Control statements

Control statements are statements which control or change the flow of execution. The following are the control statements available in Python:

- if statement
- if... else statement
- if... elif..else statement
- while loop
- for loop
- else suite
- break statement
- continue statement
- pass statement.

Please note that the switch statement found in many languages like C and Java is not available in Python.

The if Statement

This statement is used to execute one or more statements depending on whether a condition is True or not. The syntax or correct format of if statement is given below:

```
if condition:
        statements
```

First, the condition is tested. If the condition is True, then the statements given after colon (:) are executed. We can write one or more statements after colon (:). If the condition is False, then the statements mentioned after colon are not executed.
Ex:

```
num=1
if num ==1:
    print("one")
```

Output:

```
one
```

We can also write a group of statements after colon. The group of statements in Python is called a suite. While writing a group of statements, we should write them all with proper indentation. Indentation represents the spaces left before the statements.
Ex:

```
str='yes'
if str =='yes':
    print("yes")
    print("This is your response")
```

Output:

```
yes
This is your response
```

Observe that every print() function mentioned after colon is starting after 4 spaces only. When we write the statements with the same indentation, then those statements are considered as a suite (or belonging to the same group).

The if... else Statement

The if else statement executes a group of statements when a condition is True; otherwise, it will execute another group of statements. The syntax of if... else statement is given below:

```
if condition:
        Statements1
Else:
        statements2
```

If the condition is True, then it will execute statements1 and if the condition is False. then it will execute statements2. It is advised to use 4 spaces as indentation before statements and statements2. In Program 4, we are trying to display whether a given number is even or odd. The logic is simple. If the number is divisible by 2, then it is an even number, otherwise, it is an odd number. To know whether a number is divisible by 2 or not, we can use the modulus operator (%). This operator gives the remainder of division. If the remainder is 0, then the number is divisible, otherwise not.
Ex:

```
x=10
if x % 2==0:
    print(x,"is even number")
else:
    print(x," is odd number")
```

Output:

```
10 is even number
```

The same program can be rewritten to accept input from the keyboard.
Ex:

```
x= int(input("Enter a number:"))
if x % 2==0:
    print(x,"is even number")
else:
    print(x," is odd number")
```

Output:

```
==================
Enter a number:3
3  is odd number
>>>
```

The if..elif..else statement:

Sometimes, the programmer has to test multiple conditions and execute statements depending on those conditions. if... elif. else statement is useful in such situations. Consider the following syntax of if... elif.. else statement:

if condition1:
Statements1
elif condition2:
Statements2
elif condition3:
statements3
else:
statements4

When condition1 is True, the statements will be executed. If condition1 is False, the condition2 is evaluated. When condition2 is True, the statement 2 will be executed When condition2 is False, the condition 3 is tested. If condition 3 is True, then statements will be executed. When condition3 is False, the statements will be executed. It means statements 4 will be executed only if none of the conditions are True Observe colon (:) after each condition. Ex:

```
num=-5
if num==0:
    print (num, "is zero")
elif num>0:
    print(num,"is positive")
else:
    print (num, "is negative")
```

Output:

```
-5 is negative
```

The while loop

A statement is executed only once from top to bottom. For example, if is a statement that is executed by a Python interpreter only once. But a loop is useful to execute repeatedly. For example, while and for are loops in Python. They are useful to execute a group of statements repeatedly several times.

The while loop is useful to execute a group of statements several times repeatedly depending on whether a condition is True or False. The syntax or format of while loop is:

```
while condition:
statements
```

Here, statements represent one statement or a suite of statements. Python interpreter first checks the condition. If the condition is True, then it will execute the statements written after colon (:) After executing the statements, it will go back and check the condition again. If the condition is again found to be True, then it will again execute the statements. Then it will go back to check the condition once again. In this way, as long as the condition is True, the Python interpreter executes the statements again and again. Once the condition is found to be False, then it will come out of the while loop.

In the following program, we are using a while loop to display numbers from 1 to 10. Since, we should start from 1, we will store '1' into a variable x. Then we will write a while loop as: while x<=10:

Observe the condition. It means, as long as x value is less than or equal to 10, continue the while loop. Since we want to display 1,2,3,... up to 10, we need this condition. To display x value, we can use:

```
print(x)
```

Once this is done, we have to increment x value by 1, by writing x+1 or x = x+1.
Ex:

```
x=1
while x<=10:
    print(x)
    x+=1
print("End")
```

Output:

```
1
2
3
4
5
6
7
8
9
10
End
```

Ex:

```
x=100
while x>=100 and x<=120:
    print(x)
    x+=2
```

Output:

```
100
102
104
106
108
110
112
114
116
118
120
```

The for Loop

The for loop is useful to iterate over the elements of a sequence. It means, the for can be used to execute a group of statements repeatedly depending upon the numb elements in the sequence. The for loop can work with sequences like string, list,tuple,range etc. The syntax of the for loop is given below:

for var in sequence:

statements

The first element of the sequence is assigned to the variable written after for and then the statements are executed. Next, the second element of the sequence is assigned to the variable and then the statements are executed a second time. In this way, for each element of the sequence, the statements are executed once. So, the for loop is executed as many times as there are a number of elements in the sequence.

Ex:

```
str='Hello'
for ch in str:
    print(ch)
```

Output:

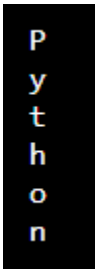
```
P
y
t
h
o
n
```

In the above program, the string 'str' contains Hello'. There are 6 characters in the string. The for loop has a variable 'ch'. First of all, the first character of the string, i.e. H' is stored into ch and the statement, i.e. print(ch) is executed. As a result, it will display H'. Next, the second character of the string, i.e. 'e' is stored into ch. Then print(ch) is once again executed and 'e' will be displayed. In the third iteration, the third character 1 will be displayed. In this way the for loop is executed for 6 times and all the 6 characters are displayed in the output.

In the program below,we are using the for loop to display the elements of the string using index'. An index represents the position number of elements in the string or sequence. For example, str[0] represents the 0th character, i.e. 'H' and str[1] represents the first character, i.e. 'e', and so on. Hence we can take an index 1' that may change from 0 to n 1 where 'n' represents the total number of elements. We can use a range(n) object that generates numbers from 0 to n-1. Ex:

```
str='Python'
n=len(str)
for i in range(n):
    print(str[i])
```

Output:



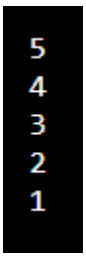
Here, 'n' value will be 6 as the length of the string is 6. We used rangeln) that gives the numbers from 0 to n-1. Hence for loop repeats from 0 to 5 and print() function displays str[0], stri, str[5] It means H.e.l.l.o will be displayed. Please note that the len() function gives the total number of elements in a sequence like string, list or tuple.

The range object, also known as range() function in Python, is useful to provide a sequence of numbers. range(n) gives numbers from 0 to n-1. For example, if we write range(10), it will return numbers from 0 to 9. We can also mention the starting number, ending number, as range(5, 10). In this case, it will give numbers from 5 to 9. We can also specify the step size. The step size represents the increment in the value of the variable at each step. For example, range(1, 10, 2) will give numbers from 1 to 9 in steps of 2. That means, we should add 2 to each number to get the next number. Thus, it will give the numbers: 1, 3, 5, 7 and 9.

Let's write a program to display numbers from 10 to 1 in descending order using the range) object. In this case, we use range(10, 0, -1). Here, the starting number is 10 and the ending number is one before 0. The step size is -1. It means we should decrement 1 every time. Thus we will get 10, 9, 8, up to 1. Observe that it will not return 0 as the last number. It will stop at 1 that is one number before 0.
Ex:

```
for x in range (5,0,-1):
    print (x)
```

Output:



We will use a for loop to access the elements of a list. As we know, a list is a sequence that contains a group of elements. It is like an array. But the difference is that an array stores only one type of element; whereas, a list can store different types of elements. Let's write a program to create a list and retrieve elements from the list using a for loop.

```
list=[10,20.5, 'A', 'America']
for element in list:
    print(element)
```

Output:

```
10
20.5
A
America
```

In the above program, each element of the list is stored into the variable 'element. It means, in the first iteration, element stores 10. In the second iteration, it stores 20.5. In the third iteration, it stores 'A' In the fourth iteration, it stores 'America'. Hence, print(element) displays all the elements of the list.

```
list=[10,20,30,40,50]
sum=0
for i in list:
    print(i)
    sum+=i
print('Sum= ',sum)
```

Output:

```
10
20
30
40
50
Sum=  150
```

Observe that there are only two statements in the for loop. They are:

print(i)

sum+=i

If we want to write the same program using while loop

```
list=[10,20,30,40,50]
sum=0
i=0
while i <len(list):
    print(list[i])
    sum+=list[i]
    i+=1
print("sum= ", sum)
```

Output:

```
10
20
30
40
50
Sum=  150
```

Infinite Loops

Please see the following loop:

```
i=1
while i<=10:
    print(i)
    i+=1
```

Here, ‘i’ value starts at 1. print(i) will display 1. Then the value of ‘i' is incremented by 1 so that it will become 2. In this way, ‘i' values 1, 2, 3,.... up to 10 are displayed. When the ‘i' value is 11, the condition, i.e. i<10 becomes 'False' and hence the loop terminates.

In the previous while loop, what happens if we forget to write the last statement, i.e. =1? The initial ‘i' value 1 is displayed first, but it is never incremented to reach 10. Hence this loop will always display 1 and it never terminates. Such a loop is called an infinite loop. An infinite loop is a loop that executes forever. So, the following is an example for an infinite loop:

```
i=1
while i<=10:
    print(i)
```

It will display the value 1 forever. To stop the program, we have to press Control+C at the system prompt. Another way of creating an infinite loop is to write ‘True' in the condition part of the while loop so that the Python interpreter thinks that the condition is True always and hence executes it forever. See the example:

```
while(True):
    print("Hi")
```

This loop will always display 'Hi' without stopping since the condition is always read as 'True' always. Infinite loops are drawbacks in a program because when the user is caught in an infinite loop, he cannot understand how to come out of the loop. So, it is always recommended to avoid infinite loops in any program.

Nested Loops

It is possible to write one loop inside another loop. For example, we can write a for loop inside a while loop or a for loop inside another for loop. Such loops are called 'nested loops'. Take the following for loops:

```
for i in range (3):
    for j in range(4):
        print('i=',i,'\t', 'j=',j)
```

Output:

```
i= 0    j= 0
i= 0    j= 1
i= 0    j= 2
i= 0    j= 3
i= 1    j= 0
i= 1    j= 1
i= 1    j= 2
i= 1    j= 3
i= 2    j= 0
i= 2    j= 1
i= 2    j= 2
i= 2    j= 3
```

The outer for loop repeats for 3 times by changing i values from 0 to 2. The inner for loop repeats for 4 times by changing j values from 0 to 3. Observe the indentation (4 spaces) before the inner for loop. The indentation represents that the inner for loop is inside the outer for loop. Similarly, print() function is inside the inner for loop. When the outer for loop is executed once, the inner for loop is executed for 4 times. It means, when i value is 0, j values will change from 0 to 3. So, print() function will display the following output:

```
i= 0     j= 0
i= 0     j= 1
i= 0     j= 2
i= 0     j= 3
```

Once the inner for loop execution is completed (j reached 3), then the Python interpreter will go back to the outer for loop to repeat the execution for a second time. This time, 'i' value will be 1. Again, the inner for loop is executed for 4 times. Hence the print function will display the following output:

```
i= 1     j= 0
i= 1     j= 1
i= 1     j= 2
i= 1     j= 3
```

Since the inner for loop execution is completed, again the interpreter will go back to outer for loop. This time i value will be 2 and the output will be:

```
i= 2     j= 0
i= 2     j= 1
i= 2     j= 2
i= 2     j= 3
```

In this way, the print() function in the inner for loop is executed for 12 times. If we observe the output given above, we can understand that the outer for loop is executed 3 times and the inner for loop is executed 4 times. Therefore the print function in the inner for loop is executed for 3x4=12 times.

We will write a Python program to display stars (s) in a right angled triangular form The expected output in our program is given below:

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

In the first row, there is only one star. In the second row, there are two stars. In the third row, there are three stars. It means,

Row number = Row number number of stars in that row.

Since there are 10 rows, we can write an outer for loop that repeats for 10 times from 1 to 10 as:

for i in range(1, 11): # to display 10 rows

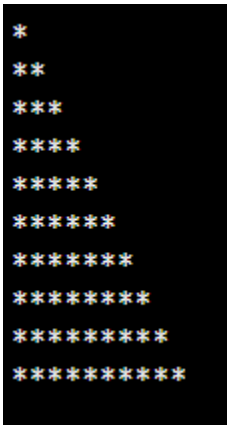
To represent the columns (or stars) in each row, we can write an inner for loop. This for loop should repeat as many times as the row number indicated by the outer for loop. Suppose the row number is ‘i' then the inner for loop should repeat for 1 to i times. Hence the inner for loop can be written as:

for j in range(1, i+1): #repeat for 1 to i times.

Ex:

```
for i in range (1,11):
    for j in range(1, i+1):
        print("*",end="")
    print()
```

Output:



In the above program, observe the following print() statement:

```
print("*",end="")
```

This will display the star symbol. The end=” ” represents that it should not throw the cursor to the next line after displaying each star. So, all the stars are displayed in the same line. But we need to throw the cursor into the next line when there is a new row starting, i.e. when the ‘i’ value changes. This can be achieved by another print statement in the outer for loop that does not display anything but simply throws the cursor into the next line.

We can do the same thing using a single loop.

```
for i in range (1,11):
        print("* "*(i))
```

Output:

This is a simple and elegant style of writing a program in python.

The else Suite

In Python, it is possible to use else statement along with for loop or while loop

for(var in sequence):	while(condition):
statements	statements
else:	else:
statements	statements

The statements written after 'else' are called suites. The else suite will be always executed irrespective of whether the statements in the loop are executed or not.
For example,

```
for i in range (3):
    print("yes")
else:
    print("No")
```

Output:

```
yes
yes
yes
No
```

It means, the for loop statement is executed and also the else suite is executed. Suppose we write:

```
for i in range (0):
    print("yes")
else:
    print("no")
```

Output:

```
no
```

Here, the statement in the for loop is not even executed once, but the else suite is executed as usual. So, the point is that the else suite is always executed. But then where is this else useful?

Sometimes, we write programs where searching for an element is done in the sequence.When the element is not found, we can indicate that in the else suite easily. In this case, else with for loop or while loop is very convenient. Let's now write a program where we take a list of elements and use a for loop to search for a particular element in the list. If the element is found in the list, we will display that it is found in the group, else we will display a message that the element is not found in the list. This is displayed using the else suite.
Ex:

```
group1=[1,2,3,4,5]
search=int(input('Enter element to search:'))
for element in group1:
    if search==element:
        print('Element found in group1')
        break
else:
    print('element not found in group1')
```

Output:

```
Enter element to search:4
Element found in group1
>>>
========== RESTART: C
Enter element to search:6
element not found in group1
>>>
```

The break Statement

The break statement can be used inside a for loop or while loop to come out of the loop. When 'break' is executed, the Python interpreter jumps out of the loop to process the next statement in the program.We have already used break inside for loop in previous program. When the element is found, it would break the for loop and comes out. We can also use break inside a while loop. Suppose, we want to display numbers from 10 to 1 in descending order using a

while loop. For this purpose, we can write a simple while loop as:

Ex:

```
x=10
while x>=1:
    print('x=',x)
    x-=1
print("out of loop")
```

Output:

```
x= 10
x= 9
x= 8
x= 7
x= 6
x= 5
x= 4
x= 3
x= 2
x= 1
out of loop
```

The continue statement

The continue statement is used in a loop to go back to the beginning of the loop.It means, when continue is executed, the next repetition will start. When continue is executed, the subsequent statements in the loop are not executed. In the following program, the while loop repeats for 10 times from 0 to 9. Every time. ‘x' value is displayed by the loop. When x value is greater than 5, continue' is executed that makes the Python interpreter go back to the beginning of the loop. Thus the next statements in the loop are not executed. As a result, the numbers up to 5 are only displayed.

```
x=0
while x<10:
    x+=1
    if x>5:
        continue
    print('x= ',x)
print("out of loop")
```

Output:

```
x=  1
x=  2
x=  3
x=  4
x=  5
out of loop
```

The pass Statement

The pass statement does not do anything. It is used with an if statement or inside a loop to

represent no operation. We use pass statement when we need a statement syntactically, but we do not want to do any operation.

The continue statement in the previous program redirected the flow of execution to the beginning of the loop. If we use ‘pass’ in the place of ‘continue’, the numbers from 1 to 10 are displayed as if there is no effect of ‘pass’.

Ex:

```
x=0
while x<10:
    x+=1
    if x>5:
        pass
    print('x=',x)
print("Out of loop")
```

Output:

```
x= 1
x= 2
x= 3
x= 4
x= 5
x= 6
x= 7
x= 8
x= 9
x= 10
Out of loop
```