8 Learning & Programming Python

0.1 Objective

The objectives of the experiment is to learn the following:

- Give a quick introduction about Python Programming.
- Define function in Python.
- Show some examples about Mathematical and String functions.
- Explain what are Modules in Python.
- Implement Data Structure types in Python.
- Show some examples about Recursion in Python.
- Learn how to use regular expression in Python

0.2 Python Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

0.2.1 Defining a function

You can define functions to provide the required functionality.

Syntax

```
def functionName( parameters ):
   "function documentation  string"
   function_suite
   return [expression]
```

Here are rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses **( )**.
- Any input parameters should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function.
- The code block within every function starts with a colon **:** .
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Example:

```
def printString( str ):
   "This prints a passed string into this function"
   print(str)
   return
```

0.2.2 Calling a function

You can execute a function by calling it from another function or directly from the Python prompt.

Example1:

```
# Function definition is here
def printString( str ):
   "This prints a passed string into this function"
   print(str)
   return;
# Now you can call printString function
printString("I'm first call to user defined function!")
printString("Again second call to the same function")
printString( str = "My string")
```

Output:

```
I`m first call to user defined function!
Again second call to the same function
My string
```

Example2:

```
def printinfo( name, age ):
   "This prints a passed info into this function"
   print("Name: "+name)
   print("Age :"+str(age))
   return;
# Now you can call printinfo function
printinfo( age=22, name="Mohammad" )
```

Output:

```
Name:  Mohammad
Age : 22
```

Example3:

```
# Function definition is here
def printinfo( name, age = 30 ):
   "This prints a passed info into this function"
   print("Name: "+name)
   print( "Age :"+str(age))
   return;
# Now you can call printinfo function
printinfo(name="mohammad" ,age=22)
printinfo( name="mohammad" )
```

Output:

```
Name:  mohammad
Age:  22
Name:  mohammad
Age : 30
```

Example4:

```
# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print ("Output is: ")
   print(arg1)
   for var in vartuple:
      print(var)
   return;
# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Output:

```
Output is:
10
Output is:
70
60
50
```

0.2.3 The help() function

Help function will give you the **docstring** of the function.

Example5:

```
my_list = []
help(my_list)
```

0.2.4 The return Statement

All the above examples are not returning any value. You can return a value from a function as follows:

Example 6:

```
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print("Inside the function : "+str(total))
   return total;
# Now you can call sum function
total = sum( 10, 20 );
print("Outside the function : "+str(total))
```

Output :

```
Inside the function :  30
Outside the function :  30
```

0.2.5 Pass by reference vs. value

All parameters in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Example7:

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([4,5]);
   print("Values inside the function: "+str(mylist))
   return
# Now you can call changeme function
mylist = [1,2,3];
changeme( mylist );
print ("Values outside the function: "+str(mylist))
```

Output:

```
Values inside the function:  [1, 2, 3, [4, 5]]
Values outside the function:  [1, 2, 3, [4, 5]]
```

Example8:

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   print("Values inside the function: "+str(mylist))
   return
# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print("Values outside the function: "+ str(mylist))
```

Output:

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

0.2.6 Scope of variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

There are two basic scopes of variables in Python:

- Global variables
- Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

Example 9:

```
total = 0; # This is global variable.
def sum( arg1, arg2 ):
  total = arg1 + arg2; # Add both the parameters and return them."
  print("Inside the function local total : ‘’+str(total))
  return total;
# Now you can call sum function
sum( 10, 20 );
print("Outside the function global total : "+str(total))
```

Output:

```
Inside the function local total :  30
Outside the function global total :  0
```

0.3 Mathematical functions

Python includes many functions that perform mathematical calculations. The following table contains some of them:

Function	Returns (description)
abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y
exp(x)	The exponential of x: e^x
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x
log(x)	The natural logarithm of x, for x> 0
log10(x)	The base-10 logarithm of x for x> 0 .
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity
modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
pow(x, y)	The value of x**y.
sqrt(x)	The square root of x for x > 0

Example 10:

```
import math
print ("abs(-45) : "+ str(abs(-45)))
print ("abs(100.12) : "+ str(abs(100.12)))
print ("math.ceil(-45.17) : "+ str(math.ceil(-45.17)))
print ("math.ceil(100.12) : "+ str(math.ceil(100.12)))
print ("math.ceil(100.72) : "+ str(math.ceil(100.72)))
print ("math.ceil(math.pi) : "+ str(math.ceil(math.pi)))
print ("max(-80, -20, -10) : "+ str(max(-80, -20, -10)))
```

Output:

```
abs(-45) : 45
abs(100.12) : 100.12
math.ceil(-45.17) : -45.0
math.ceil(100.12) : 101.0
math.ceil(100.72) : 101.0
math.ceil(math.pi) : 4.0
max(-80, -20, -10) : -10
```

0.4 String functions

Python includes functions to manipulate strings. The following table contains some of them:

Methods	Description
capitalize()	Capitalizes first letter of 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.
endswith(suffix,beg=0,end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
find(str, beg=0 end=len(string))	Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
index(str, beg=0, end=len(string))	Same as find(), but raises an exception if str not found.
isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
isalpha()	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

isdigit()	Returns true if string contains only digits and false otherwise.
islower()	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
isnumeric()	Returns true if a unicode string contains only numeric characters and false otherwise.
isspace()	Returns true if string contains only whitespace characters and false otherwise.
isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
len(string)	Returns the length of the string
lower()	Converts all uppercase letters in string to lowercase.

Example 11:

```
str1 = "this is string example.!!!";
print("str.capitalize() : "+ str1.capitalize())
sub = "i";
print ("str.count(sub, 4, 40) : ",str( str1.count(sub, 4, 40)))
sub = "is";
print("str.count(sub) : "+str( str1.count(sub)))
print ("Length of the string: "+str(len(str1)))
```

Output:

```
str.capitalize() :  This is string example.!!!
str.count(sub, 4, 40) :  2
str.count(sub) :  2
Length of the string:  26
```

0.5 Python Modules

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code

0.5.1 The *import* Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax:

```
import module1, module2,... moduleN as name
```

Example12: A simple module called hello.py.

```
def print_func1( par ):
  print("Hello : "+ par)
  return
def print_func2( par ):
  print("Good Bye  : "+ par)
  return
```

You can use the python module **hello.py** inside the following code by executing import statement.

```
# Import module hello
import hello
# Now you can call defined function that module as follows
hello.print_func1("Python")
hello.print_func2("Python")
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The from...import Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax –

```
from modname import name1, name2, ... nameN
```

The *from...import* * Statement:

It is also possible to import all names from a module into the current namespace by using the following import statement :

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences:

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

0.5.2 The dir( ) Function

He **dir()** built-in function returns a sorted list of strings containing the names defined by a module. The list contains the names of all the modules, variables and functions that are defined in a module.

Example 13:

```
# Import built-in module math
import math
content = dir(math)
print(content)
```

When the above code is executed, it produces the following result −

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log','log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

0.6 Data Structure

A data structure is a specialized format for organizing and storing data.

0.6.1 Lists

A List represents the most versatile type of data structure in Python. It can contain items of different types and it has no rule against unicity. List indices start from zero, the elements can be sliced, concatenated, and so on.

The list data type has many methods. The following is some of them:

Function	Returns (description)
append(*x*)	Add an item to the end of the list.
extend(*L*)	Extend the list by appending all the items in the given list.
insert(*i*, *x*)	Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).
remove(*x*)	Remove the first item from the list whose value is *x*. It is an error if there is no such item.
pop([*i*])	Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list
clear()	Remove all items from the list.
index(*x*)	Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.
count(*x*)	Return the number of times *x* appears in the list.
sort(*key=None*, *reverse=False*)	Sort the items of the list
reverse()	Reverse the elements of the list in place.
copy()	Return a copy of the list

When to Use Lists

Lists are best used in the following situations:

- When you need a mixed collection of data all in one place.
- When the data needs to be ordered.
- When your data requires the ability to be changed or extended. Remember, lists are mutable.

- When you don't require data to be indexed by a custom value. Lists are numerically indexed and to retrieve an element, you must know its numeric position in the list.
- When you need a stack or a queue. Lists can be easily manipulated by appending/removing elements from the beginning/end of the list.
- When your data doesn't have to be unique. For that, you would use sets.

Example14:

```
a = [66.25, 333, 333, 1, 1234.5]
print(a.count(333), a.count(66.25), a.count('x'))
a.insert(2, -1)
a.append(333)
print(a)
print(a.index(333))
a.remove(333)
print(a)
a.reverse()
print(a)
a.sort()
print(a)
print(a.pop())
print(a)
```

Output:

```
2 1 0
[66.25, 333, -1, 333, 1, 1234.5, 333]
1
[66.25, -1, 333, 1, 1234.5, 333]
[333, 1234.5, 1, 333, -1, 66.25]
[-1, 1, 66.25, 333, 333, 1234.5]
1234.5
[-1, 1, 66.25, 333, 333]
```

Using Lists as Stacks

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.

Example15:

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
print(stack)
print(stack.pop())
print(stack)
print(stack.pop())
print(stack.pop())
print(stack)
```

Output:

```
[3, 4, 5, 6, 7]
7
[3, 4, 5, 6]
6
5
[3, 4]
```

Using Lists as Queues

A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle.

Example16:

```
from collections import deque

queue = deque(["Mohammad", "Modallal"])
queue.append("Ahmad")
queue.append("Ali")

print(queue.popleft())
# The first to arrive now leaves
print(queue.popleft())
# The second to arrive now leaves
print(queue)
# Remaining queue in order of
```

Output:

```
'Mohammad'
'Modallal'
deque(['Ahmad', 'Ali'])
```

0.6.2 Sets

Python also includes a data type for sets. A set is an unordered collection with **no duplicate** elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

When to Use Sets

You should choose to use a ***set*** in the following situations:

- When you need a unique set of data: Sets check the unicity of elements based on hashes.
- When your data constantly changes: Sets, just like lists, are mutable.
- When you need a collection that can be manipulated mathematically: With sets it's easy to do operations like difference, union, intersection, etc.
- When you don't need to store nested lists, sets, or dictionaries in a data structure: Sets don't support unhashable types.

Example17:

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)
# show that duplicates have been removed
print('orange' in basket)
# fast membership testing
print('crabgrass' in basket)
# Demonstrate set operations on unique letters from two words
a = set('abracadabra')
b = set('alacazam')
print(a)
# unique letters in a
print(a-b)
# letters in a but not in b
print(a|b)
# letters in either a or b
print(a&b)
# letters in both a and b
print(a^b)
# letters in a or b but not both
```

Output:

```
{'orange', 'banana', 'pear', 'apple'}
True
False
{'a', 'r', 'b', 'c', 'd'}
{'r', 'd', 'b'}
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
{'a', 'c'}
{'r', 'd', 'b', 'm', 'z', 'l'}
```

0.6.3 Dictionary (Hash table)

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 ([]).

When to Use a Dictionary

- When you need a logical association between a `key:value` pair.
- When you need fast lookup for your data, based on a custom key.
- When your data is being constantly modified. Remember, dictionaries are mutable

Example18:

```
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
```

0.6.4 Tuples

A tuple is represented by a number of values separated by commas. Unlike lists, tuples are immutable and the output is surrounded by parentheses.

When to Use Tuples

- When you need to store data that doesn't have to change.
- When the performance of the application is very important. In this situation you can use tuples whenever you have fixed data collections.
- When you want to store your data in logical immutable pairs, triples etc.

Example19:

```
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
```

0.7 Recursion

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfils the condition of recursion, we call this function a recursive function.

Termination condition:

A recursive function has to terminate to be used in a program. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion. A recursion can lead to an infinite loop, if the base case is not met in the calls.

Example 20: factorial

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
print(factorial(5))
```

todo: rewrite example20 using for loop.

Example 21: Fibonacci numbers

The Fibonacci numbers are defined by:

$F_n = F_{n-1} + F_{n-2}$

with $F_0 = 0$ and $F_1 = 1$

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
print(fib(10))
```

Example 22: f(n) = 3 * n

Mathematically, we can write it like this:

f(1) = 3,

f(n+1) = f(n) + 3

```
def mult3(n):
    if n == 1:
        return 3
    else:
        return mult3(n-1) + 3
print(mult3(5))
```

todo: rewrite example 22 using for loop.

0.8 Python Regular expressions

The module **re** provides full support for Perl-like regular expressions in Python. The re module raises the exception **re.error** if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions.

The match Function

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function –

```
re.match(pattern, string, flags=0)
```

The Search function:

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function:

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters:

Parameter	Description
Pattern	This is the regular expression to be matched.
String	This is the string, which would be searched to match the pattern at the beginning of string.

Flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Matching Versus Searching

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search**checks for a match anywhere in the string (this is what Perl does by default).

Example 23:

```
#!/usr/bin/python
import re
line = "Cats are smarter than dogs";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print "match --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"
```

```
searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
   print("search --> searchObj.group() : ", searchObj.group())
else:
   print("Nothing found!!")
```

Output:

```
No match!!
search --> matchObj.group() :  dogs
```

Search and Replace

One of the most important **re** methods that use regular expressions is **sub**.

Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method returns modified string.

Example 24:

```
#!/usr/bin/python
import re
phone = "9999-000-111 # This is Phone Number"
# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print("Phone Num : ", num)
```

Output:

```
Phone Num :  9999000111
```

Following table lists the regular expression syntax that is available in Python

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
a| b	Matches either a or b.
\w	Matches word characters.
\W	Matches nonword characters.

\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.

0.9 Python Examples

Example 25: Rolling the dice

```
import random
min = 1
max = 6
roll_again = "yes"
while roll_again == "yes" or roll_again == "y":
   print("Rolling the dices...")
   print("The values are....")
   print(random.randint(min, max))
   print(random.randint(min, max))
   roll_again = input("Roll the dices again?")
```

Example 26:

```
# Python Hexadecimal to Decimal Conversion
def getDecDigit(digit):
        digits = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                'A', 'B', 'C', 'D', 'E', 'F']
        for x in range(len(digits)):
                if digit == digits[x]:
                        return x
def hexToDec(hexNum):
        decNum = 0
        power = 0
        for digit in range(len(hexNum), 0, -1):
                decNum = decNum + 16 ** power * getDecDigit(hexNum[digit-1])
                power += 1
```

```
        print(str(decNum))
hexToDec("A5")
```

Example 27: Merge Sort

```
def mergeSort(alist):
    print("Splitting ",alist)
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",alist)

alist = [54,26,93,17,77,31,44,55,20]
mergeSort(alist)
print(alist)
```

0.10 Todo:

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