Introduction to Python

Stephen Weston Robert Bjornson

Yale Center for Research Computing
Yale University

July 2016

What is the Yale Center for Research Computing?

- Specifically created to support your research computing needs
- ~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

Y|CRC

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
- Numerous extensions (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

Running Python

- Create a file using editor, then:
  `$ python myscript.py`
- Run interpreter interactively
  `$ python`
- Use a python environment, e.g. Anaconda

Overview of topics

- Some python data types and control statements
- Examples of python in action
- The Anaconda python environment
- Numerical computation in python
- Resources

Basic Python Types and Assignment

```
>>> 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

Other Python Types: lists

Lists are like arrays in other languages.

```
>>> l=[1,2,3,4,5,6,7,8,9,10]
>>> l[5]
6
>>> l[3:5]
[4, 5]
>>> l[5:]
[6, 7, 8, 9, 10]
>>> l[5:-3]
[6, 7]
>>> l[2]=3.145
>>> l[3]="pi"
>>> l
[1, 2, 3.145, 'pi', 5, 6, 7, 8, 9, 10]
>>> len(l)
10
```

Lists continued

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

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

```
>>> l=[1,2,3,4,5,6,7,8,9]
>>> l+[11,12,13]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13]
>>> l[3:6]=['three to six']
>>> l
[1, 2, 3, 'three to six', 7, 8, 9]
```

Tuples

Tuples are similar to lists, but cannot be modified.

```
>>> t=(1,2,3,4,5,6,7,8,9)
>>> t[4:6]
(5, 6)
>>> t[6]="changeme"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t.append('more')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Other Python Types: strings

Strings are fully featured types in python.

```
>>> s="Donald"
>>> s[0:3]
'Don'
>>> s+" Duck"
'Donald Duck'
>>> s[0]="R"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 's' object does not support item assignment
>>> len(s)
6
>>> s.upper()
'DONALD'
```

Note:

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

Other Python Types: dictionaries

Dicts are like hash tables in other languages.

```
>>> coins={'penny':1, 'nickle':5, 'dime':10, 'quarter':25}
>>> coins['penny']
1
>>> coins.keys()
['quarter', 'nickle', 'penny', 'dime']
>>> coins.values()
[25, 5, 1, 10]
>>> coins['half']=50
>>> coins
{'quarter': 25, 'nickle': 5, 'penny': 1, 'half': 50, 'dime'
>>> len(coins)
5
```

Note:

- 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.

Control Flow Statements: if

If statements allow you to do a test, and do something based on the result:

```
>>> import random
>>> v=random.randint(0,100)
>>> if v < 50:
...     print 'got a little one', v
... else:
...     print 'got a big one', v
...
got a big one 93
```

Note that the else clause is optional.

Control Flow Statements: while

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

```
>>> import random
>>> count=0
>>> while count<100:
...     count=count+random.randint(0,10)
...     print count,
...
5 11 19 19 21 28 35 37 47 53 53 57 58 59 60 66 71
75 82 86 94 101
```

Control Flow Statements: for

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

```
>>> for fruit in ['apple', 'orange', 'banana']:
...     print fruit,
...
apple orange banana
>>> for i in range(5):
...     print i,
...
0 1 2 3 4
```

Using for loops and dicts

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

```
>>> for denom in coins:  print denom
...
quarter
nickle
penny
dime
>>> for denom, value in coins.iteritems():  print denom, value
...
quarter 25
nickle 5
penny 1
dime 10
```

Control Flow Statements: altering loops

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

```
>>> for i in range(10):
...     if i%2 != 0: continue
...     print i,
...
0 2 4 6 8
>>> for i in range(10):
...     if i>5: break
...     print i,
...
0 1 2 3 4 5
```

Note on blocks of code

In the previous example:

```
>>> for i in range(10):
...     if i>5: break
...     print i,
```

How did we know that *print i* was part of the 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.

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.

```
>>> def area(w, h):
...     return w*h

>>> area(3, 4)
12
>>> area(5, 10)
50
```

Some languages differentiate between functions and procedures. In python, everything is a function. Procedures are functions that return no values.

Summary

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

Y|CRC

File Formatter example

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
...
```

File Formatter example (cont)

In pseudocode we might write:

```
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
```

In Python:

```
import sys
fp=open(sys.argv[1])
print fp.readline().strip()
for l in fp:
   flds=l.strip().split(',')
   flds[4]=flds[4][:-3]
   print ','.join(flds)
```

File Formatter example (cont)

open the input file

```
import sys
fp=open(sys.argv[1])
```

Sys is a system module with a number of useful methods.

sys.argv() returns the command line as an array of strings.

sys.argv[0] is the command, sys.argv[1] is the first argument, etc.

Open takes a filename, and returns a "file pointer".

We'll use that to read from the file.

File Formatter example (cont)

read the first header line, and print it out

```
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.

File Formatter example (cont)

for each remaining line in the file
read the line

```
for l in fp:
   ...
```

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.

File Formatter example (cont)

find the value in the 5th column
truncate it by removing the last three letters

```
flds=l.strip().split(',')
flds[4]=flds[4][:-3]
```

Just 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.

File Formatter example (cont)

put the line back together
print it out

```
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.

File Formatter example (cont)

Reviewing:

```
import sys
fp=open(sys.argv[1])
print fp.readline().strip()
for l in fp:
    flds=l.strip().split(',')
    flds[4]=flds[4][:-3]
    print ','.join(flds)
```

We would invoke it like this:

```
$ python fixfile.py badfile.txt

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

Some variations on the theme

We could skip certain lines (with other than human in the 3rd column)

We could also specify the output file on the command line

```
import sys
fp=open(sys.argv[1])
ofp=open(sys.argv[2], 'w')
print >> ofp, fp.readline().strip()
for l in fp:
    flds=l.strip().split(',')
    if flds[3] != 'human': continue
    flds[4]=flds[4][:-3]
    print >> ofp, ','.join(flds)
```

Some variations on the theme

We could operate on multiple input files

```
$ python fixfile.py badfile1.txt badfile2.txt badfile3.txt
```

```
import sys
wrotehdr=False
for f in sys.argv[1:]:
    fp=open(f)
    hdr=fp.readline().strip()
    if not wrotehdr:
        print hdr
        wrotehdr=True
    for l in fp:
        flds=l.strip().split(',')
        flds[4]=flds[4][:-3]
        print ','.join(flds)
```

Directory Walk Example

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.qp 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

Directory walk example (cont)

In pseudocode we might write:

```
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 .qp 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: quip -c file.fastq >file.fastq.qp

Using os.walk

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:

```
d1
d1/d2
d1/d2/f2.txt
d1/f1.txt
```

This is how we use os.walk:

```
>>> import os
>>> for d, dirs, files in os.walk('d1'):
...   print d, dirs, files
...
d1 ['d2'] ['f1.txt']
d1/d2 [] ['f2.txt']
```

Invoking programs from python

The subprocess module has a variety of ways to do this.

A very simple use is:

```
import subprocess
ret=subprocess.call(cmd, shell=True)
ret=subprocess.call('quip -c myfile.fastq > myfile.fastq.qp', shell=True)
```

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

Directory Walk Example (cont)

```
import os, sys, subprocess
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.qp')
            cmd='quip -c '+fn+' > '+nfn
            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)
```

To run it, we'd do: `$ python walk.py d1`

Dictionary Example

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

By nesting dictionaries, powerful lookups are 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

Y|CRC

Dictionary Example (cont)

We have a file describing the locations of genes:

```
uc001aaa.3      chr1    +       11873   14409   11873   11873   3       11873,12612,13220,      12227,12721,14409,              uc001aaa.3
uc010nxr.1      chr1    +       11873   14409   11873   11873   3       11873,12645,13220,      12227,12697,14409,              uc010nxr.1
uc010nxq.1      chr1    +       11873   14409   12189   13639   3       11873,12594,13402,      12227,12721,14409,      B7ZGX9  uc010nxq.1
uc009vis.3      chr1    -       14361   16765   14361   14361   4       14361,14969,15795,16606,        14829,15038,15942,16765,                uc00
uc009vit.3      chr1    -       14361   19759   14361   14361   9       14361,14969,15795,16606,16857,17232,17914,18267,18912,  14829,15038,15947,16
...
```

We have another file with dna sequences, and where they mapped:

```
HWI-ST830:206:D2411ACXX:1:1114:6515:89952       401     chr1    10536   0       76M     =       222691803       222681343       TACCACCGAAATCTGTGCAG
GCTCTCCGGGTCTGTGCTGAGGAGAACGC   ##B<2DDDDDDDCCDCC@CC@C@282BBCCDDBDDFHIJJJIGJIIGIGFIGJJIJJJJJJJJHGGHHFFFFDCC@    XA:i:1  MD:Z:24C51      NM:i:1  XP:Z
HWI-ST830:206:D2411ACXX:1:1114:6515:89952       177     chr1    10536   0       76M     chr3    197908818       0       TACCACCGAAATCTGTGCAGAGGAGAAC
GGTCTGTGCTGAGGAGAACGC   ##B<2DDDDDDDCCDCC@CC@C@282BBCCDDBDDFHIJJJIGJIIGIGFIGJJIJJJJJJJJHGGHHFFFFDCC@    XA:i:1  MD:Z:24C51      NM:i:1  XP:Z:chr3 19
HWI-ST830:206:D2411ACXX:1:1114:6515:89952       401     chr1    10536   0       76M     chr4    120370019       0       TACCACCGAAATCTGTGCAGAGGAGAAC
GGTCTGTGCTGAGGAGAACGC   ##B<2DDDDDDDCCDCC@CC@C@282BBCCDDBDDFHIJJJIGJIIGIGFIGJJIJJJJJJJJHGGHHFFFFDCC@    XA:i:1  MD:Z:24C51      NM:i:1  XP:Z:chr4 12
HWI-ST830:206:D2411ACXX:1:1114:6515:89952       433     chr1    10536   0       76M     chr9    141135264       0       TACCACCGAAATCTGTGCAGAGGAGAAC
GGTCTGTGCTGAGGAGAACGC   ##B<2DDDDDDDCCDCC@CC@C@282BBCCDDBDDFHIJJJIGJIIGIGFIGJJIJJJJJJJJHGGHHFFFFDCC@    XA:i:1  MD:Z:24C51      NM:i:1  XP:Z:chr9 14
...
```

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

Dictionary Example (cont)

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

intervaltree is a python module that makes that easy.

```
>>> from intervaltree import IntervalTree
>>> it=IntervalTree()
>>> it[4:7]='I1'
>>> it[5:10]='I2'
>>> it[1:11]='I3'
>>> it
IntervalTree([Interval(1, 11, 'I3'), Interval(4, 7, 'I1'),
              Interval(5, 10, 'I2')])
>>> it[7]
set([Interval(1, 11, 'I3'), Interval(5, 10, 'I2')])
>>> it[6:8]
set([Interval(4, 7, 'I1'), Interval(1, 11, 'I3'),
                           Interval(5, 10, 'I2')])
```

We'll use interval trees, one for each chromosome, to store an interval for each gene.

Then we'll find the overlaps for mapped dna sequences.

Dictionary Example (calculating overlaps)

Here is a picture of what we want:

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

Dictionary Example (cont)

Again, in pseudocode:

```
# create the interval trees
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

# 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 seq
  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
```

Dictionary Example (cont)

```
import sys
from intervaltree import IntervalTree

print "initializing"
genefinder={}
for line in open(sys.argv[1]):
    genename, chrm, strand, start, end = line.split()[0:5]
    if not chrm in genefinder:
        genefinder[chrm]=IntervalTree()
    genefinder[chrm][int(start):int(end)]=genename

print "reading sequences"
for line in open(sys.argv[2]):
    tag, flag, chrm, pos, mapq, cigar, rnext,
        pnext, tlen, seq, qual = line.split()[0:11]
    genes=genefinder[chrm][int(pos):int(pos)+len(seq)]
    if genes:
        print tag
        for gene in genes:
            print '\t',gene.data
```

Dictionary Example (cont)

```
rdb9@bdn:ruddle2:~ $ python dict_example.py knownGene.txt hits.sam
initializing
reading sequences
HWI-ST0831:196:C1YCJACXX:2:2211:2571:23347
        uc004cqm.3
        uc010nda.3
        uc004cqn.3
HWI-ST0831:196:C1YCJACXX:2:2114:9661:90395
        uc003zbm.3
HWI-ST0831:196:C1YCJACXX:2:2302:16215:62515
        uc003pvj.3
        uc003pvh.3
        uc010kdy.1
```

Python Resources we like

- *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://docs.python.org/2.7/tutorial

To get help or report problems

- Check our status page: `http://research.computing.yale.edu/system-status`
- Send an email to: `hpc@yale.edu`
- Read documentation at: `http://research.computing.yale.edu/hpc-support`
- Email us directly:
  - Stephen.weston@yale.edu, Office hours at CSSSI on Wednesday morning from 9 to 12 or by appointment
  - Robert.bjornson@yale.edu, By appointment

Y|CRC

When reporting a problem

It is best to send problem reports to our tracking system: hpc@yale.edu

Please include, as applicable:

- The cluster you're working on
- The directory you're working in
- The command you ran, and what happened, in as much detail as you can

See our intro to HPC bootcamp presentation:
http://research.computing.yale.edu/hpc-bootcamp

Y|CRC