# UNIT – 3

# CONTROL FLOW, FUNCTIONS

## 3.1 CONDITIONALS

### 3.1.1 Boolean values and operator

The Boolean values are **True** and **False**. The relational operators such as = = , !=, > , < , >= , <= and the logical operators such as *and*, *or*, *not* are the Boolean operators. The statement that prints either true or false is a Boolean expression. The following examples use the operator !=, which compares two operands and produces 'True' if they are not equal and 'False' otherwise:

```
>>> 5 != 5
True
>>> 5 != 6
False
```

So, 'True' and 'False' are special values which belong to the Boolean type; they are not strings (case sensitive):

To know the type of data, the following example can be used.

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

Boolean function works with relational operator, string comparison and logical operators.

### 3.1.2 Relational Operators

Relational operators compares values and evaluate single value either true or false

The relational operators are as follows:

```
x == y      # x is equal to y
x != y      # x is not equal to y
x > y       # x is greater than y
x < y       # x is less than y
x >= y      # x is greater than or equal to y
x <= y      # x is less than or equal to y
```

**Simple program to explain relation operators for Boolean conditions**

```
x = 10
y = 12
# Output: x > y is False
print('x > y  is',x>y)
# Output: x < y is True
print('x < y  is',x<y)
# Output: x == y is False
print('x == y is',x==y)
# Output: x != y is True
print('x != y is',x!=y)
# Output: x >= y is False
print('x >= y is',x>=y)
# Output: x <= y is True
print('x <= y is',x<=y)
```

***The result is :***

```
x > y  is False
x < y  is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

### 3.1.3 String Operators:

Boolean expression can be used in string functions. In Python, strings differ in both lower and upper case.

***For example,***

```
x== "Sunday"
y= "sunday"
print ("x==y:", x==y)
x==y : False
```

### 3.1.4 Logical Operators

There are three **logical operators**: ***and***, ***or***, and ***not***. The semantics of these operators is related to their meaning in English.

***For example,***

1. **and opertator:** Returns true only when both expression is true

   (x > 0) and (x < 10)

   - True only if x is greater than 0 and less than 10.
   - Otherwise False

2. **or operator:** Returns true if any one expression is true

   (n%2 == 0) or (n%3 == 0)

   - True if either of the conditions is true, that is, if the number is divisible by 2.
   - otherwise False

3. ***not operator****:* not operator negates a Boolean expression

   - not (x > y)
   - True if x > y is False
   - False if x>y is True

Normally the operands of the logical operators must be Boolean expressions. However, Python is very flexible. Any nonzero number is interpreted as "True."

***Example:***

```
>>> 11 and True

True
```

**Program to explain logical operations for boolean conditions**

```
x = True
y = False
# Output: x and y is False
print('x and y is',x and y)
# Output: x or y is True
print('x or y is',x or y)
# Output: not x is False
print('not x is',not x)
```

***The result is:***

x and y is False

x or y is True

not x is False

## 3.2 CONDITIONAL STATEMENT

### 3.2.1 Conditional (if)

Conditional statements provide the ability to check conditions and control the program execution accordingly. The simplest form of conditional statement is the *if* statement. The syntax of ***if*** statement is given below.

***Syntax :***

if test expression:

statement(s)

*(or)*

if test expression: statement

The program evaluates the `test expression` and will execute statement(s) only if the text expression is `True`. If the text expression is `False`, the statement(s) is not executed.

***Flowchart***

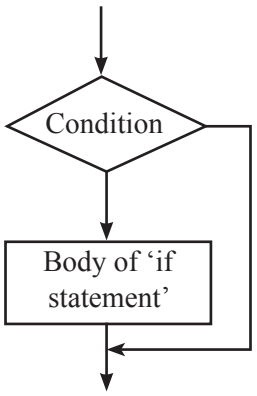



***Fig 3.1. Operatiion of if... statement***

***Example:***

if x > 0 : print 'x is positive'

In the above example, if x>0 becomes true, then the indented print statement gets executed to print 'x is positive'. Otherwise, control skips after the print statement.

*if* statements consists of boolean expression followed by one or more statements. There is no limit on the number of statements appear in the body, but there has to be at least one For example,

```
var = 24
if (var % 2 ==0):
    print “Even number”
    print “The even number is “, var
```

Occasionally, a body with no statements is used. In that case, you can use the pass statement, which does nothing. For example,

```
if x < 0:
pass        # need to handle negative values!
```

The statement which comes after condition checking and : symbol requires indentation if the statement is typed in next line.

For example to check the number is odd or even,

***Program code:***

```
num = 5
if (num%2) != 0:
   print(num, “is odd number.”)
print(“This is always printed.”)
num = 4
if (num%2) == 0:
   print(num, “is even number.”)
print(“This is also always printed.”)
```

***Result:***

```
5 is odd number.
This is always printed.
4 is even number.
This is also always printed.
```

## One more example to explain if statement.

***# program to check whether the year is leap year or not.***

***Program code:***

```
year = 2016
if (num%4) != 0:
   print(year, “is not leap year.”)
print(“This is always printed.”)
```

```
if (year%4) == 0:
    print(year, "is leap year.")
print("This is also always printed.")
```

***Result:***

```
This is always printed.
2016 is leap year.
This is also always printed.
```

### 3.2.2 if...else Statement

The **if…else** statement is called **alternative execution**, in which there are two possibilities and the condition determines which one gets executed. The syntax of if…else statement is given below.

***Syntax of if...else***

```
if test expression:
    Body of if
else:
    Body of else
```

The `if...else` statement evaluates the `test expression` and will execute body of `if` only when test condition is `True`. And if the condition is `False`, body of `else` is executed. Indentation is used to separate the blocks. Fig.3.2. illustrates the flow of *if... else statement*

***Flowchart***

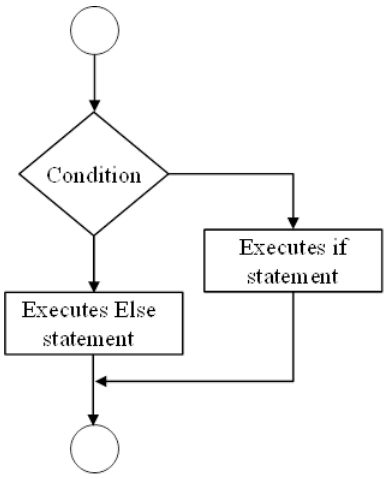



***Fig.3.2. Operation of if.. else statement***

# program to check whether the year is leap year or not using if…else statement.

Program code:

```
year = 2016
```

```
if (year % 4) == 0:
   print (year, "is leap year.")
else:
   print (year, "is not leap year.")
```

***Result:***

2016 is leap year.

If the remainder when year is divided by 4 is 0, then the program prints that the year is leap year. If the condition is false, then the else part body will be executed.

## Python Program to Check if a Number is Odd or Even

```
# Python program to check if the input number is odd or even.
# A number is even if it divides completely by 2 and gives a remainder of 0.
# If remainder is 1, it is odd number.
num = int(input("Enter a number: "))
if (num % 2) == 0:
  print("{0} is Even".format(num))
else:
  print("{0} is Odd".format(num))
```

***The result is:***

Enter a number: 43

43 is Odd

If the remainder when *x* is divided by 2 is 0, then *x* is even, and the program displays a message *x* is even. If the condition is false, then *x* is odd , so the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution.

### 3.2.3 Chained conditionals (if-elif-else)

**Chained conditionals** (if-elif-else) allows more than two possibilities and need more than two branches. The syntax of if-elif-else is shown below.The flow of if.. elif.. else statement is illustrated in the Fig.3.3.

***Syntax of if...elif...else***

```
if test expression:
   Body of if
elif test expression:
```

```
    Body of elif
else:
    Body of else
```

### *Flowchart*

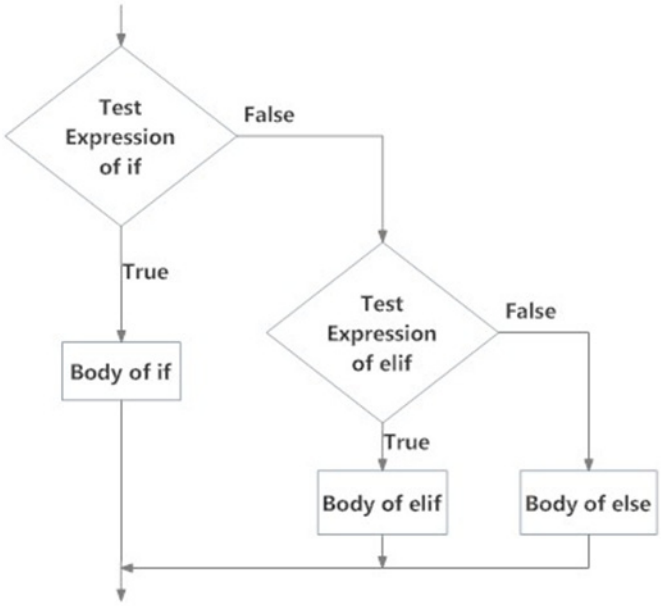



***Fig.3.3. Operation of if … elif….Statement***

The example explains the if-elif-else:

```
if x < y:
print 'x is less than y'
elif x > y:
print 'x is greater than y'
else:
print 'x and y are equal'
```

Here, *elif* is an abbreviation of "else if." However, exactly one branch will be executed. There is no limit on the number of *elif* statements. The else statement body will be at the end, but there doesn't have to be one. Consider the following example as an example. In this draw_a(), draw_b() and draw_c() are the functions which are explained later in this chapter.

```
if choice == 'a':
raw_a()
elif choice == 'b':
```

```
draw_b()
elif choice == 'c':
draw_c()
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

***Program code:***

```
num = 5.4
if num > 0:
   print("Positive number")
elif num == 0:
   print("Zero")
else:
   print("Negative number")
```

***Result:***

Positive number

## Python Program to Find the Largest Among Three Numbers

```
# Python program to find the largest number among the three input numbers
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))
if (a > b) and (a > c):
  largest = a
elif (b > a) and (b > c):
  largest = b
else:
  largest = c
print("The largest number between",a,",",b,"and",c,"is",largest)
```

***The result is:***

```
Enter first number: 4
Enter second number: 56
```

```
Enter third number: 6
('The largest number between', 4.0, ',', 56.0, 'and', 6.0, 'is', 56.0)
```

### 3.2.4 Nested conditionals

The conditional statement is written within another conditional statement. This is called nested conditionals. Any number of conditional statements can be nested inside one another. To indicate the level of nesting indentation is used. The structure of nested conditionals is shown below.

```
if test expression:
   Body of if
else:
   if test expression:
   Body of if
else:
   if test expression:
   Body of if
:
:
else:
   Body of else
```

**The following programs explain nested conditional.**

***# Program to compare the values of x and y***

```
x=3
y=4
if x == y:
      print 'x and y are equal'
else:
if x < y:
      print 'x is less than y'
else:
      print 'x is greater than y'
```

In this program the variables x and y are assigned with values 3 and 4 respectively. In the first If statement it checks whether x is equal to y. If it is true, then prints x and y are equal. If it is false, it executes the else part. Here, the else part contains the if statement ( Nested if ) checks whether x is lesser than y. If it is true, then it prints x is less than y. If it is false, then it prints x is greater than y that is the statement in else part.

***# program to check whether the number is positive , negative or zero***

```
num = float(input("Enter a number: "))# to get input from user
if num >= 0:
   if num == 0:
      print("Zero")
   else:
      print("Positive number")
else:
   print("Negative number")
```

The outer conditional if num>0 contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two, branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well. The nested conditionals are very difficult to understand quickly; even the indentation of the statements makes the structure clear.

## 3.3 ITERATION

### 3.3.1 State of a variable

It is possible to have more than one assignment for the same variable. The value which is assigned at the last is given to the variable. The new assignment makes an existing variable assigned with new value by replacing the old value.

For example, consider the following multiple assignments.

```
x=5
y=3
x=4
print x
print y
```

***The result is***

```
4
3
```

The variable *x* is first assigned with 5 then it is assigned with 4. The last assignment statement *x*=4 replaces the old value of *x* (*x*=5).

Consider the following program code for **multiple assignments.**

```
x = 5
y = a       # x and y are now equal
x = 3       # x and y are no longer equal
```

```
print x
print y
```

| *The result is* |
|---|
| 3 |
| 5 |

Here the variable *x* is assigned with the value 5. The variable *y* is assigned with the value of *x*. Finally, the value of *x* is updated to 3. So, the value 3 is assigned to *x*. This is called **state** of the variable.

The following code can also be an example for updating the variable

```
x=0
x=x+2
print x
```

| *The result is* |
|---|
| 2 |

Initially *x* is assigned with the value zero. The value of *x* is incremented by the value two. So it is updated as the value 2 (0+2).

### 3.3.2 Looping statements

Normally, program statements are executed sequentially one after another. In some situations, a block of code needs to be executed for several numbers of times. These are repetitive program codes, the computers have to perform to complete tasks. The computers are used to automate the repetitive tasks. Programming languages present various control structures like looping statements which allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times (iterations). The following types of loops are used in Python programming language to handle looping requirements.

| Loop Type | Description |
|---|---|
| while loop | This loop starts with condition checking. It repeats the execution a statement or group of statements while the given condition is TRUE. Every time it tests the condition before executing the loop body. |
| for loop | It executes a statement or a group of statements multiple times and abbreviates the code that manages the loop variable. |
| nested loops | One or more loops used in another loops. |

#### *(i) While loop*

The while loop is used to execute a block of code till the test condition is true. In each iteration, before executing the program code, the test expression will be evaluated. It stops the execution when the test expression is false. The syntax for while loop is given below.

***Syntax of while Loop in Python***

```
while test_expression:
    Body of while
```

The body of the loop is apparent through indentation used. The loop body starts with indentation and the first unintended statement denotes the end. The nonzero value is considered as True in python. The Fig.3.4 depicts the flow of while loop.

***Flowchart:***

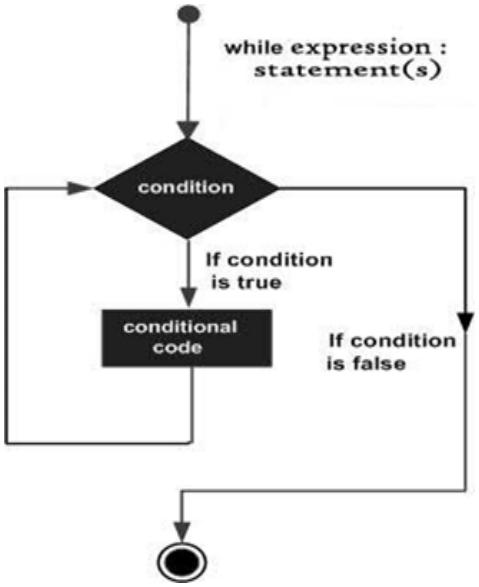



***Fig.3.4. Operation of while statement***

The flow of execution for a while statement is explained as follows:

(1) Evaluate the test condition, yielding True or False.

(2) If the condition is false, exit the while statement and continue execution at the next statement.

(3) If the condition is true, execute the body and then go back to step 1.

This type of flow of execution is called a **loop** as the third step loops back around to the top. The body of the loop should change the value of one or more variables so that ultimately the condition becomes false and the loop terminates. Otherwise the loop will iterate forever, which is called an **infinite loop**.

***Simple program to explain while statement***

```
# Prints out 0,1,2,3,4
count = 0
while count < 5:
   print(count)
   count += 1  # This is the same as count = count + 1
```

***The result is:***

0
1
2
3
4

**The following program may illustrate the working of while loop statements.**

***Program is to add natural numbers***

```
# sum= 1+2+……+n
# Program to add natural numbers upto n
 n = int(input("Enter n: "))       # To take input from the user
 sum = 0                           # initialize sum
 i = 1                             # initialize counter ( loop) variable
while i <= n:                      # while loop begins
   sum = sum + i
   i = i+1                         # update loop variable
print ("The sum is", sum)          # print the sum
```

***Result:***

Enter n: 10
The sum is 55

In this program the variable n is used to get integer from the user. The sum and loop variables are initialized with zero and one respectively. The while loop continues till the loop variable value is lesser than n. In each iteration the sum value is added with loop variable value. The increment statement is used to update the loop variable.

***While with else statement:***

When ' while' statement is used with else , then the else statement is executed only when the condition is false. Both statements executes only when certain condition is satisfied

***Syntax:***

```
While (expression):
    Statement_1
    Statement_2
    .....
Else:
    Statement_3
    Statement_4
    .....
```

In the above syntax, the while block is repeatedly tests the condition for the given expression and executes the first block of statement if it is true otherwise the else clause is executed . It might be tested for the first time and if the loop breaks it will not execute.

*For example,*

```
>>>a=0
>>>b=0
>>>while (a<6):
    b=b+a
        a=a+1
     else:
            print("Sum of first 5 integers:",b)
```

***Output:***

The sum of first 5 integers: 15

## (ii) For loop

The for loop in Python is used to iterate over a sequence of elements (list, tuple, string) or other iterable objects. Iterating over a sequence of items is called traversal. The syntax of for loop is shown below.

***Syntax of for Loop in Python***

```
for val in sequence:
Body of for
```

The val is the loop variable which takes the value of the item inside the sequence on each iteration. The loop continues until the last element is reached in the sequence. The body of for loop is marked using indentation. The Fig. 3.5 illustrates the flow of 'for loop'.

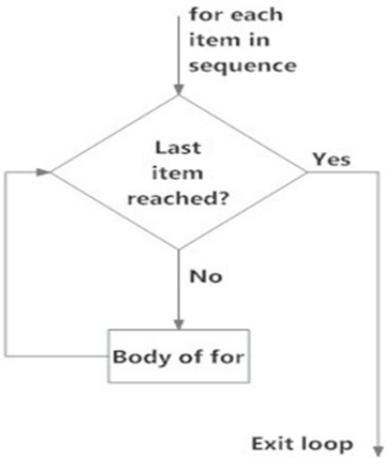



***Fig.3.5. Operation of For loop***

***# Program to find the sum of all numbers stored in a list***

```
# List of numbers
numbers = [3, 2, 5, 7, 9, 1, 4, 6, 8]
# variable to store the sum
total= 0
# iterate over the list
for item in numbers:
        total = total+item      # indentation is used to separate the loop body
# print the result
print ("The total is", total)
```

***Result:***

The total is 45

## For loop using range() function:

The body of the loop is apparent through indentation used. The loop body starts with indentation and the first unintended statement denotes the end. To generate the sequence of numbers range () function is used. The range function can be used in three different ways.

```
Range (n) # generates numbers from 0 to n-1
Range ( m,n) # generates numbers for m to n-1
Range( m,n, x) # generates numers from m upto n-1 with skip counting of x
```

The following program code generates the different sequence of elements.

```
print(range(5))
print(list(range(5)))
print(list(range(2, 5)))
print(list(range(2, 15, 3)))
```

***Result:***

```
range (0, 5)
[0, 1, 2, 3, 4]
[2, 3, 4]
[2, 5, 8, 11, 14]
```

## Python program to print range values

```
# Prints out the numbers 0,1,2,3,4
print 'first loop values'
for x in range(5):
   print(x)
```

```
# Prints out 3,4,5
print 'second loop values'
for x in range(3, 6):
   print(x)
# Prints out 3,5,7
print 'third loop values'
for x in range(3, 8, 2):
   print(x)
```

***The result is:***

```
first loop values
0
1
2
3
4
second loop values
3
4
5
third loop values
3
5
7
```

The *range()* function can be used in for loops to iterate through a sequence of numbers. It can be combined with the *len()* function to iterate through a sequence using indexing. The following example illustrate this.

**# Program to iterate through a list using indexing**

```
programming_languages = ['C', 'C++', 'Python']
# iterate over the list using index

# i is loop variable
for i in range(len(programming_languages)):
        print(" The programming-language is", programming_languages[i]).
```

***The result is :***

```
The programming-language is C
The programming-language is C++
The programming-language is Python
```

Here, range function takes the values from zero to two. The *len()* function is used to find length of the sequence. The following program also used to illustrate the working of for loop statements using range function. The program is to add natural numbers.

```
# sum= 1+2+……+n-1
# Program to add natural numbers upto n-1
n = int(input("Enter n: "))       # To take input from the user
sum = 0                           # initialize sum
    for i in range(1, n):         # while loop begins
  sum = sum + i
 print ("The sum is", sum)        # print the sum
```

| ***Result:*** |
|---|
| Enter n: 10 |
| The sum is 45 |

Here, the range of sequence is from 1 to *n*-1. The loop variable *i* started with 1.In each iteration, the value of loop variable is added with sum. The loop continues for *n*-1 values. The value of the variable n is obtained from the user.

Surprisingly, in python for loop is used with else part. This is optional. The else part of for loop is executed if the items in the sequence used in for loop exhausts. The for loop's else part get executed if no break statement occurs in for loop. The break statement (see later) can be used to stop a for loop. In such case, the else part is ignored. Here is an example to illustrate this concept

```
numbers = [0, 1, 5, 10, 15, 20]
for i in numbers:
   print(i)
else:
   print("No items left.").
```

***The result is:***

```
0
1
5
10
15
20
No items left.
```

In this program the list numbers is created to have the values 0,1,5,10, 15, 20. The for loop is accessing the list through the loop variable sequentially. It prints the numbers one by one in each

iteration until the loop exhausts. When there is no element in the list to access, it will print statement in else part that is No items left.

### Python program to display all the prime numbers within the specified range

```
# change the values of lower and upper for a different result
lower = int(input("Enter lower range: "))
upper = int(input("Enter upper range: "))
print("Prime numbers between", lower, "and", upper,"are:")
for num in range(lower,upper + 1):
  # prime numbers are greater than 1
  if num > 1:
     for i in range(2,num):
        if (num % i) == 0:
           break
     else:
        print(num)
```

| ***The result is :*** |
|---|
| Enter lower range: 100 |
| Enter upper range: 150 |
| ('Prime numbers between', 100, 'and', 150, 'are:') |
| 101 |
| 103 |
| 107 |
| 109 |
| 113 |
| 127 |
| 131 |
| 137 |
| 139 |
| 149 |

In this program, the *lower* and *upper* values are obtained from user. Here, the outer for loop ranges from *lower* to *upper* + 1 value. Since the prime numbers are started with greater than 1, the if loop checks for *num*>1. It executes the inner for loop for the range 2 to *num*-1 for every iteration of the outer for loop. If the variable num is divided by any number from 2 to *num*-1 completely, it breaks the inner for loop and access the next element. If the variable *num* is not divided by any number from 2 to *num*-1, then it prints the number (executes the for-else part).

## Python Program to Find Armstrong Number in an Interval

```
# Program to check Armstrong numbers in some interval
# To take input from the user
lower = int(input("Enter lower range: "))
upper = int(input("Enter upper range: "))
for num in range(lower, upper + 1):
  # order of number
  order = len(str(num))
    # initialize sum
  sum = 0
  # find the sum of the cube of each digit
  temp = num
  while temp > 0:
     digit = temp % 10
     sum += digit ** order
     temp //= 10
  if num == sum:
     print(num)
```

***The result is:***

```
Enter lower range: 100
Enter upper range: 500
153
370
371
407
```

## For loop with else statement

For loop can be used with if statement. This else part is executed when the items in the sequence used in the loop exhausts.

***For example,***

```
series = [0,1,2,3,4]
for i in series
        print(i)
    else:
        print("No items")
```

***Output:***

```
1
2
3
4
No items
```

## For loop using string function:

For loop can also be used in strings.

***For example,***

```
Weeks=[“Sunday”, “Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”]
    for m in months:
print (m)
```

***Output:***

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
```

## Nested Loops:

Nested loops are allowed in Python. The structure of the nested loop is shown in as below:

***Syntax for nested for loop:***

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

In the above syntax, the outer loop is encountered first and the inner nested loop is triggered and then the control returns back to the top of the outerloop completing the second iteration triggering the nested loop. The same process I repeated again until the sequence getw completed or either with the break statement. The following example illustrates the nested for loop .

***Example:***

```
num1=[1,5]
num2=[2,3,4]
```

```
for n1 in num1:
    print(n1)
    for n2 in num2:
        print(n2)
```

***Output:***

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

## (iii) Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

| Control Statement | Description |
|---|---|
| break statement | It terminates or breaks the loop statement and transfers flow of execution to the statement immediately following the loop. |
| continue statement | Causes the loop to skip the rest of its body and immediately retest its condition before reiterating. |
| pass statement | The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |

### Break statement

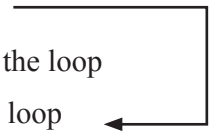
The break statement is used to change or alter the flow of execution. The looping statements iterates till the test expression is true. In some cases, the current iteration of the loop need to be terminated. The break statement is used in this case. The break statement terminates the loop containing it. Therefore, control of the program transfers to the statement immediately after the body of the loop. If break statement is used inside a nested loop (loop inside another loop), break will terminate the innermost loop.

***Syntax of break***

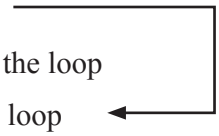
```
break
```

The working of break statement in for loop is explained as follows

```
For var in sequence:
#codes inside the loop
if condition:                # it is true; executes break
        break
# codes inside the loop
#codes outside the loop
```



```
While test expression:
#codes inside the loop
if condition:                # it is true; executes break
        break
# codes inside the loop
#codes outside the loop
```



***A simple program to illustrate break statement***

```
# program to print str
for val in "string":
   if val == "i":
      break
   print(val)
print("The end")
```

***The result is:***

```
s
t
r
The end
```

In this program val is loop variable to access the string. The if condition becomes true when value of the variable val is i. When it is true, it breaks the loop and prints The end.

***Simple program to explain break statement.***

```
# Prints out 1,2,3,4
count = 1
while True:
   print(count)
   count += 1
```

```
    if count >= 5:
      break
```

***The result is :***

1
2
3
4

This program prints the numbers from 1 to 4. When the count value is greater than or equal to 5 it breaks the loop and stops the execution.

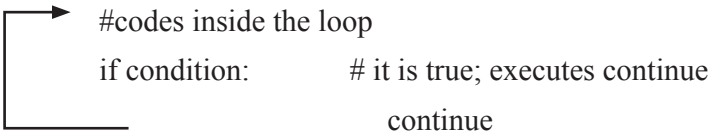
## Continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

***Syntax of Continue:***

```
    continue
```

The working of continue statement in while loop is shown below.




***A simple program to illustrate working of continue statement.***

```
# Program to show the use of continue statement inside loops
    for val in "string":
      if val == "i":
        continue
      print(val)
    print("The end")
```

***The result is :***

s
t
r
n
g
The end

In this program the loop variable *val* prints the character one by one. When the value of variable *val* becomes *i*, it executes continue statement. It skips the rest of the loop body ( not prints values of *i*) and continue from the next element in the string.

***Simple program to print odd numbers***

```
# Prints out only odd numbers 1, 3,5,7,9
    for x in range(10):
      # Check if x is even
      if x % 2 == 0:
        continue
      print(x)
```

***The result is :***

```
1
3
5
7
9
```

This program prints the odd numbers from 1 to 10. The if condition checks value of *x* belongs to even or odd. If it is odd, the if condition fails. So it will print the *x* value. If it is even the continue statement is executed to start from the next element.

## Pass Statement

Pass statement executes nothing. It results in No operation. In Python programming, `pass` is a null statement. The simple difference between a comment and `pass` statement in Python is that, the interpreter ignores a comment entirely, but not the `pass` statement.

***Syntax of pass***

```
pass
```

The pass statement can be used in places where the program code cannot be left as blank. But that can be written in future. The pass is used as placeholders. Pass is used in to construct program codes that do nothing.

***Simple program to explain pass statement***

```
# pass is just a placeholder for
# the for loop body will be implemented in future.
    sequence = {'p', 'a', 's', 's'}
    for val in sequence:
      pass
```

***The result is:***

no output will be displayed

In this program the variable sequence is assigned with some sequence of letters. The loop variable *val* is used to access the values in the sequence. But the loop contains the pass which implies no operation execution. So using pass statement an empty function or class can be created as follows.

```
def function(args):
    pass    # the function does nothing
class example:
    pass    # empty class
```

The function defines nothing the implementation may be written in future. Likewise, the class example does nothing.

## 3.4 VARIABLES AND SCOPE

Scope of a variable specifies the part of a program where a variable is accessible and lifetime of a variable specifies the time period in which the variable has valid memory.

Python uses LEGB Rule for scope resolution. It stands for

**Local -> Enclosed -> Global -> Built-in**

The arrows indicate the direction of the namespace-hierarchy search order.

- **Local** can be inside a function or class method, for example.
- **Enclosed** can be its enclosing function, e.g., if a function is wrapped inside another function.
- **Global** refers to the uppermost level of the executing script itself, and
- **Built-in** are special names that Python reserves for itself.

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions. It will be visible throughout the program. Whereas a local variable is one declared within the body of a function or a block. It is accessible only inside the function and gets deleted at the end of function.

***Example:***

```
# This is a global variable
a = 0
if a == 0:
   # This is still a global variable
   b = 1
def my_function(c):
```

```
    # this is a local variable
    d = 3
    print(c)
    print(d)
# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)
# a and b still exist
print(a)
print(b)
# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

### 3.4.1 Access a function inside a function

Python supports definition of a function inside another function.

```
def test(a):
    def add(b):
        nonlocal a
        a += 1
        return a+b
    return add
func= test(4)
print(func(4))
```

***Sample input/output:***

```
9
```

In this program, the function *add()* is nested inside *test()*. So it could not be called directly. Calling the function *add(4)* will result in a error. First, the outer function *test(4)* is called, which returns the object of inner function *add()*. The return function object can then be used to refer the inner function. The use of *nonlocal* keyword is explained in next section.

### 3.4.2 Scope Rules in Functions

When a name is used in a program, Python creates, changes, or looks up the name in namespace, a place where names live. In Python as names aren't declared ahead of time, it uses the assignment of a name to associate it with a particular namespace. By default functions add an extra namespace layer to the programs. Names assigned inside a function are associated with that function's namespace.

```
# global scope
a = 10              # a and func assigned in module: global
def func(b):        # b and c assigned in function: locals
   # local scope
   c = a + b      # a is not assigned, so it's a global
   return c
print(func(1))  # func in module: result=11
print c             #NameError: name 'z' is not defined
```

In this program, name *a* and *func()* are assigned in module (outside block/function), resulting in global scope. Whereas name *b* and *c* are assigned in function, resulting in local scope. Local variables of a function can only be used inside its function. Hence *c* is not accessible outside *func()*.

*Python 3* introduced the *nonlocal keyword* that allows you to assign to variables in an outer, but non-global, scope. The following programs explicate the difference between using and using nonlocal keyword.

### Without using *nonlocal*

```
x = 0 # x=0 assigned outside any function. (x=0 global)
def outer():
   x = 1 # x=1 assigned inside outer(). (x=1 local to outer())
   def inner():
      x = 2 # x=2 assigned inside inner(). (x=2 local to inner())
      print("inner:", x)
   inner()
   print("outer:", x)
outer()
print("global:", x)
```

***Sample input/output:***

```
inner: 2
outer: 1
global: 0
```

### Using *nonlocal*

```
x = 0
def outer():
   x = 1
```

```
    def inner():
        nonlocal x  # nonlocal permits the access to x=1 assigned in outer() but not to
global
                    #variable x=0
        x = 2
        print("inner:", x)
    inner()
    print("outer:", x)
outer()
print("global:", x)
```

***Sample input/output:***

```
inner: 2
outer: 2
global: 0
```

## Using *Global*

```
x = 0
def outer():
    x = 1
    def inner():
        global x # global permits access to x=0 assigned outside all function but not
to x=l
                 # assigned in outer()
        x = 2
        print("inner:", x)
    inner()
    print("outer:", x)
outer()
print("global:", x)
```

***Sample input/output:***

```
inner: 2
outer: 1
global: 2
```

Functions provide a nested namespace, which localizes the names they use, so that names inside the function won't clash with those names used in other module or function. When a name is used in a function, Python creates the name in local scope, unless it is declared as global in that function using global keyword.

### Without Global

```
def add(value1,value2):
result = value1 + value2
no1=3
no2=2
add(no1,no2)
print result       # NameError: name 'result' is not defined
```

This program results in an error as result, the local variable of function *add()* could not be accessed outside the function.

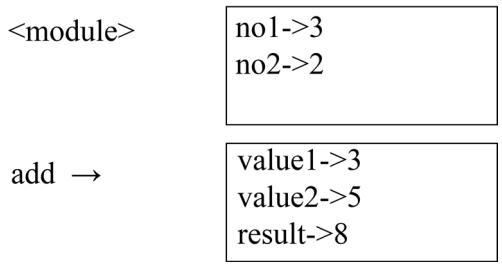



***Figure 3.1. Stack Diagram.***

### Using Global

```
def add(value1,value2):
global result
result = value1 + value2
no1=3
no2=2
add(no1,no2)
print result     #No Error – result is a global variable; it generates the output as 5
```

In this program global keyword specifies *result* as global variable which means that we can access that variable outside the function as well.

## 3.4.3 STACK DIAGRAMS

Stack diagram is a graphical representation of a stack of functions, their variables, and the values they refer to. Each function is represented by a frame. **Frame** is a box in a stack diagram

that represents a function call. It contains local variables and parameters of the function. The stack diagram for preceding program without global is shown in Figure. 3.1.

```
def add(a,b):
        c=a+b
        print c
        sub(a,b)
def sub(x,y):
        z=x-y
        print z
n1=10
n2=5
add(n1,n2)
```

***Sample output:***

```
15
5
```

The stack diagram for this example is shown in Figure.3.2. The order of frames in the stack (top to bottom) represents the order of function call. The flow of execution starts from main module. <module> frame indicates the variables *n1,n2* which are assigned outside any function. The function *add(n1,n2)* was called from main module, which in turn calls *sub(a,b)*. *add* frame includes its parameters *a*, *b* and local variable *c*. *sub* frame includes its parameters *x*, *y* and local variable *z*. Each parameter refers to the same value of its arguments. Hence *a*, *b* refers to the value of its arguments *n1*, *n2* and *x*, *y* refers to the value of its arguments *a*, *b*.

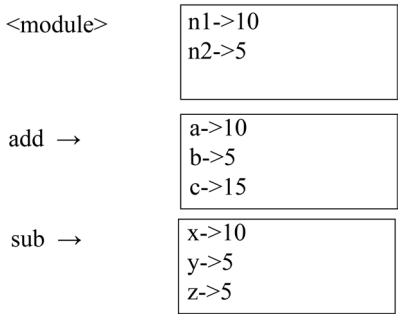



***Figure 3.2. Stack Diagram***

When an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, followed by name of function that called that and so on. The **Traceback** module works with the call stack to produce error messages.

For example, if you try to access c in sub(), you will get an following NameError.

```
Traceback (most recent call last):
  File "main. py", line 11, in <module>
      add(n1,n2)
  File "main.py", line 4, in add
      sub(a,b)
  File "main.py", line 8, in sub
      print c
  NameError: global name 'c' is not defined
```

This list of functions is called a traceback. It specifies the filename, error line number, and functions name that are being executed. The order of the function in Traceback is same as the order of frames in stack diagram.

### 3.4.4 FRUITFUL FUNCTIONS

Fruitful functions are functions that return value. While using fruitful function, the return value must be handled properly by assigning it to a variable or use it as part of expression.

```
import math
x=math.sin(90)+1
print x   # Output is 1.8939966636
```

In a script, calling a fruitful function without assigning the return value will result in loss.

```
import math
math.sin(90)  # no output - as the return value is not assigned
```

### 3.4.5 VOID FUNCTIONS

Void function is a function that always returns *None*. It represents the absence of value.

```
def show():
print 'Welcome!!!'
result=show()
print result
```

***Sample Output:***

```
Welcome!!!
None
```

The value is *None* which is not similar to string "None". *None* is a special value with its own type.

```
print type(None)
```

***Sample Output:***

<type 'NoneType'>

## 3.4.6 IMPORTING MODULES

Module is a Python file that contains collection of related variables, functions, classes and other definitions. Python provides at least three different ways to import modules.

### *Method 1: Using import*

***Syntax:***

```
import X
```

It imports the module X, and creates a reference to that module in the current namespace. Here X.name refers to the things defined in module X.

```
import datetime
tday=datetime.date.today()
print tday
```

***Sample Output:***

2017-04-15

### *Method 2: Importing individual objects*

***Syntax:***

```
from X import *
from X import a,b,c
```

It imports the module X, and creates references in the current namespace to all public objects defined by that module. You can simply use a plain name to refer to things defined in module X. * indicates that all objects under module X can be used. Whereas *a*, *b*, *c* specifies that only *a*, *b*, *c* objects from module X can be used in program.

```
from datetime import date
tday=date.today()
print tday
```

***Sample Output:***

2017-04-15

### *Method 3: __import__('')*

***Syntax:***

```
X = __import__('X')
```

It works like import X, with the following differences:

(1) pass the module name as a string, and

(2) explicitly assign it to a variable in your current namespace.

```
i = __import__('math')
print i.pi
```

***Sample Output:***

```
3.14159265359
```

## 3.4.7 INPUT FROM KEYBOARD

### The Input() Function

To read input from keyboard, Python provides *input()* function with prompt string as a optional parameter. The input of the user will be interpreted. For example, if the user enters an integer value, the input function returns this integer value. If the user on the other hand inputs a list, the function will return a list.

***Example:***

```
age=input("Enter your age:")
print 'Your age is", age, 'Type:", type(age)
color=input("Enter your favorite color:")
print (color, type(color))
Sample Input/Output:
Enter your age:21
Your age is 21 Type: <type 'int'>
Enter your favorite color:['red','green','blue']
(['red', 'green', 'blue'], <type 'list'>)
```

### Input with raw_input()

The function raw_input() does not interpret the input. It always returns the input of the user without changes. This raw input can be changed into the data type needed for the algorithm by using a casting or the *eval* function.

```
age = raw_input("Your age? ")
print(age, type(age))
```

***Sample input/output:***

```
Your age? 38
('38', <type 'str'>)
age = int(raw_input("Your age? "))
print(age, type(age))
```

***Sample input/output:***

Your age? 42

(42, <type 'int'>)

### 3.4.8 INCREMENTAL DEVELOPMENT

Complex programs with larger functions needs more time in debugging. Lengthy debugging sessions could be avoided by a process called incremental development. This process involves adding and testing small amount of code at a time. In general, code could be developed incrementally.

***Example:*** Find the distance between two points, given by the coordinates $(x_1, y_1)$ and $(x_2, y_2)$. By the Pythagorean theorem, the distance is:

$$\text{distance}=\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

**Step 1:** Identify the input parameters and return value of the function. In this example, the inputs are two points which represents four numbers (x1,y1) and (x2,y2). The return value is the distance, a floating point number.

The outline of the function:

```
def distance(x1,y1,x2,y2):
        return 0.0
```

Test the function by calling it with some arguments. This basic version does not compute distance and will always return 0.0.

```
>>>distance(1,3,5,8)
0.0
```

**Step 2:** At this point we have confirmed that the function is syntactically correct, and we can start adding code to the body. In this step, add code to compute the differences x2-x1 and y2-y1.

```
def distance(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  print 'dx is', dx
  print 'dy is', dy

  return 0.0
```

If this code works correctly it should display 'dx is 4' and 'dy is 5'. If so, we could confirm that the function parameters and function computation are working correctly. If not, there are only few lines to check.

**Step 3:** Now compute the sum of squares of dx and dy:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print 'dsquared is: ', dsquared
    return 0.0
```

Run the program at this stage and verify the output (which should be 41).

**Step 4:** Compute square root of the value using math.sqrt().

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

If the return result is correct, the coding is over. Else, print the value of the result inside the function and verify. Once the result obtained is correct, remove the print statement from the function. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product.

In initial steps, we used to add one or two lines of code at a time. Later we extend our coding in to bigger chunks. This method of coding reduces lot of debugging time. The key aspects of the process are:

**3.4.8.1** Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.

**3.4.8.2** Use temporary variables to hold intermediate values so you can display and check them.

**3.4.8.3** Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

## 3.4.9 FUNCTION COMPOSITION

Function composition is the ability to call one function from within another function. It is a way of combining functions such that the result of each function is passed as the argument of the next function. For example, the composition of two functions *f* and *g* is denoted *f(g(x))*. *x* is the argument of *g*, the result of *g* is passed as the argument of *f* and the result of the composition is the result of *f*.

**Example:** Compute area of circle with the given inputs center point (xc,yc) and perimeter point (xp,yp).

***Note:*** First find the radius of circle by computing distance between (xc,yc) and (xp,yp). Later find the area of circle using radius.

```
import math
def distance(x1, y1, x2, y2):
   dx = x2 - x1
   dy = y2 - y1
   dsquared = dx**2 + dy**2
   result = math.sqrt(dsquared)
   return result
def area(radius):
        return (math.pi*radius*radius)
xc=input("Enter xc")
yc=input("Enter yc")
xp=input("Enter xp")
yp=input("Enter yp")
print area(distance(xc,yc,xp,yp))
```

***Sample input/output:***

```
Enter xc10
Enter yc10
Enter xp15
Enter yp10
78.5398163397
```

## 3.4.10 BOOLEAN FUNCTIONS

Functions can return Boolean values (True/False). They are often used in conditional statements.

***Example:*** Program to check for even number.

```
def test_even(n):
    return (n%2==0) # result of (n%2==0) is a Boolean value
no=input("Enter a no.")
if(test_even(no)):    # Boolean functions are often used in conditions
    print 'Even'
else:
    print 'Odd'
```

***Sample input/output:***

Enter a no.3

Odd

## 3.4.11 RECURSION

Recursion is a way of programming in which a function calls itself again and again until a condition is true. A recursive function calls itself and has a termination condition.

### Advantages of recursion

**3.4.11.1** Recursive functions make the code look clean and elegant.

**3.4.11.2** A complex task can be broken down into simpler sub-problems using recursion.

**3.4.11.3** Sequence generation is easier with recursion than using some nested iteration.

### Disadvantages of recursion

(1) Sometimes the logic behind recursion is hard to follow through.

(2) Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

(3) Recursive functions are hard to debug.

### Program for factorial computation using recursion.

The process involved in factorial computation using recursion in show in Figure.3.3. For example 5! computation can be represented as

5!=5*4!; 4!=4*3!; 3!=3*2!; 2!=2*1!; 1!=1*0!; and 0!=1

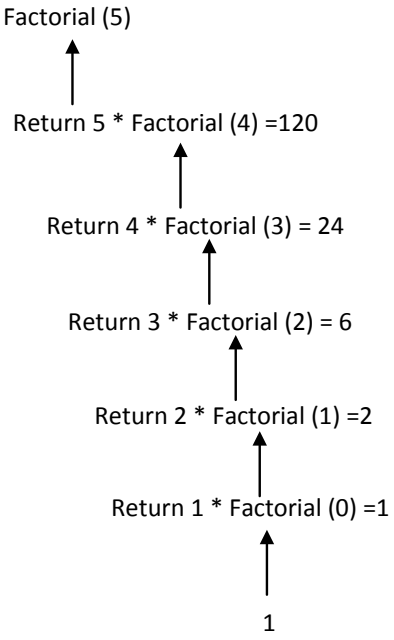



***Figure 3.3. Factorial computation***

```
def factorial( n ):
  if n ==0:   # base case
     return 1
  else:
     return n * factorial( n - 1 )
n=input('Enter a number:")
print factorial(no)
```

***Sample Input/Output:***

```
Enter a number:5
120
```

### *(i) Stack Diagram for Recursive Functions*

Stack diagram for factorial computation using recursion is shown in Figure.3.4. For each recursive call, a frame is added in stack. The top of the stack is _ _main_ _ frame with variable *n*. The value of *n* varies in each frame. The bottom frame where *n*=0 is called the **base case**.

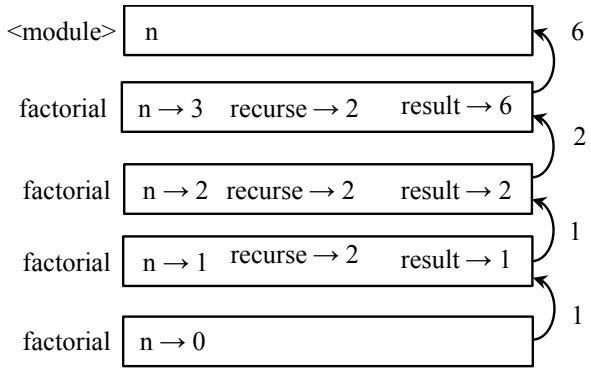
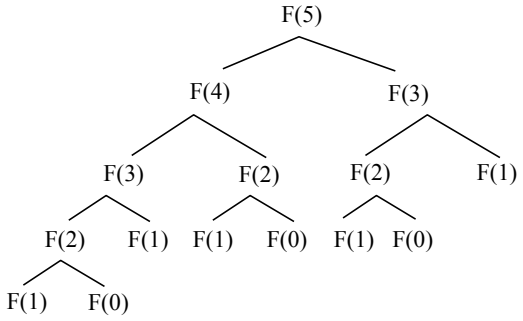



***Figure 3.4. Stack Diagram for Recursion***

## Program to display Fibonacci sequence using recursion.

**Note:** A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8....

The first two terms are 0 and 1. All other terms are obtained by adding the preceding two terms. i.e. the nth term is the sum of $(n-1)^{th}$ and $(n-2)^{th}$ term.



```
# Recursive Function Beginning
def Fibonacci_series(Number):
    if(Number == 0):
        return 0
    elif(Number == 1):
        return 1
    else:
        return (Fibonacci_series(Number - 2)+ Fibonacci_series(Number - 1))
 # End of the Function
# Fibonacci series will start at 0 and travel upto below number
Number = int(input("\nPlease Enter the Range Number: "))
# Find & Displaying Fibonacci series
for Num in range(0, Number):
    print(Fibonacci_series(Num))
```

***Sample input/output:***

```
Please Enter the Range Number: 5
0
1
1
2
3
```

### *(ii) Infinite Recursion*

**Infinite recursion** happens when recursive function fails to reach base case. The recursive call continues forever and the program never terminates.

```
def show():
    show()
show()
```

The program with infinite recursion runs repeatedly and terminates when the maximum recursion depth is reached. Python reports the following error message on infinite recursion.

```
File "main.py", line 2, in show
  show()
 File "main.py", line 2, in show
  show()
```

File “main.py”, line 2, in show

…….

……..

RuntimeError: maximum recursion depth exceeded

## 3.5 STRINGS

A string is a sequence of zero or more characters. An empty string contains no characters and has a length 0.The indices of a string’s characters are numbered from 0 from the left end and numbered from -1 from the right end. For a string ‘WELCOME’, its index values are shown in the following table:

| Character | W | E | L | C | O | M | E |
|---|---|---|---|---|---|---|---|
| Index from left end | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Index from right end | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

### 3.5.1 Subscript Operator

[ ] is a subscript operator that is used to access the characters one at a time.

***Syntax:***

< *stringname*>[<*index*>]

Index within [ ] indicates the position of the particular character in the string and it must be an integer expression.

***Sample Code***

```
s= “HELLO”
print s[0]        # prints H
print s[3]        # prints L
```

The subscript operator can be used within loops as shown in the following code:

```
s= “HELLO PYTHON!”
for index in xrange(len(s)):
   print s[index], “ is at index “, index
```

| ***Output*** |
|---|
| H is at index 0 |
| E is at index 1 |
| L is at index 2 |
| L is at index 3 |
| O is at index 4 |
| is at index 5 |

| P is at index 6 |
|---|
| Y is at index 7 |
| T is at index 8 |
| H is at index 9 |
| O is at index 10 |
| N is at index 11 |
| ! is at index 12 |

### 3.5.2 Len Function

len is a built-in function that returns the number of characters in a string.

***Sample Code***

```
s= "HELLO PYTHON!"
print "Length is ", len(s)
```

***Sample Output***

```
Length is 13
```

### 3.5.3 in operator

in is a boolean operator that takes two strings as its operands and returns True if the first string appears as a substring in the second string.

***Sample Code and Output***

```
print "PY" in "HELLO PYTHON!"     # prints True
print "NO" in "HELLO PYTHON!"     # prints False
```

### 3.5.4 String slices

A part of a string is called slice. The operator [m:n] returns the part of the string from $m^{th}$ index to $n^{th}$ index, including the character at $m^{th}$ index but excluding the character at $n^{th}$ index. If the first index is omitted, the slice starts at the beginning of the string. If the second index is omitted, the slice goes to the end of the string. If the first index is greater than or equals to the second, the slice is an empty string. If both indices are omitted, the slice is a given string itself.

***Sample Code***

```
s="Hello, Python!"
print s[0]
print s[1:4]
print s[:5]
print s[7:]
print s[3:3]
```

```
print s[5:2]
print s[:]
print s[-3:]
print s[:-3]
```

***Sample Output***

```
H
ell
Hello
Python!
Hello, Python!
on!
Hello, Pyth
```

### 3.5.5 Immutability

The string is an **immutable data structure**. This means that its characters can be accessed but the string cannot be modified.

***Sample Code and Output:***

```
s='hello, Python!'
s[0]='H'
TypeError: 'str' object does not support item assignment
```

***Sample Code and Output:***

```
s='hello, Python!'
s1='H'+s[1:len(s)]   # concatenates a new first letter onto a slice of s
print s              # prints hello, Python!
print s1             # prints Hello, Python!
```

### 3.5.6 String Methods

Python consists of the following built-in methods for manipulating strings:

| S.No | Method | Description |
|---|---|---|
| 1 | s.capitalize() | Capitalizes only the first letter of a given string *s* |
| 2 | s.center(width, fillchar) | Returns a space-padded string where a given string *s* is centered within the specified width. |
| 3 | s.count(substr) | Return the number of occurrences of *substr* within a given string *s* |

| S.No | Method | Description |
|---|---|---|
| 4 | s.endswith(substr) | Boolean function that returns True, if a string *s* ends with *substr*. Else, returns False. |
| 5 | s.find(substr) | Returns the smallest index in *s*, if *substr* is found within s. Else, returns -1. |
| 6 | s.index(substr) | Similar to find(), but raises an exception if *substr* is not found. |
| 7 | s.isalnum() | Boolean function that returns True, if *s* is nonempty and all characters are alphanumeric. Else, returns False. |
| 8 | s.isalpha() | Boolean function that returns true, if *s* is nonempty and all characters are alphabet. Else, returns False. |
| 9 | s.isdigit() | Boolean function that returns true, if *s* is nonempty and all characters are digits. Else, returns False. |
| 10 | s.islower() | Boolean function that returns true, if *s* is nonempty and all characters are in lowercase. Else, returns False. |
| 11 | s.isupper() | Boolean function that returns true, if *s* is nonempty and all characters are in uppercase. Else, returns False. |
| 12 | sp.join(seq) | Returns a string that is the concatenation of the set of strings in the given sequence (*seq*). Separator between the elements is *sp*. |
| 13 | len(s) | Returns the length of the string *s* |
| 14 | s.lower() | Returns a copy of *s* where all characters are in lowercase. |
| 15 | max(s) | Returns the biggest alphabetical character from the string *s*. |
| 16 | min(s) | Returns the smallest alphabetical character from the string *s*. |
| 17 | s.replace(old, new) | Returns a copy of *s* after replacing all occurrences of a substring *old* by a substring *new*. |
| 18 | s.split() | Returns a list of the words in *s*, using any whitespace as a delimiter. |
| 19 | s.startswith(substr) | Boolean function that returns True, if a string *s* starts with *substr*. Else, returns False. |
| 20 | s.swapcase() | Returns a copy of *s* where all characters are case-inverted. |
| 21 | title() | Returns a copy of *s* where all words begin with uppercase characters. |
| 22 | s.upper() | Returns a copy of *s* where all characters are in uppercase. |
| 23 | s.istitle() | Boolean function that returns True, if a string *s* is in title-case. Else, returns False. |

Explanation with sample coding and the corresponding output is as follows:

| Sample Code | Sample Output | Explanation |
|---|---|---|
| s='hello, Python!'<br>print "String is: ", s<br>print "Length: ", len(s) | String is: hello, Python!<br>Length: 14 | s contains the string 'hello, Python!' with length 14. |
| print s.capitalize() | Hello, python! | The first letter of a given string is capitalized. |
| print s.center(20) | hello, Python! | The given string is centered with width 20. |
| print s.center(20, 'a') | aaahello, Python!aaa | The given string is centered and space-padded with width 20 and fill character 'a'. |
| sub = 'l';<br>print s.count(sub) | 2 | Count of 'l' in 'hello, Python!'. |
| print s.count(sub, 3, 14) | 1 | Count of 'l' in 'hello, Python!' from index 3. |
| print s.endswith('on') | False | 'hello, Python!' does not end with 'on'. |
| print s.endswith('on!') | True | 'hello, Python!' ends with 'on!'. |
| print s.find('Pyth') | 7 | 'Pyth' is found at index 7 in 'hello, Python!'. |
| print s.find('pyth') | -1 | 'pyth' is not found in 'hello, Python!'. |
| print s.index('Pyth') | 7 | 'Pyth' is found at index 7 in 'hello, Python!'. |
| print s.isalnum() | False | 'hello, Python!' does not include a number. |
| print s.isalpha() | False | 'hello, Python!' includes punctuation symbols also. |
| s1='15Aug1947'<br>print s1.isalnum() | True | No space within the string.<br>'15Aug1947' includes both numbers and alphabets. |
| print s1.isalpha() | False | '15Aug1947' includes numbers also. |
| print s1.isdigit() | False | '15Aug1947' includes alphabets also. |
| s2 = '123';<br>print s2.isdigit() | True | Only digit in this string.<br>'123' includes digits alone. |

| | | |
|---|---|---|
| print s.islower() | False | 'P' is in uppercase in 'hello, Python!'. |
| print 'abc'.islower() | True | All characters are in lowercase in'abc'. |
| print s.isupper() | False | 'P'alone is in uppercase in 'hello, Python!'. |
| print 'ABC'.isupper() | True | All characters are in uppercase in'ABC'. |
| print s.lower() | hello, python! | All characters of 'hello, Python!' are converted into lowercase. |
| print s.upper() | HELLO, PYTHON! | All characters of 'hello, Python!' are converted into uppercase. |
| sj = '->'<br>seq = ('a', 'b', 'c') #<br>print sj.join( seq ) | a->b->c | This is sequence of strings.<br>'a', 'b', and 'c' are concatenated with -> |
| str = 'giraffe'<br>print max(str)<br>print min(str) | r<br>a | 'r' is the biggest character in 'giraffe'.<br>'a' is the smallest character in 'giraffe'. |
| print 'this is a tree'. replace('is', 'was') | thwas was a tree | Every 'is' is replaced by 'was' from 'this is a tree'. |
| print s.split( ) | ['hello,', 'Python!'] | 'hello, Python!'has two words 'hello,' and 'Python!', which are separated by a whitespace. |
| print s.startswith( 'hell' ) | True | 'hello, Python!' starts with 'hell'. |
| print s.swapcase() | HELLO, pYTHON! | 'hello, Python!' is case-inverted. |
| print s.title() | Hello, Python! | First character of every word in 'hello, Python!' is capitalized. |
| print s.istitle() | False | In 'hello, Python!', 'h' is in lowercase. |

### 3.5.7 String module

The string module of Python is a file that offers additional functions, classes and variables to manipulate standard strings. But, some methods that are available in the standard data structure are not found in the string module (For e.g., isalpha()).

When we import a module, the following syntax is used:

import $module_1$[, $module_2$[,... $module_n$]

When an 'import' statement is encountered by the interpreter, the corresponding module(s) is imported if it is available in the search path.

### *dir( ) Function*

It returns a sorted list of strings that includes the names of all modules, functions and variables that are defined in a module.

***Sample Code using dir()***

```
import string
content = dir(string)
print content
```

***Sample Output:***

['Formatter', 'Template', '_TemplateMetaclass', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '_float', '_idmap', '_idmapL', '_int', '_long', '_multimap', '_re', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'atof', 'atof_error', 'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize', 'capwords', 'center', 'count', 'digits', 'expandtabs', 'find', 'hexdigits', 'index', 'index_error', 'join', 'joinfields', 'letters', 'ljust', 'lower', 'lowercase', 'lstrip', 'maketrans', 'octdigits', 'printable', 'punctuation', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit', 'rstrip', 'split', 'splitfields', 'strip', 'swapcase', 'translate', 'upper', 'uppercase', 'whitespace', 'zfill']

Here, the special string variable '*__name__*' holds module's name, and '*__file__*' holds the filename from which the corresponding module is loaded.

### *Sample Code for manipulating strings*

```
import string
txt = "Python String Module"
print "upper", string.upper(txt)
print "lower", string.lower(txt)
print "split", string.split(txt)
print "join", string.join(string.split(txt), "+")
print "replace", string.replace(txt, "Python", "Java")
print "find", string.find(txt, "Python")
print "find", string.find(txt, "Java")
print "count", string.count(txt, "n")
```

***Sample Output:***

```
upper PYTHON STRING MODULE
lower python string module
split ['Python', 'String', 'Module']
join Python+String+Module
replace Java String Module
find 0
find -1
count 2
```

***Sample Code 3 for converting strings to numbers***

```
import string
print int("5217"),
print string.atoi("5217"),
print string.atoi("12141", 8), # octal
print string.atoi("1461", 16), # hexadecimal
print string.atoi("5217", 0),
print string.atoi("05217", 0),
print string.atoi("0x5217", 0)
print float("5217"),
print string.atof("1"),
print string.atof("1.23e5")
```

***Sample Output:***

```
5217 5217 5217 5217 4711
5217 2703 21015
5217.0 1.0 123000.0
```

**atoi()** function uses the second argument (optional) for identifying the number base. If the base is zero, atoi() examines the first few characters to identify the base. If '0x', the base is 16 (hexadecimal), and if '0', the base is 8 (octal). The default base is 10 (decimal).

## 3.6 ARRAYS

Python lists can store values of different data types. But, arrays in python can only store values of same data type. Array is not a fundamental data type in Python. So, the standard 'array' module has to be imported as:

```
from array import *
```

Then an array has to be declared as:

arrayID = array(typecode, [Initializers])

Here, 'arrayID' is the name of an array, 'typecode' is the type of array and 'Initializers' are the values with which an array is initialized.

***Example:***

my_array = array('i',[1,2,3,4])

***Table 3.1 Typescodes in Python arrays***

| Typecode | Description |
|---|---|
| 'b' | signed integer of size 1 byte |
| 'B' | unsigned integer of size 1 byte |
| 'c' | character of size 1 byte |
| 'u' | unicode character of size 2 bytes |
| 'h' | signed integer of size 2 bytes |
| 'H' | unsigned integer of size 2 bytes |
| 'i' | signed integer of size 2 bytes |
| 'I' | unsigned integer of size 2 bytes |
| 'w' | unicode character of size 4 bytes |
| 'l' | signed integer of size 4 bytes |
| 'L' | unsigned integer of size 4 bytes |
| 'f' | floating point of size 4 bytes |
| 'd' | floating point of size 8 bytes |

***Sample Code***

```
from array import *
myArray = array('i', [1,2,3,4,5])
for i in myArray:
   print i
```

***Sample Output:***

```
1
2
3
4
5
```

### 3.6.1 Lists as Arrays

As Python does not have a native array data structure, it is required to load the *numpy* python module. Both the visual module and the *pylab* module load *numpy*. But, if we use plain python, there is no array. Since arrays look a lot like a list, lists can be employed as arrays. However, arrays (instead of lists) should be used to perform arithmetic operations. Moreover, arrays will store data more compactly and efficiently.

In Python, a one-dimensional array can easily be represented as a list. The following code initializes an array 'myArray' and attempts to find the largest of its items, using the concept of lists in Python.

```
myArray=[45, 23, 76, 12, 33]
print 'The given elements are'
for i in range(len(myArray)):
   print myArray[i]
m=0
for i in range(len(myArray)):
 if m<myArray[i]:
  m=myArray[i]
print 'The largest is', m
```

***Sample Output:***

```
The given elements are
45
23
76
12
33
The largest is 76
```

A 2D array can be created using lists within list. The following code creates the 2×2 matrix as [[1,2],[3,4]] with the list [1,2] representing the first row and the list [3,4] representing the second row.

***Sample Code***

```
myArray=[[1,2],[3,4]]

for i in range(len(myArray)):
    for j in range(len(myArray[i])):
        print myArray[i][j]
```

***Sample Output:***

1
2
3
4

In a similar manner, a 3×2 matrix with elements [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’] is created and displayed along with their indices in the following code:

***Sample Code***

```
myArray=[[‘a’,’b’],[‘c’,’d’],[‘e’,’f’]]
for i in range(len(myArray)):
    for j in range(len(myArray[i])):
        print ‘[‘,i,’ ‘,j,’]’,myArray[i][j]
```

***Sample Output:***

```
[ 0  0 ] a
[ 0  1 ] b
[ 1  0 ] c
[ 1  1 ] d
[ 2  0 ] e
[ 2  1 ] f
```

## 3.7 ILLUSTRATIVE PROGRAMS

### 3.7.1 Program to compute sum of array elements from array

```
import * # import array module def sum_array():
my_array=array(‘i’,[ ]) # creating array of integers with no values
n=int(input(“Enter no. of elements in array”)
for i in range(0,n):
v=int(input(“Enter no.”)
my_array.insert(i,v) # add values in array (i-index pos;v-value)
sum=0
for i in my_array():
sum=sum+i
print(sum) sum_array()
```

***Sample input/output:***

Enter no. of elements in array 5 Enter no. 1
Enter no. 2
Enter no. 3
Enter no. 4
Enter no. 5
15

### 3.7.2. Program to compute gcd, square root, and exponentiation of a given number

```
import math def compute():
print(math.gcd(12,18)) print(math.sqrt(100)) print(math.exp(2))
compute()
```

***Sample input/output:***

6
10.0
7.38905609893065

### 3.7.3 Sequential search

**Sequential Search**: In computer science, **linear search** or sequential search is a method for finding a particular value in a list that checks each element in sequence until the desired element is found or the list is exhausted. The list need not be ordered.

| 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

***Case 1:***

Search value = 50

Step 1: Compare 50 with value at index 0

Step 2: Compare 50 with value at index 1

Step 3: Compare 50 with value at index 2

Step 4: Compare 50 with value at index 3 (Success)

***Case 2:***

Search value = 70

Step 1: Compare 70 with value at index 0

Step 2: Compare 70 with value at index 1

Step 3: Compare 70 with value at index 2

Step 4: Compare 70 with value at index 3

Step 5: Compare 70 with value at index 4

Failure

**Python program for sequential search**

```
def seq_search(List,item):        # define function for sequential search
        pos=0
        found=False
        while pos<len(List) and not found:      # loop through the list elements
                if(int(List[pos])==item):
                        found=true
                pos=pos+1
        return(found,pos)
ls=[ ]
n=int(input("Enter no. of elements in the list")
for i in range(0,n):
        v=input("Enter no.")                    # Input number
        ls.append(v)
print(ls)
key=input("Enter key to be searched")           # Read search key
print(seq_search(ls,key))                       # call function for sequential search
```

***Sample input/output:***

```
Enter no. of elements in the list 3
Enter no. 1
Enter no. 2
Enter no. 3
['1', '2', '3']
Enter key to be searched 2
(True, 2)
```

### 3.7.4 Binary search

A binary search or half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as divide-and-conquer search algorithm and executes in logarithmic time. The basic operation involved in binary search is as follows.

if ( value == middle element )

value is found

else if ( value < middle element )

search left half of list with the same method

else

search right half of list with the same method

***Case 1: val == a[mid]***

*val = 11*

*low = 0, high = 8*

*mid = (0 + 8) / 2 = 4*

| 1 | 2 | 7 | 9 | 11 | 13 | 17 | 23 | 27 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| ↑ | | | | ↑ | | | | ↑ |
| low | | | | mid | | | | high |

In this case, the key value we are looking for is located in middle position of the array. The search operation can stop here and an appropriate value can be returned back.

***Case 2: val > a[mid]***

*val = 23*

*low = 0, high = 8*

*mid = (0 + 8) / 2 = 4*

*new low = mid + 1 = 5*

| 1 | 2 | 7 | 9 | 11 | 13 | 17 | 23 | 27 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| ↑ | | | | ↑ | ↑ | | | ↑ |
| low | | | | mid | new low | | | high |

***Figure 3.3. Case 2-Positon updation***

In this case, the value 23 is greater than the middle. So it might be present in the right half of the array. The right half part starts from position 5 to position 8. It is shown in Figure. 3.3 that the *new low* is at position 5. With these new *low* and *high* positions, the algorithm is applied to this right half again.

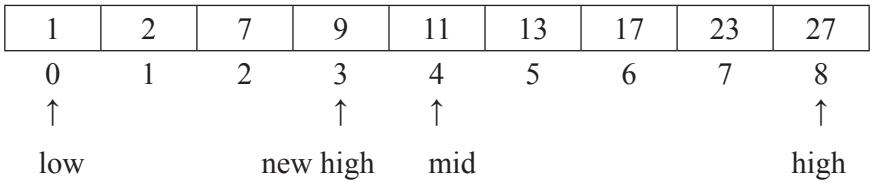
***Case 3: val < a[mid]***

*val = 7*

*low = 0, high = 8*

*mid = (0 + 8) / 2 = 4*

*new high = mid - 1 =5*




The value to be searched in this case is 7 which is less than the value at *mid* position. As the array is sorted, the left half might have the search key. The left half of the array will start from the same starting position low=0 but the *high* position is going to be changed to *mid-1 i.e. 3*. This algorithm is executed again on this left half.

***Python program for binary search.***

```
def binary_search(item_list,item):       # define function for binary search
    first = 0
    last = len(item_list)-1
    found = False
    while( first<=last and not found):
        mid = (first + last)//2          # compute middle position
        if item_list[mid] == item :   # compare with middle value
            found = True
        else:
            if item < item_list[mid]:  # continue search in left half
                last = mid - 1
            else:
                first = mid + 1    # continue search in right half
    return found
print(binary_search([1,2,3,5,8], 6))    # function call
print(binary_search([1,2,3,5,8], 5))
```

***Sample input/output:***

```
False
True
```

## Some More Examples:

### 3.7.5 Find Sum of Natural Numbers Using Recursion

```
def rec_sum(n):
if n<=1:
    return n
```

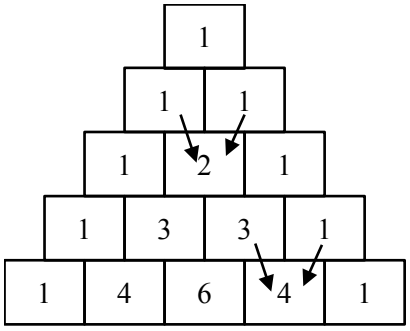
```
else:
        return n+rec_sum(n-1)
no=input("Enter a no.")
print 'Sum:',rec_sum(no)
```

***Sample input/output:***

Enter a no.5
Sum: 15

### 3.7.6 Python function that that prints out the first n rows of Pascal's triangle.

***Sample Pascal's triangle :***




***Using Recursion:***

```
def pascal(n):
   if n == 1:
      return [ [1] ]
   else:
      result = pascal(n-1)
      lastRow = result[-1]
      result.append( [ (a+b) for a,b in zip([0]+lastRow, lastRow+[0]) ] )
      return result
def pretty(tree):
   if len(tree) == 0: return ''
   line = '  ' * len(tree)
   for cell in tree[0]:
      line += '  %2i' % cell
   return line + "\n" + pretty(tree[1:])

print pretty(pascal(int(6))
```

***Sample input/output:***

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

***Using iteration:***

```
def pascal_triangle(n):
  trow = [1]
  y = [0]
  for x in range(max(n,0)):
    print(trow)
    trow=[l+r for l,r in zip(trow+y, y+trow)]
  return n>=1
pascal_triangle(6)
```

| ***Sample input/output:*** |
|---|
| [1] |
| [1, 1] |
| [1, 2, 1] |
| [1, 3, 3, 1] |
| [1, 4, 6, 4, 1] |
| [1, 5, 10, 10, 5, 1] |

# TWO MARKS QUESTION & ANSWER

**1. What are the types of control structures available in Python?**

Python programming language provides following types of decision making statements.

| Statement | Description |
|---|---|
| if statements | An if statement consists of a boolean expression followed by one or more statements. |
| if...else statements | An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE. |
| nested if statements | You can use one if or else if statement inside another if or else if statement(s). |

**2. What are the looping statements available in Python?**

Python programming language provides following types of loops to handle looping requirements.

| Loop Type | Description |
|---|---|
| while loop | Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| for loop | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| nested loops | You can use one or more loop inside any another while, for or do.. while loop. |

**3. What is the difference between break and continue statement?**

| BREAK | CONTINUE |
|---|---|
| To stop the current iteration and get out of that block | To stop the current iteration and continue for next iteration |
| Break; keyword is used | Continue; keyword used |
| It is used in switch case, for, while, do | It is used in for, while |
| Statements after the BREAK statement will not be executed | Statements after CONTINUE statement will not be executed in current iteration |

**4. What is a global variable?**

A global variable is a variable declared in the main body of the source code, outside all functions. It will be visible throughout the program. Scope of this variable is available in all the functions. Life as long as the program's execution doesn't come to an end.

***Example:***

```
# This function uses global variable s
        def f():
           print s
```

```
# Global scope
s = «Example»
f()
```

**5.. Explain in short about state diagram.**

Stack diagram is a graphical representation of a stack of functions, their variables, and the values they refer to. Each function is represented by a frame. Frame is a box in a stack diagram that represents a function call. It contains local variables and parameters of the function.

***Example:***

```
def sub(x,y):
z=x-y
print z
n1=10
n2=5
sub(n1,n2)
```

<module>

| n1->10 |
|---|
| n2->5 |

sub

| x->10 |
|---|
| y->5 |

**6. What is fruitful function?**

Fruitful functions are functions that return value. While using fruitful function, the return value must be handled properly by assigning it to a variable or use it as part of expression.

***Example:***

```
import math
x=math.sin(90)+1
print x   # Output is 1.8939966636
```

**7. What is void function?**

Void function is a function that always returns None. It represents the absence of value.

```
def show():
    print 'Welcome!!!'
result=show()
print result
```

**8. Define Recursion.**

Recursion is a way of programming in which a function calls itself again and again until a condition is true. A recursive function calls itself and has a termination condition.

***Example:***

```
def show():
    show()
    return
```

**9. What are the advantages and disadvantages of recursion.**

***Advantages of recursion***

(1) Recursive functions make the code look clean and elegant.

(2) A complex task can be broken down into simpler sub-problems using recursion.

(3) Sequence generation is easier with recursion than using some nested iteration.

***Disadvantages of recursion***

(1) Sometimes the logic behind recursion is hard to follow through.

(2) Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

(3) Recursive functions are hard to debug

**10. Define Module. What are the ways to import modules in Python?**

Module is a Python file that contains collection of related variables, functions, classes and other definitions.

Python provides the following three different ways to import modules:

(1) Using import

(2) Importing individual objects

(3) __import__(‘’)

**11. What is a string?**

A string is a sequence of zero or more characters. An empty string contains no characters and has a length 0.The indices of a string’s characters are numbered from 0 from the left end and numbered from -1 from the right end.

What is the output of the following python code?

```
s=”All the best!”
print s
print s[0]
print s[2:5]
print s*2
```

***Output:***

```
All the best!
A
l t
All the best!All the best!
```

**12. Define String slice.**

A part of a string is called slice. The operator [m:n] returns the part of the string from $m^{th}$ index to nth index, including the character at mth index but excluding the character at nth index.

- If the first index is omitted, the slice starts at the beginning of the string.
- If the second index is omitted, the slice goes to the end of the string.
- If the first index is greater than or equals to the second, the slice is an empty string.
- If both indices are omitted, the slice is a given string itself.

**13. Why do we call Python string as immutable?**

Python string is called as immutable since its characters can be accessed but the string cannot be modified.

***Eg:***

```
s='hello, Python!'
s[0]='H'
TypeError: 'str' object does not support item assignment
```

**14. List out any four methods of Python strings:**

- capitalize()
- isupper()
- islower()
- swapcase()

**15. What is the use of string module in Python?**

The string module of Python is a file that offers additional functions, classes and variables to manipulate standard strings.

**16. How will you check in a string that all characters are alphanumeric?**

*isalnum()* can be used which returns true if string has at least 1 character and all other characters are alphanumeric.

**17. What is an array? (or) Define array.**

An array is a collection of same data type elements. All elements are stored in continuous locations. Array index always start from '0'. It is represented using [ ] – square brackets

Types of Array:

(1) One dimensional array

(2) Two dimensional array

(3) Multidimensional array

**18. Define List.**

A list is a sequence of any type of values and can be created as a set of comma-separated values within square brackets. The values in a list are called elements or items.

***Eg:***

```
list1 = ['Ram', 'Chennai', 2017]        # list of different types of elements
```