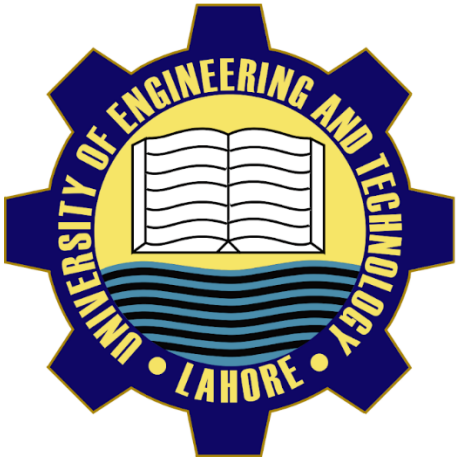
MCT-243 : COMPUTER PROGRAMMING-II

using Python 3.9



Prepared By:

Mr. Muhammad Ahsan Naeem

YouTube Playlist

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

Lab 27: MORE ON USER-DEFINED FUNCTIONS

Setting input parameters/arguments by name:

Let's see a very simple function that will print a multiplication table. The function has two input arguments; **`table`** and **`limit`** where **`table`** specifies the value of which the multiplication table is needed, and the **`limit`** specifies the limit till the table should be displayed. It is given here along with the main program:

```
def showTable(table,limit):
    'It will display the table'
    for i in range(1,limit+1):
        print(f'{table}x{i}\t=\t{table*i}')
### Main Program Starts Below ###
showTable(8,10)
```

In above program while calling the function in main program, the two input parameter values are specified such that first value corresponds to first parameter and second to the second one.
We can also specify the input arguments through their names as:

```
showTable(table=8,limit=10)
```

For this case, we need to know the variable names used in function definition, but the good thing is that now the order of parameters while calling the function does not matter. So, the following will also have the same output:

```
showTable(limit=10,table=8)
```

When we specify an argument by its name, that is also known as **Keyword Argument** and when we pass that without the name, then that is passed by the position and is known as **Positional Argument**.

Optional Input Parameters

A function can have some optional input parameter(s). By optional it means that we may or may not provide those parameters while calling that function. The optional parameter(s) have default values. If we specify the value of optional parameter, that value will be assigned to it otherwise the default value will be used.

In above example, we can make the **`limit`** parameter optional and assign its default value as **`10`**. This will be done as:

```
def showTable(table,limit=10):
    'It will display the table'
    for i in range(1,limit+1):
        print(f'{table}x{i}\t=\t{table*i}')
### Main Program Starts Below ###
showTable(8)
showTable(9,12)
```

In last two lines of above program, the function is used in two ways. When it is called without specifying the second parameter i.e. **limit**, its value is taken as 10 and table of 8 is displayed for 10 entries. In second case, when second parameter is specified as 12, the parameter **limit** is assigned 12 and table of 9 is displayed for 12 entries.

Now let's add another parameter for the order of the table to display table in ascending or descending (reverse) order. Generally, table is displayed in ascending order so we will specify the argument name as **reverse** and we should make it optional with default value as **False**. It is shown here:

```
def showTable(table,limit=10,reverse=False):
    if not reverse:
        start,stop,step=1,limit+1,1
    if reverse:
        start,stop,step=limit,0,-1
    for i in range(start,stop,step):
        print(f'{table}x{i}\t=\t{table*i}')
### Main Program Starts Below ###
showTable(8)
showTable(4,8)
showTable(9,12,True)
showTable(5,reverse=True)
showTable(5,reverse=True)
showTable(reverse=True,table=10,limit=6)
```

See carefully different ways the function is called in last six lines.

Positional-Only and Keyword-Only Arguments:

Let's print the help on **sqrt** function of the **math** module as:

```
from math import sqrt
help(sqrt)
```

This will be the output:

```
sqrt(x, /)
    Return the square root of x.
```

You can see a forward slash **/** displayed as one input argument of the function **sqrt**. Before we see the meaning of the forward slash, see that the input argument name is **x** and let's try to apply the function by specifying the name of the argument as:

```
from math import sqrt
print(sqrt(x=4))
```

And you will see this error:

```
TypeError: math.sqrt() takes no keyword arguments
```

We know we can pass an argument by its position, known as Positional Argument or by its name, known as Keyword Argument. But when we specify a forward slash, apparently as an input argument as shown

on the output of help on `sqrt` function, it means that all arguments before the forward slash are **Positional-Only** arguments. It means we cannot pass those by their name. And hence we can use the `sqrt` function as:

```
from math import sqrt
print(sqrt(4))
```

Let's add this feature in our function.

```
def showTable(table,/,limit):
    'It will display the table'
    for i in range(1,limit+1):
        print(f'{table}x{i}\t=\t{table*i}')

## Main Program ##
showTable(8,10)
showTable(8,limit=10)
showTable(table=8,limit=10)
```

Check the output of the three example uses of the function and you will see the output for first two and there will be error on the third because we are passing the argument `table` by name and that is positional-only.

Now let's see the help on function `sorted` as:

```
help(sorted)
```

This will be the first line indicating the signature of the function:

```
sorted(iterable, /, *, key=None, reverse=False)
```

Now in this case, other than `/` we also see `*` as one input argument. This `*` specifies that the arguments specified after it are **Keyword-Only** arguments. It means we have to pass those by their names.

In the table example, let's update the function as:

```
def showTable(table,/,limit=10,*,reverse=False):
    if not reverse:
        start,stop,step=1,limit+1,1
    if reverse:
        start,stop,step=limit,0,-1
    for i in range(start,stop,step):
        print(f'{table}x{i}\t=\t{table*i}')
```

It means the argument `table` is Positional-Only, argument `limit` can be positional or keyword as desired and the argument `reverse` is Keyword-Only. The arguments `limit` and `reverse` also have the default value which is an independent concept from the Positional-Only and Keyword-Only concept.

These are correct uses of the above function:

```
showTable(8)
showTable(8,12)
showTable(8,limit=12)
showTable(8,reverse=True)
showTable(8,reverse=True,limit=12)
showTable(8,12,reverse=True)
```

Tasks:

[1] A complex number can be represented in two formats: cartesian and polar. In cartesian form, it has two components known as real part and imaginary part and is represented as (x, y) or $(x + iy)$. In polar form, it again has two components known as magnitude and angle and represented as $r\angle\theta$. The angle can be is Degrees or Radians. A complex number in polar form can be converted into cartesian form using:

$x = r\cos(\theta)$

$y = r\sin(\theta)$

The task is to write a user defined function named as `pol2cart()` that will convert a polar form of complex number into a cartesian form using above relation. The units of angles must be handled correctly. Let's set the defaults units of angle to Degree but user should be able to use it if angle is in Radian.

The main program is given here:

```
x,y=pol2cart(2,45)
print(x,y)
x,y=pol2cart(2,math.pi/4,"Radian")
print(x,y)
```

[2] Just like previous task now create another user defined function, `cart2pol()` that will convert cartesian form of complex number to polar form. The default units of angle of polar form will be Degrees but can be the Radian if user specifies that.

*args (variable length argument) in Function:

Let's have a function with three input arguments and it returns the sum of squares of those:

```
def sumSqr(x,y,z):
    return x*x+y*y+z*z

print(sumSqr(4,3,5)) #It will print 50
```

Now suppose we have a list of three numbers, and we want to apply above function on those three values. One solution can be as:

```
def sumSqr(x,y,z):
    return x*x+y*y+z*z
```

```
numbers=[4,3,5]
print(sumSqr(numbers[0],numbers[1],numbers[2])) #It will print 50
```

But a better way is shown here:

```
print(sumSqr(*numbers)) #It will print 50
```

***numbers** basically un-packs the elements inside the list which are three in total and they are taken as the three input arguments of the function. If we have two values inside the list, it will get unpacked to two values. And if we are using it for the above function, it will generate an error as the function needs three arguments while we are providing only two as shown here:

```
numbers=[4,3]
print(sumSqr(*numbers)) #It will generate error
```

Following error is generated:

```
TypeError: sumSqr() missing 1 required positional argument: 'z'
```

See another possibility here:

```
numbers=[4,3]
print(sumSqr(*numbers,5)) #It will print 50
```

Now let's see a different example and create a function that will take one list as input parameter and will return sum of square of all values inside the list no matter how many those are. The function and one example use are shown here:

```
def sumSqr(aList):
    ans=0
    for i in aList:
        ans+=i*i
    return ans

numbers=[4,3,2,1]
print(sumSqr(numbers)) #It will print 30
```

To use above function, we will have to pass a list as input argument always. But if we change the above function as below:

```
def sumSqr(*a):
    ans=0
    for i in a:
        ans+=i*i
    return ans
```

Now **sumSqr(*a)** at the function definition has a different meaning. The input argument **a** is no more a list but we can pass simple values and ***a** will pack them to a Tuple and inside the function **a** is treated as a Tuple. Following examples will illustrate the idea:

```
def sumSqr(*a):
    ans=0
```

```
    for i in a:
        ans+=i*i
    return ans

print(sumSqr(4,3,2,1)) #It will print 30
print(sumSqr(10,20)) #It will print 500
print(sumSqr(1,2,3,4,5,6,7,8,9,10)) #It will print 385
```

See the last three statements carefully and observe that we can use the function for any number of input arguments. **`*a`** in definition of the function, packs all those values into a Tuple and inside the function body those values are present inside Tuple **`a`**.

Now we cannot pass a list as input argument for above function. But luckily, we have already seen a way to unpack values from a list or tuple which is same as of packing values to a list or tuple i.e. using **`*`**. So, if we want to use above function for a list, we can do it as:

```
def sumSqr(*a):
    ans=0
    for i in a:
        ans+=i*i
    return ans

numbers=[4,3,2,1]
print(sumSqr(*numbers)) #It will print 30
```

In **`sumSqr(*numbers)`**, **`*numbers`** unpacks all elements into individual elements; four values in above case. Interpreter when moves to the function definition, **`*a`** will pack all those values into a Tuple.

Order of arguments:

Let's suppose we want to implement a function for following formula:

$$F = \frac{{x_1}^2 + {x_2}^2 + {x_3}^2 + \cdots {x_n}^2}{a} + b$$

On right side, we have **`a`**, **`b`** as single values and **`n`** terms for **`x`** where **`n`** is not fixed. In order to create a user defined function for above formula, we should have two single valued input arguments (a and b) and one variable length input argument (for x's). The code is shown here:

```
def F(a,b,*x):
    ans=0
    for i in x:
        ans+=i*i
    ans=(ans/a)+b
    return ans

print(F(10,2,1,2,3,4)) #Will print 5.0
```

In the last line where the function is used, first value is assigned to **`a`**, second to **`b`** and all other remaining values are packed and assigned to **`x`** as a Tuple. For this kind of situation where single values and variable length values are required to pass as input arguments, single values must come first and at the end should

be the variable length argument. Moreover, we cannot have two variable-length input arguments as interpreter cannot decide how many of provided values be assigned to each of those.

**kwarg(keyword arguments) in Functions:

`**kwarg` is Key-Word arguments and it is a method to provide input argument as Keywords. The term is same as of passing the argument by name, but this is a different case. If we define a function as:

```
def Test(**x):
    print (x)
```

Now we can use this function by passing any number of keyword arguments. An example is shown here:

```
Test(A=1,B=2,C=3)
```

It will print:

```
{'A': 1, 'B': 2, 'C': 3}
```

The above output resembles with the dictionary output and actually if you print datatype of **`x`** within function body, you will see **`x`** is of dictionary type. So as **`*args`** packs arguments to a list **`**kwarg`** packs the provide Keywords to a dictionary with key-value pairs.

The above example is having arbitrary values. Here is one example with some meaningful key-value pairs:

```
Test(firstName='Ali',lastName='Ahmad',age=23,gender='male')
```

It will print:

```
{'firstName': 'Ali', 'lastName': 'Ahmad', 'age': 23, 'gender':
'male'}
```

Within the function body we can access keys and values individually as well. Let's see following function along with an example use:

```
def dispPerson(**person):
    for k,v in person.items():
        print(f'{k}\t= {v}')

dispPerson(firstName='Ali',lastName='Ahmad',age=23,gender='male')
```

We can directly pass in a dictionary as input argument while using such function but by using **`**`** to unpack dictionary into keyword arguments. An example shown here:

```
def dispPerson(**person):
    for k,v in person.items():
        print(f'{k}\t= {v}')

p1={'firstName':'Ali','lastName':'Ahmad','age':23,'gender':'male'}
dispPerson(**p1)
```

Order of Arguments:

If a function needs simple (single valued), variable-length arguments and keyword arguments, then there must be following order of these arguments:

1. **Simple (single-valued)** : One or More
2. **Variable length (*args)** : Only One

3. **Keyword (**kwarg)** : Only One

A general form is shown here:

```
def example(arg1, arg2, *args, **kwargs):
```

Pass-by-Value vs Pass-by-Reference:

There are two ways input arguments are passed to the function; **passed by value** and **passed by reference**. The **immutable** data types like **int**, **float**, **string** are always passed by value. It means that when such variable is passed to a function, their value is passed to the input arguments and not the variables itself. Any change made on that variable in function will not affect the actual variable value. An example is shown here:

```
def test(x):
    x+=5
    print(x)
## Main Program Starts here ###
x=10
test(x)
print(x)
```

It will print:

```
15
10
```

In main program, `x` is assigned `10` and when it is passed to the function, only the value is passed and `x` in main program will remain as `10`. This is known as passing by value.

Another way to verify this idea is to see the memory address of the **identifier** `x`. We can use `id()` function for that. Hence, instead of printing the values, the addresses are printed in following code:

```
def test(x):
    x+=5
    print(id(x))
## Main Program Starts here ###
x=10
test(x)
print(id(x))
```

The output is:

```
140709597508288
140709597508128
```

The two addresses are different which shows that the two `x` are different objects in memory. Changing one will not change the other.

The other way is known as passing the argument by **Reference**. In this way a reference of an object is passed which refers to the original object. Any change made to that object in the function will be reflected to the original object. The **mutable** data types like list, dictionary etc. are passed by reference by default.

See this example:

```
def test(a):
    a.append(10)
    print(id(a))
## Main Program Starts here ###
x=[1,5,7]
test(x)
print(id(x))
```

The output is:

```
2187532325448
2187532325448
```

You can see that although different variable names are used but the output of `id()` function is same for `x` and `a` which means that both `a` and `x` are pointing to same object. Now if you print `x` after function call you will see `10` appended to it. Verify that!

Basically, it is a very good idea to have mutable data types passed by reference because if they are required to be updated via some user-defined function, we can do that very easily.

The last point to be careful is that if we use assignment to a passed variable (mutable with default pass by reference) within user-defined function, that will create a new copy just like in case of passing argument by value. It is explained here:

```
def test(a):
    a=[2,2,2]
    print(id(a))
## Main Program Starts here ###
x=[1,5,7]
test(x)
print(id(x))
```

The output is shown here:

```
2352774534216
2352774533704
```

Now you can see different addresses because of the assignment in first line within user-defined function.

Tasks:

[3] With the same student dictionary structure as one of the above examples, write a user-defined function named as `addCourses` that will take one student and any number of subjects as input arguments and will add those subjects into the courses list of the student. Be careful that there should not be repetition of courses in the list. Use **Set** properties to achieve this use the pass by reference technique that will update the original student. Test the function on some student and verify by printing the student after function call.

[4] Greatest Common Divisor (GCD): Greatest Common Divisor (GCD) or the Highest Common Factor (HCF) of two numbers can be found with following user defined function:

```
def GCD(x,y):
    m=min(x,y)
    for i in range(1,m+1):
        if(x%i==0 and y%i==0):
            ans=i
    return ans
```

Modify the above function such that it finds the GCD of any number of input arguments.

[5] Perimeter of Polygon: Polygon is a planar closed shape with straight line segments. A polygon can have minimum 3 sides and there is no limit for the maximum sides. If a polygon has **n** sides, it will have **n** vertices. A vertex on XY plane can be represented as a Tuple with two values (x and y).

As an example, a polygon can have five vertices as:

```
v1,v2,v3,v4,v5=(2,4),(1,3),(5,-2),(3,1),(6,5)
```

Your task is to create a user-defined function **`polyParameter`** that could take any number of vertices as the input argument and will return the **Perimeter of the Polygon**. For above case, the function will be used like this: **`polyParameter(v1,v2,v3,v4,v5)`** . To find the parameter you need to find the sum of length of all sides of the polygon. It will be better to create a function **`dist(v1,v2)`** which will take two vertices as input argument and returns the length of the line segment.