PYTHON PROGRAMMING - II

Unit - 1

Object Oriented Concepts in Python

By-

Prof. A. P. Chaudhari

(M.Sc. Computer Science, SET)

HOD,

Department of Computer Science

S.V.S's Dadasaheb Rawal College, Dondaicha

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.

Overview of OOP Terminology:

- **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.

Creating Classes:

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A class is a user-defined blueprint from which objects are created. Classes provide a bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

To understand the need for creating a class let’s consider an example, let’s say you wanted to track the number of students that may have different attributes like Name, Age. If a list is used, the first element could be the students’s Name while the second element could represent its Age. Let’s suppose there are 100 different students, then how would you know which element is supposed to be which? What if you wanted to add other properties to these students? This lacks organization and it’s the exact need for classes.

Creating Classes:

Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
        # Statement-1
        . . .
        # Statement-N
```

Creating Instance Objects:

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*. It's not an idea anymore, it's an actual student, like a student is Bhushan who's eighteen years old. You can have many students to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

An object consists of :

State: It is represented by the attributes of an object. It also reflects the properties of an object.

Behavior: It is represented by the methods (functions) of an object. It also reflects the response of an object to other objects.

Identity: It gives a unique name to an object and enables one object to interact with other objects.

Creating Instance Objects:

The syntax to create the instance of the class is given below -

object_name = **class**_name([arguments])

e.g.:

s1 = stud ()

Accessing Attributes:

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows −

s1.name

s1.age

Accessing Attributes:

Now, putting all the concepts together −

```
class stud:
    name = "Bhushan"
    age = 18

s1 = stud()
print 'Name:', s1.name
print 'Age:', s1.age
```

O/P:

```
Name: Bhushan
Age: 18
```

Built-In Class Attributes :

The __init__() Function:

The above examples are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function. The __init__() method is similar to constructors in C++ and Java. All classes have a function called __init__(), which is always executed when the class is being initiated. Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created.

Note: The __init__() function is called automatically every time the class is being used to create a new object.

Built-In Class Attributes :

e.g.:

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

s1 = stud("Bhushan",18)
print 'Name:',s1.name
print 'Age:',s1.age
```

O/P:

```
 Name: Bhushan
Age: 18
```

Built-In Class Attributes :

Object Methods:

Objects can also contain methods. Methods in objects are functions that belong to the object.

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

    def show(self):
        print 'Name:',self.name
        print 'Age:',self.age

s1 = stud("Bhushan",18)
s1.show()
```

Built-In Class Attributes :

The self Parameter: 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.

e.g.:

```
class stud:
    def __init__(me, name, age):
        me.name = name
        me.age = age
    def show(my):
        print 'Name:',my.name
        print 'Age:',my.age

s1 = stud("Bhushan",18)
s1.show()
```

Built-In Class Attributes :

Modify Object Properties:

You can modify properties on objects like this - s1.age = 20

e.g.:

```
class stud:
    def __init__(me, name, age):
        me.name = name
        me.age = age
    def show(my):
        print 'Name:',my.name
        print 'Age:',my.age
s1 = stud("Bhushan",18)
s1.show()
s1.age = 20
print 'After age changes:'
s1.show()
```

Built-In Class Attributes :

Delete Objects:

You can delete objects by using the del keyword –

```
del s1
```

e.g.:

```
class stud:
    name = "Bhushan"
    age = 18

s1 = stud()
print 'Name:',s1.name
del s1
print 'Age:',s1.age
```

O/P:

```
Name: Bhushan
Age:

Traceback (most recent call last):
  File "C:/Python27/s1.py", line 8, in <module>
    print 'Age:',s1.age
NameError: name 's1' is not defined
```

Built-In Class Attributes :

The pass Statement:

Class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the **pass statement** to avoid getting an error.

e.g.:

```
class stud:
    pass
```

O/P:

Garbage Collection:

Python memory management is straight forward. You don’t need to worry about memory management, as memory allocation and deallocation is automatic. one of the mechanisms of memory management is garbage collection. Let’s understand different aspects of garbage collection,

Garbage collection:

It is the process by which shared computer memory is cleaned which is currently being put to use by a running program when that program no longer needs that memory. With garbage collection, that freed memory can be used by another program.

Garbage Collection:

There are two methods used by python for memory management −

- Reference counting
- Garbage collection

Python's garbage collection is automatic but in some programming languages, you have to clean objects yourself. In python, if you want you can delete objects manually.

e.g.:

```
x = 10
print x

del x
print x
```

O/P:

```
10


Traceback (most recent call last):
  File "C:/Python27/d1.py", line 5, in <module>
    print x
NameError: name 'x' is not defined
```

Garbage Collection:

Above we simply define one variable(x) and use it. During runtime, we delete the object(because everything in python is an object) and try to output it.

In the first two lines of the above program, object x is known. However, after the deletion of the object(x), we cannot print it anymore.

So from above, we can see that garbage collection is fully automated and we don’t need to worry about it.

Let’s understand the above concept with another example. Every object in python like in the above code, object x has a reference count and a pointer to a type. The reference count changes value depends on how we are using it, for example, if we assign the object x to another object y, its reference count increases to 2.

Garbage Collection:

```
some_list = [1, 2 ,3, 4, 5, 6, 7, 8, 9]          #Reference count of some_list = 1
other_list = some_list                            #Reference count = 2
list_total = sum(some_list)                       #Reference count = 3
# If we put the object in a list, reference count will also increase
list_of_list = [some_list, some_list, some_list]
#Let's check the reference count of object "some_list"
import sys
print sys.getrefcount(some_list)                  O/P: 6
del list_of_list
print sys.getrefcount(some_list)                  O/P: 3
del other_list
print sys.getrefcount(some_list)                  O/P: 2
del some_list
print sys.getrefcount(some_list)      O/P: NameError: name 'some_list' is not defined
```

Garbage Collection:

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

e.g.:

```
class Point:
   def __init__( self, x=0, y=0):
      self.x = x
      self.y = y

   def __del__(self):
       print 'Destructor calling'
       class_name = self.__class__.__name__
       print class_name, "destroyed"
```

```
pt1 = Point()
pt2 = pt1
# prints the ids of the obejcts
print id(pt1), id(pt2)
del pt1
del pt2
```

O/P:

```
33297872 33297872
Destructor calling
Point destroyed
```

Method Overloading:

Method overloading is one concept of Polymorphism. It comes under the elements of OOPS. It is actually a compile-time polymorphism. It is worked in the same method names and different arguments. Here in Python also supports oops concepts. But it is not oops based language. It also supports this method overloading also.

Normally in python, we don’t have the same names for different methods. But overloading is a method to do the different functionalities with the same name. i.e methods different their parameters to pass to the methods.

Method Overloading:

Here some **advantages** of Method Overloading in Python -

- Normally methods are used to reduce complexity in the program also improves the clarity of code.
- It is also used for reusability.

It has some **disadvantages** also when creating more confusion during the inheritance concepts.

In python, we create a single method with different arguments to process the Method Overloading. Here we create a method with zero or more parameters and also define the methods based on the number of parameters.

Method Overloading:

e.g.:

```
class Employee:
    def Hello_Emp(self,e_name=None):
        if e_name is not None:
            print("Hello "+e_name)
        else:
            print("Hello ")


emp1=Employee()
emp1.Hello_Emp()
emp1.Hello_Emp("Darshan")
```

O/P:

```
Hello
Hello Darshan
```

Method Overloading:

In this example class called Employee having a method called Hello_Emp(). This method is used as a method overloading concept. It will be done by using the parameters. Here we use the parameter called e_name is set as None. If I call this method using objects during that time if I pass the value to the parameter means it prints the value. Otherwise, it is not print anything. So here we have two functionalities that will be executed in this program.

In the above output, we first call the method Hello_Emp without any parameter so that it prints only Hello.

Next time we call the same method with parameter value as Darshan so it will prints Hello Darshan.

Method Overloading:

```
class Area:
    def find_area(self,a=None,b=None):
        if a!=None and b!=None:
            print "Area of Rectangle:",(a*b)
        elif a!=None:
            print "Area of square:",(a*a)
        else:
            print "Nothing to find"


obj1=Area()
obj1.find_area()
obj1.find_area(10)
obj1.find_area(10,20)
```

O/P:

Nothing to find

Area of square: 100

Area of Rectangle: 200

Method Overloading:

In this example also we implement the Method Overloading concept with help of Parameters. In this example use to find the area of different shapes. If you get the area of square means you just pass the value for the parameter itself. In the same method if you want to find a rectangle area means you pass the value for both a and b values. If you pass without any value means it will return nothing to find.

Operator Overloading:

Programmers can use pre-defined operators like +, =, *, >, <, etc. on built-in data types to write programs. However, these operators fail to work in user-defined data types. Therefore, Python comes up with operator loading capability, allowing programmers to redefine operators when working on class objects.

Operator overloading allows programmers to extend the meaning of pre-defined operators. Simply put, it provides an expanded definition over what is pre-defined, making it easier for programmers to work with both basic data types and user-defined data types.

For example, operator ‘+’ will add two numbers, either by adding two ranges or combining two lists. You can do the same by overloading the ‘+’ operator with the int and str class.

Operator Overloading:

Magic methods in Python are special methods that begin and end with a **double underscore(__)**. The **__init__()** is one such method.

Another magic method is the **__str__()** method. The **__str__()** method lets you control how an object of your class gets **printed**.

e.g:

```
class sample1:
    def __init__(self, volume):
        self.volume = volume
    def __str__(self):
        return “Volume is " + str(self.volume)

b1 = sample1(20)
print(b1)
```

O/P: Volume is 20

Operator Overloading:

Let's add the block of code which will make the '+' operator operate on objects of sample1. We have a magic method for this too, i.e., the **__add__() method**.

e.g.:

```
class sample1:
    def __init__(self, volume):
        self.volume = volume
    def __str__(self):
        return "Volume is " + str(self.volume)
    def __add__(self, other):
        volume = self.volume + other.volume
        return sample1(volume)
```

```
b1 = sample1(20)
b2 = sample1(30)
b3 = b1 + b2
print(b3)
```

O/P:

Volume is 50

Operator Overloading:

In addition to **‘self’**, the __add__() method takes another argument **‘other’**. The ‘self’ and ‘other’ refer to the two objects acting as **operands**.

We perform the addition of volumes of ‘self’ and ‘other’, and then assign this value to a new variable volume. The method then **returns** a new object of the class sample1 with volume as its **instance variable**.

Let’s see what happens behind the scenes-

- When we add b1 + b2, the interpreter calls b1.__add__(b2).
- And b1.__add__(b2) is actually executed as sample1.__add__(b1, b2).
- This will then return sample1(50).
- So, b3 = b1 + b2 is actually equivalent to b3 = sample1(50).

In this way, we can overload other operators as well.

Note that in the case of comparison operators, the magic method will return a **Boolean expression** as a result of the **comparison** and not an **object**.

Operator Overloading:

You'll find various python operators and their magic methods in the table below-

OPERATOR	EXPRESSION	MAGIC METHOD
Addition	b1 + b2	**__add__()**
Subtraction	b1 – b2	**__sub__()**
Multiplication	b1 * b2	**__mul__()**
Division	b1 / b2	**__truediv__()**
Power	b1 ** b2	**__pow__()**
Floor division	b1 // b2	**__floordiv__()**
Modulo operator	b1 % b2	**__mod__()**
Bitwise left shift	b1 << b2	**__lshift__()**
Bitwise right shift	b1 >> b2	**__rshift__()**

Operator Overloading:

OPERATOR	EXPRESSION	MAGIC METHOD
Bitwise NOT	~b1	**__invert__()**
Bitwise AND	b1 & b2	**__and__()**
Bitwise OR	b1 | b2	**__or__()**
Bitwise XOR	b1 ^ b2	**__xor__()**
Less than	b1 < b2	**__lt__()**
Less than equal to	b1 <= b2	**__le__()**
Greater than	b1 > b2	**__gt__()**
Greater than equal to	b1 >= b2	**__ge__()**
Equal to	b1 == b2	**__eq__()**
Not equal to	b1 != b2	**__ne__()**

Inheritance:

Inheritance is an important aspect of the object-oriented programming. Inheritance provides code reusability to the program because we can use an existing class to create a new class.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

class derived-**class**(base **class**):

<**class**-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket.

Syntax: **class** derive-**class**(<base **class** 1>, <base **class** 2>, ... <base **class** n>):

<**class** - suite>

Inheritance:

e.g.:

```
class Animal:
    def speak(self):
        print("Animal Speaking")

#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print(“Dog barking")

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

O/P:

```
Dog barking
Animal Speaking
```

Inheritance:

Multi-Level inheritance:

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

Syntax:

class class1:

    <**class**-suite>

class class2(class1):

    <**class** suite>

class class3(class2):

    <**class** suite>

Inheritance:

e.g.:

```
class Animal:
    def speak(self):
        print("Animal Speaking")


#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("Dog Barking")


#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating Bread...")
```

```
d = DogChild()
d.bark()
d.speak()
d.eat()
```

O/P:

```
Dog Barking
Animal Speaking
Eating Bread...
```

Inheritance:

Multiple inheritance:

Python provides us the flexibility to inherit multiple base classes in the child class.

Syntax:

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ..... BaseN):
    <class-suite>
```

Inheritance:

e.g.:

```
class Base1:
    def add(self,a,b):
        return a+b;

class Base2:
    def multi(self,a,b):
        return a*b;

class Derived(Base1,Base2):
    def divide(self,a,b):
        return a/b;

d = Derived()
print(d.add(20,10))
print(d.multi(20,10))
print(d.divide(20,10))
```

O/P:

```
30
200
2
```

Inheritance:

The issubclass(sub,sup) method:

The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

e.g.:

```
class Base1:
    def add(self,a,b):
        return a+b;
class Base2:
    def multi(self,a,b):
        return a*b;
class Derived(Base1,Base2):
    def divide(self,a,b):
        return a/b;
d = Derived()
print(issubclass(Derived,Base2))
print(issubclass(Base1,Base2))
```

O/P:

```
True
False
```

Inheritance:

The isinstance (obj, class) method:

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

e.g.:

```
class Base1:
    def add(self,a,b):
        return a+b;
class Base2:
    def multi(self,a,b):
        return a*b;
class Derived(Base1,Base2):
    def divide(self,a,b):
        return a/b;
d = Derived()
print(isinstance(d,Derived))
print(isinstance(e,Derived))
```

O/P:

```
True
False
```