CLASS-XI

Flow of Control

STUDY NOTE

Period-1

Learning Outcomes

- Ability to understand the concepts of computational thinking.
- Ability to understand the notion of data types and data structures and apply in different situations.
- Ability to appreciate the notion of an algorithm and apply its structure.
- Ability to develop the concepts of flowing construct.
- Ability to handle different flow of controls in program.
- Knowing the concepts of sequential, selection & iterative construct used in the program
- Working with the jumping construct in program.

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 the while are discussed. This chapter also discuss some jump statements of Python which are break and continue.

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 statement can belong to one of the following three types:

(a) Empty Statement

(b) Simple Statement (Single statement)

(c) Compound Statement

(a) **Empty Statement**:- This 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.

(b) **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.

(c) **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 indented inside the header line. All statements in the body are at the same level of indentation. You will learn about some other compound statements (if, for, while) in this chapter.

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 figure). 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.

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

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 known basic tools that will help you decide about the logic to solve a given program 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 pseudo code.

Program Logic development Tools:-

Before developing the solution of a problem in terms of a program, you should read and analyze the given problem and decide about basic sub-tasks needed to solve a problem and the 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 9well-definded instructions) to solve a given problem.

For instance, the algorithm to find the remainder of given two numbers is:

(a) Input fist number

(b) Input second number

(c) Divide fist number with second number and store the remainder as third number.

(d) Display the result (the third number)

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 determines if the first number is divisible by second number or not.

```
(1) Input first number
 (2) Input second number
(3) Divide fist 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) It 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 pseudo code, flow charts, or decision trees and tables. Here, we shall stick to flowcharts only, as suggested by the syllabus.

Flowcharts:-

A flowchart is a graphical representation of an algorithm. A flowchart shows different subtasks with different symbols. Following figure shows commonly used 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 Sub-process 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.

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

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.

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]
```

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:

```
If ch== ‘  ‘:
    Spaces +=1
    Chars+=1
```

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. Consider another example illustrating the use of if statement:

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

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")
```

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

```
(a) if grade == 'A'             # comparison with a literal
        print ("You did well")
(b) if a > b :                  # comparing two variables
        print ("A has more than B has")
(c) if x :                      # testing truth value of a variable
        print ("x has truth value as true")
        print ("Hence you can see this message.")
```

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

```
if not x :    # not x will return True when x has 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 and False when x is true.

Period-2

The if - else Statement:-

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

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

For instance, consider the following code fragment:

```
if a > = 0:
    print (a, "is zero or a positive number")
else :
    print 9a, "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.

Following figure illustrates if and if-else constructs of Python.

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

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

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

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

Sum 1 as the sum of all input numbers

Sum 2 as the sum of non-duplicate numbers; if there are duplicate numbers in the input, ignores them

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

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

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

Alternative 1

```
sum 1 = sum 2 = 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 + = num2
if num3 ! = num1 and num3 ! = num2 :
    sum2 + = num3
print ("Numbers are", num1, num2, num3)
print ("Sum of three given numbers is", sum1)
print ("Sum of non-duplicate numbers is", sum2)
```

Alternative 2

```
sum1 = sum2 = 0
```

```
num1 = int (input("Enter number 1:"))
num2=int(input("Enter number 2 : "))
num3 = int (input("Enter number 3 : "))
sum1 = num1 + num2 + num3
if num1 == num2 :
    if num3 ! = num1 :
        sum2 + = num3
else :
    if num1 = = num3 :
        sum2 += num2
else:
    if num2 ==num3 :
        sum2 + = num1
else :
    sum2 += num1 + num2 + num3
print ("Numbers are", num1, num2, num3)
print ("Sum of three given numbers is", sum1)
print ("Sum of non-duplicate numbers is", sum2)
```

Simple 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 0
```

Program:- 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 % number 2
if remainder = = 0 :
    print (number1, "is divisible by", number2)
else :
    print(number1, "is not divisible by", number2)
```

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
```

Program to find the multiple 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, sept =" ")
        count +=1
remainder = num3 % divisor
if remainder = =  0:
        print(num3, sept =" ")
        count +=1
remainder = num4 % divisor
if remainder = = 0 :
        print/(num4, sep = " ")
        count +=1
remainder == 0 :
        print(num5, sep = " " )
        count +=1
print( )
print (count, "multiples of", divisor, "found")
```

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) :") )
it choice = = 1 :
        area = 3.14159*radius *radius
        print ("Area of circle with radius", radius, 'is', area)
```

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
    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 by 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.

Period-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 not fifty
```

Refer to program 8.3 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]
```

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”)
```

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 is the testing a number wheter 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 >= 1000 :
        Discount = sales * 0.10
    else :
        Discount = sales * 0.05
```

Consider following program that uses an if-elif statement.

- **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 = num / 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
```

Period-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 it.

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> :
    statement(s)
```

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

From 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  <conditional1 expression> :
      statement
      [ statements]
elif <conditional expression> :
      Statement
      [statements]
else :
      If < conditional expression> :
            Statement
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> :
          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.

Following example programs illustrate the use of nested ifs.

- 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 :
      min, mid, max = x, y, z
  else :
      min, mid, max = x, z, y
elif  y  < x and y < x :
      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)
```

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

- 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 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 : S
You entered a special character.
```

Period-5

➢ 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 chapattis/dosas/appam at home, as mentioned earlier. Now if you have to write pseudo code 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 pseudo code for above task of making dosas/appam as many as you want.

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

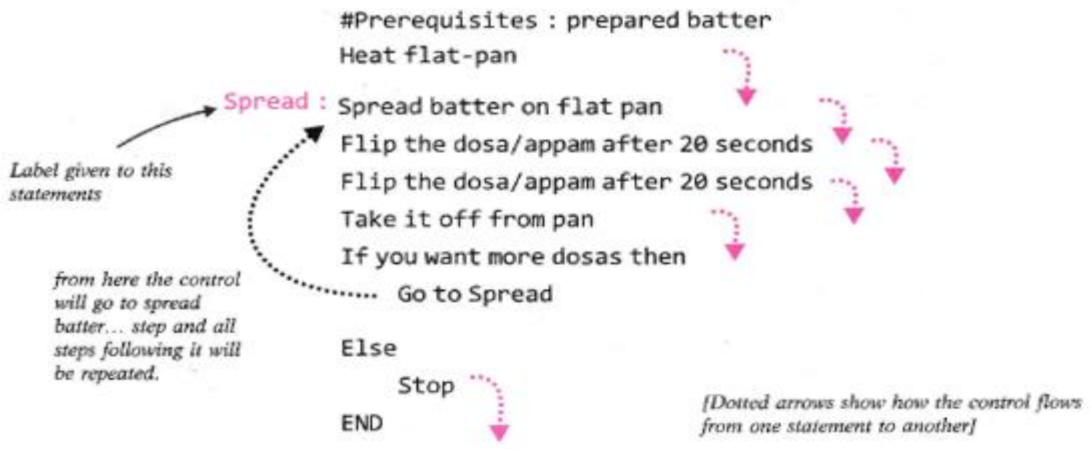
Now what should you write at the place of ???, so that it starts repeating the process again from second step ? Since there is no number associated with steps, how can one tell from where to repeat ?

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

Pseudocode Reference 1

#Prerequisites : prepared batter

Heat flat-pan



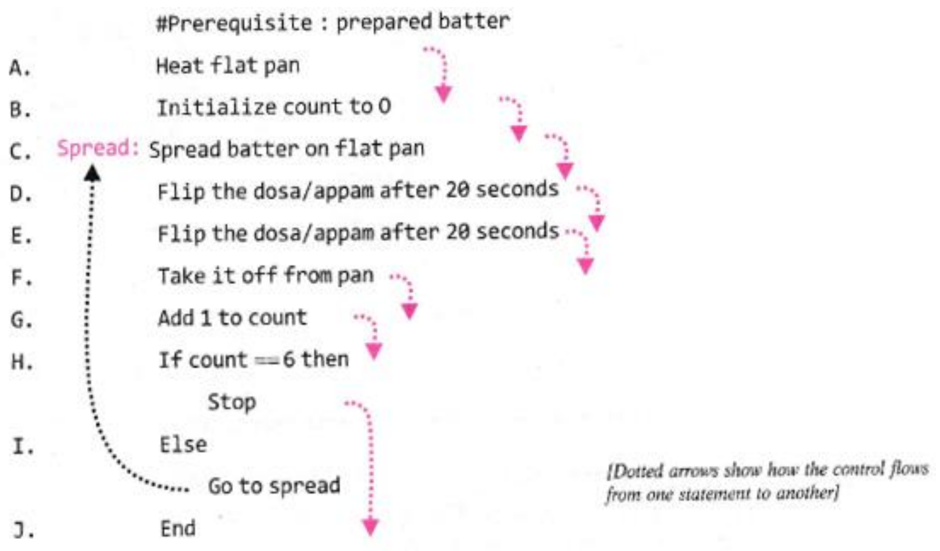
```
        #Prerequisites : prepared batter
        Heat flat-pan
Spread : Spread batter on flat pan
        Flip the dosa/appam after 20 seconds
        Flip the dosa/appam after 20 seconds
        Take it off from pan
        If you want more dosas then
            Go to Spread
        Else
     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

Label given to this statements
from here the control will go to spread batter... step and all steps following it will be repeated.
[Dotted arrows show how the control flows from one statement to another]

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

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

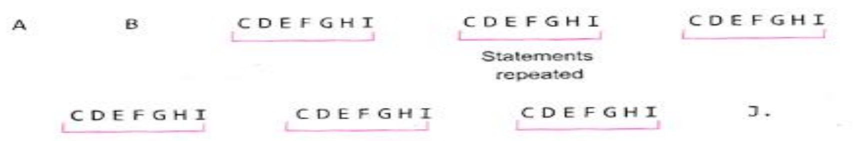
Pseudocode Reference 2.



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

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

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



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

This way we can develop logic for carrying out repetitive task, either based on a condition (pseudocode reference1) or a given number of times (pseudo code reference 2).

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

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

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

➢ **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 generation 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 figure. There are some advance sequence types also but those are beyond the scope of our discussion.

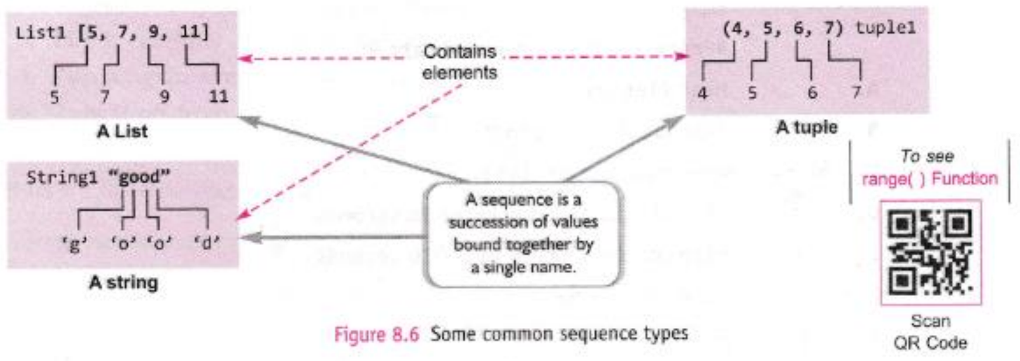


Figure 8.6 Some common sequence types

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

range (0, 5)

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

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

range (5, 0)

will return empty lit [ ] as no number falls in the a.p. beginning with 5 and ending at 10 (difference d or step-value = +1).

range (12, 18)

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

All the above examples of range () produce numbers increasing by value1. 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,....<=u-1.

range (0, 10, 2)

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

range (5, 0, -1)

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

Another form of range () is:

range (<number>)

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

range (5)

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

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

range (9, 3, -1)	9, 8, 7, 6, 5, 4
range (10, 1, -2)	10, 8, 6, 4, 2

Period-6

Operators in and not in:-

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

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

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

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

```
5 not [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.

For example, consider following code:

```
'a' in "true"
```

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

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

will also return True for the same reason.

Now consider the following code that uses the in operator:

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

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.

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 loop 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 conditional loop.

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 <variables> in < sequence> :
    statement _ to _ repeat
```

For example, consider the following loop:

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

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 vale 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:

(a) 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)

(b) Next, a will be assigned next value in the list i.e, 4 (as result of print (a)) and 16 (as result of print (a*a)) are printed.

(c) 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.

(d) 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
```

Consider another for loop :

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

The above loop will produce output as:

```
c
a
l
m
```

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

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

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

The above loop will produce output as :

```
1
8
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 in neither easy nor good on readability. Then? The what ? Arey, our very own range () function is there to rescue. Why worry? ;)

For big number based lists, you can specify range () function to represent a list as in:

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

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

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

Program :- 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:

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

Program :- 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 for.

```
    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:- 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).

Period-6

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).

The general form of Python while loop is :

```
while <logical expression> :
    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.

The understand the working of while loop, consider the following code :

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

The above code will print :

```
hello 5
hello 2
Loop over !!
```

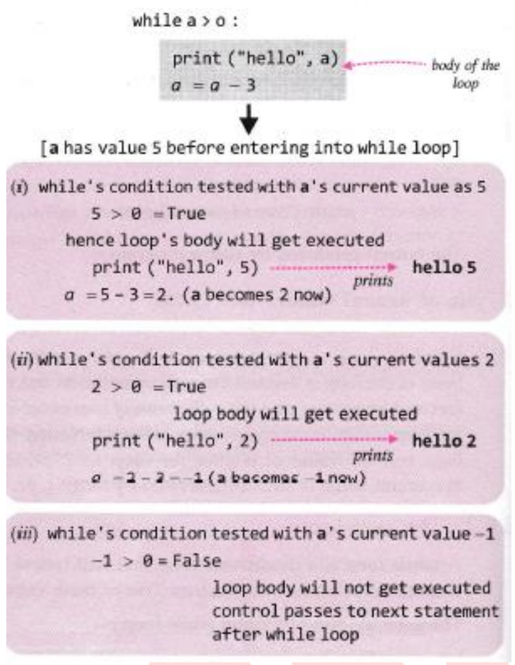
Let us see how a while loop is processed :

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$_{tval}$) then all statements in the loop's body are executed, (see section (i) and (ii) on the right).

Case - II, If the result of step 1 is false (False or false$_{tval}$) 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. After executing the statements of loop-body again, step 1, step 2 and step 3 are 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.)



```
while a > 0 :
      print ("hello", a)   <------- body of the loop
      a = a - 3

[a has value 5 before entering into while loop]

(i) while's condition tested with a's current value as 5
        5 > 0 = True
    hence loop's body will get executed
        print ("hello", 5) ------prints------> hello 5
    a = 5 - 3 = 2. (a becomes 2 now)

(ii) while's condition tested with a's current values 2
        2 > 0 = True
              loop body will get executed
        print ("hello", 2) ------prints------> hello 2
        a = 2 - 3 = -1 (a becomes -1 now)

(iii) while's condition tested with a's current value -1
        -1 > 0 = False
                 loop body will not get executed
                 control passes to next statement
                 after while loop
```

Consider another example :

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

The above code will print output as:

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

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

Anatomy of a while Loop (Loop Control Elements)

Every while loop has its elements that control and govern its execution. A while loop has four elements that have different purposes.

These elements are as given below:

1. Initialization Expression (s) (Starting)

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

2. Test Expression (Repeating or Stopping)

The test expression is an expression whose truth value decides whether the loop-body will be executed or not. If the test expression evaluates to true, the loop-body gets executed, otherwise the loop is terminated.

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

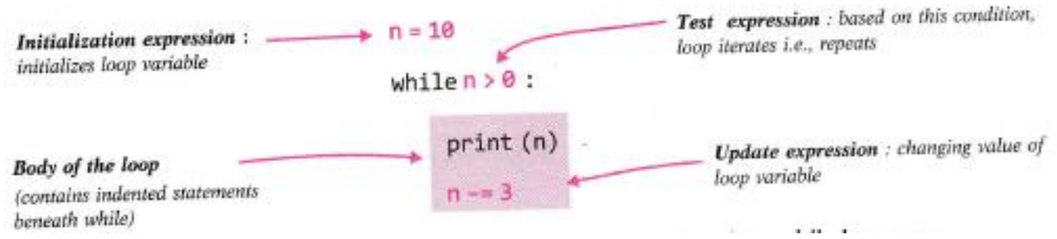
3. The Body-of -the-Loop (Doing)

The statements that are executed repeatedly (as long as the test-expression is true) form the body of the loop. In a while loop, before every iteration the test-expression is evaluated and if it is true, the body-of -the-loop is executed; if the test-expression evaluates to false, the loop is terminated.

4. Update Expression (s) (Changing)

The update expressions change the value of loop variable. The update expression is given as a statement inside the body of while loop.

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



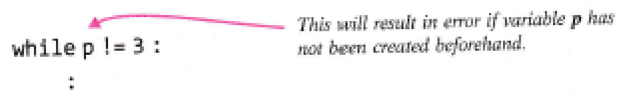
```
n = 10
while n > 0 :
    print (n)
    n -= 3
```

Initialization expression : initializes loop variable
Test expression : based on this condition, loop iterates i.e., repeats
Body of the loop (contains indented statements beneath while)
Update expression : changing value of loop variable

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

In a while loop, a loop control variable should be initialized before the loop begins as an uninitalized variable cannot be used in an expression.

Consider following code :

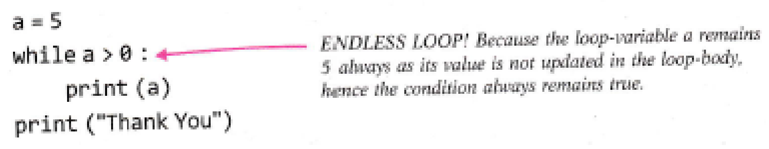


```
while p != 3 :
    :
```

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

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

Consider following code :

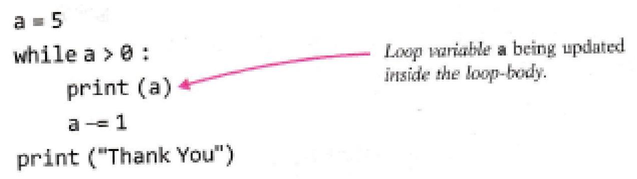


```
a = 5
while a > 0 :
    print (a)
print ("Thank You")
```

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

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

The corrected form of above loop will be:



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

Loop variable a being updated inside the loop-body.

We will talk about infinite loops once again after the break statement is discussed. Since in a while loop, the control in the form of condition-check is at the entry of the loop (loop is not entered, if the condition is false), it is an example of an entry-controlled loop. An entry-controlled loop is a loop that controls the entry in the loop by testing a condition. Python does not offer any exit-controlled loop. Now that you are familiar with while loop, let us write some programs thalt make use of while loop.

Program :- program to calculate the factorial of a number.

```
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
Enter a number : 4
The factorial of 4 is 24
```

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

```
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 nature ? 25
The sum of even integers is 156
The sum of odd integers is 169
```

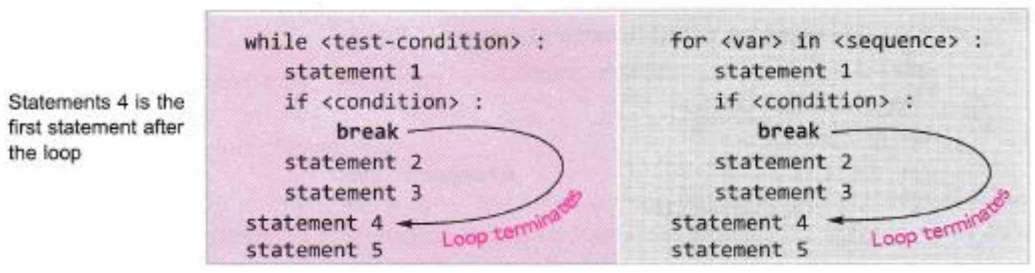
Loop else Statement:- Both loops of Python, (i.e, for loop and while loop) have an else clause, which is different from else of if-else statements. The else of a loop executes only when the loop ends normally (i.e, only when the loop is ending because the while loop's test condition has resulted in false 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.

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.

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 explains the working of a break statement:



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

```
a = b = c = 0
for i in range 91, 21) :
    a = int (input (Enter number 1 :"))
    b = int (input ("Enter number 2:"))
    if b ==0
        print ("Division by zero error ! Aborting!")
        break
    else:
        c = a/b
        print ("Quotient=",c)
    print ("program over!")
```

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

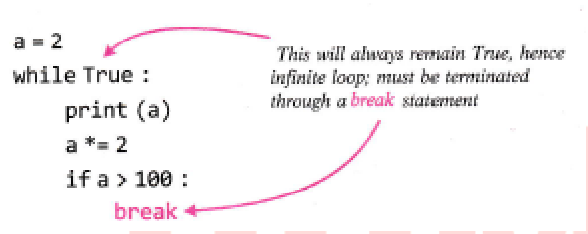
Division by zero error ! aborting!

Program over !

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

Infinite Loos 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 :

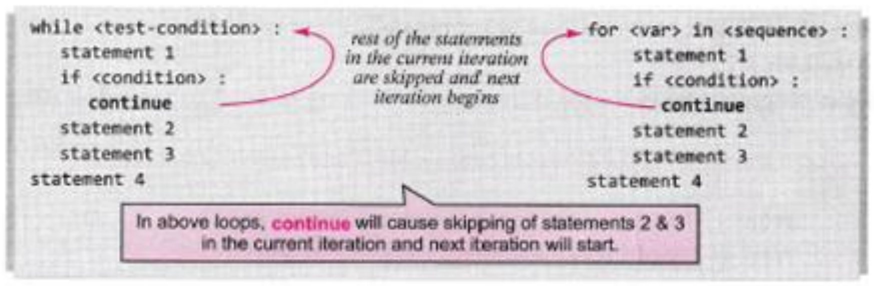


```
a = 2
while True :
    print (a)
    a *= 2
    if a > 100 :
        break
```

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

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

The following figure explain the working of continue statement :



```
while <test-condition> :
    statement 1
    if <condition> :
        continue
    statement 2
    statement 3
statement 4
```

```
for <var> in <sequence> :
    statement 1
    if <condition> :
        continue
    statement 2
    statement 3
statement 4
```

rest of the statements in the current iteration are skipped and next iteration begins
In above loops, continue will cause skipping of statements 2 & 3 in the current iteration and next iteration will start.

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

```
a = b = c = 0
for i in range (0, 3) :
    print ("Enter 2 numbers")
    a=int (input ("Number 1 : "))
```

```
        b=int (input ("Number 2 :"))
        if b == 0
                print ("/n The denominator cannot be zero. Enter again !")
                continue
        else :
                c=a//b
                print ("Quotient=", c)
```

Sample run of above code is shown below:

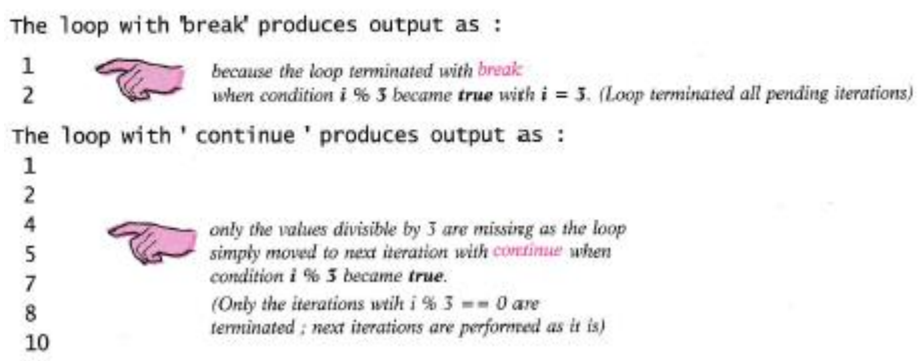
```
Enter 2 numbers
Number 1 : 5
Number 2 : 0
The denominator cannot be zero. Enter again !
Enter 2 numbers
Number 1 : 5
Number 2 : 2
Quotient = 2
Enter 2 numbers
Number 1 : 6
Number 2 : 2
Quotient = 3
```

Sometimes you needs 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 different in working of break and continue statements.

Program:- program to illustrate the difference between break and continue statements

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

The out produced by above program is :

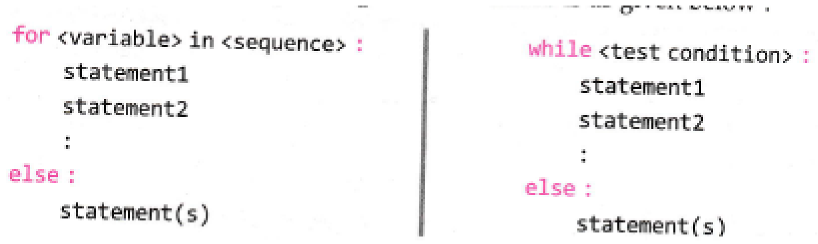


```
The loop with 'break' produces output as :
 1       because the loop terminated with break
 2       when condition i % 3 became true with i = 3. (Loop terminated all pending iterations)
The loop with ' continue ' produces output as :
 1
 2
 4       only the values divisible by 3 are missing as the 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.

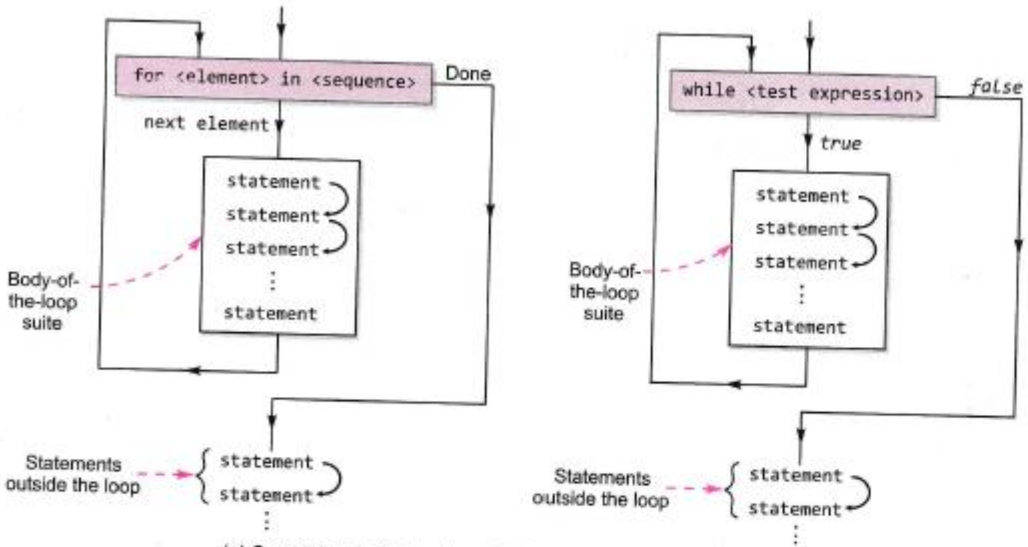
Loop else Statement:- Now that you how break statement works, we can now talk about loop-else clause in details. As mentioned earlier, the else clause of a Python loop executes when the loop terminates normally, i.e when test-condition results into false for a while loop or for loop has executed for 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:

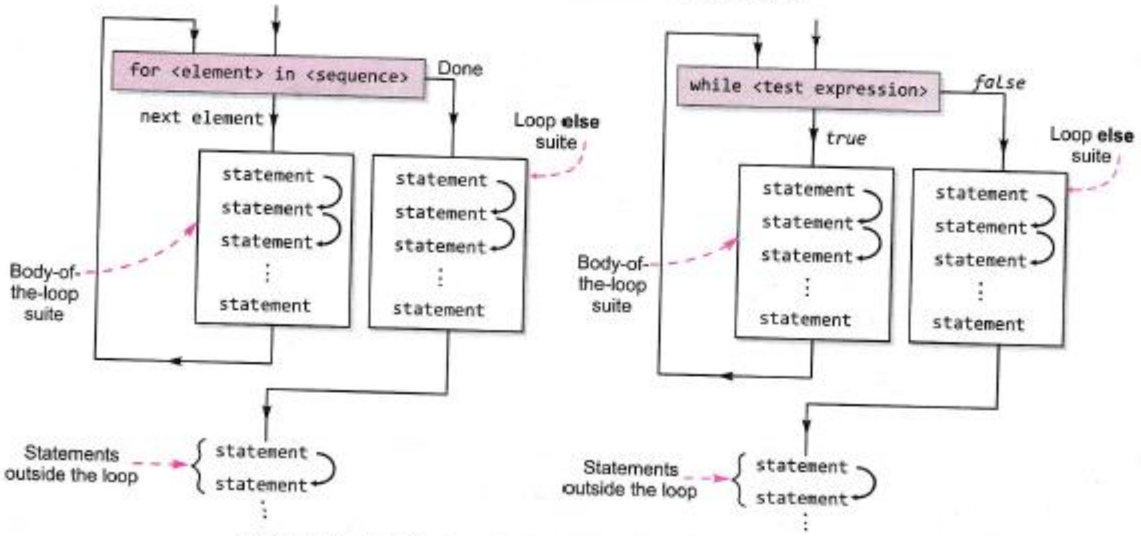


```
for <variable> in <sequence> :          while <test condition> :
    statement1                              statement1
    statement2                              statement2
    :                                       :
else :                                  else :
    statement(s)            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

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



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

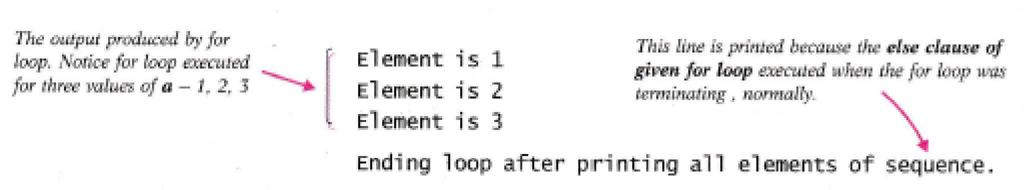


(b) Control flow in Python loops in the presence of loop-else clause.

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

```
for a in range 91, 4) :
    print ("Element is", end =' ')
    print (a)
else :
    print ("Ending loop after printing all elements of sequence")
```

The above will give the following


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

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

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")
```

Now the out put is

```
    Element is 1
```

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.

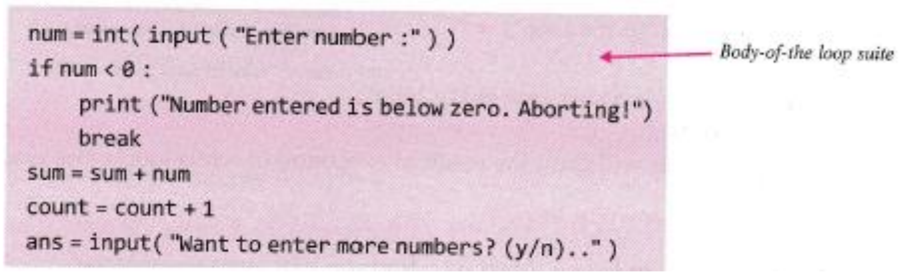
```
    for a in range (1, 4) :
        print ("Element is", end = ' ')
        print (a)
        if %2 ==0
                break
        else:
                print ("Ending loop after printing all elements of sequence")
```

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.

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

```
        count = sum= 0
```

```
    ans = 'y'
while ans =='y'
```



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

```
else :
    print ("You entered", count, "numbers so far.")
print ("Sum of numbers entered is", sum)
```

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

Program:- Program to input a number and test if it is a prime number

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

```
else :
    print (num, "is a prime number")
Enter number : 7
7 is a prime number
Enter number : 49
49 is not a prime number
```

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

Consider following code examples illustrating the same.

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

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")
```

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

```
exiting from for loop
```

Nested Loops:-

A loop may contain another 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 ()
```

The output produced by above code is:

```
*
```

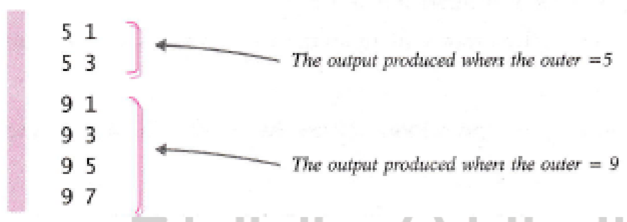
```
* *
* * *
* * * *
```

The inner for loop is executed for each value of i. the variable i takes value 1, 2, 3 and 4. The inner loop is executed once for i = 1 according to sequence in inner loop range (1, i) (because for i as 1, there is just one element 1 in range (1, 1), thus j iterates for one value, i.e, 1) twice for i = 2 (two elements in sequence range (1, i), thrice for i = 3 and four times for i = 4. While working with nested loops, you need to understand one thing and that is, the value of outer loop variable will change only after the inner loop is completely finished (or interrupted).

To understand this, consider the following code fragment:

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

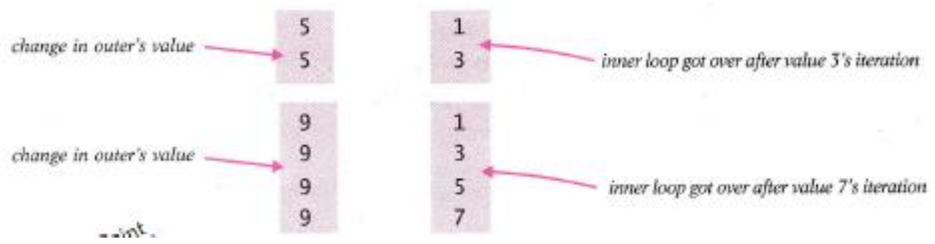
The above code will produce the output as:



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

The output produced when the outer =5
The output produced when the outer = 9

See the explanation below:



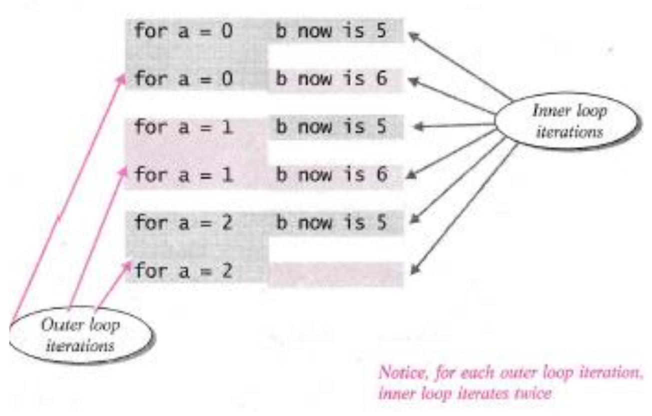
```
5    1
5    3
9    1
9    3
9    5
9    7
```

change in outer's value
inner loop got over after value 3's iteration
change in outer's value
inner loop got over after value 7's iteration

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:



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 fo outer loop (a = 0, which involves two iterations of inner loop)
- when a takes the next value in sequence, the inner will again iterate two times with values 5 and 6 for b.

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

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

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

The output produced would be:

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

Consider another nested loop :

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

The output produced would be :

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

The break Statement in a nested Loop

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

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

Program:- Program that searches for prime numbers from 15 through 25.

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

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
```

****