Python Notes

Unit-1

Classes and objects

In Procedure-oriented programming, we program using functions. We can do procedural programming in C, C++ or Python.

In object-oriented programming (OOP), we program using classes and objects. C++, Java and Python are object-oriented programming languages.

Object

An object is a collection of data and the methods that utilize data. The data value that you store inside an object is called an attribute while the functions associated with it are called methods.

An object can refer to anything that could be named such as functions, integers, strings, floats, classes, methods, and files.

In fact, everything in Python is an object.

Objects are flexible structures that can be used in different ways. They can be assigned to variables, dictionaries, lists, tuples, or sets. They can be passed as arguments.

Class

A class is a data type just like a list, string, dictionary, float, or an integer. When you create an object out of the class data type, the object is called an instance of a class.

When we say everything in python is an object, it includes classes and types as well. Both classes and types belong to the data type ‘type’.

Defining a class and creating an object

To define a class, you will use the keyword class followed by a class_name and a colon.

To create an object write: object_name = class_name ( )

Creating an object is also called instantiating a class.

```
class Myclass:
    x=13
```

```
obj = Myclass()    # Creating an object
```

```
print(obj.x)       #dot operator to access class member
```

```
13
```

As we discussed earlier, both classes and types belong to the data type ‘type’.

```
type(Myclass)
```

```
type
```

```
type(int)
```

```
type
```

```
x = 7
type(x)
```

```
int
```

Another Example: Class having attributes (data value) and method (function).

```
class Anyclass:
    x = 5
    y = "Greater Noida"
    def display(self):
        print(self.x)
        print(self.y)
        print("hello")
```

x and y are attributes.

display( ) is a method.

```
o1 = Anyclass()
```

```
o1.display()
```

```
5
Greater Noida
hello
```

Another Example:

```
class student:
    rollno = 15
    name = "Amit"
    def show(self):
        print("Roll Number is", self.rollno)
        print("Name is ", self.name)
```

```
s1 = student()
```

```
s1.show()
```

```
Roll Number is 15
Name is  Amit
```

You probably notice the parameter **'self'** in function definition inside classes. In Python, when an object calls its method, the object automatically becomes the first argument. By convention, it is called 'self'. You can use any name but for uniformity, it is best to stick with the convention. If you need to place more arguments, you have to place 'self' as the first argument.

The self Parameter or argument:

- The self argument refers to the object itself.
- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.
- __init__() can have any number of parameters, but the first parameter will always be a variable called *self.* When a new class instance is created, the instance is automatically passed to the self parameter in .__init__() so that new **attributes** can be defined on the object.

The __init__() method

The __init__() method is a class **constructor**. __init__() method executed automatically whenever a new object is created and is used to initialize the object being created. The method takes at least one argument, the 'self', to identify each object being created.

Types of constructors :

- **default constructor :**The default constructor is simple constructor which doesn't accept any arguments. It's definition has only one argument -'self'.
- **parameterized constructor :**constructor with parameters is known as parameterized constructor. The first argument is mandatorily 'self'.

Example: default constructor

```
class student:
    def __init__(self):
        self.rollno = 15
        self.name = "Amit"
    def show(self):
        print("Roll Number is", self.rollno)
        print("Name is ", self.name)
```

```
s1 = student()
s1.show()
```

```
Roll Number is 15
Name is  Amit
```

Example: parameterized constructor

```
class student:
    def __init__(self, rollno, name):
        self.rollno = rollno
        self.name = name
    def show(self):
        print("Roll Number : ",self.rollno)
        print("Name : ", self.name)
```

```
s1 = student(15,"Abhishek")
s2 = student(24,"Nikita")
```

```
s1.show()
```

```
Roll Number :  15
Name :  Abhishek
```

```
s2.show()
```

```
Roll Number :  24
Name :  Nikita
```

Class variable and Instance variable:

Object-oriented programming allows for variables to be used at the class level or the instance level.

Class Variables

- Declared inside the class definition (but outside any of the instance methods).
- They are not tied to any particular object of the class, hence shared across all the objects of the class.
- Modifying a class variable affects all objects instance at the same time.
- Class variables are also called **static variable**.

Instance Variable or object variable —

- Object variable or instance variable owned by each object created for a class.
- Declared inside the constructor method of class (the __init__ method).
- They are tied to the particular object instance of the class, hence the contents of an instance variable are completely independent from one object instance to the other.
- The object variable is not shared between objects.

```
class student:
    college = "NIET"                #class variable
    def __init__(self,name,roll):
        self.name = name            #Instance variables
        self.rollno = roll          #Instance variables

    def show(self):
        print("College: ",self.college)
        print("Name: ",self.name)
        print("Roll Number: ",self.rollno)
```

```
s4 = student("Atul",16)
```

```
print(s4.name)
```

```
Atul
```

```
print(s4.rollno)
```

```
16
```

```
print(student.college)  # class variable can be accessed using class
print(s4.college)       # and object variable
```

```
NIET
NIET
```

```
s4.name = "Atool"
```

```
s4.show()
```

```
College:  NIET
Name:  Atool
Roll Number:  16
```

Another Example: Class or static variable

```
class trial:
    x = 9        #class or static variable

print(trial.x)   # x is 9

obj = trial()
print(obj.x)     #x is 9

obj.x = 12
print(obj.x)     # x is 12
print(trial.x)   # x is 9
```

```
9
9
12
9
```

__del__( ) method

__del__( ) method is called destructor. It is opposite to __init__( ) method. It is called automatically, when an object of the class is deleted. An object is deleted when it is no longer required or we can explicitly delete an object by using **del** keyword.

```
class student:

    def __init__(self,name,roll):
        self.name = name
        self.rollno = roll
    def show(self):
        print("Name: ",self.name)
        print("Roll Number: ",self.rollno)
    def __del__(self):
        print("Object is deleted succcessfullly")
```

```
s1 = student("Manoj",29)
```

```
del s1      # when an object is deleted __del__() is called.
```

```
Object is deleted succcessfullly
```

```
s1.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-61f9a117e707> in <module>
----> 1 s1.show()

NameError: name 's1' is not defined
```

```
s2 = student("Sheetal",37)
```

```
s2 = student("Shivangi",39)  #previous s2 will be deleted automatically when a new s2 is created
                               # and hence __del__() will get invoked automatically
```

```
Object is deleted succcessfullly
```

Garbage Collection

Python's memory allocation and de-allocation method is automatic. Python deletes unwanted objects (built-in types or class instances) automatically to free the memory space.

The process, by which Python periodically frees and reclaims blocks of memory that are no longer required, is called Garbage Collection.

The biggest benefit of garbage collection is that it automatically handles the deletion of unused objects or some objects that are inaccessible to free up memory resources.

Garbage collection frees the programmer from manually dealing with memory deallocation.

Automatic collection can be disabled by calling *gc.disable()*. (disable is function of gc module).

Access Modifiers (aka Access Specifiers) in Python: Public, Private and Protected

Access modifiers are used to restrict access to the variables and methods of the class.

Access modifiers in Python have an important role to play in securing data from unauthorized access and in preventing it from being exploited.

Most programming languages has following three forms of access modifiers in a class:

- Public
- Private
- Protected

Public Access Modifier:

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

When you declare a method (function) or a property (variable) as public, those methods and properties can be accessed by:

- The same class that declared it.
- The classes that inherit the above declared class.
- Any foreign elements outside this class can also access those things.

Private Access Modifier:

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class.

Protected Access Modifier:

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.

When you declare a method (function) or a property (variable) as protected, those methods and properties can be accessed by

- The same class that declared it.
- The classes that inherit the above declared class.

Comparison

Access Modifiers	**Declaration**	**Access from own class**	**Access from derived class**	**Accessible from object**
Public	Name	Yes	Yes	Yes
Private	_ _name	Yes	No	No
Protected	_name	Yes	Yes	No

Example: Program to illustrate the use of Public and Private access modifiers

(Note: Use of Protected access modifier will be discussed later with Inheritance)

```
class employee:
    def __init__(self,name,sal):
        self.name = name          #name is public variable/data/attributes
        self.__salary = sal       #__salary is private variable
        self.gradepoint = 0       #gradepoint is public variable
    def display(self):           #dispaly is public method
        print("Name: ", self.name)
        print("salary: ",self.__salary)
    def __appraisal(self, point):    #this is private method
        self.gradepoint += point
```

```
e1 = employee("Abhijeet",40000)
```

```
e1.display()
```

```
Name:  Abhijeet
salary:  40000
```

```
print(e1.name)
print(e1.__salary)       #this will show error as __salary is private
```

```
Abhijeet
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-a6e8e882229c> in <module>
      1 print(e1.name)
----> 2 print(e1.__salary)

AttributeError: 'employee' object has no attribute '__salary'
```

```
e1.__appraisal(5)              #This will give error
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-8-9f84d5a3dade> in <module>
----> 1 e1.__appraisal(5)

AttributeError: 'employee' object has no attribute '__appraisal'
```

```
e1._employee__appraisal(5)     # Syntax is objectName._className__private method
```

```
print(e1.gradepoint)
```

```
5
```

Decorators:

Before discussing decorators in detail let us understand some concepts that will come in handy in learning the decorators.

First Class Objects

A first-class object is an entity within a programming language that can:

- Appear in an expression
- Be assigned to a variable
- Be used as an argument
- Be returned by another function

In the procedural programming, primitive data types, such as integers and strings are first-class objects. In the functional programming functions are first-class objects.

Example:

- Function can appear in an expression

```
def area(l,b):
    return l*b

volume = area(3,4) * 5
print(volume)
```

```
60
```

- Functions can be assigned to a variable

```
var = print
```

```
var("Hello")
```

```
Hello
```

```
var(5+7)
```

```
12
```

```
type(var)
```

```
builtin_function_or_method
```

- Functions can be used as an argument

```
def fun1(fun):
    fun()

def fun2():
    print("Hello NIET")

fun1(fun2)
```

```
Hello NIET
```

- Functions can be returned by another function

```
def add1(x):
    def add2(y):
        return x+y
    return add2

sum = add1(10)        # here the value of sum is add2, as add1 returns add2
result = sum(20)
print("Result = ",result)
```

```
Result =  30
```

Decorator

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class.

Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

Example:

```
def fun1(fun):
    def fun2():
        print("********")
        fun()
        print("********")
    return fun2

@fun1                          #decorator
def display():
    print("Hello NIET")


display()
```

```
********
Hello NIET
********
```

Another Example:

```
def decor(fun):
    def wrapper(*args, **kwargs):
        print("Starting function execution")
        fun(*args, **kwargs)
        print("Ending function execution")
    return wrapper

# @decor
# def display():
#     print("Hello NIET")

# @decor
# def show(x):
#     print("Hello ",x)
# @decor
# def add(x,y,z):
#     print("Sum = ",x+y+z)
@decor
def area(r):
    print("Area = ",3.14*r*r)


radius = float(input("Enter radius of the circle: "))
area(radius)
```

```
Enter radius of the circle: 5
Starting function execution
Area =  78.5
Ending function execution
```

Note: Due to *args and **kwargs you can use same decorator with function having any number of arguments.

*args = variable length arguments

**kwargs = keyword arguments

Example: (to understand *args and **kwargs)

```
def fun3(*args, **kwargs):
    print("Arguments:",args)
    print("Keyword Arguments: ",kwargs)

fun3(4,5,6,"Hello",52.6,Name = "Alok", College = "NIET")
```

```
Arguments: (4, 5, 6, 'Hello', 52.6)
Keyword Arguments:  {'Name': 'Alok', 'College': 'NIET'}
```

Instance Method, Class Method and Static Method

Instance Methods

They are most widely used methods. Instance method receives the instance of the class **(self)** as the first argument.

Instance methods can change the object state as well as the class state.

Class Methods

A class method accepts the class (**cls**) as an argument. It is declared with the **@classmethod** decorator.

Class methods are bound to the class and not to the object of the class.

Class methods can change the class state but not the object state.

Static Methods

A static method is just defined as a regular method marked with a **@staticmethod** decorator. It does not take any specific first argument (neither self nor cls).

Static methods can neither modify the object state nor class state.

It is just a way to namespace your methods. We generally use static methods to create utility functions.

Example: Syntax

```
class MyClass:
    def instancemethod(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Another Example:

```
class student:
    university = "AKTU"                #class variable or static variable
    def __init__(self):
        self.name = " "            #y and z are instance variables
        self.marks = 0
    def setvalue(self):
        self.name = input("Enter Name: ")
        self.marks = int(input("Enter Marks: "))

    def display(self):
        print("Name: ",self.name)
        print("Marks: ",self.marks)
        print("University: ",self.university)

    @classmethod
    def changeUniversity(cls):
        cls.university = "APJ Abdul Kalam Technichal University"


    @staticmethod
    def isHonours(marks):
        if marks>= 75:
            return True
        else:
            return False
```

```
s1 = student()
s1.display()
s1.setvalue()
s1.display()
student.changeUniversity()
s1.display()
s2 = student()
s2.display()
```

```
Name:
Marks:  0
University:  AKTU
Enter Name: Amit Kumar
Enter Marks: 86
Name:  Amit Kumar
Marks:  86
University:  AKTU
Name:  Amit Kumar
Marks:  86
University:  APJ Abdul Kalam Technichal University
Name:
Marks:  0
University:  APJ Abdul Kalam Technichal University
```

Instance Method	Class Method	Static Method
The Instance method takes self (object) as first argument.	The class method takes cls (class) as first argument.	The static method does not take any specific parameter.
Instance method can access and modify the class state as well as object state.	Class method can access and modify the class state.	Static Method cannot access or modify the class state.
No default decorator is used. User defined decorator can be used.	@classmethod decorator is used here.	@staticmethod decorator is used here.
Instance method takes the	The class method takes the	Static methods do not know

self as parameter to know about the state of that object as well as class state, if any.	class as parameter to know about the state of that class.	about class state. These methods are used to do some utility tasks by taking some parameters.

Magic methods

Magic methods in Python are the special methods that start and end with the double underscores. e.g. __init__()

They are also called **dunder methods** (*dunder* here means “Double Underscores)”. Magic methods are not meant to be invoked directly by you, but they are inviked by an object. e.g. *num.__add__(5).*

For example, when you add two numbers using the + operator, internally, the __add__() method will be called.

Built-in classes in Python define many magic methods. Use the dir() function to see the number of magic methods inherited by a class.

For example, the following lists all the attributes and methods defined in the int class.

>>> dir(int)

['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']

Examples:

```
num = 10
num.__add__(6)
```

```
16
```

```
num = 14
num.__floordiv__(3)
```

```
4
```

```
num = 20
num.__divmod__(3)   #gives (Quotient ,Remainder)
```

```
(6, 2)
```

__new__() method

Languages such as Java and C# use the new operator to create a new instance of a class.

In Python the __new__() magic method is implicitly called before the __init__() method. The __new__() method returns a new object, which is then initialized by __init__().

```
class Person:

    def __new__(cls):
        print ("__new__ magic method is called")
        inst = object.__new__(cls)
        return inst

    def __init__(self):
        print ("__init__ magic method is called")
        self.name= "Amit"

obj = Person()
```

```
__new__ magic method is called
__init__ magic method is called
```

__str__() and __repr__() methods

__str__ method can be overridden to return a printable string representation of any user defined class.

If we inspect our person object in interpreter session, we still got the <__main__.Person at 0x1af3758e708> output

__repr__ is invoked, when object is inspected in interpreter session. On a high level, __str__ is used for creating output for end user while __repr__ is mainly used for debugging and development..

See Example

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

obj = Person("Amit",23)
print(obj.name,obj.age)     # works fine
print(obj)                  # see the output???
```

```
Amit 23
<__main__.Person object at 0x000001AF37585DC8>
```

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return "Name: {}, Age: {}".format(self.name, self.age)

obj = Person('Sarah', 25)
print(obj)                  # print(obj) works fine with __str__
obj                         # This is printing object id
```

```
Name: Sarah, Age: 25

<__main__.Person at 0x1af3758e708>
```

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        print('inside str')
        return "Person: {}, Age: {}".format(self.name, self.age)
    def __repr__(self):
        print('inside repr')
        return " Person: {}, Age: {}".format(self.name, self.age)

obj = Person("Mohit",24)
print(obj)            # __str__ will be called implicitly
obj                   #__repr__ will be called implicitly
```

```
inside str
Person: Mohit, Age: 24
inside repr

 Person: Mohit, Age: 24
```

Some built-in functions

1. **hasattr(obj,attr):** to check if the object has the atrribute present or not. It returns true if an object has the given attribute and false if it does not.
2. **getattr(object, attribute_name [, default]):** getattr() function is to get the value of an object's attribute and if no attribute of that object is found, default value is returned.
3. **setattr(object, name, value):** to set the attribute values of an object, setattr().
4. **delattr(class_name, name):** deletes an attribute from the object (if allowed).

See Examples

```
class student:
    university = "AKTU"                #class variable or static variable
    def __init__(self):
        self.name = " "
        self.marks = 0

s3 = student()
```

```
hasattr(s3,'name')
```

```
True
```

```
hasattr(s3,'university')
```

```
True
```

```
hasattr(student,'name')
```

```
False
```

```
hasattr(student,'university')
```

```
True
```

```
getattr(student,'university','NO such attribute exist')
```

```
'AKTU'
```

```
getattr(student,'city',"NO such atttribute exist")
```

```
'NO such atttribute exist'
```

```
setattr(student,'university','AKTU Lucknow')
```

```
getattr(student,'university','NO such attribute exist')
```

```
'AKTU Lucknow'
```

```
setattr(student,'country',"India")
```

```
print(s3.country)
```

```
India
```

```
getattr(student,'country',"NO such atttribute exist")
```

```
'India'
```

```
delattr(student,'university')
```

```
getattr(student,'university',"NO such atttribute exist")
```

```
'NO such atttribute exist'
```

__name__ and ” __main__”

Before executing code, Python interpreter reads source file and define few special variables/global variables.

If the python interpreter is running that module (the source file) as the main program, it sets the special __name__ variable to have a value “__main__”. If this file is being imported from another module, __name__ will be set to the module’s name. Module’s name is available as value to __name__ global variable.

```
# This code is written in module called "aimlcc.py"

def delhi():
    print("This is AIMLCC file")

    if __name__ == "__main__":
        print("I am main file and my name is :",__name__)

    else:
        print(" I am Imported file not the main file and my name is :",__name__)

delhi()
```

```
This is AIMLCC file
I am main file and my name is : __main__
```

This is another file(__main__) that imports above module:

```
import aimlcc
```

```
aimlcc.delhi()
```

```
This is AIMLCC file
 I am Imported file not the main file and my name is : aimlcc
```

```
print(__name__)
```

```
__main__
```

Namespaces

A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary in which the keys are the object names and the values are the objects themselves. Each key-value pair maps a name to its corresponding object.

The lifetime of a namespace :

A lifetime of a namespace depends upon the scope of objects, if the scope of an object ends, the lifetime of that namespace comes to an end. Hence, it is not possible to access the inner namespace's objects from an outer namespace.

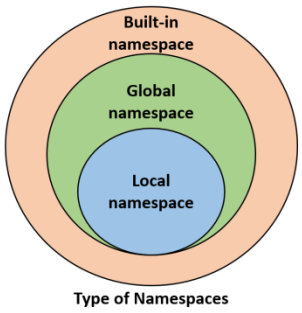
Scope of Objects in Python :

Scope refers to the coding region from which a particular Python object is accessible. Hence one cannot access any particular object from anywhere from the code, the accessing has to be allowed by the scope of the object.

In a Python program, there are three types of namespaces:

1. Built-In
2. Global
3. Local

These have differing lifetimes. As Python executes a program, it creates namespaces as necessary and deletes them when they're no longer needed. Typically, many namespaces will exist at any given time.



Type of Namespaces

1. The Built-In Namespace

- The built-in namespace contains the names of all of Python's built-in objects. These are available at all times when Python is running.
- *for example*, the StopIteration exception,
  built-in functions like min(), max(), print(), range, and len(),
  and object types like int, float and str are built-in namespaces.
- The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates.
- You can list the objects in the built-in namespace with the following command:

```
dir(__builtins__)
```

2. The Global Namespace

The global namespace contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

The interpreter also creates a global namespace for any **module** that your program loads with the import statement.

3. The Local and Enclosing Namespaces

The interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.

```
x = 10              #global namespace
m = 15
def fun():
    x=20            #local namespace
    y=25
    def fun1():
        x=30         # nested local namespace
        z=35
        print("Inside fun1: x={},y={},z={},m={}".format(x, y, z, m))
    fun1()
    print("Inside fun: x={},y={},m={} ".format(x, y, m))
fun()
print("Outside: x={},m={} ".format(x, m))
```

```
Inside fun1: x=30,y=25,z=35,m=15
Inside fun: x=20,y=25,m=15
Outside: x=10,m=15
```

Passing Object of The Class as Parameter

Below mentioned code how to pass object of the class as parameter in Python.

First make a class MyClass in the module named MyModule.

```
class MyClass():

    def my_method(self, obj):
        print("Name:", obj.name)
        print("Age:", obj.age)
```

Now, import MyClass from the module MyModule and use the my_method of MyClass to print the value of an object of class person by passing object of class Person in my_method of MyClass as parameter.

```
from MyModule import MyClass

class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person('John', 40)
obj = MyClass()
obj.my_method(person)           # passing object as an argument
```

```
Name: John
Age: 40
```

Returning object from a method

Following program uses method that returns an object from a function.

Write a program to find mid-point of line joining two points.

```
class Point:

    def __init__(self, X, Y):
        self.x = X
        self.y = Y

    def midpoint(self, target):
        mx = (self.x + target.x) / 2
        my = (self.y + target.y) / 2
        return Point(mx, my)                # returning object

    def display_point(self):
        print("Point is: ({}, {})".format(self.x,self.y))

p = Point(3, 4)
q = Point(5, 12)
mid = p.midpoint(q)                         # passing object q to method midpoint

mid.display_point()
```

```
Point is: (4.0, 8.0)
```

Solved Problems:

1. Write a program that uses class to store the name and marks of students. Use list to store the marks in three subjects.

```
class student:
    def __init__(self):
        self.name = ""
        self.marks = []

    def getdata(self):
        self.name = input("Enter Name: ")
        for i in range(3):
            x = int(input("Enter Marks: "))
            self.marks.append(x)

    def display(self):
        print("Name: ",self.name)
        print("Marks: ",self.marks)
```

2. Define a class that stores a string and create methods to count number of uppercase characters, vowels, consonants and spaces.

```
class String:
    def __init__(self):
        self.string = input("Enter String: ")
        self.uppercase = 0
        self.vowels = 0
        self.consonants = 0
        self.spaces = 0

    def count_uppercase(self):
        for x in self.string:
            if(x.isupper()):
                self.uppercase += 1

    def count_vowels(self):
        for x in self.string:
            if(x in ('a','e','i','o','u','A','E','I','O','U')):
                self.vowels += 1

    def count_consonants(self):
        for x in self.string:
            if(x in ('a','e','i','o','u','A','E','I','O','U')):
                self.consonants += 1

    def count_spaces(self):
        for x in self.string:
            if(x == " "):
                self.spaces += 1
```

```
    def compute_stat(self):
        self.count_uppercase()
        self.count_vowels()
        self.count_consonants()
        self.count_spaces()

    def show_stat(self):
        print("Upprcase: ",self.uppercase)
        print("Vowels: ",self.vowels)
        print("Consonants: ",self.consonants)
        print("Spaces: ",self.spaces)

s = String()
s.compute_stat()
s.show_stat()
```

```
Enter String: We are Indians
Upprcase:  2
Vowels:  6
Consonants:  6
Spaces:  2
```