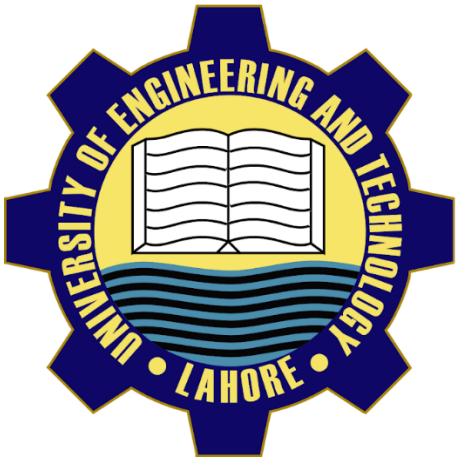
MCT-242 : COMPUTER PROGRAMMING-I

using Python 3.9



Prepared By:

Mr. Muhammad Ahsan Naeem

YouTube Playlist

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

©Department of Mechatronics and Control Engineering, University of Engineering and Technology, Lahore, Pakistan

LAB 12: ITERATION: WHILE LOOP: CLO 4

Statements can be iterated using **while** loop. Considering the same first example from where we started **for** loop, here is a simple code that takes a number from user and prints the square of it:

```
num=eval(input('Enter a number: '))
print(f'The square of {num} is {num**2}')
```

We want to execute this code ten times but using **while** loop. It can be done as:

```
[1] i=1
[2] while(i<=10):
[3]     num=eval(input('Enter a number: '))
[4]     print(f'The square of {num} is {num**2}')
[5]     i=i+1
[6] print('Thanks')
```

How this works:

- Line **2** is the syntax of the **while** loop and lines **3-5** are the block of the loop.
- In line **1** a variable **i** is initialized known as loop variable.
- In line **2**, **while** is the key word for the **while** loop followed by a conditional statement.
- From line **3** starts the block of the **while** loop, where we write the statement we want to iterate.
- The loop variable is updated inside the loop block as done in line **5**.

Here is the flow execution of the above code:

- Interpreter execute line **1** and assigns **1** to variable **i**.
- Interpreter moves to line **2** and evaluates the condition of **while** statement that is **True** as **i** holds **1** and **1<10**
- Because the condition is **True**, interpreter enters into the block and executes line **3-5**.
- After executing the last statement of the **while** block; line **5** in above program, the interpreter moves to line **2** again.
- Now **i** contains **2**. The condition is evaluated again, and it is true as **2<10**. Interpreter enters into the block again and executes lines **3-5**.
- These iterations keep repeating till **i=10** which is the last iteration. In **i=10** iteration, the line **5** will make **i=11**, the interpreter will move to line **2** but this time the condition is **False** so instead of entering into the block, the interpreter will move out of **while** block i.e. to line **6** and onwards.

A **while** loop generally has three major factors:

i. Initialization of loop variable before start of **while** loop. This can be direct initialization as in above example or can be from some other calculations/means.
ii. The condition of **while** statement.

iii. Update of loop variable within the block of **while** loop.

iv. You can see that **while** loop syntax is similar to **if** statement but instead of moving forward after executing the **if** block, interpreter loops back to **while** statement until the condition is **False**.

Tasks:

[1] Write a program that will ask user to enter a number and then will print 1 to entered number with their cubes separated by a tab. Do this using **while** loop.

Sample Output is:

```
Enter a number: 4
1    1
2    8
3    27
4    64
```

[2] We did a task of writing the **fact()** function using **for** loop. Repeat the task using **while** loop. The **for** loop version is given here:

```
def fact(a):
    ans=1
    if(a>=0):
        for i in range(1,a+1):
            ans*=i
    return ans

### Main Programs Starts from here ###
x=eval(input('Enter a number: '))
y=fact(x)

if (y==None):
        print('Factorial of negative numbers does not exist!')
else:
        print(f'{x}! = {fact(x)}')
```

while loop vs for loop:

So far, we have seen the use of **while** loop in place of **for** loop. Is that everything with **while** loop?

Any logic implemented in for loop can be written using while loop and vice versa.

The above claim is logically true but practically there are certain situations where we prefer to use one over other.

Let's consider the starting example once again to take an input from user ten times and display its square. What if we want the user to enter as many numbers as he wants instead of fixing it to ten. One possible way along with a sample output is shown here:

```
n=eval(input('How many numbers you want to enter: '))
for i in range(n):
        num=eval(input('Enter a number: '))
        print(f'The square of {num} is {num**2}.')
print('Thanks')
```

```
How many numbers you want to enter: 3
Enter a number: 4
The square of 4 is 16
Enter a number: 5
The square of 5 is 25
Enter a number: -8
The square of -8 is 64
Thanks
```

A much better way will be to do this using a **while** loop as shown here:

```
num=1
while(num!=0):
    num=eval(input('Enter a number (0 to exit): '))
    print(f'The square of {num} is {num**2}.')
```

In above code **num** is the loop variable. A number is asked from user to display its square-root. The condition is checking whether user entered **0** or non-zero value. A **0** entered by user is considered as he wants to end the process. Note that the loop variable **num** is not explicitly updated inside the loop block but the input entered by user is stored in that variable, which actually updates it in every iteration. Also note that the very first line is there to initialize loop variable to a value such that the condition is **True** and interpreter enter into the loop block. For this case it could be any value but not **0**.

Another thing you might notice in the output is the last message printed as shown here:

```
Enter a number (0 to exit): 3
The square of 3 is 9
Enter a number (0 to exit): 2
The square of 2 is 4
Enter a number (0 to exit): -4
The square of -4 is 16
Enter a number (0 to exit): 5
The square of 5 is 25
Enter a number (0 to exit): 0
The square of 0 is 0
```

The last **0** entered by the user was to end the process but of course the loop bock will display that as well. If we want to eliminate that last line, we can do as:

```
num=1
```

```
while(num!=0):
    num=eval(input('Enter a number (0 to exit): '))
    if(num!=0):
        print(f'The square of {num} is {num**2}.')
#End of while loop
print('Thanks')
```

This type of repetition is also very useful to re-execute a program if user wants to. For example, we made a program to find the roots of a quadratic equation. The user may need to find roots for more equations. In such cases we can follow three basic steps:

i. Write the complete program inside a **while** loop
ii. Ask user if he wants to re-execute at the end of program
iii. Control the **while** loop based on user's entry.

The complete example of quadratic equation is given here:

```
import math
print('This Program will solve Quadratic Equation for you')
repeat='y'
while(repeat=='y' or repeat=='Y'):
        a,b,c = eval(input("Enter the values of a,b,c:"))
        disc=b**2-4*a*c
        if(disc>=0):
                d=math.sqrt(disc)
                x1=(-b+d)/(2*a)
                x2=(-b-d)/(2*a)
        else:
                disc=-disc
                d=math.sqrt(disc)
                x1=complex(-b/(2*a),d/(2*a))
                x2=complex(-b/(2*a),-d/(2*a))
        print(f'x1={x1}')
        print(f'x2={x2}')
        print()
        repeat=input('Press Y to find another solution: ')
```

General rule to select for loop vs while loop:

Based on different examples we have seen, it can be generalized that **for** loop is preferred where we know the number of iterations before hand and **while** loop is preferred when we don't have such information.

We can perform counting, summation, product etc. with **while** loop very much same as in **for** loop.

Tasks:

[3] Write a program that will keep asking a number from user until he enters `0`. And it will display total numbers entered by user (except the last `0`) and average of those.
All you need is to use a `while` loop with condition that loop variable is not zero. Maintain two variables; one to count the entries and other to hold the sum. Coming out of while loop, use the count and sum variables to compute and display average value.

[4] Write a program that will take a positive integer from the user and will compute following sequence:
If the number is even, halve it; if the number is odd, multiply it by 3 and add 1. Repeat this sequence until the value is 1. Print all the values and also print number of steps performed.

Sample output is: ***(Only the first value i.e. 9 is entered by the user, rest are generated by the program)***

```
Enter Initial Value: 9
Next value is: 28
Next value is: 14
Next value is: 7
Next value is: 22
Next value is: 11
Next value is: 34
Next value is: 17
Next value is: 52
Next value is: 26
Next value is: 13
Next value is: 40
Next value is: 20
Next value is: 10
Next value is: 5
Next value is: 16
Next value is: 8
Next value is: 4
Next value is: 2
Next value is: 1

No. of steps: 19
```

Validating user input and keep asking for valid entry:

In many tasks we did previously, we assumed that user will enter a valid input but this might not be true in practical cases. Validating the input is simply applying the appropriate condition and a `while` loop can be used for re-taking the input until user enters a valid input. In case of quadratic equation example, the variable `a` cannot be `0` because with `a` being zero, the equation is not quadratic and division by `a`

will generate a run-time error. We can apply a check and re-ask user to enter non-zero value as shown here:

```
import math
print('This Program will solve Quadratic Equation for you')
repeat='y'
while(repeat=='y' or repeat=='Y'):
        a = eval(input("Enter value of a:"))
        while(a==0):
                print('a cannot be zero.')
                a = eval(input("Enter value of a:"))
        b = eval(input("Enter value of b:"))
        c = eval(input("Enter value of c:"))
        disc=b**2-4*a*c
        if(disc>=0):
                d=math.sqrt(disc)
                x1=(-b+d)/(2*a)
                x2=(-b-d)/(2*a)
        else:
                disc=-disc
                d=math.sqrt(disc)
                x1=complex(-b/(2*a),d/(2*a))
                x2=complex(-b/(2*a),-d/(2*a))
        print(f'x1={x1}')
        print(f'x2={x2}')
        print()
        repeat=input('Press Y to find another solution: ')
```

Tasks:

[5] Repeat task 4 such that user is not allowed to enter less than 2.

[6] We have done following task in one of previous lab sessions:

Write a program that will ask user to enter a 4-digit number. Your program will show the number of zeros in that 4-digit number. (There can be at maximum 3 zeros in 4-digit number!)

Sample output is:

```
Enter a 4-digit number: 5402
No. of zeros in 5402= 1
```

Another Sample output is:

```
Enter a 4-digit number: 2001
No. of zeros in 2001= 2
```

For this task you have to write program that will count and display number of zeros in any number; not just 4-digit one.

How to do that:

Recall that in previous task we used remainder division and floor division to get individual digit and updating count variable. Individual digits were separated as:

```
a=x%10
b=x//10
c=b%10
# And so on
```

The first line picks the unit's digit from the number and stores in variable **a**. Then the second line removes unit digit and stores the remaining 3 digits in variable **b**. We need to repeat it three times in case of 4-digit numbers (the first digit cannot be 0).

This is better explained with an example number as x=5429

x	a=x%10	x=x//10
5429	9	542
542	2	54
54	4	5
5	5	0

When the number of digits in **x** are not fixed to **4**, how many times we have to repeat above process i.e. picking the units digit and separating the remaining number?

We have to repeat this until the remaining number after removing unit's digit becomes 0; the last entry of last column in above table. This can be done as:

```
while(x!=0):
        a=x%10
        x//=10
```

Apply condition on **a** and update **count** variable.

Note: The input is taken in variable **x** and same is used in above code repeatedly to store remaining number after removing the unit digit. At the end of the loop, **x** will become **0** and actually we lost the original number. Therefore, to display the message as shown in sample output that includes prints the original entered number, you must make another copy of input number before applying above loop and use that copy for last message.

[7] Next Prime Number

Create a User Defined Function which will have one input argument (a positive integer) and it will return the first Prime Number which comes after the input number.

***Note**: The input number is NOT supposed to be a prime number.*

- If input is 25, the output should be 29.
- If input is 28, the output should be 29.

Pseudocode:

```
function isPrime
    input→ n
    output→ True if n is Prime and False if n is Composite

Take user input→ num
x=num+1
while (x is not a Prime Number)
    x→x+1
Display x
```

[8] First n Prime Numbers

Write a program that will take one positive integer from the user and will print that many Prime Numbers starting from 2.

For, example if user enter 6, it means the program should print first 6 prime numbers which are: 2,3,5,7,11,13

Pseudocode:

```
function isPrime
    input→ n
    output→ True if n is Prime and False if n is Composite

Take user input→ num
count=1
x=2
while (count is less than or equal to num)
        if (x is Prime)→Display x; count→count+1
        x=x+1
```