GACRC

Python Language Basics I

Georgia Advanced Computing Resource Center
University of Georgia
Zhuofei Hou, HPC Trainer
zhuofei@uga.edu

GACRC

Outline

- What is GACRC?
- Hello, Python!
- General Lexical Conventions
- Basic Built-in Data Types
- Program Structure: Control Flow and Loop
- Function: Procedural and Functional Programming

What is GACRC?

Who Are We?

- **G**eorgia **A**dvanced **C**omputing **R**esource **C**enter
- Collaboration between the Office of Vice President for Research (**OVPR**) and the Office of the Vice President for Information Technology (**OVPIT**)
- Guided by a faculty advisory committee (GACRC-AC)

Why Are We Here?

- To provide computing hardware and network infrastructure in support of high-performance computing (**HPC**) at UGA

Where Are We?

- http://gacrc.uga.edu (Web)
- http://wiki.gacrc.uga.edu (Wiki)
- http://gacrc.uga.edu/help/ (Web Help)
- https://wiki.gacrc.uga.edu/wiki/Getting_Help (Wiki Help)

GACRC Users September 2015

Colleges & Schools	Depts	PIs	Users
Franklin College of Arts and Sciences	14	117	661
College of Agricultural & Environmental Sciences	9	29	128
College of Engineering	1	12	33
School of Forestry & Natural Resources	1	12	31
College of Veterinary Medicine	4	12	29
College of Public Health	2	8	28
College of Education	2	5	20
Terry College of Business	3	5	10
School of Ecology	1	8	22
School of Public and International Affairs	1	3	3
College of Pharmacy	2	3	5
	40	214	970
Centers & Institutes	9	19	59
TOTALS:	49	233	1029

GACRC Users September 2015

Centers & Institutes	PIs	Users
Center for Applied Isotope Study	1	1
Center for Computational Quantum Chemistry	3	10
Complex Carbohydrate Research Center	6	28
Georgia Genomics Facility	1	5
Institute of Bioinformatics	1	1
Savannah River Ecology Laboratory	3	9
Skidaway Institute of Oceanography	2	2
Center for Family Research	1	1
Carl Vinson Institute of Government	1	2
	19	59

GACRC

Hello, Python!

- What is Python
- Where is Python on Clusters
- Run Python Interactively on Clusters
- Scientific Python Modules
- Scientific Python Distributions

What is Python

- Open source general-purpose scripting language (https://www.python.org/)
- Working with *procedural*, *object-oriented*, and *functional* programming
- Glue language with Interfaces to C/C++ (via SWIG), Object-C (via PyObjC), Java (Jython), and Fortran (via F2PY) , etc. (https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages)
- Mainstream version is 2.7.x; new version is 3.5.x (*as to March 2016*)

Where is Python on Clusters

- Currently GACRC has two clusters zcluster and Sapelo:

Version	Installation Path	Invoke command
2.4.3 (default)	/usr/bin	python
2.7.2	/usr/local/python/2.7.2	python2.7
2.7.8	/usr/local/python/2.7.8	/usr/local/python/2.7.8/bin/python
3.3.0	/usr/local/python/3.3.0	python3
3.4.0	/usr/local/python/3.4.0	python3.4
Version	**Installation Path**	**Invoke command**
2.6.6 (default)	/usr/bin	python
2.7.8	/usr/local/apps/python/2.7.8	module load python/2.7.8 ; python
3.4.3	/usr/local/apps/python/3.4.3	module load python/3.4.3 ; python3

https://wiki.gacrc.uga.edu/wiki/Python ; https://wiki.gacrc.uga.edu/wiki/Python-Sapelo

Run Python Interactively on Clusters

- Run default python interactively on clusters' interactive nodes (qlogin):

```
zhuofei@compute-14-9:~$ python
Python 2.4.3 (#1, Oct 23 2012, 22:02:41)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-54)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> a = 7
>>> e = 2
>>> a**e
49
>>>
```

```
[zhuofei@n15 ~]$ python
Python 2.6.6 (r266:84292, Jan 22 2014, 09:42:36)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> a = 7
>>> e = 2
>>> a**e
49
>>>
```

Run Python Interactively on Clusters

- Run Python script interactively on clusters' interactive nodes (qlogin):

```
zhuofei@compute-14-9:~$ python myScript.py
2.4.3 (#1, Oct 23 2012, 22:02:41)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-54)]
49
```

```
[zhuofei@n15 ~]$ python myScript.py
2.6.6 (r266:84292, Jan 22 2014, 09:42:36)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)]
49
```

- myScript.py:

```
import sys
print sys.version

a = 7
e = 2
print a**e
```

Run Python Interactively on Clusters

- Run Python script as an *executable* interactively on clusters' interactive nodes:

```
zhuofei@compute-14-9:~$ chmod u+x myScript.py
zhuofei@compute-14-9:~$ ./myScript.py
2.7.2 (default, May 28 2015, 14:19:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-51)]
49
```

```
[zhuofei@n15 ~]$ chmod u+x myScript.py
[zhuofei@n15 ~]$ ./myScript.py
2.6.6 (r266:84292, Jul 23 2015, 15:22:56)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)]
49
```

- myScript.py:

```
#!/usr/local/python/2.7.2/bin/python

import sys
print sys.version
a = 7; e = 2
print a**e
```

tell system where the python lives

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

import sys
print sys.version
a = 7; e = 2
print a**e
```

the env program will locate the python according to PATH

Scientific Python Modules

- Python has a large collection of proven built-in modules included in standard distributions:

  https://docs.python.org/2/py-modindex.html

  https://docs.python.org/3/py-modindex.html

- Packages for scientific modules:
  - NumPy
  - SciPy
  - Matplotlib
  - Sympy
  - Biopy

Scientific Python Modules

- NumPy: Matlab-ish capabilities, fast N-D array operations, linear algebra, etc. (http://www.numpy.org/)
- SciPy: Fundamental library for scientific computing (http://www.scipy.org/)
- Sympy: Symbolic mathematics (http://www.sympy.org/en/index.html)
- matplotlib: High quality plotting (http://matplotlib.org/)
- Biopy: Phylogenetic exploration (https://code.google.com/archive/p/biopy/)

A scientific Python distribution may include all those packages for you!

Scientific Python Distributions

- Anaconda
  - "A Python distribution including over 195 of the most popular Python packages for science, math, engineering, data analysis"
  - Supports Linux, Mac and Windows (https://www.continuum.io/)
- Python(x,y)
  - Windows only (http://python-xy.github.io/)
- WinPython
  - Windows only (http://winpython.github.io/)

Anaconda with Spyder IDE on my local computer:

Spyder (Python 3.5)

File Edit Search Source Run Debug Consoles Tools View Help

/home/MosesHou

Editor - /home/MosesHou/python scripts/numpy.py

numpy.py

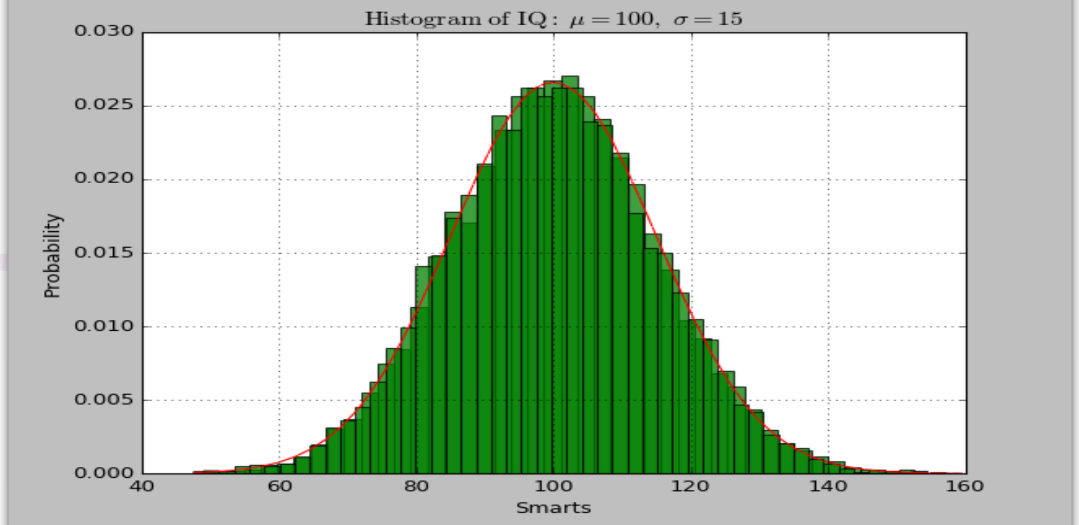
```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 14 10:56:30 2016

@author: MosesHou
"""

#!/usr/bin/env python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

mu, sigma = 100, 15

x = mu + sigma*np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='green', alpha=0.75)

# add a 'best fit' line
y = mlab.normpdf( bins, mu, sigma)
l = plt.plot(bins, y, 'r--', linewidth=1)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'$\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)

plt.show()
```

Console

Python 1

```
Python 3.5.1 |Anaconda 2.5.0 (64-bit)| (default, Dec  7 2015, 11:16:01)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> runfile('/home/MosesHou/python scripts/numpy.py', wdir='/home/MosesHou/python scripts')
>>> runfile('/home/MosesHou/python scripts/numpy.py', wdir='/home/MosesHou/python scripts')
>>>
```

Figure 1



Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 30 Column: 1 Memory: 8 %

Scientific Python Distributions

- Anaconda is installed on GACRC zcluster and Sapelo:

Version	Installation Path	Python Version	Export (2.3.0 as example)	Invoke Command
2.3.0	/usr/local/anaconda/2.3.0	2.7.11	export PATH=/usr/local/anaconda/2.3.0/bin:$PATH export PYTHONPATH=/usr/local/anaconda/2.3.0/bin:\ /usr/local/anaconda/2.3.0/lib/python2.7:$PYTHONPATH	python
3-2.2.0	/usr/local/anaconda/3-2.2.0	3.4.3		

Version	Installation Path	Python Version	Module Load (2.2.0 as example)	Invoke Command
2.2.0	/usr/local/apps/anaconda/2.2.0	2.7.11	module load anaconda/2.2.0	python
2.5.0	/usr/local/apps/anaconda/2.5.0			
3-2.2.0	/usr/local/apps/anaconda/3-2.2.0	3.4.3		

General Lexical Conventions

- A code sample:

```
x = 10; y = "Hello!”                    # this is a comment
z = 3.14                                # z is a floating number

if z == 3.14 or y == "Hello!":
   x = x + 1
   y = y + " Python!"

print x
print y
```

- Output:

```
zhuofei@compute-14-9:~$ python ./myScript_1.py
11
Hello! Python!
```

- Semicolon ; to separate statements on the same line
- Hash # denotes a comment
- Assignment uses = ; comparison uses ==
- Logical operators are words: and, or, not
- Consistent indention within a block (4 spaces)
- For numbers: + - * / % are as expected

  For strings: + means concatenation
- The basic printing statement: print

Basic Built-in Data Types

- "Python is a dynamically typed language where variable names are bound to different values, possibly of varying types, during program execution. Variables names are untyped and can be made to refer to any type of data."

—*Python Essential Reference, 4th ed.*

```
a = 10              # a is created to refer to an integer
a = 3.24            # a is referring to a floating-point number now
a = "Hello!"        # a is referring to a string now
a = True            # a is referring to a boolean (True/False) now
```

Basic Built-in Data Types

Type Category	Type Name	Description
Numbers	int	i = 10; integer
	long	l = 73573247851; arbitrary-precision integer (Python 2 only!)
	float	f = 3.14; floating point
	complex	c = 3 + 2j; complex
	bool	b = True; Boolean (True or False)
Sequences	str	s = “Hello! Python”; character string
	list	lst = [1, 2, ”abc”, 2.0]; list of any typed elements (mutable!)
	tuple	t = (1, 2, “abc”, 2.0); record of any typed elements (immutable!)
Mapping	dict	d = {1:”apple”, 2:””}; mapping dictionary of any typed pairs of key:value

Basic Built-in Data Types

- **List**: A mutable sequence of arbitrary objects of any type

list1 = [1, “David”, 3.14, “Mark”, “Ann”]

index : 0 1 2 3 4 ➔ $Index_{max} = Length - 1$

- Indexed by integer, starting with zero:

```
a = list1[1]          # returns the 2nd item “David” ; a = “David”
list1[0] = “John”     # changes the 1st item 1 to “John” ; list1 = [“John”, “David”, 3.14, “Mark”, “Ann”]
```

- Empty list is created by:

```
list2 = []            # an empty list
list2 = list()        # an empty list
```

- Append and insert new items to a list:

```
list1.append(7)       # appends a new item to the end ; list1 = [“John”, “David”, 3.14, “Mark”, “Ann”, 7]
list1. insert(2, 0)   # inserts a new item into a middle ; list1 = [“John”, “David”, 0, 3.14, “Mark”, “Ann”, 7]
```

Basic Built-in Data Types

- Extract and reassign a portion of a list by slicing operator [i, j], with an index range of i<=k<j:

```
a = list1[0:2]          # returns [“John”, “David”] ; the 3rd item 0 is NOT extracted!
b = list1[2:]           # returns [0, 3.14, “Mark”, “Ann”, 7]
list1[0:2] = [-3, -2, -1]   # replaces the first two items with the list on the right
                        # list1 = [-3, -2, -1, 0, 3.14, “Mark”, “Ann”, 7]
```

- Delete items:

```
del list1[0]            # deletes the 1st item ; list1 = [-2, -1, 0, 3.14, “Mark”, “Ann”, 7]
del list1[0:4]          # delete a slice of the first 4 items ; list1 = [“Mark”, “Ann”, 7]
```

- Concatenate and multiply lists:

```
list2 = [8, 9]          # creates a new list
list3 = list1 + list2   # list3 = [“Mark”, “Ann”, 7, 8, 9]
list4 = list1 * 3       # list4 = [“Mark”, “Ann”, 7, “Mark”, “Ann”, 7, “Mark”, “Ann”, 7]
```

Basic Built-in Data Types

- Count occurrences of items:

```
list4.count("Mark")          # returns 3
```

- Remove an item from a list:

```
list1.remove("Ann")          # Search for "Ann" and remove it from list1 ; list1 = ["Mark", 7]
```

- Sort a list in place:

```
list5 = [10, 34, 7, 8, 9]    # creates a new list
list5.sort()                 # list5 = [7, 8, 9, 10, 34]
```

- Reverse a list in place:

```
list5.reverse()              # list5 = [34, 10, 9, 8, 7]
```

- Copy a list (*shallow copy*):

```
list6 = list(list5)          # list6 is a shallow copy of list5
```

Basic Built-in Data Types

- **Tuple**: A immutable record of arbitrary objects of any type

t1 = (1, “David”, 3.14, “Mark”, “Ann”)

index : 0 1 2 3 4

- Indexed by integer, starting with zero:

```
a = t1[1]          # returns the 2nd item “David” ; a = “David”
t1[0] = “John”     # Wrong operations! Tuple is immutable!
```

- 0-tuple (empty tuple) and 1-tuple:

```
t2 = ()            # an empty tuple ; same as t2 = tuple()
t3 = (“apple”,)    # a tuple containing 1 item ; note the trailing comma!
```

- Extract a portion of a list by slicing operator [i, j], with an index range of i<=k<j:

```
a = t1[0:2]        # returns (1, “David”) ; the 3rd item 3.14 is NOT extracted!
b = t1[2:]         # returns (3.14, “Mark”, “Ann”)
```

Basic Built-in Data Types

- Concatenate and multiply tuples:

```
t4 = t1 + t3          # t4 = (1, “David”, 3.14, “Mark”, “Ann”, “apple”)
t5 =  t3 * 3          # t5 = (“apple”, “apple”, “apple”)
```

- Count occurrences of items:

```
t5.count(“apple”)     # returns 3
```

- Extract values in a tuple without using index:

```
t6 =  (1, 2, 3)                          # create a new tuple
a, b, c = t6                              # a = 1 ; b = 2 ; c = 3
person = (“John”, “Smith”, 30)            # another example
first_name, last_name, age = person       # first_name = “John” ; last_name = “Smith” ; age = 30
```

Basic Built-in Data Types

- **String**: A immutable sequence of characters

s = “HELLO”

index : 0 1 2 3 4

- To create a string, enclose characters in single(‘ ’), double(“ ”), or triple(“““ ””” or ‘‘‘ ’’’) quotes:

```
a = ‘Mark’                  # ‘ ’ is usually for short strings
b = “Python is good!”       # “ ” is usually for string messages to be visible to human
c = “““This function        # “““ ””” or ‘‘‘ ’’’ is usually for Python doc strings ; can be used for a string
is for                      # spanning multiple lines
calculation of PI”””

d = ‘we say “yes!”’         # same type of quotes used to start a string must be used to terminate it!
d = “we say ‘yes!’”
d = “““we say ‘yes!’”””
d = ‘‘‘we say “yes!”’’’
```

Basic Built-in Data Types

- Indexed by integer, starting with zero:

```
a = “Hello Python!”      # a string a[0] = ‘H’ , a[1] = ‘e’ , a[2] = ‘l’ , a[3] = ‘l’ , …… , a[11] = ‘n’ , a[12] = ‘!’
b = a[4]                 # b = ‘o’
```

- Extract a portion of a string by slicing operator [i, j], with an index range of i<=k<j:

```
b = a[0:5]               # b = ‘Hello’
b = a[6:]                # b = ‘Python!’
b = a[4:7]               # b = ‘o P’
```

- Concatenate and multiply strings:

```
c = “My name is John.”   # a new string
d = a + ’ ’ + c          # d = “Hello Python! My name is John.”
d = a * 2                # d = “Hello Python!Hello Python!”
```

Basic Built-in Data Types

- Conversion between numbers and strings :

```
a = '77' ; b = '23'          # two numeric strings
c = a + b                    # c = '7723' ; string concatenation ; NO numeric evaluation!
c = int(a) + int(b)          # c = 100
c = float(a) + int(b)        # c = 100.0

i = 77 ; f = 23.0            # two numbers
a = str(i)                   # a = '77'
b = str(f)                   # b = '23.0'
```

- Common string methods:

Next Page!

Basic Built-in Data Types

s = “python is good!”

String Methods	Description	Examples
s.capitalize()	Capitalize the 1st character	“Python is good!”
s.center(*w*, *p*) s.ljust(w, p) s.rjust(w, p)	Centers s in a field of length *w*, padding with *p* Left-align/Right-align s with *w* and *p*	(w=30, p=‘-’) : -------python is good!-------- python is good!----------
s.count(*substr*)	Counts occurrences of *substr*	s.count(‘o’) returns 3
s.isalpha() s.isdigit() s.isalnum() s.islower() s.isupper()	True if all characters in s are alphabetic/digits/alphanumeric/lowercase/uppercase	s.isalpha() returns True s.islower() returns True
s.find(*substr*)	Finds the 1st occurrence of *substr* or returns -1	s.find(‘good’) returns 10
s.index(*substr*)	Finds the 1st occurrence of *substr* or raises an error	s.index(‘good’) returns 10
s.replace(old, new)	Replaces a substring	s.replace(‘good’, ‘bad’) returns “python is bad!”
s.split(*sep*)	Splits a string using *sep* as a delimiter	s.split(‘is’) returns ['python ', ' good!']
s.partition(*sep*)	Partitions a string based on *sep*; returns (head, sep, tail)	s.partition(‘is’) returns ('python ', 'is', ' good!')

Basic Built-in Data Types

- Built-in operations common to all sequences: list, tuple, and string

s = “python is good!”

list1 = [0, 1, 2, 3, 4]

Operations	Description	Examples
seq[i] seq[i:j]	Returns the element at index i Returns a slice with an index range of i<=k<j	s[0] returns ‘p’ s[0:6] returns ‘python’
len(seq)	Number of elements in seq	len(s) returns 15
min(seq)	Minimum value in seq	min(s) returns ‘ ’
max(seq)	Maximum value in seq	max(s) returns ‘y’
sum(seq)	Sum of items in seq ; ONLY working for numeric list or tuple!	sum(list1) returns 10
all(seq)	True if all items in seq are True	all(list1) returns False
any(seq)	True if any item in seq is True	any(list1) returns True

Program Structure: Control Flow and Loop

- Control Flow:

```
if expression:
    statements
elif expression:
    statements
……
else:
    statements
```

E.g. 1:

```
if a < 0:
    print “a is negative”
elif a == 0:
    print “a is zero”
else:
    print “a is positive”
```

E.g. 2:

```
if a < b:
    minvalue = a
else:
    minvalue = b
```

E.g. 3

```
if name != “Zhuofei”:
    pass                                # do nothing
else:
    print “Hello, Zhuofei!”
```

Note: Examples are for Python2

Program Structure: Control Flow and Loop

- while loop:

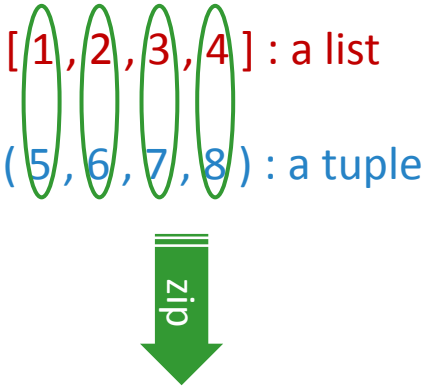
```
while expression:
    statements
```

E.g. :

```
# s and t are two sequences
i = 0
while i < len(s) and i < len(t):
    x = s[i]
    y = t[i]
    print x + y
    i += 1
```

s = [1, 2, 3, 4] : a list
t = (5, 6, 7, 8) : a tuple

Hi, this is Not Python style!



- for loop:

```
for i in seq:
    statements
```

E.g. :

```
# s and t are two sequences
for x, y in zip(s, t):
    print x + y
```

Note: Examples are for Python2

Function: Procedural and Functional Programming

- Function:

```
def functionName (params):
    statements
```

E.g. 1:

```
def f(x, y=0):                  # y has a default value of 0
    return x + y                # return a value

f(10)                           # returns 10
f(10, 2)                        # returns 12
```

E.g. 2

```
def f(x, y=0):                  # y has default value of 0
    return (x+y, x–y, x*y, x**y)  # return a tuple

v1, v2, v3, v4 = f(10)          # v1=10, v2=10, v3=0, v4=1
v1, v2, v3, v4 = f(10, 2)       # v1=12, v2=8, v3=20, v4=100
```

Function: Procedural and Functional Programming

- Procedural Programming Example:

principal.txt:

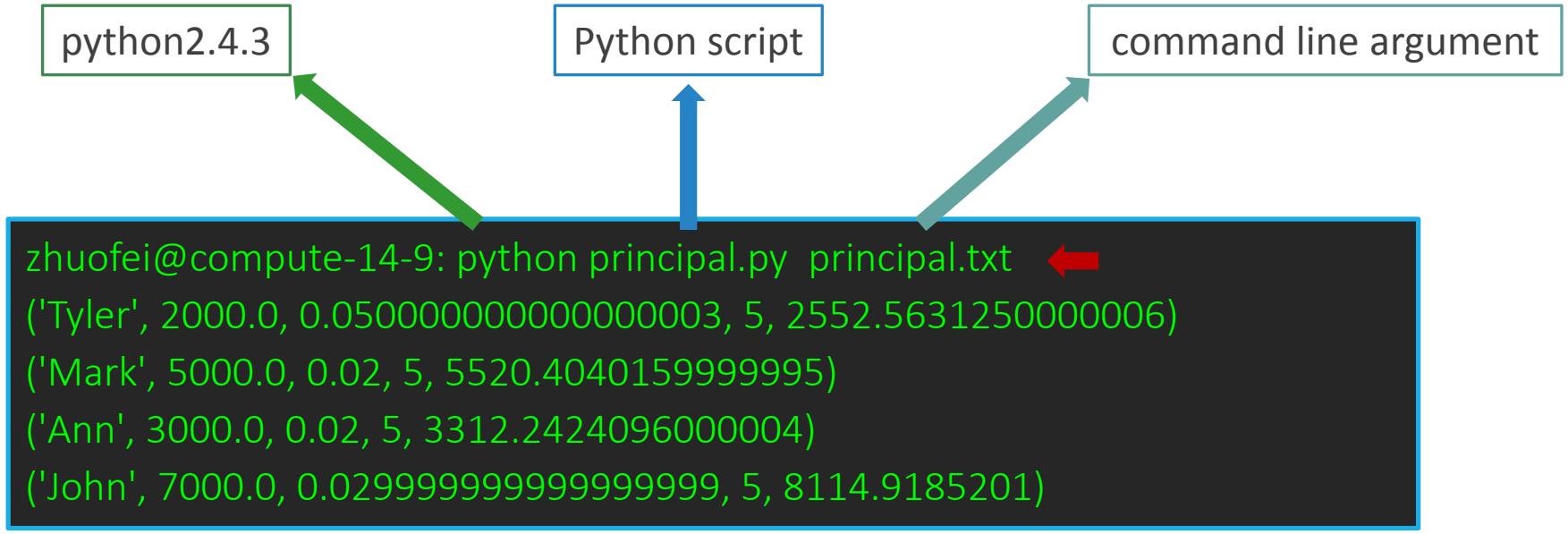
Tyler, 2000, 0.05, 5
Mark, 5000, 0.02, 5
Ann, 3000, 0.02, 5
John, 7000, 0.03, 5

Next page to run!

```
Import sys                                  # load the sys module ; NO worry, we'll talk about it on next class!
def calPrincipal(portfolio):
    """ Functions: 1. Read 4-column data line by line from a file: Name, Initial_Principal, Interest_Rate, Years
                   2. Calculate final principal for each Name
                   3. Store 5-column data as a record into a list """
    del portfolio[0:]                       # clear the storing list
    f = open(sys.argv [1], 'r')             # open a file given as the 1st parameter on the command line
    for line in f.readlines():              # read all lines ; return a list ; the ending '\n' of each line is also read
        fields = line.split(',')            # split each line using ',' as a delimiter ; return a list

        name = fields[0].strip()            # remove leading and trailing whitespace
        iniPrincipal = float(fields[1].strip())
        principal = iniPrincipal
        rate = float(fields[2].strip())
        years = int(fields[3].strip(' \n'))     # remove leading and trailing whitespace and '\n'

        year = 1
        while year <= years:                # calculate final principal for each person on each line
            principal =  principal * (1+rate)
            year += 1

        portfolio.append((name, iniPrincipal, rate, years, principal))     # store record in the list

portfolio = []                              # create the storing list
calPrincipal(portfolio)                     # call the function
for t in portfolio: print t                 # output to screen ; yes, you can put them on the same line
```

Function: Procedural and Functional Programming

- Run on zcluster's interactive nodes (qlogin) with default python2.4.3:



Function: Procedural and Functional Programming

- Functional Programming 101 – function Object: function itself is a data!

```
def square(x):                          # a simple regular function
    return x*x

def g(func):                            # g is taking a function as a parameter, i.e., function itself is a data!
    return func(10)

result = g(square)                      # result = 100
```

- Functional Programming 101 – Decorator: a function wrapper to enhance/alter the behavior of the function object being wrapped

```
def myDecorator(func):                  # wrapper
    print("Hello, I am more human friendly!")
    return func

@myDecorator                            # special @ symbol means square is going to be wrapped by myDecorator
def square(x):
    return x*x

                                        # here I am calling the wrapped square, and output:
                                        #       Hello, I am more human friendly!
print(square(10))                       #       100
```

Function: Procedural and Functional Programming

- Functional Programming 101 – Generator: a function using yield keyword to produce a sequence of values for use in iteration

```
def countdown(n):
    while n > 0:
        yield n                    # a function using yield keyword is a generator to produce a value sequence
        n -= 1
    return                         # generator can only return None

c = countdown(10)                  # define a generator object

v1 = c.next()                      # v1 = 10 ; next method of a generator is to produce a value each time it's called
v2 = c.next()                      # v2 = 9
v3 = c.next()                      # v3 = 8

for v in c:                        # normally we use a generator in a for loop
    print v                        # Output:
                                   # 7
                                   # 6
                                   # ……
                                   # 1
```

Thank You!

Let’s talk about
Python class, *module*, *package*
on next class!