CHAPTER 3

Control Statements

Loop Statements

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −

Python programming language provides following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	**while loop** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	**for loop** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**nested loops** You can use one or more loop inside any another while, for or do..while loop.

The for Loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax of for Loop

```
for val in sequence:
        Body of for
```

Here, val is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

```
# Program to find the sum of all numbers stored in a list
# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
# variable to store the sum
sum = 0
# iterate over the list
for val in numbers:
        sum = sum+val
# Output: The sum is 48
print("The sum is", sum)
```

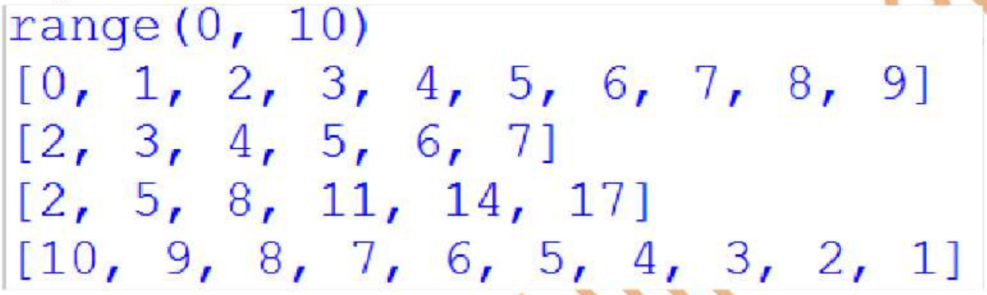
The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers). We can also define the start, stop and step size as range(start,stop,step). step defaults to 1 if not provided.

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go. To force this function to output all the items, we can use the function list(). The following example will clarify this.

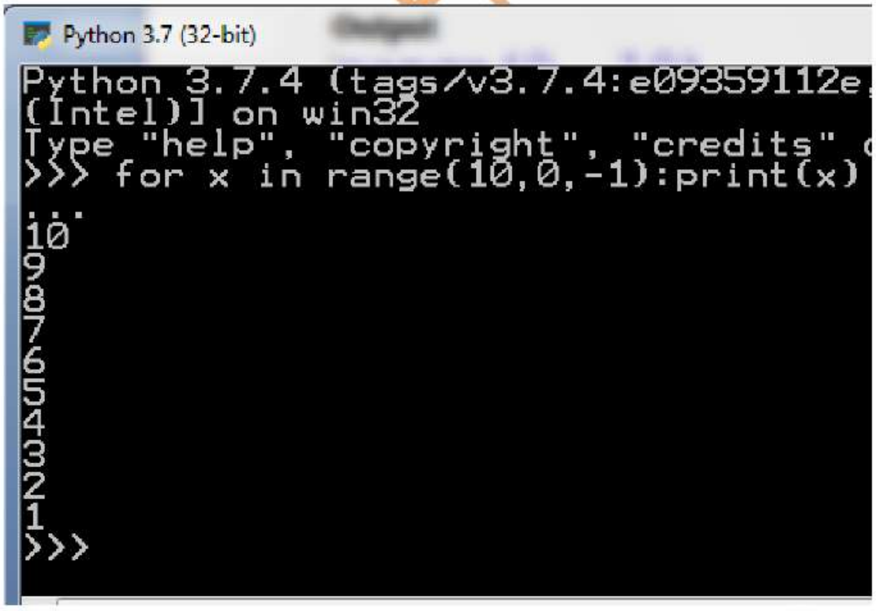
```
print(range(10))
print(list(range(10)))
print(list(range(2, 8)))
print(list(range(2, 20, 3)))
print(list(range(10,0,-1))
```

Output



```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7]
[2, 5, 8, 11, 14, 17]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
for x in range(10,0,-1): print(x)
```

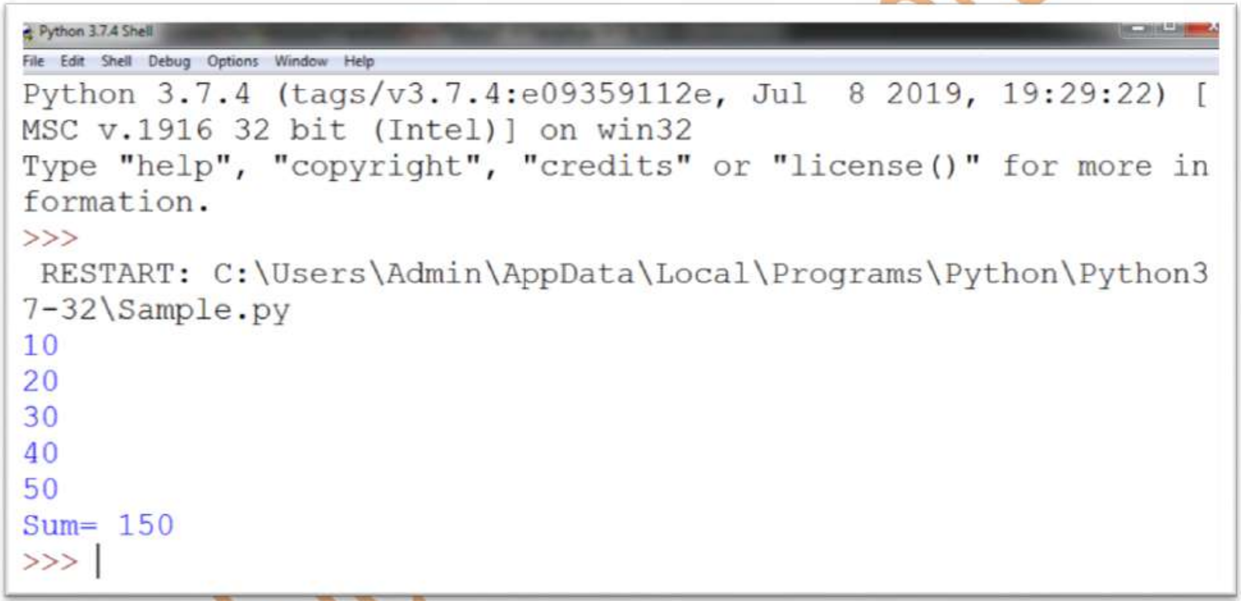


Python 3.7 (32-bit)

```
Python 3.7.4 (tags/v3.7.4:e09359112e,
(Intel)] on win32
Type "help", "copyright", "credits" o
>>> for x in range(10,0,-1):print(x)
...
10
9
8
7
6
5
4
3
2
1
>>>
```

```
#to find sum of list of numbers using for loop
list = [10,20,30,40,50]
sum =0
for i in list:
        print(i)
        sum+=i
print("Sum+=, sum)
```

Output



```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [
MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more in
formation.
>>>
 RESTART: C:\Users\Admin\AppData\Local\Programs\Python\Python3
7-32\Sample.py
10
20
30
40
50
Sum= 150
>>> |
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the **len()** function to iterate though a sequence using indexing. Here is an example.

```
str="GEHU"
for i in str: print(i)
G
E
H
U
```

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

```
0
1
2
3
```

for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts. The 'break' statement can be used to stop a for loop. In such case, the else part is ignored.

One can use an "else" clause with a "for" loop in Python. It's a special type of syntax that executes only if the for loop exits naturally, without any break statements.

Hence, a for loop's else part runs if no break occurs. Here is an example to illustrate this.

```
l = [1,5,9,14,15]
for num in l:
      if l%2==0:
          print(num)
          break # Case1: Break is called, so 'else' wouldn't be executed.
else: # Case 2: 'else' executed since break is not called
    print("No call for Break. Else is executed")
```

```
digits = [0, 1, 5]
for i in digits:
    print(i)
else:
    print("No items left.")
```

When you run the program, the output will be:

```
0
1
5
No items left.
```

The while loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true. The while loop is useful to execute a group of statement several times repeatedly depending on whether condition is True or False.

Syntax of while Loop in Python

```
while test_expression:
   Body of while
```

In while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

In Python, the body of the while loop is determined through indentation. Body starts with indentation and the first unindented line marks the end. Python interprets any non-zero value as True. None and 0 are interpreted as False.

```
# Program to add natural  numbers upto  sum = 1+2+3+...+n
# To take input from the user,
# n = int(input("Enter n: "))
n = 10
# initialize sum and counter
sum = 0
i = 1
while i <= n:
    print(i)
    sum = sum + i
    i = i+1    # update counter
# print the sum
print("The sum is", sum)
```

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (10 in our program). We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop).

while loop with else

Same as that of for loop, we can have an optional else block with while loop as well. The else part is executed if the condition in the while loop evaluates to False. The while loop can be terminated with a break statement. In such case, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

Here is an example to illustrate this.

```
# Example to illustrate the use of else statement  with the while loop
counter = 0
while counter < 3:
    print("Inside loop")
    counter = counter + 1
```

```
else:
   print("Inside else")
```

Nested Loops

It is possible to write a 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'. For example:

```
for i in range(1,6):    # to display 6 rows
        for j in range(1,i+1):    #no. of  stars = row number
                print'*', end=' ')
        print()
```

Output:

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

In above program we have used two for loops; one inside another. These are called nested for loops. We can also rewrite the same program with a single for loop. Now, consider another example:

```
for i in range(1,11):
        print('* ' *(i))   # repeat star for I times
```

Output:

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

Try some more example of single loop in place of nested loop:

```
# to display the stars in equilateral triangular form
n =40
for i in range(1,11):
    print(‘ ‘*n, end = ‘ ’)
    print(“* “ *(i))
    n-=1
```

Output:

```
        *

      *   *

    *   *   *

  *   *   *   *

*   *   *   *   *
```

Loop Control Statements or Jump Statement

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	**break statement** Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**pass statement** The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.
4	**assert** The assert statement is useful to check if a particular condition is fulfilled or not.

In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

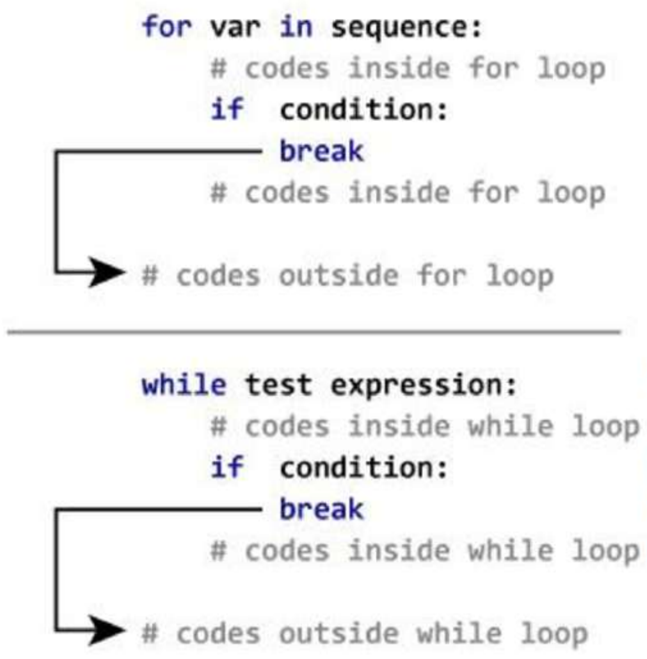
The break and continue statements are used in these cases.

Python break statement

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

The working of break statement in for loop and while loop is shown below.



```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

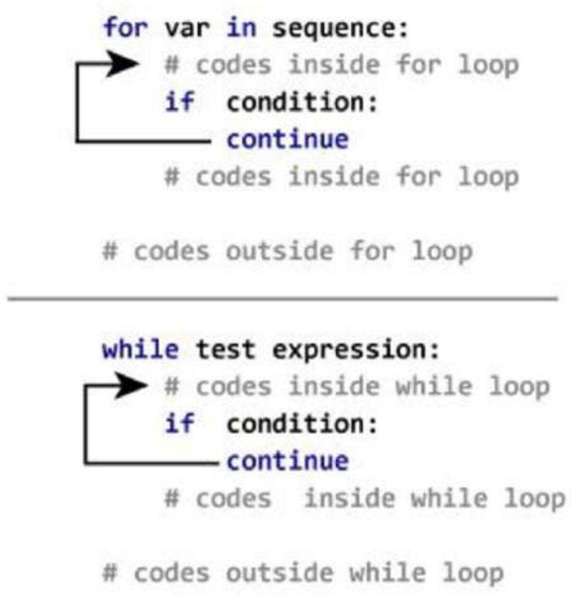
```
# Use of break statement inside loop

for val in "string":
    if val == "i":
        break
    print(val)
print("The end")
```

Python continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

The working of continue statement in for and while loop is shown below.



```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

```
# Program to show the use of continue statement inside loops
for val in "string":
    if val == "i":
        continue
    print(val)
print("The end")
```

```
# program to print all the numbers between 1 to 100
#which is not divisible by 9
for  val in range(1,100):
        if val%9==0:
                continue
        print(val)
```

pass statement

The pass statement does not do anything. It is used with 'if' statement or inside a loop to represent no operation. However, nothing happens when pass is executed. It results into no operation (NOP).

We use pass statement when we need a statement syntactically but we do not want to do any operation.

The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.

We generally use it as a placeholder.

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the pass statement to construct a body that does nothing.

The 'continue' statement in above example redirected the flow of execution to the beginning of the loop. If we use 'pass' in the place of 'continue', the number from 1 to 100 which is not divisible by 9 are displayed as if there is no effect of 'pass'.

```
# program to print all the numbers between 1 to 100
#which is not divisible by 9
for  val in range(1,100):
        if val%9==0:
                pass            #pass in place of continue
        print(val)
```

```
# pass is just a placeholder for
# functionality to be added later.
sequence = {'p', 'a', 's', 's'}
for val in sequence:
   pass
```

We can do the same thing in an empty function or class as well.

```
def function(args):
    pass
```

```
class example:
    pass
```

The assert Statement

The assert statement is useful to check if a particular condition is fulfilled or not. The syntax is as follows:

assert expression, message

In the above syntax, the message is not compulsory. Let's take an example. If we want to assure that the user should enter only a number greater than 0, we can use assert statement as:

```
assert x >0, "Wrong input entered"
```

Decision Control Statements

Decision making is required when we want to execute a code only if a certain condition is satisfied.

The if...elif...else statement is used in Python for decision making.

Python if Statement Syntax

```
if test expression:
    statement(s)
```

Here, the program evaluates the test expression and will execute statement(s) only if the text expression is True.
If the text expression is False, the statement(s) is not executed.
In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and the first unindented line marks the end.
Python interprets non-zero values as True. None and 0 are interpreted as False.

```
# If the number is positive, we print an appropriate message
num = 3
```

```
if num > 0:
    print(num, "is a positive number.")
print("This is always printed.")
```

```
num = -1
if num > 0:
    print(num, "is a positive number.")
print("This is also always printed.")
```

Output:

```
3 is a positive number
This is always printed
This is also always printed.
```

Python if...else Statement

Syntax of if...else

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

The if..else statement evaluates test expression and will execute body of if only when test condition is True.

If the condition is False, body of else is executed. Indentation is used to separate the blocks.

```
# Program checks if the number is positive or negative
# And displays an appropriate message
num = 3
# Try these two variations as well.
# num = -5
# num = 0
if num >= 0:
    print("Positive or Zero")
```

```
else:
   print("Negative number")
```

Python if...elif...else Statement

Syntax of if...elif...else

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

```
# In this program,
# we check if the number is positive or
# negative or zero and
# display an appropriate message
num = 3.4
# Try these two variations as well:
# num = 0
# num = -4.5
if num > 0:
   print("Positive number")
elif num == 0:
   print("Zero")
else:
   print("Negative number")
```

Python Nested if statements

We can have a if...else statement inside another if...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

Python Nested if Example

```
# In this program, we input a number check if the number is positive or1
# negative or zero and display an appropriate message. This time we use
nested if

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
        else:
            print("Negative number")
```