CHAPTER

3 CONTROL STRUCTURES

CHAPTER OUTLINE

control structures are used for non-sequential and repetitive execution of instructions

The functions that we have developed so far had the property that each instruction in a function was executed exactly once. Further, the instructions in these functions were executed in the order in which they appeared in the functions. Such functions are called straight line functions. However, real life problems would usually require non-sequential and repetitive execution of instructions. Python provides various control structures for this purpose. In this chapter, we will study the following control structures with suitable examples: `if`, `for`, and `while`.

3.1 `if` CONDITIONAL STATEMENT

Suppose a teacher grades the students on a scale of 100 marks. To pass the examination, a student must secure pass marks (say, 40). Before announcing the results, the teacher decides to moderate the results by giving maximum of two grace marks. Thus, if the student has scored 38 or 39 marks, he or she would be declared to have scored 40 marks. Let us see how script `moderate` (Fig. 3.1) achieves this.

First, we set `passMarks` equal to `40` in the function `main` (Fig. 3.1). The next statement prompts the user to enter the marks obtained by a student. As `marks` entered by the user is of type `str`, we transform it to an integer quantity `intMarks` using the function `int`. In line 23, we invoke the function `moderate` defined in lines 1–12 with the arguments `intMarks` and `passMarks`. Inside the function `moderate` (line 10), we check whether the value of the input parameter `marks` is less than `passMarks`

```
01  def moderate(marks, passMarks):
02      '''
03      Objective: To moderate result by maximum 1 or 2 marks to
04      achieve passMarks
05      Input Parameters:
06            marks - int
07            passMarks - int
08      Return Value: marks - int
09      '''
10      if marks == passMarks-1 or marks == passMarks-2:
11          marks = passMarks
12      return marks
13
14  def main():
15      '''
16      Objective: To moderate marks if a student just misses pass
        marks
17      Input Parameter: None
18      Return Value: None
19      '''
20      passMarks = 40
21      marks = input('Enter marks: ')
22      intMarks = int(marks)
23      moderatedMarks = moderate(intMarks, passMarks)
24      print('Moderated marks:', moderatedMarks)
25
26  if __name__=='__main__':
27      main()
```

Fig. 3.1 Program to moderate the results by giving maximum of two grace marks (moderate.py)

`marks` to be updated to `passMarks` based on a condition

by one or two using the condition `marks == passMarks-1 or marks == passMarks-2`. The assignment statement in line 11 gets executed only if the condition evaluates to `True`. Next, the function `moderate` returns `marks` (possibly modified) to the `main` function. The value returned by the function `moderate` is assigned to the variable `moderatedMarks` (line 23). Finally, we print `moderatedMarks` in line 24. If we run the script in Fig. 3.1, the system responds by asking the `marks` and displays `moderatedMarks`.

```
>>>
Enter marks: 38
Moderated marks: 40
```

```
>>>
Enter marks: 39
Moderated marks: 40

>>>
Enter marks: 40
Moderated marks: 40
```

In Fig. 3.2, we give a representation of the function `moderate` using a flowchart.

moderate `marks` only if the student is 1 or 2 marks short of `passMarks`

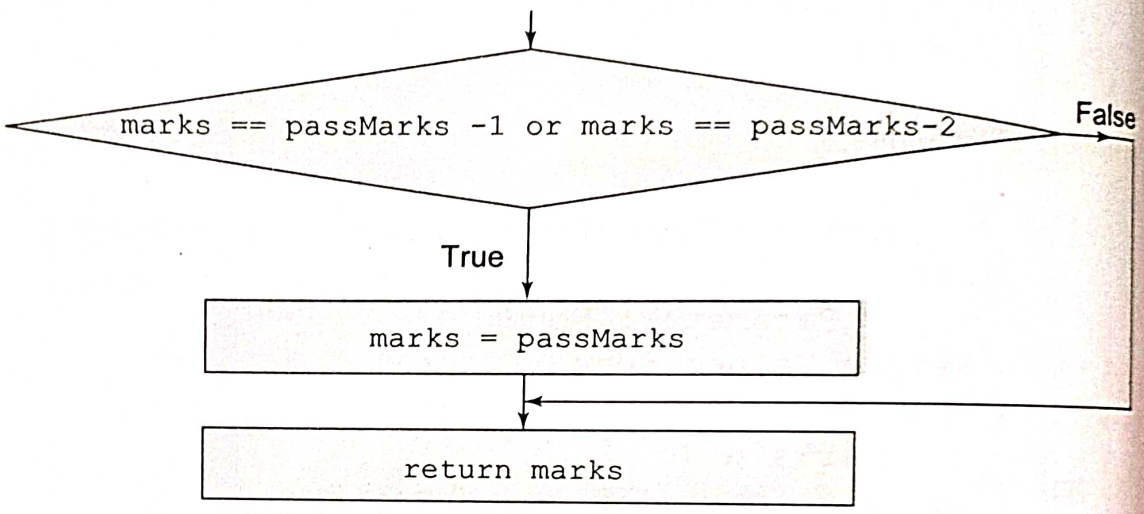


Fig. 3.2 Flow diagram of function `moderate`

▶ General Form of `if` Conditional Statement

The general form of `if` conditional statement is as follows:

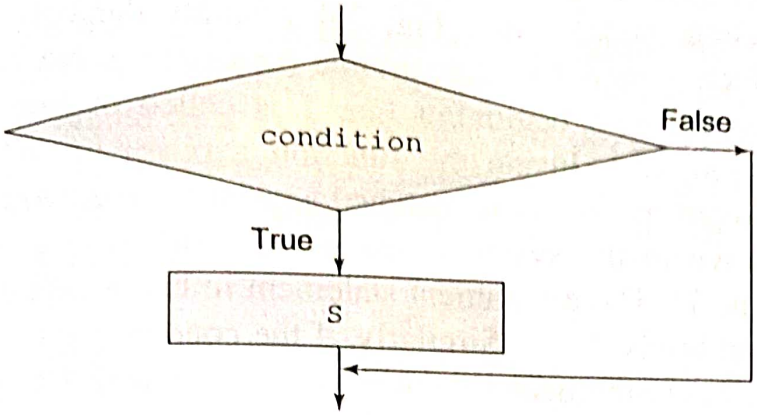
syntax for `if` conditional statement

```
if < condition >:
   < Sequence S of statements to be executed >
```

statements following `if` clause are executed only if the condition in the clause evaluates to `True`

Here, *condition* is a Boolean expression, which is evaluated when the `if` statement is executed. If this condition evaluates to `True`, then the sequence *S* of statements is executed, and the control is transferred to the statement following the `if` statement. However, if the Boolean expression evaluates to `False`, the sequence *S* of statements is ignored, and the control is immediately transferred to the statement following the `if` statement. The flow diagram for execution of the `if` statement has been shown in Fig. 3.3.

Note that *<Sequence S of statements to be executed>* following the colon is indented (i.e., shifted right). Next, suppose we want to restrict the use of a system. To keep the things simple, all the valid users are assigned a common password and password validation is the only task this program performs. We can verify the password entered by a user against the password stored in the system using an `if` statement. If both



flow-diagram of `if` statement

Fig. 3.3 Flow diagram of `if` statement

the passwords match, the program prints a welcome message, else an error message indicating password mismatch is displayed. Let us see how the script in Fig. 3.4 achieves this.

```
01  def authenticateUser(password):
02      '''
03      Objective: To authenticate user and allow access to system
04      Input Parameter: password - string
05      Return Value: message - string
06      '''
07      if password == 'magic':
08          message = ' Login Successful !!\n Welcome to system.'
09      if password != 'magic':
10          message = ' Password mismatch !!\n '
11      return message
12
13  def main():
14      '''
15      Objective: To authenticate user
16      Input Parameter: None
17      Return Value: None
18      '''
19      print(' \t LOGIN SYSTEM ')
20      print('================================')
21      password = input(' Enter Password: ')
22      message = authenticateUser(password)
23      print(message)
24
25  if __name__=='__main__':
26      main()
```

Fig. 3.4 Program to authenticate user and allow access to system `(authenticate.py)`

password validation

The function `main` in Fig. 3.4 prompts the user to enter the password, which is stored in the variable `password`. Next, the function `authenticateUser` beginning line 1 is called in line 22 with the argument `password`. Inside the function `authenticateUser`, the value of this input parameter is matched against the password `'magic'` (already known to the system), using the condition `password == 'magic'` (line 7). The assignment statement in line 8 gets executed only if the condition holds `True`. Similarly, if the condition specified in line 9 holds `True`, the assignment statement in line number 10 gets executed. String value returned by the function `authenticateUser` (line 11) is assigned to the variable `message` in line 22. Finally, we print a message indicating whether the user is authorized to use the system in line 23.

If we run the script in Fig. 3.4, the system responds by asking for the password and displays a suitable message indicating successful or unsuccessful login attempt.

```
        LOGIN SYSTEM
==============================
 Enter Password: hello
 Password mismatch !!

        LOGIN SYSTEM
==============================
 Enter Password: magic
 Login Successful !!
 Welcome to system.
```

In Fig. 3.5, we represent the function `authenticateUser` using a flowchart:

flow chart to validate the password entered by the user

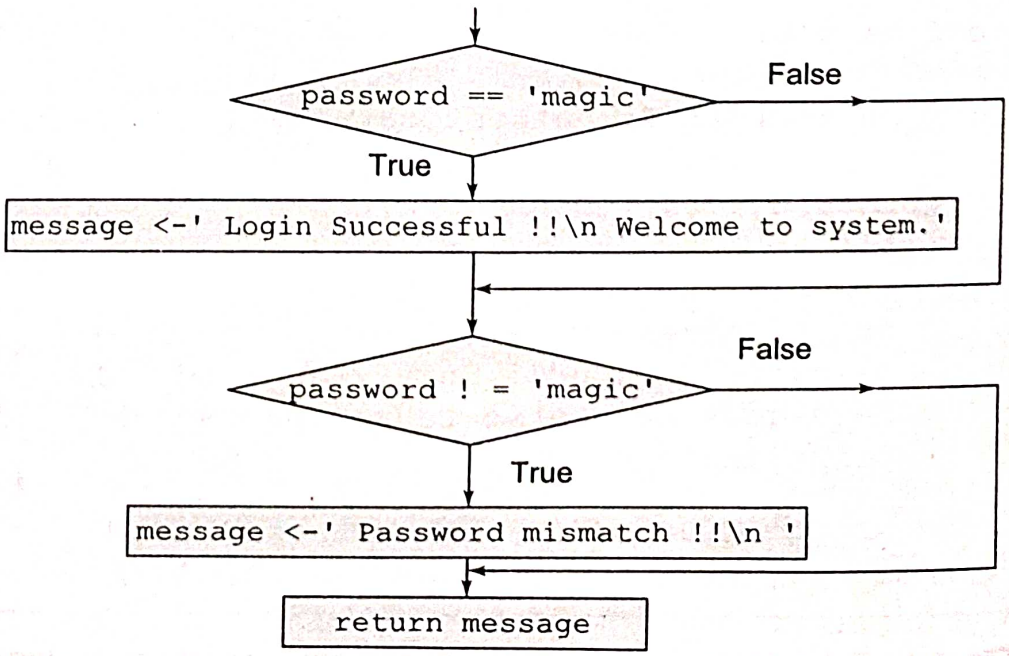


Fig. 3.5 Flow diagram of function `authenticateUser`

You must have noted in the previous program that there were two `if` statements for checking the correctness and incorrectness of the `password` entered by a user. However, it is obvious that if `password` is not correct, it must be incorrect. Indeed, Python provides an `if-else` statement to handle such situations. Let us have a look at the modified script given in Fig. 3.6.

```
01  def authenticateUser(password):
02      '''
03      Objective: To authenticate user and allow access to system
04      Input Parameter: password - string
05      Return Value: message - string
06      '''
07      if password == 'magic':
08          message = ' Login Successful !!\n Welcome to system.'
09      else:
10          message = ' Password mismatch !!\n '
11      return message
12
13  def main():
14      '''
15      Objective: To authenticate user
16      Input Parameter: None
17      Return Value: None
18      '''
19      print(' \t LOGIN SYSTEM ')
20      print('============================')
21      password = input(' Enter Password: ')
22      message = authenticateUser(password)
23      print(message)
24
25  if __name__=='__main__':
26      main()
```

Fig. 3.6 Program to authenticate user and allow access to system `(authenticate.py)`

If the condition in line 7 holds `True`, the statement in line number 8 (body of `if` statement) gets executed. If the condition in line 7 fails to hold, control is transferred to the assignment statement in line 10 (body of the `else` part). In Fig. 3.7, we illustrate the `if-else` statement with the help of a flowchart.

▶ Conditional Expression

Python allows us to write conditional expressions of the form given below:

use of `if-else` conditional statement to validate the password entered by the user

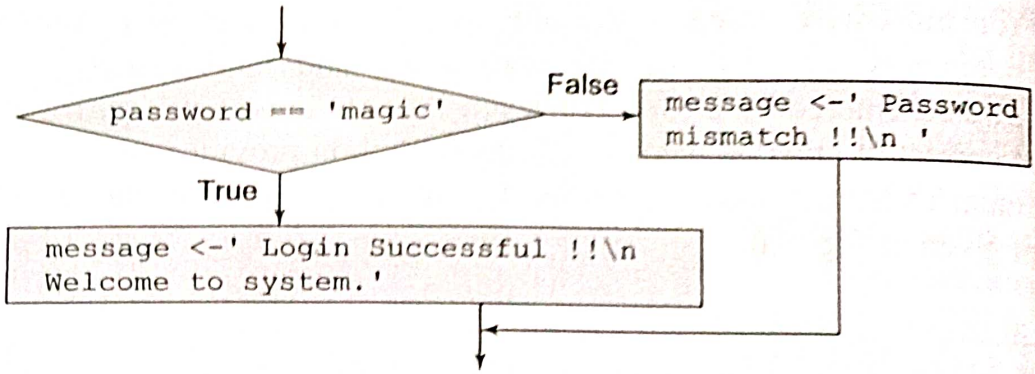


Fig. 3.7 Flow diagram of `if-else` statement in the function `authenticateUser`

```
<expression1> if <condition> else < expression2>
```

If the *condition* holds `True`, the conditional expression yields the value of *expression1*, otherwise, it yields the value of *expression2*. For example, the following piece of code

```
if password == 'magic':
        message = ' Login Successful !!\n '
else:
        message = ' Password mismatch !!\n '
```

use of conditional expression for password validation

may be replaced by

```
message = ' Login Successful !!\n ' if password ==
'magic' else ' Password mismatch !!\n '
```

When there is more than one way of doing a thing, one should prefer the one which enhances the readability of the program. As you might have noted that the use of an `if-else` statement is more readable as compared to a conditional expression. Although a conditional expression happens to be more concise, we discourage the use of conditional expressions for the beginners.

▶ General Form of `if-else` Conditional Statement

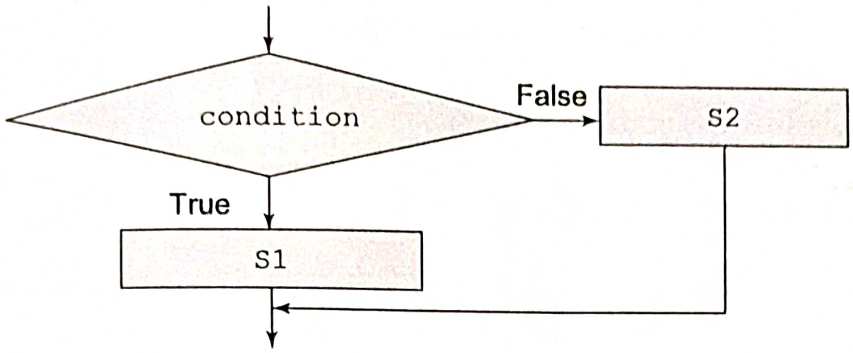
The general forms of `if-else` statement is as follows:

syntax of `if-else` conditional statement

```
if < condition >:
    < Sequence S1 of statements to be executed >
else:
    < Sequence S2 of statements to be executed >
```

Here, *condition* is a Boolean expression. If this condition evaluates to `True`, then the sequence *S1* of statements is executed, else sequence of

statements *S2* gets executed. Subsequently, the control is transferred to the statement following the `if-else` statement. The execution of `if-else` statement is illustrated in Fig. 3.8.



flow diagram of `if-else` conditional statement

Fig. 3.8 Flow diagram of `if-else` structure

In the next problem, we want to assign a grade to a student on the basis of marks obtained as per the criteria mentioned in Table 3.1.

Table 3.1 Criteria for assigning grades

Range	Grade
[90, 100]	A
[70, 89]	B
[50, 69]	C
[40, 49]	D
[0, 39]	F

conversion of grades to marks

The script `grade` (Fig. 3.9) takes marks of a student as input from the user and assigns a grade on the bases of marks obtained using `if-elif-else` statement. In Fig. 3.10, we illustrate the `if-elif-else` statement with the help of a flowchart.

```
01  def assignGrade(marks):
02      '''
03      Objective: To assign grade on the basis of marks obtained
04      Input Parameter: marks - numeric value
05      Return Value: grade - string
06      '''
07      assert marks >= 0 and marks <= 100
08      if marks >= 90:
09          grade = 'A'
```

(*Cont'd*)

Fig. 3.9 (Continued)

```
10      elif marks >= 70:
11          grade = 'B'
12      elif marks >= 50:
13          grade = 'C'
14      elif marks >= 40:
15          grade = 'D'
16      else:
17          grade = 'F'
18      return grade
19
20
21  def main():
22      '''
23      Objective: To assign grade on the basis of input marks
24      Input Parameter: None
25      Return Value: None
26      '''
27      marks = float(input('Enter your marks: '))
28      print('Marks:', marks, '\nGrade:', assignGrade(marks))
29
30  if __name__=='__main__':
31      main()
```

Fig. 3.9 Program to assign grade on the basis of marks obtained (`grade.py`)

conversion of marks to grades

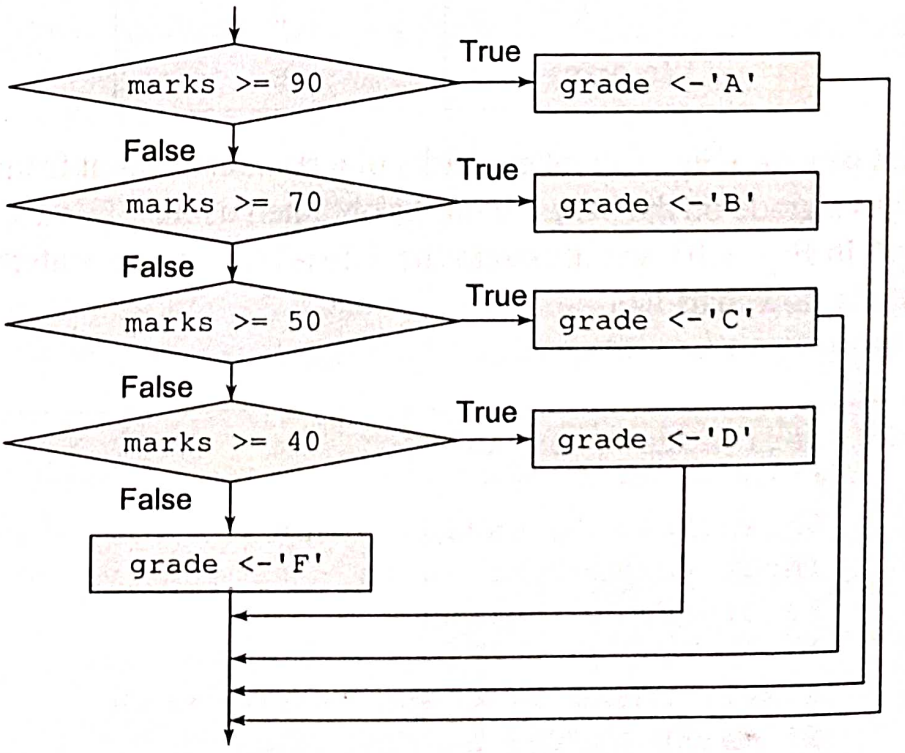


Fig. 3.10 Flow diagram of `if-elif-else` statement in the function `assignGrade`

General Form of `if-elif-else` Conditional Statement

The general form of `if-elif-else` statement is as follows:

```
if < condition1 >:
   < Sequence S1 of statements to be executed >
elif < condition2 >:
     < Sequence S2 of statements to be executed >
elif < condition3 >:
     < Sequence S3 of statements to be executed >
.
.
.
else:
     < Sequence Sn of statements to be executed >
```

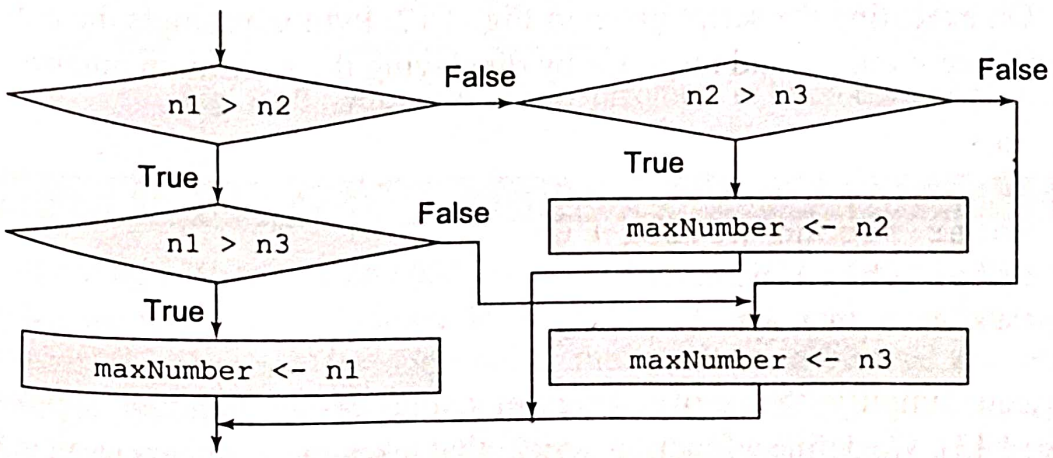
syntax of `if-elif-else` conditional statement

`elif` and `else` clauses are optional in a conditional statement

In the above description, `elif` is an abbreviation used in Python for `else if`. By now, it must be clear that the clauses `elif` and `else` of the `if` statement are optional, and that the sequence of statements defined under a clause is executed in a sequence. As the physical alignment of statements determines which statements form a block of code, one needs to be very careful about indentation while writing Python programs.

Nested `if-elif-else` Conditional Statement

Sometimes, we need a control structure within another control structure. Such a mechanism is called nesting. For example, the function `maximum3` (Fig. 3.12) finds the maximum of three numbers using nested structure. Here, an `if` clause has been used within another `if` clause. First, we test whether `n1` is greater than `n2`. If so, the condition `n1 > n3` is evaluated, and if `True`, `n1` is declared as the maximum number. The other cases are dealt with similarly. This is illustrated with the help of flow diagram in Fig. 3.11.



to find maximum of three numbers

Fig. 3.11 Flow diagram of `if` statement in the function `maximum`

```
01  def maximum3(n1, n2, n3):
02      '''
03      Objective: To find maximum of three numbers
04      Input Parameters: n1, n2, n3 - numeric values
05      Return Value: maxNumber - numeric value
06      '''
07      if n1 > n2:
08          if n1 > n3:
09              maxNumber = n1
10          else:
11              maxNumber = n3
12      elif n2 > n3:
13          maxNumber = n2
14      else:
15          maxNumber = n3
16      return maxNumber
17
18  def main():
19      '''
20      Objective: To find maximum of three numbers provided as input
21      by user
22      Input Parameter: None
23      Return Value: None
24      '''
25      n1 = int(input('Enter first number: '))
26      n2 = int(input('Enter second number: '))
27      n3 = int(input('Enter third number: '))
28      maximum = maximum3(n1, n2, n3)
29      print('Maximum number is', maximum)
30
31  if __name__=='__main__':
32      main()
```

Fig. 3.12 Program to find the maximum of three numbers (`maximum.py`)

On executing the script given in Fig. 3.12, Python prompts the user to enter three numbers and responds by displaying the maximum number.

```
>>>
Enter first number: 78
Enter second number: 65
Enter third number: 89
Maximum number is 89
```

nested function approach to find maximum of three numbers

We can simplify the above program using *nested function* approach (Fig. 3.13). We define a function `max2` that takes two numbers as an input and computes their maximum. Next, we make use of `max2` to define the function `max3` that finds the maximum of three numbers. We say that the

function `max2` is nested within the function `max3`. The functions `max2` and `max3` are known as inner function and outer function, respectively.

```
01  def max3(n1, n2, n3):
02      '''
03      Objective: To find maximum of three numbers
04      Input Parameters: n1, n2, n3 - numeric values
05      Return Value: number - numeric value
06      '''
07      def max2(n1, n2):
08          '''
09          Objective: To find maximum of two numbers
10          Input Parameters: n1, n2 - numeric values
11          Return Value: maximum of n1, n2 - numeric value
12          '''
13          if n1 > n2:
14              return n1
15          else:
16              return n2
17      return max2(max2(n1, n2), n3)
18
19  def main():
20      '''
21      Objective: To find maximum of three numbers provided as input
22      by user
23      Input Parameter: None
24      Return Value: None
25      '''
26      n1 = int(input('Enter first number: '))
27      n2 = int(input('Enter second number: '))
28      n3 = int(input('Enter third number: '))
29      maximum = max3(n1, n2, n3)
30      print('Maximum number is', maximum)
31
32  if __name__=='__main__':
33      main()
```

Fig. 3.13 Program to find maximum of three numbers (`maximum3.py`)

3.2 ITERATION (`for` AND `while` STATEMENTS)

loop: repetitive execution of instructions

Suppose we wish to read and add `100` numbers. Using the method discussed so far, we will have to include `100` statements for reading and an equal number of statements for addition. If the numbers to be read and added were `10,000` instead of `100`, we can well imagine the gigantic appearance of the program that would run into well over a hundred pages. Surely, we cannot think of doing that. The process of repetitive execution of a statement

iteration of a loop

or a sequence of statements is called a loop. Using loops, we need to write only once the sequence of statements to be repeatedly executed. Execution of a sequence of statements in a loop is known as an iteration of the loop.

The `for` statement in Python provides a mechanism to keep count of the number of times a sequence of statements has been executed. Sometimes, we say that the `for` statement loops over a sequence of statements and as such is also called `for` loop. At times, there may be a situation where the number of times, a sequence of statements has to be processed is not known in advance, but depends on the fulfillment of some condition. In such situations, we use a `while` loop.

3.2.1 for Loop

`for` loop: used to execute a sequence of instructions a fixed number of times

The control statement `for` is used when we want to execute a sequence of statements (indented to the right of keyword `for`) a fixed number of times. Suppose, we wish to find the sum of first few (say `n`) positive integers. Further, assume that the user provides the value of `n`. For this purpose, we need a counting mechanism to `count` from `1` to `n` and keep adding the current value of `count` to the old value of `total` (which is initially set to zero). This is achieved using the control statement `for` (Fig. 3.14).

```
01  def summation(n):
02      '''
03      Objective: To find sum of first n positive integers
04      Input Parameter: n - numeric value
05      Return Value: total - numeric value
06      '''
07      total = 0
08      for count in range(1, n + 1):
09          total += count
10      return total
11
12  def main():
13      '''
14      Objective: To find sum of first n positive integers based on user
15      input
16      Input Parameter: None
17      Return Value: None
18      '''
19      n = int(input('Enter number of terms: '))
20      total = summation(n)
21      print('Sum of first', n, 'positive integers: ', total)
22
23  if __name__=='__main__':
24      main()
```

Fig. 3.14 Program to find sum of first `n` positive integers (`sum.py`)

On executing the script (Fig. 3.14), Python prompts the user to enter the value of n and responds with the sum of first n positive integers:

```
Enter number of terms: 5
Sum of first 5 positive integers: 15
```

In the function `summation` (script `sum`, Fig. 3.14), the variable `total` is initialized to zero. Next, `for` loop is executed. The function call `range(1,n + 1)` produces a sequence of numbers from `1` to `n`. This sequence of numbers is used to keep count of the iterations. In general,

`range` function generates a sequence of integers

```
range(start, end, increment)
```

returns an object that produces a sequence of integers from *start* up to *end* (but not including *end*) in steps of *increment*. If the third argument is not specified, it is assumed to be 1. If the first argument is also not specified, it is assumed to be 0. Values of *start*, *end*, and *increment* should be of type integer. Any other type of value will result in error. Next, we give some examples of the use of `range` function:

Function	Sequence of values produced
`range(1,11)`	`1, 2, 3, 4, 5, 6, 7, 8, 9, 10`
`range(11)`	`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10`
`range(1,11,2)`	`1, 3, 5, 7, 9`
`range(30, -4, -3)`	`30, 27, 24, 21, 18, 15, 12, 9, 6, 3, 0, -3`

In the script `sum`, the variable `count` takes a value from the sequence `1, 2, ..., n`, one by one, and the statement

```
total += count
```

is executed for every value of `count`. On completion of the loop, the function `summation` returns the sum of first `n` positive numbers stored in the variable `total` (line 10). The value returned is displayed using `print` function in line 21.

Next, we develop a program to read and add the `marks` entered by the user in `n` subjects, and subsequently use the total marks so obtained to compute the overall `percentage` of marks (Fig. 3.15). The number of subjects `n` is taken as the input from the user. The variable `totalMarks`

```
01  def main():
02      '''
03      Objective: To compute percentage for marks entered in n
        subjects
04      Input Parameter: None
05      Return Value: None
06      '''
07      n = int(input('Number of subjects: '))
08      totalMarks = 0
09      print('Enter marks')
10      for i in range(1, n + 1):
11          marks = float(input('Subject ' + str(i) + ': '))
12          assert marks >= 0 and marks <= 100
13          totalMarks += marks
14      percentage = totalMarks / n
15      print('Percentage is: ', percentage)
16
17  if __name__=='__main__':
18      main()
```

Fig. 3.15 Program to compute percentage (`percentage.py`)

is initialized to zero in line 8 before the `for` loop. The sequence of statements at lines 11, 12, and 13 forms the body of the loop and is executed `n` times, once for each value of `i`. We say that the loop iterates for each value of `i` in the sequence `1, 2, ..., n`. Once the execution of the loop completes, `percentage` is computed.

On executing the script `percentage` (Fig. 3.15), Python prompts the user to enter the number of subjects and the marks, and responds by displaying the percentage.

```
Number of subjects: 5
Enter marks
Subject 1: 75
Subject 2: 80
Subject 3: 85
Subject 4: 90
Subject 5: 95
Percentage is:  85.0
```

▶ General Format of `for` Statement

The general form of `for` structure is as follows:

```
for variable in sequence:
    <block S of statements>
```

for loop syntax

Here, `variable` refers to the control variable used for counting, which is set in the beginning to the first value in a sequence of values (e.g., `range(10)`, `'abcdef'`). Subsequently, in each iteration of the loop, the value of the control `variable` is replaced by the successor value in the sequence. The process of executing the sequence *S* and replacing the current value of control variable by its successor in sequence is repeated until the entire sequence is exhaused.

Next, we wish to develop a function to print a multiplication table for a given number. For example, the multiplication table for 4 should appear as follows:

```
4 *  1 =  4
4 *  2 =  8
4 *  3 = 12
4 *  4 = 16
4 *  5 = 20
4 *  6 = 24
4 *  7 = 28
4 *  8 = 32
4 *  9 = 36
4 * 10 = 40
```

By now you must have noted that Python output is aligned on the left-hand side, for example:

By default, Python aligns the output on LHS

```
>>> print(7)
7
>>> print(100)
100
```

However, the numeric output looks good, if aligned on the right-hand side. For this purpose, we need to specify how many places we would like to reserve for printing a value, for example, the format string `'%5d'` may be used to indicate that at least five positions are to be reserved for an integer value.

formatted output using format string

```
>>> '%5d'% 45
'   45'
>>> print('%5d'% 45)
   45
>>> print('%5d'% 12345)
12345
```

To develop multiplication table for a given number, we would require two parameters: number (say, num) whose multiplication table is to be printed, and the number of multiples (say, nMultiples) of the number. In this exercise, we need to print nMultiples (= 10). The following code segment does this job (Fig. 3.16). In Fig. 3.17, we present a complete script to print the multiplication table of a given number.

```
01  for multiple in range(1, nMultiples + 1):
02          product = num * multiple
03          print(num,'*', '%2'% multiple, '=', '%5d'%product)
```

Fig. 3.16 Loop for printing multiplication table

```
01  def printTable(num, nMultiples = 10):
02      '''
03      Objective: To print multiplication table of a number
04      comprising of first nMultiples
05      Input Parameters:
06            nMultiples: numeric - number of multiples of a number
07            to be printed
08            num: numeric - number whose multiplication table is to
09            be printed
10      Output: Multiplication tables of a number
11      Return Value: None
12      '''
13      for multiple in range(1, nMultiples + 1):
14          product = num * multiple
15          print(num, '*', '%2d' % multiple, '=', '%5d'% product)
16
17.
18  def main():
19      '''
20      Objective: To print multiplication table of a number
21      Input Parameter: None
22      Return Value: None
23      '''
24      num = int(input('Enter the number: '))
25      printTable(num)
26
27  if __name__=='__main__':
28      main()
```

Fig. 3.17 Program to print multiplication table of a number

3.2.2 while Loop

The `while` loop is used for executing a sequence of statements again and again on the basis of some *test condition*. If the *test condition* holds `True`, the body of the loop is executed, otherwise the control moves to the statement immediately following the `while` loop. Suppose, we wish to find the sum of all the numbers entered by a user until a null string (empty string: `''`) is entered as input. As we do not know in advance the count of numbers that the user will enter before entering a null string, we make use of a `while` loop in the script `sumNumbers` (Fig. 3.18). Every time the user enters a string, its integer value is added to the old value of `total` (initialized to zero before the beginning of the loop). The process continues until the user enters a null string as input. On encountering a null string, the *test condition* in line 10 becomes `False`, the `while` loop terminates, and the control moves to line 13, where we display the value of `total`.

`while` loop: to repeat execution of an instruction sequence as long as a condition holds

empty string is also known as null string

```
01  def main():
02      '''
03      Objective: To compute sum of numbers entered by user until
04      user provides with null string as the input
05      Input Parameter: None
06      Return Value: None
07      '''
08      total = 0
09      number = input('Enter a number: ')
10      while number != '':
11          total += int(number)
12          number = input('Enter a number: ')
13      print('Sum of all input numbers is', total)
14
15  if __name__=='__main__':
16      main()
```

Fig. 3.18 Program to compute sum of numbers (`sumNumbers.py`)

On executing the script `sumNumbers` (Fig. 3.18), Python prompts for numbers until the user enters null string as an input.

```
>>>
Enter a number: 2
Enter a number: 18
Enter a number: 15
Enter a number: 15
Enter a number:
Sum of all input numbers is 50
```

Next, we show the working of the `while` loop in the context of the foregoing example (Fig. 3.19).

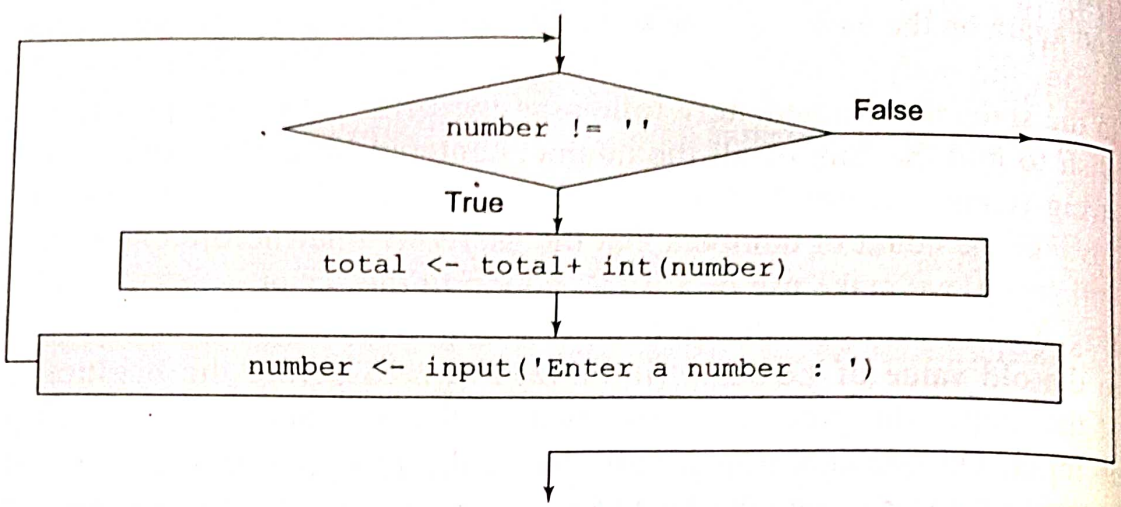


Fig 3.19 Flow diagram of `if` statement in script `sumNumbers`

▶ General Format of `while` Statement

The general form of `while` statement is as follows:

syntax for `while` loop

```
while <test condition>:
      <Sequence of statements S>
```

Here, *test condition* is a Boolean expression, which is evaluated at the beginning of the loop. If the *test condition* evaluates to `True`, the sequence *S* of statements is executed, and the control moves to the evaluation of *test condition* once again. The process is repeated until *test condition* evaluates to `False`. The execution of the `while` loop is illustrated in Fig. 3.20.

flow diagram of `while` loop

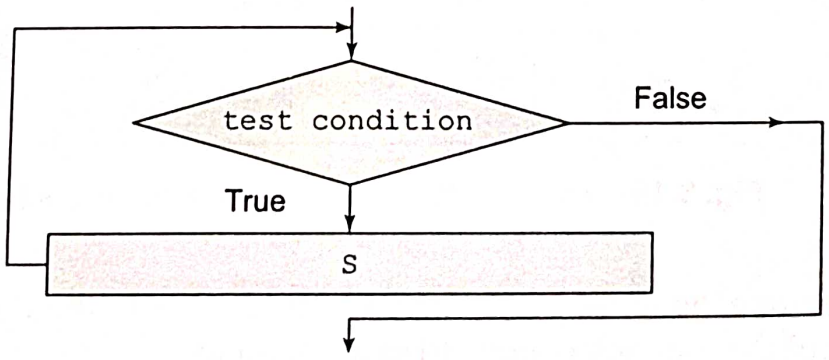


Fig. 3.20 Flow diagram of `while` structure

▶ Infinite Loops

infinite loop: loop with a condition which always evaluates to `True`

Sometimes we want a `while` loop to continue indefinitely until an enabling event takes place, for example, the screen of a laptop may remain off until a key is pressed. For this purpose, we make use of a `while` loop based on a condition that always evaluates to `True`. Such a loop is known as an infinite loop, for example:

```
import time
while True:
  try:
     print('Loop processing....')
     print('Use ctrl+c to break')
     time.sleep(1)
  except KeyboardInterrupt:
     print('User interrupted the loop... exiting...')
     break
```

in the absence of a keyboard interrupt, the `while` loop will execute infinitely.

3.2.3 `while` Statement vs. `for` Statement

Having learned the use of `while` statement, let us use a `while` statement to rewrite the following piece of code:

```
for count in range(1, n + 1):
   total += count
```

The revised code is given below:

```
count = 1
while count < n+1:
   total += count
   count += 1
```

not a wise idea: rewriting `for` loop using `while`

Note that for computing the sum of first few natural numbers, the use of `for` loop is more elegant and easy as compared to the `while` loop. Out of several control structures which may be used at a place, it is the discretion of the programmer to choose the simple and elegant one.

3.2.4 Example: To Print Some Pictures

Next, we develop a program that prints a right triangle or an inverted triangle (shown in Fig. 3.21) depending on user's choice. Further, the number of rows in the triangle is also to be taken as input from the user. Let us examine the script `triangle` (Fig. 3.22) that serves this purpose. In this program, the user needs to enter his/her choice of figure `1` or `2` depending on whether he or she wants to print a right triangle or an inverted triangle. The assertion in line 11 is used to check whether the input choice

```
*
**
***
****
*****
```
(a)

```
*********
 *******
  *****
   ***
    *
```
(b)

Fig. 3.21 (a) Right triangle and (b) inverted triangle

```
01  def main():
02      '''
03      Objective: To print right triangle or inverted triangle
04      depending on user's choice
05      Input Parameter: None
06      Return Value: None
07      '''
08      choice = int(input('Enter 1 for right triangle.\n'+\
09                         'Enter 2 for inverted triangle.\n'))
10
11      assert choice == 1 or choice == 2
12      nRows = int(input('Enter no. of rows: '))
13
14      if choice == 1:
15          rightTriangle(nRows)
16      else:
17          invertedTriangle(nRows)
18
19  if __name__=='__main__':
20      main()
```

Fig 3.22 `main` function to print right triangle and inverted triangle (`triangle.py`)

is a valid choice, i.e., 1 or 2. Subsequently in line 12, the number of rows in the figure is taken as input. Next, we develop the functions to print right triangle and inverted triangle. To print a right triangle having a number of rows (equal to `nRows`), we have to generate as many rows of output. Outline of the Python code for this purpose is given below:

```
for i in range(1, nRows + 1):
    {Generate one row of output for right triangle}
```

We note that in the first row, one `'*'` is to be output, in the second row two `'*'`s are to be output, and so on. So, in general, the `i`th row will have `i` `'*'`s. With these remarks, we modify the outline of the code given above:

`for` loop for printing a right triangle

```
for i in range(1, nRows + 1):
    print('*' * i)
```

Similarly, to print an inverted triangle comprising the number of rows (equal to `nRows`), we have to generate as many rows of output. The Python code may be outlined as follows:

```
for i in range(0, nRows):
    {Generate one row of output for inverted
    triangle}
```

Let `nSpaces` denotes the number of leading spaces in a row. We note that there are no leading spaces in the first row. So, we set

```
nSpaces = 0
```

For every subsequent row, the number of leading spaces `nSpaces` increases by one in each row. Next, let `nStars` denote the number of stars to be printed in a row. We note that in the first row, the number of stars to be printed is `2 * nRows - 1`. So, we set

```
nStars = 2 * nRows - 1
```

We also note that the number of stars to be printed decreases by two for every subsequent row. With these remarks, we modify the outline of the above piece of code as follows:

```
nSpaces = 0
nStars = 2 * nRows - 1
for i in range(1, nRows+1):
    print(' ' * nSpaces + '*' * nStars)
    nStars -= 2
    nSpaces += 1
```

`for` loop for printing inverted triangle

In light of the above discussion, we give the complete script in Fig. 3.23:

```
01  def rightTriangle(nRows):
02      '''
03      Objective: To print right triangle
04      Input Parameter: nRows - integer value
05      Return Value: None
06      '''
07      for i in range(1, nRows + 1):
08          print('*' * i)
09
10  def invertedTriangle(nRows):
11      '''
12      Objective: To print inverted triangle
13      Input Parameter: nRows - integer value
14      Return Value: None
15      '''
16      nSpaces = 0
17      nStars = 2 * nRows - 1
18      for i in range(1, nRows+1):
19          print(' ' * nSpaces + '*' * nStars)
20          nStars -= 2
21          nSpaces += 1
22
23
```

(*Cont'd*)

Fig. 3.23 (Continued)

```
24  def main():
25      '''
26      Objective: To print right triangle or inverted triangle
27      depending on user's choice
28      Input Parameter: None
29      Return Value: None
30      '''
31      choice = int(input('Enter 1 for right triangle.\n'+\
32                         'Enter 2 for inverted triangle.\n'))
33
34      assert choice == 1 or choice == 2
35      nRows = int(input('Enter no. of rows: '))
36
37      if choice == 1:
38          rightTriangle(nRows)
39      else:
40          invertedTriangle(nRows)
41
42  if __name__=='__main__':
43      main()
```

Fig. 3.23 Program to print right triangle and inverted triangle (`triangle.py`)

In the above script, the figures right triangle and inverted triangle comprise `'*'` only. However, we may make the above program more general by printing the figures made using a character provided by the user. For this purpose, the function `rightTriangle` may be modified as follows (Fig. 3.24):

```
01  def rightTriangle(nRows, char):
02      '''
03      Objective: To print right triangle
04      Input Parameters:
05          nRows - integer value
06          char - character to be used for printing figure
07      Return Value: None
08      '''
09      for i in range(1, nRows + 1):
10          print(char * i)
```

Fig. 3.24 Function `rightTriangle`

The reader is encouraged to modify the `invertedTriangle` and `main` functions in a similar manner and experiment with different figures.

3.2.5 Nested Loops

Suppose we wish to develop a function to print multiplication tables, one for each of the first few (say 10), natural numbers. Further, in the table, multiples of a number are to be arranged vertically in a column. The output should appear as shown in Fig. 3.25.

nested loop: a loop inside another loop

nesting may continue up to any level

```
1 * 1 =  1  2 * 1 =  2  3 * 1 =  3 : 10 * 1 =  10
1 * 2 =  2  2 * 2 =  4  3 * 2 =  6 : 10 * 2 =  20
1 * 3 =  3  2 * 3 =  6  3 * 3 =  9 : 10 * 3 =  30
1 * 4 =  4  2 * 4 =  8  3 * 4 = 12 : 10 * 4 =  40
1 * 5 =  5  2 * 5 = 10  3 * 5 = 15 : 10 * 5 =  50
1 * 6 =  6  2 * 6 = 12  3 * 6 = 18 : 10 * 6 =  60
1 * 7 =  7  2 * 7 = 14  3 * 7 = 21 : 10 * 7 =  70
1 * 8 =  8  2 * 8 = 16  3 * 8 = 24 : 10 * 8 =  80
1 * 9 =  9  2 * 9 = 18  3 * 9 = 27 : 10 * 9 =  90
1 * 10 = 10  2 * 10 = 20  3 * 10 = 30 : 10 * 10 = 100
```

Fig. 3.25 Multiplication table

In order that the output of the program appears nicely, we would like that each number be printed using at least the number of positions specified in the format string. This may be done as follows (5 digits are reserved for each number which is left padded with spaces):

```
>>> for i in range(1,4):
        print('{0: >5}'.format(i*9))
    9
   18
   27
```

by default, `print` function inserts new line after printing a line

Also, we would like that the output of several calls to `print` function may be printed on the same line. For this purpose, we we need to use empty string (`''`) in place of the default newline. This can be achieved using keyword argument `end`. For example,

```
>>> for i in range(1,4):
        print('{0: >5}'.format(i*9), end = '')
    9   18   27
```

using empty string as the terminator in `print` function by changing the default value of keyword argument `end`

Indeed, the code segment

```
>>> for i in range(1,4):
        print('{0: >5}'.format(i*9))
```

is equivalent to:

```
>>> for i in range(1,4):
        print('{0: >5}'.format(i*9), end = '\n')
```

To develop such a function, we would require two parameters: the number of tables, (say, `nTables`) to be printed, and the number of multiples (say, `nMultiples`) of each number to be printed. The skeleton of a function for this purpose is given in Fig. 3.26.

```
01  def printTable(nMultiples = 10,  nTables = 10):
02      '''
03      Objective: To print multiplication table of numbers in range
04      [1,nTables], comprising of first nMultiples
05      Input Parameters:
06          nMultiples: numeric - number of multiples of a number
07                      to be printed
08          nTables: numeric - number of tables to be printed
09      Output: Multiplication tables of numbers, beginning 1 ending
10              nTables
11      Return Value: None
12      '''
```

Fig. 3.26 Skeleton of function `printTable`

To print the multiplication table, we need to print `nMultiples` (=10) rows, one for each `multiple` of the number `num`. Skeleton of the code segment for printing multiplication table is given below:

```
for multiple in range(1, nMultiples + 1):
    # Print a row of multiples of each number num
```

Next, to print one row of a multiple of all numbers i.e. multiples of `num` in the range `(1, nTables+1)`, we may iterate over `num` using a `for` loop as follows:

`for` loop to print one row of multiples of numbers

```
for num in range(1, nTables + 1):
    print('{0: >2}'.format(num),'*',\
          '{0: >2}'.format(multiple),'=', \
          '{0:   >3}'.format(num*multiple),'\t',
          end = '')
print()
```

Note the use of the keyword argument `end` in the `print` function under the `for` loop, as we want the output of the entire loop to appear on the same line. Finally, when the `for` loop completes, invoking the `print`

function moves the control to the next line for printing the next row of the multiplication table. The character backslash (\) used at the end of second and third line is a line continuation character that can be used for wrapping long lines i.e. if we wish to extend a statement over multiple lines. The complete script to print multiplication table is shown in Fig. 3.27.

line continuation character backslash(\)

```
01  def printTable(nTables = 10, nMultiples = 10):
02      '''
03      Objective: To print multiplication table of numbers in range
04      [1,nTables], comprising of first nMultiples
05      Input Parameters:
06          nTables: numeric - number of tables to be printed
07          nMultiples: numeric - number of multiples of a number
08                      to be printed
09      Output: Multiplication tables of numbers, beginning 1 ending
10              nTables
11      Return Value: None
12      '''
13      for multiple in range(1, nMultiples + 1):
14          # Print a row of multiples of each number num
15          for num in range(1, nTables + 1):
16              print('{0: >2}'.format(num),'*',\
17                    '{0: >2}'.format(multiple),'=', \
18                    '{0: >3}'.format(num*multiple),'\t', end = '')
19          print()
20
21  def main():
22      '''
23      Objective: To display table of numbers in range  [1, nTables]
24      comprising of first 10 multiples
25      Input Parameter: None
26      Return Value: None
27      '''
28      nTables = int(input('Enter number of multiplication
        tables: '))
29      printTable(nTables)
30
31  if __name__=='__main__':
32      main()
```

Fig. 3.27 Program to print multiplication table (`table.py`)

The control structures `for` and `while` may be nested up to any level as required. For example, see Fig. 3.28.

```
for(...):
    #Sequence of statements S1
    for(...):
        #Sequence of statements S2
    #Sequence of statements S3
```

```
while(...):
    #Sequence of statements S1
    while(...):
        #Sequence of statements S2
    #Sequence of statements S3
```

```
for(...):
    #Sequence of statements S1
    while(...):
        #Sequence of statements S2
    #Sequence of statements S3
```

```
while(...):
    #Sequence of statements S1
    for(...):
        #Sequence of statements S2
    #Sequence of statements S3
```

```
for(...):
    #Sequence of statements S1
    for(...):
        #Sequence of statements S2
        while(...):
            #Sequence of statements
            S3
        #Sequence of statements S4
    #Sequence of statements S5
```

```
while(...):
    #Sequence of statements S1
    for(...):
        #Sequence of statements S2
        while(...):
            #Sequence of statements
            S3
        #Sequence of statements S4
    #Sequence of statements S5
```

Fig. 3.28 Examples of nested control structures

3.2.6 break, continue, and pass Statements

use break statement to exit the loop

use continue statement to transfer the control to next iteration of the loop

Sometimes, we need to alter the normal flow of a loop in response to the occurrence of an event. In such a situation, we may either want to exit the loop or continue with the next iteration of the loop skipping the remaining statements in the loop. The `break` statement enables us to exit the loop and transfer the control to the statement following the body of the loop. The `continue` statement is used to transfer the control to next iteration of the loop. When the `continue` statement is executed, the code that occurs in the body of the loop after the `continue` statement is skipped.

To illustrate the use of `break` statement, we examine the function `printSquares` (Fig. 3.29) intended to print squares of the integers entered by the user until a null string is encountered. This function uses a `while` loop. The condition in the `while` loop has been set as `True`. If the user enters a null string, the control exits the loop and moves to the statement following the `while` loop, i.e. line 14. Thus, the function

```
01  def printSquares():
02      '''
03      Objective: To print squares of positive numbers entered by
04      the user. The program terminates if user enters null string.
05      Input Parameter: None
06      Return Value: None
07      '''
08      while True:
09          numStrng = input('Enter an integer, to end press Enter: ')
10          if numStrng == '':
11              break
12          number = int(numStrng)
13          print(number, '^ 2 =', number ** 2)
14      print('End of input!!')
```

Fig. 3.29 Function to print squares of positive numbers (`square.py`)

terminates with the message `'End of input!!'`. In the other case, when the user input is not null, the input string is converted to an integer (line 12) and the square of the integer so obtained is printed (line 13).

In the script `percentage1` (Fig. 3.30), we wish to compute the overall percentage of marks obtained by a student. As the number of subjects on which examination was conducted is not known beforehand,

```
01  def main():
02      '''
03      Objective: To display percentage of marks scored by the
        student
04      Input Parameter: None
05      Return Value: None
06      '''
07      totalMarks = 0
08      nSubjects = 0
09      while True:
10          marks = input('Marks for subject ' + str(nSubjects + 1)
            + ': ')
11          if marks == '':   # End of input
12              break
13          marks = float(marks)
14          if marks < 0 or marks > 100:
15              print('INVALID MARKS !! ')
16              continue      # Marks to be entered again
17          nSubjects = nSubjects + 1
```

(Cont'd)

Fig. 3.30 (Continued)

```
18              totalMarks += marks
19          percentage = totalMarks / nSubjects
20          print('Total marks: ', int(totalMarks))
21          print('Number of Subjects: ', nSubjects)
22          print('Percentage: ', round(percentage,2))
23
24      if __name__=='__main__':
25          main()
```

Fig. 3.30 Program to print percentage and total marks (`percentage1.py`)

we use a `while` loop. When the `while` loop is executed, the user is prompted to enter the marks obtained in different subjects. If the user responds with a null string, the `break` statement is executed which results in termination of the loop, and the control moves to line 19 for computation of `percentage`. However, if `marks` entered by the user are outside the range `[0,100]`, the user is prompted again to enter marks. Thus in the case of invalid marks, the use of `continue` statement makes it possible to skip the statements (lines 17 and 18) that appear after the `continue` statement in the `while` loop and continue with the next iteration of the loop for taking the next input from the user. Sample output on executing the script `percentage` is given below:

```
Marks for subject 1: 60
Marks for subject 2: 80
Marks for subject 3: 280
INVALID MARKS !!
Marks for subject 3: 70
Marks for subject 4: 800
INVALID MARKS !!
Marks for subject 4: 80
Marks for subject 5:
Total marks:  290
Number of Subjects: 4
Percentage: 72.5
```

Pass statement

`pass` statement: execute no code

Sometimes we may want to leave out the details of the computation in a function body, to be filled in at a later point in time. The `pass` statement lets the program go through this piece of code without executing any code. It is just like a null operation. Often `pass` statement is used as a reminder for some code, to be filled in later. For example, let us think of a merchant wanting to sell clothes, who is thinking of allowing some discounts, but not as of now. So, as of now, he does not want his IT team to develop the code for discounting. In the script `sellingPrice`, we develop the function `sellingPrice`

(Fig. 3.31) that invokes the function `discount`. As the code for the function `discount` just comprises a `pass` statement, it produces `None` as the return value. Accordingly, the function `sellingPrice` ignores the value `None` returned by the function `discount` (lines 16–17) and returns `price` (that was passed on to it as an argument) as the selling price.

```
01  def discount(price):
02      '''
03      Objective: To compute discount
04      Input Parameter: price - numeric value
05      Return Value: None
06      '''
07      pass
08
09  def sellingPrice(price):
10      '''
11      Objective: To compute selling price
12      Input Parameter: price - numeric value
13      Return Value: numeric value
14      '''
15      dicountedPrice = discount(price)
16      if dicountedPrice == None:
17          return price
18      else:
19          return dicountedPrice
20
21  def main():
22      '''
23      Objective: To compute selling price
24      Input Parameter: None
25      Return Value: None
26      '''
27      price = float(input('Enter price: '))
28      print('Selling Price is', sellingPrice(price))
29
30  if __name__=='__main__':
31      main()
```

Fig. 3.31 Program to compute selling price (`sellingPrice.py`)

3.2.7 Example: To Compute sin(x)

The value of sin (x) may be computed as the sum of the following series:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \cdots$$

In the above series, the first term is `x/1`. The second term can be computed by multiplying the first term by $-x^2/(2*3)$. Similarly, the third term in the series can be computed by multiplying the second term by $-x^2/(4*5)$, and so on. In general, we can obtain a new term of the series by multiplying the previous term by $-x^2$ and dividing this product by the product of the next two terms in the sequence 1, 2, 3, 4, 5, 6, ... Thus, we can write first few terms of the series as follows:

```
x/1
x/1*(-x^2)/(2*3)  = -x^3/3!
-x^3/3!*(-x^2)/(4*5)  = x^5/5!
```

To code the above idea, we set `multBy` equal to $-x^2$ and initialize `nxtInSeq` (used to compute `divBy`) equal to 2. We compute `divBy = nxtInSeq*(nxtInSeq+1)`. Every time we compute a new term, we increment the value of `nxtInSeq` by 2. Table 3.1 illustrates these computations:

Table 3.1 Sine terms computations

term	multBy	nxtInSeq	divBy	newTerm
x/1	$-x^2$	2	2*3	$-x^3/3!$
$-x^3/3!$	$-x^2$	4	4*5	$x^5/5!$
$x^5/5!$	$-x^2$	6	6*7	$-x^7/7!$

Since the given series is an infinite one, and we can do only finite computations on a computer, we need to decide when to stop. We keep on adding more terms until the absolute value of `term` becomes smaller than a predefined value, say, `epsilon`. In Fig. 3.32, we present the complete function `mySine` that computes `sine(x)` as sum of the above series.

```
01  def mySine(x):
02      '''
03      Objective: To find sum of sine series until the absolute value
04      of newTerm becomes smaller than epsilon
05      Input Parameter: x - numeric value in radians
06      Return Value: total - numeric value
07      '''
08      epsilon = 0.00001
09      multBy = -x**2
10      term = x
11      total = x
12      nxtInSeq  = 2.0
```

(Cont'd)

Fig. 3.32 (Continued)

```
13
14      while abs(term) > epsilon:
15          divBy = nxtInSeq*(nxtInSeq+1)
16          term = term * multBy / divBy
17          total += term
18          nxtInSeq += 2
19      return total
```

Fig. 3.32 Function `mySine` (`sine.py`)

3.2.8 `else` Statement

The `else` clause is useful in such situations where we may want to perform a task only on successful execution of a `for` loop or a `while` loop. The statements specified in `else` clause are executed on normal termination of the loop. However, if the loop is terminated forcibly using `break` statement, then the `else` clause is skipped. We demonstrate the use of the `else` clause for testing whether a given number is prime. Recall that a number is said to be prime if and only if it has no divisor other than one and itself. Given a number `n`, we need to check for each number in the `range(2, n)` whether it is a divisor of `n`. If the entire sequence is exhausted and no divisor of `n` is found, it is a prime number. In the function `prime` (Fig. 3.33) if `n` is `1`, `flag` is set equal to `False` indicating that the number `n` is not prime. Inside the `for` loop, if we find a divisor `i` of `n` in the `range(2, n)`, the condition `n % i == 0` becomes `True`, indicating that the number `n` is not a prime number. So we set `flag` equal to `False` and exit the `for` loop. However, if the `for` loop executes smoothly (not on the execution of `break` statement), `flag` is set equal to `True` indicating that the number `n` is prime. Finally, `flag` is returned as the value of the function.

`else` clause: to execute a task only on successful completion of the loop

```
01  def prime(n):
02      '''
03      Objective: To check if a number is prime or not
04      Input Parameter: n  - numeric value
05      Return Value: message - boolean value
06      '''
07      if n  == 1: # 1 is not prime
08          return False
```

(*Cont'd*)

Fig. 3.33 (Continued)

```
09          for i in range(2, n ):
10              if n  % i == 0:
11                  flag = False # n  is not prime
12                  break
13          else:
14              flag = True # n  is  prime
15          return flag
16
17      def main():
18          '''
19          Objective: To check if the number entered by the user is a
20          prime number
21          Input Parameter: None
22          Return Value: None
23          '''
24          n  = int(input('Enter number: '))
25          result = prime(n )
26          if result == True:
27              print(n, 'is a prime number')
28          else:
29              print(n, 'is not a prime number')
30
31      if __name__=='__main__':
32          main()
```

Fig. 3.33 Program to check if a number is prime number or not (`prime.py`)

SUMMARY

1. Control statements (also called control structures) are used to control the flow of program execution by allowing non-sequential or repetitive execution of instructions. Python supports `if`, `for`, and `while` control structures. In a control statement, the Python code following the colon (`:`) is indented.
2. `if` statement allows non-sequential execution depending upon whether the *condition* is satisfied.

   The general form of `if` conditional statement is as follows:

   ```
   if < condition >:
      < Sequence S of statements to be executed >
   ```

   Here, *condition* is a Boolean expression which is evaluated at the beginning of the `if` statement. If *condition* evaluates to `True`, then the sequence *S* of statements is

executed, and the control is transferred to the statement following `if` statement. However, if *condition* evaluates to `False`, then the sequence *S* of statements is ignored, and the control is immediately transferred to the statement following `if` statement.

The general form of `if-else` statement is as follows:

```
if < condition >:
     < Sequence S1 of statements to be executed >
else:
     < Sequence S2 of statements to be executed >
```

Here, *condition* is a Boolean expression. If *condition* evaluates to `True`, then the sequence *S1* of statements gets executed, otherwise, the sequence *S2* of statements gets executed.

The general form of `if-elif-else` statement is as follows:

```
if < condition1 >:
     < Sequence S1 of statements to be executed >
elif < condition2 >:
     < Sequence S2 of statements to be executed >
elif < condition3 >:
     < Sequence S3 of statements to be executed >
.
.
.
else:
     < Sequence Sn of statements to be executed >
```

The clauses `elif` and `else` of `if` control structure are optional.

3. When a control structure is specified within another control structure, it is called nesting of control structures.
4. The process of repetitive execution of a statement or a sequence of statements is called a loop. Execution of a sequence of statements in a loop is known as an iteration of the loop.
5. The control statement `for` is used when we want to execute a sequence of statements a fixed number of times. The general form of `for` statement is as follows:

```
for variable in sequence:
    {Sequence S of statements}
```

Here, *variable* refers to the control variable. The sequence *S* of statements is executed for each value in *sequence*.

6. The function call `range(1,n + 1)` generates a sequence of numbers from `1` to `n`. In general, `range(start, end, increment)` produces a sequence of numbers from *start* up to *end* (but not including *end*) in steps of *increment*. If third argument is not specified, *increment* is assumed to be 1.
7. A `while` loop is used for iteratively executing a sequence of statements again and again on the basis of a *test-condition*. The general form of a `while` loop is as follows:

```
while <test-condition>:
        <Sequence S of statements>
```

Here, *test-condition* is a Boolean expression which is evaluated at the beginning of the loop. If the *test-condition* evaluates to `True`, the control flows through the *Sequence S of statements* (i.e., the body of the loop), otherwise the control moves to the statement immediately following the `while` loop. On execution of the body of the loop, the *test-condition* is evaluated again, and the process of evaluating the *test-condition* and executing the body of the loop is continued until the *test-condition* evaluates to `False`.

8. The `break` statement is used for exiting from the loop to the statement following the body of the loop.
9. The `continue` statement is used to transfer the control to next iteration of the loop without executing rest of the body of loop.
10. The `pass` statement lets the program go through this piece of code without performing any action.
11. The `else` clause can be used with `for` and `while` loop. Statements specified in `else` clause will be executed on smooth termination of the loop. However, if the loop is terminated using `break` statement, then the `else` clause is skipped.

EXERCISES

1. Write an assignment statement using a single conditional expression for the following `if-else` code:

```
if marks >=70:
    remarks = 'good'
else:
    remarks = 'Average'
```

2. Study the program segments given below. In each case, give the output produced, if any.

(a)
```
total = 0
count = 20
while count > 5:
    total += count
    count -= 1
print(total)
```

(b)
```
total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, i+1):
        total += 1
print(total)
```

(c)
```
total = 0
N = 10
for i in range(1, N+1):
    for j in range(1, N+1):
        total += 1
print(total)
```

(d)
```
total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, i+1):
        total += 1
    total -=1
print(total)
```

(e)
```
total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, N+1):
        total += i
print(total)
```

(f)
```
total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, i+1):
        total += j
print(total)
```

(g)
```
total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, N+1):
        total += i + j
print(total)
```

(h)
```
total = 0
N = 5
for i in range(1, N+1):
    for j in range(1, i+1):
        for k in range(1, j+1):
            total += 1
print(total)
```

(i)
```
number = 72958476
a, b = 0, 0
while (number > 0):
```

```
    digit = number % 10
    if (digit % 2!= 0):
        a += digit
    else:
        b += digit
    number /= 10
print(a, b)
```

3. Write a function to determine whether a given natural number is a perfect number. A natural number is said to be a perfect number if it is the sum of its divisors. For example, 6 is a perfect number because 6=1+2+3, but 15 is not a perfect number because 15≠1+3+5.
4. Write a function that takes two numbers as input parameters and returns their least common multiple.
5. Write a function that takes two numbers as input parameters and returns their greatest common divisor.
6. Write a function that accepts as an input parameter the number of rows to be printed and prints a figure like:

(a)

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

(b)

```
      1
    2 1 2
  3 2 1 2 3
4 3 2 1 2 3 4
```

(c)

```
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```

(d)

```
1
2   2
3   3   3
4   4   4   4
5   5   5   5   5
```

(e)

```
1 2 3 4 5
  2 3 4 5
    3 4 5
      4 5
        5
```

(f)

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

(g)

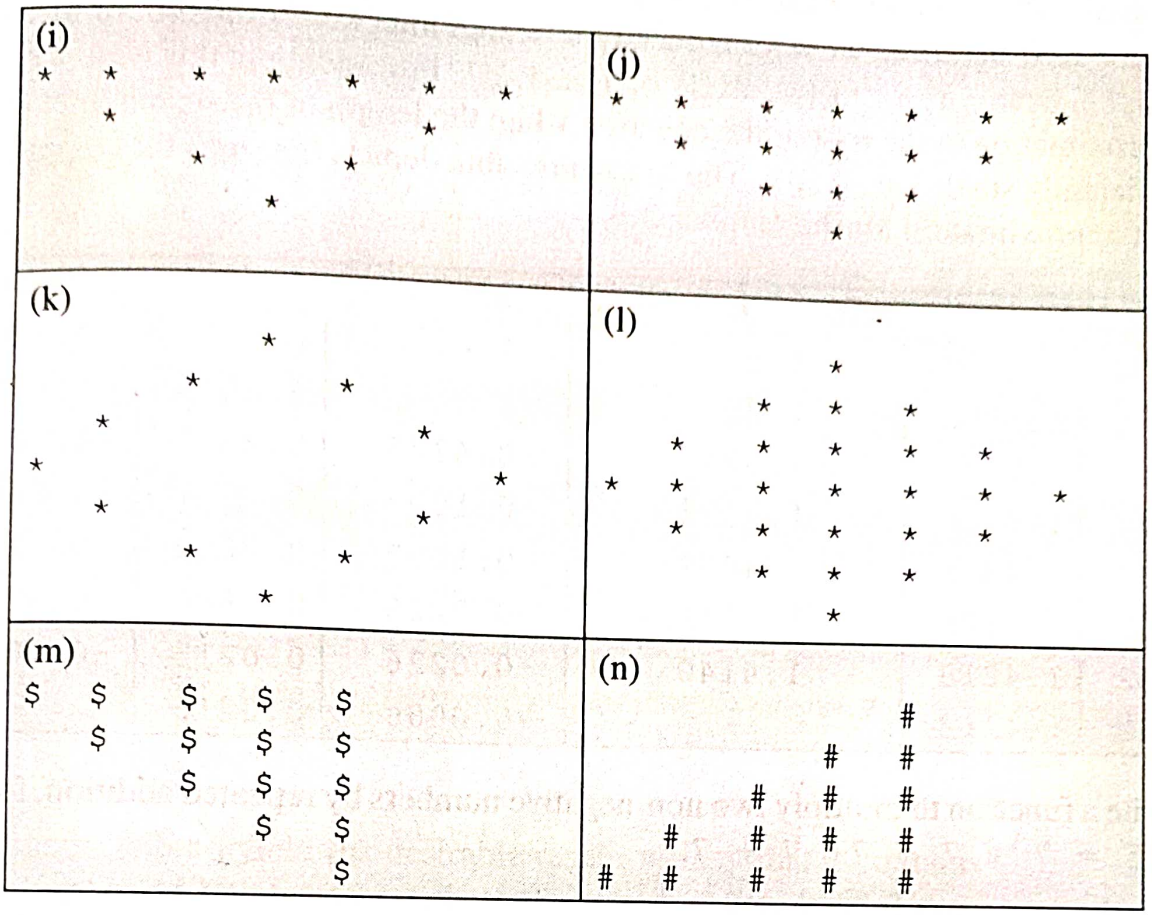
```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

(h)

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

(Cont'd)

Table (Continued)



7. Write a function that finds the sum of the `n` terms of the following series:
   (a) `1 - x² /2! + x⁴ /4! - x⁶ /6! + ... xⁿ /n!`
   (b) $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$
8. Write a function that returns `True` or `False` depending on whether the given number is a palindrome.
9. Write a function that returns the sum of digits of a number, passed to it as an argument.
10. Write a program that prints Armstrong numbers in the range 1 to 1000. An Armstrong number is a number whose sum of the cubes of the digits is equal to the number itself. For example, $370 = 3^3 + 7^3 + 0^3$.
11. Write a function that takes two numbers as input parameters and returns True or False depending on whether they are co-primes. Two numbers are said to be co-prime if they do not have any common divisor other than one.
12. Write a function sqrt that takes a non-negative number as an input and computes its square root. We can solve this problem iteratively. You will recall from high school mathematics that to find the square root of a number, say 2, we need to solve the equation `f(x) = x² - 2 = 0`. To begin with, choose two numbers `a` and `b` so that `f(a)<0` and `f(b)>0`. Now, for the equation `f(x) = x² - 2 = 0`, `f(1)<0` and `f(2)>0`. So, the root of the equation must lie in the interval `[a,b]` (i.e. `[1,2]`). We find the midpoint, say, `mid` of the interval `[a,b]`. If `f(a)<0` and `f(mid)>0`, we know that the root of the equation `f(x)=0` lies in the interval `[a,mid]`. However, in the other case, (`f(mid)<0` and

f(mid) >0), the root of the equation f(x) =0 must lie in the interval [mid,b]. Thus, for the next iteration, we have reduced the search interval for the root of the equation to half, i.e. from [a,b] to [a,mid] or [mid,b]. Proceeding in this way, we find a good approximation to the root of the equation when the length of the search interval becomes sufficiently small, say, 0.01. The following table depicts the steps for computing square root approximation for the number 2.

a	b	mid = (a + b)/2	f(a)	f(b)	f(mid)
1	2	1.5	-1	2	0.25
1	1.5	1.25	-1	0.25	-0.4375
1.25	1.5	1.375	-0.4375	0.25	-0.1093
1.375	1.5	1.4375	-0.1093	0.25	0.0664
1.375	1.4375	1.4062	-0.1093	0.0664	-0.0226
1.4062	1.4375	1.4218	-0.0226	0.0664	0.0215
1.4062	1.4218	1.4140	-0.0226	0.0215	-0.0006
1.4140	1.4218	-	-0.0006	0.0215	-

13. Write a function to multiply two non-negative numbers by repeated addition, for example, 7*5 = 7 + 7 + 7+ 7 + 7.