9.4 The if Statements of Python

The **if** statements are the conditional statements in Python and these implement **selection constructs (decision constructs)**.

An *if* statement tests a particular condition ; if the condition evaluates to ***true***, a course-of-action is followed *i.e.*, a statement or set-of-statements is executed. Otherwise (if the condition evaluates to ***false***), the course-of-action is ignored.

Before we proceed, it is important that you recall what you learnt in Chapter 7 in relational and logical operators as all that will help you form conditionals in this chapter. Also, please note in all forms of *if* statement, we are using *true* to refer to Boolean value ***True as well as truth value true*** $_{tval}$ collectively ; similarly, *false* here will refer to Boolean value ***False and truth value false*** $_{tval}$ collectively.

9.4.1 The if Statement

The simplest form of *if* statement tests a condition and if the condition evaluates to *true*, it carries out some instructions and does nothing in case condition evaluates to *false*.

The *if* statement is a compound statement and its syntax (general form) is as shown below :

```
if <conditional expression> :
    statement
    [statements]
```

The statements inside an ***if*** *are indented at same level.*

where a statement may consist of a single statement, a compound statement, or just the *pass* statement (in case of empty statement).

Carefully look at the *if statement* ; it is also a *compound statement* having a *header* and a *body* containing indented statements.

For instance, consider the following code fragment :

Conditional expression

The ***header*** *of if statement; notice colon (:) at the end*

```
if ch == ' ':
    spaces += 1
    chars += 1
```

Body of if *. Notice that all the statements in the* ***if-body*** *are indented at same level*

In an ***if statement***, if the *conditional expression* evaluates to *true*, the statements in the *body-of-if* are executed, otherwise ignored.

The above code will check whether the character variable **ch** stores a space or not ; if it does (*i.e.*, the condition ch== ' ' evaluates to *true*) , the number of spaces are incremented by *1* and number of characters (stored in variable chars) are also incremented by value *1*.

NOTE

Indentation in Python is very important when you are using a compound statement. Make sure to indent all statements in one block at the same level.

To see if Statement in action

If, however, variable ch does not store a space *i.e.*, the condition ch== ' ' evaluates to *false*, then nothing will happen ; no statement from the *body-of-if* will be executed. Consider another example illustrating the use of *if* statement :

```
ch = input ( "Enter a single character :" )
if ch >='0' and ch <='9' :
    print ("You entered a digit.")
```

The above code, after getting input in **ch**, compares its value ; if value of **ch** falls between characters '0' to '9' *i.e.*, the condition evaluates to *true*, and thus it will execute the statements in the ***if-body*** ; that is, it will print a message saying 'You entered a digit.'

Now consider another example that uses two if statements one after the other :

```
ch = input ('Enter a single character :')

if ch == ' ' :
    print ("You entered a space")

if ch >='0' and ch <='9' :
    print ("You entered a digit.")
```

*If this condition is **false** then the control will directly reach to following **if** statement, ignoring the body-of this-if statement*

The above code example reads a single character in variable **ch**. If the character input is a space, it flashes a message specifying it. If the character input is a digit, it flashes a message specifying it.

The following example also makes use of an ***if*** statement :

```
A = int ( input ( "Enter first integer :" ) )
B = int ( input ( "Enter second integer :" ) )
    if A > 10 and B < 15 :
    C = (A - B) * (A + B)
    print ("The result is", C)
print ("Program over")
```

*This statement is not part of **if** statement as it not indented at the same level as that of body of if statements.*

The above program has an ***if*** statement with two statements in its body ; last *print* statement is not part of ***if*** statement. Have a look at some more examples of conditional expressions in if statements :

(*a*)
```
if grade == 'A' :                    # comparison with a literal
    print ("You did well")
```

(*b*)
```
if a > b :                           # comparing two variables
    print ("A has more than B has")
```

(*c*)
```
if x :                               # testing truth value of a variable
    print ("x has truth value as true")
    print ("Hence you can see this message.")
```

You have already learnt about truth values and ***truth value testing*** in *Chapter 7 under section 7.4.3A*. I'll advise you to have a look at section 7.4.3A once again as it will prove very helpful in understanding conditional expressions.

Now consider one more test condition :

```
if not x :                  # not x will return True when x has a truth value as false
    print ("x has truth value as false this time")
```

The value of **not** x will be *True* only when x is $false_{tval}$ and *False* when x is $true_{tval}$.

The above discussed plain if statement will do nothing if the condition evaluates to *false* and moves to the next statement that follows if-statement.

9.4.2 The if – else Statement

This form of *if* statement tests a condition and if the condition evaluates to *true*, it carries out statements indented below **if** and in case condition evaluates to *false*, it carries out statements indented below **else**. The syntax (general form) of the ***if-else*** statement is as shown below :

```
if <conditional expression> :
    statement
    [statements]
else :
    statement
    [statements]
```

*colons after both **if** and **else***

The block below if gets executed if the condition evaluates to *true* and the block below **else** gets executed if the condition evaluates to *false*.

For instance, consider the following code fragment :

```
if a >= 0 :
    print (a, "is zero or a positive number")
else :
    print (a, "is a negative number")
```

For any value more than zero (say 7) of variable ***a***, the above code will print a message like :

```
7 is zero or a positive number
```

And for a value less than zero (say –5) of variable ***a***, the above code will print a message like :

```
-5 is a negative number
```

Unlike previously discussed plain-if statement, which does nothing when the condition results into *false*, the **if-else** statement performs some action in both cases whether condition is *true* or *false*. Consider one more example :

```
if sales >= 10000 :
    discount = sales * 0.10
else :
    discount = sales * 0.05
```

The above statement calculates *discount* as 10% of *sales amount* if it is 10000 or more otherwise it calculates discount as 5% of *sales amount*.

NOTE

An if-else in comparison to two successive **if** statements has less number of condition-checks *i.e.*, with **if-else**, the condition will be tested once in contrast to two comparisons if the same code is implemented with two successive **if**s.

Following figure (Fig. 9.4) illustrates *if* and *if-else* constructs of Python.

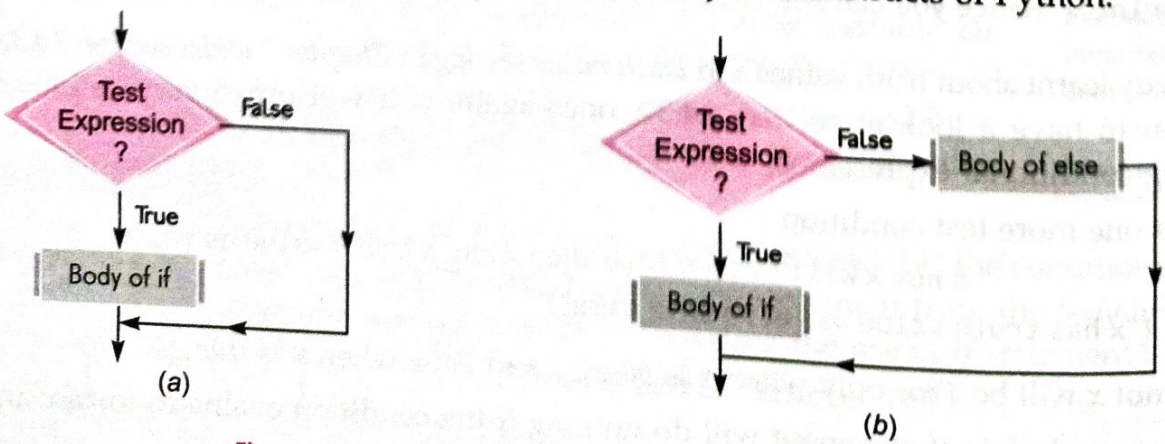


Figure 9.4 (*a*) The **if** statement's operation (*b*) The **if-else** statement's operation.

Let us now apply this knowledge of *if* and *if-else* in form of some programs.

Program 9.1 *Program that takes a number and checks whether the given number is odd or even.*

```
num = int ( input( "Enter an integer :" ))
if num % 2 == 0 :
   print (num, "is EVEN number.")
else :
   print (num, "is ODD number.")
```

The sample run of above code is as shown below.

This Program in action

Scan QR Code

```
Enter an integer : 8
8 is EVEN number.
```

Program 9.2 *Program to accept three integers and print the largest of the three. Make use of only if statement.*

```
x = y = z = 0
x = float (input( "Enter first number :" ) )
y = float (input( "Enter second number :" ) )
z = float (input( "Enter third number :" ) )

max = x
if y > max :
   max = y
if z > max :
   max = z
print ("Largest number is", max)
```

```
Enter first number : 7.235
Enter second number : 6.99
Enter third number : 7.533
Largest number is  7.533
```

Program 9.3 *Program that inputs three numbers and calculates two sums as per this :*

Sum1 as the sum of all input numbers

Sum2 as the sum of non-duplicate numbers; if there are duplicate numbers in the input, ignores them

e.g., Input of numbers 2, 3, 4 will give two sums as 9 and 9

Input of numbers 3,2, 3, will give two sums as 8 and 2 (both 3's ignored for second sum)

Input of numbers 4, 4, 4 will give two sums as 12 and 0 (all 4's ignored for second sum)

Alternative 1

```
sum1 = sum2 = 0
num1 = int(input("Enter number 1 : "))
num2 = int(input("Enter number 2 : "))
num3 = int(input("Enter number 3 : "))

sum1 = num1 + num2 + num3
if num1 != num2 and num1 != num3 :
   sum2 += num1
if num2 != num1 and num2 != num3 :
   sum2 += num2
if num3 != num1 and num3 != num2 :
   sum2 += num3

print("Numbers are", num1, num2, num3)
print("Sum of three given numbers is", sum1)
print("Sum of non-duplicate numbers is", sum2)
```

Alternative 2

```
sum1 = sum2 = 0
num1 = int(input("Enter number 1 : "))
num2 = int(input("Enter number 2 : "))
num3 = int(input("Enter number 3 : "))
sum1 = num1 + num2 + num3

if num1 == num2 :
   if num3 != num1 :
      sum2 += num3
else :
   if num1 == num3 :
      sum2 += num2
   else :
      if num2 == num3 :
         sum2 += num1
      else :
         sum2 += num1 + num2 + num3
print("Numbers are", num1, num2, num3)
print("Sum of three given numbers is", sum1)
print("Sum of non-duplicate numbers is", sum2)
```

Sample run of above program is as shown below :

```
Enter number 1 : 2
Enter number 2 : 3
Enter number 3 : 4
Numbers are 2 3 4
Sum of three given numbers is 9
Sum of non-duplicate numbers is 9
==========
Enter number 1 : 3
Enter number 2 : 2
Enter number 3 : 3
Numbers are 3 2 3
Sum of three given numbers is 8
Sum of non-duplicate numbers is 2
==========
Enter number 1 : 4
Enter number 2 : 4
Enter number 3 : 4
Numbers are 4 4 4
Sum of three given numbers is 12
Sum of non-duplicate numbers is 0
```

Program 9.4 *Program to test the divisibility of a number with another number (i.e., if a number is divisible by another number).*

```
number1 = int(input("Enter first number :"))
number2 = int(input("Enter second number :"))
remainder = number1 % number2
if remainder == 0 :
    print(number1,"is divisible by", number2)
else :
    print(number1,"is not divisible by", number2)
```

```
Enter first number : 119
Enter second number : 3
119.0 is not divisible by 3.0
==========
Enter first number : 119
Enter second number : 17
119.0 is divisible by 17.0
==========
Enter first number : 1234.30
Enter second number : 5.5
1234.3 is not divisible by 5.5
```

Program 9.5 *Program to find the multiples of a number (the divisor) out of given five numbers.*

```
print("Enter five numbers below")
num1 = float(input("First number :"))
num2 = float(input("Second number :"))
num3 = float(input("Third number :"))
num4 = float(input("Fourth number :"))
num5 = float(input("Fifth number :"))
divisor = float(input("Enter divisor number :"))
count = 0
print("Multiples of", divisor, "are :")
remainder = num1 % divisor
if remainder == 0 :
    print(num1, sep =" ")
    count += 1
remainder = num2 % divisor
if remainder == 0 :
    print(num2, sep =" ")
    count += 1
remainder = num3 % divisor
if remainder == 0 :
    print(num3, sep =" ")
    count += 1
remainder = num4 % divisor
if remainder == 0 :
    print(num4, sep =" ")
    count += 1
remainder = num5 % divisor
if remainder == 0 :
    print(num5, sep =" ")
    count += 1
print()
print(count, "multiples of", divisor, "found")
```

Sample run of above program is as shown below :

```
Enter five numbers below
First number : 185
Second number : 3450
Third number : 1235
Fourth number : 1100
Fifth number : 905
Enter divisor number : 15
Multiples of 15.0 are :
3450.0
1 multiples of 15.0 found
```

9.6 *Program to display a menu for calculating area of a circle or perimeter of a circle.*

```
radius = float ( input ( "Enter radius of the circle :" ))
print ("1. Calculate Area")
print ("2. Calculate Perimeter")
choice = int (input( "Enter your choice (1 or 2 ) :" ))
if choice == 1 :
    area = 3.14159 * radius * radius
    print ("Area of circle with radius", radius, 'is', area)
else :
    perm = 2 * 3.14159 * radius
    print ("Perimeter of circle with radius", radius, 'is', perm)
```

```
Enter radius of the circle : 2.5
    1. Calculate Area
    2. Calculate Perimeter
Enter your choice (1 or 2 ) : 1
Area of circle with radius  2.5  is 19.6349375
========== RESTART ==========
Enter radius of the circle : 2.5
    1. Calculate Area
    2. Calculate Perimeter
Enter your choice (1 or 2 ) : 2
Perimeter of circle with radius 2.5 is 15.70795
```

Notice, the test condition of *if* statement can be any relational expression or a logical statement (*i.e.*, a statement that results into either *true* or *false*). The *if* statement is a compound statement, hence its both **if** and **else** lines must end in a colon and statements part of it must be indented below it.

9.4.3 The if - elif Statement

Sometimes, you need to check another condition in case the test-condition of *if* evaluates to *false*. That is, you want to check a condition when control reaches **else** part, *i.e.*, condition test in the form of **else if**. To understand this, consider this adjacent example.

```
if runs are more than 100
    then it is a century
else if runs are more than 50
    then it is a fifty
else
    batsman has neither scored a century nor fifty
```

Refer to program 9.3 given earlier, where we have used **if** inside another **if/else**.

To serve conditions as above *i.e.*, in **else if** form (or **if** inside an **else**), Python provides *if-elif-else* statements. The general form of these statements is :

```
if <conditional expression> :
    statement
    [statements]
elif <conditional expression> :
    statement
    [statements]
```

and

```
if <conditional expression> :
    statement
    [statements]
elif <conditional expression> :
    statement
    [statements]
else :
    statement
    [statements]
```

NOTE

if, **elif** and **else** all are block or compound statements.

Now the above mentioned example can be coded in Python as :

To see if-elif Statement in action

Scan QR Code

```
if runs >= 100 :
    print ("Batsman scored a century")
elif runs >= 50 :
    print ("Batsman scored a fifty")
else :
    print ("Batsman has neither scored a century nor fifty")
```

Python will test this condition in case previous condition (runs >= 100) is false.

This block will be executed when both the if's condition (i.e., runs >=100) and elif condition (i.e., runs>=50) are false.

Let us have a look at some more code examples :

```
if num < 0 :
    print (num, "is a negative number.")
elif num == 0 :
    print (num, "is equal to zero.")
else :
    print (num, "is a positive number.")
```

Can you make out what the above code is doing ? Hey, don't get angry. I was just kidding I know you know that it is testing a number whether it is negative (< 0), or zero (= 0) or positive (> 0). Consider another code example :

```
if sales >= 30000 :
    discount = sales * 0.18
elif sales >= 20000 :
    discount = sales * 0.15
elif sales >= 10000 :
    discount = sales * 0.10
else :
    discount = sales * 0.05
```

*There can be as many **elif blocks** as you need. Here, the code is having two **elif blocks***

This block will be executed when none of above conditions are true.

Consider following program that uses an if-elif statement.

Program 9.7 *Program that reads two numbers and an arithmetic operator and displays the computed result.*

```
num1 = float ( input( "Enter first number :" ) )
num2 = float ( input( "Enter second number :" ) )
op = input( "Enter operator [ + - * / % ] :" )
result = 0
if op =='+' :
    result = num1 + num2
elif op =='-' :
    result = num1 - num2
elif op =='*' :
    result = num1 * num2
elif op =='/' :
    result = num1 / num2
elif op =='%' :
    result = num1 % num2
else :
    print ("Invalid operator!!")
print (num1, op, num2 , '=', result)
```

```
Enter first number : 5
Enter second number : 2
Enter operator [ + - * / % ] : *
5.0 * 2.0 = 10.0
========== RESTART ==========
Enter first number : 5
Enter second number : 2
Enter operator [ + - * / % ] : /
5.0 / 2.0 = 2.5
========== RESTART ==========
Enter first number : 5
Enter second number : 2
Enter operator [ + - * / % ] : %
5.0 % 2.0 = 1.0
```

Now try writing code of program 9.3 using **if-elif-else** statements.

9.4.4 The nested if Statement

Sometimes above discussed forms of if are not enough. You may need to test additional conditions. For such situations, Python also supports *nested-if form of if*.

A *nested if* is an *if* that has another **if** in its *if*'s body or in *elif*'s body or in its *else*'s body.

The **nested if** can have one of the following forms :

Form 1 (*if inside if's body*)

```
if <conditional expression> :
   if <conditional expression> :
      statement(s)
   else :
      statement(s)
elif <conditional expression> :
   statement
   [statements]
else :
   statement
   [statements]
```

Form 2 (*if inside elif's body*)

```
if <conditional expression> :
   statement
   [statements]
elif <conditional expression> :
   if <conditional expression> :
      statement(s)
   else :
      statement(s)
else :
   statement
   [statements]
```

Form 3 (*if inside else's body*)

```
if <conditional expression> :
   statement
   [statements]
elif <conditional expression> :
   statement
   [statements]
else :
   if <conditional expression> :
      statement(s)
   else :
      statement(s)
```

Form 4 (*if inside if's as well as else's or elif's body, i.e., multiple ifs inside*)

```
if <conditional expression> :
   if <conditional expression> :
      statement(s)
   else :
      statement(s)
elif <conditional expression> :
   if <conditional expression> :
      statement(s)
   else :
      statement(s)
else :
   if <conditional expression> :
      statement(s)
   else :
      statement(s)
```

In a **nested if** statement, either there can be *if* statement(s) in its *body-of-if* or in *its body-of-elif* or in its *body-of-else* or in any two of these or in all of these. Recall that you used **nested-if** unknowingly in program 9.3. Isn't that superb ? ;)

Following example programs illustrate the use of nested ifs.

Program 9.8 *Program that reads three numbers (integers) and prints them in ascending order.*

```
x = int(input( "Enter first number :" ))
y = int(input( "Enter second number :" ))
z = int(input( "Enter third number :" ))
min = mid = max = None

if x < y and x < z :
    if y < z :
        min, mid, max = x, y, z
    else :
        min, mid, max = x, z, y

elif y < x and y < z :
    if x < z :
        min, mid, max = y, x, z
    else :
        min, mid, max = y, z, x
else :
    if x < y :
        min, mid, max = z, x, y
    else:
        min, mid, max = z, y, x

print ("Numbers in ascending order :", min, mid, max)
```

Two sample runs of the program are as given below :

```
Enter first number : 5
Enter second number : 9
Enter third number : 2
Numbers in ascending order : 2 5 9
========== RESTART ==========
Enter first number : 9
Enter second number : 90
Enter third number : 19
Numbers in ascending order : 1 19 9
```

Let us now have a look at some programs that use different forms of *if* statement.

Program 9.9 *Program to print whether a given character is an uppercase or a lowercase character or a digit or any other character.*

```
ch = input( "Enter a single character :" )
if ch >= 'A' and ch <= 'Z' :
  print ("You entered an Upper case character.")
elif ch >= 'a' and ch <= 'z' :
  print ("You entered a lower case character.")
elif ch >= '0' and ch <= '9' :
  print ("You entered a digit.")
else :
  print ("You entered a special character.")
```

Sample run of the program is as shown below :

```
Enter a character : 5
You entered a digit.
========== RESTART ==========
Enter a character : a
You entered a lower case character.
========== RESTART ==========
Enter a character : H
You entered an Upper case character.
========== RESTART ==========
Enter a character : $
You entered a special character.
```

9.10 *Program to calculate and print roots of a quadratic equation : $ax^2 + bx + c = 0$ $(a \neq 0)$.*

Program

```
import math

print ("For quadratic equation, ax**2 + bx + c = 0, enter coefficients below")
a = int( input ( " Enter a :" ) )
b = int( input ( " Enter b :" ) )
c = int( input ( " Enter c :" ) )

if a == 0 :
    print ("Value of", a, ' should not be zero ')
    print ("\n Aborting !!!!!!")
else :
    delta = b * b - 4 * a * c
    if  delta > 0 :
        root1 = ( -b + math.sqrt(delta)) / (2 * a)
        root2 = (-b - math.sqrt(delta)) / (2 * a)
        print ("Roots are REAL and UNEQUAL")
        print ("Root1 =", root1, ", Root2 =", root2)
    elif delta == 0 :
        root1 = -b / (2 * a)
        print ("Roots are REAL and EQUAL")
        print ("Root1 =", root1, ", Root2 =", root1)
    else :
        print ("Roots are COMPLEX and IMAGINARY")
```

NOTE

You can use sqrt( ) function of math package to calculate squareroot of a number. Use it as :

```
math.sqrt(<number or expression>)
```

To use this function, add first line as **import math** to your program

NOTE

An *if* statement is also known as a conditional.

```
For quadratic equation, ax**2 + bx + c = 0, enter coefficients below
  Enter a : 3
  Enter b : 5
  Enter c : 2
Roots are REAL and UNEQUAL
Root1 =  - 0.666666666667 ,  Root2 =  -1.0
========== RESTART ==========
For quadratic equation, ax**2 + bx + c = 0, enter coefficients below
  Enter a : 2
  Enter b : 3
  Enter c : 4
Roots are COMPLEX and IMAGINARY
========== RESTART ==========
For quadratic equation, ax**2 + bx + c = 0, enter coefficients below
  Enter a : 2
  Enter b : 4
  Enter c : 2
Roots are REAL and EQUAL
Root1 =  -1 , Root2 =  -1
```

Storing Conditions

Sometimes the conditions being used in code are complex and repetitive. In such cases, to make your program more readable, you can use **named conditions** *i.e.*, you can store conditions in a name and then use that named conditional in the if statements.

For example, to create conditions similar to the ones given below :

- If a deposit is less than ₹2000 and for 2 or more years, the interest rate is 5 percent.
- If a deposit is ₹2000 or more but less than ₹6000 and for 2 or more years, the interest rate is 7 percent.
- If a deposit is more than ₹6000 and is for 1 year or more, the interest is 8 percent.
- On all deposits for 5 years or more, interest is 10 percent compounded annually.

The traditional way of writing code will be :

```
if deposit < 2000 and time >= 2 :
    rate = 0.05
    :
```

But if you name the conditions separately and use them later in your code, it adds to readability and understandability of your code, *e.g.*,

```
eligible_for_5_percent = deposit < 2000 and time >= 2
eligible_for_7_percent = 2000 <= deposit < 6000 and time >= 2
eligible_for_8_percent = deposit > 6000 and time >= 1
eligible_for_10_percent = time >= 5
```

Now you can use these *named conditionals* in the code as follows :

```
if eligible_for_5_percent :
    rate = 0.05
elif eligible_for_7_percent :
    rate = 0.075
    :
```

TIP

You can use named conditions in if statements to enhance readability and reusability of conditions.

Check Point 9.2

1. What is a selection statement? Which selection statements does Python provide ?
2. How does a conditional expression affect the result of if statement?
3. Correct the following code fragment :

```
if (x = 1)
  k = 100
else
  k = 10
```

4. What will be the output of following code fragment if the input given is (*i*) 7 (*ii*) 5 ?

```
a = input('Enter a number')
if (a == 5) :
  print ("Five" )
else :
  print ("Not Five")
```

5. What will be the output of the following code fragment if the input given is (*i*) 2000 (*ii*) 1900 (*iii*) 1971 ?

```
year = int(input( "Enter 4-digit year"))
if (year % 100 == 0) :
  if (year % 400 == 0):
    print ("LEAP century year")
else :
  print ("Not century year")
```

6. What will be the output of the following code fragment if the input given is (*i*) 2000 (*ii*) 1900 (*iii*) 1991 ?

```
year = int(input( "Enter 4-digit year"))
if (year % 100 == 0) :
  if (year % 400 == 0):
    print ("LEAP century year")
else :
  print ("Not century year")
```

DECISION CONSTRUCT *if* — Progress In Python 9.1

This 'Progress in Python' session is aimed at laying strong foundation of decision constructs.

:

>>>❖<<<

Repetition of Tasks – a Necessity

We mentioned earlier that there are many situations where you need to repeat same set of tasks again and again *e.g.*, the tasks of making chapatis/dosas/appam at home, as mentioned earlier. Now if you have to write pseudocode for this task how would you do that.

Let us first write the task of making dosas/appam from a prepared batter.

1. Heat flat pan/tawa
2. Spread evenly the dosa/appam batter on the heated flat pan.
3. Flip it carefully after some seconds.
4. Flip it again after some seconds.
5. Take it off from pan.
6. To make next dosa/appam go to step 2 again.

Let us try to write pseudocode for above task of making dosas/appam as many as you want.

```
#Prerequisites : Prepared batter
Heat flat-pan/tawa
Spread batter on flat pan
Flip the dosa/appam after 20 seconds
Flip the dosa/appam after 20 seconds
Take it off from pan
if you want more dosas then
    ????
else
    stop
```

Now what should you write at the place of **????**, so that it starts repeating the process again from second step ? Since there is no number associated with steps, how can one tell from where to repeat ?

There is a way out : If there is a **label** that marks the statement from where you want to repeat the task, then we may send the control to that label and then that statement onwards, statements get repeated. That is,

Pseudocode Reference 1

```
#Prerequisites : prepared batter
Heat flat-pan
Spread : Spread batter on flat pan
Flip the dosa/appam after 20 seconds
Flip the dosa/appam after 20 seconds
Take it off from pan
If you want more dosas then
    Go to Spread
Else
    Stop
```

Label given to this statements

from here the control will go to spread batter... step and all steps following it will be repeated.

[Dotted arrows show how the control flows from one statement to another]

Now the above pseudocode is fit for making as many dosas/appam as you want. But what if you already know that you need only 6 dosas/appam ? In that case above pseudocode will not serve the purpose.

Let us try to write pseudocode for this situation. For this, we need to maintain a count of dosa/appam made.

Pseudocode Reference 2

```
                #Prerequisite : prepared batter
A.              Heat flat pan
B.              Initialize count to 0
C.  Spread:     Spread batter on flat pan
D.              Flip the dosa/appam after 20 seconds
E.              Flip the dosa/appam after 20 seconds
F.              Take it off from pan
G.              Add 1 to count
H.              If count == 6 then
                    Stop
I.              Else
                    Go to spread
J.              End
```

[Dotted arrows show how the control flow from one statement to another]

If we number the pseudocode statements as A.B.C.... as shown above for our reference, if you notice carefully, the statements are executed as :

A B C D E F G H I C D E F G H I C D E F G H I

Statements repeated

C D E F G H I C D E F G H I C D E F G H I J.

You can see that statements C to I are repeated 6 times or we can say that loop repeated 6 times.

This way we can develop logic for carrying out repetitive tasks, either based on a condition (*pseudocode reference 1*) or a given number of times (*pseudocode reference 2*).

To carry out repetitive tasks, Python does not have any *go to statement* rather it provides following iterative/looping statements :

- *Conditional loop* **while** (condition based loop)
- *Counting loop* **for** (loop for a given number of times).

Let us learn to write code for repetitive tasks, in Python.

9.6 The range( ) Function

Before we start with loops, especially *for loop* of Python, let us discuss the ***range( )*** function of Python, which is used with the Python ***for loop***. The *range ( )* function of Python generates a **list** which is a special sequence type. A sequence in Python is a succession of values bound together by a single name. Some Python sequence types are : *strings, lists, tuples* etc. (see Fig. 9.5). There are some advance sequence types also but those are beyond the scope of our discussion.

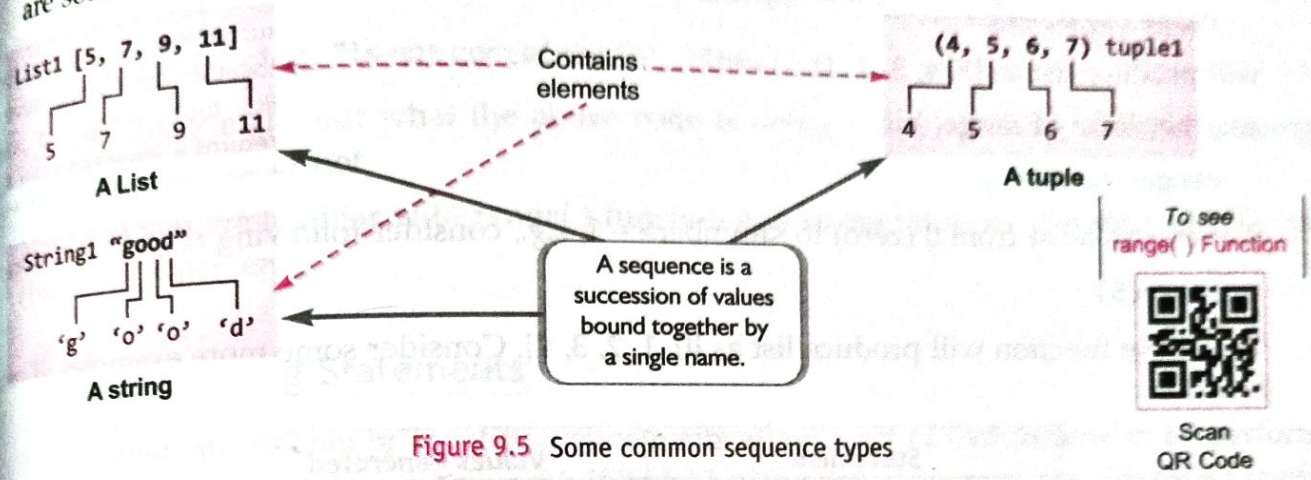


Figure 9.5 Some common sequence types

Let us see how *range( )* function works. The common use of *range( )* is in the form given below :

```
range( <lower limit>, <upper limit>)    # both limits should be integers
```

The function in the form **range(l, u)** will produce a list having values starting from $l, l+1, l+2$ ... $(u-1)$ (l and u being integers). Please note that the lower limit is included in the list but upper limit is not included in the list, *e.g.*,

```
range(0,5)
```


will produce list as **[0, 1, 2, 3, 4]**.

As these are the numbers in arithmetic progression (a.p.) that begins with lower limit 0 and goes up till *upper limit minus 1 i.e.*, $5 - 1 = 4$.

```
range(5,0)
```


will return empty list [ ] as no number falls in the a.p. beginning with 5 and ending at 0 (difference d or step-value = +1).

```
range(12, 18)
```


will give a list as [12, 13, 14, 15, 16, 17].

All the above examples of *range( )* produce numbers increasing by value 1. What if you want to produce a list with numbers having gap other than 1, *e.g.*, you want to produce a list like [2, 4, 6, 8] using range( ) function ?

For such lists, you can use following form of *range( )* function :

```
range( <lower limit>, <upper limit>, <step value>)    #all values should be integers
```

That is, function

```
range(l, u, s)          # l, u and s are integers
```

will produce a list having values as $l, l + s, l + 2s, \ldots <= u - 1$.

```
range(0, 10, 2)
```


will produce list as [0, 2, 4, 6, 8]

```
range (5, 0, -1)
```


will produce list as [5, 4, 3, 2, 1].

Another form of range( ) is :

```
range(<number>)
```

that creates a list from 0 (zero) to <number> – 1, *e.g.*, consider following *range( )* function :

```
range(5)
```

The above function will produce list as [0, 1, 2, 3, 4]. Consider some more examples of *range( )* function :

Statement	Values generated
range(10)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
range(5, 10)	5, 6, 7, 8, 9
range(3, 7)	3, 4, 5, 6
range(5, 15, 3)	5, 8, 11, 14
range(9, 3, –1)	9, 8, 7, 6, 5, 4
range(10, 1, –2)	10, 8, 6, 4, 2

NOTE

A sequence in Python is a succession of values bound together by a single name e.g. list, tuple, string. The range( ) returns a sequence of list type.

Operators in and not in

Let us also take about **in** operator, which is used with *range( )* in for loops.

To check whether a value is contained inside a list you can use **in** operator, *e.g.*,

```
3 in [1,2,3,4]
```


will return ***True*** as value 3 is contained in sequence [1, 2, 3, 4].

```
5 in [1,2,3,4]
```

will return ***False*** as value 5 is not contained in sequence [1, 2, 3, 4]. But

```
5 not in [1,2,3,4]
```

will return ***True*** as this fact is true that as value 5 is not contained in sequence [1, 2, 3, 4]. The operator **not in** does opposite of **in** operator.

NOTE

The **in** operator tests if a given value is contained in a sequence or not and returns **True** or **False** accordingly.

You can see that **in** and **not in** are the operators that check for membership of a value inside a sequence. Thus, **in** and **not in** are also called **membership operators**. These operators work with all sequence types *i.e., strings, tuples* and *lists* etc. For example, consider following code :

```
'a' in "trade"
```

will return ***True*** as '*a*' is contained in sequence (string type) "trade".

```
"ash" in "trash"
```

will also return ***True*** for the same reason.

Now consider the following code that uses the **in** operator :

```
line = input( "Enter a line :" )
string = input( "Enter a string :" )
if string in line :
    print (string, "is part of", line.)
else :
    print (string, "is not contained in", line.)
```

if the given ***string*** *is a part of* ***line***

NOTE

Operators in and not in are also called membership operators.

You can easily make out what the above code is doing – it is checking whether a string is contained in a line or not.

Now that you are familiar with **range( )** function and **in** operator, we can start with Looping (Iteration) statements.

9.7 Iteration/Looping Statements

The iteration statements or repetition statements allow a set of instructions to be performed repeatedly until a certain condition is fulfilled. The iteration statements are also called ***loops*** or ***looping statements***. Python provides *two* kinds of loops : *for loop* and *while loop* to represent *two* categories of loops, which are :

- *counting loops* — the loops that repeat a certain number of times ; Python's *for loop* is a counting loop (or ***count-controlled loop***).
- *conditional loops* — the loops that repeat until a certain thing happens *i.e.*, they keep repeating as long as some condition is *true* ; Python's *while loop* is conditional loop.

If the condition is test before executing the *loop-body*, *i.e.*, before entering in the loop, the loop is called **entry-controlled loop** or **pre-condition loop**. And if the condition is tested after executing the *loop-body* at least once, it is called ***exit-controlled loop*** or ***post-condition loop***.

9.7.1 The for Loop

The ***for loop*** of Python is designed to process the items of any sequence, such as a list or a string, one by one.

To see for Loop in action

Scan QR Code

The general form of ***for loop*** is as given below :

```
for <variable> in <sequence> :
    statements_to_repeat
```

This determines how many time the loop will get repeated.

Colon is must here

here the statements to be repeated will be placed (body-of-the-loop)

For example, consider the following loop :

```
for a in [1, 4, 7] :
    print (a)
    print(a * a)
```

The loop variable ***a****. Variable* ***a*** *will be assigned each value of list one by one, i.e., for the first time* ***a*** *will be* ***1****, then* ***4*** *and then* ***7****.*

This is the ***body of the for loop****. All statements in the body of the loop will be executed for each value of loop variable* ***a****, i.e., firstly for* ***a*** *=****1****; then for* ***a*** *=* ***4*** *and then for* ***a****=****7***

A *for loop* in Python is processed as :

- The loop-variable is assigned the first value in the sequence.
- all the statements in the body of *for loop* are executed with assigned value of loop variable (step 2)
- once step 2 is over, the loop-variable is assigned the next value in the sequence and the loop-body is executed (*i.e.*, step 2 repeated) with the new value of loop-variable.
- This continues until all values in the sequences are processed.

NOTE

Each time, when the loop-body is executed, is called an *iteration*.

That is, the given *for loop* will be processed as follows :

(*i*) firstly, the loop-variable ***a*** will be assigned first value of list *i.e., 1* and the statements in the body of the loop will be executed with this value of *a*. Hence value 1 will be printed with statement **print(a)** and value 1 will again be printed with **print(a*a)**. (*see execution shown on right*)

(*ii*) Next, ***a*** will be assigned next value in the list *i.e., 4* and loop-body executed. Thus 4 (as result of **print(a)**) and 16 (as result of **print(a*a)**) are printed.

(*iii*) Next, ***a*** will be assigned next value in the list *i.e., 7* and loop-body executed. Thus 7 (as result of **print(a)**) and 49 (as result of **print(a*a)**) are printed.

(*iv*) All the values in the list are executed, hence loop ends.

Therefore, the output produced by above *for loop* will be :

```
1
1
4
16
7
49
```

```
for a in [1, 4, 7]:
    print (a)          <-- body of the loop
    print (a * a)
```

(*i*)
```
a = 1
a assigned first value and with this value, all statements in loop body are executed :
print (1)      --> prints 1
print (1*1)    --> prints 1
```

(*ii*)
```
a = 4
a assigned next value and loop body executed with a as 4:
print (4)      --> prints 4
print (4*4)    --> prints 16
```

(*iii*)
```
a = 7
a assigned next value and loop body executed with a as 7
print (7)      --> prints 7
print (7*7)    --> prints 49
```

```
No more values
```

Consider another *for loop* :

```
for ch in 'calm' :
    print (ch)
```

The above loop will produce output as :

```
c
a
l
m
```

(variable **ch** given values as 'c', 'a', 'l', 'm' – one at a time from string 'calm'.)

Here is another *for loop* that prints the cube of each value in the list :

```
for v in [1, 2, 3 ] :
    print (v * v * v)
```

The above loop will produce output as :

```
1
8
27
```

The *for loop* works with other sequence types such as *tuples* also, but we'll stick here mostly to *lists* to process a sequence of numbers through *for loops*.

As long as the list contains small number of elements, you can write the *list* in the *for loop* directly as we have done in all above examples. But what if we have to iterate through a very large list such as containing natural numbers between 1 – 100 ? Writing such a big list is neither easy nor good on readability. Then ? Then what ? ☺ Arey, our very own *range( )* function is there to rescue. Why worry? ;) For big number based lists, you can specify *range( )* function to represent a list as in :

```
for val in range(3, 18) :
    print (val)
```

In the above loop, *range(3, 18)* will first generate a list [3, 4, 5, … , 16, 17] with which *for loop* will work. Isn't that easy and simple ? ☺

You need not define loop variable (**val** above) before-hand in a *for loop*. Now consider following code example that uses *range( )* function in a *for loop* to print the table of a number.

9.11 *Program to print table of a number, say 5.*

Program

```
num = 5
for a in range(1, 11) :
    print (num, 'x', a, '=', num * a)
```

The above code will print the output as shown here :

```
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

NOTE

(i) *for* loop ends when the loop is repeated for the last value of the sequence.

(ii) The *for loop* repeats ***n*** number of times, where ***n*** is the length of sequence given in for-loop's header.

NOTE

You need not pre-define loop variable of a *for loop*.

Program 9.12 *Program to print sum of natural numbers between 1 to 7. Print the sum progressively, i.e., after adding each natural number, print sum so far.*

```
sum = 0
for n in range(1, 8) :
    sum += n
    print ("Sum of natural numbers <=", n, "is", sum)
```

```
Sum of natural numbers <= 1 is 1
Sum of natural numbers <= 2 is 3
Sum of natural numbers <= 3 is 6
Sum of natural numbers <= 4 is 10
Sum of natural numbers <= 5 is 15
Sum of natural numbers <= 6 is 21
Sum of natural numbers <= 7 is 28
```

The above code will print the output as shown here :

Program 9.13 *Program to print sum of natural numbers between 1 to 7.*

```
sum = 0
for n in range(1, 8) :
    sum += n
print ("Sum of natural numbers <=", n, 'is', sum)
```

```
Sum of natural numbers <= 7 is 28
```

Carefully look at above program. It again emphasizes that body of the loop is defined through indentation and one more fact and important one too – *the value of loop variable after the* ***for loop*** *is over, is the highest value of the list.* Notice, the above loop printed value of ***n*** after ***for loop*** as 7, which is the maximum value of the list generated by *range(1, 8)*.

NOTE

The loop variable contains the highest value of the list after the ***for loop*** is over. Program 9.13 highlights this fact.

9.7.2 The while Loop

A ***while loop*** is a conditional loop that will repeat the instructions within itself as long as conditional remains *true* (Boolean *True* or truth value $true_{tval}$). The general form of Python while loop is :

```
while <logicalexpression> :
    loop-body
```

where the loop-body may contain a *single statement* or *multiple statements* or *an empty statement* (*i.e., pass* statement). The loop iterates while the *logical expression* evaluates to *true*. When the expression becomes *false*, the program control passes to the line after the loop-body.

To understand the working of ***while loop***, consider the following code :

```
a = 5
while a > 0 :
    print ("hello", a)
    a = a - 3
print ("Loop Over!!")
```

This condition is tested, if it is ***true****, the loop-body is executed. After the loop-body's execution, the condition is tested again, loop-body is executed as long as condition is* ***true****.*
The loop ends when the condition evaluates to ***false***

The above code will print :

```
hello 5
hello 2
Loop Over!!
```

These two lines are the result of ***while loops'*** *body execution (which executed twice).*

This line is because of print statement after the ***while loop.***

Let us see how a *while loop* is processed :

```
while a > 0 :
    print ("hello", a)
    a = a - 3
```

body of the loop

[a has value 5 before entering into while loop]

Step 1 The *logical/conditional expression in the while loop* (e.g., `a > 0` above) is evaluated.

Step 2 *Case I* if the result of step 1 is *true* (*True* or *true*$_{val}$) then all statements in the loop's body are executed, (see section (*i*) & (*ii*) on the right).

Case II if the result of step 1 is *false* (*False* or *false*$_{val}$) then control moves to the next statement after the body-of the loop *i.e.*, loop ends. (see section (*iii*) on the right).

Step 3 Body-of-the loop gets executed.

This step comes only if *Step* 2, *Case I* executed, *i.e.*, the result of *logical expression* was *true*. This step will execute two statements of loop body.

(*i*) while's condition tested with a's current value as 5

5 > 0 = True

hence loop's body will get executed

print ("hello", 5) ⟶ *prints* hello 5

a = 5 - 3 = 2. (a becomes 2 now)

(*ii*) while's condition tested with a's current values 2

2 > 0 = True

loop body will get executed

print ("hello", 2) ⟶ *prints* hello 2

a = 2 - 3 = -1 (a becomes -1 now)

(*iii*) while's condition tested with a's current value -1

-1 > 0 = False

loop body will not get executed

control passes to next statement after while loop

After executing the statements of loop-body again, *Step 1*, *Step 2* and *Step 3* are repeated as long as the condition remains *true*.

The loop will end only when the logical expression evaluates to *false*. (*i.e.*, *Step 2, Case II* is executed.) Consider another example :

NOTE

The variable used in the condition of *while loop* must have some value before entering into *while loop*.

```
n = 1
while n < 5 :
    print ("Square of ", n , 'is', n * n)
    n += 1
print ("Thank You.")
```

This statement is not part of the loop-body

The above code will print output as :

```
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Thank You.
```

These lines of output are because of the loop-body execution (loop executed 4 times)

Thus you see that as long as the condition $n < 5$ is *true* above, the *while loop's* body is executed (for values $n = 1$, $n = 2$, $n = 3$, $n = 4$). After exiting from the loop the statement following the *while* loop is executed that prints *Thank You*.

Anatomy of a while Loop (Loop Control Elements)

Every while loop has its elements that control and govern its execution. A **while** loop has four elements that have different purposes. These elements are as given below :

1. Initialization Expression(s) (Starting). Before entering in a *while* loop, its loop variable must be initialized. The initialization of the loop variable (or control variable) takes place under initialization expression(s). The initialization expression(s) give(s) the loop variable(s) their first value(s). The initialization expression(s) for a *while loop* are outside the *while* loop before it starts.

2. Test Expression (Repeating or Stopping). The test expression is an expression whose truth value decides whether the loop-body will be executed or not. If the test expression evaluates to *true*, the loop-body gets executed, otherwise the loop is terminated.

In a *while* loop, the test-expression is evaluated before entering into a loop.

3. The Body-of-the-Loop (Doing). The statements that are executed repeatedly (as long as the test-expression is *true*) form the *body of the loop*.

In a *while* loop, before every iteration the test-expression is evaluated and if it is *true*, the *body-of-the-loop* is executed; if the test-expression evaluates to *false*, the loop is terminated.

4. Update Expression(s) (Changing). The update expressions change the value of loop variable. The update expression is given as a statement inside the body of *while* loop.

Consider the following code that highlights these elements of a *while* loop :

Initialization expression : *initializes loop variable*

```
n = 10
while n > 0 :
    print (n)
    n -= 3
```

Test expression : *based on this condition, loop iterates i.e., repeats*

Body of the loop *(contains indented statements beneath while)*

Update expression : *changing value of loop variable*

Other important things that you need to know related to *while loop* are :

- In a *while loop*, a loop control variable should be initialized before the loop begins as an uninitialized variable cannot be used in an expression. Consider following code :

```
while p != 3 :
    :
```

This will result in error if variable **p** *has not been created beforehand.*

- The loop variable must be updated inside the *body-of the-while* in a way that after some time the test-condition becomes *false* otherwise the loop will become an **endless loop** or **infinite loop**. Consider following code :

```
a = 5
while a > 0 :
    print (a)
print ("Thank You")
```

ENDLESS LOOP! Because the loop-variable a remains 5 always as its value is not updated in the loop-body, hence the condition

The above loop is an endless loop as the value of loop-variable *a* is not being updated inside loop body, hence it always remains 5 and the loop-condition $a > 0$ always remains *true*.

To see while Loop in action

Scan QR Code

The corrected form of above loop will be :

```
a = 5
while a > 0 :
    print (a)
    a -= 1
print ("Thank You")
```

Loop variable **a** being updated *inside the loop-body.*

We will talk about infinite loops once again after the *break* statement is discussed.

Since in a *while loop*, the control in the form of condition-check is at the entry of the loop (loop is not entered, if the condition is *false*), it is an example of an **entry-controlled loop** or **pre-condition loop**. An *entry-controlled loop* is a loop that controls the entry in the loop by testing a condition. Python does not offer any *exit-controlled loop*.

NOTE

To cancel the running of an endless loop, you can press CTRL+C keys anytime during its endless repetition to stop it.

Now that you are familiar with *while loop*, let us write some programs that make use of *while loop*.

9.14 *Program to calculate the factorial of a number.*

Program

```
num = int( input ( "Enter a number :" ) )
fact = 1
a = 1
while a <= num :
    fact *= a                    # fact = fact*a
    a += 1                       # a = a + 1
print ("The factorial of" , num, "is" , fact)
```

This program in action

Scan QR Code

Sample runs of above program are given below :

```
Enter a number : 3
The factorial of 3 is 6
========== RESTART ==========
Enter a number : 4
The factorial of 4 is 24
```

9.15 *Program to calculate and print the sums of even and odd integers of the first n natural numbers.*

Program

```
n = int(input( "Up to which natural number ?" ))
ctr = 1
sum_even = sum_odd = 0
while ctr <= n :
    if ctr % 2 == 0 :                    # number is even
        sum_even += ctr
    else :
        sum_odd += ctr
    ctr += 1                             #increment the counter
print ("The sum of even integers is" , sum_even)
print ("The sum of odd integers is" , sum_odd)
```

The output produced by above code is :

```
Up to which natural number? 25
The sum of even integers is 156
The sum of odd integers is 169
```

9.7.3 Loop else Statement

Both loops of Python (*i.e.*, **for** loop and **while** loop) have an else clause, which is different from *else* of *if-else* statements. The **else of a loop executes only when the loop ends normally** (*i.e.*, only when the loop is ending because the *while* loop's test condition has resulted in *false* or the *for loop* has executed for the last value in sequence.) You'll learn in the next section that if a **break** statement is reached in the loop, *while* loop is terminated pre-maturely even if the test-condition is still *true* or all values of sequence have not been used in a *for* loop.

In order to fully understand the working of *loop-else clause*, it will be useful to know the working of *break* statements. Thus let us first talk about **break** statement and then we'll get back to *loop-else* clause again with examples.

NOTE

The **else clause of a Python loop** executes when the loop terminates normally, not when the loop is terminating because of a ***break*** statement.

9.7.4 Jump Statements – break and continue

Python offers two jump statements to be used within loops to jump out of loop-iterations. These are *break* and *continue* statements. Let us see how these statements work.

NOTE

A *break* statement skips the rest of the loop and jumps over to the statement following the loop.

9.7.4A The break Statement

The ***break*** statement enables a program to skip over a part of the code. A ***break*** statement terminates the very loop it lies within. Execution resumes at the statement immediately following the body of the terminated statement.

The following figure (Fig. 9.6) explains the working of a break statement :

Statements 4 is the first statement after the loop

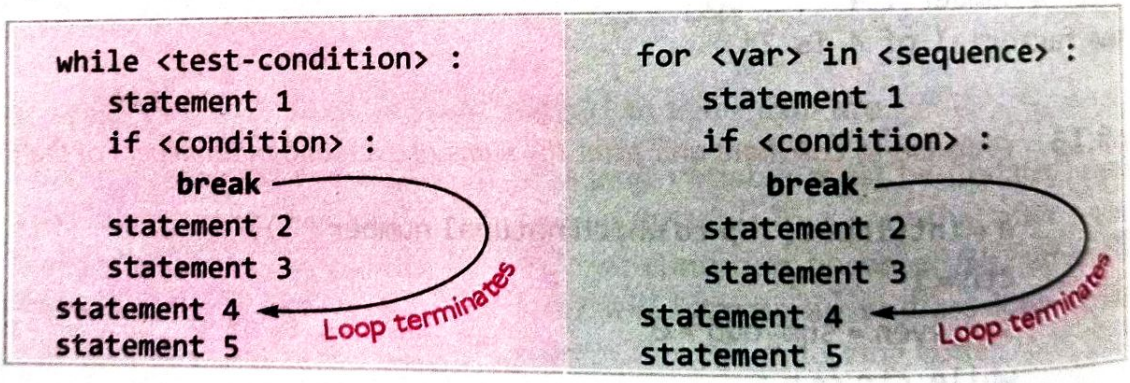


Figure 9.6 The working of a **break** statement.

The following code fragment gives you an example of a ***break*** statement :

```
a = b = c = 0
for i in range(1, 21) :
    a = int (input ( "Enter number 1 :" ))
    b = int (input ( "Enter number 2 :" ))
    if b == 0 :
        print ("Division by zero error! Aborting!")
        break
    else :
        c = a // b
        print ("Quotient = ", c)
print ("Program over !")
```

The above code fragment intends to divide ten pairs of numbers by inputting two numbers *a* and *b* in each iteration. If the number *b* is zero, the loop is immediately terminated displaying message '**Division by zero error! Aborting!**' otherwise the numbers are repeatedly input and their quotients are displayed.

Sample run of above code is given below :

```
Enter number 1 : 6
Enter number 2 : 2
Quotient =  3
Enter number 1 : 8
Enter number 2 : 3
Quotient =  2
Enter number 1 : 5
Enter number 2 : 0
Division by zero error! Aborting!
Program over !
```

This message is printed just prior to execution of break *statement. Notice that the next message is because of the statement outside the loop. That is, after* break*, next statement to be executed is the statement outside the loop.*

NOTE

A loop can end in two ways : (*i*) if the while's condition results in false or for loop executes with last value of the sequence (*NORMAL TERMINATION*). (*ii*) Or if during the loop execution, **break** statement is executed. In this case even if the condition is true or the for has not executed all the values of sequence, the loop will still terminate.

Consider another example of *break* statement in the form of following program.

9.16 *Program to implement 'guess the number' game. Python generates a number randomly in the range[10, 50]. The user is given five chances to guess a number in the range 10 <=number <=50.*

Program

- *If the user's guess matches the number generated, Python displays 'You win' and the loop terminates without completing the five iterations.*
- *If, however, cannot guess the number in five attempts, Python displays 'You lose'.*

To generate a random number in a range, use following function :

```
random.randint(lower limit, upper limit)
```

Make sure to add first line as import **random**

```
import random
number = random.randint(10,50)
ctr = 0
while ctr < 5 :
    guess = int(input( "Guess a number in range 10..50 :" ))
    if guess == number :
        print ("You win!! :)")
        break
    else :
        ctr += 1
if not ctr < 5 :                    # i.e., whether the loop terminated after 5 iterations
    print ("You lose :( \n The number was", number)
```

This will generate a number randomly, within the range 10-50.

The moment this statement is reached the loop will terminate without completing all iterations, even if the loop condition **ctr < 5** *is still true.*

Sample run of above program is as shown below :

```
Guess a number in range 10..50 : 31
Guess a number in range 10..50 : 23
Guess a number in range 10..50 : 34
Guess a number in range 10..50 : 40
Guess a number in range 10..50 : 50
You lose :(
The number was 28
========== RESTART ==========
Guess a number in range 10..50 : 39
Guess a number in range 10..50 : 46
You win!! :)
```

*At this point the **break** statement executed.*

From the above program, you can make out that if the ***while loop*** is terminated because of a **break** statement, the test-condition of the ***while loop*** will still be ***true*** even outside the loop. Thus, you can check the test-condition outside the loop to determine what caused the loop termination – the **break** statement or the test-condition's result. Recall the last part of above program, *i.e.*,

If the loop-condition is still true outside the loop that means break caused the loop termination.

```
if not ctr < 5 :
    print ("You lose :( \n The number was" , number)
```

Infinite Loops and break Statement

Sometimes, programmers create infinite loops purposely by specifying an expression which always remain *true*, but for such purposely created infinite loops, they incorporate some condition inside the loop-body on which they can break out of the loop using **break** statement.

For instance, consider the following loop example :

This will always remain True, hence infinite loop; must be terminated through a break statement

```
a = 2
while True :
    print (a)
    a *= 2
    if a > 100 :
        break
```

NOTE

For purposely created ***infinite*** (or ***endless***) loops, there must be a provision to reach break statement from within the body of the loop so that loop can be terminated.

9.7.4B The continue Statement

The **continue** statement is another jump statement like the **break** statement as both the statements skip over a part of the code. But the **continue** statement is somewhat different from **break**. Instead of forcing termination, the continue *statement forces the next iteration of the loop to take place*, skipping any code in between.

The following figure (Fig. 9.7) explains the working of **continue** statement :

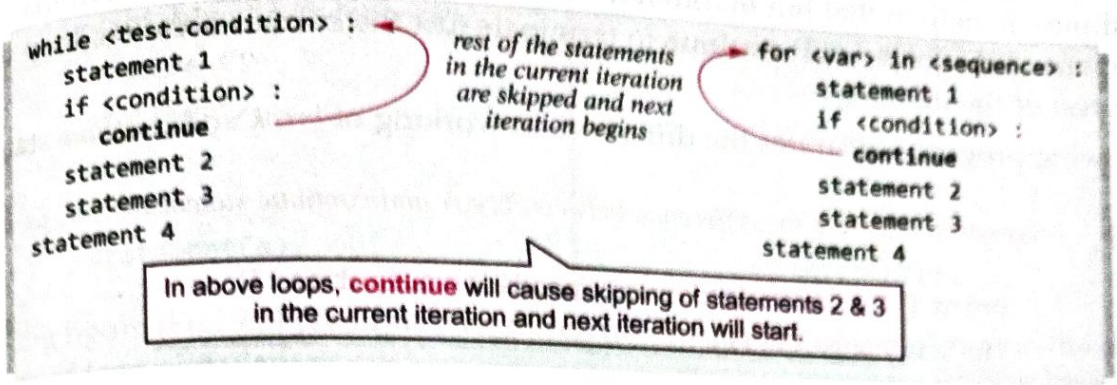


Figure 9.7 The working of a continue statement.

For the *for loop*, **continue** causes the next iteration by updating the loop variable with the next value in sequence ; and for the *while loop*, the program control passes to the conditional test given at the top of the loop.

NOTE

The **continue** statement skips the rest of the loop statements and causes the next iteration of the loop to take place.

```
a = b = c = 0
for i in range(0, 3) :
    print ("Enter 2 numbers" )
    a = int (input( "Number 1 :" ))
    b = int (input( "Number 2 :" ))
    if b == 0 :
        print ("\n The denominator cannot be zero. Enter again !")
        continue
    else :
        c = a//b
        print ("Quotient =", c)
```

Sample run of above code is shown below :

```
Enter 2 numbers
Number 1 : 5
Number 2 : 0
The denominator cannot be zero. Enter again !
Enter 2 numbers
Number 1 : 5
Number 2 : 2
Quotient = 2
Enter 2 numbers
Number 1 : 6
Number 2 : 2
Quotient = 3
```

At this point continue forced next iteration to take place.

Sometimes you need to abandon iteration of a loop prematurely. Both the statements **break** and **continue** can help in that but in different situations : statement **break** to terminate all pending iterations of the loop ; and **continue** to terminate just the current iteration, loop will continue with rest of the iterations.

Following program illustrates the difference in working of *break* and *continue* statements.

Program 9.17 *Program to illustrate the difference between break and continue statements.*

```
print ("The loop with 'break' produces output as :")
for i in range (1,11) :
    if i%3 == 0 :
        break
    else :
        print (i)

print ("The loop with 'continue' produces output as :")
for i in range (1,11) :
    if i%3 == 0 :
        continue
    else :
        print (i)
```

The output produced by above program is :

```
The loop with 'break' produces output as :
1
2
```

because the loop terminated with break *when condition* **i % 3** *became* **true** *with* **i = 3**. *(Loop terminated all pending iterations)*

```
The loop with 'continue' produces output as :
1
2
4
5
7
8
10
```

only the values divisible by 3 are missing as the loop simply moved to next iteration with continue *when condition* **i % 3** *became* **true**. *(Only the iterations wtih i % 3 == 0 are terminated ; next iterations are performed as it is)*

Though, **continue** is one prominent jump statement, however, experts discourage its use whenever possible. Rather, code can be made more comprehensible by the appropriate use of if/else.

9.7.5 Loop else Statement

Now that you know how **break** statement works, we can now talk about **loop-else** clause in details. As mentioned earlier, the *else clause of a Python loop* executes when the loop terminates normally, *i.e.*, when test-condition results into *false* for a ***while loop*** or ***for loop*** has executed for the last value in the sequence ; *not when* the **break** statement terminates the loop.

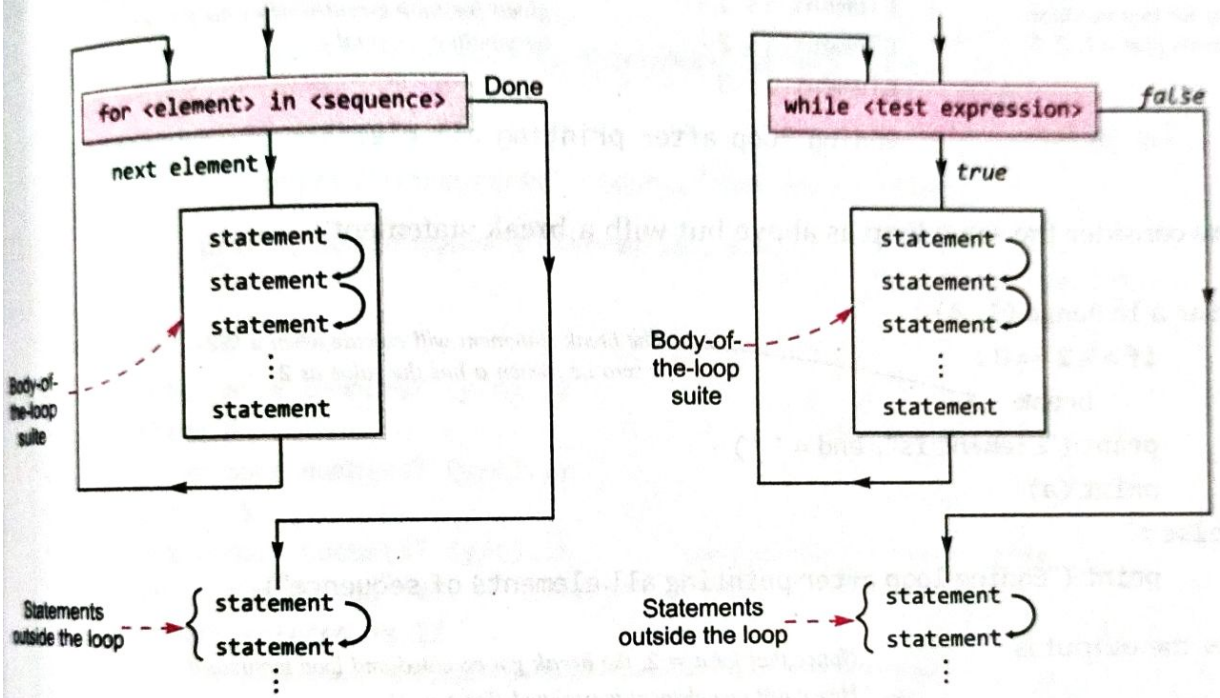
Before we proceed, have a look at the syntax of Python *loops along with else clauses*. Notice that the **else** clause of a loop appears at the same indentation as that of loop keyword **while** or **for**.

Complete syntax of Python loops along with else clause is as given below :

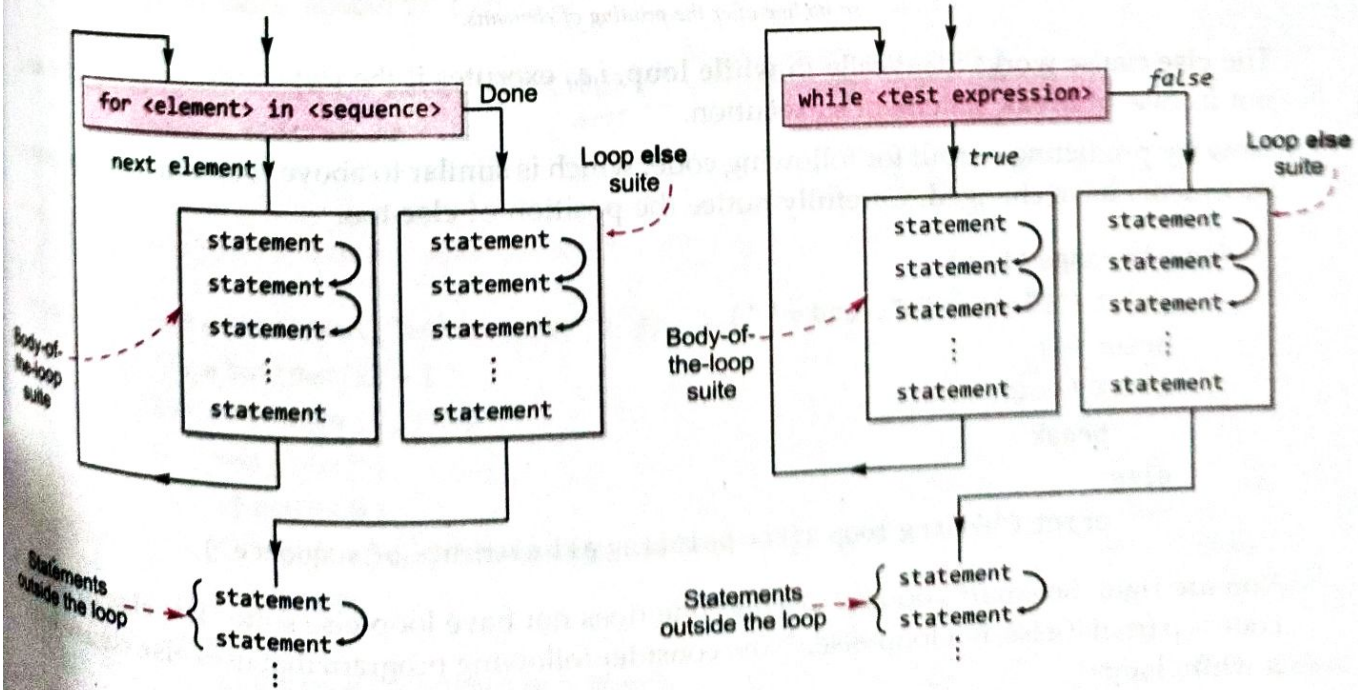
```
for <variable> in <sequence> :
    statement1
    statement2
    :
else :
    statement(s)
```

```
while <test condition> :
    statement1
    statement2
    :
else :
    statement(s)
```

Following figure (Fig. 9.8) illustrates the difference in working of these loops in the absence and presence of loop else clauses.



(a) Control flow in Python loops in the absence of loop-else change



(b) Control flow in Python loops in the presence of loop-else clause.

Figure 9.8 Working of Python Loops with or without else clauses.

Let us understand the working of the Python Loops with or without **else** clause with the help of code examples.

NOTE

The loop-else suite executes only in the case of normal termination of loop.

```
for a in range (1, 4) :
    print ("Element is", end = ' ')
    print (a)
else :
    print ("Ending loop after printing all elements of sequence")
```

The above will give the following output :

The output produced by for loop. Notice for loop executed for three values of ***a*** *– 1, 2, 3*

```
Element is 1
Element is 2
Element is 3
```

This line is printed because the ***else clause of given for loop*** *executed when the for loop was terminating , normally.*

```
Ending loop after printing all elements of sequence.
```

Now consider the same loop as above but with a **break** statement :

```
for a in range (1, 4) :
    if a % 2 == 0 :
        break
    print ("Element is", end = ' ')
    print (a)
else :
    print ("Ending loop after printing all elements of sequence")
```

The break statement will execute when ***a*** *%2 is zero i.e., when* ***a*** *has the value as* ***2***

Now the output is

```
Element is 1
```

Notice that for a = 2, the ***break*** *got executed and loop terminated. Hence just one element got printed (for a = 1). As break terminated the loop, the* ***else clause*** *also did not execute, so no line after the printing of elements.*

The **else** clause works identically in **while loop**, *i.e.*, executes if the test-condition goes *false* and not in case of **break** statement's execution.

Now try predicting output for following code, which is similar to above code but order of some statements have changed. Carefully notice the position of **else** too.

```
for a in range (1, 4) :
    print ("Element is", end = ' ')
    print (a)
    if a % 2 == 0 :
        break
    else:
        print ("Ending loop after printing all elements of sequence")
```

You are right. So smart you are. ☺ This code does not have loop-else suite. The **else** in above code is part of if-else, not loop-else. Now consider following program that uses **else clause** with a **while loop**.

9.18 *Program to input some numbers repeatedly and print their sum. The program ends when the users say no more to enter (normal termination) or program aborts when the number entered is less than zero.*

```
count = sum = 0
ans = 'y'
while ans == 'y' :
    num = int( input ( "Enter number :" ) )
    if num < 0 :
        print ("Number entered is below zero. Aborting!")
        break
    sum = sum + num
    count = count + 1
    ans = input( "Want to enter more numbers? (y/n).." )
else :
    print ("You entered" , count, "numbers so far.")
print ("Sum of numbers entered is", sum)
```

Body-of-the loop suite

loop-else suite

This statement is outside the loop

```
Enter number: 3
Want to enter more numbers? (y/n)..y
Enter number: 4
Want to enter more numbers? (y/n)..y
Enter number : 5
Want to enter more numbers? (y/n)..n
You entered 3 numbers so far.
Sum of numbers entered is 12
========== RESTART ==========
Enter number : 2
Want to enter more numbers? (y/n)..y
Enter number : -3
Number entered is below zero. Aborting!
Sum of numbers entered is 2
```

Result of else clause. Loop terminated normally

No execution of else clause. Loop terminated because of break statement

9.19 *Program to input a number and test if it is a prime number.*

```
num = int(input("Enter number :"))
lim = int(num/2) + 1
for i in range (2, lim) :
    rem = num % i
    if rem == 0 :
        print(num, "is not a prime number")
        break
else:
    print(num, "is a prime number")
```

```
Enter number :7
7 is a prime number
==========
Enter number :49
49 is not a prime number
```

Sometimes the loop is not entered at all, reason being empty sequence in a *for* loop or test-condition's result as *false* before entering in the while loop. In these cases, the body of the loop is not executed but loop-else clause will still be executed.

Consider following code examples illustrating the same.

```
while ( 3 > 4) :
    print ("in the loop")
else :
    print ("exiting from while loop")
```

*Body of the loop will not be executed (condition is false) but **else clause** will*

The above code will print the result of execution of while loop's else clause, *i.e.*,

```
exiting from while loop
```

Consider another loop – this time a **for** loop with empty sequence.

```
for a in range(10,1):
    print ("in the loop")
else:
    print ("exiting from for loop")
```

*Body of the loop will not be executed (empty sequence) but **else clause** will*

The above code will print the result of execution of for loop's else clause, *i.e.*,

```
exiting from for loop
```

NOTE

The else clause of Python loops works in the same manner. That is, it gets executed when the loop is terminating normally – after the last element of sequence in **for loop**. and when the test-condition becomes *false* in **while loop**.

9.7.6 Nested Loops

A loop may contain another loop in its body. This form of a loop is called **nested loop**. But in a nested loop, the inner loop must terminate before the outer loop. The following is an example of a nested loop, where a **for loop** is nested within another **for loop**.

To see Nested loop in action

Scan QR Code

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

end = ' ' at the end to cause printing at same line

to cause printing from next line.

The output produced by above code is :

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

The inner for loop is executed for each value of *i*. The variable *i* takes values 1, 2, 3 and 4. The inner loop is executed once for ***i* = 1** according to sequence in inner loop ***range (1, i)*** (because for *i* as 1, there is just one element 1 in *range (1, 1)* thus *j* iterates for one value, *i.e.*, 1), twice for *i* = 2 (two elements in sequence *range(1, i)*, thrice for *i* = 3 and four times for *i* = 4.

While working with nested loops, you need to understand one thing and that is, the value of outer loop variable will change only after the inner loop is completely finished (or interrupted).

To understand this, consider the following code fragment :

```
for outer in range (5, 10, 4) :
    for inner in range(1, outer, 2) :
        print (outer, inner)
```

The above code will produce the output as :

```
5 1
5 3
9 1
9 3
9 5
9 7
```

The output produced when the outer =5

The output produced when the outer = 9

See the explanation below :

	5	1
change in outer's value →	5	3 ← *inner loop got over after value 3's iteration*
	9	1
change in outer's value →	9	3
	9	5 ← *inner loop got over after value 7's iteration*
	9	7

Check Point 9.3

1. What are iteration statements ? Name the iteration statements provided by Python.
2. What are the two categories of Python loops ?
3. What is the use of range() function ? What would range(3, 13) return ?
4. What is the output of the following code fragment ?

```
for a in "abcde" :
    print (a,' +', end = ' ')
```

5. What is the output of the following code fragment ?

```
for i in range(0, 10) :
    pass
print (i)
```

6. Why does "Hello" not print even once ?

```
for i in range(10,1) :
    print ("Hello")
```

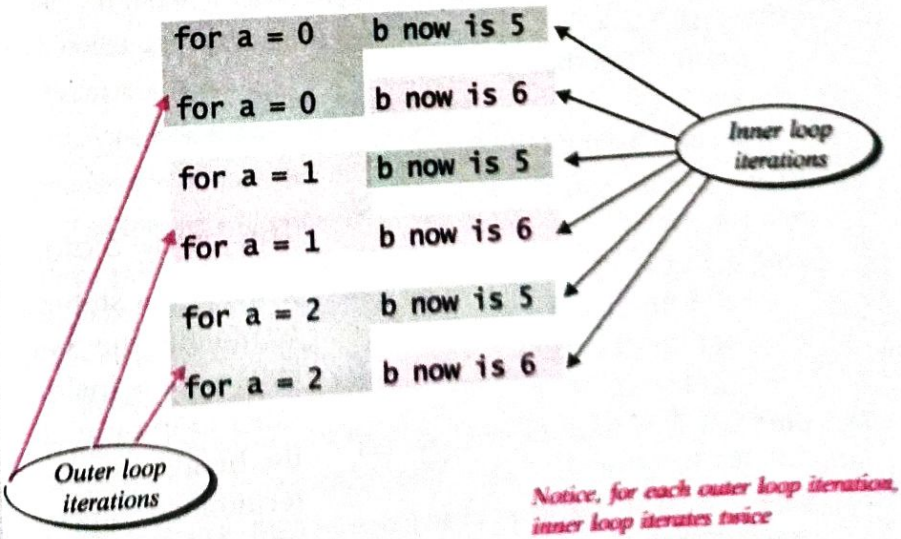
7. What is the output of following code ?

```
for a in range (2, 7) :
    for b in range(1, a) :
        print (' #', end = ' ')
    print ( )
```

Let us understand how the control moves in a nested loop with the help of one more example :

```
for a in range(3) :
    for b in range(5,7) :
        print ("for a =", a, "b now is", b)
```

The above nested loop will produce following output :



Notice, for each outer loop iteration, inner loop iterates twice

For the outer loop, firstly *a* takes the value as 0 and for $a = 0$, inner loop iterates as it is part of outer for's loop-body.

- for $a = 0$, the inner loop will iterate twice for values $b = 5$ and $b = 6$
- hence the first two lines of output given above are produced for the first iteration of outer loop ($a = 0$; which involves two iterations of inner loop)
- when *a* takes the next value in sequence, the inner will again iterate two times with values 5 and 6 for *b*.

Thus we see, that for each iteration of outer loop, inner loop iterates twice for values $b = 5$ and $b = 6$.

Consider some more examples and try understanding their functioning based on above lines.

```
for a in range(3) :
    for b in range(5,7) :
        print ("* ", end = ' ')
    print ()
```

The output produced would be :

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

Consider another nested loop :

```
for a in range(3) :
    for b in range(5,8) :
        print ("* ", end = ' ')
    print ()
```

notice end = ' ' at the end of print

The output produced would be :

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

The break Statement in a Nested Loop

If the break statement appears in a nested loop, then it will terminate the very loop it is in. That is, if the break statement is inside the inner loop then it will terminate the inner loop only and the outer loop will continue as it as. If the break statement is in outer loop, then outer loop gets terminated. Since inner loop is part of outer loop's body it will also not execute just like other statement of outer loop's body.

Check Point 9.3 (contd...)

8. Write a for loop that displays the even numbers from 51 to 60.
9. Suggest a situation where an empty loop is useful.
10. What is the similarity and difference between for and while loop ?
11. Why is while loop called an entry controlled loop ?
12. If the test-expression of a while loop evaluates to false in the beginning of the loop, even before entering in to the loop :
    (*a*) how many times is the loop executed?
    (*b*) how many times is the loop-else clause executed ?
13. What will be the output of following code:

```
while (6 + 2 > 8) :
    print ("Gotcha!")
else:
    print ("Going out!")
```

14. What is the output produced by following loop ?

```
for a in (2, 7) :
    for b in (1, a) :
        print (b, end = ' ')
        print ()
```

15. What is the significance of break and continue statements?
16. Can a single break statement in an inner loop terminate all the nested loops ?

Following program illustrates this. Notice that **break** will terminate the inner loop only while the outer loop will progress as per its routine.

Program that searches for prime numbers from 15 through 25.

Program 9.20

```
for num in range(15,25):
    for i in range(2,num):
        if num % i == 0:                #to determine factors
            j = num/i
            print ("Found a factor( ", i ," ) for", num)
            break                       # need not continue - factor found
                                        # else part of the inner loop
    else:
        print (num, "is a prime number")
```

This break will terminate the inner for loop only

```
Found a factor( 3 ) for 15
Found a factor( 2 ) for 16
17 is a prime number
Found a factor( 2 ) for 18
19 is a prime number
Found a factor( 2 ) for 20
Found a factor( 3 ) for 21
Found a factor( 2 ) for 22
23 is a prime number
Found a factor( 2 ) for 24
```

More examples of nested loops and other simple loops you'll find in solved problems given at the end of this chapter.

PYTHON LOOPS — Progress In Python 9.2

This 'Progress in Python' session works on the objective of Python loops practice.

>>>❖<<<

LET US REVISE

- *Statements are the instructions given to the computer to perform any kind of action.*
- *Python statements can be on one of these types : empty statement, single statement and compound statement.*
- *An empty statement is the statement that does nothing. Python offers* ***pass*** *statement as empty statement.*
- *Single executable statement forms a simple statement.*
- *A compound statement represents a group of statements executed as a unit.*
- *Every compound statement of Python has a header and an indented body below the header.*
- *Some examples of compound statements are : if statement, while statement etc.*
- *The flow of control in a program can be in three ways : sequentially (the sequence construct), selectively (the selection construct), and iteratively (the iteration construct).*
- *The sequence construct means statements get executed sequentially.*
- *The selection construct means the execution of statement(s) depending upon a condition-test.*
- *The iteration (looping) constructs mean repetition of a set-of-statements depending upon a condition-test.*
- *Python provides one selection statement if in many forms – if..else and if..elif..else.*
- *The if..else statement tests an expression and depending upon its truth value one of the two sets-of-action is executed.*
- *The if-else statement can be nested also i.e., an if statement can have another if statement.*
- *A sequence is a succession of values bound together by a single name. Some Python sequences are strings, lists and tuples.*