UNIT 13 CLASSES IN PYTHON

Structure

13.0 INTRODUCTION TO OBJECT ORIENTED PARADIGMS

Object-oriented programming is a programming paradigm which define a class(a group of similar types of objects)and objects (containing both data and methods) to achieve the modularity and portability of the various components of the programs.

In Python programming language, we can define new classes. These classes are customized to a particular application which helps the user to perform its task in an easy way and maintain all the things required in the application.

The reason to define a new class helps in solving the structured application program in terms of object-oriented programming.In structured programming, language functions are defined. These functions are used by the users, but the implementations are hidden by the users. Similarly, a class defines methods and these methods are used by the users but how these methods are implemented are hidden from the users. A customized namespace is associated with every class and objects.

13.1 OBJECTIVES

After going through this unit you will be able to :

- Understand concepts of Object Oriented programming in Python
- Create your own Classes and Objects of classes
- Apply the concept of Inheritance
- Understand the concept of Overloading and Overriding

13.2 CLASSES AND INSTANCES

Classes

A class is a group of similar types of objects.For example, a university is a class, and various objects of the class are open_university, government_university, central_university, state_private_university, private_university, deemed_university.

To create a class we use keyword 'class'. Following is the syntax of class:

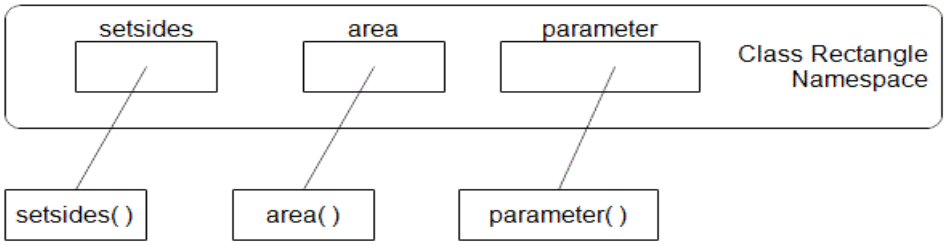
```
Classclass_name:
        class_variable1= value_of_variable
        class_variable2= value_of_variable
        class_variable3= value_of_variable
        …
        defclass_method1(self,arg1,arg2,…):
                method1_code
        def class_method2(self,arg1,arg2,…):
                method2_code
        …
```

The first class we defined is a Rectangle.

```
>>>
class Rectangle:
        defsetsides (self,x,y):
                self.height=x
                self.width=y
                print('The sides of the rectangle are {} and {}:'.format(self.height,self.width))
        def area(self):
                return(self.height*self.width)
        def parameter(self) :
                return(2*(self.height*self.width))
```

It's a convention to write the name of a class with the first character in a capital letter. But it supportsa small letter also. After writing the name of a class,it must be preceded by a colon ':'. In C++ or Java, we were using pair brackets () but python-support colon ':' only.The statement is written after the colon ':' will be taken by the Python interpreter as part of the class document.

When we create a class,a namespace is created for the class Rectangle. This namespace store all the attributes of the class Rectangle. This namespace will specify the names of class Rectangle method.



Object

In Python, whatever value we are storing everything is taken as an object. For example, the string value 'IGNOU University' or the list ['IGNOU',22] or the integer value 28 all are stored in memory as an object. We can think object as a container which stores value in computer memory (RAM).Objects are the containers used to store the values, and it hides the details of the storage from the programmers and only gives the information which is required during the implementation.
Each object contains two things: *types*and*values*.
The type specifies what type of values can be assigned to object or the type of operations that can be operated by the objects.
>>>object1=Rectangle()

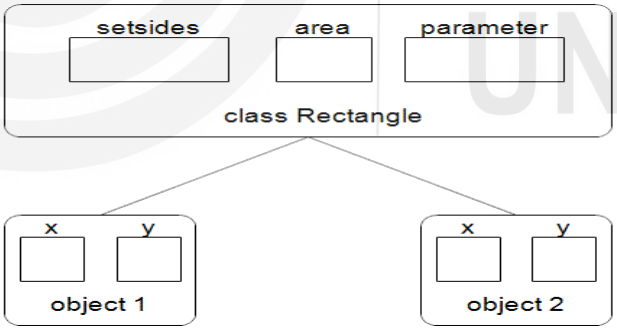
In Python, a namespace is created for each class. Each namespace is specified with a name, and this name is same asclass name—all the attributes of the class use this namespace for the storage.Thus in our example namespace Rectangle must contain names of all the class methods.

Whenever an object object1is created of the class Rectangle(),the separate namespace is created for the object1 also.

Object Object1 namespace

Let us describe the various methods of the class Rectangle:

setsides(x,y) : method to describe the sides of rectangle (height and width).
area() : method which return the area of rectangle.
parameter() : method which return the parameter of rectangle.



The function setsides(),area() and parameter() are defined in the name space. The syntax of methodwould be:

```
defsetsides(self, x,y):
        # implementation of setsides()
def area():
        # implementation of area()
def parameter():
        # implementation of parameter()
```

Instance

Variables that point to the object namespace is called instance variables. Each instance variable containsa different value for different objects. All the

variable define inside the __init__valiable are called as instance variables. And all the variable defined inside the class but outside the __init__valiable is called class variables.A class variable is also called as static variables.

Eample1: Program to calculate area and parameter of a rectangle.

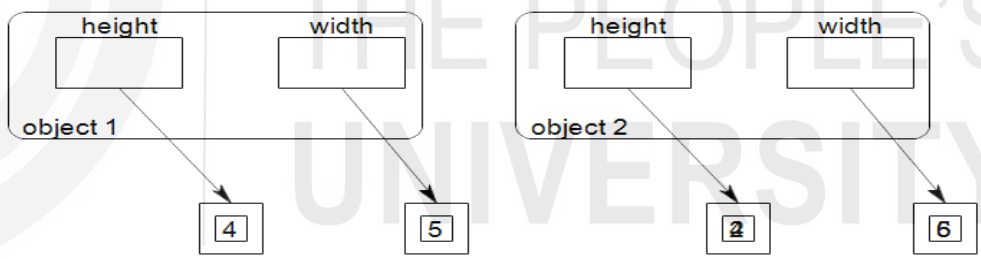
```
class Rectangle:
        def setsides (self,height,weight):              # function to set height and weight of rectangle
                self.side1=height
                self.side2=weight
                print('The sides of the rectangle are {}  and {}:'.format(self.side1,self.side2))
        def area_of_rectangle( self ):                  # function to calculate are of rectangle
                return(self.side1*self.side2)
        def parameter_of_rectangle(self) :              # function to calculate parameter of rectangle
                return(2*(self.side1*self.side2))

object1=Rectangle()
object1.setsides(4,5)
object2=Rectangle()
object2.setsides(2,6)
#area_of_rectangle=object1.area_of_rectangle()
#paramater_of_rectangle=object1.parameter_of_rectangle()
print('The area of Rectangle 1 is {} '.format(int(object1.area_of_rectangle())))
print('The parametera of Rectangle 1 is {} '.format(int(object1.parameter_of_rectangle())))
print('The area of Rectangle 2 is {} '.format(int(object2.area_of_rectangle())))
print('The parametera of Rectangle 2 is {} '.format(int(object2.parameter_of_rectangle())))
```

Ln: 20 Col: 91

```
==================== RESTART: D:/python/IGNOU/Example 1.py ====================
The sides of the rectangle are 4  and 5:
The sides of the rectangle are 2  and 6:
The area of Rectangle 1 is 20
The parametera of Rectangle 1 is 40
The area of Rectangle 2 is 12
The parametera of Rectangle 2 is 24
>>>
>>>
```

Ln: 12 Col: 4



13.3 CLASSES METHOD CALLS

To call a method of a class first, we have to create an object of the same class. After object creation, we will envoke the method defined in a class with the dot '.' operator.

When we create an object of the class, the constructors declared in the classenvoke automatically to which it belongs. In Python, there is a unique method _ _init_ _ to implement constructor of the class in which it is defined. The first argument of the method _ _init _ _ must be self. Self is a reference to a class to which this object belongs.

Example 2: Program to call a constructor

```
class Rectangle:
        def __init__ (self):
                print ("Method envoke without call")
        obj1=Rectangle()
        obj2=Rectangle()
```

```
Ln: 13  Col: 0
===================== RESTART: D:\python\IGNOU\Example 2.py ====================
Method envoke without call
Method envoke without call
```

When two objects obj1 and obj2 are created __init__ method is called automatically.

Class	Methods
Predefined word *class*is used to describe a class.	Predefined word *def*is used to describe the method of a class.
Each class statement defines a new type with a given name.	A def statement defines a new function with a given name.
Each class name is preceded by a colon ':' statement.	Each method name is preceded by a colon ':' statement.
Example -- class Rectangle:	Example -- defsetsides(self):

Table.13.1 Comparison between class and method

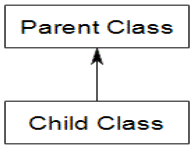
13.4 INHERITANCE AND COMPOSITIONS

Inheritance is the mechanism by which object acquire the property of another object derived in a hierarchy.

The class which is inherited is called a base class or parent class or superclass, and the inheriting class is called the derived class or child class or subclass.For our reference, we will use the term parent class and child class. A child class acquire all the properties of the parent class, but the vice versa is not true; parent class can't access any features of a child class. The main motto behind the inheritance is code reusability. Once a code is defined in a class if it is required by another class then by inheriting the class code can be reused.

Syntax of inheritance:

class<Child Class>:
If this child class inherits the parent class, then the statement will be changed to
class<Child Class> (<Parent class>):



Consider aclass Parent containing two methods methodA1() and methodA2() and a class Child with method methodB1() and methodB2().Child class is defined to be the subclass of Parent class.Therefore it inherits both the methods methodA1() and methodA2() of Parent class .

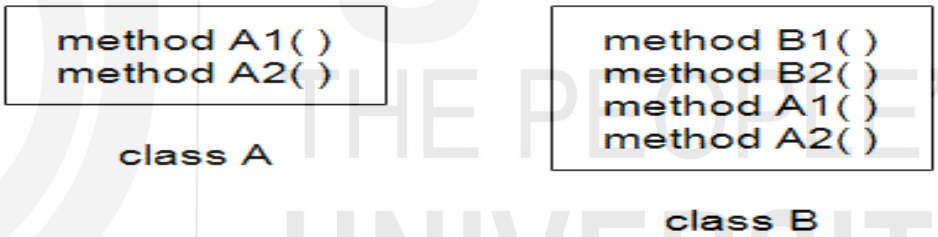
Example 3: Inheritance of a parent class by child class

```
class Parent:
        def methodA1():
                print("Method 1 of Parent class ")
        def methodA2():
                print("Method 2 of Parent class ")
class Child(Parent):
        def methodB1():
                print("Method 1 of Child class ")
        def methodB2():
                print("Method 2 of Child class ")
obj1=Parent
obj1.methodA1()
obj1.methodA2()
```

Ln: 6 Col: 20

```
===================== RESTART: D:/python/IGNOU/Example 3.py =====================
Method 1 of Parent class
Method 2 of Parent class
>>>
```

After inheritance class child contains methods methodB1() and methodB2() as well as methods of class Parent , methodA1() and methodA2().



Example 4: Calling method of parent class by object of child class.

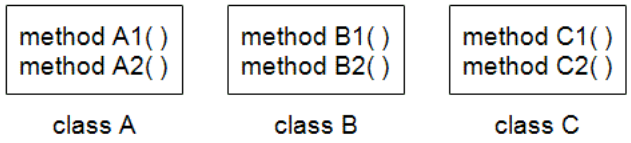
```
class Parent:
        def methodA1():
                print("Method 1 of Parent class ")
        def methodA2():
                print("Method 2 of Parent class ")
class Child(Parent):
        def methodB1():
                print("Method 1 of Child class ")
        def methodB2():
                print("Method 2 of Child class ")
obj2=Child
obj2.methodA1()
obj2.methodA2()
obj2.methodB1()
obj2.methodB2()
```

Ln: 8 Col: 35

```
===================== RESTART: D:/python/IGNOU/Example 4.py =====================
Method 1 of Parent class
Method 2 of Parent class
Method 1 of Child class
Method 2 of Child class
>>>
```

Multilevel inheritance:

Consider three classes class A,class B and class C

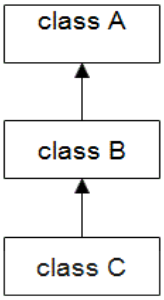


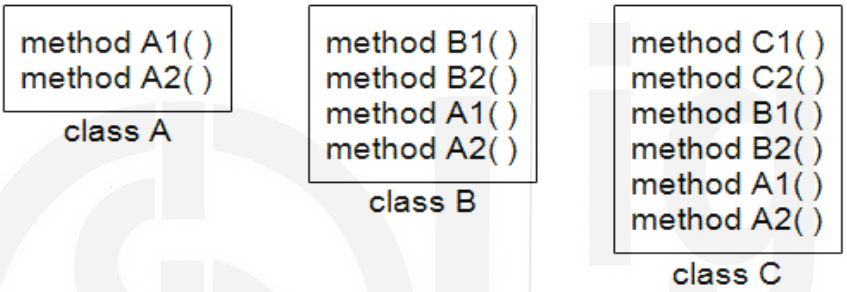
When an inherited class is inherited by another class, this is called multilevel inheritance.

Consider Class B inherits class A, and class C inherits class B.

Class B is called subclass of Class A,and Class C is called subclass of class B.

Class C willcontain all the attributes and methods of class A and class B.

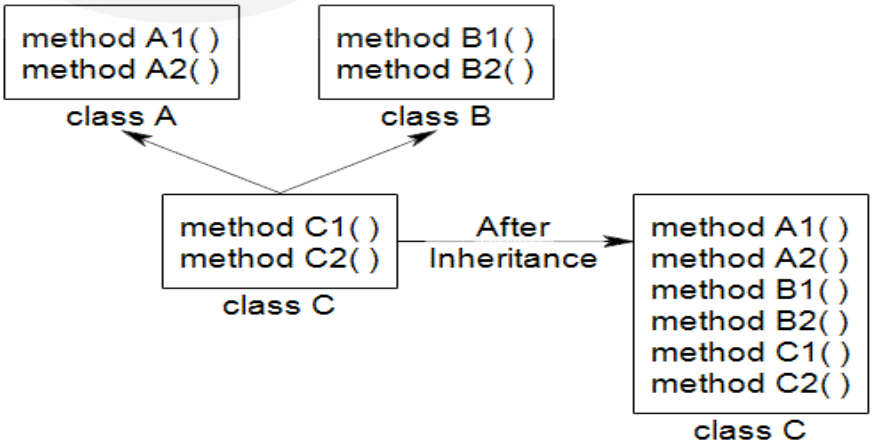




Multiple Inheritance:

When a single class inherit two or more classes,then it is called multiple inheritances.

Consider two classes, class A with two methods methodA1(),methodA2() and class B with methodB1() and methodB2(). If another class C inherits both class A and class B,then class B will inherit the features of class A and B.



Example 5: Implementation of multiple inheritance

```
class C1:                    #Parent class 1
        def methodA1():
                print("Method 1 of class C1")
        def methodA2():
                print("Method 2 of class C1")
class C2:                     #Parent class 2
        def methodB1():
                print("Method 1 of class C2")
        def methodB2():
                print("Method 2 of class C2")

class C3(C1,C2):            # Child class of class C1 and Class C2
```

Ln: 12 Col: 62

```
==================== RESTART: D:/python/IGNOU/Example 5.py ====================
Method 1 of class C1
Method 1 of class C2
Method 1 of class C3
>>>
```

The behaviour of constructor in inheritance:

Example 6: Constructor of class

Example 6.py - D:\python\IGNOU\Example 6.py (3.8.5)

File Edit Format Run Options Window Help

```
class C1:
        def __init__(self):  #Contructor of class C1
                print("Constructor of C1")
        def methodC1():
                print("Method 1 of class C1")
class C2(C1):
        def __init__(self):  #Contructor of class C1
                print("Contructor of C2")

obj1=C1()
obj2=C2()
```

Ln: 3 Col: 41

```
==================== RESTART: D:\python\IGNOU\Example 6.py ====================
Constructor of C1
Contructor of C2
>>>
```

Since the creation of an object of a class will automatically invoke the constructor of the class. Here in the example class C1 object will call a constructor of class C1 automatically, and creation of an object of class C2 will call the constructor of class C2 automatically.

Creating an object of child classwill first search the__init__method of the child class. If __init__ method is in the child class then it will be executed first if it is not in the child class then it will go to the __init__ method of the parent class.

Example 7:

```
Example 7.1.py - D:/python/IGNOU/Example 7.1.py (3.8.5)
File  Edit  Format  Run  Options  Window  Help
class C1:
        def __init__(self):
                print("Contructor of C1")
        def methodA1():
                print("Method 1 of class C1")
class C2(C1):
        def methodC2():
                print("Method of Class B")
obj1=C2()
                                                        Ln: 5  Col: 31
==================== RESTART: D:/python/IGNOU/Example 7.1.py ===================
Contructor of C1
>>>
```

If we want to call the init method of the parent class also then we can call it with the help of keyword super.

Example 8: Use of super keyword to call a method of the parent class.

```
class C1:
        def __init__(self):          # Constructor of class C!
                print("Contructor of C1")
        def methodC1():
                print("Method 1 of class C1")
class C2(C1):
        def __init__(self):
                super().__init__()   #Calling parent class C1 constructor
                print ("Constructor of C2")
        def methodC2():
                print("Method of Class C2")
obj1=C2()
                                                        Ln: 2  Col: 0
Contructor of C1
Constructor of C2
>>>
```

Thus when we create an object of the child class, it will call the init method of the child class first. If we have called super, then it will first call init of the parent class then it will call int of the child class.

If we have two classes, A and B inherited by class C. If init method is defined in all three classes then what will happen if call inits of the parent class with the help of super() keyword? Then which class init method will be called?

Example 9: Uses of the super keyword in multiple inheritances.

```
class C1:
        def __init__(self):          #Constructor of class C1
                print("Controctor of C1")
        def methodA1():              #Method of class C1
                print("Method 1 of class C1")
class C2:
        def __init__(self):          #Constructor of class C2
                print("Controctor of C2")
        def methodB1():              #Method of class C2
                print("Method 1 of class C2")

class C3(C1,C2):                     # Inheriting class C1 and C2
        def __init__(self):
                super().__init__()
                print("Controctor of C3")
obj1=C3()
```

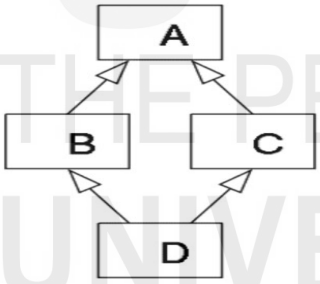
Ln: 2 Col: 0

```
Controctor of C1
Controctor of C3
>>>
```

It will first call the init method of the child class then with the help of super() keyword it will call the init method of the left parent class that is A in our example.

In multiple inheritances, when we inherit the classes, then the method is executed based on the order specified, and this is called Method Resolution Order (MRO).

If class A is inherited by class B and Class C and then class D inheriting class B and class C.



Then according to the Method of Resolution Order (MRO) the orderof execution of the methods and attributes are: class D→class B→class C→class A

13.5 STATIC AND CLASS METHODS

A static variable is that variable whose value remainsthe same for all the objects of the class. It creates one copy of the variable which is shared by all objects of the class. To declare a static variable, it must be declared inside the class. No method is used to declare a static variable it is declared without any method.

To use the static variable, we will use the class name to which this static variable belong, or we will use the reference of the object. But it is always preferred to use class name in place of reference of the object.

Example 10: Implementation of static variable.

```
class Check:
        a=100        # static variable
        def __init__(self):
                self.b=200
obj1=Check()
obj2=Check()
print("obj1:", obj1.a, obj1.b)
print("obj 2:", obj2.a, obj2.b)
Check.a=888
obj1.b=999
print("t1:", obj1.a, obj1.b)
print("t2:", obj2.a, obj2.b)
```

Ln: 13 Col: 0

```
obj1: 100 200
obj 2: 100 200
t1: 888 999
t2: 888 200
>>>
```

In Python, there are three types of methods: Instance methods, class methods and static methods.

Instance methods:

Example 11: Implementation of an instance method.

```
class MyClass:
        def  __init__(self,a1,b2,c3):   # constructor
                self.a1=a1
                self.b2=b2
                self.c3=c3
        def avg(self):                  # instance method
                return((self.a1+self.b2+self.c3))/3
obj1=MyClass(3,3,6)
print(obj1.avg())
```

Ln: 10 Col: 0

```
==================== RESTART: D:/python/IGNOU/Example 11.py ====================
4.0
>>>
```

In the above example first method __init__ is a constructor of the class. The second method avg(self)is the instance method of the class.This method contains one parameter self, which points to the instance of the classMyClass, 3.0 will be printed as an output.

When the method is called Python, replace the self argument with the instance of the object.Thus instance method always works on the object.

Class method

```
class MyClass:
        classvariable= "my class variable"
        @classmethod
        def classmethod(cls):           #class method
                return cls.classvariable
print(MyClass.classmethod())
```

```
Ln: 10  Col: 0
==================== RESTART: D:/python/IGNOU/Example 12.py ====================
my class variable
>>>
```

A class method is a method which is bound to the class and not the object of the class. A special symbol called a decorator ‘ @’ followed by the keyword classmethod is used to define the class method.

A class method can be called by the class or by the object of the class to which this method belongs. The first parameter of the class method is class itself.Thus as an instance method is used to call the instance variable similarly class method is used to call class variables.

Static method

Suppose we are looking to a method which is not a concern to the instance variable neither to the class variable. In that case, we will use a static method.Static method in Python isused when such methods are called another class or methods are used to perform some mathematic calculations based on the values received as an argument. A special symbol called as a decorator ‘@’ is used followed by the keywordstaticmethod is used to define a static method. Thus a static method can be invoked without the use of the object of the class.

```
class MyClass:
    @staticmethod
    def staticmethod():     #static method
        print("Calling of static method")

MyClass.staticmethod()
```

```
Ln: 6  Col: 22
==================== RESTART: D:\python\IGNOU\Example 13.py ====================
Calling of static method
>>>
```

Thus if we want to work with the variables other then class variable and instance variable, we will use static.

13.6 OPERATOR OVERLOADING

Consider an operator ‘+.’

>>>2+5

7

Here it performs arithmetic addition of two numbers.

```
>>>[2,3,4]+[5,6]

[2,3,4,5,6]
```

Here it performs concatenation of two lists.

```
>>> 'IGNO'+ 'U University'

IGNOU University
```

Here it performs concatenation of two strings.

The operator '+' is said to be an overloaded operator. If an operator is defined for more than one classes, then it is called the overloaded operator. For each class, the implementation of the overloaded operator is different. In our example, the overloaded operator ' +' is defined for the in-class,list class and string class.
The Python interpreter will take the operator '+' as x.__add__(y), and this is called method invocation.
__add__(..)

x.__add__(y) equivalent to x+y
Thus when we are performing 2+5, it is

```
>>int(2).__add(5)
7
>>>[2,3,4].__add__[5,6]
[2,3,4,5,6]

>>> 'IGNO'. __add__ 'U University.'
'IGNOU University'
```

Operator	Method	Number	List &String
n1 + n2	n1.__add__(n2)	Adding n1&n2	Concatenation
n1–n2	n1.__sub__(n2)	Subtracting n2 from n1	—
n1 * n2	n1.__mul__(n2)	Multiplyingn1 & n2	Self concatenation
n1 / n2	n1.__truediv__(n2)	Dividing n1 by n2	—
n1 // n2	n1.__floordiv__(n2)	Integer division of n1 by n2	—
n1 % n2	n1.__mod__(n2)	Remainder after division of n1 by n2	—
n1 == n2	n1.__eq__(n2)	n1 & n2 both are same	
n1 != n2	n1.__ne__(n2)	n1 & n2 both are different	
n1>n2	n1.__gt__(n2)	n1 is larger than n2	
n1 >= n2	n1.__ge__(n2)	n1 is large than n2or equal to n2	
n1 < n2	n1.__lt__(n2)	n1 is small than n2	
n1<= n2	n1.__le__(n2)	n1 is small than or equal to n2	
repr(n1)	n1.__repr__()	Canonical representation of string n1	
str(n1)	n1.__str__()	Informal representation of string n1	
len(n1)	n1.__len__() —	Size of n1	
<type>(n1)	<type>.__init__(n1)	Constructor	

Table 13.2: Overloaded operators

If we want to add a and b where a=2 and b= ‘RAM.’

```
>>>a+b
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Because different types are not defined with the operator addition.

Consider a Student class

Example 14: Addition of two objects.

```
class Student:
        def __init__(self,n1,n2):
                self.n1=n1
                self.n2=n2
s1=Student(82,73)
s2=Student(56,97)
s3=s1+s2
```

Ln: 2 Col: 0

```
==================== RESTART: D:/python/IGNOU/Example 14.py ====================
Traceback (most recent call last):
  File "D:/python/IGNOU/Example 14.py", line 7, in <module>
    s3=s1+s2
TypeError: unsupported operand type(s) for +: 'Student' and 'Student'
>>>
```

When we add two objects s1 & s2, it will show an error. Here it is not defined to the Python interpreter to add two objects. Hence we have to overload operator add.

Example 15: Operator overloading

```
class Student:
        def __init__(self,n1):
                self.n1=n1

        def __add__(self,other):
                return self.n1+other.n1

s1= Student("Ignou")
s2= Student(" University")

print(s1+s2)
```

Ln: 5 Col: 25

```
==================== RESTART: D:/python/IGNOU/Example 15.py ====================
Ignou University
>>>
```

Here we overloaded operator add.

Suppose we want to compare s1and s2

Example 16: Implementation of a comparison operator

```
class Student:
        def __init__(self,n1):
                self.n1=n1
s1= Student(10)
s2= Student(20)
if s1>s2:
                print('s1 wins')
else:
                print('s2 wins')
```

Ln: 11 Col: 0

```
==================== RESTART: D:/python/IGNOU/Example 16.py ====================
Traceback (most recent call last):
  File "D:/python/IGNOU/Example 16.py", line 6, in <module>
    if s1>s2:
TypeError: '>' not supported between instances of 'Student' and 'Student'
>>>
```

The comparison operator is not defined to the Python interpreter for the objects. Hence we have to redefine it or overload it. The function which corresponds to the symbol greater than ‘>’ is __gt__ (refer to table 13.2). So we have to overload this operator by defining the function.

Example 17: Overloading operator '>.'

```
class Student:
        def __init__(self,n1):        #class constructor
                self.n1=n1
        def __gt__(self,other):       # overloading operator >
                p1=self.n1
                p2=other.n1
                if p1>p2:
                        return True
                else:
                        return False
s1= Student(10)
s2= Student(20)
if s1>s2:
                print('s1 wins')
else:
                print('s2 wins')
```

Ln: 12 Col: 15

```
==================== RESTART: D:/python/IGNOU/Example 17.py ====================
s2 wins
>>>
```

Ln: 41 Col: 48

because the value of s2 is higher than s1 hence s2 wins.

Now, what will happen if we want to print object s1

```
>>>print(s1)
…
<__main__.Student object at 0x00000192B57EE390>
```

It will print the address of the object. Now we want to print the value of the object then we have to redefined the function __str__().

```
>>>def __str__(self):
                return (' {} , {}'.format(self.n1,self.n2))
```

The value of two objects s1 and s2 printed.

13.7 POLYMORPHISM

Poly means 'multiple' and Morph means 'forms'. Thus polymorphism is multiple forms. For example, human being behaves differently in a different environment. The behaviour of a human in the office is different from his behaviour at home, which is different from his behaviour with friends at a party.

Thus in terms of object orientation due to polymorphic characteristic object behave differently in a different situation.

There are four ways to implement polymorphism in Python:
Duck Typing
Operator loading
Method Overloading
Method Overriding

Duck typing

There is a sentence in the English language "if this is a bird which is walking like a duck,quacking like a duck and swimming like a duck then that bird is a duck". It means if the behaviour of the bird is like a duck, then we can say it a Duck.

X=4
and
X= 'Mohit'

In the first statement, X is a variable storing integer value. The type of the X is int. However, in the second statement X= 'Mohit' the storage memory taken by the

variable is a string. In Python, we can't specify the type explicitly. During the runtime, whatever value we are storing in a variable the type is considered automatically. And this is called Duck Typing principle.

Example 18: Duck Typing principle

```
Example 18.py - D:/python/IGNOU/Example 18.py (3.8.5)
File  Edit  Format  Run  Options  Window  Help
class Animal_Dog:
        def execute(self) :
                print ("Bow..Bow")
class Bird_Duck:
        def execute(self):
                print ("Quack..Quack")
class Animal:
        def code(self,ide):
                ide.execute()


ide=Bird_Duck()
obj=Animal()
obj.code(ide)
Ln: 17  Col: 0
==================== RESTART: D:/python/IGNOU/Example 18.py ====================
Quack..Quack
>>>
```

In the above example, the obj is an object of class Animal. When we call the code of Animal class, we have to pass an object ide. So before passing ide, we have to define it at the object of Duck class,one more ide for the class dog is defined. At the moment when we assign ide as an object of Dog class, then it will execute the dog method.

Example 19:

```
class Animal_Dog:
        def execute(self) :
                print ("Bow..Bow")
class Bird_Duck:
        def execute(self):
                print ("Quack..Quack")
class Animal:
        def code(self,ide):
                ide.execute()


ide=Animal_Dog()
obj=Animal()
obj.code(ide)
Ln: 17  Col: 0
==================== RESTART: D:/python/IGNOU/Example 19.py ====================
Bow..Bow
>>>
```

So it doesn't matter which class object we are passing the matter is that object must have executed method. And this is called duck typing.

Operator Loading

Consider A=2 and B=5 are two variables. In a programming language, when we are performing a+b, it will perform the addition of two integer numbers. If X= 'Hello' and Y= 'Hi' then x+y will perform addition of two strings.

A=2

B=5

>>>print(A+B)
…
7
Python interpretor will take it as print(init.__add__(a,b)), where add() is a method of the init class.
>>>print(init.__add__(a,b))
7

X= 'Hello'
Y= 'Hi'
>>>print (X+Y)
HelloHi
Python interpretor will take print(X+Y) as print(str.__add__(X,Y)), where add()is a method osstr class.
>>>print(str.__add__(X,Y))
HelloHi

Method Overloading

Consider two classes having methods with the same name, but with a different number of parameters or different types of parameters, then the methods are called overloaded. In overloaded methods, the number of arguments is different, or types of arguments are different.

```
class student:
        marks(a,b)
        marks(a,b,c)
```

here two methods marks are defined one with two parameters and another having three parameters. These two methods are called method overloading. Python does not support method overloading.

If there are multiple methods in a class having the same name then Python will consider only the method which is described at the end of the class.

Example 20: Implementation of method overloading.

```
class MethodOverloading:
        def methodl(self):
                print('Methodl without argument')
        def methodl(self,x):
                print('Method with only single argument')
        def methodl(self,x,y):
                print('Method with two arguments')
obj1=MethodOverloading()
obj1.methodl()
obj1.methodl(2)
obj1.methodl(3,4)
```

Ln: 12 Col: 0

```
==================== RESTART: D:/python/IGNOU/Example 20.py ====================
Traceback (most recent call last):
  File "D:/python/IGNOU/Example 20.py", line 9, in <module>
    obj1.methodl()
TypeError: methodl() missing 2 required positional arguments: 'x' and 'y'
>>>
```

Method Overriding

In Python, we can't create the same methods with the same name and the same number of parameters. But we can create methods of the same method in the classes

derived in a hierarchy. Thus the child class redefined the method of the parent class, and this is called method overriding.

Example 21: Implementation of method overriding.

```
class University:
        def UnivName(self):
                print('IGNOU University')
        def course(self):
                print('BCA')
class student(University):
        def course(self):        # overridded method
                print('MCA')

studl=student()
studl.UnivName()
studl.course()
```

Ln: 3 Col: 27

```
==================== RESTART: D:/python/IGNOU/Example 21.py ====================
IGNOU University
MCA
>>>
```

A method which is overridden in a child class can also access the method of its parent class with the help of keyword super().

Example 22: Method overriding with the super keyword

```
class University:
        def UnivName(self):
                print('IGNOU University')
        def course(self):
                print('BCA')
class student(University):
        def course(self):        # overrided method
                super().course()# calling super class method
                print('MCA')
studl=student()
studl.UnivName()
studl.course()
```

Ln: 13 Col: 0

```
==================== RESTART: D:/python/IGNOU/Example 22.py ====================
IGNOU University
BCA
MCA
>>>
```

Check Your Progress - 1

1. Implement the class Circle that represents a circle. The class must contain the following methods:

   Circle_setradius():Takes one number of values as input and sets the radius of the circle.

   circle_perimeter(): Returns the perimeter of the circle.

   Circle_area(): Returns the area of the circle.

   ---

2. Define method overloading and constructor overloading in Python.

3.Select the correct output generated from the following program code:

```
class code1:
        def __init__(self,st="Welcome to Python World"):
                self.st=st
        def output(self):
                print(self.st)
        obj=code1()
        obj.output()
```

a) The code result an error because constructor are defined with default arguments

b) Output in not displayed

c) "Welcome to Python World" is printed

d) The code result an error because parameters are not defined in a function

---

4.What type of inheritance is illustrated in the following Python code?

```
class Class1():
        pass
class Class2(Class1):
        pass
class Class3(Class2):
        pass
```

a) Multilevel inheritance

b) Multiple inheritance

c) Hierarchical inheritance

d) Single-level inheritance

---

5. What is polymorphism and what is the main reason to use it?

13.8 SUMMARY

In this unit, we discussed the concepts of classes and objects. Concept of a namespace to store object and class in memory is also defined in this unit. Different types of class methods namely static methods ,instance methods and class methodsare discussed in this unit.

This unit also focused on inheritance and various types of inheritance: single level inheritance,multilevel inheritance and multiple inheritances.

In this unit, it is described that the same operator can be used for multiple purposes, and this is called operator overloading. I list of operators and corresponding methods, also called as magic methods are also given in the unit.

Polymorphism means it is the ability to behave differently in a different situation. The concept of polymorphism and the implementation of polymorphism with Duck Typing, Operator loading, Method Overloading and Method Overriding are also described in the unit.

SOLUTIONS TO CHECK YOUR PROGRESS

Check Your Progress -1

1.
```
class Circle:
        defcircle_setradius(self, radius):
        self.r = radius
defcicle_perimeter(self):
        return 2 * 3.142 * self.r
defcicle_area(self):
        returnself.x * self.x*3.142
```
2. Iftwo or methods are defined with the same then but they differ in return types or having a different number of arguments or having different types of arguments then these methods are called as method overloading.

   Two constructors are overloaded when both are having different no or different types of arguments.

   In the python method overloading and constructor, overloading is not possible.
3. C
4. A
5. Polymorphism is one of the main features of object-oriented programming languages. With this characteristic, we can implement elegant software.