7 Learning & Programming Python

0.1 Introduction

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).This experiments gives enough understanding on Python programming language.

Python 3.0 was released in 2008. The final 2.x version 2.7 release came out in mid-2010, with a statement of extended support for this end-of-life release. The 2.x branch will see no new major releases after that. 3.x is under active development and has already seen over five years of stable releases, including version 3.3 in 2012, 3.4 in 2014, and 3.5 in 2015. This means that all recent standard library improvements, for example, are only available by default in Python 3.x. For more information, see this Link.

0.2Objective

The objectives of the experiment is to learn the following:

- Give a quick introduction about Python Programming.
- Python syntax.
- Show some examples about Python.
- Show some Python functions and Libraries.

0.3 Basic Concept

Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.
- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.
- **It can be easily integrated** with C, C++, COM, ActiveX, CORBA, and Java.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.

The following is a partial list of python libraries:

NumPy: it`s the fundamental package for scientific computing with Python. It contains among other things such as a powerful N-dimensional array object, tools for integrating C/C++ code, useful linear algebra, Fourier transform, and random number capabilities.
matplotlib: A numerical plotting library. It is very useful for any data scientist or any data analyzer.
Pygame: This library will help you achieve your goal of **2d game** development.
Requests: The most famous **http** library.
Twisted: The most important library for any **network** application developer.
Opencv: it`s a library of Python bindings designed to solve computer vision problems.
Collections: The most famous **data structure** library.

0.4 Download Python

Python runs on many operating systems such as MS-Windows, Mac OS, Mac OS X, Linux, FreeBSD, OpenBSD, Solaris, AIX, and many varieties of free UNIX like systems. Python comes with many Linux versions. The easiest way to install the Python is to use package manger such as apt-get, yum, and so on.

```
$ sudo apt-get install python
```

To see which version of Python you have installed, run the following commands in terminal:

```
$ python --version
```

To run Python just type:

```
$ python
```

0.5.1 Structure of a Python program

Python program is a text file. You can use any text editor to create the program such as gedit, emacs or even vi. Normally the following line will be the first line.

```
!/usr/bin/python
```

This tells Linux to use /usr/bin/python executable to interpret rest of the lines in the program. This line may vary depending on the location of python binary. Sometimes it may be /usr/local/bin/python or some other place.
Commonly **.py** extension is used (e.g. file.py), however you can write the python code without extension also. It will still work fine.

0.5.2 Comments

The symbol **#** is used for comments. All text from # till end of line is treated as comment.

```
# This is a full line comment
```

Note: There is multiline comment in Python Language.

```
‘’’
This is a multiline
Comment.
‘’’
```

0.6 Variable types

Python has five standard data types:

- Numbers
- String
- List
- Tuple
- Dictionary

0.6.1 Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them.

```
var1 = 2
var2 = 11
```

You can delete a single object or multiple objects by using the **del** statement.

```
del var
del var_a, var_b
```

Python supports four different numerical types:

- int (signed integers)
- Long
- float (floating point real values)
- complex (complex numbers)

Int	Float	Complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

- A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

Example:

```
Print("Numbers Types in  Python!")
a = 10
print(a)    #Print in Number
b = 5.7
print(b)    #Print Float Number
d = 3J
e = 2+4j
res =d+e    #add two complex numbers
print(res)  #Print Complex Number
```

Output:

```
Numbers Types in  Python!
10
5.7
(2+7j)
```

0.6.2 Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator (**[ ]** and **[:]**) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

```
str = 'Hello World!'
print(str)        # Prints complete string
print(str[0])     # Prints first character of the string
print(str[2:5])   # Prints characters starting from 3rd to 5th
print(str[2:])    # Prints string starting from 3rd character
print(str * 2)    # Prints string two times
print(str + "TEST") # Prints concatenated string
```

0.6.3 Python List

List contains items separated by commas and enclosed within square brackets **([])**. To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
smalllist = [123, 'john']
print(list)          # Prints complete list
print(list[0])       # Prints first element of the list
print(list[1:3])     # Prints elements starting from 2nd till 3rd
print(list[2:])      # Prints elements starting from 3rd element
print(smalllist * 2)  # Prints list two times
print(list + smalllist) # Prints concatenated lists
```

0.6.4 Python Tuple

Tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets (**[ ]**) and their elements and size can be changed, while tuples are enclosed in parentheses (**( )**) and cannot be updated. Tuples can be thought of as **read-only** lists.

Example:

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')
print(tuple)          # Prints complete list
print(tuple[0])       # Prints first element of the list
print(tuple[1:3])     # Prints elements starting from 2nd till 3rd
print(tuple[2:])      # Prints elements starting from 3rd element
print(tinytuple * 2)  # Prints list two times
print(tuple + tinytuple) # Prints concatenated lists
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists.

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

0.6.5 Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'Mohammad','code':6734, 'dept': 'sales'}
print(dict['one'])      # Prints value for 'one' key
print(dict[2])          # Prints value for 2 key
print(tinydict)         # Prints complete dictionary
print(tinydict.keys())  # Prints all the keys
print(tinydict.values()) # Prints all the values
```

Data type conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Function	Description
int(x ,[base])	Converts x to an integer. base specifies the base if x is a string.
long(x, [base])	Converts x to a long integer. base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real ,[imag])	Creates a complex number.

Example:

```
Print("Data type conversion !")
x ="101"
x = int(x,2)     #convert from string  to integer
print(x)
x=float(x)
print(x)         #convert to Float number
y=complex(2,3)   #Creates complex Number
print(y)
```

Output:

```
Data type conversion !
5
5.0
(2+3j)
```

0.7 Basic Operators

Python language supports the following types of operators

0.7.1 Python Arithmetic Operators

Assume variable **a** holds 10 and variable **b** holds 20 then:

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20

Example:

```
print("Python Arithmetic Operators !")
a = 10
b = 20
print(a+b)  #Addition
print(a-b)  #Subtraction
print(a*b)  #Multiplication
print(a/b)  #Division
print(a%b)  #Modulus
print(a**2) #Exponent
```

Output:

```
Python Arithmetic Operators !
30
-10
200
0
10
100
```

0.7.2 Python comparison Operators

Assume variable a holds 10 and variable b holds 20, then:

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

Example:

```
print "Python comparison Operators!"
a = 10
b = 20
print (a == b)
print (a != b)
print (a > b)
print (a < b)
print (a >= b)
print (a <= b)
```

Output:

```
Python comparison Operators!
False
True
False
True
False
True
```

0.7.3 Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a

Example:

```
print "Python Assignment Operators!"
a = 10
b = 20
d = 30
c=a+b
print (c)
a+=a
print (a)
b-=a
print (b)
d**=2
print (d)
```

Output:

```
Python Assignment Operators!
30
20
0
900
```

0.7.4 Python Bitwise Operation

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a=60, and b = 13.

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Example:

```
print "Python Bitwise Operation!"
a = 12
b = 9
res = a ^ b  #a=1100 b=1001, a^b=1100^1001=0101=5
print (res)
res = a | b  #a=1100 b=1001,a|b=1100|1001=1101=13
print (res)
res = a & b  #a=1100 b=1001,a$b=1100$1001=1000=8
print (res)
res = ~a
print (res)
res = b >>2
print (res)
```

Output:

```
Python Bitwise Operation!
5
13
8
-13
2
```

0.7.5 Python Logical Operation

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then:

Operator	Description	Example
AND	If both the operands are true then condition becomes true.	(a and b) is true.
OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
NOT	Used to reverse the logical state of its operand.	Not (a and b) is false.

Example:

```
print("Python Logical Operation")
a = 1
b = 0
print (a and b)
print (a or b)
print (not(a and b))
```

0.8 If Statement

```
if expression:
  statement(s)
else:
  statement(s)
```

Example:

```
var1 = 100
if var1:
  print("1 - Got a true expression value")
  print(var1)
else:
  print("1 - Got a false expression value")
  print(var1)
```

Note: There is **one-line if** clause in Python Language

```
var = 100
if ( var  == 100 ) : print("Value of expression is 100")
```

0.9 Loops

Python programming language provides following types of loops to handle looping requirements.

0.9.1 While Loop

The syntax of a **while** loop in Python programming language is:

```
while expression:
  statement(s)
```

Example:

```
count = 0
while (count < 10):
  print('The count is:', count)
  count = count + 1
```

Python supports to have an **else** statement associated with a loop statement.

```
count = 0
while count < 5:
  print(count, " is  less than 5")
  count = count + 1
else:
  print(count, " is not less than 5")
```

Note: There is **one-line while** clause in Python Language

```
flag = 1

while (flag): print('Given flag is really true!')

print("Good bye!")
```

0.9.2 For Loop

The syntax of a **while** loop in Python programming language is:

```
for iterating_var in sequence:
  statements(s)
```

Example:

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
  print('Current fruit :', fruits[index])
print ("Good bye!")
```

Using else Statement with Loops

```
for num in range(10,20):          #to iterate between 10 to 20
  for i in range(2,num):          #to iterate on the factors of the number
    if num%i == 0:                #to determine the first factor
      j=num/i                     #to calculate the second factor
      print('%d equals %d * %d' % (num,i,j))
      break                       #to move to the next number, the #first FOR
  else:                           # else part of the loop
    print(num, 'is a prime number')
```

0.10 Loop control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

0.10.1 Break statement

```
for letter in 'Python':
  if letter == 'h':
    break
  print('Current Letter :', letter)
```

0.10.2 Continue Statement

```
for letter in 'Python':
  if letter == 'h':
    continue
  print('Current Letter :', letter)
```

0.11 Input in Python

Input can come in various ways, for example from a files, keyboard, database, another computer, mouse clicks and movements or from the internet.

0.11.1 Reading Keyboard Input

Python provides built-in function to read a line of text from standard input, which by default comes from the keyboard. This function is **input().**

Example:

```
name = input("what`s your Name ?")
Print("Nice to meet you "+ name)
age =input("Your age ?")
print("your age is :"+age)
```

0.11.2 File Input/output

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

Open file Syntax:

```
file object = open(file_name ,[access_mode],[buffering])
```

Here are parameter details:

- **file_name:** The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode:** The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).
- **buffering:** If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

Here is a list of the different modes of opening a file :

Modes	Description
R	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
Rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
W	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
Wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
A	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
Ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The file Object Attributes

Here is a list of all attributes related to file object:

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

Example:

```
# Open a file
fo = open("foo.txt", "wb")
print("Name of the file: ",fo.name)
print( "Closed or not : ", fo.closed)
print ("Opening mode : ",fo.mode)
fo.close()
```

Output:

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
```

Close File:

```
# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
# Close opend file
fo.close()
```

Write to file:

Syntax:

```
fileObject.write(string);
```

Example:

```
# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language !!\n");
# Close opened file
fo.close()
```

Read from file:

Syntax:

```
fileObject.read([count]);
```

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example:

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print ("Read String is : "+str)
# Close opend file
fo.close()
```

The next example is to compute the average of the numbers in the file. Open file called **foo.txt** and add this content:

```
1
15
20
33
14
20.5
```

Run the following code after storing it in **fileName.py** files.

```
#Find average for file integers
#read line by line
Avg=0
Sum=0
i=0
with open('filename') as f:
        for line in f:
            Sum+=float(line)
            i=i+1
Avg=Sum/i
print(Avg)
f.close()
```

todo: write result on a file.

0.12 Python Examples

Example1**:**

```
# Getting the number of digits of nonnegative integers
num=input("Please, Enter the number: \n")
num=int(num)
result = 0
while num > 0:
      num = int(num / 10)
      result += 1
print("The number of digits = ")
print(result)
```

todo: find the sum of the digits.

Example2**:**

```
# Finding Average of a List
l=[1,2,3,4,5,6,7,8,9,10]
print (sum(l)/len(l))
```

todo: rewrite this code using **For** loop.

Example3:

```
# Finding Max Number of a List
l=[1,22,3,14,5,6,7,8,9,10]
maxN=l[0]
for i in range(len(l)):
  if l[i] > maxN:
     maxN=l[i]
print maxN
```

todo: find the Min number.

0.13 Todo:

This part will be given to you by the teacher assistant in the lab time.