WORKING WITH CONTROL STATEMENTS

- **About Flow Control**

The flow control statements can be classified into Conditional Statements and Iteration Statements. The Conditional Statements selects a particular set of statements for execution depending upon a specified condition. While the Iteration Statements repeatedly executes a block of statements with respect to some condition.

- Elements of flow control
- **Block/Clause**

To indicate a block of code in Python, you must indent each line of the block by the same amount. The two blocks of code in our example if-statement are both indented four spaces, which is a typical amount of indentation for Python.

- **Conditional Statements**

Conditional Statement in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

--> if statement
--> if..else statements
--> nested if statements
--> if-elif-else

- **Simple if**

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if condition:
        # condition is true
# Statements to execute if
```

Example:

```
i = 10
if (i > 15):
   print ("10 is less than 15")
print ("I am Not in if")
```

Output:

```
I am Not in if
```

- **if...else**

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

Example:

```
i = 20
if (i < 15):
    print ("i is smaller than 15")
    print ("i'm in if Block")
else:
    print ("i is greater than 15")
    print ("i'm in else Block")
print ("i'm not in if and not in else Block")
```

nested if

A nested if is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```

Example:

```
i = 10
if (i == 10):
    #  First if statement
    if (i < 15):
```

```
    print ("i is smaller than 15")
# Nested - if statement
# Will only be executed if statement above
# it is true
if (i < 12):
    print ("i is smaller than 12 too.....")
else:
    print ("i is greater than 15")
```

Output:
i is smaller than 15
i is smaller than 12 too.....

- **if...elif...else**

The elif is short for else if. It allows us to check for multiple expressions.

If the condition for if is False, it checks the condition of the next elif block and so on.

If all the conditions are False, body of else is executed.

Only one block among the several if...elif...else blocks is executed according to the condition.

The if block can have only one else block. But it can have multiple elif blocks.

Syntax:

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

Example:

```
i = 20
```

```
if (i == 10):
    print ("i is 10")
elif (i == 15):
    print ("i is 15")
elif (i == 20):
    print ("i is 20")
else:
    print ("i is not present")
```

Output:

i is 20

- **Looping Statements**

Python programming language provides following types of loops to handle looping requirements. Python provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

- **while loop**

In python, while loop is used to execute a block of statements repeatedly until a given a condition is satisfied. And when the condition becomes false, the line immediately after the loop in program is executed.

Syntax:

```
while expression:
   statement(s)
```

Example:

```
count = 0
while (count < 3):
   count = count + 1
   print("Hello While loop")
```

output:

Hello While loop
Hello While loop
Hello While loop

- **while ... else**

Using else statement with while loops: As discussed above, while loop executes the block until a condition is satisfied. When the condition becomes false, the statement immediately after the loop is executed.
The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed

Syntax:

```
while condition:
    # execute these statements
else:
    # execute these statements
```

Example:

```
count = 0
while (count < 3):
   count = count + 1
   print("Hello while loop")
else:
   print("In Else Block")
```

Output:

```
Hello while loop
Hello while loop
Hello while loop
In Else Block
```

- **for loop**

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc**.**

Syntax:

```
for iterator_var in sequence/range:
   statements(s)
```

Example:

```
fruits = ["apple", "banana", "cherry"]
```

```
for x in fruits:
  print(x)
```
output:
apple
banana
cherry

- **for ... else**

The else block just after for/while is executed only when the loop is NOT terminated by a break statement.
Example:

```
for i in range(1, 4):
    print(i)
else:  # Executed because no break in for
    print("No Break")
```

output:
1
2
3
No Break

using range() in for loop

The range() is a built-in function of Python which returns a range object, which is nothing but a sequence of integers. i.e., Python range() generates the integer numbers between the given start integer to the stop integer, which is generally used to iterate over with for loop.

Python range() function syntax and arguments

Syntax:
range (start, stop[, step])

- range() takes three arguments.
- Out of the three 2 arguments are optional. I.e., Start and Step are the optional arguments.
- A **start** argument is a starting number of the sequence. i.e., lower limit. By default, it starts with 0 if not specified.

- A **stop** argument is an upper limit. i.e.generate numbers up to this number, The range() function doesn't include this number in the result.
- The **step** is a difference between each number in the result. The default value of the step is 1 if not specified.

three variant of range() function.

Example one – Using only one argument in range()

```
print("Print first 5 numbers using range function")
for i in range(5):
    print(i, end=', ')
```

Output:

```
Print first 5 numbers using range function
0, 1, 2, 3, 4,
```

Note: Only stop argument is passed to range() function.
So by default, it takes start = 0 and step = 1.

Example Two – using two arguments (i.e., start and stop) in range() function

```
print("Print integers within given start and stop number using range() function")
for i in range(5, 10):
    print(i, end=', ')
```

Output:

```
Print integers within given start and stop number using range() function
5, 6, 7, 8, 9,
```

Note: Only two arguments (the start and stop) are passed to the range function.
So by default, it took step argument value as 1.

Example Three – using all three arguments in range() function

```
print("using start, stop, and step arguments in Python range() function")
print("Printing All odd numbers between 1 and 10 using range()")
for i in range(1, 10, 2):
    print(i, end=', ')
```

Output:

using start, stop, and step arguments in Python range() function
Printing All odd numbers between 1 and 10 using range()
1, 3, 5, 7, 9,
All three arguments are specified. i.e. start = 1, stop = 10, step = 2.

Note:- In the above program step value is 2 so the difference between each number is 2.

Points to remember about python range() function arguments

- range() function only works with the integers.
- All arguments must be integers. You can not pass a string or float number or any other type in a start, stop and step argument of a range().
- All three arguments can be positive or negative.
- The step value must not be zero. If a step is zero python raises a ValueError exception.

Example1:

```
print("Double the list numbers using for loop and range() function")
sampleList = [3, 6, 9, 12, 15]
for i in range(len(sampleList)):
    print( "Element Index[", i, "]", "Previous Value ", sampleList[i], "Now ", sampleList[i] * 2)
```

Output:

Double the list numbers using for loop and range() function
Element Index[0] Previous Value 3 Now 6
Element Index[1] Previous Value 6 Now 12
Element Index[2] Previous Value 9 Now 18
Element Index[3] Previous Value 12 Now 24
Element Index[4] Previous Value 15 Now 30

- **workingwith infinite loops and nested loops**

We can create an infinite loop using while statement. If the condition of while loop is always True, we get an infinite loop.

```
# An example of infinite loop
# press Ctrl + c to exit from the loop
while True:
   num = int(input("Enter an integer: "))
   print("The double of",num,"is",2 * num)
```

output:

```
Enter an integer: 3
The double of 3 is 6
Enter an integer: 5
The double of 5 is 10
Enter an integer: 6
The double of 6 is 12
Enter an integer:
Traceback (most recent call last):
```

nested loops:

Python programming language allows to use one loop inside another loop.

Syntax:

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

The syntax for a nested while loop statement in Python programming language is as follows −

```
while expression:
       while expression:
          statement(s)
       statement(s)
```

Example:

```
i = 2
while(i < 50):
   j = 2
    while(j <= (i/j)):
       if not(i%j):
            break
       j = j + 1
```

```
if (j > i/j):
      #print("i=",i)
      #print("j=",j)
      print(i, " is prime")
  i = i + 1
print("Good bye!")
```

nested for loop Example:

```
for i in range(1,6):
       for j in range(i):
          print("*",end=' ')
        print()
```

output:

```
*
* *
* * *
* * * *
* * * * *
```

using while loop print stars:

```
i=6
while(i>0):
   j=6
   while(j>i):
      print("*",end=' ')
      j-=1
   i-=1
   print()
```

output:

```
*
* *
* * *
* * * *
* * * * *
```

Break statement

The break statement is used to terminate the loop or statement in which it is present. After that, the control will pass to the statements that are present after the break statement, if available. If the break statement is present in the nested loop, then it terminates only those loops which contains break statement.

Syntax:

break

EXample:

```
s = 'Hello Naveen'
# Using for loop
for letter in s:
        print(letter)
    # break the loop as soon it sees 'e'
    # or 's'
        if letter == 'e' or letter == 's':
                break

print("Out of for loop")
print()
```

Output:

```
H
e
Out of for loop
```

ii)Example2:

```
i = 0
# Using while loop
while True:
    print(s[i])

    # break the loop as soon it sees 'e'
    # or 's'
    if s[i] == 'e' or s[i] == 's':
        break
    i += 1
```

```
print("Out of while loop")
```

Output:
H
e
Out of while loop

Continue statement
Continue is also a loop control statement just like the break statement. continue statement is opposite to that of break statement, instead of terminating the loop, it forces to execute the next iteration of the loop.
As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.
Syntax:
```
continue
```

Example:

```
for i in range(1, 6):
    if i == 3:
        continue
    else:
        print(i, end = " ")
```

Output:
1 2 4 5

- **Pass statement:**

Definition and Usage

The pass statement is used as a placeholder for future code.

When the pass statement is executed, nothing happens, but you avoid getting an error when empty code is not allowed.

Empty code is not allowed in loops, function definitions, class definitions, or in if statements.

Syntax:

pass

Example:

```
for x in [0, 1, 2]:
  pass
```

output:

#No output given