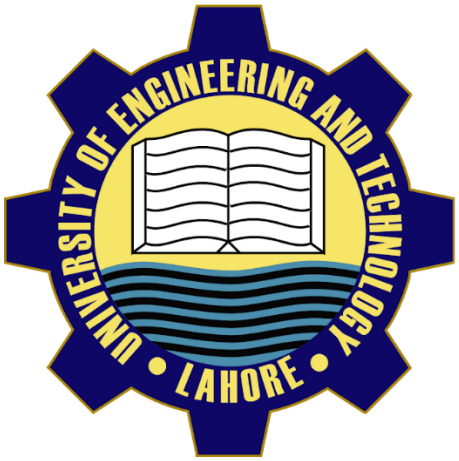
MCT-242 : COMPUTER PROGRAMMING-I

using Python 3.9



Prepared By:

Mr. Muhammad Ahsan Naeem

YouTube Playlist

https://www.youtube.com/playlist?list=PLWF9TXck7O_wMDB-VriREZ6EvwkWLNB7q

LAB 18: LIST AS FUNCTION PARAMETER: CLO-4

Using list as Function Parameter:

A list can be passed as function input parameter. If we consider a function that will calculate the sum of squares of every element of list, it can be written as:

```
def squareSum(aList):
    "Returns the sum of squares of list elements"
    ans=0
    for i in aList:
        ans+=i**2
    return ans
### Main Program starts from here ###
mainList = [3, -5, 8, 7]
s=squareSum(mainList)
print(f"The sum of squares of elements in List is : {s}")
```

Likewise, we can use a list as output parameter of a function. You can put any number of values in a list and simply return that single list.

Here is an example that takes a list as input parameter and returns a list of same size with all elements as square of elements of input list.

```
def squareIt(aList):
    "Returns a new list with each element as square of input list"
    sqList=[]
    for i in range(len(aList)):
        sqList.append(aList[i]**2)
    return sqList

mainList = [3, -5, 8, 7]
print(mainList)
sqMainlist=squareIt(mainList)
print(sqMainlist)
```

Tasks:

[1] Write a program that will take **numeric values** from user until he enters **a blank**. Then it will display all values **less than the average value**, followed by that, it should display **all average values** (if any present in the list), followed by all values **greater than average value**.
You have to do it by creating a function having one input argument which will be the list and three output argument which will also be the lists; first with values greater than average, second with average values (if any) and third with values greater than average.

[2] Write a function with one integer as input parameter and will return a list containing all factors of input integer.

[3] Create a user-defined function named as **isSorted** with one input argument as list. The function should return **True** if the list is sorted and **False** if it is not sorted. In main program, take numeri values from user inside a list until a blank is entered and use the function to display if the list is sorted or not.

[4] Create a function named as **bestOfTwo** having two lists as input argument with assumption that both will have the same number of elements. The function must return one list having the elements as greater of corresponding values at same indices of the input lists. For example, if the two input lists are:

```
[2,4,-3,1.2,6,-10]
[3,3,2,0,100,-20]
```

Then the returned list will be:

```
[3,4,2,1.2,100,-10]
```

[5] There is a built-in function in List class named as **count** which counts the number of a specific element, provides as input argument, inside a list. Your task is to create your own function **countRange** which will count the number of values within a given range. This function should have three input arguments: first the list, and second and third as start and end of the range (both values included). Create a list with different numbers in the main program and use the function to find count of elements in some range.

[6] While writing a list of items in simple text, it is common to separate the values by a comma. Moreover "and" is used before the last item. If there is just one element, we don't need to include "and". For example, below are four examples of displaying one, two, three and four items:

Python
Python and Java
Python, Java and C
Python, Java, C and Ruby

Create a function named as **dispList** , with one list as input argument and it should return a string that contains all of the items in the list formatted as described above. Example use in Main program is shown here:

```
fruits=['Apple','Banana','Orange','Pear','Mango']
print(dispList(fruits))
```

The output will be:

```
Apple, Banana, Orange, Pear and Mango
```

Another example is:

```
fruits=['Apple','Banana']
print(displist(fruits))
```

The output will be:

```
Apple and Banana
```

POST LAB DISCUSSION

The following discussion is not related to Lists but is needed before we move to the next Lab Session.

Scope of Variables:

a) Local and Global Variables

By scope of a variable, it means the part of the code where that variable is accessible. By default, the variables defined inside user defined functions in Python are local i.e., they are accessible within the function block where they are defined.

The variables defined in main program i.e., outside the functions are global by default and are accessible inside the functions as well. Let see the code:

```
def testFunction():
    print(x)

### Main Program Starts Here ###
x=100
testFunction()
```

Variable **x** is not defined inside the **testFunction** but the **x** defined in main program is global by default and is accessible in the function as well; hence we will see **100** displayed on screen when above code is executed.

Now let's see the code below:

```
def testFunction():
    x=50
    print(x)

### Main Program Starts Here ###
x=100
testFunction()
print(x)
```

Here variable **x** is defined inside the function and is local by default. So, with **x=50**, the local **x** is **50** but global **x** of the main program is still **100** and hence this will be the output:

```
50
100
```

Hence, if a variable is defined in main program it is global variable and is accessible in any other function but if it is re-defined in any function with same name, then it will be a local variable in that function. Predict the output of these two versions of code and verify:

```
def testFunction():
    y=x/2
    return y

### Main Program Starts Here ###
x=100
z=testFunction()
print(z)
```

```
def testFunction():
    x=50
    y=x/2
    return y

### Main Program Starts Here ###
x=100
z=testFunction()
print(z)
```

Now let's see the following code:

```
def testFunction():
    y=x/2
    x=x+10
    return y

### Main Program Starts Here ###
x=100
z=testFunction()
print(z)
```

In first line inside function block **y=x/2** will treat **x** as global variable but will **x** be treated as local or global on second line **x=x/10** ?

Actually, we will get an error on first line of the block of the function:

```
UnboundLocalError: local variable 'x' referenced before assignment
```

Because whenever we use assignment of variable inside a function, it is a local variable, and anything written before that, considering it to be a global one, will generate error as a variable cannot be local and global at the same time within one function block.

What if we want to treat a variable as global in a function and still assign it a new value?

For such cases we can use the keyword **global** as shown here:

```
def testFunction():
    global x
    y=x/2
    x=x+10
    return y

### Main Program Starts Here ###
x=100
z=testFunction()
print(z)
print(x)
```

The first line inside the function declares **x** as global and hence if **x** is changed by any means it is reflected in main program as well. The above code has this output:

```
50.0
110
```

There is possibility that one function using the global variable and second using local with the same name. See the code below:

```
def testFunction():
    global x
    y=x/2
    x=x+10
    return y
def testFunction2():
    x=200
    y=x+10
    return y
### Main Program Starts Here ###
x=100
z=testFunction()
z=testFunction2()
print(z)
print(x)
```

Above code will have following output:

```
210
110
```

When to use global variables?

Generally, global variables are discouraged. Occasionally you run into a need for them that cannot otherwise be done.

When using a global variable, you need to be aware of them everywhere in your code. The possible cases where we need to use global variables is the possibility to update variable in multiple functions e.g. score of a player in some game. There can be multiple functions in a game that can change or reset the score so in such case we should declare score variable as a global variable.

Tasks:

[7] Let's make a simple game between two players based on two dice rolls. Here are the rules of the game:

- Starting score of both players is zero.
- After a double-dice rolls (Die1 and Die2) if die1>die2, the difference will be added to player1 score.
- After a double-dice rolls (Die1 and Die2) if die2>die1, the difference will be added to player2 score.
- At any stage if score of both players become equal, both scores will get reset to zero.
- The player reaching to score 20 first will be the winner.

In main program you need to roll the two dice, show the output of the dice and show the player scores. To calculate/update player score you need to create a function named as **`scoreUpdate()`**. This function will not have any input and output argument. It will take the main program variables and update the scores.

b) Enclosing Variable Scope:

Other than local and global scope of the variables, there can be **Enclosing** Scope of a variable. This is applicable in case of nested functions. Nested function means a function having the inner function. Let's see such case:

```
def testFunction():
    x=20
    print(f'x inside test function: {x}')
    def test2():
        x=30
        print(f'x inside test2 function: {x}')
    test2()
    print(f'x inside test function: {x}')

## Main Program ##
x=10
testFunction()
print(f'x inside Main: {x}')
```

The function **`test2`** is the inner function of the function **`testFunction`**. Firstly, why do we need to have an inner function? There can be a couple of reasons; the simplest is that the outer function might need some calculations multiple times and then it can have the inner function. But that can be achieved

by creating a normal function instead of the inner function as well. Basically, when we have an inner function, that can be used only inside the outer function. For example, if you will try to use the inner function **test2** in Main program, it will generate an error. The inner function is also known as **Enclosed** function and outer function is also known as **Enclosing** function.

The variable scope named as Enclosing is meant for the inner function. In above code, all three functions i.e., outer, inner and Main are having their own defined variable **x**. So, **x** is local in all three locations and will have its own value for each of those. Here is the output of above code:

```
x inside test function: 20
x inside test2 function: 30
x inside test function: 20
x inside Main: 10
```

Now if we make the **x** of **testFunction** as global, it will share the scope of the Main program, but the **x** inside inner function is still local. Here is the code:

```
def testFunction():
    global x
    x=20
    print(f'x inside test function: {x}')
    def test2():
        x=30
        print(f'x inside test2 function: {x}')
    test2()
    print(f'x inside test function: {x}')

## Main Program ##
x=10
testFunction()
print(f'x inside Main: {x}')
```

And the output will be:

```
x inside test function: 20
x inside test2 function: 30
x inside test function: 20
x inside Main: 20
```

If we also change the **x** of the inner function to global, then all **x** of the above program are same. So far, scope of inner function is just like any other function as local or global. But there can be a different scenario.

What if we want the outer function **testFunction** has different scope than of the Main program? We can simply remove **global x** from that function. And now we want the inner function **test2** share the same scope as of the outer function? Let's try that by making the **x** of inner function as global and keep the outer function **x** as local as shown here:

```
def testFunction():
    x=20
    print(f'x inside test function: {x}')
    def test2():
        global x
        x=30
        print(f'x inside test2 function: {x}')
    test2()
    print(f'x inside test function: {x}')

## Main Program ##
x=10
testFunction()
print(f'x inside Main: {x}')
```

Do you think **x** of the inner function is sharing the scope of outer function? It's very easy to see that it is not sharing the scope of the outer function, but it is sharing the scope of the Main program and this will be the output:

```
x inside test function: 20
x inside test2 function: 30
x inside test function: 20
x inside Main: 30
```

So, our problem is still there. We want the **x** of inner function share the scope of the outer function. For that, we can use the keyword **nonlocal**, indicating that it is not local and it is sharing the scope of outer function.

This should be the code:

```
def testFunction():
    x=20
    print(f'x inside test function: {x}')
    def test2():
        nonlocal x
        x=30
        print(f'x inside test2 function: {x}')
    test2()
    print(f'x inside test function: {x}')
```

```
## Main Program ##
x=10
testFunction()
print(f'x inside Main: {x}')
```

And here is the output:

```
x inside test function: 20
x inside test2 function: 30
x inside test function: 30
x inside Main: 10
```

You can see **x** inside the **testFunction** was **20** initially but is changed to **30** when **x** of inner **test** function was changed to **30**, because they share the scope. And we call this as Enclosing Scope of the inner function.

The LEGB Rule:

The variable scope follows the LEGB rule where:

L → Local
E → Enclosing
G → Global
B → Built-in

It means that when a variable is used in any function, the interpreter will first find that in the Local Domain of that location of code. If found, that value will be used for that variable. If not found, the interpreter will look into the Enclosed Domain and if not found it will move to the Global Domain. If it not there in the Global Domain, it will move to the Built-in Domain. The Built-ins are preset Python variables and you can find all of those by running this code:

```
import builtins
print(dir(builtins))
```

Creating your own Module:

We have used a few modules like **math**, **random**, **msvcrt** etc. A module contains the definition of functions and possible some other variables and we can use those in our program. We can also create our own Module and that is not difficult at all. For example, we have been using the calculations for a **Prime Number** in many tasks, so it will be better to have a module and define a function **isPrime** inside that. Then we can simply import that module where we need the logic of a Prime Number.

Creating our own Module is as simple as creating any Python file with extension **py**. Lets name the file as **my_module.py** having this code:

```
from math import sqrt
def isPrime(n):
    for i in range(2,int(sqrt(n))+1):
        if(n%i==0):
            return False
    return True
```

Now if we have to use the function **isPrime** in any other program e.g. in file **lab18.py** , we can import the function in that file and use that. For example:

```
from my_module import isPrime
num=int(input('Enter a number:'))
if(isPrime(num)):
    print(f'{num} is a Prime Number')
else:
    print(f'{num} is NOT a Prime Number')
```

Of course, there can be as many functions as needed in the module.

The important thing is the location where to place the file **myModule.py**. Python first look for the module inside the same folder where we have the other file, importing that module. If you want to see all locations where Python searches for the Module file in order, you can find that by importing the **sys** module and inside the module is a list named as **path** having all those paths. To better read all paths, you can run this code:

```
import sys
allPaths=sys.path
for p in allPaths:
    print(p)
```

In my case, this is the output:

```
e:\UET\Mechatronics\BScClasses\CP1\01_Python\Session2019\CP-I\Codes
C:\Users\Ahsan\AppData\Local\Programs\Python\Python39\python39.zip
C:\Users\Ahsan\AppData\Local\Programs\Python\Python39\DLLs
C:\Users\Ahsan\AppData\Local\Programs\Python\Python39\lib
C:\Users\Ahsan\AppData\Local\Programs\Python\Python39
C:\Users\Ahsan\AppData\Roaming\Python\Python39\site-packages
C:\Users\Ahsan\AppData\Local\Programs\Python\Python39\lib\site-
packages
```

Finally, if you want to keep the file at some other location and want that to be included in above paths, you can update the System Variables of your system. In case of **Windows**, search for '**Advanced System Settings**' on windows search bar and a new window with system settings will be opened. Click on **'Environment Variables'**. (Or directly search for **Edit Environment Variables**). In new opened window under **'User variables for username'** window, click **'New'**. On variable name write **'PYTHONPATH'** and on variable value write the whole path e.g. **C:\Users\Ahsan\Desktop**. Now you should restart the IDE and the code can pick module from there.