
  - struct
  - anydbm (multiple dbm-ish modules)
  - fileinput (like Perl's "while(<>)")
  - os
  - signal
  - rexec
  - xml (standard in v2.0)
- more, less-blessed stuff in the Vaults of Parnassus (like CPAN)

SUBTERFUGUE

- tool to monitor and mess with the reality of programs
- "strace" meets "expect"
- implemented in Python
- several thousand lines of Python
- parts in C as extension module, for speed
- class interface for "tricks" (like plugins)

SUBTERFUGUE

Here is a regular xclock.

- "xclock -update 1"

SUBTERFUGUE - TimeWarp.py (1)

```
class TimeWarp(Trick):
    def __init__(self, options):
        self.options = options
        self.w = 1.0
        self.delta = 0
        self.start_t = time.time()
        if options.has_key('w'):
            self.w = float(options['w'])
            if self.w <= 0:
                sys.exit("error: %s: 'w' must be"
                         + " positive\nusage:%s"
                         % (self.__class__.__name__,
                            self.usage()))
        if options.has_key('now'):
            self.delta = float(options['now']) - self.start_t
        if options.has_key('delta'):
            self.delta = self.delta + float(options['delta'])

    def callmask(self):
        return { 'time' : 1, 'gettimeofday' : 1, 'ftime' : 1,
                 'nanosleep' : 1, 'select' : 1, 'poll' : 1, }
```

SUBTERFUGUE

Here is an xclock run under the TimeWarp trick.

- "sf --trick=TimeWarp:w=50.0 xclock -update 1"

SUBTERFUGUE - TimeWarp.py (2)

```
    def callbefore(self, pid, call, args):
        if call == 'time':
            return (args[0], None, None, None)
        elif call == 'gettimeofday':
            return (args[0], None, None, None)
        elif call == 'select':
            timeout = args[4]
            if timeout:
                if _warp_timeval(pid, self.w, timeout):
                    return (timeout, None, None, None)
        elif call == 'poll':
            if args[2] > 0:
                return (None, None, None, (args[0], args[1],
                                           args[2] / self.w))
        elif call == 'nanosleep':
            oldtimeval = _warp_timeval(pid, self.w, args[0], 1000)
            if oldtimeval:
                return ((oldtimeval, args), None, None, None)
```

SUBTERFUGUE - TimeWarp.py (3)

```
    def callafter(self, pid, call, result, state):
        now = time.time()
        delta_t = now - self.start_t
        now_w = self.start_t + delta_t * self.w + self.delta
        if call == 'time':
            now_wi = int(now_w)
            if state:
                ptrace.pokedata(pid, state, now_wi)
            return now_wi
        elif call == 'gettimeofday' and result != -1:
            _put_timeval(pid, state, now_w)
        elif call == 'select' and result != -1 and state:
            _warp_timeval(pid, 1.0 / self.w, state)
        elif call == 'nanosleep':
            oldtimeval, args = state
            if args[1]:
                _warp_timeval(pid, 1.0 / self.w, args[1], 1000)
                # save and restore it so it doesn't get stepped on
                remaining = _get_timeval_2(pid, args[1])
            _put_timeval_2(pid, args[0], oldtimeval) # here
            if args[1]:
                _put_timeval_2(pid, args[1], remaining)
```

Web Programming - urllib

- generic url interface

```
import urllib
params = urllib.urlencode({'spam': 1, 'eggs': 2,
                           'bacon': 0})
 # GET
f = urllib.urlopen("http://www.musi-cal.com/"
                   + "cgi-bin/query?%s"
                   % params)
 # or POST
f = urllib.urlopen("http://www.musi-cal.com/"
                   + "cgi-bin/query",
                   params)
print f.read()
```

Web Programming - cgi

- basic cgi functionality

```
print "Content-Type: text/html"
print                     # blank line, end of
headers

form = cgi.FieldStorage()
form_ok = 0
if form.has_key("name") and form.has_key("addr"):
    form_ok = 1
if not form_ok:
    print "<H1>Error</H1>"
    print "Please fill in the name and addr
fields."
    return
print "<p>name:", form["name"].value
print "<p>addr:", form["addr"].value
 # further form processing here
```

Internet Protocols

- protocol-specific modules ftplib, httplib, imaplib, poplib, nntplib, smtplib, telnetlib gopherlib

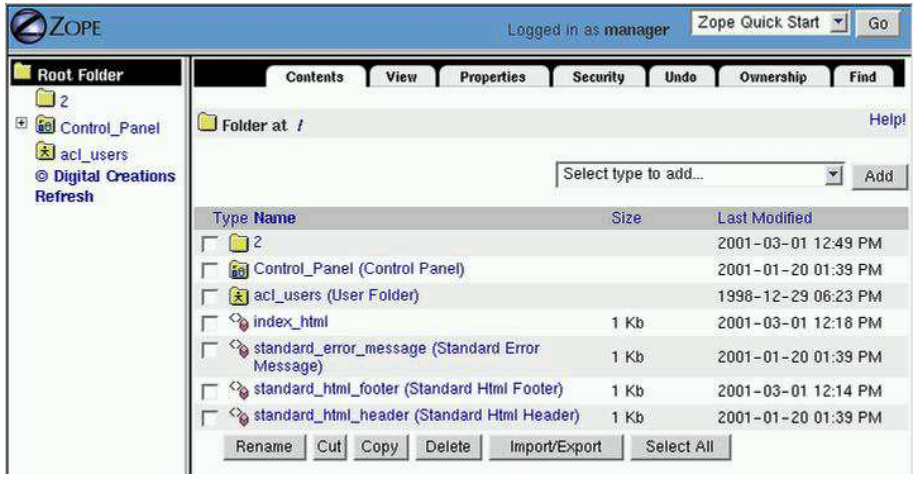
```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.cwi.nl')
>>> ftp.login()  # user anonymous, passwd
user@hostname
>>> ftp.retrlines('LIST')     # list directory
contents
total 24418
drwxrwsr-x   5 ftp-usr  pdmaint     1536 Mar 20
09:48 .
dr-xr-srwt 105 ftp-usr  pdmaint     1536 Mar 21
14:32 ..
-rw-r--r--   1 ftp-usr  pdmaint     5305 Mar 20
09:48 INDEX
>>> ftp.retrbinary('RETR README',
                   open('README', 'wb').write)
'226 Transfer complete.'
```

Web Programming - Higher Level Solutions

- mod_python
  - basic, fast cgi interface
- WebWare
  - application server
  - persistent process sits behind web server
  - like Java app servers
  - alpha/beta

Web Programming - ZOPE

- ZOPE
  - application server
  - ORB, persistent store
  - steep learning curve



A Biased Summary

- Python
  - simple
  - clear
  - minimal
  - "designed not accreted"
  - the way forward?

- Perl
  - more mature (but Python is catching up)
  - larger developer base (but Python is catching up)
  - arcane
  - could go the way of tcl?

A Parting Shot

YODA: Code! Yes. A programmer's strength flows from code maintainability. But beware of Perl. Terse syntax... more than one way to do it... default variables. The dark side of code maintainability are they. Easily they flow, quick to join you when code you write. If once you start down the dark path, forever will it dominate your destiny, consume you it will.

LUKE: Is Perl better than Python?

YODA: No... no... no. Quicker, easier, more seductive.

LUKE: But how will I know why Python is better than Perl?

YODA: You will know. When your code you try to read six months from now.

-- funkster@midwinter.com