Introduction to Python

Stephen Weston and Robert Bjornson

¶

Yale Center for Research Computing

Mar 2017

What is the Yale Center for Research Computing?

- Independent center under the Provost's office
- Created to support your research computing needs
- Focus is on high performance computing and storage
- ~15 staff, including applications specialists and system engineers
- Available to consult with and educate users
- Manage compute clusters and support users
- Located at 160 St. Ronan st, at the corner of Edwards and St. Ronan
- http://research.computing.yale.edu (http://research.computing.yale.edu)

Why Python?

- Free, portable, easy to learn
- Wildly popular, huge and growing community
- Intuitive, natural syntax
- Ideal for rapid prototyping but also for large applications
- Very efficient to write, reasonably efficient to run as is
- Can be very efficient (numpy, cython, ...)
- Huge number of packages (modules)

You can use Python to...

- Convert or filter files
- Automate repetitive tasks
- Compute statistics
- Build processing pipelines
- Build simple web applications
- Perform large numerical computations
- ...

You can use Python instead of bash, Java, or C

Python can be run interactively or as a program

Different ways to run Python

1. Create a file using editor, then:

   ```
   $ python myscript.py
   ```
2. Run interpreter interactively

   ```
   $ python
   ```
3. Use a python environment, e.g. Anaconda

Installing Anaconda (includes Jupyter notebook)

1. at www.continuum.io, download python 3.6 version of anaconda, and install
2. in a terminal, run python

To get tutorial files

1. browse to https://github.com/ycrc/Python-Bootcamp (https://github.com/ycrc/Python-Bootcamp)
2. click "clone or download" then "download zip"
3. unzip file into desired location

To follow presentation in Jupyter notebook

1. in a terminal, cd to Python-Bootcamp
2. run this command: jupyter notebook PythonNotebook.ipynb

Python 2 versus 3

- Two major versions of python in use
- For beginners, almost the same, major difference is print
- This tutorial uses python 3

see https://wiki.python.org/moin/Python2orPython3 (https://wiki.python.org/moin/Python2orPython3)

Basic Python Types

In [ ]:
```
radius=2
pi=3.14
diam=radius*2
area=pi*(radius**2)
title="fun with strings"
pi="cherry"
longnum=3141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067982148086\51
delicious=True
```

- variables do not need to be declared or typed
- integers and floating points can be used together
- the same variable can hold different types
- lines can be broken using \
- python supports arbitrary length integer numbers

In [ ]:
```
print (radius)
```

Other Python Types: *lists*

Lists are like arrays in other languages.

In [ ]:
```
l=[1,2,3,4,5,6,7,8,9,10]
l[5]
```

In [ ]:
```
l
```

In [ ]:
```
l[5:7]
```

In [ ]:
```
l[5:]
```

In [ ]:
```
l[2]=3.14
l[3]="pi"
l
```

In [ ]:
```
l.append(999)
l
```

In [ ]:
```
len(l[4:8])
```

Lists are more flexible than arrays, e.g.:

- Insert or append new elements
- remove elements
- nest lists
- combine values of different types into lists

In [ ]:
```
l=[1,2,3,4,5,6,7,8,9]
l[2]=[11,12,13]
l
```

In [ ]:
```
l[3:6]=['four to six']
l
```

Other Python Types: *tuples*

tuples are like lists, but not modifiable

In [ ]:
```
t=(1,2,3,4,5,6,7,8,9)
t
```

In [ ]:
```
t[4:6]
```

In [ ]:
```
t[5]=99
```

Other Python Types: *strings*

Strings are fully featured types in python.

- strings are defined with ' or "
- strings cannot be modified
- strings can be concatenated and sliced much like lists
- strings are objects with lots of useful methods

In [ ]:
```
s="Donald Duck"
s
```

In [ ]:
```
s="int\"s"
print(s)
```

In [ ]:
```
s[0]='Cl'
```

In [ ]:
```
s.upper()
```

Other Python Types: *dictionaries*

Dicts are what python calls "hash tables"

- dicts associate keys with values, which can be of (almost) any type
- dicts have length, but are not ordered
- looking up values in dicts is very fast, even if the dict is BIG.

In [ ]:
```
coins={'penny':1, 'nickle':5, 'dime':10, 'quarter':25}
coins['dime']
```

In [ ]:
```
coins['dime']
```

In [ ]:
```
 sorted(coins.keys())
```

Control Flow Statements: *if*

- if statements allow you to do a test, and do something based on the result
- *else* is optional

In [ ]:
```
import random

v=random.randint(0,100)
if v < 50:
    print ("small", v)
    print ("another line")
else:
    print ("big", v)
print ("after else")
```

Control Flow Statements: *while*

- While statements execute one or more statements repeatedly until the test is false

In [ ]:
```
import random
count=0
while count<100:
   count=count+random.randint(0,10)
   print (count)
```

Control Flow Statements: *for*

For statements take some sort of iterable object and loop once for every value.

In [ ]:
```
for fruit in ['apple', 'orange', 'banana']:
   print(fruit)
```

In [ ]:
```
for i in range(3,7):
   print(i)
```

Using for loops and dicts

If you loop over a dict, you'll get just keys. Use items() for keys and values.

In [ ]:
```
for denom in coins:
   print (denom)
```

In [ ]:
```
for denom, value in coins.items():
   print (denom, value)
```

Control Flow Statements: altering loops

While and For loops can skip steps (continue) or terminate early (break).

In [ ]:
```
for i in range(10):
   if i%2 != 0: continue
   print (i)
```

In [ ]:
```
for i in range(10):
   if i>5: break
   print (i)
```

Note on code blocks

In the previous example:

In [ ]:
```
for i in range(10):
   if i>5: break
   print (i)
```

How did we know that `print(i)` was part of the loop? What defines a loop?

Many programming languages use { } or Begin End to delineate blocks of code to treat as a single unit.

Python uses white space (blanks). To define a block of code, indent the block to the same level.

By convention and for readability, indent a consistent number, usually 3 or 4 spaces. Many editors will do this for you.

Functions

Functions allow you to write code once and use it many times.

Functions also hide details so code is more understandable.

In [ ]:
```
def area(w, h):
   return w*h

area(6, 10)
```

Summary of basic elements of Python

- 4 basic types: int, float, boolean, string
- 3 complex types: list, dict, tuple
- 4 control constructs: if, while, for, def

Try it yourself!

1. Start terminal
2. cd to Python-Bootcamp
3. $ jupyter notebook
4. click on Examples.ipynb
5. Try writing some code

Example 1: File Reformatter

Task: given a file of hundreds or thousands of lines:

```
FCID,Lane,Sample_ID,SampleRef,index,Description,Control,Recipe,...
160212,1,A1,human,TAAGGCGA-TAGATCGC,None,N,Eland-rna,Mei,Jon_mix10
160212,1,A2,human,CGTACTAG-CTCTCTAT,None,N,Eland-rna,Mei,Jon_mix10
160212,1,A3,human,AGGCAGAA-TATCCTCT,None,N,Eland-rna,Mei,Jon_mix10
160212,1,A4,human,TCCTGAGC-AGAGTAGA,None,N,Eland-rna,Mei,Jon_mix10
...
```

Remove the last 3 letters from the 5th column:

```
FCID,Lane,Sample_ID,SampleRef,index,Description,Control,Recipe,...
160212,1,A1,human,TAAGGCGA-TAGAT,None,N,Eland-rna,Mei,Jon_mix10
160212,1,A2,human,CGTACTAG-CTCTC,None,N,Eland-rna,Mei,Jon_mix10
160212,1,A3,human,AGGCAGAA-TATCC,None,N,Eland-rna,Mei,Jon_mix10
160212,1,A4,human,TCCTGAGC-AGAGT,None,N,Eland-rna,Mei,Jon_mix10
...
```

In tnis example, we'll show:

- reading lines of a file
- parsing and modifying the lines
- writing them back out
- creating a script to do the above and running it
- passing the script the file to modify

In pseudocode

```
open the input file
read the first header line, and print it out
for each remaining line in the file
  read the line
  find the value in the 5th column
  truncate it by removing the last three letters
  put the line back together
  print it out
```

Step 1: open the input file

In [ ]:
```
import sys
fp=open('badfile.txt')
```

In [ ]:
```
fp
```

Open takes a filename, and returns a ``file pointer''.

We'll use that to read from the file.

Step 2: read the first header line, and print it out

In [ ]:
```
import sys
fp=open('badfile.txt')
print (fp.readline().strip())
```

We'll call readline() on the file pointer to get a single line from the file. (the header line).

Strip() removes the return at the end of the line.

Then we print it.

Step 3: for each remaining line in the file, read the line

In [ ]:
```
import sys
fp=open('badfile.txt')
print (fp.readline().strip())
for l in fp:
  print(l)
```

A file pointer is an example of an iterator.

Instead of explicitly calling readline() for each line, we can just loop on the file pointer, getting one line each time.

Since we already read the header, we won't get that line.

Step 4: find the value in the 5th column, and remove last 3 letters

In [ ]:
```
import sys
fp=open('badfile.txt')
print (fp.readline().strip())
for l in fp:
    flds=l.strip().split(',')
    flds[4]=flds[4][:-3]
    print(flds)
```

Like before, we strip the return from the line.

We split it into individual elements where we find commas.

The 5th field is referenced by flds[4], since python starts indexing with 0. [:-3] takes all characters of the string until the last 3.

Step 5: put the line back together, and print it

In [ ]:
```
import sys
fp=open('badfile.txt')
print (fp.readline().strip())
for l in fp:
    flds=l.strip().split(',')
    flds[4]=flds[4][:-3]
    print (','.join(flds))
```

Join takes a list of strings, and combines them into one string using the string provided. Then we just print that string.

We would invoke it like this:

```
$ python fixfile.py badfile.txt

$ python fixfile.py badfile.txt > fixedfile.txt
```

Variations for you to try

1. modify to accept multiple input files and combine them into one fixed file
2. modify to accept the name of an output file and write the output to that file

Example 2: directory walk with file ops

Imagine you have a directory tree with many subdirectories.

In those directories are files named *.fastq. You want to:

- find them
- compress them to fastq.gz using a program
- delete them if the conversion was successful

In this example, we'll demonstrate:

- traversing an entire directory tree
- executing a program on files in that tree
- testing for successful program execution

In psuedocode

```
for each directory
   get a list of files in that directory
   for each file in that directory
     if that file's name ends with .fastq
       create a new file name with .gz added
       create a command to do the compression
       run that command and check for success
       if success
         delete the original
       else
         stop
```

The conversion command is: `gzip -c file.fastq > file.fastq.gz`

Step 1: directory traversal

We need a way to traverse all the files and directories. `os.walk(dir)` starts at dir and visits every subdirectory below it. It returns a list of files and subdirectories at each subdirectory.

For example, imagine we have the following dirs and files:

```
Ex2dir
Ex2dir/d1
Ex2dir/d1/d2
Ex2dir/d1/d2/f2.fastq
Ex2dir/d1/f1.fastq
```

In [ ]:
```
import os
for d , dirs, files in os.walk('Ex2dir'):
   print (d, dirs, files)
```

Step 2: Invoking other programs from python

The subprocess module has a variety of ways to do this. A simple one:

```
import subprocess

ret=subprocess.call(cmd, shell=True)
```

ret is 0 on success, non-zero error code on failure.

In [ ]:
```
import subprocess
ret=subprocess.call('gzip -c myfile.fastq > myfile.fastq.gz', shell
=True)
ret
```

Put it all together

In [ ]:
```
import os, sys, subprocess
sys.argv=['dummy', 'Ex2dir'] # for Jupyter we'll cheat
start=sys.argv[1]
for d, subdirs, files in os.walk(start):
    for f in files:
        if f.endswith('.fastq'):
            fn=d+'/'+f
            nfn=fn.replace('.fastq', '.fastq.gz')
            cmd='gzip -c '+fn+' > '+nfn
            print ("running", cmd)
            ret=subprocess.call(cmd, shell=True)
            if ret==0:
                if os.path.exists(nfn):
                    os.remove(fn)
            else:
                print ("Failed on ", fn)
                sys.exit(1)
```

Example 3: Nested Dictionaries

Dictionaries associate names with data, and allow quick retrieval by name.

By nesting dictionaries, powerful lookups are fast and easy.

In this example, we'll:

- create a dict containing objects
- load the objects with search data
- use the dict to retrieve the appropriate object for a search
- perform the search

genes.txt describes the locations of genes:

(name, chrom, strand, start, end)

```
uc001aaa.3      chr1    +       11873   14409
uc010nxr.1      chr1    +       11873   14409
uc010nxq.1      chr1    +       11873   14409
uc009vis.3      chr1    -       14361   16765
uc009vit.3      chr1    -       14361   19759
...
```

mappedreads.txt describes mapped dna sequences

(name, chrom, position, sequence)

```
seq1 chr1  674540   ATCTGTGCAGAGGAGAACGCAGCTCCGCCCTCGCGGT
seq2 chr19 575000   AGAGGAGAACGCAGCTCCGCCCTCGCGGTGCTCTCCG
seq3 chr5  441682   TCTGCATCTGCTCTGGTGTCTTCTGCCATATCACTGC
...
```

We'd like to be able to quickly determine the genes overlapped by a dna sequence.

First, we need a simple way to determine if two intervals overlap.

intervaltree is a python module that makes that easy.

In [ ]:
```
from intervaltree import IntervalTree
it=IntervalTree()
it[4:7]='gene1'
it[5:10]='gene2'
it[1:11]='gene3'
it
```

In [ ]:
```
it[3:5]
```

General plan

- use interval trees, one for each chromosome
- organize the trees in a dictionary by chromosome
- store an interval for each gene the tree for it's chromosome

```
{'chr1': IntervalTree([Interval(1000, 1100, 'GeneA'),
                       Interval(2000, 2100, 'GeneB'), ...
 'chr2': IntervalTree([Interval(4000, 5100, 'GeneC'),
                       Interval(7000, 8100, 'GeneD'), ...
 'chr3':
 ...
```

In psuedocode

setup the lookup table

```
create empty dict
open the gene file
for each line in the file
   get gene name, chrom, start, end
   initialize an intervaltree for the chrom, if needed, and add to dict
   add the interval and gene name to the interval tree
```

In [ ]:
```
import sys
from intervaltree import IntervalTree

print("initializing table")
table={}
sys.argv=['dummy', 'genes.txt', 'mappedreads.txt'] # for Jupyter
for line in open(sys.argv[1]):
    genename, chrm, strand, start, end = line.split()
    if not chrm in table:
        table[chrm]=IntervalTree()
    table[chrm][int(start):int(end)]=genename
print("done")
```

In [ ]:
```
table['chr1'][670000]
```

use the interval trees to find overlapped genes

open the dna sequence file
for each line in the file:
  get chrom, mapped position, and dna sequence
  look up the interval tree for that chrom in the dict
  search the interval tree for overlaps [pos, pos+len]
  print out the gene names

In [ ]:
```
print("reading sequences")

for line in open(sys.argv[2]):
    name, chrm, pos, seq = line.strip().split()
    genes=table[chrm][int(pos):int(pos)+len(seq)]
    if genes:
        print(name, chrm, pos, seq)
        for gene in genes:
            print ('\t',gene.data)
```

Python Resources we like

- anaconda python: www.continuum.io
- pycharm debugger: www.jetbrains.com
- *Introducing Python*, Bill Lubanovic, O'Reilly
- *Python in a Nutshell*, Alex Martelli, O'Reilly
- *Python Cookbook*, Alex Martelli, O'Reilly
- Google's python class: https://www.youtube.com/watch?v=tKTZoB2Vjukxo (https://www.youtube.com/watch?v=tKTZoB2Vjukxo)
- https://docs.python.org/3.5/tutorial (https://docs.python.org/3.5/tutorial)

To get help or report problems

- Check our status page: http://research.computing.yale.edu/system-status (http://research.computing.yale.edu/system-status)
- Send an email to our tracking system: hpc@yale.edu
- Read documentation: http://research.computing.yale.edu/hpc-support (http://research.computing.yale.edu/hpc-support)
- Office hours: http://research.computing.yale.edu/hpc-support/office-hours-support (http://research.computing.yale.edu/hpc-support/office-hours-support)
- Email us directly:
  - Stephen.weston@yale.edu
  - Robert.bjornson@yale.edu