Unit-1: Review of Python & Concept of Oops

Chapter-1 : Review of Python

Learning Objectives:

At the end of this chapter the students will be able to understand:

- Interactive Mode
- Script Mode
- Data Types
- Functions in Python
- Sequential Statement
- Selective Statements
- Looping Statements
- String and String Methods
- List and List Methods
- Tuple and Tuple Methods
- Dictionary and Dictionary Methods

Introduction:

We have learnt Python programming language in the 11th class and continue to learn the same language program in class 12th also. We also know that Python is a high level language and we need to have Python interpreter installed in our computer to write and run Python program. Python is also considered as an interpreted language because Python programs are executed by an interpreter. We also learn that Python shell can be used in two ways, viz., interactive mode and script mode.

Interactive Mode: Interactive Mode, as the name suggests, allows us to interact with OS. Hear, when we type Python statement, interpreter displays the result(s) immediately. That means, when we type Python expression / statement / command after the prompt (>>>), the Python immediately responses with the output of it. Let's see what will happen when we type print "WELCOME TO PYTHON PROGRAMMING" after the prompt.

```
>>>print "WELCOME TO PYTHON PROGRAMMING"
WELCOME TO PYTHON PROGRAMMING
```

Example:

```
>>> print 5+10
15
>>> x=10
>>> y=20
>>> print x*y
200
```

Script Mode: In script mode, we type Python program in a file and then use interpreter to execute the content of the file. Working in interactive mode is convenient for beginners and for testing small pieces of code, as one can test them immediately. But for coding of more than few lines, we should always save our code so that it can be modified and reused.

Python, in interactive mode, is good enough to learn, experiment or explore, but its only drawback is that we cannot save the statements and have to retype all the statements once again to re-run them.

Example: Input any two numbers and to find Quotient and Remainder.

Code: (Script mode)

```
a = input ("Enter first number")
b = input ("Enter second number")
print "Quotient", a/b
print "Remainder", a%b
```

Output: (Interactive Mode)

```
Enter first number10
Enter second number3
Quotient 3
Remainder 1
```

Variables and Types: One of the most powerful features of a programming language is the ability to manipulate variables. When we create a program, we often like to store values so that it can be used later. We use objects (variables) to capture data, which then can be manipulated by computer to provide information. By now, we know that object/variable is a name which refers to a value.

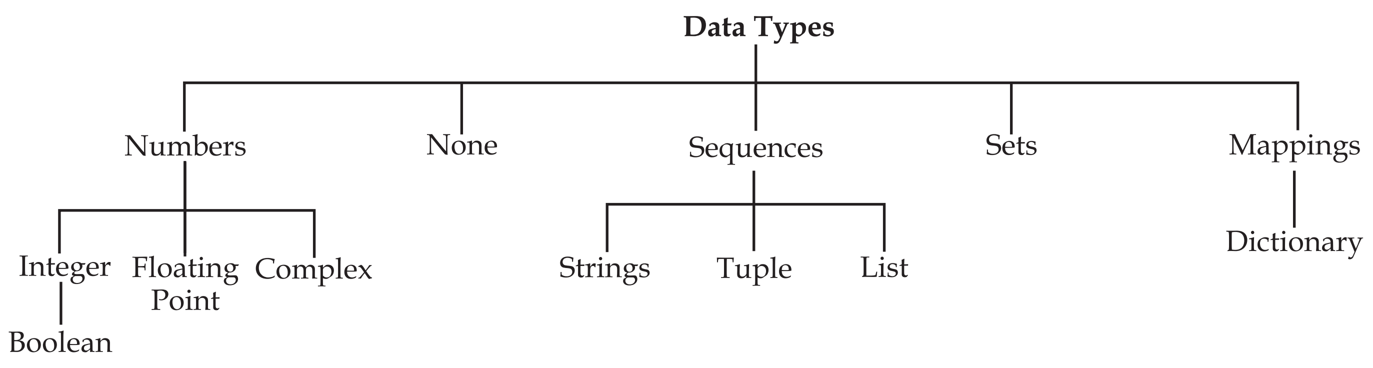
Every object has:

- An Identity,
- A type, and
- A value.

A. **Identity** of the object is its address in memory and does not get change once it is created. We may know it by typing id (variable)

We would be referring to objects as variable for now.

B. **Type** (i.e data type) is a set of values, and the allowable operations on those values. It can be one of the following:



1. **Number:** Number data type stores Numerical Values. This data type is immutable i.e. value of its object cannot be changed. Numbers are of three different types:
   - Integer & Long (to store whole numbers i.e. decimal digits without fraction part)
   - Float/floating point (to store numbers with fraction part)
   - Complex (to store real and imaginary part)
2. **None:** This is special data type with a single value. It is used to signify the absence of value/false in a situation. It is represented by None.
3. **Sequence:** A sequence is an ordered collection of items, indexed by positive integers. It is a combination of mutable (a mutable variable is one, whose value may change) and immutable (an immutable variable is one, whose value may not change) data types. There are three types of sequence data type available in Python, they are Strings, Lists & Tuples.

   3.1 **String-** is an ordered sequence of letters/characters. They are enclosed in single quotes (' ') or double quotes (" "). The quotes are not part of string. They only tell the computer about where the string constant begins and ends. They can have any character or sign, including space in them. These are **immutable.** A string with length 1 represents a character in Python.

   3.2 **Lists:** List is also a sequence of values of any type. Values in the list are called elements / items. These are **mutable** and indexed/ordered. List is enclosed in square brackets ([]).

   3.3 **Tuples:** Tuples are a sequence of values of any type and are indexed by integers. They are immutable. Tuples are enclosed in ().
4. **Sets:** Set is unordered collection of values of any type with no duplicate entry. It is immutable.
5. **Mapping:** This data type is unordered and mutable. Dictionaries fall under Mappings.

   5.1 **Dictionaries:** It can store any number of python objects. What they store is a **key -value** pairs, which are accessed using key. Dictionary is enclosed in curly brackets ({}).

C. **Value:** Value is any number or a letter or string. To bind value to a variable, we use assignment operator (=).

Keywords - are used to give some special meaning to the interpreter and are used by Python interpreter to recognize the structure of program.

A partial list of keywords in Python 2.7 is

and	del	from	not
while	as	elif	global
or	with	assert	else
if	pass	Yield	break
except	import	print	class
exec	in	Raise	continue
finally	is	return	def
for	lambda	try	

Operators and Operands

Operators are special symbols that represent computation like addition and multiplication. The values that the operator is applied to are called operands. Operators when applied on operands form an expression. Operators are categorized as Arithmetic, Relational, Logical and Assignment. Following is the partial list of operators:

Mathematical/Arithmetic operators: +, -, *, /, %, ** and //.

Relational operators: <, <=, >, >=, != or <> and ==.

Logical operators: or, and, and not

Assignment Operator: =, +=, -=, *=, /=, %=, **= and //=

Input and Output

Program need to interact with end user to accomplish the desired task, this is done using Input-Output facility. Input means the data entered by user (end user) of the program. In python, raw_input() and input ( ) functions are available for input.

Syntax of raw_input() is:

Variable = raw_input ([prompt])

Example:

```
>>>x = raw_input ('Enter your name: ')
Enter your name: ABC
```

Example:

y = int(raw_input ("enter your roll no"))

will convert the accepted string into integer before assigning to 'y'.

Syntax for input() is:

Variable = input ([prompt])

Example:

x = input ('enter data:')

Enter data: 2+ ½.0

Will supply 2.5 to x

Print: This statement is used to display results.

Syntax:

print expression/constant/variable

Example:

```
>>> print "Hello"
Hello
```

Comments: As the program gets bigger and more complicated, it becomes difficult to read it and difficult to look at a piece of code and to make out what it is doing by just looking at it. So it is good to add notes to the code, while writing it. These notes are known as comments. In Python, comments start with '#' symbol. Anything written after # in a line is ignored by interpreter. For more than one line comments, we use the following;

- Place '#' in front of each line, or
- Use triple quoted string. (""" """)

Functions in Python: A function is named sequence of statement(s) that performs a computation. It contains line of code(s) that are executed sequentially from top to bottom by Python interpreter. They are the most important building block for any software in Python. For working in script mode, we need to write the Python code in functions and save it in the file having .py extension. Functions can be categorized as belonging to

- Modules
- Built in
- User Defined

1. **Module:**

   A module is a file containing Python definitions (i.e. functions) and statements. Standard library of Python is extended as module(s) to a Programmer. Definitions from the module can be used into code of Program. To use these modules in a program, programmer needs to import the module. Once we import a module, we can reference (use) to any of its functions or variables in our code. There are two ways to import a module in our program, they are

- import
- from

Import: It is simplest and most common way to use modules in our code.

Syntax:

import modulename1 [, module name 2, ---------]

Example: Input any number and to find square and square root.

Example:

```
import math
x = input ("Enter any number")
y = math.sqrt(x)
a = math.pow(x,2)
print "Square Root value=",y
print "square value=",a
```

output:

```
Enter any number25
Square Root value= 5.0
square value= 625.0
```

From statement: It is used to get a specific function in the code instead of complete file. If we know beforehand which function(s), we will be needing, then we may use 'from'. For modules having large number of functions, it is recommended to use from instead of import.

Syntax

```
>>> from modulename import functionname [, functionname…..]
from modulename import *
```

will import everything from the file.

Example: Input any number and to find square and square root.

Example:

```
from math import sqrt,pow
x=input("Enter any number")
y=sqrt(x)        #without using math
a=pow(x,2)       #without using math
print "Square Root value =",y
print "square value =",a
```

```
Enter any number100
Square Root value = 10.0
square value = 10000.0
```

The functions available in math module are:

ceil() floor() fabs() exp() log()   log10() pow()   sqrt() cos() sin() tan() degrees() radians()

Some functions from random module are:

random()   randint() uniform() randrange()

2. **Built in Function:**

   Built in functions are the function(s) that are built into Python and can be accessed by Programmer. These are always available and for using them, we don't have to import any module (file). Python has a small set of built-in functions as most of the functions have been partitioned to modules. This was done to keep core language precise.

   abs() max() min() bin() divmod() len() range() round() bool() chr() float() int() long() str( ) type( ) id( ) tuple( )

3. **User Defined Functions:**

   In Python, it is also possible for programmer to write their own function(s). These functions can then be combined to form module which can be used in other programs by importing them. To define a function, keyword 'def' is used. After the keyword comes an identifier i.e. name of the function, followed by parenthesized list of parameters and the colon which ends up the line, followed by the block of statement(s) that are the part of function.

Syntax:

```
def NAME ([PARAMETER1, PARAMETER2, …..]):
#Square brackets include optional part of statement
 statement(s)
```

Example: To find simple interest using function.

Example:

```
def SI(P,R,T):
    return(P*R*T)
```

Output:

```
>>> SI(1000,2,10)
20000
```

Parameters and Arguments

Parameters are the value(s) provided in the parenthesis when we write function header. These are the values required by function to work. If there is more than one value required by the function to work on, then, all of them will be listed in parameter list separated by comma.

Example: def SI (P,R,T):

Arguments are the value(s) provided in function call/invoke statement. List of arguments should be supplied in same way as parameters are listed. Bounding of parameters to arguments is done 1:1, and so there should be same number and type of arguments as mentioned in parameter list.

Example: Arguments in function call

```
>>> SI (1000,2,10)
```

1000,2,10 are arguments. An argument can be constant, variable, or expression.

Example: Write the output from the following function:

```
def SI(p,r=10,t=5):
  return(p*r*t/100)
```

if we use following call statement:

```
SI(10000)
SI(20000,5)
SI(50000,7,3)
```

Output

```
>>> SI(10000)
5000
>>> SI(20000,5)
5000
>>> SI(50000,7,3)
10500
```

Flow of Execution

Execution always begins at the first statement of the program. Statements are executed one after the other from top to bottom. Further, the way of execution of the program shall be categorized into three ways; (i) sequence statements, (ii) selection statements, and (iii) iteration or looping statements.

Sequence statements: In this program, all the instructions are executed one after another.

Example:

Program to find area of the circle.

```
r = input("enter any radius of the circle")
a = 3.14*r*r
print "Area=",a
```

output:

```
enter any radius of the circle7
Area = 153.86
```

In the above program, all three statements are executed one after another.

Selective Statements: In this program, some portion of the program is executed based upon the conditional test. If the conditional test is true, compiler will execute some part of the program, otherwise it will execute other part of the program. This is implemented in python using if statement.

Syntax:

```
if (condition):                if (condition):
    statements                     statements
else                    (or)   elif (condition):
    statements                         statements
                               else:
                                       Statements
```

Example:

1. Program to find the simple interest based upon number of years. If number of years is more than 12 rate of interest is 10 otherwise 15.

   Code:

   ```
   p = input("Enter any principle amount")
   t = input("Enter any time")
   if (t>10):
       si = p*t*10/100
   else:
       si = p*t*15/100
   print "Simple Interest = ",si
   ```

   output:

   ```
   Enter any principle amount 3000
   Enter any time12
   Simple Interest = 3600
   ```

2. Write a program to input any choice and to implement the following.

Choice	**Find**
1.	Area of square
2.	Area of rectangle
3.	Area of triangle

Code:

```
c = input ("Enter any Choice")
if(c==1):
    s = input("enter any side of the square")
    a = s*s
    print"Area = ",a
elif(c==2):
    l = input("enter length")
    b = input("enter breadth")
    a = l*b
    print"Area = ",a
elif(c==3):
      x = input("enter first side of triangle")
       y = input("enter second side of triangle")
       z = input("enter third side of triangle")
       s = (x+y+z)/2
       A = ((s-x)*(s-y)*(s-z))**0.5
       print"Area=",A
else:
    print "Wrong input"
```

Output:

```
Enter any Choice2
enter length4
enter breadth6
Area = 24
```

Iterative statements: In some programs, certain set of statements are executed again and again based upon conditional test. i.e executed more than one time. This type of execution is called looping or iteration. Looping statement in python is implemented by using 'for' and 'while' statement.

Syntax: (for loop)

```
for variable in range(start,stop+1,step):
          statements
```

Syntax: (while loop)

```
while (condition):
         Statements
```

Example:

1. Write a program to input any number and to print all natural numbers up to given number.

   Code:

   ```
   n = input("enter any number")
   for i in range(1,n+1):
         print i,
   ```

   Output:

   ```
   enter any number10
   1 2 3 4 5 6 7 8 9 10
   ```

2. Write a program to input any number and to find sum of all natural numbers up to given number.

   Code:

   ```
   n = input("Enter any number")
   sum= 0
   for i in range(1,n+1):
        sum = sum+i
   print "sum=",sum
   ```

   Output:

   ```
   Enter any number5
   sum = 15
   ```

3. Write a program to input any number and to find reverse of that number.

   Code:

   ```
   n = input("Enter any number")
   r = 0
   while(n>0):
         r = r*10+n%10
         n = n/10
   ```

```
print "reverse number is", r
Output:
>>>
Enter any number345
reverse number is 543
>>>
```

Example: Write the output from the following code:

1.
   ```
   sum = 0
   for i in range(1,11,2):
       sum+ = i
   print "sum = ", sum
   output:
   sum = 25
   ```

2.
   ```
   sum = 0
   i = 4
   while (i<=20):
       sum+=i
       i+= 4
   print "Sum = ",sum
   output:
   Sum = 60
   ```

Example: Interchange for loop into while loop

1.
   ```
   for i in range(10,26,2):
       print i
   Ans:
   i=10
   while(i<26):
       print i
       i+=2
   ```

2.
   ```
   s=0
   for i in range(10,50,10):
         s + =i
   ```

```
print " Sum= ", s
```

Ans:

```
s = 0
i = 10
while(i<50):
    s+ = i
    i+ = 10
print "Sum=",s
```

Example: Interchange while loop in to for loop.

```
i = 5
s = 0
while (i<25):
    s+ = i
    i + = 5
print " Sum =", s
```

Ans:

```
s = 0
for i in range(5,25,5):
    s+=i
print "Sum = ", s
```

Example: How many times following loop will execute.

1.
```
for i in range(10,50,5):
    print i
```

Ans:

i values are 10,15,20,25,30,35,40,45

8 times

2.
```
i=4
while(i<25):
    print i
    i+=4
```

Ans:

i values are 4,8,12,16,20,24

6 times

String:

In python, consecutive sequence of characters is known as a string. An individual character in a string is accessed using a subscript (index). The subscript should always be an integer (positive or negative) and starts from 0. A literal/constant value to a string can be assigned using a single quotes, double quotes or triple quotes. Strings are immutable i.e. the contents of the string cannot be changed after it is created.

Strings Operations:

+ (Concatenation)

* (Repetition)

in (Membership)

not in

range (start, stop[,step])

slice[n:m]

Example: Write the output from the following code:

1. A = 'Global'
   B = 'warming'
   print A+B
   Ans: Globalwarming
2. A = 'Global'
   Print 3*A
   Ans: 'GlobalGlobalGlobal'
3. A='Global'
   'o' in A
   Ans: True
4. A='Global'
   'g' in A
   Ans: False
5. A='Global'
   'o' not in A
   Ans: False
6. A='Global'
   'g' not in A
   Ans: True

String methods & built in functions:

len() capitalize() find(sub[,start[, end]]) isalnum() isalpha()

isdigit() lower() islower() isupper() upper() lstrip()
rstrip() isspace() istitle() replace(old,new) join ()
swapcase() partition(sep) split([sep[,maxsplit]])

Example:

```
>>> s='Congratulations'
>>> len(s)
15
>>> s.capitalize()
'Congratulations'
>>> s.find('al')
-1
>>> s.find('la')
8
>>> s[0].isalnum()
True
>>> s[0].isalpha()
True
>>> s[0].isdigit()
False
>>> s.lower()
'congratulations'
>>> s.upper()
'CONGRATULATIONS'
>>> s[0].isupper()
True
>>> s[1].isupper()
False
>>> s.replace('a','@')
'Congr@tul@tions'
>>> s.isspace()
False
>>> s.swapcase()
'cONGRATULATIONS'
>>> s.partition('a')
('Congr', 'a', 'tulations')
>>> s.split('ra',4)
```

```
['Cong', 'tulations']
>>> s.split('a')
['Congr', 'tul', 'tions']
>>> a=' abc '
>>> a.lstrip()
'abc '
>>> a.rstrip()
' abc'
```

Examples:

Example: Write a program to input any string and count number of uppercase and lowercase letters.

Code:

```
s=raw_input("Enter any String")
rint s
u=0
l=0
i=0
while i<len(s):
    if (s[i].islower()==True):
        l+=1
    if (s[i].isupper()==True):
        u+=1
        i+=1
    print "Total upper case letters :", u
    print "Total Lower case letters :", l
```

Output:

```
Enter any String Python PROG
Python PROG
Total upper case letters: 5
Total Lower case letters: 5
```

Example:

Write the output from the following code:

```
s = 'Indian FESTIVALS'
i = 0
while i<len(s):
```

```
if (s[i].islower()):
        print s[i].upper(),
if (s[i].isupper()):
        print s[i].lower(),
i +=1
```

Ans:

```
i N D I A N f e s t i v a l s
```

List:

Like a string, list is a sequence of values. In a string, the values are characters, whereas in a list, they can be of any type. The values in the list are called elements or items or members. It is an ordered set of values enclosed in square brackets [ ]. Values in the list can be modified, i.e. it is mutable. As it is set of values, we can use index in square brackets [ ] to identify a value belonging to it.

List Slices: Slice operator works on list. This is used to display more than one selected values on the output screen. Slices are treated as boundaries and the result will contain all the elements between boundaries.

Syntax:

Seq = L [start: stop: step]

Where start, stop & step - all three are optional. If you omit first index, slice starts from '0' and omitting of stop will take it to end. Default value of step is 1.

Example:

```
>>> L=[10,20,30,40,50]
>>> L1=L[2:4]
>>> print L1
[30, 40]
```

List Methods:

append() extend () pop() del() remove()

insert() sort() reverse() len()

Example:

```
>>> L=[500,1000,1500,2000]
>>> L.append(2500)
>>> print L
[500, 1000, 1500, 2000, 2500]
>>> L1=[3000,3500]
>>> L.extend(L1)
>>> print L
[500, 1000, 1500, 2000, 2500, 3000, 3500]
>>> L.pop()
3500
>>> L.pop(3)
2000
>>> print L
[500, 1000, 1500, 2500, 3000]
>>> del L[2]
>>> print L
[500, 1000, 2500, 3000]
>>> L.remove(1000)
>>> print L
[500, 2500, 3000]
>>> L.insert(3,3500)
>>> print L
[500, 2500, 3000, 3500]
>>> L.reverse()
>>> print L
[3500, 3000, 2500, 500]
>>> L.sort()
>>> print L
[500, 2500, 3000, 3500]
>>> print len(L)
4
```

Note: Operator + & * can also be applied on the lists. + is used to concatenate the two lists and * is used to repeat the list given number of times.

Example:

```
>>> l=[10,20,30]
>>> m=[40,50]
>>> l=l+m
>>> print l
[10, 20, 30, 40, 50]
>>> b=m*3
```

```
>>> print b
[40, 50, 40, 50, 40, 50]
```

Dictionaries:

A dictionary is like a list, but more in general. In a list, index value is an integer, while in a dictionary index value can be any other data type and are called keys. The key will be used as a string as it is easy to recall. A dictionary is an extremely useful data storage construct for storing and retrieving all key value pairs, where each element is accessed (or indexed) by a unique key. However, dictionary keys are not in sequences and hence maintain no left-to-right order.

Key-value pair: We can refer to a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps a value. The association of a key and a value is called a key-value pair.

Syntax:

```
my_dict = {'key1': 'value1','key2': 'value2','key3': 'value3'…'keyn': 'valuen'}
```

Note: Dictionary is created by using curly brackets(ie. {}).

Dictionary methods:

cmp( )   len( )   clear( )   get()   has_key( )

items( )   keys()   values()   update()   dict()

Example:

```
>>> month=dict()
>>> print month
{}
>>> month["one"]="January"
>>> month["two"]="Feb"
>>> print month
{'two': 'Feb', 'one': 'January'}
>>> len(month)
2
>>> month.get("one")
'January'
>>> month.get("one","feb")
'January'
>>> month.keys()
['two', 'one']
>>> month.has_key("one")
True
```

```
>>> month.has_key("three")
False
>>> month.items()
[('two', 'Feb'), ('one', 'January')]
>>> month.values()
['Feb', 'January']
>>> m=month
>>> cmp(month,m)
0
>>> n=dict()
>>> cmp(m,n)
1
>>> cmp(n,m)
-1
>>> m.clear()
>>> print m
{}
```

Tuples:

A tuple is a sequence of values, which can be of any type and they are indexed by integer. Tuples are just like list, but we can't change values of tuples in place. Thus tuples are immutable. The index value of tuple starts from 0.

A tuple consists of a number of values separated by commas. For example:

```
>>> T=10, 20, 30, 40
>>> print T
(10, 20, 30, 40)
```

But in the result, same tuple is printed using parentheses. To create a tuple with single element, we have to use final comma. A value within the parenthesis is not tuple.

Tuple Slices: Slice operator works on Tuple also. This is used to display more than one selected value on the output screen. Slices are treated as boundaries and the result will contain all the elements between boundaries.

Syntax:

Seq = T [start: stop: step]

Where start, stop & step - all three are optional. If we omit first index, slice starts from '0'. On omitting stop, slice will take it to end. Default value of step is 1.

Example:

```
>>> T=(10,20,30,40,50)
>>> T1=T[2:4]
>>> print T1
(30, 40)
```

In the above example, starting position is 2 and ending position is 3(4-1), so the selected elements are 30 & 40.

Tuple functions:

cmp( ) len( ) max( ) min( ) tuple( )

Example:

```
>>> T=tuple()
>>> print T
()
>>> T=["mon","tue","wed","thu","fri","sat","sun"]
>>> print T
['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun']
>>> len(T)
7
>>> min(T)
'fri'
>>> max(T)
'wed'
>>> T1=T
>>> T2=(10,20,30)
>>> cmp(T,T1)
0
>>> cmp(T2,T1)
1
>>> cmp(T1,T2)
-1
```

LET'S REVISE

Interactive Mode: Interactive Mode, as the name suggests, allows us to interact with OS.

Script Mode: In script mode, we type Python program in a file and then use interpreter to execute the content of the file.

Number: Number data type stores Numerical Values.

Sequence: A sequence is an ordered collection of items, indexed by positive integers.

Arithmetic operators: +, -, *, /, %, ** and //.

Relational operators: <, <=, >, >=, != or <> and ==.

Logical operators: or, and, and not

Assignment Operator: =, +=, -=, *=, /=, %=, **= and //=

Functions in Python: A function is named sequence of statement(s) that performs a computation.

Module: A module is a file containing Python definitions (i.e. functions) and statements. Standard library of Python is extended as module(s) to a Programmer.

String: In python, consecutive sequence of characters is known as a string. An individual character in a string is accessed using a subscript (index).

List: Like a string, list is a sequence of values. List can be of any type.

Dictionaries: A dictionary is like a list, but more in general. In a list, index value is an integer, while in a dictionary index value can be any other data type and are called keys.

Tuples: A tuple is a sequence of values, which can be of any type and they are indexed by integer.

EXERCISE

1. Write the output from the following code:

a)
```
x = 10
y = 20
if (x>y):
    print x+y
else:
    print x-y
```

b)
```
print "Inspirational stories \n for \t Children"
```

c)
```
s = 0
for I in range(10,2,-2):
    s+=I
print "sum= ",s
```

d)
```
n = 50
i = 5
s = 0
while i<n:
    s+ = i
    i+ = 10
print "i=",i
print "sum=",s
```

e)
```
y = 2000
if (i%4==0):
    print "Leep Year"
else:
    print "Not leep year"
```

2. Write for statement to print the following series:

a) 10,20,30…….300

b) 105,98,91,….7

3. Write the while loop to print the following series:

a) 5,10,15,…100

b) 100,98,96,…2

4. How many times are the following loop executed?

a)
```
for a in range(100,10,-10):
    print a
```

b)
```
i = 100
while(i<=200):
    print i
    i + =20
```

c)
```
for b in (1,10):
    print b
```

d)
```
i = 4
while (i>=4):
    print i
    i+ = 10
```

f)
```
i=2
while (i<=25)
    print i
```

5. Rewrite the following for loop into while loop:

a)
```
for a in range(25,500,25):
    print a
```

b)
```
for a in range(90,9,-9):
    print a
```

6. Rewrite the following while loop into for loop:

a)
```
i = 10
while i<250:
    print i
    i = i+50
```

b)
```
i=88
while(i>=8):
    print i
    i- = 8
```

7. Which command is used to convert text into integer value?

8. Find the errors from the following code.

a.
```
T=[a,b,c]
Print T
```

b.
```
for i in 1 to 100 :
    print I
```

c.
```
 i=10 ;
while [i<=n] :
     print i
   i+=10
```

d.
```
 if (a>b)
     print a:
else if (a<b)
 print b:
 else
 print "both are equal"
```

9. Find the output from the following code:

```
L=[100,200,300,400,500]
L1=L[2:4]
print L1
L2=L[1:5]
print L2
L2. .extend(L1)
print L2
```

10. Write program to input any number and to print all factors of that number.

11. Write a program to input any number and to check whether given number is Armstrong or not. (Armstrong 1,153,etc. 13 =1 , 13+53 +33 =153)

12. Write a program to input employee no, name basic pay and to find HRA, DA and netpay.

Basic pay	Hra	Da
>100000	15%	8%
<=100000&>50000	10%	5%
<=50000	5%	3%

13. Write a program to find all prime numbers up to given number.
14. Write a program to convert decimal number to binary.
15. Write a program to convert binary to decimal.
16. Write a program to input two complex numbers and to find sum of the given complex numbers.
17. Write a program to input two complex numbers and to implement multiplication of the given complex numbers.
18. Write a program to find sum of two distances with feet and inches.
19. Write a program to find difference between two times with hours, minutes and seconds.
20. Write a program to find the sum of all digits of the given number.
21. Write a program to find the reverse of that number.
22. Write a program to input username and password and to check whether the given username and password are correct or not.
23. Which string method is used to implement the following:
    a) To count the number of characters in the string.
    b) To change the first character of the string in capital letter.
    c) To check whether given character is letter or a number.
    d) To change lower case to upper case letter.
    e) Change one character into another character.
24. Write a program to input any string and to find number of words in the string.
25. Write a program to input any two strings and to check whether given strings are equal are not.
26. Differentiate between tuple and list.
27. Write a program to input n numbers and to insert any number in a particular position.
28. Write a program to input n numbers and to search any number from the list.
29. Write a program to input n customer name and phone numbers.
30. Write a program to search input any customer name and display customer phone number if the customer name is exist in the list.
31. Explain in detail about cmp() function.
32. Write a program to input n numbers and to reverse the set of numbers without using functions.

Chapter-2: Concept of Object Oriented Programming

Learning Objectives:

At the end of this chapter the students will be able to:

- Understand about Object Oriented Programming(OOP) classes and objects
- Know the concepts related to OOP
  - Objects
  - Classes
  - Encapsulation
  - Data Hiding
  - Abstraction
  - Polymorphism
  - Inheritance
- Know about the advantages of OOP over earlier programming methodologies

An object-oriented programming (OOP) is a programming language model which is organized around "objects" rather than "actions" and data rather than logic. Before the introduction of the Object Oriented Programming paradigm, a program was viewed as a logical procedure that takes input data, processes it, and produces output. But in case of OOP a problem is viewed in terms of objects rather than procedure for doing it. Now the question arises what is an object?

An object can be anything that we notice around us. It can be a person (described by name, address, date of Birth etc, his typing speed), a cup (described by size , color , price etc.) , a car (described by model , color , engine etc., its mileage, speed) and so on. In fact it can be an identifiable entity. The whole idea behind an object oriented model is to make programming closer to they real world thereby making it a very natural way of programming. The core of pure object-oriented programming is to combine into a single unit both data and functions or methods that operate on that data.

Simula was the first object-oriented programming language. Java, Python, C++, Visual Basic, .NET and Ruby are the most popular OOP languages today.

Basic Concepts of Object Oriented Programming

The basic concepts related to OOP are as follows:

1. Objects
2. Classes

3. Encapsulation
4. Abstraction
5. Data Hiding
6. Polymorphism
7. Inheritance

Object

An object is the basic key concept of Object Oriented Programming. As mentioned before it can be anything around us - a person, place, any activity or any other identifiable entity. Every object is characterised by:

- ***Identity:*** This is the name that identifies an object. For example a Student is the name given to anybody who is pursuing a course. Or an i-phone is a mobile phone that has been launched by Apple Inc.
- ***Properties:*** These are the features or attributes of the object. For example a student will have his name age, class, date of birth etc. as his attributes or properties. A mobile phone has model, color, price as its properties.
- ***Behaviour:*** The behaviour of an object signifies what all functions an object can perform. For example a student can pass or fail the examination. A mobile phone can click and store photographs (behave like a camera).

  So an object clearly defines an entity in terms of its properties and behaviour. Consider an example of an object - Windows mobile phone. This phone has certain properties and certain functions which are different from any other mobile phone- say an Android phone. Both are mobile phones and so possess common features that every mobile phone should have but yet they have their own properties and behaviours. The data of a particular object can be accessed by functions associated with that object only. The functions of one object cannot access the data of another object.

Classes

A class is group of objects with same attributes and common behaviours. It is basically a blueprint to create objects. An object is a basic key concept of OOP but classes provide an ability to generalize similar type of objects. Both data and functions operating on the data are bundled as a unit in a class for the same category of objects. Here to explain the term 'same category of object', let us take the example of mobile phone. A Windows phone, Android phone and i-phone, all fall into the category of mobile phones. All of these are instances of a class, say Mobile_phone and are called objects.

Similarly we can have another example where students named Rani and Ravish are objects. They have properties like name, date of birth, address, class, marks etc. and the behaviour can be giving examinations. Anybody pursuing any course, giving any type of examination will come into the category of students. So a student is said to be a class as they share common properties and behaviours. Although a student can be a school student, a college student or a university student or a student pursuing a music course and so on, yet all of these have some properties and behaviours in common which will form a class. An analogy is that you can have variables of type int which translates to saying that variables that store integers are variables which are instances (objects) of the int class.

A real instance of a class is called an object and creating the new object is called instantiation. Objects can also have functionality by using functions that belong to a class. Such functions are called methods of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object. Collectively, the fields and methods can be referred to as the attributes of that class. Let us take the example of the class Mobile_phone which is represented in the block diagram below:

Class: Mobile_phone
Data: Model Color Price

Functions:
Store_number()
Missed_calldetail()

Fig 1: Class Mobile_phone

A class is defined before the creation of objects of its type. The objects are then created as instances of this class as shown in the figure below.

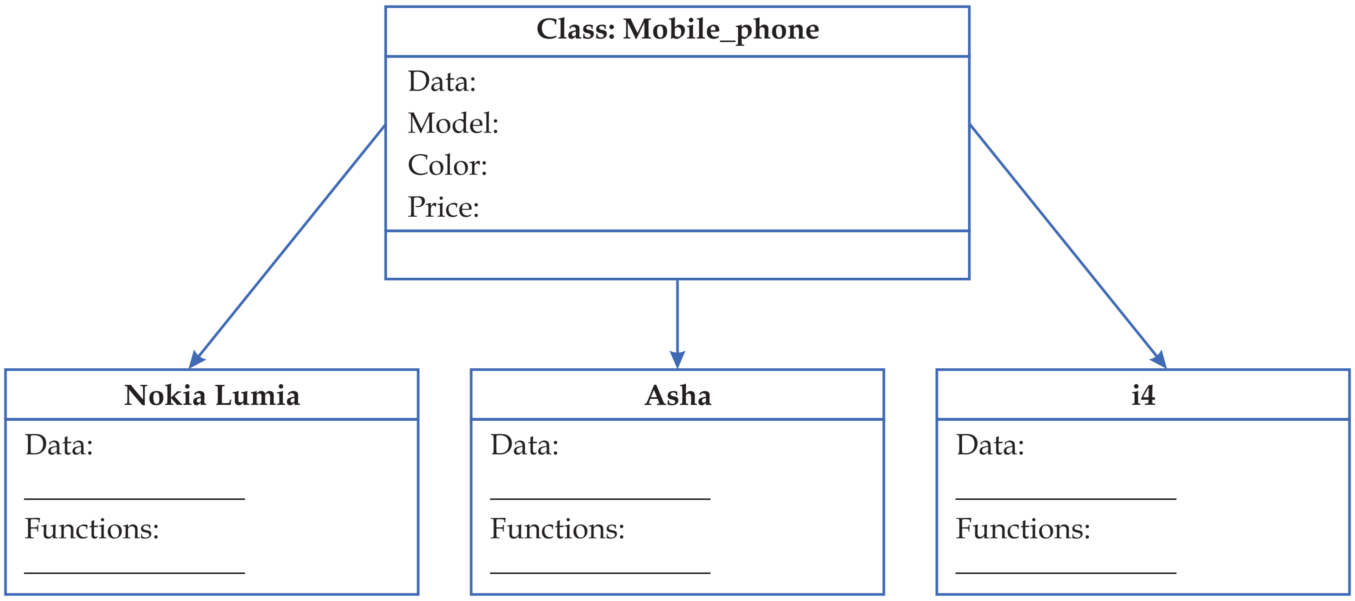


Fig 2: Class and Objects

In the above example, Nokia Lumia, Asha and i4 are all instances of the class Mobile_phone. All these instances are similar in the sense that all have basic features that a mobile phone should have. So all of these are objects of the class Mobile_phone

The general form of class definition in Python and creation of objects will be discussed in the next chapter.

Encapsulation

Encapsulation is the most basic concept of OOP. It is the combining of data and the functions associated with that data in a single unit. In most of the languages including python, this unit is called a class. In Fig -1 showing class Mobile_phone, given under the subtopic Classes, we see that the name of the class, its properties or attributes and behaviours are all enclosed under one independent unit. This is encapsulation, implemented through the unit named class.

In simple terms we can say that encapsulation is implemented through classes. In fact the data members of a class can be accessed through its member functions only. It keeps the data safe from any external interference and misuse. The only way to access the data is through the functions of the class. In the example of the class Mobile_phone, the class encapsulates the data (model, color, price) and the associated functions into a single independent unit.

Data Hiding

Data hiding can be defined as the mechanism of hiding the data of a class from the outside world or to be precise, from other classes. This is done to protect the data from any accidental or intentional access.

In most of the object oriented programming languages, encapsulation is implemented through classes. In a class, data may be made private or public. Private data or function of a class cannot be accessed from outside the class while public data or functions can be accessed from anywhere. So data hiding is achieved by making the members of the class private. Access to private members is restricted and is only available to the member functions of the same class. However the public part of the object is accessible outside the class. (You will study about private and public members in detail in the next chapter.)

Data Abstraction

Do you know the inner details of the monitor of your PC or your mobile phone? What happens when you switch ON the monitor or when any call is received by you on your phone? Does it really matter to you what is happening inside these devices? No, it does not. Right? Important thing for you is whether these devices are working as per your requirement or not? You are never concerned about their inner circuitry. This is what we call abstraction.

The process of identifying and separating the essential features without including the internal details is abstraction. Only the essential information is provided to the outside world while the background details are hidden. Classes use the concept of abstraction. A class encapsulates the relevant data and functions that operate on data by hiding the complex implementation details from the user. The user needs to focus on what a class does rather than how it does.

Let us have a look at the Mobile_phone class. The case or body of the mobile phone is abstraction. This case is the public interface through which the user interacts. Inside the case there are numerous components such as memory, processor, RAM etc. which are private and so are hidden behind the public interface called case/body. Thus this case/body is the abstraction which has separated the essential components from implementation details. So when you purchase a mobile, you are given information about only the functions and operations of the mobile. The inside mechanism of the working of the mobile is of no concern to you. As long as the mobile is functioning properly, you are not bothered about the inner circuitry of the mobile phone.

Abstraction and Encapsulation are complementary concepts. Through encapsulation only we are able to enclose the components of the object into a single unit and separate the private and public members. It is through abstraction that only the essential behaviours of the objects are made visible to the outside world. So we can say that encapsulation is the way to implement data abstraction. In another example of class Student, only the essential information like roll no, name, date of birth, course etc. of the student are visible. The secret information like calculation of grades, allotment of examiners etc. is hidden.

Inheritance

Inheritance is one of the most useful characteristic of object-oriented programming as it enforces reusability of code. Inheritance is the process of forming a new class (derived class) from an existing class (called the base class). The data members and the methods associated with the data are accessible in the inherited class.

Let us understand this characteristic with the help of the class Mobile_phone.

An i-phone is a class in itself. It is a type of mobile phone. So we can have Mobile_phone as the base class and i_phone as its derived class as shown in the figure below:

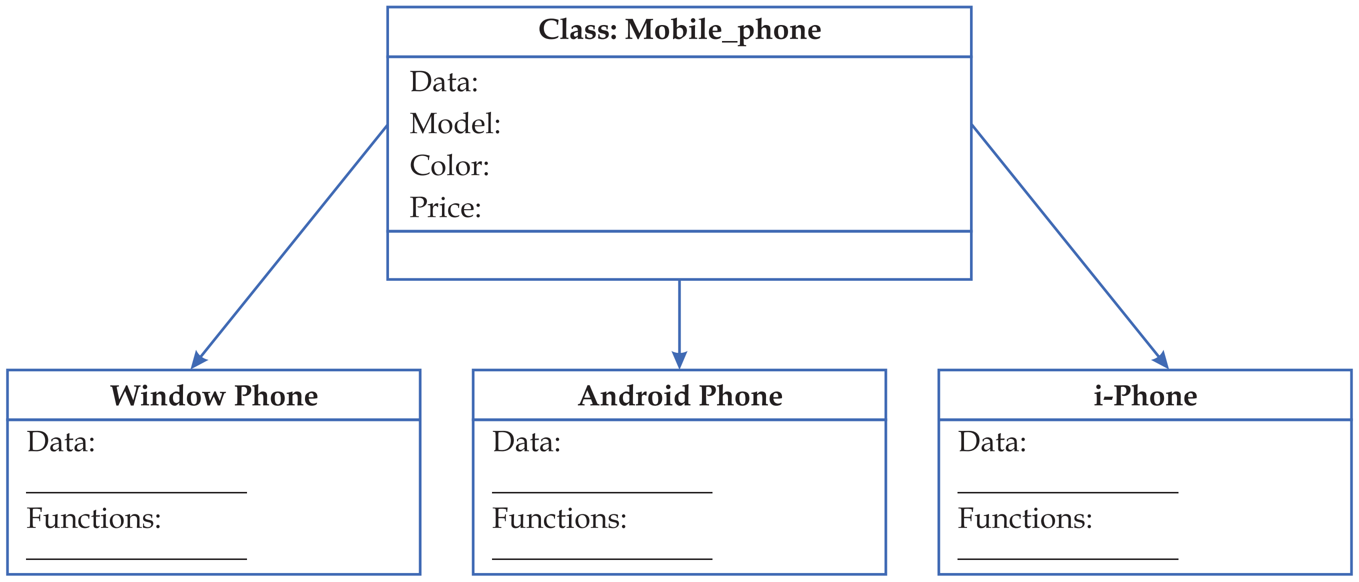


Fig 3: Inheritance

Such hierarchical classification helps to obtain a new class from an existing class. The derived class can also contain some new attributes of itself. So the derived class contains features of the base class as well as of itself. For example an i-phone will have all the features of a Mobile_phone class in addition to its own characteristics. Such a relationship between the two classes is known as "a kind of "relationship. For example an i-phone is a kind of mobile phone.

So we see that the base class can be reused again and again to define new classes. Another advantage of inheritance is its transitive nature. If a class i_phone inherits properties of another class Mobile_phone, then all derived classes of i_phone will inherit properties of the class Mobile_phone. All these factors make inheritance a very important characteristic of object oriented programming.

Polymorphism

The word Polymorphism is formed from two words - poly and morph where poly means many and morph means forms. So polymorphism is the ability to use an operator or function in various forms. That is a single function or an operator behaves differently depending upon the data provided to them. Polymorphism can be achieved in two ways:

1. **Operator Overloading**

   In class XI you have worked with '+ 'operator. You must have noticed that the '+' operator behaves differently with different data types. With integers it adds the two numbers and with strings it concatenates or joins two strings. For example:

   Print 8+9 will give 17 and

   Print "Python" + "programming" will give the output as Pythonprogramming.

   This feature where an operator can be used in different forms is known as Operator Overloading and is one of the methods to implement polymorphism.

2. **Function Overloading**

   Polymorphism in case of functions is a bit different. A named function can also vary depending on the parameters it is given. For example, we define multiple functions with same name but different argument list as shown below:

```
def test():                  #function 1
        print "hello"
def test(a, b):              #function 2
        return a+b
def test(a, b, c):           #function 3
    return a+b+c
```

   In the example above, three functions by the same name have been defined but with different number of arguments. Now if we give a function call with no argument, say test(), function 1 will be called. The statement test(10,20) will lead to the execution of function 2 and if the statement test(10,20,30) is given Function 3 will be called. In either case, all the functions would be known in the program by the same name. This is another way to implement polymorphism and is known as Function Overloading.

As we see in the examples above, the function called will depend on the argument list - data types and number of arguments. These two i.e. data types and the number of arguments together form the function signature. Please note that the return types of the function are no where responsible for function overloading and that is why they are not part of function signature.

Here it must be taken into consideration that Python does not support function overloading as shown above although languages like Java and C/C++ do. If you run the code of three test functions, the second test() definition will overwrite the first one. Subsequently the third test() definition will overwrite the second one. That means if you give the function call test(20,20) , it will flash an error stating, "Type Error: add() takes exactly 3 arguments (2 given)". This is because, Python understands the latest definition of the function test() which takes three arguments.

Static and Dynamic Binding

Binding is the process of linking the function call to the function definition. The body of the function is executed when the function call is made. Binding can be of two types:

Static Binding: In this type of binding, the linking of function call to the function definition is done during compilation of the program.

Dynamic Binding: In this type of binding, linking of a function call to the function definition is done at run time. That means the code of the function that is to be linked with function call is unknown until it is executed. Dynamic binding of functions makes the programs more flexible. You will learn more on dynamic binding in the next chapter.

Advantages of OOP

Object Oriented programming has following advantages:

- ***Simplicity:*** The objects in case of OOP are close to the real world objects, so the complexity of the program is reduced making the program structure very clear and simple. For example by looking at the class Mobile_phone, you can simply identify with the properties and behaviour of an actual mobile phone. This makes the class Mobile_phone very simple and easy to understand.
- ***Modifiability:*** It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.
- ***Extensibility and Maintainability:*** It is quite easy to add new features and extend the program in case of object oriented programming. It can be simply done by introducing a few new objects and modifying some existing ones. The original base class need not be modified at all. Even objects can be

maintained separately. There by making locating and fixing problems easier. For example if a new version of i-phone is introduced, a new derived class of the class i_phone for the new version may be created and no other class in the class hierarchy need to be modified. Similarly if any behaviour of a Windows phone changes, maintenance has to be done only for the class Windows phone.

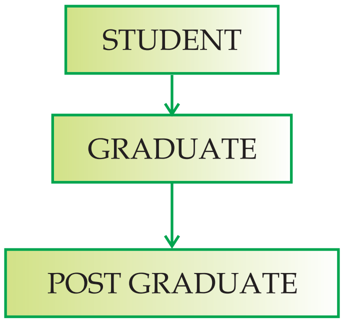
- ***Re-usability:*** Objects can be reused in different programs. The class definitions can be reused in various applications. Inheritance makes it possible to define subclasses of data objects that share some or all of the main class characteristics. It forces a more thorough data analysis, reduces development time, and ensures more accurate coding.
- ***Security:*** Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of data hiding provides greater system security and avoids unintended data corruption.

LET'S REVISE

- **Object:** clearly defines an entity in terms of its properties and behaviour.
- **Class:** a blueprint of an object.
- **Encapsulation:** combining of data and the functions associated with that data in a single unit
- **Data Hiding:** the mechanism of hiding the data of a class from the outside world
- **Abstraction:** providing only essential information to the outside world and hiding their background details
- **Inheritance:** forming a new class (derived class) from an existing class (called the base class).
- **Polymorphism:** ability to use an operator or function in various forms.
- **Static Binding:** the linking of function call to the function definition is done during compilation of the program.
- **Dynamic Binding:** the linking of function call to the function definition is done during the execution of the program.

EXCERCISE

- Fill in the blanks:
  a. Act of representing essential features without background detail is called ___________.
  b. Wrapping up of data and associated functions in to a single unit is called_________.
  c. ____________is called the instance of a class.
- What is Object Oriented Programming? List some of its advantages.
- Differentiate between an object and a class.
- What is inheritance? Explain with an example.
- List its three features that make it an important characteristic of OOP.
- Consider the figure given below and answer the questions that follow:



  a. Name the base class and the derived class.
  b. Which concept of OOP is implemented in the figure given above?
- How do abstraction and encapsulation complement each other?
- Explain polymorphism with an example.
- Explain Data Hiding with respect to OOP.
- Explain Function overloading with an example.
- Is function overloading supported by Python? Give reasons.
- Write the overloaded function definitions of add()- on adds two numbers and other concatenates two strings.
- Predict the output of the following program. Also state which concept of OOP is being implemented?

  def sum(x,y,z):

```
        print "sum= ", x+y+z
def sum(a,b):
        print "sum= ", a+b
sum(10,20)
sum(10,20,30)
```

- State whether the following are function overloading or not. Explain why or why not.

  a. def (a,b):

     def(x,y) :

  b. def(x,y,z)

     def(e,f)

- Define binding.
- Differentiate between static and dynamic binding.
- Write a program to find area of following using function overloading.
- Area of circle (function with one parameter)
- Area of rectangle (function with two parameters)
- Area of triangle (function with three parameters)
- Write a program to find out volume of the following using function overloading.
- volume of cube
- volume of cuboid
- volume of cylinder

Chapter-3: Classes in Python

Learning Objectives:

At the end of this chapter the students will be able to:

- Understand name spaces and scope rules
- Define classes (attributes , methods)
- Use __init__()
- Undersatnd the importance of "self"
- Create instance objects
- Distinguish between class attributes and instance attributes
- Add methods dynamically
- Access attributes and methods
- Use built-in class attributes (dict , doc , name , module , bases)
- Use __del__() and __str__() in a class
- Understand data hiding
- Understand static methods
- Destroy objects (garbage collection)

In the previous chapter you have studied that classes and objects are one of the most important characteristic of Object Oriented Programming. It is through classes that all the characteristics of OOP are implemented - may it be encapsulation, abstraction or inheritance.

This chapter deals with classes in Python. As Python is fully object-oriented, you can define your own classes, inherit from your own or built-in classes, and instantiate the classes that you've defined. But before we start with our study on classes let us understand about namespaces and scope rules in Python.

Namespaces

In Class XI, you had studied that variables refer to an object and they are created when they are first assigned a value. In fact the variables are bound to their values using the assignment operator(=). So a namespace is a place where a variable's name is stored and the value of the variable is bound to this namespace.

A namespace is a mapping from names to objects. It is a thing which associates the names with its values. In simple terms, it is a place where name lives. They are created at different moments and have different lifetimes. The examples of namespaces are:

- **Built-in names**

  These consist of functions such as max() , min() and built-in exception names. This namespace is created when the Python interpreter starts up, and is never deleted. The built-in names actually also live in a module called__ builtin__.

- **Global names in a module**

  The global namespace for a module is created when the module definition is read in and normally lasts until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered to be part of a module called__main__ and they have their own global namespace.

- **Local names in a function invocation**

  The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. Even each recursive invocation has its own local namespace.

  If we talk about classes and objects, the set of attributes of an object also form a namespace. It must be noted that there is absolutely no relation between names in different namespaces. Two different modules may both define same function without any confusion because the functions are prefixed with the module name. That means module1.cmp() has no relation with module2.cmp().

Scope Rules

A scope is a region of a Python program where a namespace is directly accessible.The location where the names are assigned in the code determines the scope of visibility of the name in that code. Although scopes are determined statically i.e. during creation of the program, yet they are used dynamically i.e. during execution of the program. At any time during execution, there are at least four main things to remember in the context of scope rules:

i) In Python, names of all types of variables are treated in same manner. That means numbers, strings, functions, types, modules, classes - all are treated in the same way. Also a name can refer to only one thing at a time. For example, consider the following program:

```
var = 10 + 5
print var
def var(y):
    return y*10
    print var
var = "Hello"
print var
```

In the code given above, the variable var is bound to 15(10 + 5). Then def var(y) binds var to a function. The previous binding of var to 15 is lost and is replaced by the function. Thereafter var is bound to a string, so its binding with the function is no longer existing.

ii) The scope of a variable is its enclosing function or class or file. As discussed before, each name belongs to a namespace. For example, if a variable is created in a particular function, then its scope is that function only, since that function creates its own namespace where it resides. So any variable inside the function will be local to that namespace. In the following example, the scope of the variable x is the test function.

```
def test():
    x = 5
    print x
```

Now let us modify the program -

```
x = 10
def exam():
    print x
def test():
    x = 5
    print x
def marks(x):
    print x
print x
exam()
test()
marks(20)
```

On executing the above code, the output will be

```
10
10
5
20
```

The first line creates a variable x that belongs to the namespace of the file, so its scope is the entire file. Hence 10 is displayed. The exam function creates its namespace, but that namespace doesn't have an x in it. As Python doesn't find x there, it checks the next larger enclosing namespace and finds x. So exam uses the variable x defined at the top and displays 10.

However, the test function defines its own variable named x with value 5, which has higher priority over the first definition of x. So any mention of x within the test function will refer to that x, hence displaying 5. The marks function also has an x in its own namespace, just like test function has. So x gets bound to whatever value is passed as an argument to marks function (20 in the given example). Hence the outer x is shadowed again in this function displaying the output as 20.

iii) The names always belong to the namespace where they are bound, irrespective of whether they are bound before or after they are referred. This is the reason which makes Python a lexically scoped language. The variable scopes are determined entirely by the locations of the variables in the source code of your program files, not by function calls. If a binding for a variable appears anywhere inside a function, the variable name is local to that function. Let us understand this with the help of an example:

```
x = 10
def func1():
    x=50
    print x
def func2():
    print x
    x=25
def func3(p):
    if p<10:
        x=2
    print x
func1()
func2()
func3(20)
func3(5)
```

In the above example, the func1 function creates a local variable x in its own namespace, shadowing the outer variable x. So the line print x prints 50. The func2 function also has a local variable x in its namespace but the assignment to x is after the print statement. The local variable x shadows the outer x, even though the local x is initially not bound to anything. The line print x looks for x in the local namespace, finds that it is not bound to anything, and so the reference to x leads to an error (an Unbound Local Error occurs). Similarly, in func3(), the variable x is local.

When we call func3(20), the line x = 2 is not executed, so print x causes an error. But when we call func3(5), the line x = 2 is executed , so print x prints 2.

iv) Names declared with global keyword have to be referred at the file level. This is because the global statement indicates that the particular variable lives in the global scope. If no global statement is being used, the assignment to the name is always in the innermost local scope. Consider the following example:

```
x=5
def func1():
    x=2
    x=x+1
def func2():
    global x
    x=x+1
print x
func1()
print x
func2()
print x
```

The above example prints 5; then calling func1()it prints 3. This is because func1 only increments a local x. Then func2()increments the global x and prints 6.

LEGB Rule

From the examples discussed above, we come up to the LEGB rule. According to this rule, when a name is encountered during the execution of the program, it searches for that name in the following order:

L. Local - It first makes a local search i.e. in current def statement. The import statements and function definitions bind the module or function name in the local scope. In fact, all operations that introduce new names use the local scope.

E. Enclosing functions - It searches in all enclosing functions, form inner to outer.

G. Global (module) - It searches for global modules or for names declared global in a def within the file.

B. Built-in (Python) - Finally it checks for any built in functions in Python.

The examples given above give the output according to the LEGB rule only.

Defining Classes

We have already studied in the previous chapter that a class is a unit that encapsulates data and its associated functions. In this section we will learn how to define classes.

To define a class in Python, we need to just define the class and start coding. A Python class starts with the reserved word 'class', followed by the class name and a colon(:). The simplest form of class definition looks like this:

```
class Class Name:
    <statement-1>
    ….
    …
    <statement-2>
```

Everything in a class is indented after the colon, just like the code within a function, if statement or for loop. The first thing not indented is not part of the class definition. A class may contain attributes which can be data members or/and member functions i.e. methods. In fact the word attribute is used for any name following a dot. Let us take the example of class Test:

```
class Test:
    var = 50
    marks = 10
    def display():
        print var
```

In the above class definition, marks, var, display(), all are attributes of the class Test and also of all its objects. Similarly when we import modules, in the expression module1.fuctionname, module1 is a module object and function name is a method but also referred to as an attribute of module 1. The module's attributes and the global names are defined in the same namespace. You will learn about object creation and usage later in this chapter.

Class definitions, like function definitions using def statements must be given before they are referenced for use. When a class definition is entered a new namespace is created and then used as local scope. Hence all assignments to the local variables are attached with this namespace. The function definitions are also bound to the same namespace.

Attributes of a class may be read-only or writable. In the latter case, assignment to attributes is possible. That means the following statement is valid:

```
test1.marks=10
```

Writable attributes may also be deleted using the del statement. For example:

```
del test1.marks
```

The above statement will remove the attribute marks from the object named test1.In fact the del statement removes the binding of the attribute (marks) from the namespace (test1) referenced by the class's local scope.

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition. We will learn more about class objects in the next section.

Note that calling a method on an object can also change the object. This implies that an object is mutable. A function can modify an outer mutable object by calling a method on it. Consider the example below:

```
x= [10]
def List_ex():
    x.append(20)
def add_list():
    x=[30,40]
    x.append(50)
print x
list_ex()
print x
add_list()
print x
```

The above example prints

```
[10]
[10,20]
[30,40,50]
```

The list_ex()calls the append method of the global x, whereas the add_list(), x refers to a local x.

Also data attributes override method attributes with the same name. That means if the data attribute of the class and the method attribute are in same scope, then the data attribute will be given higher priority. So to avoid accidental name conflicts, we may use conventions like:

- capitalizing method names
- prefixing data attribute names with a small unique string(generally an underscore)
- using verbs for methods and nouns for data attributes.

If the class does not contain any statements i.e. it is a class without any attributes or methods , then a keyword 'pass' is given within the body of the class as shown below :

```
class mobile:
    pass
```

In the above example 'pass' is a keyword. Giving pass in the class definition means that the class doesn't define any methods or attributes. But since there needs to be something in the definition, so you use pass. It's a statement that does nothing.

Constructors in Python (Using __init__)

A constructor is a special method that is used to initialize the data members of a class. In python, the built in method __init__ is a sort of constructor. Notice the double underscores both in the beginning and end of init. In fact it is the first method defined for the class and is the first piece of code executed in a newly created instance of the class. But still it should also be remembered that the object has already been constructed by the time __init__ is called, and you already have a valid reference to the new instance of the class through the first argument, self of the __init__ method. Consider the following example:

```
class Initialize:
'''An example of __init__'''
    int var
    def __init__(self, var=10):  #double underscore before and after init
        Initialize.var=var
    def display():
    print var
```

__init__ method can take any number of arguments, and just like functions, the arguments can be defined with default values, making them optional to the caller. Initial values for attributes can be passed as arguments and associated to attributes. A good practice is to assign them default values, even None. In this case, var has a default value of 10. After the class definition, object.__init__(self[, ...]) is called when the instance is created. The arguments are those passed to the class constructor expression. This means the statements given below will give the output 20.

```
P = Initialize(20)
P.display()
```

Also note that if no argument was passed while creating the object, then the __init__ would have taken the default value of var and the output would have been 10.

In Python, the first argument of every class method, including __init__, is always a reference to the current instance of the class and by convention, this argument is always named 'self'. In case of __init__, self refers to the newly created object or the instance whose method was called. Note that the __init__ method never returns a value.

Importance of self

Class methods have only one specific difference from ordinary functions - they must have an extra argument in the beginning of the parameter list. This particular argument is self which is used for referring to the instance. But you need not give any value for this parameter when you call the method. Python provides it automatically. self is not a reserved word in Python but just a strong naming convention and it is always convenient to use conventional names as it makes the program more readable. So while defining your class methods, you must explicitly list self as the first argument for each method, including __init__.

To understand why you don't need to give any value for self during the method call, consider an example. Say you have a class called My_Photo and an instance of this class called My_Object. When you call a method of this object as My_Object.method(arg1, arg2), this is automatically converted by Python into My_Photo.method (My_Object, arg1, arg2). This feature makes self special and it also implies that if you have a method which takes no arguments, then you still have to define the method to have a self argument.

Self is an instance identificator and is required so that the statements within the methods can have automatic access to the current instance attributes. Here is the example showing a class definition using__init__and self.

```
class Mobile:
    '''A sample class definition'''
    price = 0
    model = "Null"
    def __init__(self, price, model = None):
        self.price=price
        self.model="Nokia Lumia 720"
    def displaydata(self):
        print self. price, self. model
```

In the above example:

- The variables price and model are the class variables whose value would be shared among all instances of this class. This can be accessed as Mobile.price, Mobile. model from inside the class or outside the class.
- The first method __init__() is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is self. While giving a call to the method, the instance name is automatically taken as the first argument for self.

If after the given class definition of class Mobile, the following statements are executed

```
M= Mobile(1000, 'Samsung')
M.displaydata()
```

the output is

```
1000 Samsung
```

Class instances (Objects)

Having a class defined, you can create as many objects as required. These objects are called instances of this class. In fact after the class definition is made, a class instance is created automatically once the definition is left normally i.e. the indentation of statements is removed and the class object is called. All the instances created with a given class will have the same structure and behaviour. They will only differ regarding their state, i.e regarding the value of their attributes.

Classes and instances have their own namespaces, that is accessible with the dot ('.') operator. These namespaces are implemented by dictionaries, one for each instance, and one for the class.A class object is bound to the class name given in the class definition header. A class object can be used in two ways - by Instantiation and attribute references.

i) **Instantiation: Creating instance objects**

To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts.

Test = T(1,100)

In the above example T is the instance of class Test.

ii) **Attribute Reference: Accessing attributes of a class**

This is the standard syntax used for all attribute references which is

Object Name. Attribute Name.

As discussed before all the names that were given during the class definition and hence were in the class's namespace are valid attribute names. You access the object's attributes using the dot operator with object as shown in the example below:

```
test.display()
unit_test.display()
print "Marks =", test. marks
```

The search for the referenced attribute is done in the following sequence:

a) A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched.

b) When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes.

c) If no class attribute is found, the object's __getattr__() method is called to satisfy the lookup. You will study about this method later in the chapter.

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a __setattr__() or __delattr__() method, this is called instead of updating the instance dictionary directly. You will learn about these methods later in this chapter.

Class Atttributes v/s Instance Attributes

Attributes can be classified into - Class Attributes and Instance attributes

Class Attributes

These belong to the class itself. These attributes will be shared by all the instances. Such attributes are defined in the class body part, usually at the top, for legibility. Consider the following example:

```
class Health_profile:
            ...
    weight = 89
    blood_group= 'B+'
    ...
```

To access this attribute, you use the dot notation:

```
>>>Health_profile.weight
89
>>>Health_profile.blood_group
B+
```

Instances attributes

As we have learnt, a class may define attributes for its instances. These are called instance attributes and they belong to each instance/object of a class. For example, for the class Health_profile given above, let H1 be an instance. So, the attributes of H1, such as the weight, are directly available through the dot operator:

```
>>>H1.weight
89
```

The dictionary for the instance attributes is also accessible by its __dict__ variable about which you will learn in the next section. To list the attributes of an instance, we have two functions:

i) vars() : This function displays the attributes of the instance in the form of a dictionary. Consider the following example:

```
>>>vars(H1)
{'weight': '89', 'blood group': 'B+'}
```

ii) dir(): This function lists more attributes than vars()because it is not limited to the dictionary of instance. It also displays the class attributes. For example

```
>>>dir(H1)
['__doc__', '__init__', '__module__', 'weight', 'blood_group',]
```

You can add, remove or modify attributes to an instance that were not defined by the class, such as the height in the following:

```
>>> H1.height = 197        # adds 'height' as attribute
>>>vars(H1)
```

```
{'weight': '89', 'blood group': 'B+',height='197'}
>>>H1. height=180          #modifies the value of height
>>>vars(H1)
{'weight': '89', 'blood group': 'B+',height='180'}
>>>del H1.height        #deleted the attribute height
>>>vars(H1)
{'weight': '89', 'blood group'}
```

Here it should always be remembered that this feature of adding and deleting attributes should be used carefully, since by doing this, you start to have instances that have different behaviour than that is specified in the class.

Adding methods dynamically

As you can add, modify and delete the attributes of a class dynamically i.e. at run time, similarly, you can add methods dynamically to an object or class. Consider the code given below:

```
    class Health_profile:
                ...
                weight = 89
                blood_group= 'B+'
    def play():
        print " Come on lets play"
    H=Health_profile()
    H.play=play()
    H.play()
```

In the above example, play is just a function which does not receive self. There is no way by which H can know that play is a method. If you need self, you have to create a method and then bind it to the object. For this you have to import MethodType from types module as shown in the example below:

```
from types import MethodType
class Health_profile(object):
        weight = 89
        blood_group= 'B+'
        def __init__(self,name):
            self.name=name

        def play():
            print " Come on lets play", self. name
```

```
H=Health_profile("Shalini")
H.play=MethodType(play,H)
H.play()
```

In the above code, the built in function Method Type from the types module takes two arguments - the name of the function which has to be bound dynamically and the instance with which it has to bind the function. In the above example the play method will be bound only with the instance, H. No other instances of the class Health_profile will have play method. If we want the other instances also to have play method, then we have to add the method to the class and for that we make use of self as shown in the example below:

```
class Health_profile(object):
        weight = 89
        blood_group= 'B+'
        def __init__(self,name):
            self.name=name
def play(self):
        print " Come on lets play", self.name
Health_profile.play=play()
H1=Health_profile("Shalini")
H1.play()
H2=Health_profile("Ritu")
H2.play()
```

In the above example, note that no method is created with types.MethodType. This is because all functions in the body of the class will become methods and receive self unless you make it a static method about which you will study later in the chapter.

Accessing Attributes and methods

Attributes of a class can also be accessed using the following built in methods / functions:

- **getattr(obj, name[, default]):** This function is used to access the attribute of object.It is called when an attribute lookup has not found the referenced attribute in the class. The built in method for the same is object. __getattr__(self , name)which is called automatically if the referenced attribute is not found. For example:

```
            getattr(H1,weight)
    Built in method for the same will be
            H1.__getattr__(self,weight)
```

The above statement returns the value of the weight attribute otherwise raises an Attribute Error exception. Note that if the attribute is found through the normal mechanism, __getattr__() is not called.

- **hasattr (obj,name):** It is used to check if an attribute exists or not. For example hasattr(H1,weight)

  #will return a True if 'weight' attribute exists

- **setattr (obj, name, value):** It is used to set an attribute. Alternatively object.__setattr__(self, name, value) built in method is called when an attribute assignment is attempted , where name is the name of the attribute and value is its value that is to be assigned. If an attribute does not exist, then it would be created. For example:

```
        setattr(H1,weight,90)
   The built in method for the same will be
        H1.__setattr__(self,weight,90)
   Either of the above statements set the value of the attribute weight as 90.
```

- **delattr(obj, name):** It is used to delete an attribute.The built in method for the same is object.__delattr__(self , name). For example :

```
     delattr(H1,weight)          # deletes the attribute weight
   The built in method for the same will be
     H1.__delattr__(self,weight)
```

Accessing Methods

When an instance attribute other than the data attribute is referenced, the corresponding class is searched. If it is a valid class attribute (a function object), a method is created by pointing to the instance object and the function object. When this method object is called with an argument list, a new argument list is constructed from the instance object and the argument list. The function object is then called with this new argument list. The methods of a class can be accessed in the same manner as the data attributes i.e.

```
     ObjectName.Methodname
```

Now, putting all the concepts together, let us see the following example:

```
class Health_profile:
    weight=0
    blood_group='B+'
    def __init__(self,weight,blood_group):
        self.weight=weight
        self.blood_group=blood_group

    def display(self):
        print " Weight :" , self.weight
        print "Blood Group : " , self.blood_group
```

The following statement will create an object (instance) of class Health_profile

```
    H2=Health_profile(61 ,'A+')        #Assuming weight is 61 and blood group is A+
```

On executing the statement H1.display(), the output will be :

```
    Weight :61
    Blood Group :A+
```

A function object need not always be textually enclosed in the class. We can also assign a function object to a local variable in a class. For example

```
def test ( a ,b):
    return x+y

class myclass:
        F=test(10,20)
        def G(self):
            return F     " Using a function that is defined outside the class"
        H=G
```

In the above example F, G and H are all attributes of class myclass. They refer to the function objects and hence are all methods of instances of myclass.

Methods may also be referenced by global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. Although global data in a method is rarely used, functions and modules imported into a global scope can be used by methods, functions and classes defined in it. Usually a class containing the method is itself defined in this global scope.For example

```
class myclass:
    Yf=x.f()
    while true:
        printYf()
```

In the above example, you will notice that it is not necessary to call a method right away. x.f() is a method object and can be stored(in Yf) and called later (in the while loop).

In case of methods, the object is passed as the first argument of the function. So even if no argument is given in the function call while it was defined in the function definition, no error is flashed. In the above example x.f() is exactly equivalent to myclass.f(x). So we can say that calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

Built in class attributes

Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute:

i) **__dict__ :** It gives the dictionary containing the class's namespace.

ii) **__doc__ :** It returns the class's documentation string(also called docstring) and if no docstring is defined for a class this built in attribute returns None

iii) **__name__:** It gives the class name.

iv) **__module__:** It specifies the module name in which the class is defined. This attribute is called "__main__" in interactive mode.

v) **__bases__ :** It gives a possibly empty tuple containing the base classes, in the order of their occurrence in the base class list. (You will learn about base classes in the next chapter on Inheritance)

For the previously defined class Test let's try to access all the above built in attributes:

```
class Test:
'"A sample class to demonstrate built in attributes"'
    rollno=1
    marks=75
    def __init__(self,rollno,marks):
        self.rollno=rollno
        self.marks=marks
    def display(self):
        print " Roll No : " , self.rollno
        print "Marks : " , self.marks

print "Test.__doc__:" , Test.__doc__
print "Test.__name__:" , Test.__name__
print "Test.__module__:" , Test.__module__
print "Test.__bases__:" , Test.__bases__
print "Test.__dict__:" , Test.__dict__
```

When the above code is executed, it produces the following result:

```
Test.__doc__: A Sample class to demonstrate built in attributes
    Test.__name__: Test
    Test.__module__: __main__
    Test.__bases__: ()
Test.__dict__: {'__module__' : '__main__' , 'display' :
<function display at 0xb8a9872> , 'rollno' :1 , 'marks':75,
'__doc__ : 'A Sample class to demonstrate built in attributes',
'__init__ : <function __init__at 0xb8a89432c>}
```

Using __ del()__

This function is called when the instance is about to be destroyed. This is also called a destructor. It calls the method - object.__del__(self)

When __del()__ is invoked in response to the module being deleted (for example , when the execution of the program is done), the other global variables referenced by __del()__ method may already have been deleted or must be in the process. Let us understand the concept through the class Test whose instance is T1.Consider the following command is given

```
>>>del T1
```

The above command doesn't directly call T1.__del__(). First the reference count is decremented for T1 by one and __del()__ is called only when T1's reference count reaches zero i.e. when all the variables referenced by T1 have been deleted.

Using __ str()__

It is a special function which returns the string representation of the objects. It calls the method object.__str__(self). If the class defines a __str__ method, Python will call it when you call the str() or use print statement. The str() built-in function, when used along with the print statement computes the "informal" string representation of an object. Consider the following example of the class Test defined above.

```
class Test:
    ……….
    ………..
def __str__(self):
    return "Hello, How are you?"
```

Now give the following command on the Python interpreter:

```
>>> T=Test()
>>> print T
    Hello, How are you?
```

When you give the command to "print T", Python calls str(T) to get the string representation of T. If the class of T has a __str__ method, str(T) becomes a call to T.__str__(). This returns the string to print.

Private Members - Limited Support

"Private" instance variables are those that cannot be accessed from outside the class. These attributes, may it be data or methods, can only be accessed from inside an object. These are used if we want to hide an attribute but these types of members don't exist in Python. However, if you still want to hide a member, Python provides a limited support called name mangling. According to this,a name is prefixed with two leading underscores and no more than one trailing underscore will be considered as a private member. For example __pvt , __pvt_will be considered as mangled and so will be treated as private members of the class but not __pvt__ or _pvt__ .

On encountering name mangled attributes, Python transforms these names by a single underscore and the name of the enclosing class, for example:

```
class Test:
    Weight=89
    Blood_group='B+'
    __BP=None              #private member
```

On execution of dir() , notice that the attribute BP if prefixed with an underscore and the class name.

```
>>>dir(Test)
['__doc__', '__init__', '__module__', 'weight', 'blood_group',_Test__BP]
```

Data Hiding

Data hiding is a means to procure encapsulation. Python doesn't really enforce data-hiding in the true sense. The Python approach to getting encapsulation is by simply not using "private" data members. As discussed before, if something is particularly "private" you can use the double leading underscores to mark it as such, but this of course is nothing more than a marking. Basically it just acts as a reminder that such data is intended to be used only within the class. Remember that Python is a dynamic language and that you can add attributes or methods dynamically on an object. In the example given below the attribute pulse is dynamically added to the class Health_profile through H1.

```
class Health_profile:
    pass
>>H1 =Health_profile()
H1.pulse = 75
```

Python's sense of encapsulation means that you don't have to worry about implementation details. Also it does not matter whether or not an attribute has public or private access. As explained in the previous topic, a leading double underscore means that an attribute is private in the sense that if you try to access the attribute directly (via it's name), you won't be able to and this is done by secretly name-mangling the variable with a leading underscore, the name of the class, and then the name of the variable. If you try to access this hidden attribute, then an error message will be displayed.

Consider the following example of class Health_profile:

```
class Health_profile:
        Weight=89
        Blood_group='B+'
        __BP=None                    #private member
        def __init__(self):
                self.__BP = "Hidden attribute"
        def display_BP(self):
                print self.__BP
   >>>H =Health_profile()
   >>>H.__BP
   Traceback (most recent call last):
    File "<input>", line 1, in <module&gt;
   AttributeError: Example instance has no attribute '__BP'
   H.display__BP()
   "Hidden attribute"
```

In the above example, __BP has been defined as private and so an error message is displayed when an attempt is made to access it from outside the class. Even when you try to import using the command- 'from mymodule import *', anything with a double leading underscore doesn't get imported. But still there is an alternative way to access the private members as shown in the following example :

```
   H2=Health_profile()
   H2.Heath_profile__BP="180/90"
```

I know you must be wondering are private members actually hidden? The answer is no.

Static methods

A static method of a Python class is a method that does not obey the usual convention in which self, an instance of the class, is the first argument to the method. It is just like any other function defined in the class but it is callable without creating the instance of the class. They belong to the class but they don't use the object self at all. To declare a static method, declare the method normally but precede it with the built-in staticmethod decorator called@staticmethod. Good candidates for static methods are methods that do not reference the self variable. Consider the following example

```
class Test:
   @staticmethod
```

```
    def Displaytest():
        print "This is a static method"
>>>Test.Displaytest()
This is a static method
```

Note in the above example that the function Displaytest() is not having self as the first argument and that it is being called without the instance. It eases the readability of the code as seeing @staticmethod, we know that the method does not depend on the state of the object. Once you have declared a method to be static, the arguments you pass to it are exactly the arguments it receives. Static methods return the underlying functions with no changes at all. As a result, the function becomes identically accessible from either an object or a class. For example

```
class Test:
    @staticmethod
    def square(x):
        Test.result= x*x
>>>Test.square(5)
>>>Test.result
25
>>>Test().square(6)
>>>Test.result
36
>>>T1=Test()
>>>T1.square(2)
>>>T1.result
>>>4
```

Destroying Objects (Garbage Collection)

Python automatically allocates and de-allocates memory. The user does not have to preallocate or deallocate memory by hand as one has to when using dynamic memory allocation in languages such as C or C++. Python uses two strategies for memory allocation-reference counting and automatic garbage collection.

i) Reference Counting

Prior to Python version 2.0, the Python interpreter only used reference counting for memory management. Reference counting works by counting the number of times an object is referenced by other objects in the system. Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of

aliases that point to it change. An object's reference count increases when it is assigned a new name or placed in a container (list, tuple or dictionary). The object's reference count decreases when it is deleted with del, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically. Consider the code given below:

```
X=50         # an object X is created which is bound to 50.
Y=X          # increase in reference count of 50
Z[0]={Y}     # increase in reference count of 50
del X        # decrease in reference count of 50
Y=10         # decrease in reference count of 50
```

You can see that a class can implement the special method __del__(), called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non-memory resources used by an instance.

Reference counting is extremely efficient but it does have some caveats. One such caveat is that it cannot handle reference cycles. A reference cycle is when there is no way to reach an object but its reference count is still greater than zero. The easiest way to create a reference cycle is to create an object which refers to itself as in the example below:

```
def reference_cycle():
    x=[ ]
    x.append(x)
reference_cycle()
```

In the above example since reference_cycle( ) creates an object x which refers to itself (statement x.append(x)), x will not automatically be freed when the function returns. This will cause the memory that x is using to be held onto until the Python garbage collector is invoked.

ii) Automatic Garbage Collection

In this case garbage collection is a scheduled activity. Python schedules garbage collection based upon a threshold of object allocations and object de-allocations. Python deletes the objects which are not required, may it be built-in types or class instances, through the process named garbage collection. When the number of allocations minus the number of de-allocations are greater than the threshold number, the garbage collector is run and the unused block of memory is reclaimed. One can inspect the threshold for new objects by loading the gc module and asking for garbage collection thresholds.

Automatic garbage collection will not run if your Python device is running out of memory. In such case, your application will throw exceptions, which must be handled otherwise your application will crash. Also, the automatic garbage collection occurs more for the number of free objects than the size of objects.

LET'S REVISE

- **Namespace:** A mapping from names to objects. Examples of namespaces are built-in names, global names in a module and local names in function invocation
- **Scope:** A region of Python program where a namespace is directly accessible.
- In Python a name, storing any type of data, can refer to only one thing at a time.
- The scope of a variable is its enclosing function, class or file.
- The names always belong to the namespace where they are bound.
- Names declared with global keyword have to be referred at the file level.
- **LEGB rule:** when a name is encountered during the execution of the program , it searches for that name in the following order:

  L. Local - It first makes a local search i.e. in current def statement.

  E. Enclosing functions - It searches in all enclosing functions, form inner to outer.

  G. Global (module) - It searches for global modules or for names declared global

  B. Built-in (Python) - Finally it checks for any built in functions in Python.
- Class definitions should be given before it is referenced.
- __init__ is a special method used to initialize the members of a class.
- *self* is the first argument that is passed to the methods of a class.
- A class object can be used in two ways - Instantiation and Attribute reference
- Class attributes belong to the class and will be shared by all instances
- Instance attributes belong to a particular instance of a class only.
- The attributes - data and methods can be added to the class dynamically.
- getattr(obj, name,[default]): is used to access the attribute of the object
- hasattr(obj,name): is used to check if an attribute exists or not
- setattr(obj,name,value): is used to set an attribute with a value.
- delattr(obj,name) : is used to delete an attribute
- __dict__ : gives the dictionary containing class namespace
- __doc__: returns the docstring of a class
- __name__: it gives the class name
- __module__: specifies the module name in which the class is defined

- __bases__: it gives a tuple containing base classes
- __del__: is invoked when the module is being deleted
- __str__: returns the string representation of the objects
- Private variables can only be accessed from inside the objects.
- **Name Mangling:** A name is prefixed with two leading underscores and no more than one trailing underscore.
- **Static Method:** is a method that does not obey the usual convention in which self, an instance of the class, is the first argument to the method.
- Python uses two strategies for memory allocation- Reference counting and Automatic garbage collection.
- **Reference Counting:** works by counting the number of times an object is referenced by other objects in the system. When an object's reference count reaches zero, Python collects it automatically.
- **Automatic Garbage Collection:** Python schedules garbage collection based upon a threshold of object allocations and object de-allocations. When the number of allocations minus the number of de-allocations are greater than the threshold number, the garbage collector is run and the unused block of memory is reclaimed.

EXERCISE

1. Give one word for the following:

   a. A sort of constructor in Python ____________

   b. A region of a Python program where a namespace is directly accessible. _____

   c. It returns the docstring of a class. ___________

   d. It returns the string representation of the object. ___________

   e. A method used to delete an attribute. ___________

2. Define a namespace. Give examples of namespaces with respect to Python.

3. Is data of different types treated differently in Python? Support your answer with an example.

4. Explain LEGB rule.

5. Is object of a class mutable? Why/why not?

6. Explain the usage of keyword 'pass' in class definition.

7. What is the use of __init__? When is it called? Explain with an example.

8. Explain the importance of self in Python classes.

9. Differentiate between class attributes and instance attributes.

10. Explain __str__ with an example.

11. What do you mean by name mangling? Support your answer with relevant example.

12. Differentiate between reference counting and automatic garbage collection with respect to Python.

13. If we want to make the method that is added dynamically to a class available to all the instances of the class, how can it be done? Explain with the help of an example.

14. Predict the output of the following code snippet

    (i)
```
ptr=40
def result():
    print ptr
    ptr=90
def func(var):
    if var<=60:
    ptr=30
```

```
        print ptr
    result()
    func(60)
    func(70)
(ii) ptr=50
    def result():
        global ptr
        ptr=ptr+1
    print ptr
    result()
    print ptr
```

15. Name the methods that can be used to
    a. access attribute of an object
    b. delete an attribute of an object
16. Give the statement to
    a. Check whether the attribute str exists in the class Test whose object is T1
    b. Assign a value "Hello" to the attribute str of class Test and object T1.
17. Consider the following class definition:

```
class Yourclass
        marks=10
        name= "ABC"
        def __init__(self,marks,name):
                self.marks=marks
                self.name=name
        def display(self):
                print marks
                print name
```

Give the statement to create an object of class Yourclass.

18. In the code of Q-13 what will be output if the following command is given: (Assuming YC is the object) YC.display()

19. Predict the output of the following code :

```
class Match:
    ' "Runs and Wickets" '
    runs=281
    wickets=5
    def __init__(self,runs,wickets):
            self.runs=runs
            self.wickets=wickets
        print " Runs scored are : " ,runs
        print "Wickets taken are : " ,wickets
print "Test.__do__:" , Match.__doc__
print "Test.__name__:" , Match.__name__
print "Test.__module__:" , Match.__module__
print "Test.__bases__:" , Match.__bases__
print "Test.__dict__:" , Match.__dict__
```

20. Create the class SOCIETY with following information:

society_name

house_no

no_of_members

flat

income

Methods

- An __init__ method to assign initial values of society_name as "Surya Apartments", flat as "A Type", house_no as 20, no_of_members as 3, income as 25000.
- Inputdata( ) - to read data members(society,house_no,no_of_members&income) and call

allocate_flat().

- allocate_flat( ) - To allocate flat according to income

Income	Flat
>=25000	A Type
>=20000 and <25000	B Type
<15000	C Type

- Showdata( ) - to display the details of the entire class.

21. Define a class ITEMINFO in Python with the following description:

    ICode (Item Code)

    Item (Item Name)

    Price (Price of each item)

    Qty (quantity in stock)

    Discount (Discount percentage on the item)

    Netprice (Final Price)

Methods

- A member function FindDisc( ) to calculate discount as per the following rule:

If Qty<=10	Discount is 0
If Qty (11 to 20)	Discount is 15
If Qty>=20	Discount is 20

- A constructor(__init__ method) to assign the value with 0 for ICode, Price, Qty, Netprice and Discount and null for Item respectively
- A function Buy( ) to allow user to enter values for ICode, Item, Price, Qty and call function FindDisc( ) to calculate the discount and Netprice(Price*Qty-Discount).
- A Function ShowAll( ) to allow user to view the content of all the data members.

Chapter-4: Inheritance

Learning Objectives:

At the end of this chapter the students will be able to:

- Understand the concept of Inheritance
- Understand the types of Inheritance
- Understand Single Inheritance
- Use super() in derived class to invoke _init_()
- Understand Multiple Inheritance
- Understand Overriding methods
- Override methods of parent class
- Learn about abstract methods

In the previous chapter you have studied how to implement concept of classes and object using python. This chapter deals with Inheritance in Python.

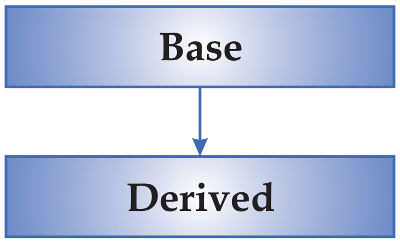
In object oriented programming, inheritance is a mechanism in which a new class is derived from an already defined class. The derived class is known as a subclass or a child class. The pre-existing class is known as base class or a parent class or a super class.The mechanism of inheritance gives rise to hierarchy in classes. The major purpose of inheriting a base class into one or more derived class is code reuse. The subclass inherits all the methods and properties of the super class. The subclass can also create its own methods and replace methods of the superclass. The process of replacing methods defined in super with new methods with same name in the derived class is known as overriding.

Before we learn how we implement the concept of inheritance in Python, let us learn about types of inheritance.

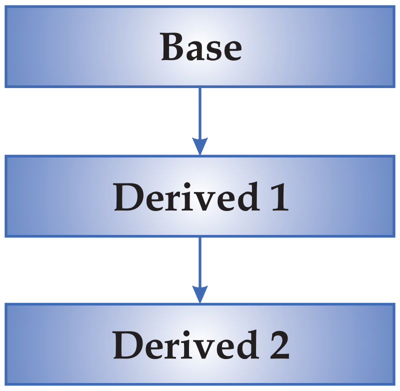
Inheritance can be categorised into five types:

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

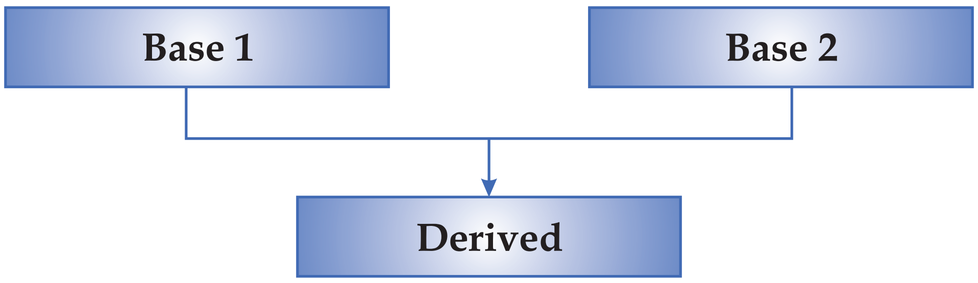
Single Inheritance: This is the simplest kind of inheritance. In this, a subclass is derived from a single base class.



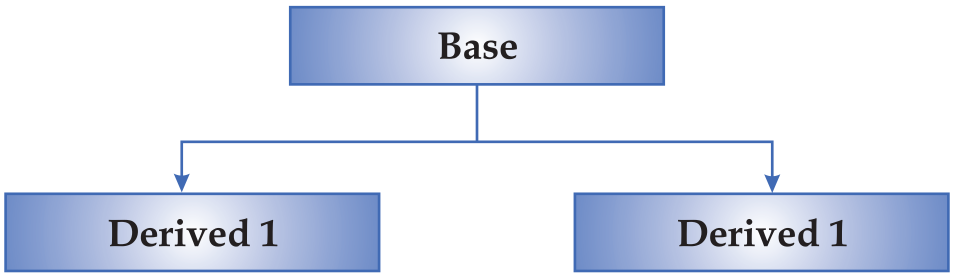
Multilevel Inheritance: In this type of inheritance, the derived class becomes the base of another class.



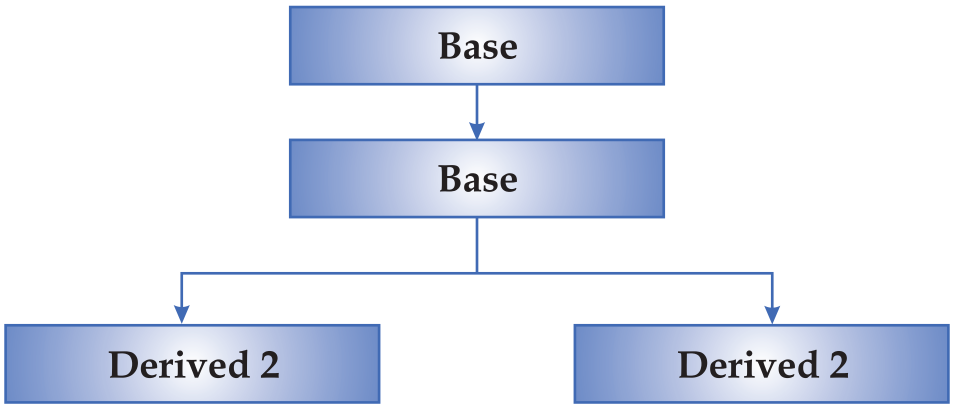
Multiple Inheritance: In this type of inheritance, the derived class inherits from one or more base classes.



Hierarchical Inheritance: In this type of inheritance, the base class is inherited by more than one class.



Hybrid Inheritance: This inheritance is a combination of multiple, hierarchical and multilevel inheritance.



Now let us see how inheritance can be established in Python .This can be done by using the following syntax.

Syntax: class subclass (super):

For Example

```
class person(object):
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def getName(self):
            return self.name
    def getAge(self):
            return self.Age
class student(person):
    def __init__(self,name,age,rollno,marks):
        super(student,self)._init_(self, name, age)
        self.rollno=rollno
        self.marks=marks
    def getRoll(self):
            return self.rollno
    def getMarks(self):
            return self.Marks
```

The above example implements single inheritance. The class student extends the class person. The class student adds two instance variables rollno and marks. In order to add new instance variables, the _init_() method defined in class person needs to be extended. The __init__() function of subclass student initializes name and age attributes of superclass person and also create new attributes rollno and marks.

```
>>p=person("Ram",15)
>>p.getName()
Ram
>>P.getAge( )
15
>> s=student("Sita",16,15,78)
>>s.getName( )
Sita
>>S.getAge( )
16
>>s.getMarks( )
78
```

In the above code segment, the object of class person takes values for instance variables name and age. The object of class student takes values for instance variables name, age, rollno and marks. The object of class student can invoke getName() function because it inherits this function from the base class. Similarly, object of class student can also access instance variables of class person.

In python the above task of extending __init__ () can be achieved the following ways:

i) By using super() function

ii) By using name of the super class.

Method-I

By using super() function

In the above example, the class student and class person both have __init__ () method. The __init__ () method is defined in class person and extended in class student. In Python, super() function is used to call the methods of base class which have been extended in derived class

Syntax:

super(type, variable) bound object

Example

```
class student(person):
    def __init__(self,name,age,rollno,marks):
        super(student, self).__init__(self, name, age)
        self.rollno=rollno
        self.marks=marks
```

Method-II

By using name of the super class

As discussed above, the class student and class person both have __init__ () method. The __init__ () method is defined in class person and extended in class student. In Python, name of the base class can also be used to access the method of the base class which has been extended in derived class.

Example

```
class student(person):
    def __init__(self,name,age,rollno,marks):
        person.__init__(self, name, age)
        self.rollno=rollno
        self.marks=marks
```

Multiple Inheritance

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class SubClassName( Base1, Base2, Base3):
<statement1>
.
.
.
.
<statement N>
```

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in SubClassName, it is searched in Base1, then (recursively) in the base classes of Base1, and only if it is not found there, it is searched in Base2, and so on.

For Example:

```
class student(object):
    def __init__(self,Id,name):
        self.Id=Id
        self.name=name
    def getName(self):
        return self.name
    def getId(self):
        return self.Id
    def show(self):
        print self.name
        print self.Id
class Teacher(object):
    def __init__(self,tec_Id,tec_name, subject):
        self.tec_Id=tec_Id
        self.tec_name=tec_name
        self.subject=subject
    def getName(self):
        return self.tec_name
    def getId(self):
        return self.tec_Id
    def getSubject(self):
        return self.ubject
    def show(self):
        print self. tec_name
        print self.tec_Id
        print self.subject
class school(student,Teacher):
    def __init__(self, ID, name, tec_Id, tec_name, subject, sch_Id):
        student.__init__(self,ID,name)
        Teacher. __init__(self,tec_Id,tec_name, subject)
        self.sch_Id= sch_Id
    def getId(self ):
        return self.sch_Id
    def display(self):
        return self.sch_Id
```

In above example class school inherits class student and teacher.

Let us consider these outputs

```
>>> s1=school(3,"Sham",56,"Ram","FIT",530)
>>> s1.display()
530
>>> s1.getId()
530
>>> s1.show()
Sham
3
```

The object of class school takes six instance variables. The first five instance variables have been defined in the base classes (school and Teacher). The sixth instance variable is defined in class school.

s1.display()displays the sch_Id and s1.getId() also returns the sch_id. The classes school and Teacher both have the method show (). But as shown in above, the objects1 of class school access the method of class student. This because of depth-first, left-to-right rule.

Also, method getId()of class school is overriding the methods with same names in the base classes.

Overriding Methods

The feature of overriding methods enables the programmer to provide specific implementation to a method in the subclass which is already implemented in the superclass. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

Example 1:

```
class student(person):
    def __init__(self,name,age,rollno,marks):
        super(student,self).__init__(self, name, age)
        self.rollno=rollno
        self.marks=marks
def getRoll(self):
        return self.rollno
def getMarks(self):
        return self.Marks
def show(self):
        print self.rollno
        print self.marks
```

As shown in the above example class student inherits class person. Both the classes have a function show( ). The function show() in student is overriding function show() in person(). The object of class person will print name and age. The object of class student will print rollno and marks. In case it is required that function show of class student should display name, age, rollno and marks, we should make the following change in class student

```
class student(person):
    def __init__(self,name,age,rollno,marks):
        super(student,self).__init__(self,name,age)
```

```
            self.rollno=rollno
            self.marks=marks
    def getRoll(self):
            return self.rollno
    def getMarks(self):
            return self.Marks
    def show(self):
        person.show( )
        print self.rollno
        print self.marks
```

Example 2:

```
class student(object):
    def __init__(self,Id,name):
        self.Id=Id
        self.name=name
    def getName(self):
        print self.name
    def getId(self):
        print self.Id
    def show(self):
        print self.name
        print self.Id
class Teacher(object):
    def __init__(self,tec_Id,tec_name, subject):
        self.tec_Id=tec_Id
        self.tec_name=tec_name
        self.subject=subject
    def getName(self):
        print self.tec_name
    def getId(self):
        print self.tec_Id
    def getSubject(self):
        print self.subject
    def show(self):
        print self. tec_name
        print self.tec_Id
        print self.subject
class school(student,Teacher):
    def __init__(self, ID, name, tec_Id, tec_name, subject, sch_Id):
        student.__init__(self,ID,name)
        Teacher. __init__(self,tec_Id,tec_name, subject)
        self.sch_Id= sch_Id
    def getId(self ):
                student.getId(self)
                Teacher.getId(self)
                return self.sch_Id
    def getName(self):
        student.getName(self)
        Teacher.getName(self)
    def show(self):
        student.show(self)
        Teacher.show(self)
        return self.sch_Id
```

Abstract Methods

An abstract method is a method declared in a parent class, but not implemented in it. The implementation of such a method can be given in the derived class.

Method to declare an abstract method

Method

```
>>> class circle(object):
            def get_radius(self):
                    raise NotImplementedError
>>> c=circle()
>>> c.get_radius()
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
c.get_radius()
```

```
File "<pyshell#0>", line 3, in get_radius
raise NotImplementedError
NotImplementedError
```

In the above example, the class circle has a unimplemented method get_radius().

As soon as we call get_radius function using the object of class circle, it raises a "Not Implemented Error".

Now Let us see a class circle1() which implements the abstract method of class circle

```
>>> class circle1(circle):
        def __init__(self,radius):
            self.radius=radius
        def get_radius(self):
            return self.radius
>>> c=circle1(10)
>>> c.get_radius()
10
```

Any class inheriting the class circle canimplement and override the get_radius() method. In case the derived class doesn't implement get_radius() method, it will have to raise"NotImplementedError" exception. This has been shown in the example shown below

```
>>> class circle1(circle):
        def __init__(self,radius):
            self.radius=radius
        def get_radius(self):
            raise NotImplementedError
>>> c1=circle1(10)
>>> c1.get_radius()
Traceback (most recent call last):
File "<pyshell#34>", line 1, in <module>
c1.get_radius()
File "<pyshell#32>", line 6, in get_radius
raise NotImplementedError
NotImplementedError
```

The method of implementing abstract methods by using the "NotImplementedError" exception has a drawback.

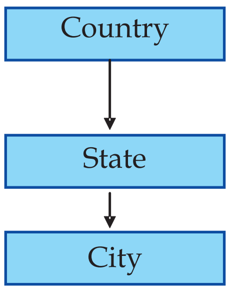
If you write a class that inherits from circle and forget to implement get_radius, the error will only be raised when you'll try to use that method.

LET'S REVISE

- **Inheritance:** In object oriented programming, inheritance is a mechanism in which a new class is derived from an already defined class. The derived class is known as a subclass or a child class. The pre-existing class is known as base class or a parent class or a super class.
- **Single Inheritance:** In single inheritance a subclass is derived from a single base class.
- **Multilevel Inheritance:** In multilevel inheritance, the derived class becomes the base of another class.
- **Multiple Inheritance:** In this type of inheritance, the derived class inherits from one or more base classes.
- **Hierarchical Inheritance:** In this type of inheritance, the base class is inherited by more than one class.
- **Hybrid Inheritance:** This inheritance is a combination of multiple, hierarchical and multilevel inheritance.
- **Overriding Methods:** The feature of overriding methods enables the programmer to provide specific implementation to a method in the subclass which is already implemented in the superclass.
- **Abstract Methods:** An abstract method is a method declared in a parent class, but not implemented in it. The implementation of such a method can be given in the derived class.

EXERCISE

1. Define the term inheritance.
2. Give one example for abstract method?
3. What is single inheritance?
4. What is multiple inheritance? Explain with an example.
5.. Give one example of multilevel inheritance.
6.

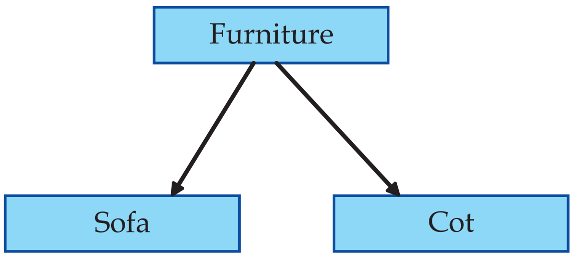


Based on the above diagram, answer the following;

(i) Write the name of the base class and derived class of state.

(ii) Write the type of inheritance depicted in the diagram.

7.

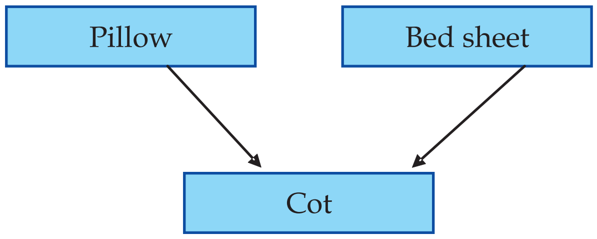


Based on the above diagram, answer the following;

(i) Write the name of the base class and derived classes.

(ii) Write the type of inheritance depicted in the above diagram.

8.



Based on the above diagram, answer the following;

(i) Write the name of the base classes and derived class

(ii) Name the type of inheritance depicted in the above example.

9. Explain the significance of super( ) function.

10. What are abstract methods?

11. Explain the concept of overriding methods.

12. Explain multiple inheritance with the help of an example.

13. How do we implement abstract method in Python. Support your answer with an example.

14. Find the output of the following code and write the type of inheritance:

a)

```
class person(object):
    def __init__(self,name,age):
    self.name=name
    self.age=age
    def display1(self):
    print "Name :",self.name
    print "Age  :",self.age
class student(person):
 def __init__(self,name,age,rollno,marks):
    super(student,self).__init__(name,age)
    self.rollno=rollno
    self.marks=marks
    def display(self):
     self.display1()
    print " Roll No:",self.rollno
     print " Marks  :",self.marks
    p=student('Mona',20,12,99)
p.display()
```

b)

```
class person(object):
def __init__(self,name,age):
    self.name=name
    self.age=age
def display1(self):
    print "Name :",self.name
    print "Age  :",self.age
class student(person):
def __init__(self,name,age,rollno,marks):
    super(student,self).__init__(name,age)
    self.rollno=rollno
    self.marks=marks
def display(self):
    self.display1()
    print " Roll No:",self.rollno
    print " Marks  :",self.marks
class Gstudent(student):
def __init__(self,name,age,rollno,marks,stream):
    super(Gstudent,self).__init__(name,age,rollno,marks)
 self.stream=stream
def display2(self):
    self.display()
    print " stream:",self.stream
p=Gstudent('Mona',20,12,99,'computer')
p.display2()
```

c)

```
class person(object):
    def __init__(self,name,age):
    self.name=name
    self.age=age
def display1(self):
    print "Name :",self.name
    print "Age  :",self.age
class student(object):
def __init__(self,rollno,marks):
    self.rollno=rollno
    self.marks=marks
def display(self):
    print " Roll No:",self.rollno
    print " Marks  :",self.marks
class Gstudent(person,student):
def __init__(self,name,age,rollno,marks,stream):
    super(Gstudent,self).__init__(self,stream)
    person.__init__(self,name,age)
    student.__init__(self,rollno,marks)
    self.stream=stream
def display2(self):
    self.display1()
    self.display()
    print " stream:",self.stream
p=Gstudent('Mona',20,12,99,'computer')
p.display2()
```

15. Rewrite the following code after removing errors. Underline each correction and write the output after correcting the code:

```
class First():
def __init__(self):
    print "first":
class Second(object):
def __init__(self):
    print "second"
class Third(First, Second):
def __init__(self):
    First.__init__(self):
    Second.__init__(self):
    print "that's it"
t=Third()t=Third()
```

16. Complete the following code:

```
class employee(object):
def __init__(self,no,name,age):
    self.no=no
    self.name=_____            #complete the statement
    self.age=______           #complete the statement
def printval(self):
    print "Number:",self.no
    print "Name :",self.name
    print "Age  :",self.age
class pay(object):
    def __init__(self,dept,salary):   #complete the definition
    _____________
    _____________
```

def display(self):   #complete the definition

____________________   # call printval()

__________________ # print dept

___________________# print salary

17. Define a class furniture in Python with the given specifications:

    Instance variables:

    - Type
    - Model

    Methods

    - getType()   To return instance variable Type
    - getModel() To return instance variable Model
    - show()   To print instance variable Type and Model

    Define a class sofa:

    - No_of_seats
    - Cost

    Methods

    - getSeats()   To return instance variable No_of_seats
    - getCost()   To return instance variable Cost
    - show()   To print instance variable No_of_seats and Cost

    This class inherits class furniture

18. Define a class trainer in Python with the given specifications:

    Instance variables:

    - Name
    - T_ID

    Methods

    - getName()To print instance variable Name

2   getTID()To print instance variable T_ID

Define a class learner in Python with the given specifications:

Instance variables:

- L_Name
- L_ID

Methods

- getName()To print instance variable L_Name
- getLID()To print instance variable L_ID

Define a class Institute in Python with the given specifications:

Instance variables:

- I_Name
- I_Code

Methods

- getName()To print instance variable I_Name
- getICode()To print instance variable I_Code

The class Institute inherits the class Trainer and Learner

19. Define a class student in Python with the given specifications:

    Instance variables:

    Roll number, name

    Methods:

    Getdata()- To input roll number and name

    Printdata()- To display roll number and name

    Define another class marks, which is derived from student class

    Instance variable

    Marks in five subjects

    Methods:

    Inputdata() - To call Getdata() and input 5 subjects marks.

    Outdata() - To call printdata() and to display 5 subjects marks.

Implement the above program in python.

20. Define a class employee in Python with the given specifications:

Instance variables:

Employee number, name

Methods:

Getdata()- To input employee number and name

Printdata()- To display employee number and name

Define another class Teaching, which is derived from employee

Instance variable

Department name

Methods:

Inputdata() - To call Getdata() and input department name.

Outdata() - To call printdata() and to display department name.

Define another class Non_teaching, which is derived from employee

Instance variable

Designation

Methods:

Inputdata() - To call Getdata() and input designation.

Outdata() - To call printdata() and to display designation.

Implement the above program in python.

21. Define a class employee in Python with the given specifications:

Instance variables:

Employee number, name

Methods:

Getdata()- To input employee number and name

Printdata()- To display employee number and name

Define another class payroll, which is derived from employee

Instance variable

Salary

Methods:

Inputdata() - To call Getdata() and input salary.

Outdata() - To call printdata() and to display salary.

Define another class leave, which is derived from payroll.

Instance variable

No of days

Methods:

acceptdata() - To call Inputdata() and input no of days.

showdata() - To call Outdata() and to display no of days.

Implement the above program in python.

22. Pay roll information system:

- Declare the base class 'employee' with employee's number, name, designation, address, phone number.
- Define and declare the function getdata() and putdata() to get the employee's details and print employee's details.
- Declare the derived class salary with basic pay, DA, HRA, Gross pay, PF, Income tax and Net pay.
- Declare and define the function getdata1() to call getdata() and get the basic pay,
- Define the function calculate() to find the net pay.
- Define the function display() to call putdata() and display salary details .
- Create the derived class object.
- Read the number of employees.
- Call the function getdata1() and calculate() to each employees.
- Call the display() function.

23. Railway reservation System

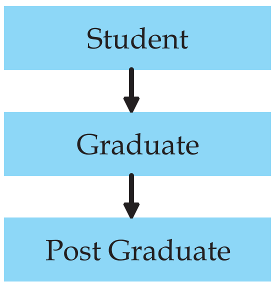
- Declare the base class Train with train number, name, Starting station, destination, departure time, arrival time, etc.
- Define and declare the function getdata() and putdata() to get the train details and print the details.
- Define search() function to search train detail using train number.

- Declare the derived class passenger with ticket number, PNR name of the passenger, gender, age, address, phone number, etc.
- Declare and define the function getdata1() to call search() function to get the train details and get the passenger's information.
- Define the function display() to call putdata() and display passenger's details.
- Create base class object.
- Read the number of trains.
- Create the derived class object.
- Read the number of passengers.
- Call the function getdata1() to each passenger.
- Call the display() function.

24. SKP Hotel offers accommodation, meals facilities.

- Create a class Accommodation with Room Number, type of room, and rent, etc..
- Create a class meals services includes: meals code, name, price, etc..
- Create a class customer with customer number, name, address, etc.
- Customer class is derived by using Accommodation and meals classes.

25. Implement the following using multilevel information.



- Create a student class with student number and name.
- Class graduate is created by using student.
- Graduate class is created using subject code and subject name.
- Class Post Graduate is created by using Graduate.
- Post Graduate class is created using master subject code and master subject name