Python

Henning Schulzrinne
Department of Computer Science
Columbia University

(based on tutorial by Guido van Rossum)

Introduction

- Most recent popular (scripting/ extension) language
  - although origin ~1991
- heritage: teaching language (ABC)
  - Tcl: shell
  - perl: string (regex) processing
- object-oriented
  - rather than add-on (OOTcl)

Python philosophy

- Coherence
  - not hard to read, write and maintain
- power
- scope
  - rapid development + large systems
- objects
- integration
  - hybrid systems

Python features

Lutz, *Programming Python*

no compiling or linking	rapid development cycle
no type declarations	simpler, shorter, more flexible
automatic memory management	garbage collection
high-level data types and operations	fast development
object-oriented programming	code structuring and reuse, C++
embedding and extending in C	mixed language systems
classes, modules, exceptions	"programming-in-the-large" support
dynamic loading of C modules	simplified extensions, smaller binaries
dynamic reloading of C modules	programs can be modified without stopping

Python features

Lutz, *Programming Python*

universal "first-class" object model	fewer restrictions and rules
run-time program construction	handles unforeseen needs, end-user coding
interactive, dynamic nature	incremental development and testing
access to interpreter information	metaprogramming, introspective objects
wide portability	cross-platform programming without ports
compilation to portable byte-code	execution speed, protecting source code
built-in interfaces to external services	system tools, GUIs, persistence, databases, etc.

Python

- elements from C++, Modula-3 (modules), ABC, Icon (slicing)
- same family as Perl, Tcl, Scheme, REXX, BASIC dialects

Uses of Python

- shell tools
  - system admin tools, command line programs
- extension-language work
- rapid prototyping and development
- language-based modules
  - instead of special-purpose parsers
- graphical user interfaces
- database access
- distributed programming
- Internet scripting

What not to use Python (and kin) for

- most scripting languages share these
- not as efficient as C
  - but sometimes better built-in algorithms (e.g., hashing and sorting)
- delayed error notification
- lack of profiling tools

Using python

- /usr/local/bin/python
  - `#! /usr/bin/env python`
- interactive use

```
Python 1.6 (#1, Sep 24 2000, 20:40:45)  [GCC 2.95.1 19990816 (release)] on sunos5
Copyright (c) 1995-2000 Corporation for National Research Initiatives.
All Rights Reserved.
Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
>>>
```

- `python –c` *`command`* `[`*`arg`*`] ...`
- `python –i` *`script`*
  - read script first, then interactive

Python structure

- modules: Python source files or C extensions
  - import, top-level via from, reload
- statements
  - control flow
  - create objects
  - indentation matters – instead of {}
- objects
  - everything is an object
  - automatically reclaimed when no longer needed

First example

```
#!/usr/local/bin/python
# import systems module
import sys
marker = '::::::'
for name in sys.argv[1:]:
  input = open(name, 'r')
  print marker + name
  print input.read()
```

Basic operations

- Assignment:
  - `size = 40`
  - `a = b  = c = 3`
- Numbers
  - integer, float
  - complex numbers: `1j+3, abs(z)`
- Strings
  - `'hello world', 'it\'s hot'`
  - `"bye world"`
  - continuation via \ or use """ long text """"

String operations

- concatenate with + or neighbors
  - `word = 'Help' + x`
  - `word = 'Help' 'a'`
- subscripting of strings
  - `'Hello'[2]` → 'l'
  - slice: `'Hello'[1:2]` → 'el'
  - `word[-1]` → last character
  - `len(word)` → 5
  - immutable: cannot assign to subscript

Lists

- lists can be heterogeneous
  - `a = ['spam', 'eggs', 100, 1234, 2*2]`
- Lists can be indexed and sliced:
  - `a[0]` → spam
  - `a[:2]` → ['spam', 'eggs']
- Lists can be manipulated
  - `a[2] = a[2] + 23`
  - `a[0:2] = [1,12]`
  - `a[0:0] = []`
  - `len(a)` → 5

Basic programming

```
a,b = 0, 1
# non-zero = true
while b < 10:
  # formatted output, without \n
   print b,
  # multiple assignment
   a,b = b, a+b
```

Control flow: if

```
x = int(raw_input("Please enter #:"))
if x < 0:
  x = 0
  print 'Negative changed to zero'
elif x == 0:
  print 'Zero'
elif x == 1:
  print 'Single'
else:
  print 'More'
```

- no case statement

Control flow: for

```
a = ['cat', 'window', 'defenestrate']
for x in a:
  print x, len(x)
```

- no arithmetic progression, but
  - `range(10)` → [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  - `for i in range(len(a)):`
    `print i, a[i]`
- do not modify the sequence being iterated over

Loops: break, continue, else

- break and continue like C
- else after loop exhaustion

```
for n in range(2,10):
  for x in range(2,n):
    if n % x == 0:
      print n, 'equals', x, '*', n/x
      break
  else:
    # loop fell through without finding a factor
    print n, 'is prime'
```

Do nothing

- pass does nothing
- syntactic filler

```
while 1:
    pass
```

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to n."""
  a, b = 0, 1
  while b < n:
    print b,
    a, b = b, a+b

>>> fib(2000)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables

Functions: default argument values

```
def ask_ok(prompt, retries=4,
  complaint='Yes or no, please!'):
  while 1:
    ok = raw_input(prompt)
    if ok in ('y', 'ye', 'yes'): return 1
    if ok in ('n', 'no'): return 0
    retries = retries - 1
    if retries < 0: raise IOError,
  'refusenik error'
    print complaint

>>> ask_ok('Really?')
```

Keyword arguments

- last arguments can be given as keywords

```
def parrot(voltage, state='a stiff', action='voom',
  type='Norwegian blue'):
  print "-- This parrot wouldn't", action,
  print "if you put", voltage, "Volts through it."
  print "Lovely plumage, the ", type
  print "-- It's", state, "!"

parrot(1000)
parrot(action='VOOOM', voltage=100000)
```

Lambda forms

- anonymous functions
- may not work in older versions

```
def make_incrementor(n):
    return lambda x: x + n

f = make_incrementor(42)
f(0)
f(1)
```

List methods

- `append(x)`
- `extend(L)`
  - append all items in list (like Tcl lappend)
- `insert(i,x)`
- `remove(x)`
- `pop([i]), pop()`
  - create stack (FIFO), or queue (LIFO) → pop(0)
- `index(x)`
  - return the index for value *x*

List methods

- `count(x)`
  - how many times x appears in list
- `sort()`
  - sort items in place
- `reverse()`
  - reverse list

Functional programming tools

- `filter(`*`function, sequence`*`)`

```
def f(x): return x%2 != 0 and x%3 0
filter(f, range(2,25))
```

- `map(`*`function, sequence)`*
  - call function for each item
  - return list of return values
- `reduce(`*`function, sequence`*`)`
  - return a single value
  - call binary function on the first two items
  - then on the result and next item
  - iterate

List comprehensions (2.0)

- Create lists without `map()`, `filter()`, `lambda`
- = expression followed by for clause + zero or more for or of clauses

```
>>> vec = [2,4,6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [{x: x**2} for x in vec}
[{2: 4}, {4: 16}, {6: 36}]
```

List comprehensions

- cross products:

```
>>> vec1 = [2,4,6]
>>> vec2 = [4,3,-9]
>>> [x*y for x in vec1 for y in vec2]
[8,6,-18, 16,12,-36, 24,18,-54]
>>> [x+y for x in vec1 and y in vec2]
[6,5,-7,8,7,-5,10,9,-3]
>>> [vec1[i]*vec2[i] for i in
  range(len(vec1))]
[8,12,-54]
```

List comprehensions

- can also use `if`:

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

del – removing list items

- remove by index, not value
- remove slices from list (rather than by assigning an empty list)

```
>>> a = [-1,1,66.6,333,333,1234.5]
>>> del a[0]
>>> a
[1,66.6,333,333,1234.5]
>>> del a[2:4]
>>> a
[1,66.6,1234.5]
```

Tuples and sequences

- lists, strings, **tuples**: examples of *sequence* type
- tuple = values separated by commas

```
>>> t = 123, 543, 'bar'
>>> t[0]
123
>>> t
(123, 543, 'bar')
```

Tuples

- Tuples may be nested

```
>>> u = t, (1,2)
>>> u
((123, 542, 'bar'), (1,2))
```

- kind of like structs, but no element names:
  - (x,y) coordinates
  - database records
- like strings, immutable → can't assign to individual items

Tuples

- Empty tuples: ()

```
>>> empty = ()
>>> len(empty)
0
```

- one item → trailing comma

```
>>> singleton = 'foo',
```

Tuples

- sequence unpacking → distribute elements across variables

```
>>> t = 123, 543, 'bar'
>>> x, y, z = t
>>> x
123
```

- packing always creates tuple
- unpacking works for any sequence

Dictionaries

- like Tcl or awk associative arrays
- indexed by keys
- keys are any immutable type: e.g., tuples
- but not lists (mutable!)
- uses 'key: value' notation

```
>>> tel = {'hgs' : 7042, 'lennox': 7018}
>>> tel['cs'] = 7000
>>> tel
```

Dictionaries

- no particular order
- delete elements with del

```
>>> del tel['foo']
```

- keys() method → unsorted list of keys

```
>>> tel.keys()
['cs', 'lennox', 'hgs']
```

- use has_key() to check for existence

```
>>> tel.has_key('foo')
0
```

Conditions

- can check for sequence membership with `is` and `is not`:

```
>>> if (4 in vec):
...  print '4 is'
```

- chained comparisons: a less than b AND b equals c:

```
a < b == c
```

- and and or are short-circuit operators:
  - evaluated from left to right
  - stop evaluation as soon as outcome clear

Conditions

- Can assign comparison to variable:

```
>>> s1,s2,s3='', 'foo', 'bar'
>>> non_null = s1 or s2 or s3
>>> non_null
foo
```

- Unlike C, no assignment within expression

Comparing sequences

- unlike C, can compare sequences (lists, tuples, ...)
- lexicographical comparison:
  - compare first; if different → outcome
  - continue recursively
  - subsequences are smaller
  - strings use ASCII comparison
  - can compare objects of different type, but by type name (list < string < tuple)

Comparing sequences

```
(1,2,3) < (1,2,4)
[1,2,3] < [1,2,4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1,2,3) == (1.0,2.0,3.0)
(1,2) < (1,2,-1)
```

Modules

- collection of functions and variables, typically in scripts
- definitions can be imported
- file name is module name + .py
- e.g., create module `fibo.py`

```
def fib(n): # write Fib. series up to n
   ...
def fib2(n): # return Fib. series up to n
```

Modules

- import module:

```
import fibo
```

- Use modules via "name space":

```
>>> fibo.fib(1000)
>>> fibo.__name__
'fibo'
```

- can give it a local name:

```
>>> fib = fibo.fib
>>> fib(500)
```

Modules

- function definition + executable statements
- executed only when module is imported
- modules have private symbol tables
- avoids name clash for global variables
- accessible as *module.globalname*
- can import into name space:

```
>>> from fibo import fib, fib2
>>> fib(500)
```

- can import all names defined by module:

```
>>> from fibo import *
```

Module search path

- current directory
- list of directories specified in PYTHONPATH environment variable
- uses installation-default if not defined, e.g., .:/usr/local/lib/python
- uses sys.path

```
>>> import sys
>>> sys.path
['', 'C:\\PROGRA~1\\Python2.2', 'C:\\Program Files\\Python2.2\
  \DLLs', 'C:\\Program Files\\Python2.2\\lib', 'C:\\Program
  Files\\Python2.2\\lib\\lib-tk', 'C:\\Program Files\
  \Python2.2', 'C:\\Program Files\\Python2.2\\lib\\site-
  packages']
```

Compiled Python files

- include byte-compiled version of module if there exists fibo.pyc in same directory as fibo.py
- only if creation time of fibo.pyc matches fibo.py
- automatically write compiled file, if possible
- platform independent
- doesn't run any faster, but *loads* faster
- can have only .pyc file → hide source

Standard modules

- system-dependent list
- always sys module

```
>>> import sys
>>> sys.p1
'>>> '
>>> sys.p2
'... '
>>> sys.path.append('/some/directory')
```

Module listing

- use `dir()` for each module

```
>>> dir(fibo)
['___name___', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__', '__st
din__', '__stdout__', '_getframe', 'argv', 'builtin_module_names', 'byteorder',
'copyright', 'displayhook', 'dllhandle', 'exc_info', 'exc_type', 'excepthook', '
exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getrecursionlimit', '
getrefcount', 'hexversion', 'last_type', 'last_value', 'maxint', 'maxunicode', '
modules', 'path', 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setpr
ofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
 'version_info', 'warnoptions', 'winver']
```

Classes

- mixture of C++ and Modula-3
- multiple base classes
- derived class can override any methods of its base class(es)
- method can call the method of a base class with the same name
- objects have private data
- C++ terms:
  - all class members are public
  - all member functions are virtual
  - no constructors or destructors (not needed)

Classes

- classes (and data types) are objects
- built-in types cannot be used as base classes by user
- arithmetic operators, subscripting can be redefined for class instances (like C++, unlike Java)

Class definitions

```
Class ClassName:
   <statement-1>
   ...
   <statement-N>
```

- must be executed
- can be executed conditionally (see Tcl)
- creates new namespace

Namespaces

- mapping from name to object:
  - built-in names (`abs()`)
  - global names in module
  - local names in function invocation
- attributes = any following a dot
  - `z.real`, `z.imag`
- attributes read-only or writable
  - module attributes are writeable

Namespaces

- scope = textual region of Python program where a namespace is directly accessible (without dot)
  - innermost scope (first) = local names
  - middle scope = current module's global names
  - outermost scope (last) = built-in names
- assignments always affect innermost scope
  - don't copy, just create name bindings to objects
- global indicates name is in global scope

Class objects

- `obj.name` references (plus module!):

```
class MyClass:
    "A simple example class"
    i = 123
    def f(self):
        return 'hello world'
>>> MyClass.i
123
```

- `MyClass.f` is method object

Class objects

- class instantiation:

```
>>> x = MyClass()
>>> x.f()
'hello world'
```

- creates new instance of class
  - note `x = MyClass` vs. `x = MyClass()`
- `___init__()` special method for initialization of object

```
def __init__(self,realpart,imagpart):
  self.r = realpart
  self.i = imagpart
```

Instance objects

- attribute references
- data attributes (C++/Java data members)
  - created dynamically

```
x.counter = 1
while x.counter < 10:
  x.counter = x.counter * 2
print x.counter
del x.counter
```

Method objects

- Called immediately:

```
x.f()
```

- can be referenced:

```
xf = x.f
while 1:
    print xf()
```

- object is passed as first argument of function → 'self'
  - x.f() is equivalent to MyClass.f(x)

Notes on classes

- Data attributes override method attributes with the same name
- no real hiding → not usable to implement pure abstract data types
- clients (users) of an object can add data attributes
- first argument of method usually called self
  - '`self`' has **no** special meaning (cf. Java)

Another example

- bag.py

```
class Bag:
  def __init__(self):
    self.data = []
  def add(self, x):
    self.data.append(x)
  def addtwice(self,x):
    self.add(x)
    self.add(x)
```

Another example, cont'd.

- invoke:

```
>>> from bag import *
>>> l = Bag()
>>> l.add('first')
>>> l.add('second')
>>> l.data
['first', 'second']
```

Inheritance

```
class DerivedClassName(BaseClassName)
  <statement-1>
  ...
  <statement-N>
```

- search class attribute, descending chain of base classes
- may override methods in the base class
- call directly via `BaseClassName.method`

Multiple inheritance

```
class DerivedClass(Base1,Base2,Base3):
    <statement>
```

- depth-first, left-to-right
- problem: class derived from two classes with a common base class

Private variables

- No real support, but textual replacement (name mangling)
- `__var` is replaced by `_classname_var`
- prevents only accidental modification, not true protection

~ C structs

- Empty class definition:

```
class Employee:
  pass

john = Employee()
john.name = 'John Doe'
john.dept = 'CS'
john.salary = 1000
```

Exceptions

- syntax (parsing) errors

```
while 1 print 'Hello World'
File "<stdin>", line 1
  while 1 print 'Hello World'
                ^
SyntaxError: invalid syntax
```

- exceptions
  - run-time errors
  - e.g., `ZeroDivisionError`, `NameError`, `TypeError`

Handling exceptions

```
while 1:
  try:
    x = int(raw_input("Please enter a number: "))
    break
  except ValueError:
    print "Not a valid number"
```

- First, execute `try` clause
- if no exception, skip `except` clause
- if exception, skip rest of `try` clause and use except clause
- if no matching exception, attempt outer `try` statement

Handling exceptions

- try.py

```
import sys
for arg in sys.argv[1:]:
  try:
    f = open(arg, 'r')
  except IOError:
    print 'cannot open', arg
  else:
    print arg, 'lines:', len(f.readlines())
    f.close
```

- e.g., as `python try.py *.py`

Language comparison

		Tcl	Perl	Python	JavaScript	Visual Basic
Speed	development	✓	✓	✓	✓	✓
	regexp	✓	✓	✓		
breadth	extensible	✓		✓		✓
	embeddable	✓		✓		
	easy GUI	✓		✓ (Tk)		✓
	net/web	✓	✓	✓	✓	✓
enterprise	cross-platform	✓	✓	✓	✓	
	I18N	✓		✓	✓	✓
	thread-safe	✓		✓		✓
	database access	✓	✓	✓	✓	✓