Python Language Basics I

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

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

Who Are We?

- **Georgia Advanced Computing Resource Center**
- 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](http://gacrc.uga.edu) (Web)
- [http://wiki.gacrc.uga.edu](http://wiki.gacrc.uga.edu) (Wiki)
- [http://gacrc.uga.edu/help/](http://gacrc.uga.edu/help/) (Web Help)
- [https://wiki.gacrc.uga.edu/wiki/Getting_Help](https://wiki.gacrc.uga.edu/wiki/Getting_Help) (Wiki Help)
## GACRC Users September 2015

<table>
<thead>
<tr>
<th>Colleges &amp; Schools</th>
<th>Depts</th>
<th>PIs</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Franklin College of Arts and Sciences</td>
<td>14</td>
<td>117</td>
<td>661</td>
</tr>
<tr>
<td>College of Agricultural &amp; Environmental Sciences</td>
<td>9</td>
<td>29</td>
<td>128</td>
</tr>
<tr>
<td>College of Engineering</td>
<td>1</td>
<td>12</td>
<td>33</td>
</tr>
<tr>
<td>School of Forestry &amp; Natural Resources</td>
<td>1</td>
<td>12</td>
<td>31</td>
</tr>
<tr>
<td>College of Veterinary Medicine</td>
<td>4</td>
<td>12</td>
<td>29</td>
</tr>
<tr>
<td>College of Public Health</td>
<td>2</td>
<td>8</td>
<td>28</td>
</tr>
<tr>
<td>College of Education</td>
<td>2</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>Terry College of Business</td>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>School of Ecology</td>
<td>1</td>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>School of Public and International Affairs</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>College of Pharmacy</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td><strong>Centers &amp; Institutes</strong></td>
<td><strong>40</strong></td>
<td><strong>214</strong></td>
<td><strong>970</strong></td>
</tr>
</tbody>
</table>

**TOTALS:** 49  233  1029
## GACRC Users September 2015

<table>
<thead>
<tr>
<th>Centers &amp; Institutes</th>
<th>Pls</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center for Applied Isotope Study</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Center for Computational Quantum Chemistry</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Complex Carbohydrate Research Center</td>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td>Georgia Genomics Facility</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Institute of Bioinformatics</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Savannah River Ecology Laboratory</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>Skidaway Institute of Oceanography</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Center for Family Research</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Carl Vinson Institute of Government</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>19</strong></td>
<td><strong>59</strong></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Version</th>
<th>Installation Path</th>
<th>Invoke command</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.3 (default)</td>
<td>/usr/bin</td>
<td>python</td>
</tr>
<tr>
<td>2.7.2</td>
<td>/usr/local/python/2.7.2</td>
<td>python2.7</td>
</tr>
<tr>
<td>2.7.8</td>
<td>/usr/local/python/2.7.8</td>
<td>/usr/local/python/2.7.8/bin/python</td>
</tr>
<tr>
<td>3.3.0</td>
<td>/usr/local/python/3.3.0</td>
<td>python3</td>
</tr>
<tr>
<td>3.4.0</td>
<td>/usr/local/python/3.4.0</td>
<td>python3.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Installation Path</th>
<th>Invoke command</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.6 (default)</td>
<td>/usr/bin</td>
<td>python</td>
</tr>
<tr>
<td>2.7.8</td>
<td>/usr/local/apps/python/2.7.8</td>
<td>module load python/2.7.8 ; python</td>
</tr>
<tr>
<td>3.4.3</td>
<td>/usr/local/apps/python/3.4.3</td>
<td>module load python/3.4.3 ; python3</td>
</tr>
</tbody>
</table>

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):

  ```bash
  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
  
  myScript.py:
  ```
  ```python
  import sys
  print sys.version
  a = 7
  e = 2
  print a**e
  ```

  ```bash
  [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
  ```
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
  ```

- myScript.py:

  ```python
  #!/usr/local/python/2.7.2/bin/python
  import sys
  print sys.version
  a = 7; e = 2
  print a**e
  ```

  ```python
  #!/usr/bin/env python
  import sys
  print sys.version
  a = 7; e = 2
  print a**e
  ```

  tell system where the python lives

  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/)
- 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:

```python
# Use numpy and matplotlib
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 100, 15
x = np.random.randn(10000)

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

# Add a 'best fit' line
y = np.sqrt(x, 50, normpdf(bins, mu, sigma)
plt.plot(bins, y, 'r--', linewidth=1)

plt.xlabel('Scores')
plt.ylabel('Probability')
plt.title('Histogram of IQ: M = 100, S = 15')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```
### Scientific Python Distributions

- **Anaconda** is installed on GACRC zcluster and Sapelo:

<table>
<thead>
<tr>
<th>Version</th>
<th>Installation Path</th>
<th>Python Version</th>
<th>Export (2.3.0 as example)</th>
<th>Invoke Command</th>
</tr>
</thead>
</table>
| 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          |                                                                                          |                |

<table>
<thead>
<tr>
<th>Version</th>
<th>Installation Path</th>
<th>Python Version</th>
<th>Module Load (2.2.0 as example)</th>
<th>Invoke Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.0</td>
<td>/usr/local/apps/anaconda/2.2.0</td>
<td>2.7.11</td>
<td>module load anaconda/2.2.0</td>
<td>python</td>
</tr>
<tr>
<td>2.5.0</td>
<td>/usr/local/apps/anaconda/2.5.0</td>
<td>2.7.11</td>
<td></td>
<td>python</td>
</tr>
<tr>
<td>3-2.2.0</td>
<td>/usr/local/apps/anaconda/3-2.2.0</td>
<td>3.4.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
General Lexical Conventions

• A code sample:

```python
x = 10; y = "Hello!"
z = 3.14
# this is a comment
# 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 indentation 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

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

- **List**: A **mutable** sequence of arbitrary objects of any type
  
  ```python
  list1 = [1, "David", 3.14, "Mark", "Ann"]
  ```

  ```plaintext
  index : 0 1 2 3 4 ➔ Index_{max} = Length – 1
  ```

  - Indexed by integer, starting with **zero**:
    ```python
    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:
    ```python
    list2 = []  # an empty list
    list2 = list()  # an empty list
    ```

  - Append and insert **new items** to a list:
    ```python
    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 $i \leq k < j$:

  ```python
  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:

  ```python
  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:

  ```python
  list2 = [8, 9]  # creates a new list
  list3 = list1 + list2  # list3 = ["Mark", "Ann", 7, 8, 9]
  ```
Basic Built-in Data Types

- Count occurrences of items:
  ```python
  list4.count("Mark")  # returns 3
  ```

- Remove an item from a list:
  ```python
  list1.remove("Ann")  # Search for "Ann" and remove it from list1; list1 = ["Mark", 7]
  ```

- Sort a list in place:
  ```python
  list5 = [10, 34, 7, 8, 9]
  list5.sort()  # creates a new list
  list5 = [7, 8, 9, 10, 34]
  ```

- Reverse a list in place:
  ```python
  list5.reverse()  # list5 = [34, 10, 9, 8, 7]
  ```

- Copy a list (shallow copy):
  ```python
  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:
  ```
  ```python
  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  
  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

  \[ s = \text{"HELLO"} \]

  \[
  \begin{array}{c}
  \text{index} \\
  0 \\
  1 \\
  2 \\
  3 \\
  4 \\
  \end{array}
  \]

  ➢ 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 is for 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!”””
  d = ‘’’we say “yes!”’”’
Basic Built-in Data Types

- Indexed by integer, starting with **zero**:
  
  ```python
  b = a[4]             # b = 'o'
  ```

- Extract a portion of a string by **slicing operator [i, j]**, with an index range of \( i \leq k < j \):
  
  ```python
  b = a[0:5]            # b = 'Hello'
  b = a[6:]             # b = 'Python!'  
  b = a[4:7]            # b = 'o P'
  ```

- Concatenate and multiply strings:
  
  ```python
  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

### String Methods

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

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

- **Control Flow:**
  ```python
  if expression:
      statements
  elif expression:
      statements
      ......
  else:
      statements
  ```

- **E.g. 1:**
  ```python
  if a < 0:
      print "a is negative"
  elif a == 0:
      print "a is zero"
  else:
      print "a is positive"
  ```

- **E.g. 2:**
  ```python
  if a < b:
      minvalue = a
  else:
      minvalue = b
  ```

- **E.g. 3**
  ```python
  if name != "Zhuofei":
      pass
  # do nothing
  else:
      print "Hello, Zhuofei!"
  ```

Note: Examples are for Python2
Program Structure: Control Flow and Loop

• while loop:

```python
while expression:
    statements
```

E.g. :

```python
# 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
```

Note: Examples are for Python2

• for loop:

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

E.g. :

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

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

Hi, this is Not Python style!

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

[(1, 5), (2, 6), (3, 7), (4, 8)]
Function: Procedural and Functional Programming

• Function:

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

E.g. 1:

```python
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:

```python
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:**

  ```python
  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()
          iniPrincipal = float(fields[1].strip())
          principal = iniPrincipal
          rate = float(fields[2].strip())
          years = int(fields[3].strip(' \
'))
          # remove leading and trailing whitespace
          year = 1
          while year <= years:
              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
  
  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!
Function: Procedural and Functional Programming

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

```
python2.4.3 Python script command line argument
```

```
zhuofei@compute-14-9: python principal.py principal.txt
('Tyler', 2000.0, 0.05000000000000003, 5, 2552.5631250000006)
('Mark', 5000.0, 0.02, 5, 5520.4040159999995)
('Ann', 3000.0, 0.02, 5, 3312.2424096000004)
('John', 7000.0, 0.029999999999999999, 5, 8114.9185201)
```
Function: Procedural and Functional Programming

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

```
def square(x):
    return x**x

def g(func):
    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):
    print("Hello, I am more human friendly!")
    return func

@myDecorator
def square(x):
    return x**x

print(square(10))
# Hello, I am more human friendly!
# 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

```python
def countdown(n):
    while n > 0:
        yield n
    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* and scientific programming with *NumPy, SciPy*... on next class!