Python: A Primer

A HMS Research Computing Training Session

Alex Truong

alex_truong@hms.harvard.edu

https://rc.hms.harvard.edu/

Introduction

Why Python?

Python is...

simple - resembles plain English

easy - no need to declare (in most cases), memory management

clean - whitespace-formatted for visibility

interpreted - good for developing, bad for performance (more on this later)

Interpreted	**Compiled**
Rapid prototyping	Faster Performance
Requires interpreter	Requires compiler (GCC, etc.)
Dynamic typing (sort of)	Static typing
Code-level optimization	Code- and Compiler-level optimization

Accessing Python on Orchestra

Logging into orchestra

Open a terminal and ssh into orchestra.med.harvard.edu

```
$ ssh training01@orchestra.med.harvard.edu
training01@orchestra.med.harvard.edu's password:

(snip)

training01@mezzanine:~$
```

Accessing Python on Orchestra

```
$ module avail dev/python
--------- /opt/Modules/3.2.10/modulefiles ---------
dev/python/2.7.10 dev/python/2.7.6  dev/python/3.4.2

$ module load dev/python/2.7.6

$ which python
/opt/python-2.7.6/bin/python

$ python
Python 2.7.6 (default, Aug  3 2015, 17:43:52)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Today's course will use 2.7.6.

Alternately, virtual environment:

```
$ module load dev/python/2.7.6
$ which virtualenv
/opt/python-2.7.6/bin/virtualenv
$ virtualenv nameofenv --system-site-packages
New python executable in nameofenv/bin/python
Installing setuptools, pip, wheel...done.
$ source nameofenv/bin/activate
(nameofenv)$ which python
~/nameofenv/bin/python
```

To deactivate:

```
(nameofenv)$ deactivate
$
```

Why virtual environment?

Python has modules. if you want to install your own, you need a virtual environment.

The version on Orchestra has a certain number of builtin modules; the `--system-site-packages` flag allows your virtual environment to inherit those packages so you don't have to reinstall them yourself.

For more information, you can visit our Personal Python Packages wiki page.

Installing Python modules

To install modules, generally use `pip` or `easy-install` (we recommend `pip`):

```
(nameofenv)$ pip install packagename
```

If that doesn't work (e.g. you've downloaded the archive manually), follow the instructions in the provided README file, but it'll go something like

```
(nameofenv)$ python setup.py install
```

(some will have you use `build` before `install`). Make sure you read the (hopefully provided) instructions when installing modules manually.

Viewing modules

to see which external modules have been installed/compatible with the package installer (`pip`):

```
$ pip freeze
/opt/python-2.7.6/lib/python2.7/site-
packages/pip/_vendor/requests/packages/urllib3/util/ssl_.py:90:
InsecurePlatformWarning: A true SSLContext object is not available.
This prevents urllib3 from configuring SSL appropriately and may cause
certain SSL connections to fail. For more information, see
https://urllib3.readthedocs.org/en/latest/security.html#insecureplatformw

  InsecurePlatformWarning
You are using pip version 7.1.0, however version 7.1.2 is available.
You should consider upgrading via the 'pip install --upgrade pip'
command.
alabaster==0.7.6
Babel==1.3
.
.
.
```

Viewing modules cont.

To see ALL modules available on your Python build, type either of:

```
$ python -c "help('modules')"
```

and

```
$ python
Python 2.7.6 (default, Aug  3 2015, 17:43:52)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> help('modules')
```

Viewing modules cont.

and it will output something like:

```
Please wait a moment while I gather a list of all available modules...

(snip)

BETA                    audiodev                intervaltree            rlcompleter
BaseHTTPServer          audioop                 intervaltree_bio        rmagic
Bastion                 autoreload              io                      robotparser
Bio                     babel                   iptools                 runpy
BioSQL                  base64                  iso8601                 sched
.
.
.
```

Let's Learn Python!

Syntax

Statements are terminated by newlines (e.g. enter key)

Examples:

```
>>> a = 1
>>> print "hello world"
hello world
```

Nuances: Quotations

In general, double and single quotes are interchangeable, both for printing and argument passing:

```
>>> print "hello world"
hello world
>>> print 'hello world'
hello world

>>> str = 'abcdefg'
>>> str1 = "abcdefg"
>>> str == str1
True
```

Nuances: Escaped characters

Sometimes, you'll need to escape (backslash) certain special characters to get them to display correctly when printing. For example, if you want to preserve double quotations in your string:

```
>>> print ""hello world""
  File "", line 1
    print ""hello world""
                ^
SyntaxError: invalid syntax
>>> print "\"hello world\""
"hello world"
```

A list of escape characters can be found at the Python documentation and elsewhere on the internet.

Comments

Comments are used to make code more legible. In python, they are denoted with the octothorpe:

```
#!/usr/bin/env python
...
# do stuff here
...
```

Multi-line comments can either be manually be broken down, or held in a docstring with triple quotes:

```
"""
this is a
multi-line comment
"""
```

We'll revisit the `#!` line in a little bit.

Basic Features

Data structures, looping, etc.

Data types

Python is dynamically typed; there is no need to declare (e.g. `int n = num`). You can also reassign:

```
>>> number = 1
>>> number
1
>>> number = 2
>>> number
2
>>> str = 'abc'
>>> str
'abc'
>>> str = 'def'
>>> str
'def'
>>> str = 1
>>> str
1
```

Data types, an aside

However, strings (and some other stuff like tuples) are not *mutable*; you cannot modify its content. When you reassign the variable as seen previously, you're actually generating a new object. You can't modify the original object:

```
>>> s = "abc"
>>>id(s)
4702124
>>> s[0]
'a'
>>> s[0] = "o"
Traceback (most recent call last):
  File "", line 1, in
TypeError: 'str' object does not support item assignment
```

The above example was shamelessly lifted off Stack Overflow.

Back to data types

Like most languages, Python has higher level data structures. Of note are lists (arrays) and dictionaries. Relatedly, there are also tuples and sets. Each structure fills different niches.

Lists

Lists are the most versatile; they are the effective equivalent of arrays in other languages. They are ordered, and you can fetch specific indices. You can mix and match types, have duplicates, nest lists, etc. To generate a list:

```
>>> lst = [] #initialize a list of indeterminate size
>>> lst.append('a')
>>> lst
['a']
>>>lst.extend(['b', 'c'])
>>>lst
['a', 'b', 'c']

>>> lst2 = [None]*5 #initialize a list of predetermined size (an array)
>>> lst2
[None, None, None, None, None]
>>> lst2[1] = 'foo'
>>>lst2[1]
'foo'
>>> lst2
[None, 'foo', None, None, None]
```

Sets

Sets are lists that do not allow duplicates. Sets are useful if you need to take unions, intersections, etc. To generate sets:

```
>>> st = set(lst)
>>> st
set(['a', 'c', 'b'])

>>> lst3 = [1, 1, 2, 3, 4, 4]
>>> st2 = set(lst3)
>>> st2
set([1, 2, 3, 4])

>>> st3 = {1, 2, 3}
>>> st3
set([1, 2, 3])
```

Sample set operations

Here are some elementary operations you can perform with sets:

```
>>> set1 = {1, 2, 3}
>>> set2 = {3, 4, 5}
>>> set1 | set2          # union
set([1, 2, 3, 4, 5])
>>> set1 & set2          # intersection
set([3])
>>> set1 - set2          # set difference
set([1, 2])
>>> set2 - set1
set([4, 5])
>>> set1 ^ set2          # symmetric difference
set([1, 2, 4, 5])
>>> set2 ^ set1
set([1, 2, 4, 5])
```

More information can be found in the documentation.

Tuples

Tuples are lists, but immutable. Once created, they cannot be modified. If you know the size of your data structure, tuples are preferred over lists because Python will know exactly how much memory to allocate. To create tuples:

```
>>> empty_tuple = ()
>>> empty_tuple
()
>>> one_element = 1,
>>> one_element
(1,)
>>> mixed_tuple = (1, 'a', [1, 'a'])
>>> mixed_tuple
(1, 'a', [1, 'a'])
>>> lst = [1, 2, 3]
>>> tuple_from_list = tuple(lst)
>>> tuple_from_list
(1, 2, 3)
```

Dictionaries

Dictionaries are associative collections (maps). They are composed of key:value pairs (most of the time). To create a dictionary:

```
>>> empty_dict = {}    # careful not to confuse this with empty sets;
to initialize an empty set, use set()
>>> empty_dict
{}
>>> dict1 = {1: 'a', 2: 'b', 3: 'c'}
>>> dict1
{1: 'a', 2: 'b', 3: 'c'}
>>> dict2 = dict([(1, 'a'), (2, 'b'), (3, 'c')])
>>> dict2
{1: 'a', 2: 'b', 3: 'c'}
>>> dict3 = dict(a=1, b=2, c=3)     # only works if keys are strings
>>> dict3
{'a': 1, 'c': 3, 'b': 2}
>>> dict4 = dict(1=a, 2=b, 3=c)
  File "", line 1
SyntaxError: keyword can't be an expression
```

Basic data structure manipulations

lists are zero-indexed (count from 0), and you can reference positions like so:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[1]
2
>>> lst[1:]     # sublist from second entry to end
[2, 3, 4, 5]
>>> lst[1:3]    # note that the fourth element is omitted; when end indices are specified, Python stops BEFORE they are reached
[2, 3]
>>> lst[:3]     # sublist from beginning to fourth entry
[1, 2, 3]
>>> lst[1::2]  # sublist from second to end, every other element
[2, 4]
>>> lst[-1]     # last entry (reverse indexing)
5
>>> lst2 = [1, 2, 3, [4, 5]]    # nested list
>>> lst2[3][1]  # specify all relevant indices from outside in
5
```

Basic data structure manipulations cont.

Dictionaries don't have indices to reference, but they do have keys and values. To interact with dictionaries:

```
>>> dict1 = {1: 'a', 2: 'b', 3: 'c'}
>>> dict1[1]
'a'
>>> dict1.keys()
[1, 2, 3]
>>> dict1.values()
['a', 'b', 'c']
```

There's a lot more you can do with these data structures, but they're beyond the scope of this course (i.e. fetching keys from values, comprehensions). They'll be demonstrated at the end if there's time.

Flow Control

for, while, if/else

For loops

For loops are very straightforward to implement, if a little obfuscated. To iterate over a list:

```
>>> things = ['apple', 'banana', 'cherry']
>>> for thing in things:
...     print thing
...
apple
banana
cherry
>>>
```

This will go through every item in your list (or tuple or whatever) in order. If you also wanted to fetch indices, you'd use:

```
>>> for idx, thing in enumerate(things):
...     print idx, thing
...
0 apple
1 banana
2 cherry
>>>
```

The `enumerate()` function is actually an *iterator* that spits out a tuple consisting of (index, value) and assigns each to idx, name. This is called a **named tuple**.

The `for` *`a`* `in` *`b`* structure can be replaced with any generic construct (within reason). For a generic loop, you can do something like:

```
>>>for i in range(5):
...     print i
...
0
1
2
3
4
```

While loops

Similar in function to `for` loops, while loops are used to iterate, and are useful if you don't know how long to iterate for (e.g. searching for convergence, etc.). A generic while loop looks like this:

```
>>>toggle = 'true'
>>> while toggle == 'true':
...     toggle = 'false'
...     print toggle
...
false
```

A very simplistic example, but the above while loop runs one iteration, then exits because `toggle` is no longer `true`.

If/elif/else

`if/else` statements are useful when you need to handle different cases in your workflow. A generic if/elif/else statement structure:

```
# take some input (let's say, a number)
if input == 0:        #if input is zero
    print 'zero'
elif input % 2 == 0: #if input is otherwise even
    print 'even'
else:                 #if input is odd
    print 'odd'
```

You may use as many `elifs` as you require to solve your problem.

An aside: try/except

Python has the interesting distinction of relatively lightweight exception handling. Exceptions are only computationally expensive if they trigger. For more information, this page has a good analysis of if/else versus try/except.

A few pertinent modules

SciPy

SciPy has a bunch of interesting functions that automate various aspects of scientific computing, like statistics and higher-level mathematics. Most of these are callable in one line. Briefly, this is how to use the SciPy module. You will need to consult the documentation for complex usage.

```
>>> from scipy import constants         # `from scipy import *` will not
behave as expected
>>> constants.c                         # speed of light
299792458.0
>>> from scipy.stats import norm
>>> norm.cdf(5, 0, 3)                   # P(x<5) if X~N(0,3)
0.9522096477271853
```

NumPy

Numpy is the go-to module if you need to perform complex arithmetic or do matrix operations (manipulate data frames. It is much more efficient (and easy to use) than using system Python utilities. An example:

```
>>> import numpy as np       # the general way; using `as` allows you
to set an alias (if the name is too long to type)
>>> cvalues = [25.3, 24.8, 26.9, 23.9]    # a typical python list of
Celcius values
>>> C = np.array(cvalues)       # create a numpy array
>>> print C
[ 25.3  24.8  26.9  23.9]       # note that they look identical
>>> print(C * 9 / 5 + 32)       # convert to Fahrenheit using scalar
multiplication
[ 77.54  76.64  80.42  75.02]
>>> fvalues = [x*9/5 + 32 for x in cvalues]    # with a typical list,
you need to loop over it and compute element-wise instead
>>> print fvalues
[77.54, 76.64, 80.42, 75.02]       # same result, less
efficient/readable
```

More NumPy

A brief look at arrays: NumPy data structures

```
>>> nested_list = [[3.4, 8.7, 9.9], [1.1, -7.8, -0.7], [4.1, 12.3,
4.8]]      # a python nested list
>>> nested_list
[[3.4, 8.7, 9.9], [1.1, -7.8, -0.7], [4.1, 12.3, 4.8]]
>>> A = np.array ([ [3.4, 8.7, 9.9], [1.1, -7.8, -0.7], [4.1, 12.3,
4.8]])     # a numpy array (matrix)
>>> A
array([[  3.4   8.7   9.9]
 [  1.1  -7.8  -0.7]
 [  4.1  12.3   4.8]])
```

Note the formatting of the print. This is indicative of the logic and illustrates why numpy is the preferred implementation of Python matrix operations. The SciPy documentation linked previously also includes NumPy documentation.

Matplotlib

Matplotlib is a basic plotting software. You can generate plots in real time (X11) or create them and write to files to view later. There is also a degree of customization afforded in plots. The below example covers the easiest example:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,2,3,4])
[(some memory address)]
>>> plt.ylabel('some numbers')
(another memory address)
>>> plt.show()
```

You can save the image using something like `plt.savefig('image.png')`.

See the matplotlib documentation for more information.

Object Oriented Python

PSA: Python can also implement classes.

Using classes over functions in Python is generally to the user's discretion, but standard common sense and logic applies as usual. If your program expands to the point where you think an object is more effective than functions, then feel free to implement it.

No instruction on classes will be given here, as it is beyond the scope of the course. For more information, you can consult an in-depth article like this or look straight at the Python documentation.

A brief Introduction to Scripting

Up until now, we've mostly been playing inside the interpreter. Here, we'll briefly go over what is required to write a proper Python program.

To start:

Strictly speaking, all you need for a Python program is a text file with the shebang line on top. Recall:

```
#!/usr/bin/env python
```

This line indicates to the computer that this is a python program, and it should look in this location to execute. Similar shebangs may look like:

```
#!usr/bin/python
#!/bin/bash
#!/usr/bin/perl
etc.
```

The shebang is telling the computer to look in the specified directory for the proper method of execution.

Why use `env`?

env is used for portability. On *nix machines, there is a path `/usr/bin` where most system programs/binaries are installed. If you need to install multiple versions, this can be an issue. Which version of Python do you want to use?

This is what env is for. It tells the computer to look at the current environment, and choose the Python that is currently in use. This is especially helpful on Orchestra, where we have multiple versions installed at the same time.

A basic program

Once you've included the shebang, you can get right to it. Start typing lines just as you would in the interpreter, and once you execute your program, each line will resolve itself in order.

```
#!/usr/bin/env python

print "hello world"
```

Type this into a text file, and save it as whatever, and include .py at the end for your own convenience.

To execute this program, just type at the terminal:

```
$ python file.py
```

You've just written your first python program!

Basic Structure

The previous program is not likely going to be your typical use case. More complex programs may use modules, functions, classes, etc.

A typical program will have the following structure:

```
#!/usr/bin/env python

# IMPORT EVERYTHING HERE

# CREATE FUNCTIONS AND/OR CLASSES HERE

def main():
    # CODE THAT DOES NOT BELONG IN FUNCTIONS GOES HERE

if __name__ == '__main__':
    main()
```

To extend the previous trivial example:

```
#!/usr/bin/env python

# no modules were required, so none were imported

# no functions were needed, so none were created

def main():
    print "hello world"

if __name__ == '__main__':
    main()
```

What are Functions?

Functions are typically used when you have repetitive code. Instead of pasting the code over and over, just put it in a function, and use ("call") it when needed. For example:

```
def do_thing():
    print "hello world"
    return
```

Then to reference it, just type elsewhere in your program:

```
do_thing()
```

and the program will execute the code within the `do_thing` function.

You can also make functions take arguments:

```
def do_thing(phrase):
    print phrase
    return

...

phrase = "hello world"
do_thing(phrase)
```

You can also assign values to variables with functions. To do this, make use of the `return` keyword. Note that previously, it has been naked, which means nothing is returned. Python is smart enough to know what you mean if you omit `return`, but it's always nice to include it for consistency.

To extend the previous example:

```
def do_thing(phrase):
    print phrase

    new_phrase = "goodbye world"

    return new_phrase

...

phrase = "hello world"

phrase2 = do_thing(phrase)
print phrase2
```

Putting it all together:

```
#!/usr/bin/env python

def do_thing(phrase):
    """Transform the input phrase into a new phrase."""
    print phrase

    new_phrase = "goodbye world"

    return new_phrase

def main():
    phrase = "hello world"

    phrase2 = do_thing(phrase)

    print phrase

if __name__ == '__main__':
```

And that's it! You can use as many functions as you want/need to solve your problem. Or, you can jam everything into your main() method if that is how it works.

Bonus Features!

More than one way to skin a cat

There are multiple ways to accomplish things in Python; some are more "Pythonic" than others. A few examples:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst2 = [0]*5
>>> for idx, num in enumerate(lst): #loop to reassign - multiple lines
...     lst2[idx] = num + 1
...
>>>lst2
[2, 3, 4, 5, 6]

>>> lst3 = [x + 1 for x in lst]      #list comprehension - one line
>>> lst3
[2, 3, 4, 5, 6]
```

As of 2.7, dictionary comprehensions are also possible.

Generally, you'll end up using the first method you discover (e.g. via Stack Overflow, etc.).

`lambda` is used if you have a simple function that only needs to be called once. Compare:

```
mult3 = filter(lambda x: x % 3 == 0, [1, 2, 3, 4, 5, 6, 7, 8, 9])
```

versus

```
def filterfunc(x):
    return x % 3 == 0

mult3 = filter(filterfunc, [1, 2, 3, 4, 5, 6, 7, 8, 9])
```

versus

```
mult3 = [x for x in [1, 2, 3, 4, 5, 6, 7, 8, 9] if x % 3 == 0]
```

This example was shamelessly stolen from Stack Overflow.

Python and LSF

The original LSF API is written in C, but the internet has ported it to Python (in various states of disarray). This is the one I'm using.

IBM has since made an official one available, but it does not correspond to the version of LSF currently on Orchestra.

Python and LSF

The API is useful if you need higher control over your job submissions and/or workflow.

```
>>> from pylsf import *               #import entire pylsf module
>>> a = lsb_init('')                  # start LSF daemon
>>> d = lsb_openjobinfo()             # form connection and acquire all job
information
>>> for x in range(d):                # iterate
...     job = lsb_readjobinfo(x)      # acquire job information about
current job
...     if job[1] == 'at237':
...         print job
...
```

For more information, refer to the pylsf.pyx file.

Final Thoughts: Compatibility

Today's course was taught on 2.7.6.

We have many versions of Python available, and not all are created equal. Most distinct is the difference between 2.x and 3.x; there are several syntax changes that will be required to use version 3.x.

For more information, look at this trusty 2.x to 3.x compatibility cheat sheet.

That's it! Thanks for coming! Time permitting, more will be demonstrated in the terminal.

If you have any questions, feel free to email me directly at alex_truong@hms.harvard.edu or visit our website to submit a ticket. We also do consulting!

Please take the time to fill out this survey so we can improve this course in the future!