Chapter 5: Classes and Object-Oriented Programming

Overview of OOP Terminology

- **Class** – A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** – A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** – A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** – The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** – A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** – The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** – An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** – The creation of an instance of a class.
- **Method** – A special kind of function that is defined in a class definition.
- **Object** – A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** – The assignment of more than one function to a particular operator.

Features of OOPS:

- Classes and Objects
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Classes and Objects:

- Classes are essentially a template to create your objects.
- Objects are an encapsulation of variables and functions into a single entity. Objects get their variables and functions from classes.

General form of class:

class classname(object):

"docstring of the class"

attributes

def __init__(self):

def method1():

def method2():

- The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon.
- 'object' represents the base class name from where all classes in python are derived. It is not compusary.
- The first string is called docstring and has a brief description about the class. Although not mandatory, this is recommended.
- "__init__" is a reserved method in python classes. It is known as a constructor in object oriented concepts. This method called when an object is created from the class and it allow the class to initialize the attributes of a class.
- self represents the instance of the class. By using the "self" keyword we can access the attributes and methods of the class in python. When we create an instance for a class , a separate memory block is allocated on the heap and that memory location is by default stored a 'self'.

Example of class:

```
class MyClass:
        "This is my class"
        def __init__(self):
                self.x=10
        def func (self):
                print("Value of x = ",self.x)
ob = MyClass()# create a new MyClass
ob.func()   # Output: Value of x=10 # Calling function func()
```

Creating an Object in Python

class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
>>> ob = MyClass()
```

- This will create a new instance object named ob. We can access attributes of objects using the object name prefix.
- Attributes may be data or method. Method of an object are corresponding functions of that class.
- Any function object that is a class attribute defines a method for objects of that class.
- In above example **'ob'** nothing but instance name. When we create instance than internally following steps take place:
  1. A block of memory is allocated on heap memory according to attributes and methods.
  2. After allocating memory block, it will call a special method named __init__(self) is called internally. Methods stores the initial data into variables like a constructor.
  3. Finally the allocated memory address of the instance is returned to 'ob' object. you can find the address of ob by **id(ob) method.**

Encapsulation

- It is a mechanism where the data (variables) and the code(methods) that act on the data will bind together.
- For example, if we take a class, we write he variable and methods inside the class. Thus, class is binding them together.
- **class** is a example of encapsulation.

Abstraction (Data Hiding)

- There may be a lot of data, a class contains and the user does not need the entire data.
- The user requires only some part of the available data.
- In this case, we can hide the unnecessary data from the user and expose only that data that is interest to the user. This is called abstraction.
- A good Example for abstraction is a car. Any car will have some parts like engine, radiator, battery etc. The user of the car should know how to drive and does not require knowledge of these parts.

Abstraction in python

- In language like java, we have keywords like public, private, protected to implement various level of abstraction. **In python these keywords are not available.**
- **Everything written in the class comes under public.** Suppose we don't want to make a variable available outside the class than write the variable two double scores before it as:

```
class Myclass:
    def __init__(self):
        self.__age=50
    def data(self):
        print ("Hello", self.age)
s = Myclass()
#print (s.age) # It gives error because age is private
print(s._Myclass__age) # valid way to access private member
```

output:

50

if comment section will removed than output is:

output :

```
Traceback (most recent call last):
  File "C:/Python27/private_demo.py", line 7, in <module>
    print (s.age) # It gives error because age is private
AttributeError: Myclass instance has no attribute 'age'
```

Here , one underscore before class name and two underscore after class name is used to access private variable that is called **name mangling**.

The following table shows the different behavior: [**instance member name rule**]

Name	Notation	Behavior
name	Public	Can be accessed from inside and outside
_name	Protected	Like a public member, but **they shouldn't be directly accessed** from outside.
__name	Private	Can't be seen and accessed from outside directly but can access using name mangling

Example public, private, protected data declaration: [Note : access is always public in python, it is only declaration]

File: encapsulation.py

```
class Encapsulation(object):
    def __init__(self, a, b, c):
        self.public = a
        self._protected = b
        self.__private = c
```

```
>>> from encapsulation import Encapsulation
>>> x = Encapsulation(11,13,17)
>>> x.public
11
>>> x._protected
13
>>> x._protected = 23
>>> x._protected
23
>>> x.__private
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Encapsulation' object has no attribute '__private'
>>> x._Encapsulation__private
17
```

Inheritance

Syntax:

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

```
class BaseClass:
  Body of base class

class DerivedClass(BaseClass):
  Body of derived class
```

Polymorphism

- If an object or method is exhibiting different behavior in different context, it is called polymorphism nature.
- It is provide flexibility to use same method call to perform different operations depending on the requirement.

Example:

```
def add (a,b):
        print a+b
add (5,10)                  #15
add("hello","students")     #hellostudents
```

Constructor in Python (__init__(self) function in class):

- A constructor is a special type of method (function) that is called when it instantiates an object using the definition found in your class.
- The constructors are normally used to initialize (assign values) to the instance variables. Constructors also verify that there are enough resources for the object to perform any start-up task.
- A constructor is a class function that begins with **double underscore (__)**.
- The name of the constructor is always the same **__init__()**.
- While creating an object, a constructor can accept arguments if necessary.
- When you create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.

Example:

```
class Myclass:
        def __init__(self):
                self.name = 'Raj';
                self.age=24;
        def data(self):
                print ("Hello", self.name)
                print ("Hello", self.age)
s = Myclass()
s.data()
```

output:

```
('Hello', 'Raj')
('Hello', 50)
```

Parameterized constructor:

```
class Myclass:
        def __init__(self,name,age):
                self.name = name;
                self.age=age;
        def data(self):
                print ("Hello", self.name)
                print ("Hello", self.age)
s = Myclass("Raj",50)
s.data()
```

Output:

```
('Hello', 'Raj')
('Hello', 50)
```

Destructor in Python (__del__(self) function in class):

- Destructors are called when an object gets destroyed. It's the polar opposite of the constructor, which gets called on creation.
- This method is called before destruction of the object. It is not called manually but completely automatic.
- A destructor is a special type of method (function) definition found in your class that is called when it destroyed an object using del.
- The destructor is normally used to close the resources used by instance variables.
- A destructor is a class function that begins with **double underscore (__)**.
- The name of the constructor is always the same **__del__(self)**.
- While creating an object, a constructor can accept arguments if necessary.
- When you create a class without a destructor, Python automatically creates a default destructor that doesn't do anything.
- **Example:**

```
class Vehicle:
    def __init__(self):
        print('Vehicle created.')
    def __del__(self):
        print('Destructor called, vehicle deleted.')

car = Vehicle()
del car
```

Output:

Vehicle created.
Destructor called, vehicle deleted.

Types of variables:

1. Instance variable
2. class variable or static variable

❖ Instance Variable :

- When instance variable is created separated copy is created in every instance. If 'x' is an instance variable and if we create 3 instance , there will be 3 copies and we modify the copy of 'x' in any instance, it will not modify other two copies.

Example:

```
class Myclass:
   def __init__(self):#consructor that incement n when instance is created
      self.x=10
   def modify(self):
      self.x +=1
s1=Myclass()
s2=Myclass()
print ("Before Modify for s1 , x is = ", s1.x)
print ("Before Modify for s2 , x is = ", s2.x)
s1.modify()
print ("After Modify for s1 , x is = ", s1.x)
print ("After Modify for s2 , x is = ", s2.x)
```

output:

```
('Before Modify for s1 , x is = ', 10)
('Before Modify for s2 , x is = ', 10)
('After Modify for s1 , x is = ', 11)
('After Modify for s2 , x is = ', 10)
```

❖ class variable / static variable :

- class variable are the variables whose single copy is available to all the instances of the class.
- If we modify the copy of class variable in an instance , it will modify all the copies in the other instances.

Example:

```
class Myclass:
   x=10
   @classmethod
   def modify(cls):
      cls.x +=1
s1=Myclass()
s2=Myclass()
print ("Before Modify for s1 , x is = ", s1.x)
print ("Before Modify for s2 , x is = ", s2.x)
s1.modify()
print ("After Modify for s1 , x is = ", s1.x)
print ("After Modify for s2 , x is = ", s2.x)
```

Output:

```
('Before Modify for s1 , x is = ', 10)
('Before Modify for s2 , x is = ', 10)
('After Modify for s1 , x is = ', 11)
('After Modify for s2 , x is = ', 11)
```

Types of methods

1. Instance Methods
2. class Methods
3. static methods

❖ Instance methods

- It is the method which act upon the instance variable of the class. It is bounded to instance and called as: instance_name.method_name()
- It requires memory address of the instance that is provided through 'self' variable by default as first parameter.
- While calling it we need not pass any value to the 'self' variable.

Example:

```
class MyClass:
        "This is my class"
        def __init__(self):
                self.x=10
        def func (self):
                print("Value of x = ",self.x)
ob = MyClass()# create a new MyClass
ob.func()
```

Output:

10

Instance methods are of two types: 1. accessor methods 2. mutator methods

Accesor methods simply access or read the data of the variables. It is getXXX() method.

```
def getName(self):
        return self.name
```

Mutator methods can modify data, It is setXXX() method.

```
def setName(self):
        self.name=name
```

❖ Class Methods

- These methods act on class level.class methods are the methods which act on the class variables or static variables.
- It is written as @classmethod decorator. First parameter of class methods must be 'cls'.
- It is generally called as class_name.method_name()

Example:

```
class MyClass:
   "This is my class"
   x =10
   @classmethod
   def func(cls):
      print("Value of x = ",cls.x)
MyClass.func()
```

Output :

('Value of x = ', 10)

❖ Static methods

- We need static methods when the processing is at the class level but we need not involve the class or instances.
- For example , setting environment variable, counting the no of objects etc.
- It is written as @staticmethod & called as class_name.method()

Example:

```
class Myclass:
   n=0#class  or static variable
   def __init__(self):#consructor that incement n when instance is created
      Myclass.n+=1
   #static method to display no of instance
   @staticmethod
   def noObj():
      print("No of obj created is {}".format(Myclass.n))
obj1=Myclass()
obj2=Myclass()
obj3=Myclass()
Myclass.noObj()
```

Output :

No of obj created is 3

Inheritance

- Inheritance is a powerful feature in object oriented programming.
- It refers to defining a new class with little or no modification to an existing class.
- The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Python Inheritance Syntax

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

```
class BaseClass:
  Body of base class

class DerivedClass(BaseClass):
  Body of derived class
```

```
class Animal:
    def eat(self):
      print 'Eating...'

class Dog(Animal):
   def bark(self):
      print 'Barking...'

d=Dog()
d.eat()
d.bark()
```

output:

```
Eating...
Barking...
```

Multilevel Inheritance in Python

Multilevel inheritance is also possible in Python unlike other programming languages. You can inherit a derived class from another derived class. This is known as multilevel inheritance. In Python, multilevel inheritance can be done at any depth.

Python Multilevel Inheritance Example

```
class Animal:
    def eat(self):
      print 'Eating...'
class Dog(Animal):
   def bark(self):
      print 'Barking...'
class BabyDog(Dog):
    def weep(self):
```

```
        print 'Weeping...'
d=BabyDog()
d.eat()
d.bark()
d.weep()
```

Output:

```
Eating...
Barking...
Weeping
```

The super() method

- super() is a built in method which is useful to call the super class constructor or method from the sub class.

Example:

```
class father(object):
    def __init__(self,property=0):
        self.property=property
    def display(self):
        print "Father's property =" , self.property
class son(father):
    def __init__(self,property1=0,property=0):
        super(son,self).__init__(property)
        self.property1=property1
    def display(self):
        super(son,self).display()
        print "Son's property = ", self.property+self.property1
s = son(2000,3000)
s.display()
```

output:

```
Father's property = 3000
Son's property =  5000
```

Multiple Inheritance in Python

- Python supports multiple inheritance also. You can derive a child class from more than one base (parent) class.
- The multi derived class inherits the properties of both class base1 and base2.Let's see the syntax of multiple inheritance in Python.

Example:

```
class father(object):
    def height(self):
        print "Height is 6.0 feet"
class mother(object):
    def color(self):
        print "Color is brown"
class son(father,mother):
    def display(self):
        pass
s = son()
s.color()
s.height()
```

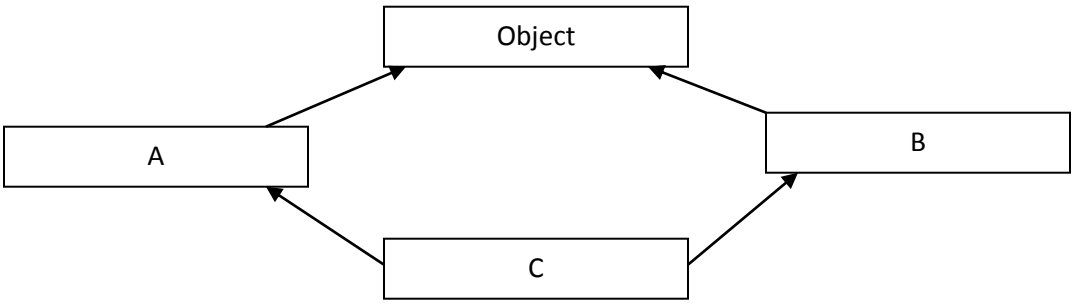
output:

Color is brown

Height is 6.0 feet

Problems in Multiple Inheritance :

- If the sub class has a constructor , it overrides the super class constructor and hence the super class constructor is not available to the sub class.
- Let's take a example : class C is derived from class A and B . If class C wants to call super class constructor than : super().__init__() is used.
- Here First constructor of 'C' class is used and than 'A' class.



- In the above figure, class A is at the left side and class B is at the right side for the class constructor is accessed first . As a result i will display 'c'.
- Here, super().__init__() will call the super class constructor of the class which is at the left the side. So class A's constructor is called and gives output 'A'.
- If class A doesn't have a constructor than it will call the constructor of right hand side class.

- If class C wants to access instance variable of both of its super class than super().__init__() will be used in every class.
- Here search will be start from class C.First it will display 'c'.
- Here, super().__init__() will call the super class constructor of the class which is at the left the side.
- So class A's constructor is called and gives output 'A'. Here super().__init__() of class A is called , since 'object' is a super class for class A but object class does not have constructor.
- So the search will continue down to right hand side class of object class , that class B. Hence, class B's constructor is executed and 'b' is displayed.
- After that class B's super().__init__() called , that calls 'object' class , Since it is already visited so search stops here.
- Searching in this manner for constructors or methods is called **" Method Resolution Order [MRO]"**.

Method Resolution Order [MRO]:

- In the multiple inheritance , any specified attribute or method is searched first in the sub class.
- If not found , the search continues into parent classes in depth first left to right fashion without searching the same class again.
- Searching in this manner for constructors or methods is called **" Method Resolution Order [MRO].**

There are three principles followed by MRO :

1. The first principle is to search for the sub class before going to super classes.
2. The second principle is that when a class is inherited from several classes , it searches in the order from left to right in the base class.
3. The third principle is that it will not visit any class more than once. That means a class in the inheritance hierarchy is traversed only once exactly.

To know the MRO , we can mro() method as:

```
class_name.mro()
```

Example that demonstrate MRO:

```
class A(object):
    def __init__(self):
        print('A: before init')
        super(A,self).__init__()
        print('A: after init')

class B(A):
    def __init__(self):
        print('B: before init')
        super(B,self).__init__()
        print('B: after init')

class C(object):
    def __init__(self):
        print('C: before init')
        super(C,self).__init__()
        print('C: after init')

class D(B, C):
    def __init__(self):
        print('D: before init')
        super(D,self).__init__()
        print('D: after init')

d = D()
print D.mro()
```

Output:

```
====================== RESTART: E:/LJ/Python/MRODemo.py ======================
D: before init
B: before init
A: before init
C: before init
C: after init
A: after init
B: after init
D: after init
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class '__mai
n__.C'>, <type 'object'>]
>>>
```

Polymorphism

- If an object or method is exhibiting different behavior in different context, it is called polymorphism nature.
- Python has built-in polymorphism. Method overloading, method overriding, duck typing and operator overloading are example of polymorphism.

❖ Duck typing:

- python variables are names that point to memory locations where data is stored.
- if 'x' is a variable and we assign x =5 or x="hello" than we can conclude two points :
  1. python type system is strong because every variable or object has a type that we can check with type() function.
  2. python's type system is 'dynamic' since the type of a variable is not explicitly declared but it changes with the content being stored.
- Similarly , if we want to call a method on an object we do not need to check the type of the object and we do not need to check whether that method really belongs to that object or not.
- So in python we never worry about the type (class) of objects. The object type is distinguished only at runtime.
- If " it walks like a duck and talks like a duck , it must be a duck " this is the principle we follow , This is called duck typing.

❖ Method Overloading :

- Method overloading is not supported in python. Writing same name for more than one method is not possible in python.
- We can achieve method overloading by writing same method with several parameter. The method performs the operation depending on the no of arguments passed in the method call.

Example:

```
class Myclass:
   def sum(self,a=None,b=None,c=None):
      if a!=None and b!=None and c!=None :
         print "a+b+c =", a+b+c
      elif a!=None and b!=None :
         print "a+b = ", a+b
      else:
         print "Enter two or three no as a input"
s1=Myclass()
s1.sum(10,20,30)
s1.sum(10,20)
s1.sum(10)
```

Output:

```
a+b+c = 60
a+b =  30
Enter two or three no as a input
```

❖ Method Overriding

- When there is a method in the super class, writing the same method in sub class so that it replaces the super class method is called 'method overriding'.

Example:

```
class Parent:       # define parent class
  def myMethod(self):
    print 'Calling parent method'
class Child(Parent): # define child class
  def myMethod(self):
    print 'Calling child method'
c = Child()         # instance of child
c.myMethod()        # child calls overridden method
```

output:

Calling child method

❖ Operator Overloading

- Python operators work for built-in classes.
- But same operator behaves differently with different types.
- For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.
- This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

```
print (10+15)
s1="hello"
s2="student"
print(s1+s2)
a=[1,2,3]
b=[4,5,6]
print(a+b)
```

output:

```
25
hellostudent
[1, 2, 3, 4, 5, 6]
```

- But we can't use the addition operator to two objects as obj1+obj2.
- Our intention of writing this to add data of two objects but it is not possible. Let's see the below example:

```
class BookX:
   def __init__(self,pages):
      self.pages=pages
class BookY:
   def __init__(self,pages):
      self.pages=pages
```

```
p1=BookX(100)
p2=BookY(200)
print('Total Pages', p1+p2)
```

Output :

```
Traceback (most recent call last):
  File "C:/Python27/overloading1.py", line 9, in <module>
    print('Total Pages', p1+p2)
TypeError: unsupported operand type(s) for +: 'instance' and 'instance'
```

TypeError was raised since Python didn't know how to add two objects together.
The '+' operator internally written as a special method, __add__(self,other). It is internally called a.__add__(b)

```
class BookX:
    def __init__(self,pages):
        self.pages=pages

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

class BookY:
    def __init__(self,pages):
        self.pages=pages
p1=BookX(100)
p2=BookY(200)
print('Total Pages:' p1+p2)
output:
('Total Pages', 300)
```

What actually happens is that, when you do p1 + p2, Python will call p1.__add__(p2) which in turn is BookY.__add__(p1,p2). Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator Overloading Special Functions in Python		
Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)

Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()
Less than or equal to	p1<=p2	p1.__le__(p1)
Equal to	p1 == p1	p1.__eq__(p1)
Not equal to	p1!=p2	p1.__nq__(p1)
Greter than or equal to	p1>=p2	p1.__ge__(p1)

Operator	Function Implimentation	Method Description
+	__add__(self, other)	Addition
*	__mul__(self, other)	Multiplication
-	__sub__(self, other)	Subtraction
%	__mod__(self, other)	Remainder
/	__truediv__(self, other)	Division
<	__lt__(self, other)	Less than
<=	__le__(self, other)	Less than or equal to
==	__eq__(self, other)	Equal to
!=	__ne__(self, other)	Not equal to
>	__gt__(self, other)	Greater than
>=	__ge__(self, other)	Greater than or equal to
[index]	__getitem__(self, index)	Index operator
in	__contains__(self, value)	Check membership
len	__len__(self)	The number of elements
str	__str__(self)	The string representation

Abstract class

- Abstract method is a method whose action is redefined in the sub classes as per the requirement of the objects.
- Abstract methods are written without body since their body will be defined in the sub classes anyhow. But it is possible to write abstract method with body also.
- To mark as abstract, we should use the **decorator @abstractmethod**.
- Abstract class is a class that generally contains some abstract methods.
- Since, abstract class contains abstract methods whose implementation is later defined in the sub classes, it is not possible to estimate the total memory required to create the object for the abstract class. So PVM **can't create objects to an abstract class**. It is possible to create an object of sub classes.
- Since all abstract classes should be derived from the meta class ABC which belongs to abc (Abstract base class) module , we should import this module into our program.
- **A meta class** is a class that defines the behavior of other classes. The meta class ABC defines that the class which is derived from it becomes an abstract class.
- To import abc module's ABC class and abstract method decorator we can write as follow:
  from abc import ABC, abstarctmethod
  or from abc import *

Example:

```
from abc import ABCMeta , abstractmethod
class Myclass():
   __metaclass__ = ABCMeta #It is used for python 2 version
   @abstractmethod
   def calculate(self,x):
      pass
class sub1(Myclass):
        def calculate(self,x):
                print('Square value =', x*x)
import math
class sub2(Myclass):
        def calculate(self,x):
                print('Square root=', math.sqrt(x))
class sub3(Myclass):
        def calculate(self,x):
                print('Cube Value=', x**3)
obj1=sub1()
obj1.calculate(16)
obj2=sub2()
obj2.calculate(16)
```

```
obj3=sub3()
obj3.calculate(16)
```

output:

```
Square Value= 256
Square root = 4.0
Cube Value = 4096
```

Interface in Python

- Abstract class is a class which contains abstract method as well as concrete method.
- Where interface is an abstract class but it contains only abstract methods.
- None of the methods in the interface will have body.
- Only method headers will be written in the interface.
- The interface concept is not explicitly available in python.
- We have to use abstract classes as interfaces in python.
- Since an interface contains methods without body, it is not possible to create objects to an interface.
- So we can create sub classes where we can implement all the methods with body, it is possible to create objects to the sub classes.

Example of interface:

```
 #abstract class works like an interface
from abc import *
class Myclass():
    __metaclass__ = ABCMeta
    def connect(self):
        pass
    def disconnect(self):
        pass
#this is a sub class
class Oracle(Myclass):
    def connect(self):
        print 'Connecting to Oracle Database'
    def disconnect(self):
        print 'Disconnected from Oracle Database'
#this is a another sub class
class Mysql(Myclass):
    def connect(self):
```

```
        print 'Connecting to Mysql Database'
    def disconnect(self):
        print 'Disconnected from Mysql Database'
class Database:
    #accept database name as a string
    str = input('Enter Database Name')
    #convert the string into classname
    classname = globals()[str]
    #create an object to that class
    x=classname()
    #call the connect() and disconnect() methods
    x.connect()
    x.disconnect()
'''
#Output :
Enter Database Name "Oracle"
Connecting to Oracle Database
Disconnected from Oracle Database
'''
```

OR

```
'''
x = Mysql()
x.connect()
x.disconnect()

x = Oracle()
x.connect()
x.disconnect()
```

Output:

```
Connecting to Mysql Database
Disconnected from Mysql Database
Connecting to Oracle Database
Disconnected from Oracle Database
```

Abstract data type

- Abstract data type is a mathematical model for data types where data types are defined based on its behavior from user's point of view.
- String, List,tupel,dictionary,stack are example of abstract data types.

Stack ADT:

- stack is a collection of elements which are added and removed from the end known as 'top'. Stack is known as LIFO.

Stack operations:

1. stack(): creates new empty stack.
2. push(element): insert new element .
3. pop(): removed top element.
4. peek(): return top element.
5. isEmpty(): check whether stack is empty or not.
6. size(): return the size of stack.

Example :

```
class Stack:
    def __init__(self):
        self.items = []
    def IsEmpty(self):
        return self.items == []
    def push(self,item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
s=Stack()
s.push(10)
s.push(3.5)
s.push(2)
s.push('zishan')

print("is Stack empty ? ", s.IsEmpty())
print("Size of Stack is ::", s.size())
print("Size of Stack is ::", s.pop())
print("Size of Stack is ::", s.peek())
```

Output:

```
('is Stack empty ? ', False)
('Size of Stack is ::', 4)
('Size of Stack is ::', 'zishan')
```

('Size of Stack is ::', 2)

Object creation in python and __new__(cls) function in class:

- __new__ is a static method which creates an instance. We will see the method signature soon. One reason i could think of having __new__ as a static method is because the instance has not been created yet when __new__ is called. So, they could not have had it as an instance method.
- __new__ gets called when you *call the class*. *Call the class* means issuing the statement "a=A(1,2)". Here A(1,2) is like calling the class. A is a class and we put two parenthesis in front of it and put some arguments between the parenthesis. So, its like "calling the class" similar to calling a method.
- __new__ must return the created object.
- Only when __new__ returns the created instance then __init__ gets called. If __new__ does not return an instance then __init__ would not be called. Remember __new__ is always called before __init__.
- __new__ gets passed all the arguments that we pass while calling the class. Also, it gets passed one extra argument that we will see soon.
- _new__ handles object creation and __init__ handles object initialization

Example:

```
class A(object):  # -> don't forget the object specified as base
   def __new__(cls):
      print "A.__new__ called"
      return super(A, cls).__new__(cls)
   def __init__(self):
      print "A.__init__ called"
A()
```

Output:

```
A.__new__ called
A.__init__ called
```

Note: __new__ is called automatically when calling the class name (when instantiating), whereas __init__ is called every time an instance of the class is returned by __new__ passing the returned instance to __init__ as the 'self' parameter, therefore even if you were to save the instance somewhere globally/statically and return it every time from __new__, then __init__ will be called every time you do just that.

Knowing this it means that if we were to ommit calling super for __new__ then __init__ won't be executed. Let's see if that's the case:

Example:

```
class A(object):  # -> don't forget the object specified as base

   def __new__(cls):
      print "A.__new__ called"
      #return super(A, cls).__new__(cls)

   def __init__(self):
      print "A.__init__ called"
a = A()
print a
```

Output:

```
A.__new__ called
None
```

Generators:

A generator is a function that returns an object (iterator) which we can iterate over (one value at a time). It is as easy as defining a normal function with **yield** statement instead of a **return** statement.

If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both **yield** and **return** will return some value from a function.

The difference is that, while a **return** statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

Differences between Generator function and a Normal function

Here is how a generator function differs from a normal function.

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

Python Generators with a Loop and next():

Example:

```
def mygen(n):
    for i in range(0,n):
        yield i

genobj = mygen(5)

print ("Type : ", type(genobj))
print ("Value : 1 : ", next(genobj))
print ("Value : 2 : ", next(genobj))

print ("Using loop :")
for data in genobj:
    print (" data - ",data,)
```

Output:

```
Type :  <class 'generator'>
Value : 1 :  0
Value : 2 :  1
Using loop :
 data -  2
 data -  3
 data -  4
```

Example:

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
        yield my_str[i]
#For loop to reverse the string
for char in rev_str("hello"):
    print(char)
```

Output:

o
l
l
e
h

Example : Consider the code:

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

    def __str__(self):
        return self.name

class Grades(object):
    """A mapping from students to a list of grades"""
    def __init__(self):
        """Create empty grade book"""
        self.students = []

    def addStudent(self, student):
        """Assumes: student is of type Student
        Add student to the grade book"""
        if student in self.students:
            raise ValueError('Duplicate student')
        self.students.append(student)

    def getStudents(self):
        """Return a list of the students in the grade book"""
        for s in self.students:
            yield s

s1 = Student('Jyovita')
s2 = Student('Hemali')
s3 = Student('Ankita')

Grade_list = Grades()
Grade_list.addStudent(s1)
Grade_list.addStudent(s2)
Grade_list.addStudent(s3)

for s in Grade_list.getStudents():
    print s
```

Output:

```
Ankita
Jyovita
Hemali
>>>
```

Any function containing a yield statement is treated in a special way.

The presence of yield tells the Python system that the function is a generator.

Generators are typically used in conjunction with for statements.

At the start of the first iteration of a for loop, the interpreter starts executing the code in the body of the generator. It runs until the first time a yield statement is executed, at which point it returns the value of the expression in the yield statement.

On the next iteration, the generator resumes execution immediately following the yield, with all local variables bound to the objects to which they were bound when the yield statement was executed, and again runs until a yield statement is executed. It continues to do this until it runs out of code to execute or executes a return statement, at which point the loop is exited.

The version of getStudents in the above code, allows programmers to use a for loop to iterate over the students in objects of type Grades in the same way they can use a for loop to iterate over elements of built-in types such as list.

Que: Create a class Employee with data members: name, department and salary. Create suitable methods for reading and printing employee information

File: EmployeeDemo.py

```
#Create a class Employee with data members: name, department and salary.
#Create suitable methods for reading and printing employee information
class Employee:
    def __init__(self,name='hemali',dept='ce',salary=15000):
        self.name = name
        self.department = dept
        self.salary = salary

    def printdata(self):
        print "Employee Detail:"
        print "Name :", self.name
        print "department :", self.department
        print "salary :", self.salary

    def getdata(self):
        self.name = raw_input('Enter name:')
        self.department = raw_input('Enter department:')
        self.salary = raw_input('Enter salary:')
print "Employee - 1"
e1 = Employee()
e1.printdata()
print "Employee - 2"
e2 = Employee('Kalpan','IT',100000)
e2.printdata()
print "Employee - 3"
e3 = Employee()
e3.getdata()
```

e3.printdata()

Output:

```
============== RESTART: E:/LJ/Python/Sar
Employee - 1
Employee Detail:
Name : hemali
department : ce
salary : 15000
Employee - 2
Employee Detail:
Name : Kalpan
department : IT
salary : 100000
Employee - 3
Enter name:Aarav
Enter department:CE
Enter salary:12000000
Employee Detail:
Name : Aarav
department : CE
salary : 12000000
>>>
```

Que: Write a python program in which Maruti and Santro sub classes implement the abstract methods of the super class Car.

File:CarDemo.py

#Write a python program in which Maruti and Santro sub classes
#implement the abstract methods of the super class Car.

```
from abc import ABCMeta , abstractmethod
class Car():
    __metaclass__ = ABCMeta #It is used for python 2 version
    @abstractmethod
    def whoAmI(self,x):
        pass
class Maruti(Car):
    def whoAmI(self,x):
        print 'I am  = Maruti - Owned By - ', x

class Santro(Car):
    def whoAmI(self,x):
        print 'I am  = Santro - Owned By - ', x

c1 = Maruti()
c1.whoAmI("Hemali")
c2 = Santro()
c2.whoAmI("Kalpan")
```

Output:

```
================ RESTART: E:/LJ/Python/SampleCode/CarDemo.py ================
I am  = Maruti - Owned By -  Hemali
I am  = Santro - Owned By -  Kalpan
>>>
```

Que: Create a class student with following member attributes: roll no, name, age and total marks. Create suitable methods for reading and printing member variables. Write a python program to overload '==' operator to print the details of students having same marks.

File: StudentDemo.py

```
#Create a class student with following member attributes: roll no, name,
#age and total marks. Create suitable methods
#for reading and printing member variables. Write a python program
#to overload '==' operator to print the details of students having same marks.

class Student:
    def __init__(self,name='hemali',rollno=0,age=20,totalmarks = 50):
        self.name = name
        self.rollno = rollno
        self.age = age
        self.totalmarks = totalmarks

    def printdata(self):
        print "Student Detail:"
        print "Name :", self.name
        print "department :", self.rollno
        print "age :", self.age
        print "totalmarks :", self.totalmarks

    def getdata(self):
        self.name = raw_input('Enter name:')
        self.rollno = raw_input('Enter rollno:')
        self.age = raw_input('Enter age:')
        self.totalmarks = raw_input('Enter totalmarks:')

    def __eq__(self, ostudent):
        print "comapring toatl marks of ", self.name , " To ", ostudent.name
        return self.totalmarks == ostudent.totalmarks

print "Student - 1"
s1 = Student(rollno=1)
```

```
s1.printdata()
print "Student - 2"
s2 = Student('Kalpan',2,21)
s2.printdata()
print "Student - 3"
s3 = Student()
s3.getdata()
s3.printdata()

print "s1 == s2", (s1==s2)
print "s2 == s3", (s2==s3)
print "s3 == s1", (s3==s1)
print "s1 == s1", (s1==s1)
```

Output:

```
============== RESTART: E:/LJ/Python/SampleCode/StudentDemo.py ==============
Student - 1
Student Detail:
Name : hemali
department : 1
age : 20
totalmarks : 50
Student - 2
Student Detail:
Name : Kalpan
department : 2
age : 21
totalmarks : 50
Student - 3
Enter name:Aarav
Enter rollno:3
Enter age:5
Enter totalmarks:40
Student Detail:
Name : Aarav
department : 3
age : 5
totalmarks : 40
s1 == s2 comapring toatl marks of  hemali  To  Kalpan
True
s2 == s3 comapring toatl marks of  Kalpan  To  Aarav
False
s3 == s1 comapring toatl marks of  Aarav  To  hemali
False
s1 == s1 comapring toatl marks of  hemali  To  hemali
True
>>>
```

Que: Write a Python program to overload + operator. [demo of all operator overload]

File:OperatorOverloadDemo.py

```
import math

class Circle:

    def __init__(self, radius):
        self.__radius = radius

    def setRadius(self, radius):
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def area(self):
        return math.pi * self.__radius ** 2

    def __add__(self, another_circle):
        return Circle( self.__radius + another_circle.__radius )

    def __gt__(self, another_circle):
        return self.__radius > another_circle.__radius

    def __lt__(self, another_circle):
        return self.__radius < another_circle.__radius

    def __str__(self):
        return "Circle with radius " + str(self.__radius)

c1 = Circle(4)
print(c1.getRadius())

c2 = Circle(5)
print(c2.getRadius())

c3 = c1 + c2
print(c3.getRadius())

print( c3 > c2) # Became possible because we have added __gt__ method

print( c1 < c2) # Became possible because we have added __lt__ method

print(c3) # Became possible because we have added __str__ method
```

Output:

```
>>>
========== RESTART: E:/LJ/Python/SampleCode/OperatorOverloadDemo.py ==========
4
5
9
True
True
Circle with radius 9
>>>
```