Objective: The worksheet provides a quick overview for few control statements like *if/else* block, *for* and *while* loops.

From this worksheet onward, the complexity of the code would gradually grow. Participants are expected to emulate all the examples given in the worksheets and do all the exercises.

Comparison or Conditional Operators

- Commonly used comparison operators in Python are following:

==	***if equal***
!=	***if not equal***
<	***if less than***
<=	***if less than or equal***
>	***if greater***
>=	***if greater than or equal***

if/else Statements

- The general usage of an *if* statement is

  if expression comparison_operator value:
  statement(s) which need to be executed when above if clause evaluates to True
- Unlike C/C++, no parentheses or curly braces are not required. *If* statement must end with a colon (:)
- Three independent code blocks are shown below which use *if* statements. They look somewhat identical. Before your review the answers, attempt yourself the following questions:
  - What will be printed from these blocks?
  - Why after *if* statement, there are spaces for the next statements in few cases?
  - If these spaces are not there, what will happen?

```
shirtSize = 32
if shirtSize == 32:
    print("Shirst are available!");
    print ("How many you need?")
```

```
shirtSize = 32
if shirtSize != 32:
    print("Shirst are available!");
    print ("How many you need?")
```

```
shirtSize = 32
if shirtSize != 32:
    print("Shirst are available!");
print ("How many you need?")
```

- The above code blocks will print the following:

```
Shirst are available!
How many you need?
```

```

```

```
How many you need?
```

- The blank spaces that are present before the statements after *if* statement are called ***indentation***. Continuously indented statements after *if* statement form a ***block***. It will be executed if the condition of the *if* statement evaluates to True. Unlike C/C++, usage of curly braces are not there in Python to define a block.
- Jupyter note book automatically indents the next statements after an *if* statement. If one does not do so for the very next statement, there would be an error. However, after one block statement, user can again continue from the extreme left, as shown in the 3rd code block example above.

- It is a good practice to follow a 4-spaces indent and not to use TAB in place of it.
- Proper indentation makes the code readable and maintenance of the code easier.
- Along with *if*, *else* statement is also normally used. The following example conveys the idea:

```
shirtSize = 44
if shirtSize < 34:
    print("You are a fit person...Shirst are available.");
else:
    print ("Sorry. We do no have oversized shirts.")
```

```
Sorry. We do no have oversized shirts.
```

Exercise:

1. Write a program when runs, prompts the user to enter the shirt size and then using *if/else* print the appropriate message. Remember to convert string to an integer.
2. What will be printed from the following Python code?

```
shirtSize = 34
if shirtSize == 34:
    print("You are a fit person...Shirst are available.");
else:
    print ("Sorry. We do no have oversized shirts.")
    print ("How many you need?")
print ("Try elsewhere.")
```

for Loop

- Before we review *for* loop, it is worth to review the function ***range()*** that is available in Python. In Jupyter Notebook cell, if one simply types ***range?*** and run the cell, a popup window will appear that describes the usage of the function. it can be done for seeking help for any function. ***range()*** function can be called in many variations. The normally used ones are described below:
    - **range (stop):** Returns an object that produces a sequence of integers from 0 to stop (exclusive) incremented by 1. For example: range(5) will produce 0, 1, 2, 3 and 4.
    - **range (start, stop):** Returns an object that produces a sequence of integers from start (inclusive) to stop (exclusive) in the step of 1. For example: range(3, 8) will produce 3, 4, 5, 6 and 7.
    - **range (start, stop, step):** Returns an object that produces a sequence of integers from start (inclusive) to stop (exclusive) incremented by step. For example: range(3, 8, 2) will produce 3, 5 and 7.
- The above can be verified writing a simple code that uses *for* statement:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

```
for i in range(3, 8):
    print(i)
```

```
3
4
5
6
7
```

```
for i in range(3, 8, 2):
    print(i)
```

```
3
5
7
```

- As you might have guessed, the *for* statement is used for performing something iteratively or in other words to perform the looping.
- The indentation and block structure concept for the *for* statement is same as it is for *if.*
- Let us say, you have to print the power of 2 for few integers starting from 4 to 10, how will you do that? We know that we have to do this iteratively for 4 to 10. So for loop will be handy as shown below.

```
for i in range(4,11):
    print(i, i**2)
```

```
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Exercise:

1. Fibonacci Series is 1, 1, 2, 3, 5, 8, 13, 21.....A programmer writes a Python code using *for* loop to print the first 15 terms of the Fibonacci Series as shown below. Do you think it is correct? If not, correct the code.

```
x = 1
y = 1
print (x)
print (y)
for i in range(3,16):
    z = x+y
    print(z)
    y = z
```

2. Modify the above code so that user enters in run time how many terms he wants to be printed.

while Loop

- The *for* loop that we just reviewed is very useful when we already know in advance how many times the iterations are required. For example: print something 10 times, calculate first 15 Fibonacci terms etc. Sometimes, it is also necessary to keep performing the iterations until some condition is met. For example, as long as the next Fibonacci term is less than 500. In such situations, *while* loop is useful.
- The indentation and the block construct of *while* loop is identical to *for* loop.
- Let us review a simple example first. We want to calculate the power of integers, until the power is less than or equal to 256. The code below shows how that can be performed using the *while* loop:

```
power = 1
i = 1
while (power < 256):
    power = i**2
    print (i, power)
    i = i+1
```

Exercise:

1. In the above example, if the while expression is changed to ***while (power <= 256)***, what would happen and why?
2. Write the equivalent *while* loop for the following *for* loop. Which one would you prefer?

```
for i in range(3,8):
    print(i)
```

3. Modify the Fibonacci Series code so that it prints all Fibonacci Terms which are less than or equal to 1500.