CHAPTER

6

CONTROL STATEMENTS

When we write a program, the statements in the program are normally executed one by one. This type of execution is called 'sequential execution'. Let's write a program to understand about the concept of sequential execution. In Program 1, we are calculating the area of a circle. We know that the formula used to calculate the area of circle is pi * r * r. Since 'pi' is a constant in 'math' module, we can refer to it as math.pi. Also, to find the square of r, we can use exponentiation operator (**) as r **2. So the area of a circle is given by math.pi * r**2. Since we are using the constant 'pi' from the math module, we are supposed to import that module in our program. Consider Program 1 in which a constant 'pi' is imported from the math module and used to calculate the area of a circle.

Program

Program 1: A Python program to calculate the area of a circle.

```
# to find area of a circle
import math  # here math module is imported
r = float(input('Enter radius: '))
area = math.pi * r**2  # pi is a constant in math module
print('Area of circle= ', area)
print('Area of circle= {:0.2f}'.format(area))
```

Output:

```
C:\>python area.py
Enter radius: 15.5
Area of circle= 754.7676350249478
Area of circle= 754.77
```

If we consider the output of the preceding program, we can understand that the statements written in Program 1 are executed one by one by the Python interpreter. This is called sequential execution. This type of execution is suitable only for developing simple programs. It is not suitable for developing critical programs where complex logic is needed.

To develop critical programs, the programmer should be able to change the flow of execution as needed by him. For example, the programmer may wish to repeat a group of statements several times or he may want to directly jump from one statement to another statement in the program. For this purpose, we need control statements. In this chapter, you will learn more about control statements.

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
- assert statement
- return 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 statement 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.

To understand the simple if statement, let's write a Python program to display a digit in words.

Program

Program 2: A Python program to express a digit in a word.

```
# understanding if statement
num=1
if num==1:
    print("One")
```

Output:

```
C:\>python Demo.py
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. The default indentation used in Python is 4 spaces. Let's write a program to display a group of messages using if statement.

Program

Program 3: A Python program to display a group of messages when the condition is true.

```
# understanding if statement
str = 'Yes'
if str == 'Yes':
    print("Yes")
    print("This is what you said")
    print("Your response is good")
```

Output:

```
C:\>python Demo.py
Yes
This is what you said
Your response is good
```

Observe that every print() function mentioned after colon is starting after 4 spaces only. When we write the statements with same indentation, then those statements are considered as a suite (or belonging to same group). In Program 3, the three print statements written after the colon form a suite.

A Word on Indentation

Understanding indentation is very important in Python. Indentation refers to spaces that are used in the beginning of a statement. The statements with same indentation belong to same group called a *suite*. By default, Python uses 4 spaces but it can be increased or decreased by the programmers. Consider the statements given in Figure 6.1:

```
if x==1:
    print('a')
    print('b')
    if y==2:
        print('c')
        print('d')
print('end')
```

Figure 6.1: Indentation in if... Statements

In Figure 6.1, the statements:

```
if x==1:
print('end')
```

belong to same group as they do not have any spaces before them. So, after executing the if statement, Python interpreter goes to the next statement, i.e. print('end'). Even if the statement 'if x==1' is not executed, interpreter will execute print('end') statement as it is the next executable statement in our program.

In the next level, the following statements are typed with 4 spaces before them and hence they are at the same level (same suite).

```
    print('a')
    print('b')
    if y==2:
```

These statements are inside 'if x==1' statement. Hence if the condition is True (i.e. x==1 is satisfied), then the above 3 statements are executed. Thus, the third statement 'if y==2' is executed only if x==1 is True. At the next level, we can find the following statements:

```
        print('c')
        print('d')
```

These two statements are typed with 8 spaces before them and hence they belong to the same group (or suite). Since they are inside 'if y==2' statement, they are executed only if condition y==2 is True. Table 6.1 shows the outputs that we obtain when the statements given in Figure 6.1 are executed with different value of x and y:

Table 6.1: The Effect of Indentation

Output when x=1, y=0	Output when x=1, y=2	Output when x=0, y=2
a b end	a b c d end	end

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 statements1 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 modulus operator (%). This operator gives remainder of division. If the remainder is 0, then the number is divisible, otherwise not.

Program

Program 4: A Python program to test whether a number is even or odd.

```
# to know if a given number is even or odd
x=10
if x % 2 == 0:
    print(x, " is even number")
else:
    print(x, " is odd number")
```

Output:

```
C:\>python Demo.py
10 is even number
```

The same program can be rewritten to accept input from keyboard. In Program 5, we are accepting an integer number from keyboard and testing whether it is even or odd.

Program

Program 5: A Python program to accept a number from the keyboard and test whether it is even or odd.

```
# to know if a given number is even or odd - v2.0
x = int(input("Enter a number: "))
if x % 2 == 0:
    print(x, " is even number")
else:
    print(x, " is odd number")
```

Output:

```
C:\>python Demo.py
Enter a number: 11
11  is odd number
```

We will write another program where we accept a number from the user and test whether that number is in between 1 and 10 (inclusive) or not. If the entered number is x, then the condition x>=1 and x<=10 checks whether the number is in between 1 and 10.

Program

Program 6: A Python program to test whether a given number is in between 1 and 10.

```
# using 'and' in if ... else statement
x = int(input('Enter a number: '))
if x>=1 and x<=10:
    print("You typed", x, "which is between 1 and 10")
else:
    print("You typed", x, "which is below 1 or above 10")
```

Output:

```
C:\>python Demo.py
Enter a number: 9
You typed 9 which is between 1 and 10
```

Observe the condition after 'if'. We used 'and' to combine two conditions x>=1 and x<=10. Such a condition is called compound condition.

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 statements1 will be executed. If condition1 is False, then condition2 is evaluated. When condition2 is True, the statements2 will be executed. When condition2 is False, the condition3 is tested. If condition3 is True, then statements3 will be executed. When condition3 is False, the statements4 will be executed. It means statements4 will be executed only if none of the conditions are True.

Observe colon (:) after each condition. The statements1, statements2, ... represent one statement or a suite. The final 'else' part is not compulsory. It means, it is possible to write if ... elif statement, without 'else' and statements4. Let's write a program to understand the usage of if ... elif ... else statement. In Program 7, we are checking whether a number is positive or negative. A number becomes positive when it is greater than 0. A number becomes negative when it is lesser than 0. Apart from positive and negative, a number can also become zero (neither +ve nor –ve). All these 3 combinations are checked in the program.

Program

Program 7: A Python program to know if a given number is zero, positive or negative.

```
# to know if a given number is zero or +ve or -ve
num=-5
if num==0:
    print(num, "is zero")
elif num>0:
    print(num, "is positive")
else:
    print(num, "is negative")
```

Output:

```
C:\>python Demo.py
-5 is negative
```

In the previous program, observe the indentation. There are 4 spaces before every statement after colon. Now, we write another program that accepts a numeric digit from keyboard and prints it in words. In this program, we will use several elif conditions. After colon, we are leaving only one space before every print() statement. While this is not recommended way of using indentation, this program will run properly since every statement has equal number of spaces (1 space) as indentation. Consider Program 8.

Program

Program 8: A program to accept a numeric digit from keyboard and display in words.

```
# to display a numeric digit in words
x = int(input('Enter a digit: '))
if x==0: print("ZERO")
elif x==1: print("ONE")
elif x==2: print("TWO")
elif x==3: print("THREE")
elif x==4: print("FOUR")
elif x==5: print("FIVE")
elif x==6: print("SIX")
elif x==7: print("SEVEN")
elif x==8: print("EIGHT")
elif x==9: print("NINE")  # else part not compulsory
```

Output:

```
C:\>python Demo.py
Enter a digit: 5
FIVE
```

In the above program, if we enter a digit '15', then the program does not display any output. On the other hand, if we want to display a message like 'Please enter digit between 0 and 9', we can add 'else' statement at the end of the program, as:

```
else: print('Please enter digit between 0 and 9')
```

The while Loop

A statement is executed only once from top to bottom. For example, 'if' is a statement that is executed by 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' represents 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, 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 Program 9, we are using 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. Now, consider Program 9 in which while loop is used to display numbers.

Program

Program 9: A Python program to display numbers from 1 to 10 using while loop.

```
# to display numbers from 1 to 10
x=1
while x<=10:
    print(x)
    x+=1
print("End")
```

Output:

```
C:\>python Demo.py
1
2
3
4
5
6
```

```
7
8
9
10
End
```

In the previous program, observe that the first two statements are written using same indentation. They are:

```
print(x)
x+=1
```

That means, these two statements will come under one group. Hence, both these statements are executed when the condition is True. The last statement,

```
print("End")
```

is having same indentation as the while. Hence this statement will not come under same group like the previous two statements. This will come under a separate statement having same level as while loop. So, Python interpreter executes it after coming out of while loop. Suppose, we write the print("End")statement with the same indentation as the other two statements as:

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

Then the output will be as shown below:

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

In Program 10, we will use while loop to display even numbers between 100 and 200. We will take x value as 100 since it is the first even number. Every time we will add 2 to the value of x to get the next even number. We will repeat the while loop as long as x>=100 and x<=200.

Program

Program 10: A Python program to display even numbers between 100 and 200.

```
# to display even numbers between 100 and 200
x=100
while x>=100 and x<=200:
    print(x)
    x+=2
```

Output:

```
C:\>python Demo.py
100
102
104
:
:
200
```

We can improve the Program 10 so that the user can enter his choice regarding from where to where he wants to display even numbers. In this program 'm' and 'n' represent the minimum and maximum range of even numbers to be displayed.

Program

Program 11: A Python program to display even numbers between m and n.

```
# to display even numbers between m and n
m, n = [int(i) for i in input("Enter minimum and maximum range:
       ").split(',')]
x=m  # start from m onwards
if x % 2 !=0 :   # if x is not even, start from next number
    x=x+1

while x>=m and x<=n:
    print(x)
    x+=2
```

Output:

```
C:\>python Demo.py
Enter minimum and maximum range: 11,20
12
14
16
18
20
```

The for Loop

The for loop is useful to iterate over the elements of a sequence. It means, the for loop can be used to execute a group of statements repeatedly depending upon the number of elements in the sequence. The for loop can work with sequence 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 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 number of elements in the sequence.

We will write a Python program to retrieve the letters of a string using for loop. Now, consider Program 12.

Program

Program 12: A Python program to display characters of a string using for loop.

```
# to display each character from a string
str='Hello'
for ch in str:
    print(ch)
```

Output:

```
C:\>python Demo.py
H
e
l
l
o
```

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 'l' 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 Program 13, we are using the for loop to display the elements of the string using 'index'. An index represents the position number of element 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 'i' that may change from 0 to n-1 where 'n' represents the total number of elements. We can use range(n) object that generates numbers from 0 to n-1. Now, consider Program 13.

Program

Program 13: A Python program to display each character from a string using sequence index.

```
# to display each character from a string - v2.0
str='Hello'
n = len(str)  # find no. of chars in str
```

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

The output of the above program will be the same as that of Program 12. Here, 'n' value will be 6 as the length of the string is 6. We used range(n) that gives the numbers from 0 to n-1. Hence for loop repeats from 0 to 5 and print() function displays str[0], str[1], ... 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 is 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. This can be verified through Program 14.

Program

Program 14: A Python program to display odd numbers from 1 to 10 using range() object.

```
# to display odd numbers between 1 and 10
for i in range(1, 10, 2):
    print(i)
```

Output:

```
C:\>python Demo.py
1
3
5
7
9
```

Let's write another 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. Now, consider Program 15.

Program

Program 15: A program to display numbers from 10 to 1 in descending order.

```
# to display numbers in descending order
for x in range (10, 0, -1):
    print(x)
```

Output:

```
C:\>python Demo.py
10
```

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

We will use 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 elements; 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 for loop.

Program

Program 16: A program to display the elements of a list using for loop.

```
# take a list of elements
list = [10,20.5,'A','America']

# display each element from the list
for element in list:
    print(element)
```

Output:

```
C:\>python Demo.py
10
20.5
A
America
```

In Program 16, 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.

In Program 17, we are going to take a list of integer numbers. We will use a for loop to find the sum of all those numbers.

Program

Program 17: A Python program to display and find the sum of a list of numbers using for loop.

```
# to find sum of list of numbers using for.
# take a list of numbers
list = [10,20,30,40,50]
sum=0  #initially sum is zero
for i in list:
    print(i)  #display the element from list
    sum+=i    #add each element to sum
print('Sum= ', sum)
```

Output:

```
C:\> python Demo.py
10
20
30
40
50
Sum= 150
```

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

```
print(i)
sum+=i
```

These two statements form a group (or suite) as they are typed with the same indentation, i.e. 4 spaces before each statement. The last print() function does not come into this group as it is typed at the same level as that of for loop. Hence, print() is considered as a next statement after for loop. So, it is executed only after completion of for loop.

If we want to write the same program using while loop, we have to put a few extra statements as seen in Program 18.

Program

Program 18: A Python program to display and sum of a list of numbers using while loop.

```
# to find sum of a list of numbers using while - v2.0
# take a list of numbers
list = [10,20,30,40,50]
sum=0  #initially sum is zero
i=0    #take a variable
while i <len(list):
    print(list[i])  #display the element from list
    sum+=list[i]    #add each element to sum
    i+=1
print('Sum= ', sum)
```

Output:

```
Same as that of previous program.
```

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 '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. 'i+=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 never terminates. Such a loop is called '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 press Control+C at 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("Hai")
```

This loop will always display 'Hai' without stopping since the condition is 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):  # i values are from 0 to 2
    for j in range(4):  # j values are from 0 to 3
        print('i=', i, '\t', 'j=',j)  # display i and j values
```

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 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 Python interpreter will go back to outer for loop to repeat the execution for 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 for 3 times and the inner for loop is executed for 4 times. Therefore the print() function in the inner for loop is executed for 3 X 4 = 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 = 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 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.
```

This logic is given in Program 19.

Program

Program 19: A Python program that displays stars in right angled triangular form using nested loops.

```
# to display stars in right angled triangular form
for i in range(1, 11):  # to display 10 rows
    for j in range(1, i+1):  # no. of stars = row number
        print("* ", end="")
    print()
```

Output:

```
C:\>python Demo.py
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

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 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.

In Program 19, 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 Program 20.

Program

Program 20: A Python program that displays stars in right angled triangular form using a single loop.

```
# to display stars in right angled triangular form - v2.0
for i in range(1, 11):
    print ("* "*(i))  # repeat star for i times
```

The code in Program 20 is an example for elegant (simple and beautiful) style of writing a program in Python.

Program 20 can be improved to display stars in equilateral triangular form. For this purpose, we have to start displaying the stars slightly at the center of the screen or monitor. For this purpose, we should release some spaces and then display the star. This can be done using the following statements:

```
print(' '*n, end='')  # repeat space for n times in the same line.
print ('* ')) # then display the star and throw cursor to next line.
```

Every time, we need to reduce the number of spaces so that the stars will form the equilateral triangle. For this purpose, we write:

```
n-=1   # reduce the spaces by 1 in every row.
```

Program

Program 21: A Python program to display the stars in an equilateral triangular form using a single for loop.

```
# to display stars in equilateral triangular form
n=40
for i in range(1, 11):
    print(' '*n, end='')  # repeat space for n times
    print ('* '*(i))  # repeat star for i times
    n-=1
```

Output:

```
C:\>python Demo.py
                                        *
                                      * *
                                    * * *
                                  * * * *
                                * * * * *
                              * * * * * *
                            * * * * * * *
                          * * * * * * * *
                        * * * * * * * * *
                      * * * * * * * * * *
```

Program 21 can also be rewritten using the elegant style of Python as:

```
n=40
for i in range(1, 11):
    print(' '*(n-i) + '* '*(i))
```

Finally, we will write another program using nested loops to display numbers from 1 to 100 in 10 rows and 10 columns. To display numbers the following loops are sufficient:

```
for x in range(1, 11):
    for y in range(1, 11):
        print(x*y, end='')  # print product of x and y
    print()
```

But this code gives the following output that looks awkward:

```
12345678910
2468101214161820
36912151821242730
481216202428323640
```

```
5101520253035404550
6121824303642485460
7142128354249566370
8162432404856647280
9182736455463728190
10203040506070809010
```

We need to display each number in a column with fixed size so that the output will look nice. For this purpose, we can specify column size using replacement field {} and format() method as:

```
print('{:8}'.format(x*y), end='')
```

Here, {:8} represents that the produce value (x*y) should be displayed in the field size of 8. Now, see the Program 22. The output is displayed in Figure 6.2.

Program

Program 22: A Python program to display numbers from 1 to 100 in a proper format.

```
# Displaying numbers from 1 to 100 in 10 rows and 10 cols.
for x in range(1, 11):
    for y in range(1, 11):
        print('{:8}'.format(x*y), end='')  # each column size is 8
    print()
```

Output:

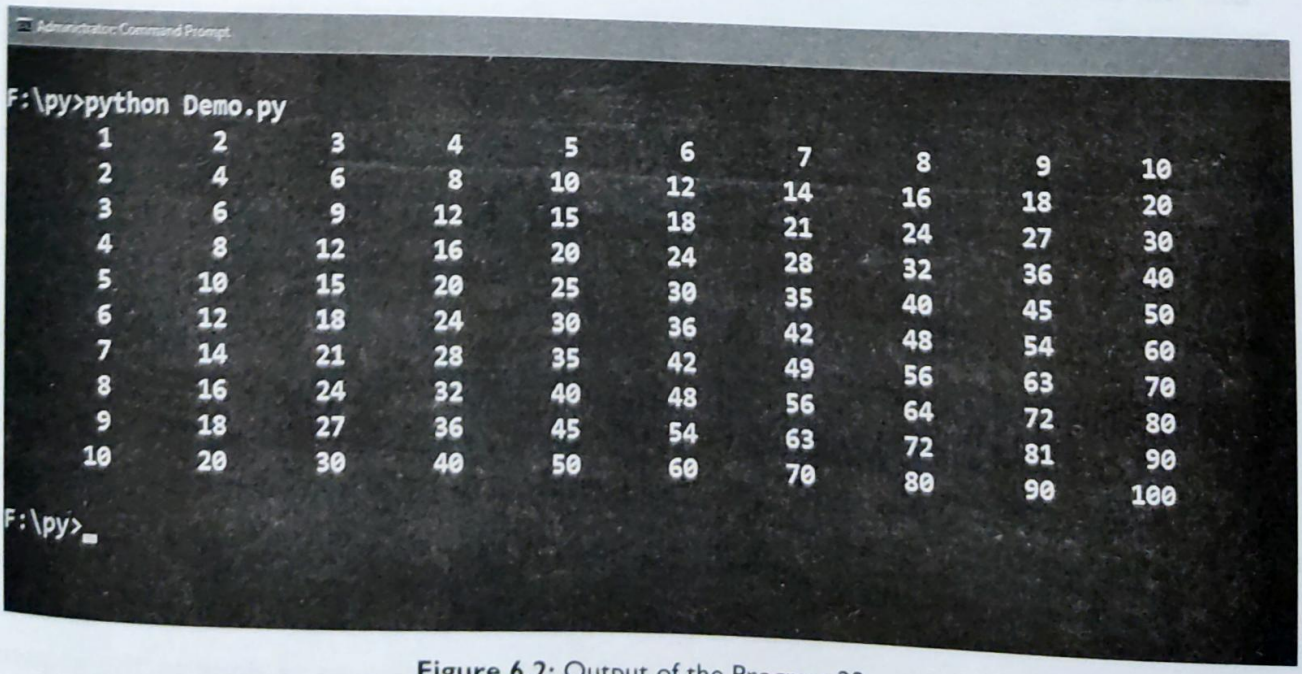

Figure 6.2: Output of the Program 22

The else Suite

In Python, it is possible to use 'else' statement along with for loop or while loop in the form shown in Table 6.2:

Table 6.2: Else suite with loops

for with else	while with else

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

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

```
for i in range(5):
    print("Yes")
else:
    print("No")
```

this will display the following output:

```
Yes
Yes
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")
```

This will display the following 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 this: the else suite is always executed. But then where is this else suite 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 using for loop we will search for a particular element in the list. If the element is found in the list, we will display that is found in the group; else we will display a message that the element is not found in the list. This is displayed using else suite.

Program

Program 23: A Python program to search for an element in the list of elements.

```
# searching for an element in a list
group1 = [1,2,3,4,5]   #take a list of elements
search = int(input('Enter element to search: '))
for element in group1:
    if search == element:
        print('Element found in group1')
        break  # come out of for loop
else:
    print('Element not found in group1')   # this is else suite
```

Output:

```
C:\>python Demo.py
Enter element to search: 4
Element found in group1

C:\>python Demo.py
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 Program 23. 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 while loop. For this purpose, we can write a simple while loop as:

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

The above loop will display the following output:

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

Now, suppose we want to display x values up to 6 and if it is 5 then we want to come out of the loop, we can introduce a statement like this:

```
if x==5:  # if x is 5 then come out from while loop
    break
```

This is what is done in Program 24. A break statement is breaking the while loop when a condition is met.

Program

Program 24: A Python program to display numbers from 10 to 6 and break the loop when the number about to display is 5.

```
# Using break to come out of while loop
x = 10
while x>=1:
    print ('x= ', x)
    x-=1
    if x==5:  # if x is 5 then come out from while loop
        break
print("Out of loop")
```

Output:

```
x= 10
x= 9
x= 8
x= 7
x= 6
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 Program 25, 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 to 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.

Program

Program 25: A Python program to display numbers from 1 to 5 using continue statement.

```
# Using continue to execute next iteration of while loop
x = 0
while x<10:
    x+=1
    if x>5:  # if x > 5 then continue next iteration
```

```
            continue
        print ('x= ', x)
print("Out of loop")
```

Output:

```
C:\>python Demo.py
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 '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 Program 25 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'.

Program

Program 26: A program to know that pass does nothing.

```
# Using pass to do nothing
x = 0
while x<10:
    x+=1
    if x>5:  # if x > 5 then continue next iteration
        pass
    print ('x= ', x)
print("Out of loop")
```

Output:

```
C:\>python Demo.py
x= 1
x= 2
x= 3
x= 4
x= 5
x= 6
x= 7
x= 8
x= 9
x= 10
Out of loop
```

A more meaningful usage of the 'pass' statement is to inform the Python interpreter not to do anything when we are not interested in the result. This can be seen in Program 27.

Program

Program 27: A Python program to retrieve only negative numbers from a list of numbers.

```
# Retrieving only negative numbers from a list.
num = [1,2,3,-4,-5,-6,-7, 8, 9]
for i in num:
    if(i>0):
        pass  # we are not interested
    else:
        print(i)  # this is what we need
```

Output:

```
C:\>python Demo.py
-4
-5
-6
-7
```

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"
```

In this case, the Python interpreter checks if x>0 is True or not. If it is True, then the next statements will execute, else it will display AssertionError along with the message "Wrong input entered". This can be easily understood from Program 28.

Program

Program 28: A program to assert that the user enters a number greater than zero.

```
# understanding assert statement
x = int(input('Enter a number greater than 0: '))
assert x>0, "Wrong input entered"
print('U entered: ',x)
```

Output:

```
C:\>python Demo.py
Enter a number greater than 0: 5
U entered: 5
C:\> python Demo.py
Enter a number greater than 0: -2
Traceback (most recent call last):
    File "Demo.py", line 3, in <module>
        assert x>0, "Wrong input entered"
AssertionError: Wrong input entered
```

The 'AssertionError' shown in the above output is called an exception. An exception is an error that occurs during runtime. To avoid such type of exceptions, we can handle them using 'try ... except' statement. After 'try', we use 'assert' and after 'except', we write the exception name to be handled. In the except block, we write the statements that are executed in case of exception. Consider Program 29.

Program

Program 29: A Python program to handle the AssertionError exception that is given by assert statement.

```
# to handle AssertionError raised by assert
x = int(input('Enter a number greater than 0: '))
try:
    assert(x>0)  # exception may occur here
    print('U entered: ',x)
except AssertionError:
    print("Wrong input entered")  # this is executed in case of
    # exception
```

Output:

```
C:\>python Demo.py
Enter a number greater than 0:  -5
Wrong input entered
```

We will discuss exceptions clearly in a later chapter.

The return Statement

A function represents a group of statements to perform a task. The purpose of a function is to perform some task and in many cases a function returns the result. A function starts with the keyword *def*that represents the definition of the function. After 'def', the function should be written. Then we should write variables in the parentheses. For example,

```
def sum(a, b):
    function body
```

After the function name, we should use a colon (:) to separate the name with the body. The body of the statements contains logic to perform the task. For example, to find sum of two numbers, we can write:

```
def sum(a, b):
    print(a+b)
```

A function is executed only when it is called. At the time of calling the sum() function, we should pass values to variables a and b. So, we can call this function as:

```
sum(5, 10)
```

Now, the values 5 and 10 are passed to a and b respectively and the sum() function displays their sum. In Program 30, we will now write a simple function to perform sum of two numbers.

Program

Program 30: A function to find the sum of two numbers.

```
# a function to find sum of two numbers
def sum(a, b):
    print('Sum= ', a+b)

sum(5, 10)  # call sum() and pass 5 and 10
sum(1.5, 2.5)  # call sum() and pass 1.5 and 2.5
```

Output:

```
C:\>python Demo.py
Sum= 15
Sum= 4.0
```

In the above program, the sum() function is not returning the computed result. It is simply displaying the result using print() statement. In some cases, it is highly useful to write the function as if it is returning the result. This is done using 'return' statement. return statement is used inside a function to return some value to the calling place. The syntax of using the return statement is:

```
return expression
```

We will rewrite Program 30 such that the sum() function returns the result with the help of return statement.

Program

Program 31: A Python program to write a function that returns the result of sum of two numbers.

```
# a function to return sum of two numbers
def sum(a, b):
    return a+b  # result is returned from here

# call sum() and pass 5 and 10
# get the returned result into res
res = sum(5, 10)
print('The result is ', res)
```

Output:

```
C:\>python Demo.py
The result is 15
```

When a function does not return anything, then we need not write 'return' statement inside the function.

Once we understand control statements, we would be ready to develop better programs. In the next program, we want to display prime numbers. The general logic for checking whether a number 'n' is prime or not is to divide 'n' with every number starting from 2 to n-1. If 'n' is not divisible by all numbers from 2 to n-1, then it is prime and should be displayed. On the other hand, if 'n' is divisible by any number from 2 to n-1, then we can say it is not prime and hence it should be left. This logic is used in Program 32.