Python Crash-Course

C. Basso

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova

December 11, 2007

What is Python?

- Python is a (simple and easy to use) programming language

`www.python.org`

What is Python?

- Python is a (simple and easy to use) programming language
- The reference implementation is written in C. There are other implementations: Jython (Java), IronPython (C#), PyPy (Python).

www.python.org

What is Python?

- Python is a (simple and easy to use) programming language
- The reference implementation is written in C. There are other implementations: Jython (Java), IronPython (C#), PyPy (Python).
- wrt matlab: Python is general-purpose, it comes with batteries included (aka the standard library).

www.python.org

What is Python?

- Python is a (simple and easy to use) programming language
- The reference implementation is written in C. There are other implementations: Jython (Java), IronPython (C#), PyPy (Python).
- wrt matlab: Python is general-purpose, it comes with batteries included (aka the standard library).
- wrt C/C++: Python is interpreted, not compiled, which implies a faster development cycle.

www.python.org

What is Python?

- Python is a (simple and easy to use) programming language
- The reference implementation is written in C. There are other implementations: Jython (Java), IronPython (C#), PyPy (Python).
- wrt matlab: Python is general-purpose, it comes with batteries included (aka the standard library).
- wrt C/C++: Python is interpreted, not compiled, which implies a faster development cycle.
- the typing is not static: type-checking is performed at runtime (this means, for instance, you do not need templates as in C++)

www.python.org

Installation

Linux

- Gentoo: already there
- Debian, Ubuntu et similia: `sudo apt-get install python`
- Fedora: `sudo yum install python`

Windows

The is a binary installer

Mac OS X

Usually is pre-installed but old, better to use the up-to-date binary installer

The Intepreter

The interpreter, `python`, is a program that iteratively

- read expressions and statements
- evaluate them (which can mean process some data structure)
- print the result
- wait for more

just like matlab or a shell

```
$ python
Python 2.4.3 (#1, Sep  1 2006, 18:35:05)
[GCC 4.1.1 (Gentoo 4.1.1)] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

Those three greater-than signs (`>>>`) are the Python prompt where you write statements and expressions. To quit Python, press Ctrl-D.

Basic Expressions

The interpreter evaluates expressions:

```
>>> 5
5

>>> 10 + 4
14

>>> "Hello"
'Hello'
```

Quotes, single or double, are used to create strings. They can be nested:

```
>>> "'Hello'" + '"Hello"'
'\'Hello\'"Hello"'
```

Running Scripts

The interpreter can also run scripts:

```
$ python [options to python] foo.py [options to foo]
```

A script is typically divided in three parts:

```
# import section
import module1, module2
import module3 as m3
from module4 import foo4

# define some functions
def foo():
    ...

# main section
if __name__=='__main__':
    ...
```

Sequences

A string is a sequence:

```
>>> s = "Hello"
>>> s[0]
'H'
>>> s[-1]
'o'
>>> s[1:3]
'el'
>>> s[::2]
'Hlo'
```

Note that the first expression (assignment) has no result. Sequences are indexed starting from 0, reverse indexing is allowed, and slices are allowed too.
Other types of sequences are lists and tuples.

Lists and Tuples

These are a list and a tuple:

```
>>> [1,3,2]
[1, 3, 2]
>>> (1,3,2)
(1, 3, 2)
```

Lists are mutable, while tuples are immutable:

```
>>> l = [1,3,2]
>>> l[0] = 11
>>> l
[11, 3, 2]
>>> l = (1,3,2)
>>> l[0] = 11
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

Sequences

Sequences support common operations, as well as indexing.
Example with concatenation:

```
>>> [1,3,2] + [11,3,2]
[1, 3, 2, 11, 3, 2]
>>> (1,3,2) + (11,3,2)
[1, 3, 2, 11, 3, 2]
>>> '1,3,2,' + '11,3,2'
'1,3,2,11,3,2'
```

Operation	Result
x in s	True if an item of s is equal to x, else False
x not in s	False if an item of s is equal to x, else True
s + t	the concatenation of s and t
s * n , n * s	n ***shallow*** copies of s concatenated
s[i:j:k]	slice of s from i to j with step k (i,j and k all optionals)
len(s)	length of s
min(s)	smallest item of s
max(s)	largest item of s

Dictionaries

Dictionaries, also known as maps in C++ STL or hash in other languages, are another standard data structure.

```
>>> a = {0: 0, '1': 1, 2:'2'}
>>> a.keys()
[0, '1', 2]
>>> a.values()
[0, 1, '2']
>>> a.items()
[(0, 0), ('1', 1), (2, '2')]
```

Dictionaries are unordered, and indexed by heterogeneous keys (which must be hashable). They are mutable:

```
>>> a[0] = 3
>>> a[3] = 0
>>> a
{0: 3, '1': 1, 2: '2', 3: 0}
```

Sets

Sets are what the name suggests: unordered collections, without indexes. Repetitions are not allowed.

```
>>> s = set([1,2,3,4,4])
>>> s
set([1, 2, 3, 4])
```

They support set operations like union, intersection, etc.:

```
>>> t = set([3,4,5,6])
>>> s.union(t)
set([1, 2, 3, 4, 5, 6])
>>> s.intersection(t)
set([3, 4])
```

More on Variables

```
>>> blog = ["My first post", "Python is cool"]
>>> blog
['My first post', 'Python is cool']
```

`blog` is now ***bound*** to a list of strings.

```
>>> golb = blog
>>> golb
['My first post', 'Python is cool']
>>> blog = 0
>>> blog
0
>>> golb
['My first post', 'Python is cool']
```

The binding mechanism is ***extremely*** important to understand. That's NOT like C.

Functions

```
>>> blog = blog + ["A new post."]
>>> blog
['My first post', 'Python is cool', 'A new post.']
```

this operation can be defined via a function:

```
>>> def add_post(blog, new_post):
...     return blog + [new_post]
...
>>> golb = add_post(blog, 'fruz')
>>> golb
['My first post', 'Python is cool', 'A new post.', 'fruz']
>>> blog
['My first post', 'Python is cool', 'A new post.']
```

note how `blog` is left untouched and the ***indentation***

More on Functions

```
>>> def add_post(blog, new_post):
...     blog = blog + [new_post]
...
>>> golb = add_post(blog, 'fruz')
>>> golb
>>> blog
['My first post', 'Python is cool', 'A new post.']
```

First: `golb` is now empty because we do not return anything. Second: it does not work, because `blog` is bound locally to a new list, but the list to which `blog` is bound globally is left unchanged.
However, I can do the following:

```
>>> def add_post(blog, new_post):
...     blog.append(new_post)
...
>>> add_post(blog, 'fruz')
>>> blog
['My first post', 'Python is cool', 'A new post.', 'fruz']
```

Control Flow

As any programming language you have constructs which control the program flow:

- `if expr:` ... `elif expr:` ... `else:`
- `while expr:` ...
- `for item in iterable:` ...
- `break` and `continue`
- `try:` ... `except Exception:` ...

As in the case of the function definition, the blocks inside e.g. an if must be indented:

```
if check_condition():
    do_something()
else:
    do_something_else()
```

Objects and Classes

That's an empty class:

```
>>> class Post(object):
...     pass
...
>>>
```

That's the object:

```
>>> cool = Post()
>>> cool
<__main__.Post object at 0xb7ca642c>
```

And these are attributes we can set on the fly:

```
>>> cool.title = "Cool"
>>> cool.body = "Python is cool."
>>> cool.title
'Cool'
>>> cool.body
'Python is cool.'
```

Methods

```
>>> class Post(object):
...     def set_title(self, title):
...         self._title = title
...     def get_title(self):
...         return self._title
...
>>>
```

`self` is the keyword referring to the object, it must be the first argument for any class methods

```
>>> cool = Post()
>>> cool.set_title("Cool")
>>> cool.get_title()
'Cool'
>>> Post.set_title(cool, "Cooler")
>>> cool.get_title()
'Cooler'
```

Private Scope

```
>>> cool = Post()
>>> cool.set_title("Cool")
>>> cool.get_title()
'Cool'
>>> cool._title
'Cool'
```

there are no private methods/attibutes, the leading underscore is a convention

Special Methods

Classes have special methods `__foo__`.
Most notable example, the constructor:

```
>>> class Post(object):
...     def __init__(self, title, body):
...         self.set_title(title)
...         self.set_body(body)
>>> cool = blog.Post("Cool", "Python is cool")
>>> cool.get_title()
'Cool'
>>> cool.get_body()
'Python is cool'
>>>
```

- `__repr__`: special method called when a string representation of the object is needed (e.g. with `print someobj`)
- `__getitem__` and `__setitem__`: used to emulate sequence types (support the indexing operator `[]`)
- `__add__`, `__sub__`, ...: used to emulate numeric types (support the numerical operators `+`, `-`, ...)

Inheritance

Python does support for (multiple) class inheritance:

```
>>> class A(object):
...     def amethod(self):
...         pass
...
>>> a = A()
>>> a.amethod()
>>> class B(A):
...     def bmethod(self):
...         pass
...
>>> b = B()
>>> b.amethod()
>>> b.bmethod()
>>> a.bmethod()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'A' object has no attribute 'bmethod'
```

Input and Output (console)

Output to console is easy:

```
>>> print 'bla'
bla
>>> a = 0
>>> print a
0
>>> print '%d' % a
0
>>> print '%f' % a
0.000000
>>> print '%.2f' % a
0.00
```

Formatting is governed with the `%` operator (C-style). Input:

```
>>> raw_input('say something... ')
say something... bla
'bla'
```

Input and Output (file)

I/O to text files is... easy again:

```
for line in open('somefile.txt'):
   process_line(line) # line is a string

fh = open('somefile.txt', 'w')
for line in buffer:
   fh.write(line) # line is a string
fh.close()
```

Errors and Exceptions

As mentioned before, exceptions are supported via the following statements:

```
try:
    ...
except MyException, e:
    ...
[else:
    ...]
[finally:
    ...]
```

All exceptions derive from a base class, `Exception`.

Standard Library

The batteries included in Python are the modules of the standard library. Some of them:

- `bz2, gzip, zlib`: compress/uncompress (also on the fly)
- `ConfigParser`: parser for configuration files (.INI-style)
- `pickle/cPickle`: object serialization
- `csv`: I/O of CSV files
- `ctypes`: allows to call functions from dlls/shared libraries
- `heapq`: implementation of a priority queue
- `profile/cProfile`: profiling
- `optparse`: parsing of program options/flags
- `os`: access to OS functionalities
- `re`: regular expressions

References - Articles & Tutorials


Guido van Rossum (aka Benevolent Dictator For Life, BDFL)
Python Tutorial.
http://docs.python.org/tut/tut.html


Jose P. E. Fernandez.
Programming Python, Part I
Linux Journal, 158:2, 2007.
http://portal.acm.org/citation.cfm?id=1275014


Magnus Lie Hetland.
Instant Python
http://hetland.org/writing/instant-python.html


Sebastian Bassi.
A Primer on Python for Life Science Researchers
PLoS Computational Biology, 3(11): e199, 2007.
http://compbiol.plosjournals.org/perlserv/?request=get-document&doi=10.1371/journal.pcbi.0030199

References - Books


VV. AA.
Python 2.5 Documentation.
`http://docs.python.org/`


Alex Martelli.
Python in a Nutshell.
O'Reilly, 2003.


Alex Martelli and Anna Martelli Ravenscroft and David Ascher.
Python Cookbook
O'Reilly, 2005