Python for Programmers

Alex Martelli (aleaxit@gmail.com)

http://www.aleax.it/Python/py4prog.pdf

Python is (lotsa cool buzzwords...)

- very high level language (VHLL)
- clean, spare syntax
- simple, regular semantics
- extreme modularity
- high productivity
- open-source, cross-platform
- object-oriented
- ...

Python is (just like Java...)

- compiler->bytecode + VM/interpreter
  - but: compilation is implicit ("auto-make")
  - uses own VM (or: JVM, Parrot, MS CLR...)
  - JIT/SC compilers (psyco, starkiller, pypy)
- everything inherits from `object`
- uniform "object-reference" semantics:
  - assignment, argument passing, return ...
  - also for numbers (immutable, like strings) -- more uniform
- vast, powrful standard library
- introspection/reflection, serialization, threads, ...

Python is (just like C++...)

- multi-paradigm
  - object-oriented, procedural, generic, FP
- multiple inheritance (structural, mix-in)
- operator overloading
- signature-based polymorphism
  - as if "everything was a template"... but with simple, clean syntax
- even too many choices for:
  - GUI, Web server framework, database access, COM/Corba/...

Python is (just like C...)

"the spirit of C"... 87% (more than Java/C++...)
as defined 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)
   - not 100% in Python, but, e.g.: float == whatever the current platform offers

Python is (very different...)

- typing is strong but dynamic
  - objects have types (strong ones), names don't
  - no declarations: just statements
- clean syntax, minimal ornamentation
  - blocks use no { } -- just indentation
  - if and while use no ( )
- just about everything is a **first-class object**
  - including: classes, functions, methods, modules, packages, ...
- focus on high/very-high level
  - metaclasses, generators, descriptors, ...

Python versions / releases

- **Classic** Python: now 2.4 (2.5 "cooking")
  - implemented in C ISO (1990 level)
- **Jython**: now 2.2 (2.3/2.4 "almost ready")
  - implemented in "100% pure Java"
  - deployment like Java, on a JVM
  - can use/extend/implement arbitrary Java classes/interfaces, compile into Java
- Others: experimental or research level
  - Stackless, IronPython (.NET/MS CLR), Vyper (in O'CAML), **pypy** (EU-financed research), ...

Python resources

- http://www.python.org
  - the hub of the Python community
- news:comp.lang.python
  - best place for general discussions
- http://www.jython.org
- http://www.google.com
  - no, **really!!!**

Python Fundamentals

- interactive interpreter (text-mode, IDLE, ...)
  - to try trings out, or as a calculator
  - prompt >>>, shows expressions' values
- program files (afile.py, afile.pyc, ...)
  - for most uses
  - automatic compilation (on first **import**)
- assignment (simplest form):

  `name = <arbitrary expression>`
  - creates name if needed, binds it to the value
  - names aren't declared, and, per se, have no type (objects do have a type)

Assigment and print

```
myvar = 'hello'                 # creates name
myvar = 23                      # rebinds name
quest = ans = 42
myvar, ans = ans, myvar
print myvar, ans, quest
42, 23, 42

if myvar<20: myvar = 10     # doesn't execute
if myvar>40: myvar = 50     # executes
print myvar
50
```

Conditionals

```
if ques>30:                 # 'if' “guards”
   ques = 20                #   a block that is
   x = 33                   #   right-indented
else:                       # 'else' is optional
   x = 99                   #   indenting again

if x<30: myvar = 10         # not satisfied
elif x<40: myvar = 20       # satisfied
elif x<40: myvar = 40       # not even tested
else: myvar = 80            # ditto

print x, ques, myvar
33 20 20
```

Comparing, tests, truth

```
equality, identity:      ==  !=  is  is not
order:                   <  >  <=  >=
containment:             in  not in
comparisons “chain”:     5<x<9  a==b==c

false: all numbers == 0, “”, None,
       all empty containers
true:  everything else
the bool type (subtype of int):
    False==0,  True==1
    bool(x)   True o False (any x)
not x == not bool(x)          (any x)
```

and/or "short-circuit"

and & or "short-circuit", and return either of their operands as their result:

```
x = y and z  is like: if y: x=z
                      else: x=y
x = y or  z  is line: if y: x=y
                      else: x=z

print 0 and 0j, 0j and 0, 0 or 0j, 0j or 0
0 0j 0j 0
```

Numbers

```
int (32 bit) and long (unbounded):
print 2**100
1267650600228229401496703205376

float (usually 64 bit, IEEE):
print 2**100.0
1.26765060023e+030

complex (two float attributes, .real & .imag):
print 2**100.0j
(0.980130165912+0.19835538276j)
```

Arithmetic

```
add, sub, mul, pow:  +  -  *  **
div (true, trunc, remainder):  /  //  %
bit-wise and shifts:    ~  &  |  ^ <<  >>
built-in: abs divmod max min pow round sum
type conversions:  complex float int long

see also modules: math, cmath, operator

import math, cmath
print math.sin(0.1)
0.0998334166468
print cmath.sin(0.3+0.2j)
(0.301450338429+0.192343629802j)
```

Loops

```
while myvar>10: myvar -= 7
print myvar
3
for i in 0, 1, 2, 3: print i**2
0 1 4 9
for i in range(4): print i**2 # u.b. exc
0 1 4 9
while & for usually control blocks
    indented (standard: 4 spaces)
    may contain break and/or continue
may have an else (==”natural exit”)
```

Files (eg: file copy)

```
fin = file('in', 'r')       # or file('in')
fou = file('ou', 'w')       # 'a', 'r+', 'wb'..

fou.write(fin.read())                   # or...
data = fin.read(); fou.write(data)        # or...
fou.writelines(fin.readlines())           # or...
for line in fin: fou.write(line)

fin.close()       # a good idea, but it's not
fou.close()       # mandatory (in CPython)
```

Strings (eg: file listing)

```
for n, r in enumerate(fin):
    fou.write('Line %s: %s' % (n+1, r))

or, single instruction (“list comprehension”):

fou.writelines([ 'Line %s: %s' % (n+1, r)
                 for n, r in enumerate(fin) ])

   or (Python 2.4) “generator expression”:

   fou.writelines('Line %s: %s' % (n+1, r)
                  for n, r in enumerate(fin))
```

Strings are sequencese

```
for c in 'ciao': print c,
c i a o

print len('cip'), 'i' in 'cip', 'x' in 'cip'
3 True False

print 'Amore'[2], 'Amore'[1:4], 'Amore'[::-1]
o mor eromA

print 'ci'+'ao', 'cip'*3, 4*'pic'
ciao cipcipcip picpicpicpic
```

Lists (array-like)

```
x = [1, 2, 'beep', 94]

x[1] = 'plik'          # lists are mutable
print x
[1, 'plik', 'beep', 94]

x[1:2] = [6,3,9]       # slice assignment
print x
[1, 6, 3, 9, 'beep', 94]

print [it*it for it in x[:4]]
[1, 36, 9, 81]
```

Lists are sequences

```
print x
[1, 6, 3, 9, 'beep', 94]

print len(x), 6 in x, 99 in x
6 True False

print x[2], x[1:5], x[1::2]
3 [6, 3, 9, 'beep'] [1, 9, 94]

print [1]+[2], [3, 4] * 3
[1, 2] [3, 4, 3, 4, 3, 4]
```

Indexing and slicing

```
x = 'helloworld'
print x[1], x[-3]
e r

print x[:2], x[2:], x[:-3], x[-3:]
he lloworld hellowo rld

print x[2:6], x[2:-3], x[5:99]
llow llowo world

print x[::2], x[-3:4:-1]
hlool row
```

Packing and unpacking

```
x = 2, 3, 6, 9          # tuple (immutable)
print x
(2, 3, 6, 9)

a, b, c, d = x          # unpacking
print c, d, b, a
6 9 3 2

RED, YELLOW, GREEN = range(3) # enum-like
a, b, c, d = 'ciao' # unpacking
print c, d, b, a
a o i c
```

String methods

```
x = 'ciao mondo'
print x.upper(), x.title(), x.isupper()
CIAO MONDO Ciao Mondo False
print x.find('o'), x.count('o'), x.find('z')
3 3 -1
print x.replace('o', 'e')
ciae mende
print ', '.join(x.split())
ciao, mondo
print x.join('bah')
bciao mondoaciao mondoh
```

List methods

```
x = list('ciao')
print x
['c', 'i', 'a', 'o']
print x.sort()
None
print x
['a', 'c', 'i', 'o']
print ''.join(x)
acio
x.append(23); print x
['a', 'c', 'i', 'o', 23]
```

List comprehensions

```
[ <expr> for v in seq ]
[ <expr> for v in seq if <cond> ]

def divisors(x):
  “”” nontrivial divisors of integer x “””
  return [ n for n in range(2,x) if x%n==0 ]
def prime(x):
  “”” is x prime? “””
  return not divisors(x)
# squares of primes between 3 and 33
print [x*x for x in range(3,33) if prime(x)]
[9, 25, 49, 121, 169, 289, 361, 529, 861, 961]
```

Reference semantics

```
x = [a', 'b', 'c']
y = x
x[1] = 'zz'
print x, y
['a', 'zz', 'c'] ['a', 'zz', 'c']

# when you want a copy, ask for a copy:
y = list(x)   # or x[:], x*1, copy.copy(x)...
x[2] = 9999
print x, y
['a', 'zz', 9999] ['a', 'zz', 'c']
```

Dicts are mappings

```
x = {1:2, 'beep':94}
x[1] = 'plik'          # dicts are mutable
print x
{1:'plik', 'beep':94}
x['z'] = [6, 3, 9]   # adding an element
print x              # NB: order is arbitrary
{1:'plik', 'z':[6, 3, 9], 'beep':94}

# building a dict from a seq. of pairs:
print dict([ (i, i*i) for i in range(6) ])
{0:0, 1:1, 5:25, 2:4, 3:9, 4:16}
```

Dict keys

```
Must be hashable (usually, immutable):
x = {}
x[[1,2]] = 'a list cannot be a key'
TypeError: list objects are unhashable
x[{1:2}] = 'a dict cannot be a key'
TypeError: dict objects are unhashable
x[1,2] = 'a tuple is OK' # tuples are hashable
x[0j] = 'a complex is OK' # numbers too, BUT:
print x[0.0], x[0]  # 0==0.0==0.j, so...:
a complex is OK a complex is OK
```

Dicts aren't sequences

```
print x
{1:'plik', 'z':[6. 3, 9], 'beep':94}
for k in x: print k,
1 z beep
print x.keys(), x.values()
[1, 'z', 'beep'] ['plik', [6, 3, 9], 94]
print x.items()
[(1,'plik'), ('z',[6,3,9]), ('heep',94)]
# same as: zip(x.keys(), x.values())
print len(x), 'z' in x, 99 in x
3 True False
```

Dict methods

```
print x.get(1), x.get(23), x.get(45, 'bu')
plik None bu
print x.setdefault(1, 'bah')
plik
print x.setdefault(9, 'wo')
wo
print x
{1:'plik', 9:'wo', 'z':[6,3,9], 'beep':94}
```

E.g.: indexing a textfile

```
# build map words->line numbers
index = {}
for n, line in enumerate(file('xx.txt')):
    for word in line.split():
        index.setdefault(word, []).append(n)
# show index in alphabetical order
words = index.keys(); words.sort()
for word in words:
    print “%s: “ % word,
    for n in index[word]: print n,
    print
```

C++ standard equivalent

```
#include <string>
#include <iostream>
#include <sstream>
#include <map>
#include <vector>
typedef std::vector<int> svi;

int main() {
  std::map<std::string, svi> idx;
  std::string line;
  int n = 0;
  while(getline(std::cin, line)) {
    std::istringstream sline(line);
    std::string word;
    while(sline>>word)
      idx[word].push_back(n);
    n += 1;
  }
  for(std::map<std::string, svi>::iterator i = idx.begin(); i != idx.end(); ++i) {
    std::cout<< i->first << ": ";
    for(svi::iterator j = i->second.begin(); j != i->second.end(); ++j)
      std::cout<< ' ' << *j;
    std::cout<< "\n";
  }
  return 0;
}
```

on KJB, 4.4MB text:

C++ 8.5/17.40 sec (optimized, 7.38/15.01)

Python 5.4/11.22 (optimized, 3.85/8.09)

Functions

```
def sumsquares(x, y): return x*x+y*y
print sumsquares(1, 2)
5
def sq1(x, y=1): return sumsquares(x, y)
print sq1(1, 2), sq(3)
5 10
def sq(*args):          # variable # of args
    total = 0
    for a in args: total += a*a
    return total
# in 2.4: return sum(a*a for a in args)
```

Lexical closures

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

add23 = makeAdder(23)
add42 = makeAdder(42)

print add23(100),add42(100),add23(add42(100))
123 142 165
```

Classes

```
class sic:
    cla = []                # class attribute
    def __init__(self): # constructor
        self.ins = {}   # instance attribute
    def meth1(self, x):
        self.cla.append(x)
    def meth2(self, y, z):
        self.ins[y] = z
# call the class to create instances:
es1 = sic()
es2 = sic()
```

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, es1.ins is es2.ins
True False
```

Subclasses

```
class sub(sic):
    def meth2(self, x, y=1):   # override
        sic.meth2(self, x, y)  # supercall

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

class override_data(sub):
    cla = repetita()
```

"New-style" classes

```
class ns(object):
    def ciao(): return 'hello'
    ciao = staticmethod(ciao)
    def hi(cls): return 'hi,%s'%cls.__name__
    hi = classmethod(hi)
class sn(ns): pass
print ns.ciao(), sn.ciao(), ns.hi(), sn.hi()
hello hello hi,ns hi,sn
x = ns(); y = sn()
print x.ciao(), y.ciao(), x.hi(), y.hi()
hello hello hi,ns hi,sn
```

Decorators

```
class ns(object):
    @staticmethod
    def ciao(): return 'hello'
    @classmethod
    def hi(cls): return 'hi,%s'%cls.__name__
In general, in Python 2.4:
@whatever
def f(...
    ...
like (construct good in 2.4, 2.3, 2.2, ...):
def f(...
    ...
f = whatever(f)
```

Properties

```
class makeeven(object):
    def __init__(self, num): self.num = num
    def getNum(self): return self.x * 2
    def setNum(self, num): self.x = num // 2
    num = property(getNum, setNum)

x = makeeven(23); print x.num
22
x.num = 27.12; print x.num
26.0
```

Why properties matter

expose instance attributes, e.g.:

```
  x.foo = y.bar + z.baz
```

w/all encapsulation benefits, as, at need:

```
  def setFoo(self, afoo): ...
  foo = property(getFoo, setFoo)
```

so, no need for getThis, setThat; mostly, this avoids horrid boilerplate such as:

```
  def getThisOneToo(self):
      return self._this_one_too
```

Operator overloading

```
class liar(object):
    def __add__(self, other): return 23
    def __mul__(self, other): return 42
x = liar()
print x+5, x+x, x+99, x*12, x*None, x*x
23 23 23 42 42 42
```

May overload: arithmetic, indexing, slicing, attribute access, length, truth, creation, initialization, copy (shallow/deep), ...
But, NOT “assignment to a name” (no assignment TO objects, but OF objects TO names)

Exceptions

```
In error cases, Python raises exceptions:
x = [1, 2, 3]; x[3] = 99
Traceback (most recent call last):
   ...
IndexError: list assignment out of range
I can define new exception classes:
class WeirdError(Exception): pass
I can explicitly raise exceptions:
raise WeirdError, 223961
```

Exception handling

```
try:
    x[n] = avalue
except IndexError:
    x.extend((n-len(x))*[None])
    x.append(avalue)
else:
    print “all fine, no sweat”

f = file('somefile')
try: process(f)
finally: f.close()
```

Iterators

An iterator encapsulates any loop's logic
    can always use a simple for instead of
    any complicated loop
Create by calling the iter built-in on any
    iterable (for does that implicitly)
An iterator supplies a method 'next' which
    returns the next item at each call
When no items are left, next raises exception
    StopIteration (exception, but not error!)

A non-terminating iterator

```
class fiboniter(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 fiboniter():
    if rabbits > 100: break
    print rabbits,
1 1 2 3 5 8 13 21 34 55 89
```

A terminating iterator

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

Generators build iterators

```
def fiboniter_gen(max=None):
    r, i, j = 0, 1, 1
    while max is None or r <= max:
        if r: yield r
        r, i, j = i, j, i+j

for rabbits in fiboniter_gen(100):
    print rabbits,
1 1 2 3 5 8 13 21 34 55 89
```

The enumerate generator

```
# it's built-in, but, if it wasn't...:
def enumerate(iterable):
    n = 0
    for item in iterable:
        yield n, item
        n += 1

print list(enumerate('ciao'))
[(0, 'c'), (1, 'i'), (2, 'a'), (3, 'o')]
```

2.4: generator expressions

```
# usually known as genexp
X = 'ciao'
x = ((n*n, x+x) for n,x in enumerate(X))
print list(x)
[(0,'cc'),(1,'ii'),(4,'aa'),(9,'oo')]
```

Like a list comprehension, but one step at a time ("lazy" evaluation)

The itertools module

Supplies high-level building blocks to build or transform iterators (including generators, genexp, etc).
E.g....:

```
def enumerate(seq):
    import itertools as it
    return it.izip(it.count(), seq)
```

Offers an "abstraction benefit" (the reverse of the "abstraction penalty" that some other languages may typically exact)

Importing modules

```
import math   # standard library module
print math.atan2(1, 3)
0.321750554397
print atan2(1, 3)
Traceback (most recent call last):
   ...
NameError: name 'atan2' is not defined
atan2 = math.atan2
print atan2(1, 3)
0.321750554397
# or, as a shortcut: from math import atan2
```

Defining modules

Even simpler...:

- any Python source wot.py is a module
- import with import wot
- must be in a dir/zipfile in sys.path
- sys.path.append('/addsome/dir')
- module attributes are names it defines
- aka "global variables" of the module
- NB: classes, functions "variables" too!

Packages

A package is a module which may contain other modules (and recursively other packages too)

- a directory including a file __init__.py
- __init__.py is the “body”, may be empty
- package's modules are files in directory
- sub-packages are subdirectories
  - as long as they have an __init__.py
- import and use w/“structured names”:
  - import email.MIMEImage
  - from email import MIMEImage

"Batteries included"

```
standard Python library (roughly)...:
180 mod.: math, sys, os, sets, struct, re,
    random, pydoc, gzip, threading, socket,
    select, urllib, ftplib, rfc822, copy,
    pickle, SimpleXMLRPCServer, telnetlib...
8 packages w/70 more mod.: bsddb, compiler,
  curses, distutils, email, logging, xml...
80 codec mod., 280 unit-test, 180 demo
180 mod. in Tools (12 major+60 minor)

...but wait, there's more...
```

More batteries: GUI, DB

GUI:
Tkinter (w/ Tcl/Tk)
wxPython (w/ wxWidgets)
PyQt (w/ Qt; also, PyKDE)
Pythonwin (w/ MFC – Windows only)
Cocoa (Mac OS X only)
PyGTK, PyUI, anygui, fltk, FxPy, EasyGUI, ...

DB (relational):
Gadfly, PySQLite, MkSQL (w/ Metakit), MySQL, PostgreSQL, Oracle, SAP/DB, Firebird, DB2...

More batteries: computing

Numeric (& numarray)
PIL (image processing)
SciPy
weave (inline, blitz, ext_tools)
fft, ga, special, integrate, interpolate, ...
plotting: chaco, plt, xplt, gplt, ...
gmpy (multiprecision, wraps GMP)
pycrypto

More batteries: networking

Integration w/ Apache:
mod_python
Higher-level web frameworks (Webware,
    Quixote, CherryPy, ...)
“Application servers”/CMS (Zope, Plone)
...and an asynchronous networking framework
    ever more powerful and complete:
Twisted (w/ protocols, web templating, GUI
    integration, wrapping of thread/processes/
    DB, persistence, configuration, ...)

Integration with C/C++/...

Python C API
SWIG
Boost Python, sip, CXX/SCXX (C++)
PyObjC (Objective-C)
pyrex
pyfort, f2py

COM, XPCOM, Corba, AppleEvents,...

Integration with Java

Jython: complete, trasparent integration (and just that -- uses a JVM, not classic Python runtime): import Java classes, implement Java interfaces, generates JVM bytecode, Java-accessible interpreter/compiler, ...

JPE: “arm's length” integration of classic Python + Java

(and similarly for IronPython w/ C# & friends, within .NET / Mono CLR)