Python For Programmers

http://www.aleax.it/goo_py4prog.pdf

This talk's audience

- mildly to very experienced programmers in 1 (or, better, more) of Java, C++, C, Perl, ...
- no previous exposure to Python needed
  - a little bit can't hurt
  - but, if you already know Python well, you will probably end up rather bored!-)
- ready for some very fast & technical parts
  - as well as some more relaxed ones:-)
- tolerant of short names and too-compact formatting (to squeeze in more code/slide!)

What's Python [1]?

- a "very high-level language", with:
  - clean, spare syntax
  - simple, regular, powerful semantics
  - object-oriented, multi-paradigm
  - focus on productivity via modularity, uniformity, simplicity, pragmatism
- a rich standard library of modules
- lots of 3rd-party tools/add-ons
- several good implementations
  - CPython 2.5, pypy 1.0, IronPython 1.1 [.NET], (Jython 2.2 [JVM])

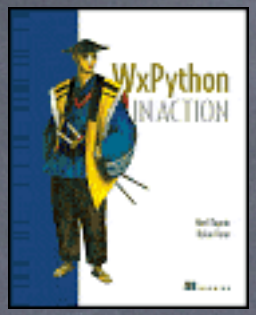
Google

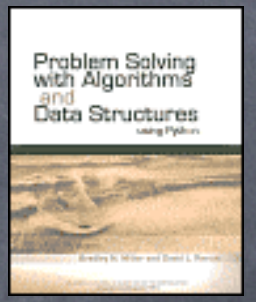
What's Python [2]?

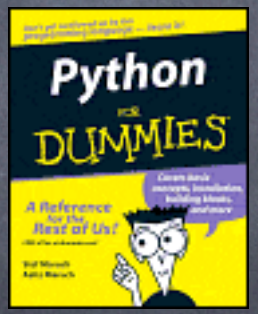
- a strong open-source community
  - many users (both individuals and companies) in all fields
  - the Python Software Foundation
  - sub-communities for special interests
  - many local interest groups
  - sites, newsgroups, mailing lists, ...
- courses, workshops, tutorials, ...
- ... and an inexhaustible array of BOOKS!
  - online-only, paper-only, or both

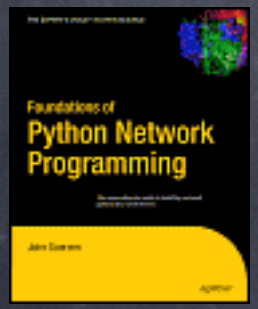
Google

Lots & LOTS of good books









Google

Similarities to Java

- typically compiled to bytecode
  - but: compilation is implicit ("auto-make")
- everything inherits from "object"
  - but: also numbers, functions, classes, ...
  - "everything is first-class"
- uniform object-reference semantics
  - assignment, argument passing, return
- vast, powerful standard library
- garbage collection
- introspection, serialization, threads, ...

Google

Similarities to C++

- multi-paradigm
  - OOP, procedural, generic, a little FP
- multiple inheritance (structural/mixin)
- operator overloading
  - but: not for "plain assignment"
- signature-based polymorphism
  - as if "everything was a template":-)
- lots of choices for all the "side issues"
  - GUI, Web & other network work, DB, IPC and distributed computing, ...

Similarities to C

"Spirit of C" @87% (more than Java/C++...), as per ISO C Standard's "Rationale":

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

(this is the one bit not @ 100% in Python:-)

Python vs Java/C++/C

- typing: strong, but dynamic
  - names have no type: objects have types
- no "declarations" -- just statements
- spare syntax, minimal ornamentation:
  - no { } for blocks, just indentation
  - no ( ) for if/while conditions
  - generally less punctuation
- "everything" is a first-class object
  - classes, functions, methods, modules, ...
- the focus is on high and very high levels

Python fundamentals

- interactive interpreter (prompts: main one is >>> , ... means "statement continuation")
  - to try things out, see expressions' values
- source files such as foo.py
  - auto-compiled to foo.pyc on import
- plain assignment:

  `<name> = <expression>`

  binds or rebinds name to expressions' value
  - names are not "declared"
  - names don't have "a type" (objects do)
- None: placeholder value for "nothing here"

Elementary I/O

- two built-in functions for elementary input:
  - `input(prompt)`: user may enter any Python expression -- returns its value
  - `raw_input(prompt)`: returns string
    - trims trailing \n
- one statement for elementary output:
  - `print` <0+ comma-separated expressions>
  - separates results with a space
  - \n at end (unless trailing comma)
  - `print` itself does no fancy formatting

Some trivial examples

```
x = 23     # name x now means 23
print x    # emits 23
x = 'foo'  # but now it means 'foo' instead
print x    # emits foo
del x      # name x becomes undefined
print x    # is an error ("exception")

y = None   # name y is defined... to None
print y    # emits None
```

Flow Control

- `if <expr>: <indented block>`
  - then, 0+ `elif <expr>: <indented block>`
  - then, optionally: `else: <indented block>`
- `while <expr>: <indented block>`
  - within the block, may have
    - `break`
    - `continue`
  - then, optionally: `else: <indented block>`
- `for <name> in <iterable>:`
  - `break`, `continue`, `else:`, just like `while`

Flow-control examples

```
a = 23
b = 45
if a > b:
  print a, 'is greater than', b
elif a == b:
  print a, 'is equal to', b
else:
  print a, 'is smaller than', b
while a < b:
  print a, b
  a = a * 2
```

Built-in "simple" types

- numbers: int, long, float, complex
  - 23 9437217438928l9 0x17 2.3 4.5+6.7j
  - operators: + - * ** / // % ~ & | ^ << >>
  - built-ins: abs min max pow round sum
- strings: plain and Unicode
  - 'single' "double" r'raw' u"nicode" \n &c
  - operators: + (cat), * (rep), % (format)
    - rich "format language" (like printf)
  - built-ins: chr ord unichr
  - are R/O sequences: len, index/slice, loop
  - methods galore: capitalize, center, ...

Built-in "container" types

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

Sequences

- strings, tuples and lists are sequences (other sequence types are in the 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_: first bound included, last bound excluded (per Koenig's advice:-)
- lists are _mutable_ sequences, so you can _assign_ to an indexing and/or slice
  - assignment to slice can change length
- dicts and sets are not sequences

Comparisons, tests, truth

- equality, identity: `==  !=  is  is not`
- order: `<  >  <=  >=`
- containment: `in  not in`
- "chaining": `3 <= x < 9`
- false: numbers `== 0`, `""`, None, empty containers, `False` (aka `bool(0)`)
- true: everything else, `True` (aka `bool(1)`)
- `not x == not bool(x)` for any x
- and, or "short-circuit" (-> return operand)
- so do built-ins `any`, `all` (-> return a bool)

Exceptions

- Errors (and "anomalies" which aren't errors) "raise exceptions" (instances of Exception or any subtype of Exception)
- Statement `raise` explicitly raises exception
- Exceptions propagate "along the stack of function calls", terminating functions along the way, until they're caught
- Uncaught exceptions terminate the program
- Statement `try/except` may catch exceptions (also: `try/finally`, and its nicer form `with` for "resource allocation is initialization")

iterators and for loops

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

also: (<expr> for i in c <opt.clauses>)
      [<expr> for i in c <opt.clauses>]
("genexp" and "list comprehension" forms)
```

functions

`def <name>(<parameters>): <body>`

- <body> compiled, not yet evaluated
- <parameters>: 0+ local variables, initialized at call time by the <args> passed by caller
- default values may be given (to 0+ trailing parameters) with <name>=<expr> (expr is evaluated only once, when def executes)
- <parameters> may optionally end with *<name> (tuple of arbitrary extra positional arguments) and/or **<name> (dict of arbitrary extra named arguments)

function eg: sum squares

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

Or, more general:

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

Lower-level of abstraction, but also OK:

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

Generators

- functions that use yield instead of return
- each call builds and returns an iterator (object w/method next, suitable in particular for looping on in a for)
- end of function raises StopIteration

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

An unending 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
```

Closures

Exploiting the fact that def is an executable statement that creates a new function object (and also exploiting lexical scoping)...:

```
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 immediately apply a HOF.
(<decorator> may be a name or a call)

Classes ("new-style")

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

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

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

Class instantiation

To create an instance, just call the class:

```
class eg(object):
  cla = []                  # class attribute
  def __init__(self): # inst. initializer
    self.ins = {}           # inst. atttribute
  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
```

Lookup internals

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

```
inst.aname [[whether to call it, or not!]]
   ==> ("descriptors" may alter this...)
```

1. look in inst.__dict__['aname']
2. look in type(inst).__dict__['aname']
3. look in each of type(inst).__bases__
4. try type(inst).__getattr__(inst, 'aname')
5. if everything 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()
```

Properties

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

Now...:

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

Why properties matter

- you never need to "hide" attributes behind getter/setter methods to remain flexible
- just expose interesting attributes directly
- if your next release needs a getter to compute the value, and/or a setter,
  - just code the new methods as needed,
  - and wrap them up into a property
  - all code using your class keeps working!
- down with boilerplate -- never code like:

```
def getFoo(self): return self._foo
```

Operator overloading

- "special methods" names start and end with double underscores -- there are legions....:

```
__new__ __init__ __del__      # init/final.
__repr__ __str__ __int__      # conversions
__lt__ __gt__ __eq__ ...      # comparisons
__add__ __sub__ __mul__ ... # arithmetic
__call__ __hash__ __nonzero_ ...
__getattr__ __setattr__ __delattr__
__getitem__ __setitem__ __delitem__
__len__ __iter__ __contains__
```

- Python calls special methods on the type when you operate on the type's instances

An "unending" 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
```

Builtin functions

- don't call special methods directly: builtin functions do it for you "properly"
- e.g.: abs(x), NOT x.__abs__()
- there are many interesting builtins, e.g.:

```
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 more useful functions and types are in modules in the standard library

Example: index a textfile

```
# build word -> [list of linenumbers] map
indx = {}
with open(filename) as f:
  for n, line in enumerate(f):
    for word in line.split():
      indx.setdefault(word, []).append(n)
# display by alphabetical-ordered word
for word in sorted(indx):
  print "%s:" % word,
  for n in indx[word]: print n,
  print
```

Importing modules

- import modulename
- from some.package import modulename
  - in either case, use modulename.whatever
  - naming shortcuts available, but not recommended (namespaces are good!):
    - may shorten names with as clause:
    - import longmodulename as z
      - then use z.whatever
    - from longmodulename import whatever
    - from longmodulename import *

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 in the current namespace
- handy in interactive sessions, but often unclear in "real" programs -- avoid!
- even more so:

```
from math import *
```

Defining modules

- every Python source file wot.py is a module
- just `import wot`
  - must reside in the import-path
  - ...which is list `path` in stdlib module `sys`, each item a string that names a directory (or zipfile, ...) containing Python modules
  - also importable: bytecode files (wot.pyc), automatically made by the Python compiler when you import a source file
  - also importable: binary extensions (wot.pyd), coded in C (or pyrex, SWIG, ...)

Google

What's in a module?

- a module is a simple object w/attributes
- the attributes of a module are its "top-level" names
- as bound by assignments, or by binding statements: `class`, `def`, `import`, `from`
- module attributes are also known as "global variables" of the module
- may also be bound or rebound "from the outside" (questionable practice, but useful particularly for testing purposes, e.g. in the Mock Object design pattern)

Packages

- a package is a module containing other modules (& possibly sub-packages...)
- lives in a directory with an __init__.py:
  - __init__.py is the "module body"
  - often empty (it then just "marks" the directory as being a package)
  - modules are .py files in the directory
  - subpackages are subdirs w/__init__.py
- parent directory must be in sys.path
- import foo.bar or from foo import bar

"Batteries Included"

- standard Python library (round numbers):
  - 190 plain ("top-level") modules
    - math, sys, os, struct, re, random, gzip...
    - socket, select, urllib, ftplib, rfc822, ...
  - 13 top-level packages w/300 modules
    - bsddb, compiler, ctypes, curses, email ...
  - 115 encodings modules
  - 430 unit-test modules
  - 185 modules in Demo/
  - 165 modules in Tools/

"Other batteries"

- http://cheeseshop.python.org/pypi : 2222 packages registered as of Apr 8, 2007
- Major topics of these 3rd-party extensions:
  - Communications (94)
  - Database (152)
  - Desktop Environment (22)
  - Education (25)
  - Games/Entertainment (39)
  - Internet (359)
  - Multimedia (106)
  - Office/Business (44)
  - Scientific/Engineering (168)
  - Security (44)
  - Software Development (933)
  - System (153)
  - Terminals (12)

3rd-party extensions

- GUIs (Tkinter, wxPython, PyQt, platform-sp)
- SQL DBs (sqlite, gadfly, mysql, postgresql, Oracle, DB2, SAP/DB, Firebird, MSSQL...) and wrappers (SQLObject, SQLAlchemy...)
- computation (numpy and friends, PIL, SciPy, gmpy, mxNumber, MDP, pycripto, ...)
- net & web (mod_python, WSGI, TurboGears, Django, pylons, Quixote, Twisted, ...)
- development environments and tools
- games, multimedia, visualization, ...
- integration w/C, C++, Java, .NET, Fortran...

Google

stdlib: a µm deeper

- some fundamentals: bisect, copy, collections, functools, heapq, inspect, itertools, re, struct, sys, subprocess, threading, Queue...
- testing/debugging: doctest, unittest, pdb, ...
- file and text processing: fileinput, linecache, cStringIO, readline, curses, textwrap, tempfile, codecs, unicodedata, gzip, bz2...
- persistence/DBs: marshal, pickle, shelve, dbm, bsddb, sqlite3 (other DB: 3rd-party)
- time/date: time, datetime, sched, calendar
  - key 3rd-party helpers: pytz, dateutil
- math, cmath, operator, random, decimal
- plus: tons and tons of net/web stuff

Google

GvR's "simple wget"

```
import sys, urllib, os
def hook(*a): print a
for url in sys.argv[1:]:
    fn = os.path.basename(url)
    print url, "->", fn
    urllib.urlretrieve(url, fn, hook)
```

A multi-threaded wget

```
import sys, urllib, os, threading, Queue
q = Queue.Queue()
class Retr(threading.Thread):
  def run(self):
    self.setDaemon(True)
    def hook(*a): print '%s: %s' % (fn, a)
    while True:
      url = q.get()
      fn = os.path.basename(url)
      print url, "->", fn
      urllib.urlretrieve(url, fn, hook)
for i in range(10): Retr().start()
for url in sys.argv[1:]: q.put(url)
```

some stdlib packages

- compiler: parse and compile Python code
- ctypes: access arbitrary DLL/.so
- distutils: build/distribute/install packages
- email: parse/create RFC2822-related files
- hotshot: one of several Python profilers
- idlelib: support for IDLE & other IDEs
- logging: guess what
- xml: XML handling (subpackages: xml.sax, xml.dom, xml.etree, xml.parsers)

ctypes toy example

```
if sys.platform == 'win32':
  libc = ctypes.cdll.msvcrt
elif sys.platform == 'darwin':
  libc = ctypes.CDLL('libc.dylib')
else:
  libc = ctypes.CDLL('libc.so.6')
nc = libc.printf("Hello world\n")
assert nc == 12
```