Persistent

Overview of Python

Pre-requisites

- Basic knowledge of any programming language
- Basic knowledge of Linux

Objectives

At the end of this first module, you will be able to:

- Get an overview of Python language
- Understand the features of Python language
- Installing and running Python programs
- Development environments
- Data types, variable,
- Conditional statements/Control Structures
- Operators
- Basic Data Structures in Python

Persistent

Python

Python is an interpreted general-purpose high-level programming language. Python aims to combine “remarkable power with very clear syntax”, and its standard library is large and comprehensive

What is Python?

- Python is an open source programing language with strong similarities to Perl, but with powerful typing and object-oriented features
- Developed by Guido Van Rossum in the early 1990s and named after Monty Python
- Python is an Interpreter language that helps quick development of applications and helps integrating systems more effectively
- Commonly used for producing HTML content on websites. Great for text files
- Clean syntax, powerful extensions
- Useful built-in types (lists, dictionaries)

Why Python?

- Easy to learn
- Easy to use, widely used
- Object oriented and portable
- Relatively fast
- C is much faster but much harder to use
- Java is about as fast and slightly harder to use
- Perl is slower, is as easy to use, but is not strongly typed
- Big list of available libraries to help raid development of large enterprise systems

Python features

- No compiling or linking
- No type declarations
- Automatic memory management
- High-level data types and operations
- Object-oriented programming
- Interactive, dynamic nature
- Rapid development cycle
- Simpler, shorter, more flexible
- Garbage collection
- Relatively fast
- Embedding and extending in C
- Access to interpreter information

Uses of Python

- Shell tools
  - System admin tools, command line programs
- Extension-language work
- Rapid prototyping and development
- Language-based modules
  - Instead of special-purpose parsers
- Graphical user interfaces
- Database access
- Distributed programming
- Internet scripting

Where can I get it?

- Many web hosts and Unix/Linux machines already have Python 3.x installed and running, so check with your host or system administrator first
- If you're installing on a local machine, go to http://www.python.org/getit/ find the version for your operating system and click on the corresponding download link
- For example https://www.python.org/downloads/
- To download Python 3.x Installer for Windows
- Python **3.x** is the recommended version for this course

How can I tell if it’s installed?

On UNIX:

Open a Unix/Linux terminal using putty connection and type python at the prompt.

Python prompts with ‘>>>’.

On Windows:

Open a command prompt and type python at the prompt. If Python is installed correctly, you should see the version information.

Python 3.7.3 (default, Apr 10, 2012, 22.71:26) [MSC v.1500 32 bit (Intel)] on win32 Type "help", "copyright", "credits" or "license" for more information.

>>>

Python interpreter: Running interactively on UNIX/Windows

Open a Unix/Linux terminal using putty or on windows, type

% python

Python prompts with '>>>'

Enter the following at the Python prompt >>>

>>> 3+3

6

>>> **print("Hello")**

Hello

To exit Python prompt, type **CONTROL-D**

Demo

Steps on Windows:

- Use any text editor to write above script in a file named ‘hello_world.py’ (save it for example D:\python_workshop)

- Run it on the DOS prompt as:
  D:\python_workshop >python hello_world.py

First Python script (contd.)

Steps on UNIX:

- Open a Unix/Linux terminal using putty connection
- Make a python_workshop directory in your home dir
- Move into the python_workshop directory
- Create a file named ‘hello_world.py’
- Open the file in your text editor
- Code below program
- Save the file
- Make the program executable (use chmod Unix command)
- Test the program on the terminal as – % python **hello_world.py**

First Python script

In this first Python script:

```
#!/usr/bin/python
print ("Hello world!“)
```

Line 1: Shebang line

Pre-defined Python function – print

Development environments

IDLE

- Is a GUI development environment
- Shell for interactive evaluation
- Text editor with color-coding and smart indenting for creating Python files
- Menu commands for changing system settings and running files

Eclipse

- Install PyDev eclipse plugin
- http://pydev.org/manual_101_install.html

Python Interpreter

Python Basic Fundamentals

Variables

Variable: A named piece of memory that can store a value.

Python throws an error when trying to access a variable before it has been properly created.

```
>>> y
Traceback (most recent call last):
File "<pyshell#16>", line 1, in -toplevel-
y`
NameError: name ‘y' is not defined
>>> y = 3
>>> y
3
```

Assignment statement: Stores a value into a variable. A variable that has been given a value can be used in expressions.

Basic data types

Integers (default for numbers)

```
z = 5 / 2 # Answer is 2, integer division.
```

Floats

```
x = 3.456
```

Strings

Can use **“”** or **‘’** to specify. **“abc” ‘abc’** (same thing)

Unmatched ones can occur within the string. **“matt’s”**

Use triple double-quotes for multi-line strings or strings than contain both **‘** and **“** inside of them:

“““a’b”c”””

Naming rules

Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores

Bob Bob _bob _2_bob_ bob_2 BoB

There are some reserved words:

and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while

Numbers

- **int(x)** converts x to an integer
- **float(x)** converts x to a floating point
- The interpreter shows
  a lot of digits

```
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

- Built into Python
- Same operations are supported as integer and float

```
>>> x = 3 + 2j
>>> y = -1j
>>> x + y
(3+1j)
>>> x * y
(2-3j)
```

Python basic operators

Python language supports the following types of operators:

- Arithmetic operators
- Comparison (i.e., relational) operators
- Assignment operators
- Logical operators
- Bitwise operators
- Membership operators
- Identity operators
- Conditional operators

Python arithmetic operators

Assume variable a holds value 10 and variable b holds value 20. The variable a and b are called as operands.

Operator	Description	Example
+	Addition	a + b will give 30
-	Subtraction	a – b will give -10
*	Multiplication	a * b will give 200
/	Division	b / a will give 2
%	Modulus	b % a will give 0
**	Exponent	a**b will give 10 to the power 20
//	Floor Division	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

Python assignment operators

The following table shows few examples.

Operator	Description	Example
=	Simple assignment	c = a + b will assign value of a + b into c
+=	Add AND assignment	c += a is equivalent to c = c + a
-=	Subtract AND assignment	c -= a is equivalent to c = c - a
*=	Multiply AND assignment	c *= a is equivalent to c = c * a
/=	Divide AND assignment	c /= a is equivalent to c = c / a
%=	Modulus AND assignment	c %= a is equivalent to c = c % a

Python comparison operators

The following table shows few examples.

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true	(a == b) is not True
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true	(a != b) is True
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true	(a < b) is not False
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true	(a > b) is not True

Python logical operators

The following table shows few examples.

Operator	Description	Example
and	Called Logical AND operator. If both the operands are true, then then condition becomes true	(a and b) is true
or	Called Logical OR Operator. If any of the two operands are non-zero, then then condition becomes true	(a or b) is true
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false	Not (a and b) is false

Python Data Structures

Sequence types

1. **Tuple**
   - A simple **immutable** ordered sequence of items
   - Items can be of mixed types, including collection types
2. **Strings**
   - **Immutable**
   - Conceptually very much like a tuple
3. **List**
   - **Mutable** ordered sequence of items of mixed types

Strings

- String is an ordered collection of characters that stores and represents text-based information
- Strings in Python are **immutable sequences**
- There is no char type like in C++ or Java
- + is overloaded to do concatenation
- Can use single or double quotes, and three double quotes for a multi-line string

>>> x = 'hello'

>>> x = x + ' there'

>>> x

'hello there‘

>>> s = 'And me too!\nthough I am much longer\nthan the others :)‘

>>> print(s)

And me too!

though I am much longer

than the others :)‘

Single, double and triple quoted strings

- Single and double quotes are interchangeable as below:

  >>> ‘Python’, “Python”

- Empty literal: ‘’ or “”
- It allows you to embed a quote character of the other type inside a string as below:

  >>> “knight’s”,‘knight”s’

- Python automatically concatenates adjacent strings as shown below:

  >>>”Title” ‘ of’ “ the book”

- Triple-quoted strings “”” or ‘’’ are useful for programs that output strings with special characters, such as quote characters. Triple-quoted strings also are used for large blocks of text, because triple quoted strings can span multiple lines. By convention, docstrings are triple-quoted strings.

Escape sequences

- Escape sequence is a special byte code embedded into string, that can not be easily typed on a keyboard
- \ with one (or more) character(s) is replaced by a single character in the resulting string
- \n – Newline
- \t – Horizontal Tab

For example

```
>>> s='a\nb\tc'   # 5 characters!
'a\nb\tc'
>>> print s
a
b
c
```

Raw strings

>>>print (‘C:\temp\new.txt’)

>>>print (‘C:\\temp\\new.txt’)

- Raw string suppress escape
- Format: r“text” or r‘text’(R“text” or R‘text’)

>>>print (r‘C:\temp\new.txt’)

- Raw strings may be used for directory paths, text pattern matching.

Substrings and methods

- len(String) – returns the number of characters in the String
- str(Object) – returns a String representation of the Object
- join(seq) – Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string
- count(str, beg= 0,end=len(string)) – Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given

Indexing example

```
>>> s = '012345'
>>> s[3]
'3'
```

Slicing examples

```
>>> s[1:4]
'123‘

>>> s[2:]
'2345'

>>> s[:4]
'0123'
```

String formatting

- Similar to C’s printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation

>>> "One, %d, three" % 2

'One, 2, three‘

>>> **"%d, two, %s" % (1,3)**

'1, two, 3'

>>> **"%s two %s" % (1, 'three')**

'1 two three'

More string examples:

Example	Result	Comment
"hello"+"world"	"helloworld"	Example of concatenation
"hello"*3	"hellohellohello"	Example of repetition
"hello"[0]	"h"	Example of indexing
"hello"[-1]	"o"	Example of negative indexing from end
"hello"[1:4]	"ell"	Example of slicing
len("hello")	5	Example of size
"hello" < "jello"	True	Example of comparison
"e" in "hello"	True	Example of search

Lists

- Lists are **flexible, mutable container objects** that can hold an arbitrary number of python objects unlike strings that consist only of character data and are immutable
- The objects that you can place in a list can include standard types and objects as well as user-defined ones
- Lists can contain different types of objects and are more flexible than Python arrays (available through the external array module) because arrays are restricted to contain objects of a single type
- Lists indices start from zero, can be grown and shrunk. They can be taken apart or sliced and put together with other lists or concatenated
- Individual or multiple items can be inserted, updated, or removed as needed

Tuples

- Tuples in Python are extremely similar in nature to Lists, but immutable
- A Tuple is represented by a number of values separated by commas
- The only visible difference between tuples and lists is that **tuples use parentheses ( )** and **lists use square brackets [ ]**
- Both data structures differ functionally from each other in the fact that **Tuples are immutable** while **Lists are mutable**
- Even though Tuples are immutable, they can hold mutable data if needed
- Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for Lists, parentheses for Tuples default to Tuples

How to create and assign lists & tuples

- Lists are delimited by surrounding square brackets [ ]
- Creating and assigning lists are practically identical to lists, with the exception of empty tuples. These require a trailing comma (,) enclosed in the tuple delimiting parentheses (( ))

Example 1:

```
>>> listOne= [123, 'abc’, 4.56]
>>> print (listOne)
[123, 'abc', 4.56]

>>> tupleOne= (123, 'abc’, 4.56)
>>> print (tupleOne)
(123, 'abc', 4.56)
```

How to create and assign lists & tuples (contd.)

Example 2:

```
>>> listTwo= [4.56, ['inner', 'list‘]]
>>> print (listTwo )
[4.56, ['inner', 'list‘]]
>>> tupleTwo = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> print (tupleTwo)
(123, 'abc', 4.56, ['inner', 'tuple'], (7-9j))
```

Example 3:

```
>>> aListThatStartedEmpty = []
>>> print (aListThatStartedEmpty )
[]
>>> emptiestPossibleTuple = (None,)
>>> print (emptiestPossibleTuple       )
(None,)
```

How to access values in lists & tuples

- Values in Lists and in Tuples can be accessed using the square bracket [ ] along with the index or range of indices
- Trying to access a value by specifying an index out of range results in an “index out of range” Python error

Example 1:

```
>>> listOne= [123, 'abc’, 4.56]
>>> listOne[0]
123

>>> tupleOne= (123, 'abc’, 4.56)
>>> tupleOne[0]
123
```

How to access values in lists & tuples (contd.)

Example 2:

>>> listTwo= [123, ‘abc’, 4.56, ['inner', 'list‘]]

>>> listTwo[1:4]

[‘abc’,4.56, ['inner', 'list‘]]

>>> tupleTwo= [123, ‘abc’, 4.56, ['inner', 'list‘]]

>>> tupleTwo[1:4]

(‘abc’,4.56, ['inner', 'list‘])

Example 3:

>>> listThree= [123, 'abc’, 4.56, ‘def’]

>>> listThree[:3]

[123, ‘abc’, 4.56]

>>> tupleThree= (123, 'abc’, 4.56, ‘def’)

>>> tupleThree[:3]

(123, ‘abc’, 4.56)

How to update lists

New values can be added to a List with the **append()** method.

Example 1: Replacing a value in List

```
>>> listOne= [123, 'abc’, 4.56]
>>> listOne[2] = 'float replacer‘
>>> listOne
[123, ‘abc’, 'float replacer‘]
```

Example 2: Inserting a value at the end of a List

```
>>> listTwo= [123, ‘abc’, 4.56]
>>> listTwo.append(234)
 >>> listTwo
[123, ‘abc’, 4.56,234]
```

How to update lists (contd.)

Example 3: Inserting a value at a specific position

```
>>> listThree= [123, ‘abc’, 4.56]
>>> listThree.insert(2, ‘c’)
>>> listThree
[123, ‘abc’, ‘c’, 4.56]
```

Example 4: Inserting a value at a specific negative position

```
>>> listFour= [123, ‘abc’, 4.56]
>>> listFour.insert(-1, 234)
>>> listFour
[123, ‘abc’, 234, 4.56]
```

Example 5: Removing from a List based on a specific value

```
>>> listFive= [123, ‘abc’, 4.56]
>>> listFive.remove(4.56)
>>> listFive
[123, ‘abc’]
```

How to update lists (contd.)

Example 6: Return the last value

```
>>> listSix= [123, ‘abc’, 4.56]
>>> listSix.pop()
4.56

>>> listSix
[123, ‘abc’]
```

Example 7: Return the third value

```
>>> listSeven= [123, ‘abc’, 4.56]
>>> listSeven.pop(2)
4.56

>>> listSeven
[123, ‘abc’]
```

How to update tuples

Removing individual tuple elements is not possible though it is possible to delete an entire tuple with the del statement.

Example 1:

```
>>> tupleOne= [123, ‘abc’, 4.56]
>>> tupleTwo= [789, ‘def’, 2.24]
>>> tupleThird= tupleOne[0], tupleTwo[1]
>>> tupleThird
(123, ‘def’)
```

Example 2:

```
>>> tupleOne= (123, ‘abc’, 4.56)
>>> tupleOne
(123, ‘abc’, 4.56)
del tupleOne
```

Whole list operations

Example 1: Concatenates a list on to the existing list

```
>>> listOne= [123, ‘abc’, 4.56]
>>> listOne.extend([1, 2])
>>> listOne
[123, ‘abc’, 4.56, 1, 2]
```

Example 2: Reverse a List

```
>>> listTwo= [123, ‘abc’, 4.56]
>>> listTwo.reverse()
>>> listTwo
[4.56,  ‘abc’, 123]
```

The membership (in, not in) operator

- With lists and tuples the membership operator can be used to check if an object is a “member” of a list or a tuple
- The membership operators return a value of True or False depending on the existence or non-existence of the object in the list

Example 1:

>>> listOne= [123, ‘abc’, 4.56]

>>> ‘abc’ in listOne

True

Example 2:

>>> tupleOne= ([65535L, 2e+030, (76.45-1.3j)], -1.23, 16.0, -49)

>>> 16.0 not in tupleOne

False

The concatenation (+) operator

- It does not allow to concatenate two different types even if both are sequences
- extend() method can also be used in place of the concatenation operator to append the contents of a list to another. Using extend() is advantageous over concatenation in case of Lists because it actually appends the elements of the new list to the original in place, rather than creating a new list (with a new memory reference) from scratch like + does

Example:

```
>>> number_list = [43, -1.23, -2]
>>> string_list = [‘hello', ‘world']
>>> string_list + number_list
[‘hello', ‘world‘, 43, -1.23, -2]
```

The repetition (*) operator

Use of the repetition operator may make more sense with strings, but as a sequence type, lists and tuples can also benefit from this operation.

Example 1:

```
>>> number_list = [43, -1.23, -2]
>>> number_list * 2
[43, -1.23, -2, 43,-1.23, -2]
```

Example 2:

```
>>> number_tuple = (43, -1.23, -2)
>>> number_tuple * 3
(43, -1.23, -2, 43,-1.23, -2, 43,-1.23, -2)
```

List type built-in methods

- list.append(obj) appends object obj to list
- list.count(obj) returns count of how many times obj occurs in list
- list.extend(seq)[a] appends the contents of seq to list
- list.index(obj) returns the lowest index in list that obj appears
- list.insert(index, obj) inserts object obj into list at offset index
- list.pop(obj=list[-1])[a] removes and returns last object or obj from list
- list.remove(obj) removes object obj from list
- list.reverse() reverses objects of list in place
- list.sort([func]) sorts objects of list, use compare func if given

Set

- A set is an **unordered collection** with **no duplicate values.**
- A set can be created by using the keyword set or by using curly braces {}. However, to create an empty set you can only use the set construct, curly braces alone will create an empty dictionary. The set construct accepts one argument, a list

Example 1: Creating a Set using set construct

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

Example 2: Creating a Set using {}

```
>>> s = {1, 2, 3}
>>> s
{1, 2, 3}
```

Sets (contd.)

- Sets are used to eliminate duplicate values from within a list.

Example:

```
>>> listOne = [1, 2, 3, 3]

>>> s = set(listOne)

>>> s

{1, 2, 3}
```

- The way a set detects if a clash between non-unique elements has occurred is by indexing the data in memory, creating a hash for each element. This means that all elements in a set must be hashable.

Example:

```
>>> set ([1, [1,2]])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unhashable type: 'list'
```

Set operations

Example 1: Set Union

```
>>> setOne = set([1, 2, 3])
>>> setTwo = set([3, 4, 5])
>>> setOne | setTwo
{1, 2, 3, 4, 5}
```

Example 2: Set Intersection

```
>>> setOne = set([1, 2, 3])
>>> setTwo = set([3, 4, 5])
>>> setOne & setTwo
{3}
```

Dictionaries

- What makes dictionaries different from sequence type containers like lists and tuples is the way the data is stored and accessed
- Sequence types use numeric keys only (numbered sequentially as indexed offsets from the beginning of the sequence). Mapping types may use most other object types as keys, strings being the most common
- Dictionaries use **“keys”** that map directly to **values**. Keys can be any immutable type, but values can be of any type
- The most common data structure that maps keys with associated values are hash tables
- Hash tables are a data structure that store each piece of data, called a value, based on an associated data item, called a key. Together, these are known as key-value pairs
- Python dictionaries are implemented as resizable hash tables

How to create and assign dictionaries

- Creating dictionaries simply involves assigning a dictionary to a variable, regardless of whether the dictionary has elements or not
- A Dictionary can be created as empty or with elements
- Dictionaries are delimited by surrounding braces {}

Example:

```
>>> dictionaryOne = {}

>>> dictionaryTwo = {‘name': ‘python', ‘course': ‘moc’}

>>> dictionaryOne, dictionaryTwo

({}, {‘name': ‘python’, ‘course': ‘moc'})
```

How to access values in dictionaries

Values in dictionaries can be accessed using the square bracket [ ] along with the key.

Example 1:

```
>>> dictionaryOne = {‘name': ‘python', ‘course': ‘moc’}
>>> dictionaryOne[‘name’]
‘python’
```

Example 2:

```
>>> dictionaryTwo = {‘name': ‘python', ‘course': ‘moc’}
>>> dictionaryTwo [‘server’]
    Traceback (innermost last):
    File "<stdin>", line 1, in ?
    KeyError: server
```

How to access values in dictionaries (contd.)

- The best way to check if a dictionary has a specific key is to use in operator
- Operator –in will return True if a dictionary has that key and False otherwise.

Example 3:

```
>>> dictionaryThree = {‘name': ‘python', ‘course': ‘moc’}
>>> ‘server in ’dictionaryThree
False
```

Example 4:

```
>>> dictionaryFour = {‘name': ‘python', ‘course': ‘moc’}
>>> ‘name’ in dictionaryFour
True
```

How to update dictionaries

- A dictionary can be updated by adding a new entry or element (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry
- Assigning to an existing key replaces its value
- Keys must be unique

Example 1: Update an existing entry

```
>>> dictionaryOne = {‘name': ‘python', ‘course': ‘moc’}
>>> dictionaryOne['name'] = ‘java'
```

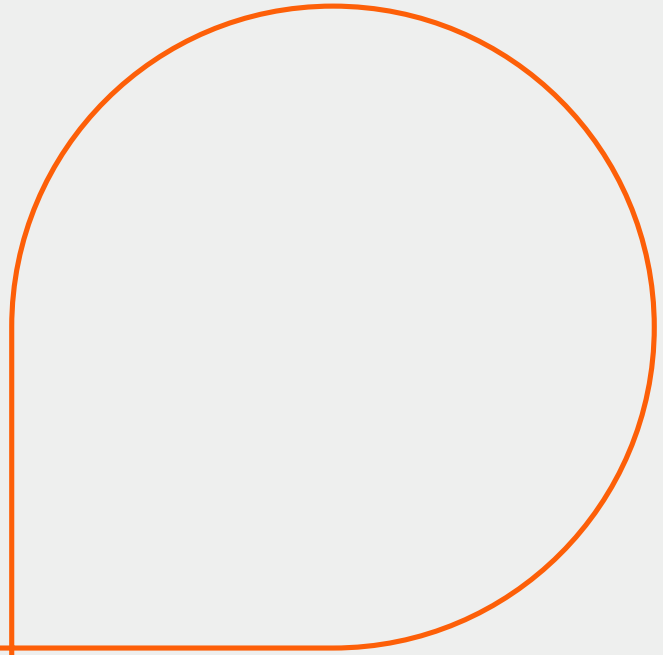
Example 2: Add a new entry

```
>>> dictionaryTwo = {‘name': ‘python', ‘course': ‘moc’}
>>> dictionaryTwo['arch'] = 'sunos5‘
>>> dictionaryTwo[‘arch’]
‘sunos5’
```

Dictionary built-in methods

- dict.clear() removes all elements of dictionary dict
- dict.copy() returns a (shallow) copy of dictionary dict
- dict.get(key, default=None) for key key, returns value or default if key not in dictionary (note that default's default is None)
- dict.items() returns a list of dict's (key, value) tuple pairs
- dict.keys() returns list of dictionary dict's keys
- dict.setdefault key, default=None) similar to get(), but will set
- dict[key]=default if key is not already in
- dict.update(dict2) adds dictionary dict2's key-values pairs to dict
- dict.values() returns list of dictionary dict's values

Control Structures



Overview of Python indentation

Code blocks in Python are identified by indentation rather than using symbols like curly braces.

Python employs indentation as a means of delimiting blocks of code. Code at inner levels are indented via spaces or TABs. Indentation requires exact indentation, in other words, all the lines of code in a suite must be indented at the exact same level (eg. same number of spaces). Indented lines starting at different positions or column numbers is not allowed; each line would be considered part of another suite and would more than likely result in syntax errors.

A new code block is recognized when the amount of indentation has increased, and its termination is signaled by a "dedentation," or a reduction of indentation matching a previous level. Code that is not indented, i.e., the highest level of code, is considered the "main“ portion of the script.

If statement

- The **if** statement in Python is made up of three main components: keyword itself, an expression which is tested for its truth value, and a code to execute if the expression evaluates to non-zero or true
- The syntax for an if statement is as shown below
- The syntax is as shown below

    if expression1: expr1_true_suite

    elif expression2: expr2_true_suite

    elif expressionN: exprN_true_suite

    else : default_suite

>>> a=2

>>> b=3

>>> if (a<b): print (“a is less than b”)

elif (a>b): print (“a is greater than b”)

else: (print “a and b are equal”)

Python while statement

- The suite in a while clause will be executed continuously in a loop until that condition is no longer satisfied
- The general syntax of the while loop is as below

```
while condition: body
else: post-code
```

Example:

```
>>> count = 0
>>> while (count < 3):
        print('the index is:', count)
        count = count + 1
```

Output of the above code is:

```
the index is: 0
the index is: 1
the index is: 2
```

Python for statement

- In Python, a for loop iterates over the values returned by any iterable object – that is, any object that can yield a sequence of values
- For example, a for loop can iterate over every element in a list, a tuple, or a string. The general syntax of the for statements is

  for item in sequence: body

  else: post-code
- The body will be executed once for each element of sequence. A variable is set to be the first element of sequence, and body is executed; then, variable is set to be the second element of sequence, and body is executed; and so on, for each remaining element of the sequence

Example:

```
>>> x = [1.0, 2.0, 3.0]
>>> for n in x: print(1 / n)
```

Output of the above code is:

```
1.0
0.5
0.333333333333
```

The range function

- Sometimes there arises a need to loop with explicit indices to use the position at which values occur in a list. The range function can be used in such scenarios
- The range command together with the len function can be used on lists to generate a sequence of indices for use by the for loop
- The following code example prints all the positions in a list where it finds negative numbers:

```
>>> x = [1, 3, -7, 4, 9, -5, 4]
>>> for i in range(len(x)):
        if x[i] < 0:
            print("Found a negative number at
                index ", i)
```

Output of the above code is:

```
('Found a negative number at index ', 2)
('Found a negative number at index ', 5)
```

The break statement

- The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in other programming languages
- The break statement can be used in both while and for loops

Example:

The following code can be placed in break_example.py and executed by typing python break_example.py in the Python console.

```
#!/usr/bin/python

for letter in ‘World’:

if letter == ‘l’:

                    break

print(Current Letter :', letter)

print ("Good bye!“)
```

List and dictionary comprehensions

- A simple example of when a list or dictionary comprehensions can be used is when there is a need to construct a new list based on the elements from another list.
- The pattern of a list comprehension is as follows:
- new_list = [expression for variable in old_list if expression]

Example:

```
>>> listOne = [1, 2, 3]
>>> new_listOne = [x*2 for x in listOne]
>>> new_listOne
[2, 4, 6]
```

List and dictionary comprehensions (contd.)

- The pattern of a dictionary comprehension is as follows

  new_dict = {expression:expression for variable in list if expression}

Example:

>>> dictionaryOne = [1, 2, 3, 4]

>>> dictionaryTwo = {item: item * item for item in dictionaryOne }

>>> dictionaryTwo

{1: 1, 2: 4, 3: 9, 4: 16}

- List and Dictionary comprehensions are very flexible and powerful and make list-processing operations much simpler

Summary :

With this we have come to an end of our first session, where we discussed about ….

- Get an overview of Python language
- Understand features of Python language
- Installing & Running Python programs
- Development Environments
- Data Types, variable, Basic data structures in Python

In the next session we will discuss about

- Python Functions

Persistent

Reference material

- http://www.tutorialspoint.com/python
- http://www.learnpython.org/

Questions

Key contacts

Sakshi Jamgaonkar

sakshi_jamgaonkar@persistent.com

Asif Immanad

asif_immanad@persistent.co.in

Persistent

Persistent

Thank you!

Persistent University