

4

Conditional and Iterative Statements

In This Chapter

- | | |
|---|--|
| <ul style="list-style-type: none">4.1 Introduction4.2 Types of Statements in Python4.3 Statement Flow Control4.4 Program Logic Development Tools | <ul style="list-style-type: none">4.5 The if Statements of Python4.6 Repetition of Tasks — A Necessity4.7 The Range() Function4.8 Iteration / Looping Statements |
|---|--|

4.1 INTRODUCTION

Generally a program executes its statements from beginning to end. But not many programs execute all their statements in strict order from beginning to end. Programs, depending upon the need, can choose to execute one of the available alternatives or even repeat a set of statements. To perform their manipulative miracles, programs need tools for performing repetitive actions and for making decisions. Python, of course, provides such tools by providing statements to attain so. Such statements are called *program control statements*. This chapter discusses such statements in details. Firstly, selection statement *if* and later iteration statements *for* and *while* are discussed.

This chapter also discusses some jump statements of Python which are *break* and *continue*.

4.2 TYPES OF STATEMENTS IN PYTHON

Statements are the instructions given to the computer to perform any kind of action, be it *data movements*, and be it *making decisions* or be it *repeating actions*. Statements form the smallest executable unit within a Python program. Python statements can belong to one of the following *three types*:

- ⇒ Empty Statement
- ⇒ Simple Statement (Single statement)
- ⇒ Compound Statement

1. Empty Statement

The simplest statement is the *empty statement* i.e., a statement which does nothing. In Python an empty statement is **pass** statement. It takes the following form :

```
pass
```

Wherever Python encounters a **pass** statement, Python does nothing and moves to next statement in the flow of control.

A **pass** statement is useful in those instances where the syntax of the language requires the presence of a statement but where the logic of the program does not. We will see it in loops and their bodies.

NOTE

The **pass** statement of Python is a *do nothing* statement i.e., empty statement or null operation statement.

2. Simple Statement

Any single executable statement is a simple statement in Python. For example, following is a simple statement in Python :

```
name = input( "Your name" )
```

Another example of simple statement is :

```
print( name )      # print function called
```

As you can make out that simple statements are single line statements.

3. Compound Statement

A compound statement represents a group of statements executed as a unit. The compound statements of Python are written in a specific pattern as shown below :

```
<compound statement header> :  
    <indented body containing multiple simple and/or compound statements>
```

That is, a compound statement has :

- ⇒ a **header line** which begins with a keyword and ends with a colon.
- ⇒ a **body** consisting of one or more Python statements, each indented¹ inside the header line. All statements in the body are at the same level of indentation.

NOTE

A *compound statement* in Python has a *header* ending with a colon (:) and a *body* containing a sequence of statements at the same level of indentation.

You'll learn about some other compound statements (*if*, *for*, *while*) in this chapter.

1. Conventionally, Python uses four spaces to move to next level of indentation.

4.3 STATEMENT FLOW CONTROL

In a program, statements may be executed sequentially, selectively or iteratively. Every programming language provides constructs to support sequence, selection or iteration. Let us discuss what is meant by sequence, selection or iteration constructs.

Sequence

The sequence construct means the statements are being executed sequentially. This represents the default flow of statement (see Fig. 4.1).

Every Python program begins with the first statement of program. Each statement in turn is executed (sequence construct). When the final statement of program is executed, the program is done. **Sequence** refers to the normal flow of control in a program and is the simplest one.

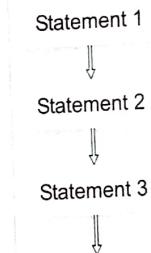


Figure 4.1 The sequence construct.

Selection

The selection construct means the execution of statement(s) depending upon a condition-test. If a condition evaluates to *True*, a course-of-action (a set of statements) is followed otherwise another course-of-action (a different set of statements) if followed. This construct (*selection construct*) is also called *decision construct* because it helps in making decision about which set-of-statements is to be executed.

Following figure (Fig. 4.2) explains selection construct.

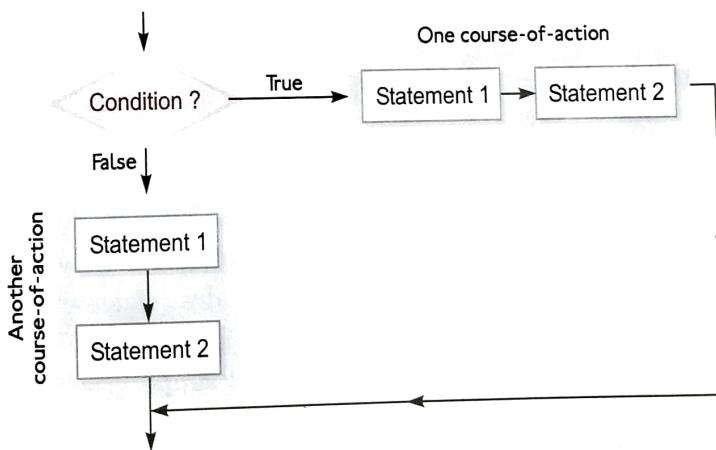


Figure 4.2 The selection construct.

You apply decision-making or selection in your real life so many times e.g., if the traffic signal light is **red**, then **stop**; if the traffic signal light is **yellow** then **wait**; and if the signal light is **green** then **go**.

You can think of many such real life examples of selection/decision-making.

Iteration (Looping)

The iteration constructs mean *repetition of a set-of-statements* depending upon a *condition-test*. Till the time a condition is *True* (or *False* depending upon the loop), a set-of-statements are repeated again and again. As soon as the condition becomes *False* (or *True*), the repetition stops. The iteration construct is also called *looping construct*.

The adjacent figure (Fig. 4.3) illustrates an iteration construct.

The set-of-statements that are repeated again and again is called the *body of the loop*. The condition on which the execution or exit of the loop depends is called *the exit condition* or *test-condition*.

You can find many examples of iteration or looping around you. For instance, you often see your mother cook *chapatis* or *dosas* or *appams* for you.

Let's see, what she does for it :

- (i) put rolled chapati or dosa batter on flat pan or tawa
- (ii) turn it 2-3 times
- (iii) once done take it off.

Repeat the above process (steps (i), (ii), (iii)) for next chapati/dosa/appam. This is looping or iteration.

You can find numerous other examples of repetitive work in your real life e.g., washing clothes; colouring in colouring book etc. etc.

Check Point 4.1

1. What is a statement ? How many types of statements are there in Python ?
2. What is the significance of a *pass* statement?
3. What is a compound statement ? Give example of a compound statement.
4. What are the three constructs that govern statement flow ?
5. What is the need for selection and looping constructs ?

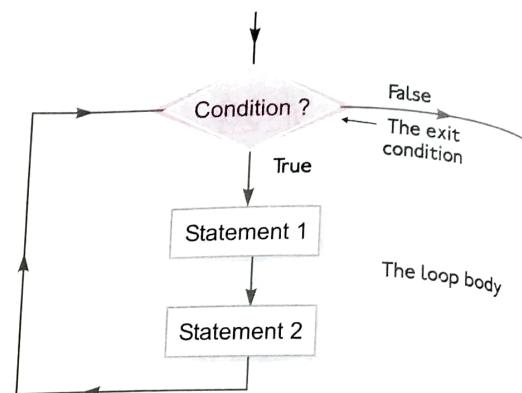


Figure 4.3 The iteration construct.

Every programming language must support these three types of constructs as the sequential program execution (the default mode) is inadequate to the problems we must solve. Python also provides statements that support these constructs. Coming sections discuss these Python statements – *if* that supports selection and *for* and *while* that support iteration. Using these statements you can create programs as per your need.

But before we talk about these statements, you should know basic tools that will help you decide about the logic to solve a given problem i.e., the algorithm. For this, there are multiple tools available. In the following section, we are going to talk about three such tools – *Flowcharts*, *decision trees* and *pseudocode*.

4.4 PROGRAM LOGIC DEVELOPMENT TOOLS

Before developing the solution of a problem in terms of a program, you should read and order of these subtasks. In other words, you figure out the algorithm for the solution of a given problem and represent it through various logic development tools.

An **algorithm** is a step-by-step procedure (well-defined instructions) to solve a given problem. For instance, the

1. **Input** first number.
2. **Input** second number.
3. **Divide** first number with second number and store the **remainder** as third number.
4. **Display** the result (the *third number*).

ALGORITHM

An **algorithm** is a step-by-step procedure (well-defined instructions) to solve a given problem.

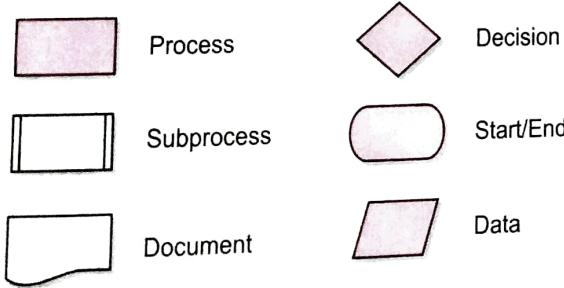
An algorithm is a set of ordered and finite steps (the subtasks) to solve a given problem. Consider another example that extends the above problem – *using the same logic determine if the first number is divisible by second number or not.*

1. **Input** first number.
2. **Input** second number.
3. **Divide** first number with second number and store the **remainder** in *third number*.
4. **Check** if the *third number* is 0.
 - (a) If **Yes**, Display ‘the first number IS divisible by second number’.
 - (b) If **No**, Display ‘the first number IS NOT divisible by second number’.

As you can see that the subtasks in above algorithms are shown in bold letters and there is a proper order of carrying out these steps/subtasks. Algorithms are commonly written out with tools like *pseudocode*, *flow charts*, or *decision trees* and *tables*. Let us talk about these tools one by one.

4.4.1 Flowcharts

A flowchart is a graphical representation of an algorithm. A flowchart shows different subtasks with different symbols. Following figure (4.4) shows commonly used flowchart symbols.



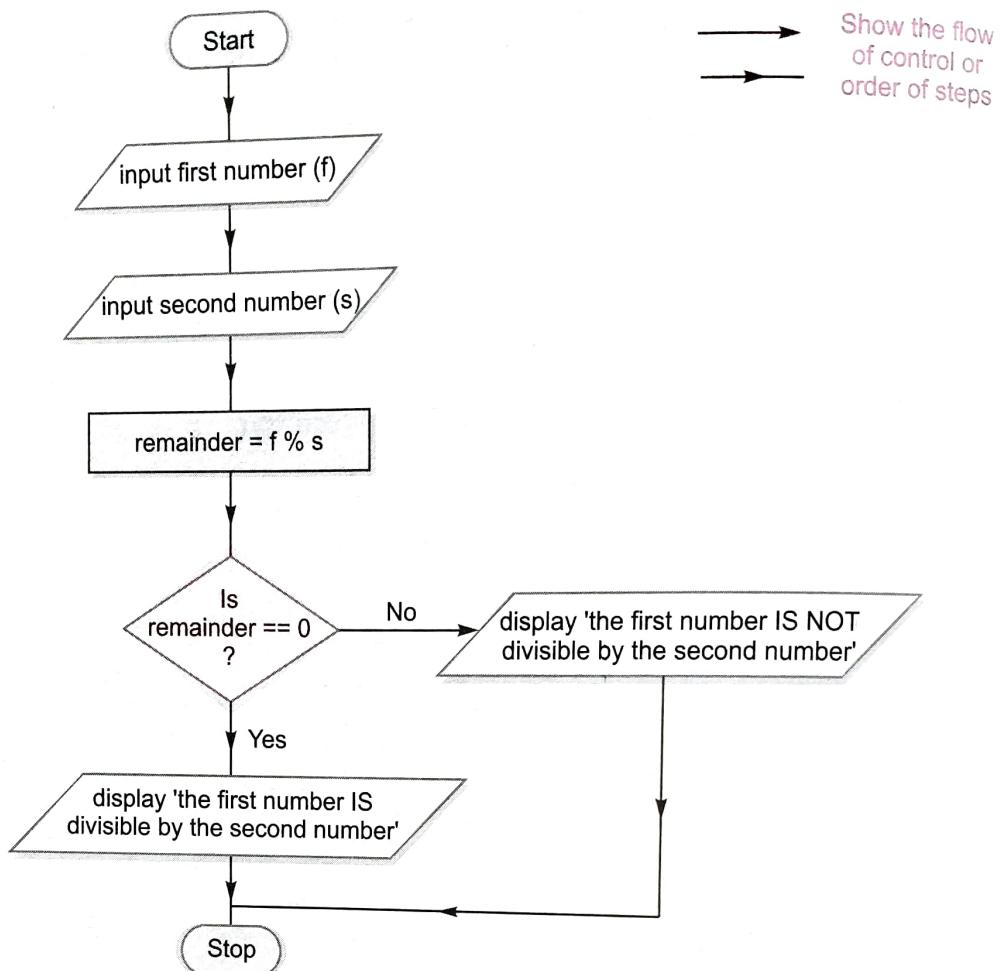
FLOWCHART

A **flowchart** is a graphical representation of steps an algorithm to solve a given problem.

Figure 4.4 Flowchart symbols.

- ⇒ Use **Data** symbol for Input/Output (I/O) operation (taking input and showing output).
- ⇒ Use **Process** symbol for any type of computation and internal operations like initialization, calculation etc.
- ⇒ Use **Subprocess** symbol to invoke a procedure written already.

The flowchart for the above algorithm (*determine if the first number is divisible by second number or not*) will be :



→ Show the flow of control or order of steps
 → Show the flow of control or order of steps

With flowcharts, a programmer can pictorially view the subtasks of an algorithm and the order their execution.

4.4.2 Pseudocode

Pseudocode is an informal language that helps programmers describe steps of a program's solution without using any programming language syntax. Pseudocode is a "text-based" detail (algorithmic) design tool. A pseudocode creates an outline or a rough draft of a program that gives the idea of how the algorithm works and how the control flows from one step to another.

For example, consider the following pseudocode :

```

If student's marks is greater than or equal to 50
  display "passed"
else
  display "failed"
  
```

NOTE

Pseudocode is an informal way of describing the steps of a program's solution without using any strict programming language syntax or underlying technology considerations.

The above pseudocode gives you an idea how a program determines whether a student has *passed* or *failed* comparing the marks of the student.

Consider the same algorithm that we talked above (*determine if the first number is divisible by second number or not*). The pseudocode of this algorithm can be like :

```

Input first number in variable firstnum.
Input second number in variable secondnum
remainder = firstnum modulus secondnum
if the remainder is 0                                # modulus to get remainder
    display 'the first number IS divisible by second number'.
else
    display 'the first number IS NOT divisible by second number'.

```

Please note there are no standard rules to write a pseudocode as it is totally informal way of representing an algorithm.

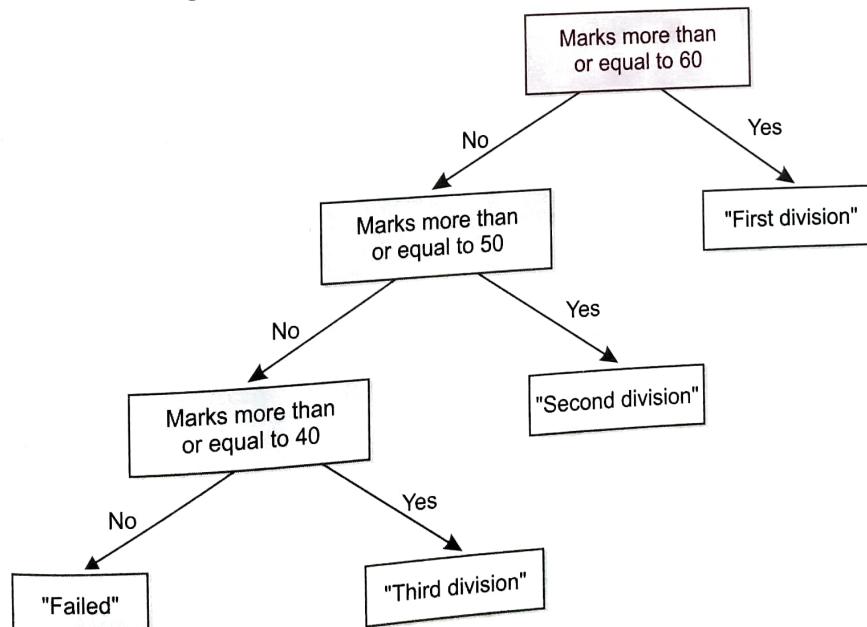
4.4.3 Decision Trees

Decision trees are a way of presenting rules in a hierarchical and sequential structure, where based on the hierarchy of *rules*, certain *outcomes* are predicted.

Let us consider a familiar example. You all know how divisions are determined based on marks, e.g.,

| Marks | Division |
|-----------|----------|
| ≥ 60 | First |
| 50 – 60 | Second |
| 40 – 50 | Third |
| < 40 | Failed |

Now consider following diagram :



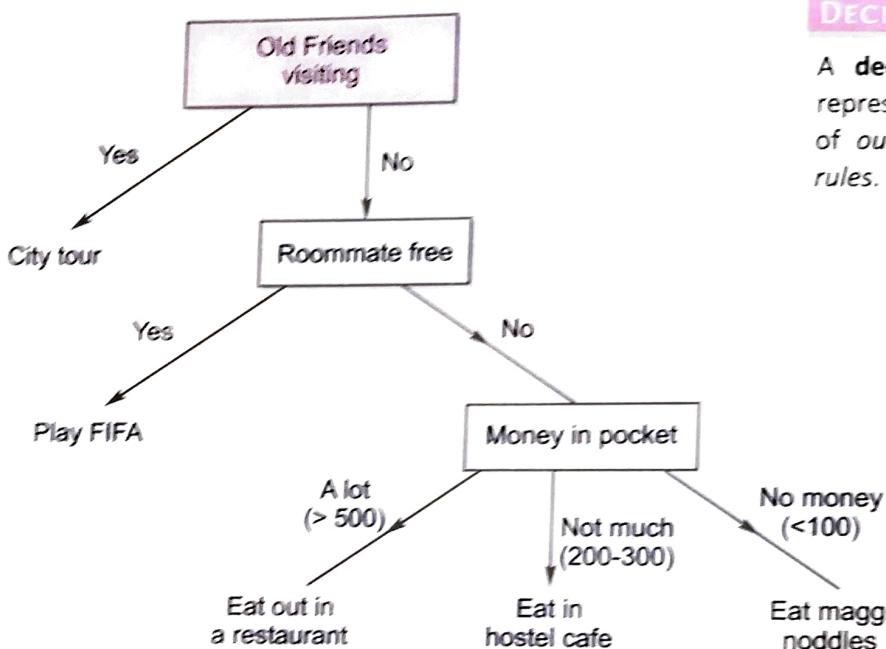
Does it make sense ? Isn't it more readable ?

A Decision tree is an excellent tool to represent a hierarchical structure of outcomes based on certain rules. In other words, a decision uses a tree like structure that can represent a number of possible rules based *decision paths* and an outcome for each path.

Let us imagine another hypothetical situation. Aryan has taken admission in a new school in another city and is staying in the school hostel. This Sunday Aryan's old friends may visit him. So, Aryan has decided to spend his Sunday as per this plan :

If his friends visit him, he will go with them for *city tour*. If they don't come, he will *play FIFA* with his roommate if his roommate is free, otherwise, depending upon money in his pocket, he may *eat out in a restaurant* or from small *Cafe in hostel premises* or will simply *eat maggi noodles*.

Now this situation can also be depicted through a decision tree :



DECISION TREE

A **decision tree** is a tool to represent a hierarchical structure of *outcomes* based on certain rules.

So the rules for spending Sunday as per this decision tree can be written as :

If old friends are visiting, go for city tour

Or

If old friends are not visiting and roommate is free then play FIFA

Or

If old friends are not visiting and roommate is not free and has a lot of money (say more than ₹ 500) then eat out in a restaurant

Or

If old friends are not visiting and roommate is not free and has not much money (roughly ₹ 200-300) then eat in the hostel cafe

Or

If old friends are not visiting and roommate is not free and has no money (less than ₹ 100) then eat maggi noodles.

Decision trees are majorly used in learning algorithms of modern programming languages. After learning about these logic development tools, let us learn about Python's selection and iterative statements.

4.5 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 3 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.

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

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

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.

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.

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.

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 3 under section 3.4.3A*. I'll advise you to have a look at section 3.4.3A 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 $false_{\text{val}}$ and *False* when *x* is $true_{\text{val}}$.

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.

4.5.2 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 :             ← colons after both if and else
    statement
    [statements]
```

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 *ifs*.

Following figure (Fig. 4.5) illustrates *if* and *if-else* constructs of Python.

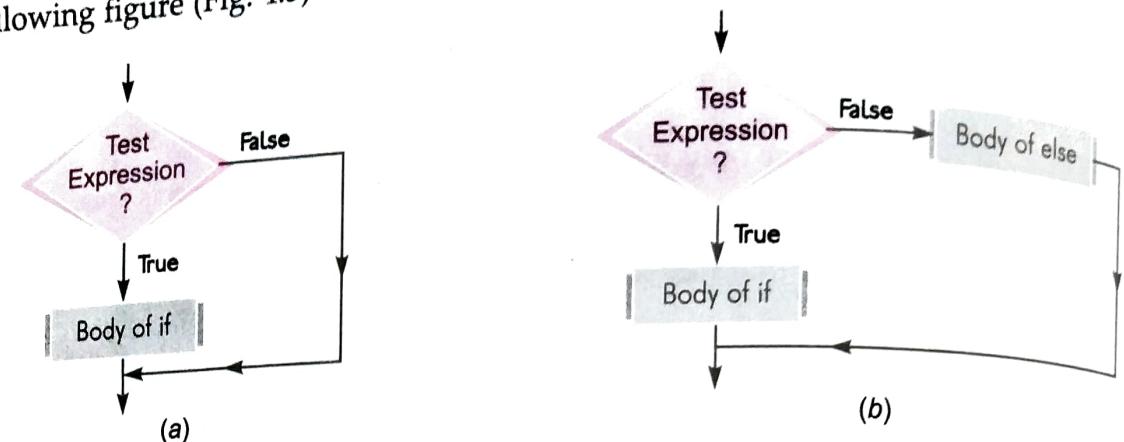


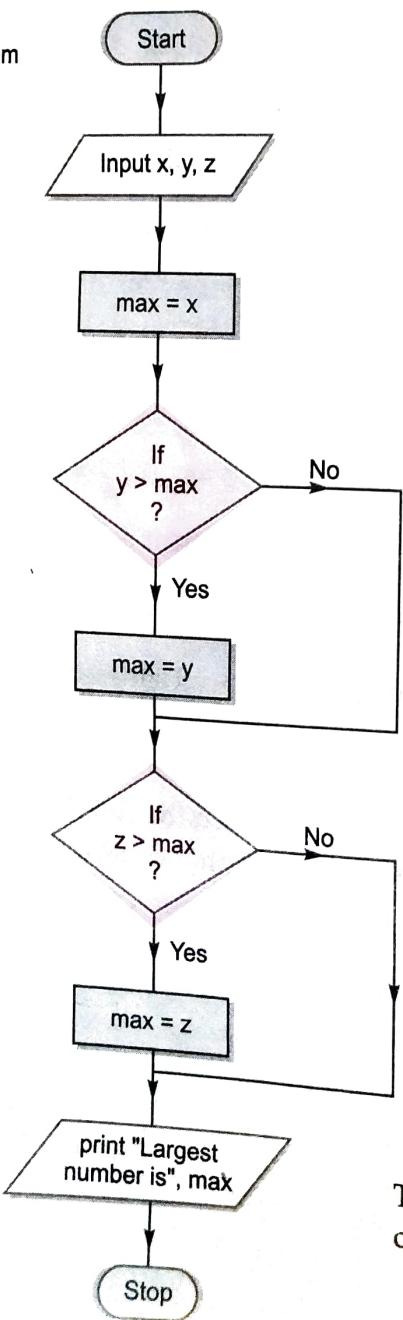
Figure 4.5 (a) The *if* statement's operation (b) The *if-else* statement's operation.

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

P

4.1

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



Input three numbers x, y, z

max = x #first number

**If second number y is more than max then
max = y**

**If third number z is more than max then
max = z**

Display max as the largest number

Pseudocode

Code in Python

`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`

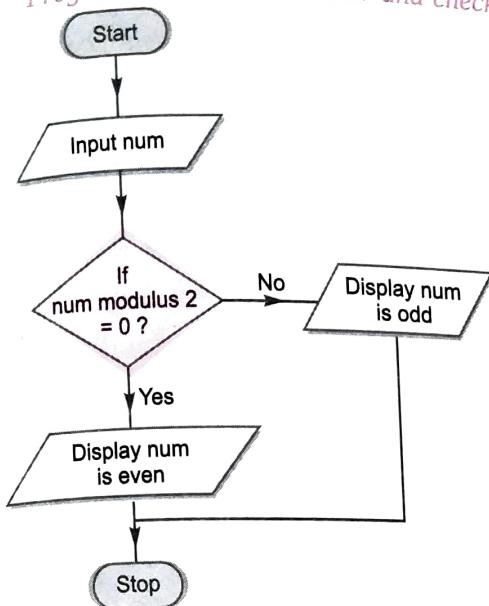
`print ("Largest number is", max)`

The sample run of above code is as shown here.

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

Program 4.2

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

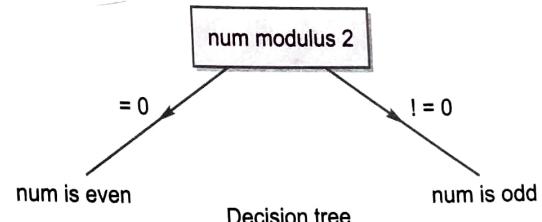


Flowchart

Input num

If num modulus 2 is equal to 0 then
display num is even
else
display num is odd

Pseudocode



Decision tree

Decision Tree

Code in Python

```

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

Sample run of above program
is as shown below :

Enter an integer : 8
8 is EVEN number.

Program 4.3

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)

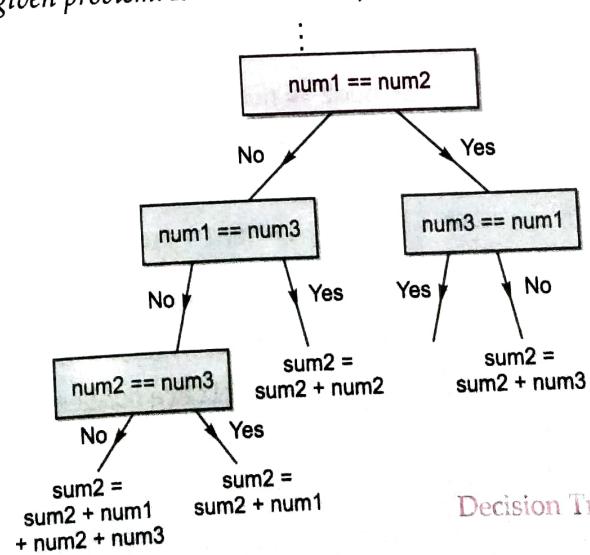
Input of numbers 4, 4, 4 will give two sums as 12 and 0 (all 4's ignored for second sum)

Here we are giving two different ways of solving the given problem. 1st alternative's pseudocode is given and 2nd alternative's decision tree is given :

```

sum1 = 0, sum2 = 0
input three numbers as num1, num2, num3
sum1 = num1 + num2 + num3
if num1 not equal to num2 or num3
    sum2 = sum2 + num1
if num2 not equal to num1 or num3
    sum2 = sum2 + num2
if num3 not equal to num1 or num2
    sum2 = sum2 + num3
display sum1 and sum2
  
```

Pseudocode



Decision Tree

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)

```

NOTE

Please note that the pseudocode and decision trees are given here just for understanding purpose, to make you understand that how easily you get the logic behind these solutions using these tools.

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 4.4

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)
```

Program 4.5

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 the program
is as shown below :

```
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
```

Sample run of the 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
```

P 4.6 Program

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)
```

Two sample runs of above program is as shown below :

```
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.

4.5.3 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 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 4.3 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* and *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 :

```
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.

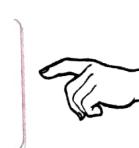
This block will be executed when both the if's condition (i.e., runs >=100) and elif condition (i.e., runs>=50) are false.

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

4.7

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)

```

Some sample runs of above program are being given below :

```

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 4.3 using **if-elif-else** statements.

4.5.4 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 4.3. Isn't that superb? ;)

Following example programs illustrate the use of nested ifs.

P rogram 4.8

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 : 19 9 90

```

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

P rogram 4.9

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.

```

Program 4.10 Program to calculate and print roots of a quadratic equation : $ax^2 + bx + c = 0$ ($a \neq 0$).

```

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.

Check Point

4.2

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

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

- 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")
```

- 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")
```

- 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")
```

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.

DECISION

CONSTRUCT if

Progress In Python 4.1

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

4.6 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

????

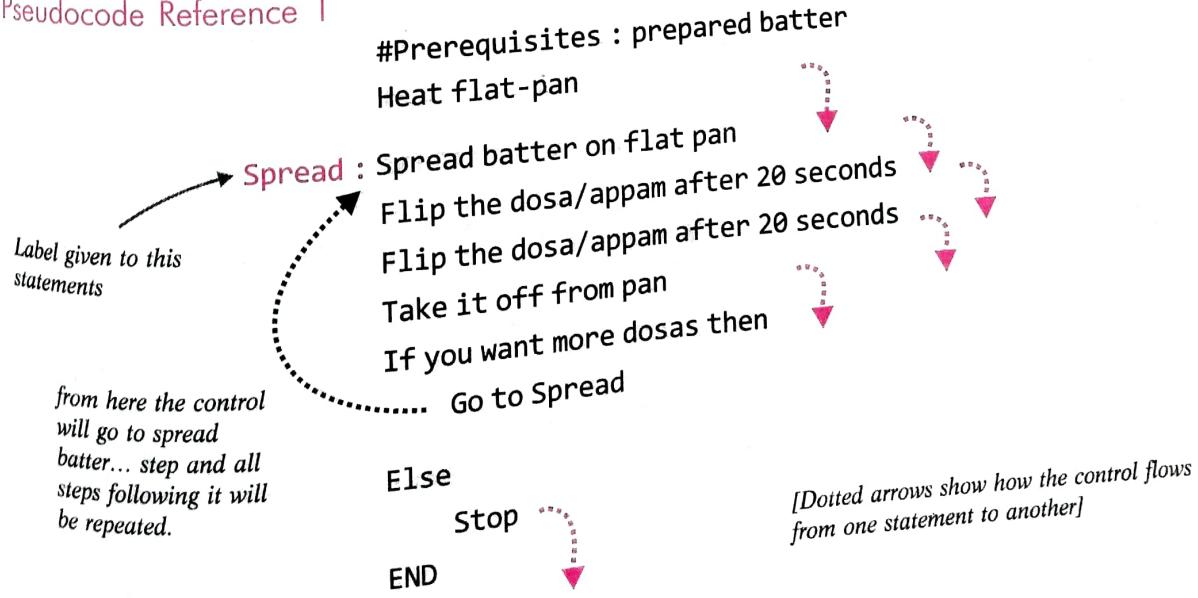
else

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



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. 4.6). There are some advance sequence types also but those are beyond the scope of our discussion.

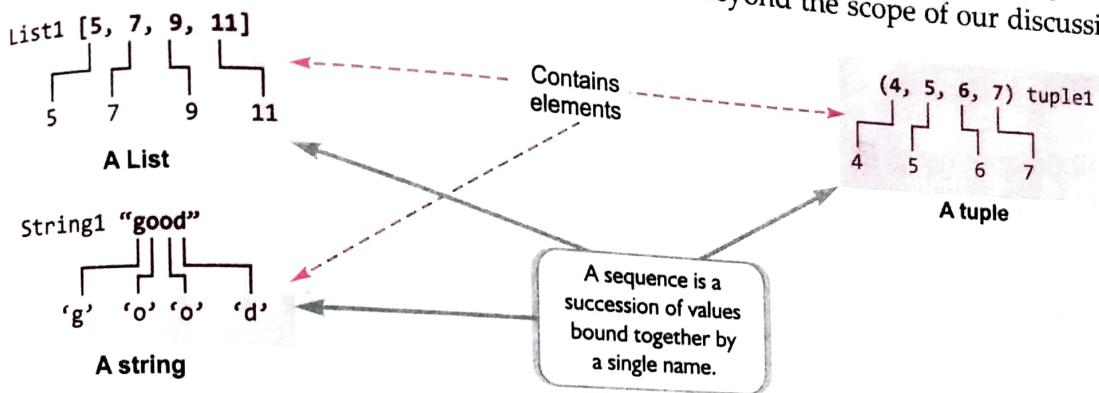


Figure 4.6 Some common sequence types

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 $l, l+1, l+2 \dots (u-1)$ (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)` *the default step-value in values will be +1*

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)` *default step-value = +1*

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)` *default step-value = +1*

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(1, u, s)` # 1, u and s are integers

will produce a list having values as $l, l+s, l+2s, \dots \leq u-1$.

`range(0, 10, 2)` *step-value = +2*

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

`range(5, 0, -1)` *step-value = -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 |
|-------------------------------|------------------------------|
| <code>range(10)</code> | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 |
| <code>range(5, 10)</code> | 5, 6, 7, 8, 9 |
| <code>range(3, 7)</code> | 3, 4, 5, 6 |
| <code>range(5, 15, 3)</code> | 5, 8, 11, 14 |
| <code>range(9, 3, -1)</code> | 9, 8, 7, 6, 5, 4 |
| <code>range(10, 1, -2)</code> | 10, 8, 6, 4, 2 |

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]` *This expression will test if value 3 is contained in the given sequence*

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.

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.

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.

NOTE

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

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: ← if the given string is a part of line
    print (string, "is part of", line.)
else:
    print (string, "is not contained in", line.)
```

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.

4.8 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.
- ⇒ **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 a conditional loop.

4.8.1 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. 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 :

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.

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

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.

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

NOTE

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

for a in [1, 4, 7]:

print (a)
print (a * a)

body of the
loop

(i)

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 square 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
81

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 $[3, 4, 5, \dots, 16, 17]$ 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.



4.11 Program to print table of a number, say 5.

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

The above code will print the output as shown here :

NOTE

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

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*.

$$\begin{aligned}5 \times 1 &= 5 \\5 \times 2 &= 10 \\5 \times 3 &= 15 \\5 \times 4 &= 20 \\5 \times 5 &= 25 \\5 \times 6 &= 30 \\5 \times 7 &= 35 \\5 \times 8 &= 40 \\5 \times 9 &= 45 \\5 \times 10 &= 50\end{aligned}$$

Program

4.12 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)
```

The output produced by above program will be :

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

Program

4.13 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)
```

The output produced by above program is :

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)*.

4.8.2 The while Loop

A **while loop** is a conditional loop that will repeat the instructions within itself as long as a conditional remains *true* (Boolean *True* or truth value *true_{val}*).

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.

NOTE

The loop variable contains the highest value of the list after the *for loop* is over. Program 4.13 highlights this fact.

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 :

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.

while $a > 0$:

```
print ("hello", a)
a = a - 3
```

body of the loop

[a has value 5 before entering into while loop]

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

$$5 > 0 = \text{True}$$

hence loop's body will get executed

print ("hello", 5) -----> hello 5

$$a = 5 - 3 = 2. \quad (a \text{ becomes } 2 \text{ now})$$

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

$$2 > 0 = \text{True}$$

loop body will get executed

print ("hello", 2) -----> hello 2

$$a = 2 - 3 = -1 \quad (a \text{ becomes } -1 \text{ now})$$

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

$$-1 > 0 = \text{False}$$

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.)

P 4.14 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)

```

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

```

P 4.15 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

```

4.8.3 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.

4.5.4 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.

4.5.4.1 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. 4.7) explains the working of a *break* statement :

Statements 4 is the first statement after the loop

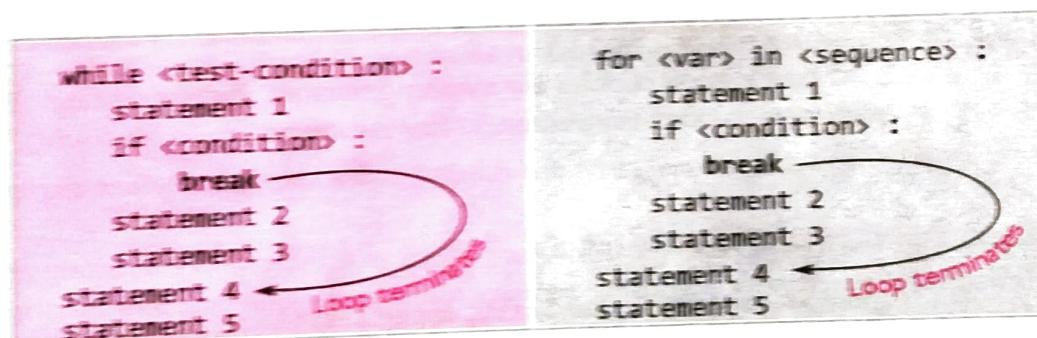


Figure 4.7 The working of a *break* statement.

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!")
```

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.

NOTE

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

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.



4.16

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.

- ❖ 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.

i.e., whether the loop terminated after 5 iterations

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 not ctr < 5 : ← If the loop-condition is still true outside the loop
    print ("You lose :(\n The number was", number)
    that means break caused the loop termination.

```

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 :

```

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

```

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

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.

4.8.4B 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.

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.

4.17 Program to illustrate the difference between break and continue statements.

Program

```

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

The loop with 'continue' produces output as :

1
2
4  only the values divisible by 3 are missing as the loop
5 simply moved to next iteration with **continue** when
7 condition $i \% 3$ became true.
8 (Only the iterations with $i \% 3 == 0$ are
10 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.

4.8.5 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. 4.9) illustrates the difference in working of these loops in the absence and presence of loop else clauses.

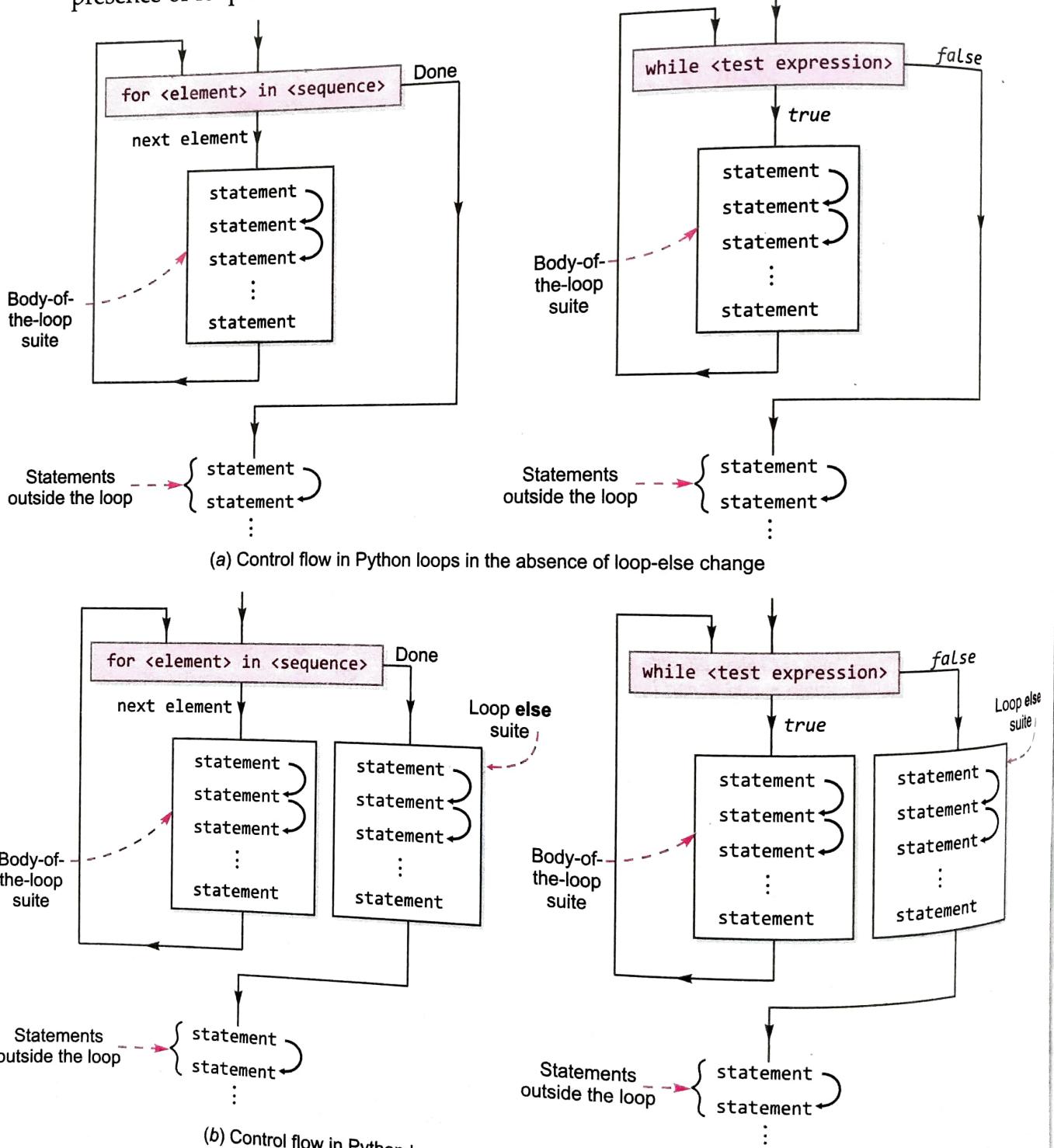


Figure 4.9 Working of Python Loops with or without else clauses.

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

```
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**.

NOTE

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

P 4.18 Program

```

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.           ← Result of else clause. Loop terminated normally
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! ← No execution of else clause. Loop terminated because of break statement
Sum of numbers entered is 2

```

P 4.19 Program

```

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.

4.8.6 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**.

```
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 :

| | |
|-------------------------|---|
| change in outer's value | 5 |
| | 5 |
| change in outer's value | 9 |
| | 9 |
| | 9 |
| | 9 |

1 inner loop got over after value 3's iteration

3 inner loop got over after value 7's iteration

Check Point

4.3

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

for a in "abcde" :

print (a, '+', end = ' ')

- What is the output of the following code fragment ?

for i in range(0, 10) :

pass

print (i)

- Why does "Hello" not print even once ?

for i in range(10,1) :

print ("Hello")

- What is the output of following code ?

for a in range (2, 7) :

for b in range(1, a) :

print (' #', end = ' ')

print ()

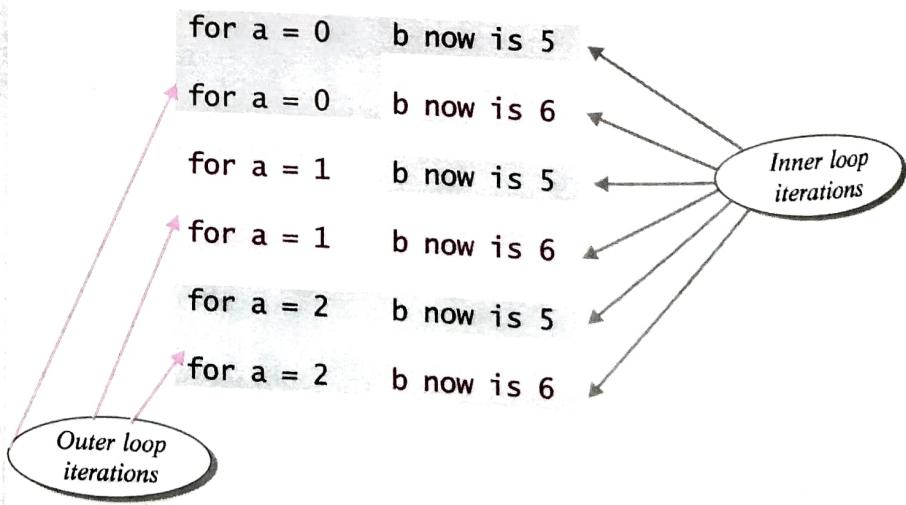
Let us understand how the control moves in a nested loop with the help of one more example :

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 $b = 6$.

Check Point

4.3 (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?

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 is. 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.

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



4.20 Program that searches for prime numbers from 15 through 25.

This break will terminate the inner for loop only

```
program
    for num in range(15,25):
        for i in range(2,num):
            if num % i == 0:
                j = num/i
                print ("Found a factor( ", i , " ) for", num)
                break
            else:
                print (num, "is a prime number")
```

#to determine factors
need not continue - factor found
else part of the inner loop

The output produced by above program will be :

```
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.

This brings us to the end of this chapter. Let us quickly revise what we have learnt so far.



PYTHON LOOPS

Progress In Python 4.2

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

Carefully read the instruction and then carry out the problem. Once through with conceptual practice questions, you can move on to practicing programming problems.

:

:



Please check the practical component-book – **Progress in Computer Science with Python** and fill it there in **PriP 4.2** under Chapter 4 after practically doing it on the computer.

LET US REVISE

- ❖ Statements are the instructions given to the computer to perform any kind of action.
- ❖ Python statements can be 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.
- ❖ The range() function generates a sequence of list type.
- ❖ The statements that allow a set of instructions to be performed repeatedly are iteration statements.
- ❖ Python provides two looping constructs – **for** and **while**. The **for** is a counting loop and **while** is a conditional loop.
- ❖ The while loop is an entry-controlled loop as it has a control over entry in the loop in the form of test condition.
- ❖ Loops in Python can have else clause too. The else clause of a loop is executed in the end of the loop only when loop terminates normally.
- ❖ The **break** statement can terminate a loop immediately and the control passes over to the statement following the statement containing break.
- ❖ In nested loops, a **break** terminates the very loop it appears in.
- ❖ The **continue** statement abandons the current iteration of the loop by skipping over the rest of the statements in the loop-body. It immediately transfers control to the beginning of the next iteration of the loop.

Solved Problems

1. What is a statement ? What is the significance of an empty statement ?

Solution. A statement is an instruction given to the computer to perform any kind of action. An empty statement is useful in situations where the code requires a statement but logic does not. To fill these two requirements simultaneously, empty statement is used. Python offers **pass** statement as an empty statement.

2. If you are asked to label the Python loops as determinable or non-determinable, which label would you give to which loop? Justify your answer.

Solution. The 'for loop' can be labelled as **determinable loop** as number of its iterations can be determined before-hand as the size of the sequence, it is operating upon.

The 'while loop' can be labelled as **non-determinable loop**, as its number of iterations cannot be determined before-hand. Its iterations depend upon the result of a test-condition, which cannot be determined before-hand.

3. There are two types of else clauses in Python. What are these two types of else clause?

Solution. The two types of Python else clauses are :

- (a) else in an if statement (b) else in a loop statement

The *else* clause of an *if* statement is executed when the condition of the *if* statement results into *false*. The *else* clause of a *loop* is executed when the loop is terminating normally i.e., when its test-condition has gone *false* for a *while* loop or when the *for* loop has executed for the last value in sequence.

4. Use the Python range() function to create the following list : [7, 3, -1, -5]

Solution.

`range (7, -6, -4)`

5. Identify and correct the problem with following code :

```
countdown = 10                                    # Count from 10 down to 1
while countdown > 0 :
    print (countdown, end = ' ')
    countdown - 1
print ("Finally.")
```

Solution. It is an infinite loop. The reason being, in the above *while* loop, there is an expression **countdown - 1**, which subtracts 1 from value of **countdown** BUT this statement is not updating the loop variable **countdown**, hence loop variable **countdown** always remains unchanged and hence it is an **infinite loop**.

The solution to above problem will be replacement of **countdown - 1** with following statement in the body of while loop :

`countdown = countdown - 1`

Now the while loop's variable will be updated in every iteration and hence loop will be a finite loop.

6. Following code is meant to be an interactive grade calculation script for an entrance test that converts percentage into a letter grade :

90 and above is A+,
80-90 is A,
60-80 is A-,
and everything below is fail (U know high standards).

The program should prompt the user 100 times for grades. Unfortunately, the program was written by a terrible coder (me :/), so there are numerous bugs in the code. Find all the bugs and fix them.

```
For i : range(1 .. 100)
    grade == float (input( "What's the grade percentage ?" ))
    if grade > 90
        print ("That's an A + !")
```

```

if 80 > grade > 90:
    print (" " "An A is really good! " ")
elif 60 < grade :
    print (You got an A-!)
else grade < 60 :
    print ("Sorry, you need an A- to qualify!")

```

Solution. The corrected code is given below, with corrections marked in colour.

```

for i in range (0, 100) :
grade = float (input( "What\''s the grade percentage ?" ))
if grade >= 90 :
    print ("That's an A + !")
if 80 <= grade < 90:
    print ("An A is really good!")
elif 60 <= grade :           # although triple-quoted string is also OK
    print ("You got an A-!")
else grade < 60:
    print ("Sorry, you need an A- to qualify!")

```

7. What is the output of following code ?

```

if (4 + 5 == 10) :
    print ("TRUE")
else :
    print ("FALSE")
print ("TRUE")

```

Solution.

FALSE
TRUE

8. Following code contains an endless loop. Could you find out why ? Suggest a solution.

```

n = 10
answer = 1
while ( n > 0 ) :
    answer = answer + n ** 2
    n = n + 1
print (answer)

```

Solution. For a while loop to terminate, it is necessary that its condition becomes false at one point. But in the above code, the test-condition of while loop will never become *false*, because the condition is *n > 0* and *n* is always incrementing ; thus *n* will always remain > 0 . Hence it will repeat endlessly.

There are two possible solutions :

(i) change the condition to some reachable limit (as per update expression) e.g.,

```
while (n < 100) :
```

(ii) change the updation equation of loop variable so that it makes the condition *false* at some point of time, e.g.,

```

while (n > 0) :
    :
    n = n - 1

```

9. Consider below given two code fragments. What will be their outputs if the inputs are entered in the given order are (i) 'Jacob' (ii)'12345' (iii) 'END' (iv) 'end' ?

```
(a) name = " "
    while True :
        name = input( "Enter name ('end' to exit):")
        if name == "end" :
            break
        print ("Hello", name)
    else :
        print ("Wasn't it fun ?")
```

```
(b) name = " "
    while name != "end" :
        name = input( "Enter name ('end' to exit):" )
        if name == "end" :
            pass
        print ("Hello", name)
    else :
        print ("Wasn't it fun ?")
```

Solution. Code (a) will print

for input (i)
Hello Jacob
for input (ii)
Hello 12345
for input (iii)
Hello END
for input (iv)
no output will be printed

Code (b) will print

for input (i)
Hello Jacob
for input (ii)
Hello 12345
for input (iii)
Hello END
for input (iv)
Hello end
Wasn't it fun ?

10. Write Python code to add the odd numbers up to (and including) a given value N and print the result.
Solution.

```
N = int (input ('Enter number'))
sum = 0
i = 1
while i <= N :
    sum = sum + i
    i = i + 2
print (sum)
```

11. Consider the following Python program :

```
N = int(input ( "Enter N :" ))
i = 1
sum = 0
```

```

while i < N :
    if i % 2 == 0 :
        sum = sum + i
    i = i + 1
print (sum)

```

(a) What is the output when the input value is 5 ?

(b) What is the output when the input value is 0 ?

Solution. (a) 6 (b) 0

12. Consider the following Python program, intended to calculate factorials :

```

number = int(input ("Enter number"))
n, result = number, 1
while True or n :
    result = result * n
    n = n - 1
factorial = result
print ("factorial of ", number, "is", factorial)

```

(a) What happens when the user enters a value of 5 ?

(b) How would you fix this program ?

Solution. The problem is that the program will repeat infinitely.

The problem lies with the condition of while loop this condition will never go false.

Correct condition will be any of these :

```

while n :
    :
or    while n > 0 :
    :

```

13. Write a program to find lowest and second lowest number from the 10 numbers input.

Solution.

```

# to find lowest and second lowest integer from 10 integers
small = smaller = 0
for i in range(10):
    n = int(input("Enter number :"))           # first number read
    if i == 0:
        small = n                            # second number read
    elif i == 1:
        if n <= small:
            smaller = n
    else:
        smaller = small
        small = n                            # for every integer read 3rd integer onwards
    else:

```

```

if n < smaller :
    small = smaller
    smaller = n
elif n < small :
    small = n

print("The lowest number is : ", smaller)
print("The second lowest number is : ", small)

```

Sample Run

The lowest number is : -6
 The second lowest number is : -4

The sample run of above program (from the ten input numbers as : -6, 13, 20, -3, 15, 18, 99, -4, 11, 23) has been given above.

- 14.** *Number Guessing Game – This program generates a random number between 1 and 100 and continuously asks the user to guess the number, giving hints along the way.*

Solution.

```

import random
secretNum = random.randint(1,100)           #generate a random integer between [1 100]
guessNumString = int(input ("Guess a number between 1 and 100 inclusive:"))
while guessNum != secretNum :
    if guessNum < secretNum :
        print ("Your guess is too low.")
    else:
        print ("Your guess is too high.")
    guessNum = int(input ("Guess a number between 1 and 100 inclusive:"))
print ("Congratulations! You guessed the number correctly.")

```

- 15.** *Write a Python script to print Fibonacci series' first 20 elements. Some initial elements of a Fibonacci series are:*

0 1 1 2 3 5 8...

Solution.

```

first = 0
second = 1
print (first)
print (second)
for a in range(1, 19) :
    third = first + second
    print (third)
    first, second = second, third

```

- 16.** *Write a Python script to read an integer > 1000 and reverse the number.*

Solution.

```

num = int(input( "Enter an integer (>1000) :"))
tnum = num
reverse = 0
while tnum :
    digit = tnum % 10
    tnum = tnum / 10
    reverse = reverse * 10 + digit
print ("Reverse of", num , "is", reverse)

```

17. Write a Python script to generate divisors of a number.

Solution.

```
num = int(input("Enter an integer : "))
mid = num / 2
print ("The divisors of" , num, "are :")
for a in range (2, mid + 1) :
    if num % a == 0 :
        print (a, end = ' ')
    else :
        print ("-End-")
```

18. Input three angles and determine if they form a triangle or not.

Solution.

```
#check triangle from angles
angle1 = angle2 = angle3 = 0
angle1 = float(input("Enter angle 1 : "))
angle2 = float(input("Enter angle 2 : "))
angle3 = float(input("Enter angle 3 : "))
if (angle1 + angle2 + angle3) == 180 :
    print("These angles form a triangle .")
else:
    print("These angles do not form a triangle .")
```

19. Numbers in the form $2^n - 1$ are called Mersenne Numbers, e.g., $2^1 - 1 = 1$, $2^2 - 1 = 3$, $2^3 - 1 = 7$.

Write a Python script that displays first ten Mersenne numbers.

Solution.

```
#program to print mersenne numbers
print("First 10 Mersenne numbers are:")
for a in range(1, 11):
    mersnum = 2 ** a - 1
    print(mersnum, end = " ")
print()
```

Sample Run

First 10 Mersenne numbers are :
1 3 7 15 31 63 127 255 511 1023

20. Mersenne numbers that are prime numbers also, are called Mersenne Prime numbers. Make modifications in previous question's Python script so that it also displays 'Prime' next to the Mersenne Prime numbers. Also, this time print 20 Mersenne numbers.

Solution.

```
# program to print mersenne prime numbers
for a in range(1, 21):
    mersnum = 2 ** a - 1
    mid = int(mersnum / 2) + 1
    for b in range(2, mid):
        if mersnum % b == 0 :
            print(mersnum)
            break
    else:
        print(mersnum, "\tPrime")
```

Sample Run

(With first 10 Mersenne numbers)

| | |
|------|-------|
| 1 | Prime |
| 3 | Prime |
| 7 | Prime |
| 15 | |
| 31 | Prime |
| 63 | |
| 127 | Prime |
| 255 | |
| 511 | |
| 1023 | |

GLOSSARY

| | |
|----------------------------|---|
| Block | A group of consecutive statements having same indentation level. |
| Body | The block of statements in a compound statement that follows the header. |
| Empty statement | A statement that appears in the code but does nothing. |
| Infinite Loop | A loop that never ends. <i>Endless loop</i> . |
| Iteration Statement | Statement that allows a set of instructions to be performed repeatedly. |
| Jump Statement | Statement that unconditionally transfers program control within a function. |
| Looping Statement | Iteration statement. Also called a loop. |
| Nesting | Same program-construct within another of same type of construct. |
| Nested Loop | A loop that contains another loop inside its body. |
| Suite | Block. |

Assignments

Type A : Short Answer Questions/Conceptual Questions

1. What is the common structure of Python compound statements ?
2. What is the importance of the three programming constructs ?
3. What is empty statement ? What is its need ?
4. Which Python statement can be termed as empty statement ?
5. What is an algorithm ?
6. What is a flowchart ? How is it useful ?
7. What is pseudocode ? How is it useful in developing logic for the solution of a problem ?
8. What is decision tree ? Where are these useful ?
9. Write pseudocode for checking if a number is a factor of 3215 ?
10. Draw flowchart for displaying first 10 odd numbers.
11. Draw decision tree for grade calculation as per :

| | | | | | | | |
|--------------|------|-------|-------|-------|-------|-------|------|
| Marks | > 90 | 80-90 | 70-80 | 60-70 | 50-60 | 40-50 | < 40 |
| Grade | A1 | A2 | B1 | B2 | C1 | C2 | F |

12. Write pseudocode for comparing two products of two sets of numbers (a, b and c, d) and displaying if both products are same or not.
13. What is entry-controlled loop ?
14. What are the four elements of a while loop in Python ?
15. What is the difference between else clause of if-else and else clause of Python loops ?
16. In which cases, the else clause of a loop does not get executed ?
17. What are jump statements ? Name them.
18. How and when are named conditions useful ?
19. What are endless loops ? Why do such loops occur ?

Type B : Application Based Questions

1. Rewrite the following code fragment that saves on the number of comparisons :

```
if (a == 0) :  
    print ("Zero")  
if (a == 1) :  
    print ("One")  
if (a == 2) :  
    print ("Two")  
if (a == 3) :  
    print ("Three")
```

2. Under what conditions will this code fragment print "water" ?

```
if temp < 32 :  
    print ("ice")  
elif temp < 212:  
    print ("water")  
else :  
    print ("steam")
```

3. What is the output produced by the following code ?

```
x = 1  
if x > 3 :  
    if x > 4 :  
        print ("A", end = ' ')  
    else :  
        print ("B", end = ' ')  
elif x < 2:  
    if (x != 0):  
        print ("C", end = ' ')  
print ("D")
```

4. What is the error in following code ? Correct the code :

```
weather = 'raining'  
if weather = 'sunny' :  
    print ("wear sunblock")  
elif weather = "snow":  
    print ("going skiing")  
else :  
    print (weather)
```

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

```
if int('zero') == 0 :  
    print ("zero" )  
elif str(0) == 'zero' :  
    print (0)  
elif str(0) == '0 ' :  
    print (str(0))  
else:  
    print ("none of the above")
```

6. Find the errors in the code given below and correct the code :

```
if n == 0  
    print ("zero")
```

```

    elif : n == 1
        print ("one")
    elif
        n == 2:
            print ("two")
    else n == 3:
        print ("three")

```

7. What is following code doing ? What would it print for input as 3 ?

```

n = int (input( "Enter an integer: " ))
if n < 1 :
    print ("invalid value")
else :
    for i in range(1, n + 1):
        print (i * i)

```

8. How are following two code fragments different from one another ? Also, predict the output of the following code fragments :

(a) `n = int(input("Enter an integer: "))
if n > 0 :
 for a in range(1, n + n) :
 print (a / (n/2))
 else :
 print ("Now quiting")`

(b) `n = int (input("Enter an integer:"))
if n > 0 :
 for a in range(1, n + n) :
 print (a / (n/2))
 else :
 print ("Now quiting")`

9. Rewrite the following code fragments using for loop :

(a) `i = 100
while (i > 0) :
 print (i)
 i -= 3`

(b) `while num > 0 :
 print (num % 10)
 num = num/10`

(c) `while num > 0 :
 count += 1
 sum += num
 num -= 2
 if count == 10 :
 print (sum/float(count))
 break`

10. Rewrite following code fragments using while loops :

(a) `min = 0
max = num
if num < 0 :
 min = num
max = 0 # compute sum of integers from min to max
for i in range(min, max + 1):
 sum += i`

(b) `for i in range(1, 16) :`

`if i % 3 == 0 :
 print (i)`

(c) `for i in range(4) :`

`for j in range(5):
 if i + 1 == j or j + i == 4 :
 print ("+", end = ' ')
 else :`

`print ("o", end = ' ')`

`print ()`

11. Predict the output of the following code fragments :

(a) count = 0

```
while count < 10:
    print ("Hello")
    count += 1
```

(c) keepgoing = True

```
x = 100
while keepgoing :
    print (x)
    x = x - 10
    if x < 50 :
        keepgoing = False
```

(f) for x in range(5):
 print (x)

(i) for z in range(-500, 500, 100):
 print (z)

(k) x = 10
y = 5
for i in range(x-y * 2):
 print ("% ", i)

(m) for x in range(3):
 for y in range(4):
 print (x, y, x + y)

(b) x = 10

```
y = 0
while x > y:
    print (x, y)
    x = x - 1
    y = y + 1
```

(d) x = 45

```
while x < 50 :
    print (x)
```

(e) for x in [1,2,3,4,5]:
 print (x)

(g) for p in range(1,10):
 print (p)

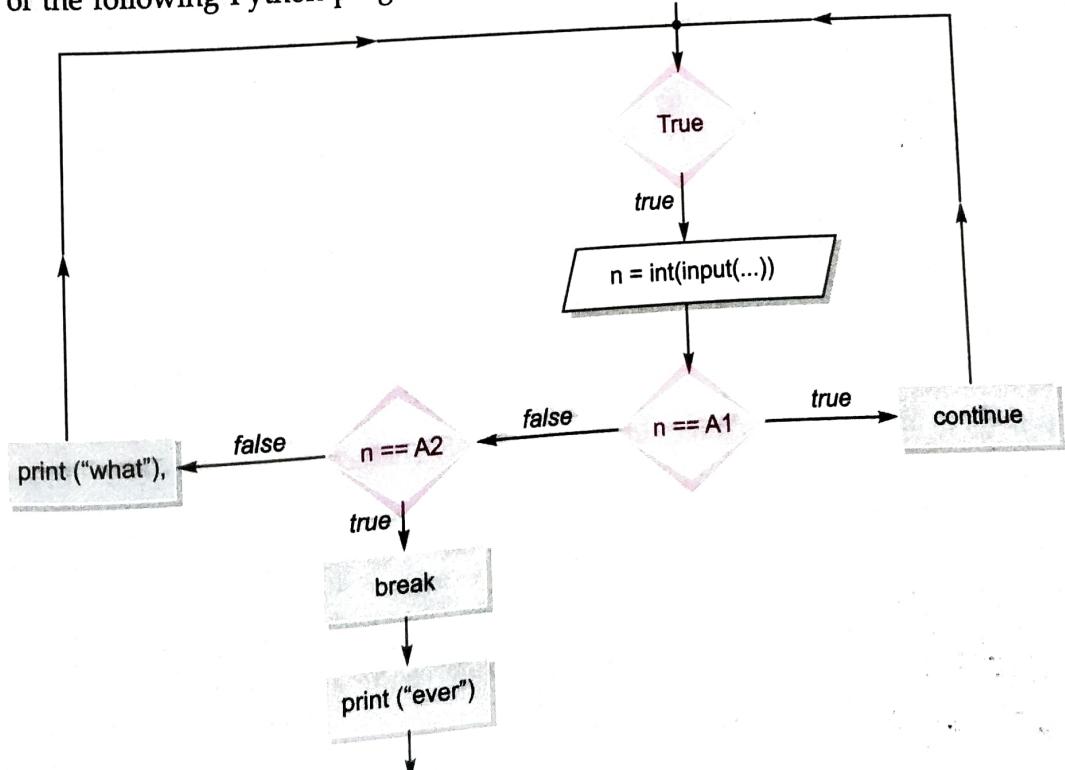
(h) for q in range(100,50,-10):
 print (q)

(j) for y in range(500,100,100):
 print (" * ", y)

(l) for x in [1,2,3]:
 for y in [4, 5, 6]:
 print (x, y)

(n) c = 0
for x in range(10):
 for y in range(5):
 c += 1
 print (c)

12. Which of the following Python programs implement the control flow graph shown ?



```
(a) while True :
    n = int(input("Enter an int:"))
    if n == A1 :
        continue
    elif n == A2 :
        break
    else :
        print ("what")
else :
    print ("ever")
```

```
(c) while True :
    n = int(input("Enter an int:"))
    if n == A1 :
        continue
    elif n == A2 :
        break
    print ("what")
    print ("ever")
```

```
(b) while True :
    n = int(input("Enter an int:"))
    if n == A1 :
        continue
    elif n == A2 :
        break
    else :
        print ("what")
        print ("ever")
```

13. What is the output of the following code ?

```
for i in range(4):
    for j in range(5):
        if i + 1 == j or j + i == 4:
            print("+", end = ' ')
        else:
            print("o", end = ' ')
    print()
```

14. In the nested for loop code above (Q. 13), how many times is the condition of the if clause evaluated ?

Type C : Programming Practice/Knowledge based Questions

1. Write a Python script that asks the user to enter a length in centimetres. If the user enters a negative length, the program should tell the user that the entry is invalid. Otherwise, the program should convert the length to inches and print out the result. There are 2.54 centimetres in an inch.
2. A store charges ₹ 120 per item if you buy less than 10 items. If you buy between 10 and 99 items, the cost is ₹ 100 per item. If you buy 100 or more items, the cost is ₹ 70 per item. Write a program that asks the user how many items they are buying and prints the total cost.
3. Write a program that asks the user for two numbers and prints **Close** if the numbers are within .001 of each other and **Not close** otherwise.
4. A year is a leap year if it is divisible by 4, except that years divisible by 100 are not leap years unless they are also divisible by 400. Write a program that asks the user for a year and prints out whether it is a leap year or not.
5. Write a program to input length of three sides of a triangle. Then check if these sides will form a triangle or not.
(Rule is : $a + b > c$; $b + c > a$; $c + a > b$)
6. Write a short program to input a digit and print it in words.
7. Write a short program to check whether square root of a number is prime or not.
8. Write a short program to print first n odd numbers in descending order.

9. Write a short program to print the following series :
- 1 4 7 10 40.
 - 1 - 4 7 - 10 - 40
10. Write a short program to find average of list of numbers entered through keyboard.
11. Write a program to input 3 sides of a triangle and print whether it is an equilateral, scalene or isosceles triangle.
12. Write a program to take an integer a as an input and check whether it ends with 4 or 8. If it ends with 4, print "ends with 4", if it ends with 8, print "ends with 8", otherwise print "ends with neither".
13. Write a program to take N ($N > 20$) as an input from the user. Print numbers from 11 to N . When the number is a multiple of 3, print "Tipsy", when it is a multiple of 7, print "Topsy". When it is a multiple of both, print "TipsyTopsy".
14. Write a short program to find largest number of a list of numbers entered through keyboard.
15. Write a program to input N numbers and then print the second largest number.
16. Given a list of integers, write a program to find those which are palindromes. For example, the number 4321234 is a palindrome as it reads the same from left to right and from right to left.
17. Write a complete Python program to do the following :
- read an integer X .
 - determine the number of digits n in X .
 - form an integer Y that has the number of digits n at ten's place and the most significant digit of X at one's place.
 - Output Y .
- (For example, if X is equal to 2134, then Y should be 42 as there are 4 digits and the most significant number is 2).

18. Write a Python program to print every integer between 1 and n divisible by m . Also report whether the number that is divisible by m is even or odd.

19. Write Python programs to sum the given sequences :

(a) $\frac{2}{9} - \frac{5}{13} + \frac{8}{17} \dots \dots \text{ (print 7 terms)}$

(b) $1^2 + 3^2 + 5^2 + \dots + n^2 \text{ (Input } n\text{)}$

20. Write a Python program to sum the sequence

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{n!} \text{ (Input } n\text{)}$$

21. Write a program to accept the age of n employees and count the number of persons in the following age group :

(i) 26 - 35 (ii) 36 - 45 (iii) 46 - 55.

22. Write programs to find the sum of the following series :

$$(a) x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \frac{x^5}{5!} - \frac{x^6}{6!} \text{ (Input } x\text{)} \quad (b) x + \frac{x^2}{2} + \frac{x^3}{3} + \dots + \frac{x^n}{n} \text{ (Input } x \text{ and } n \text{ both)}$$

23. Write programs to print the following shapes :

(a)

*
* *
* * *
* *
*

(b)

*
* *
* * *
* *
*

(c)

* *
* * *
* * *
* *
*

(d)

*
* *
* * *
* * *
* *
*

24. Write programs using nested loops to produce the following patterns :

(a) A

A B

A B C

A B C D

A B C D E

A B C D E F

(b) 0

2 2

4 4 4

8 8 8 8

25. Write a program using nested loops to produce a rectangle of *'s with 6 rows and 20 *'s per row.

26. Given three numbers A, B and C, write a program to write their values in an ascending order. For example, if A = 12, B = 10, and C = 15, your program should print out :

Smallest number = 10

Next higher number = 12

Highest number = 15

27. Write a Python script to input temperature. Then ask them what units, Celsius or Fahrenheit, the temperature is in. Your program should convert the temperature to the other unit. The conversions are $F = \frac{9}{5}C + 32$ and $C = \frac{5}{9}(F - 32)$.

28. Ask the user to enter a temperature in Celsius. The program should print a message based on the temperature :

- ▲ If the temperature is less than -273.15, print that the temperature is invalid because it is below absolute zero.
- ▲ If it is exactly -273.15, print that the temperature is absolute 0.
- ▲ If the temperature is between -273.15 and 0, print that the temperature is below freezing.
- ▲ If it is 0, print that the temperature is at the freezing point.
- ▲ If it is between 0 and 100, print that the temperature is in the normal range.
- ▲ If it is 100, print that the temperature is at the boiling point.
- ▲ If it is above 100, print that the temperature is above the boiling point.