Module-4

OBJECT-ORIENTED PROGRAMMING

Topics

Classes and Objects

Creating Classes in Python

Creating Objects in Python

The Constructor Method

Classes with Multiple Objects

Class Attributes versus Data Attributes

Encapsulation

Inheritance

The Polymorphism.

Classes and Objects

- In python, everything is an object
- We can write a class to represent properties (attributes) and actions (behaviour) of object. Properties can be represented by variables, Actions can be represented by Methods.
- Ex:

```
l=[1,2,3,4]
l.append(5)
print(l)
print(type(l))
```

```
[1, 2, 3, 4, 5]
<class 'list'>
```

- student info
  - Class
  - Object
  - Reference variable

Class, Object, Reference variable

- **Class** is a blueprint to construct objects, it defines all properties and operations of object

- **Object** is physical existence of a class or real world entity. Memory will be allocated for object. Per class any number of objects can be created

  referencevariable = classname()

- **Reference variable** is pointing to our object, by using it we can perform operations on the object.

- In this example, 'l' is

  reference variable

```
l=[1,2,3,4]
l.append(5)
print(l)
print(type(l))
```

```
[1, 2, 3, 4, 5]
<class 'list'>
```

Creating Classes in Python

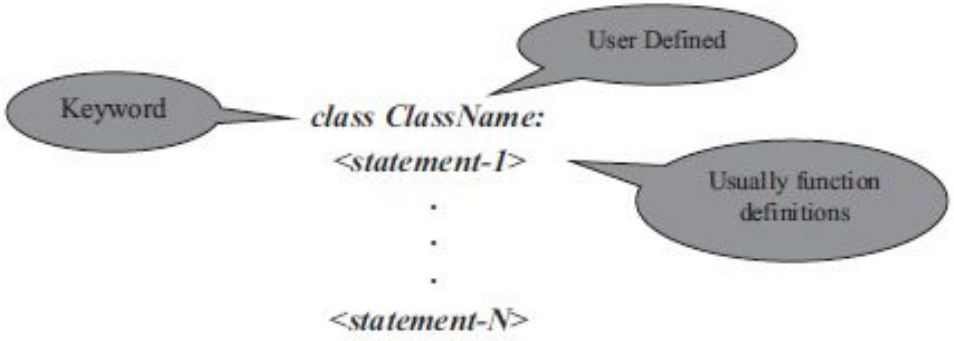
- Class contains both variables , methods

class className:

''' documenttation string '''

variables: instance variables, static and local variables(properties)

methods: instance methods, static methods, class methods(actions/ behaviour)



Class

- class classname:

  ''' docstring'''

  Variables(properties)

  Methods(actions/behaviour)

Variables:

- Instance variables
- Static variables
- Local variables

Methods:

- Instance methods
- Static methods
- Class methods

Creating Objects in Python

- Object refers to a particular instance of a class where the object contains variables and methods defined in the class.
- Class objects support two kinds of operations:
  - attribute references and
  - instantiation.
  - The term attribute refers to any name (variables or methods) following a dot. This is a syntactic construct.
  - The act of creating an object from a class is called *instantiation*.

    The names in a class are referenced by objects and are called *attribute references*.

Creating Objects in Python

There are two kinds of attribute references,

- data attributes and
- method attributes.
- Variables defined within the methods are called *instance variables* and are used to store data values.
- New instance variables are associated with each of the objects that are created for a class. These instance variables are also called *data attributes*.
- Method attributes are methods inside a class and are referenced by objects of a class. Attribute references use the standard dot

notation syntax as supported in Python.

Example

```
class student:
    '''student information'''
    def __init__(self):
        self.name='Manu'
        self.age=40
        self.marks=80
    def studinfo(self):
        print("Hello I am :",self.name)
        print("My Age is:",self.age)
        print("My Marks are:",self.marks)
s=student()
s.studinfo()
#print(s.name)
```

Example

```
class Student:
    def __init__(self,Name,USN,CIE):
        self.name=Name
        self.rollno=USN
        self.marks=CIE

    def studinfo(self):
        print("Student Name is:",self.name)
        print("Student USN is:",self.rollno)
        print("Student Marks are:",self.marks)

s1=Student("Manu",56,45)
s1.studinfo()
s2=Student("vinu",61,24)
s2.studinfo()
s3=Student("vani",108,48)
s3.studinfo()
s4=Student("Manu",150,30)
s4.studinfo()
```

Self:

- Within the class to refer current object, python provides an implicit variable which is ‘self’
- Self variable always pointing to current object
- Self variable is the first argument for constructor and instance methods
- By using self we can declare instance variables, we can access instance variables and instance methods of object
- We can use self within the class only and from outside of class we cannot use
- At the time of calling constructor and instance methods we are not required to provide value for self variable. PVM itself is responsible to provide value
- Instead of ‘self’ we can use any name , but recommended to use self

Constructor Concept:

- Constructor is a special method in python.
- The name of the constructor should be __init__(self)
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize instance variables.
- Per object, constructor will be executed only once.
- Constructor can take at least one argument(at least self)
- Constructor is optional and if we are not providing any constructor then python will provide default constructor.

Example:

def __init__(self,name,rollno,marks):

self.name=name

self.rollno=rollno

self.marks=marks

```
def __init__(self,name,rollno,marks):
    self.name=name
    self.rollno=rollno
    self.marks=marks
```

```
def __init__(self,Name,USN,CIE):
    self.name=Name
    self.rollno=USN
    self.marks=CIE
```

Method vs Constructor

Method	Constructor
Name of method can be any name	Constructor name should be always __init__
Method will be executed if we call that method	Constructor will be executed automatically at the time of object creation.
Per object, method can be called any number of times.	Per object, Constructor will be executed only once
Inside method we can write logic	Inside Constructor we have to declare and initialize instance variables

Types of Variables:

Inside Python class 3 types of variables are allowed.

1. Instance Variables (Object Level Variables)

2. Static Variables (Class Level Variables)

3. Local variables (Method Level Variables)

Instance Variables:

If the value of a variable is varied from object to object, then such type of variables are called instance variables.

For every object a separate copy of instance variables will be created.

Where we can declare Instance variables:

1. Inside Constructor by using self variable
2. Inside Instance Method by using self variable
3. Outside of the class by using object reference variable

Inside Constructor by using self variable:

- We can declare instance variables inside a constructor by **using se**lf keyword. Once we creates object, automatically these variables will be added to the object.

```
class Studinfo:
    def __init__(self):
        self.name='Manu'
        self.rollno=150
        self.marks=45

s=Studinfo()
print(s.__dict__)
```

```
{'name': 'Manu', 'rollno': 150, 'marks': 45}
```

Inside Instance Method by using self variable:

We can also declare instance variables inside instance method by using self variable. If any instance variable declared inside instance method, that instance variable will be added once we call that method.

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20

    def m1(self):
        self.c=30


t=Test()
t.m1()
print(t.__dict__)
```

```
{'a': 10, 'b': 20, 'c': 30}
```

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20

    def m1(self):
        self.c=30


t=Test()
#t.m1()
print(t.__dict__)
```

```
{'a': 10, 'b': 20}
```

Outside of the class by using object reference variable:

- We can also add instance variables outside of a class to a particular object.

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
    def m1(self):
        self.c=30

t=Test()
t.m1()
t.d=40
print(t.__dict__)
```

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

How to access Instance variables:

We can access instance variables with in the class by using **self variable** and outside of the class by using object reference.

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
    def display(self):
        print(self.a)
        print(self.b)

t=Test()
t.display()
print(t.a,t.b)
```

```
10
20
10 20
```

How to delete instance variable from the object:

1. Within a class we can delete instance variable as follows

del self.variableName

2. From outside of class we can delete instance variables as follows

del objectreference.variableName

Example:

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
        self.c=30
        self.d=40
    def m1(self):
        del self.d

t=Test()
print(t.__dict__)
t.m1()
print(t.__dict__)
del t.c
print(t.__dict__)
```

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30}
{'a': 10, 'b': 20}
```

Example:

Note: The instance variables which are deleted from one object, will not be deleted from other objects.

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
        self.c=30
        self.d=40

t1=Test()
t2=Test()
del t1.a
print(t1.__dict__)
print(t2.__dict__)
```

```
{'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

Example

If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20

t1=Test()
t1.a=888
t1.b=999
t2=Test()
print('t1:',t1.a,t1.b)
print('t2:',t2.a,t2.b)
```

```
t1: 888 999
t2: 10 20
```

Static variables:

- If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such type of variables are called Static variables.

- For total class only one copy of static variable will be created and shared by all objects of that class.

- We can access static variables either by class name or by object reference. But recommended to use class name.

- In the case of instance variables for every object a separate copy will be created, but in the case of static variables for total class only one copy will be created and shared by every object of that class.

Various places to declare static variables:

1. In general we can declare within the class directly but from out side of any method

2. Inside constructor by using class name

3. Inside instance method by using class name

4. Inside class method by using either class name or cls variable

5. Inside static method by using class name

6. from outside of class by using classname

How to access static variables:

1. inside constructor: by using either self or classname

2. inside instance method: by using either self or classname

3. inside class method: by using either cls variable or classname

4. inside static method: by using classname

5. From outside of class: by using either object reference or classnmae

Where we can modify the value of static variable:

Anywhere either with in the class or outside of class we can modify by using classname.

But inside class method, by using cls variable.

In general we can declare within the class directly but from out side of any method

```
#Static variable

class studinfo:
    section='6B'
    def __init__(self):
        self.usn=105

t=studinfo()

print('t:',t.section,  t.usn)
#you can use classname or object name to access static varialbe
print('t:',studinfo.section, t.usn)
```

```
t: 6B 105
t: 6B 105
```

Inside constructor by using class name

```
#To use static variable inside the constructor
class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
t=test()
print(test.__dict__)
#only a and c are printed as b is a instance varialbe
```

```
{'__module__': '__main__', 'a': 10, '__init__': <function test.__init__ at 0x00
000000050BEA68>, '__dict__': <attribute '__dict__' of 'test' objects>, '__weakr
ef__': <attribute '__weakref__' of 'test' objects>, '__doc__': None, 'c': 30}
```

Inside instance method by using class name

```
#To use static variable inside the instance method by using class name

class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
    def m1(self):
        self.d=40
        test.e=50

t=test()
t.m1()
print(test.__dict__)
#only a,c, e  are printed as b, d are instance varialbe
```

```
{'__module__': '__main__', 'a': 10, '__init__': <function test.__init__ at 0x00
000000050BE048>, 'm1': <function test.m1 at 0x00000000050BE828>, '__dict__': <a
ttribute '__dict__' of 'test' objects>, '__weakref__': <attribute '__weakref__'
of 'test' objects>, '__doc__': None, 'c': 30, 'e': 50}
```

Inside class method by using either class name or cls variable

```
# sttic varialbe Inside class method by using either class name or cls variable

class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
    def m1(self):
        self.d=40
        test.e=50
    @classmethod
    def m2(cls):
        cls.f=60
        test.g=70
# cls referes to current class object, instead of cls other name canbe used.
t=test()
t.m1()
test.m2()
print(test.__dict__)
```

Inside static method by using class name

```
#Static varialbe inside static method by using class name

class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
    def m1(self):
        self.d=40
        test.e=50

    @classmethod
    def m2(cls):
        cls.f=60
        test.g=70

    @staticmethod
    def m3():
        test.h=80

# if we dont class static method also works
t=test()
t.m1()
#t.m2()
test.m2()
#t.m3()
test.m3()
print(test.__dict__)
```

from outside of class by using classname

```
#Static varialbe outside the class using class name

class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
    def m1(self):
        self.d=40
        test.e=50

    @classmethod
    def m2(cls):
        cls.f=60
        test.g=70

    @staticmethod
    def m3():
        test.h=80

# if we dont class static method also works
t=test()
t.m1()
#t.m2()
test.m2()
#t.m3()
test.m3()
test.i=90
#print(t.a,t.c,t.e,t.f,t.g,t.h,t.i)
print(test.a,test.c,test.e,test.f,test.g,test.h,test.i)
#print(test.__dict__)
```

```
10 30 50 60 70 80 90
```

Static variable

If we change the value of static variable by using either self or object reference variable, then the value of static variable won't be changed, just a new instance variable with that name will be added to that particular object.

How to delete static variables of a class:

We can delete static variables from anywhere by using the following syntax

del classname.variablename

But inside classmethod we can also use cls variable

del cls.variablename

How to access static variables:

```
#How to access static variables:

class Test:
    a=10
    def __init__(self):
        print(self.a)
        print(Test.a)
    def m1(self):
        print(self.a)
        print(Test.a)
    @classmethod
    def m2(cls):
        print(cls.a)
        print(Test.a)
    @staticmethod
    def m3():
        print(Test.a)
#cls: pointing to current class, self: pointing to current object
t=Test()
print(Test.a)
print(t.a)
t.m1()
t.m2()
t.m3()
```

Static variable contd..

By using object reference variable/self we can read static variables, but we cannot modify or delete.

If we are trying to modify, then a new instance variable will be added to that particular object.

t1.a = 70

If we are trying to delete then we will get error.

We can modify or delete static variables only by using classname or cls variable.

Local variables:

Sometimes to meet temporary requirements of programmer, we can declare variables inside a method directly, such type of variables are called local variable or temporary variables.

Local variables will be created at the time of method execution and destroyed once method completes.

Local variables of a method cannot be accessed from outside of method.

```
#Local varialbe
class Test:
    def m1(self):
        a=1000
        print(a)
    def m2(self):
        b=2000
        print(b)
t=Test()
t.m1()
t.m2()
```

```
1000
2000
```

Types of Methods:

Inside Python class 3 types of methods are allowed

1. Instance Methods

2. Class Methods

3. Static Methods

Instance Methods:

Inside method implementation if we are using instance variables then such type of methods are called instance methods.

Inside instance method declaration, we have to pass self variable.

def m1(self):

By using self variable inside method we can able to access instance variables.

Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

Setter and Getter Methods:

We can set and get the values of instance variables by using getter and setter methods.

Setter Method:

setter methods can be used to set values to the instance variables. setter methods also known as mutator methods.

syntax:

def setVariable(self,variable):

- **self.variable=variable**

Example:

def setName(self,name):

- **self.name=name**

Getter Method:

Getter methods can be used to get values of the instance variables. Getter methods also known as accessor methods.

syntax:

```
def getVariable(self):
   return self.variable
```

Example:

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

Class Methods:

Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.

We can declare class method explicitly by using @classmethod decorator.

For class method we should provide cls variable at the time of declaration

We can call classmethod by using classname or object reference variable.

Static Methods:

In general these methods are general utility methods.

Inside these methods we won't use any instance or class variables.

Here we won't provide self or cls arguments at the time of declaration.

We can declare static method explicitly by using @staticmethod decorator

We can access static methods by using classname or object reference

Note: In general we can use only instance and static methods. Inside static method we can access class level variables by using class name.

class methods are most rarely used methods in python.

Destructors:

Destructor is a special method and the name should be __del__

Just before destroying an object Garbage Collector always calls destructor to perform clean up activities (Resource deallocation activities like close database connection etc).

Once destructor execution completed then Garbage Collector automatically destroys that object.

Note: The job of destructor is not to destroy object and it is just to perform clean up activities.

Note: If the object does not contain any reference variable then only it is eligible fo GC. ie if the reference count is zero then only object eligible for GC

Garbage Collection:

In Python, We have some assistant which is always running in the background to destroy useless objects. Because this assistant the chance of failing Python program with memory problems is very less. This assistant is nothing but Garbage Collector.

Hence the main objective of Garbage Collector is to destroy useless objects.

If an object does not have any reference variable then that object eligible for Garbage Collection.

How to enable and disable Garbage Collector in our program:

By default Garbage collector is enabled, but we can disable based on our requirement. In this context we can use the following functions of

gc module.

1. gc.isenabled()

Returns True if GC enabled

2. gc.disable()

To disable GC explicitly

3. gc.enable()

To enable GC explicitly