CHAPTER 9

Classes and Objects

Classes and Objects

Classes and objects are the two main aspects of object oriented programming. In fact, a class is the basic building block in Python. A class creates a new type and object is an instance (or variable) of the class. Classes provides a blueprint or a template using which objects are created. In fact, *in Python, everything is an object or an instance of some class*. For example, all integer variables that we define in our program are actually instances of class int. Similarly, all string variables are objects of class string. Recall that we had used string methods using the variable name followed by the dot operator and the method name. We have already studied that we can find out the type of any object using the type() function.

```
class class_name:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Creating Objects

Once a class is defined, the next job is to create an object (or instance) of that class. The object can then access class variables and class methods using the dot operator (.). The syntax to create an object is given as,

```
object_name = class_name()
```

Creating an object or instance of a class is known as class instantiation. From the syntax, we can see that class instantiation uses function notation. Using the syntax, an empty object of a class is created. Thus, we see that in Python, to create a new object, call a class as if it were a function. The syntax for accessing a class member through the class object is

```
object_name.class_member_name
```

Programming Tip: Python does not require the new operator to create an object.

Example:

```
class ABC:
    var = 10    # class variable
obj = ABC()
print(obj.var)   # class variable is accessed using class object
```

OUTPUT

```
10
```

Programming Tip: `self` in Python works in the same way as the "`this`" pointer in C++.

Data Abstraction and Hiding through Classes

Classes provide methods to the outside world to provide the functionality of the object or to manipulate the object's data. Any entity outside the world does not know about the implementation details of the class or that method.

Data encapsulation, also called data hiding organizes the data and methods into a structure that prevents data access by any function (or method) that is not specified in the class. This ensures the integrity of the data contained in the object.

Encapsulation defines different access levels for data variables and member functions of the class. These access levels specifies the access rights for example,

• Any data or function with access level public can be accessed by any function belonging to any class. This is the lowest level of data protection.

• Any data or function with access level private can be accessed only by the class in which it is declared. This is the highest level of data protection.

Class Method And Self Argument

Class methods **(or functions defined in the class) are exactly same as ordinary functions that we have been defining so far with just one small difference. Class methods must have the first argument named as self. This is the first argument that is added to the beginning of the parameter list. Moreover, you do not pass a value for this parameter when you call the method. Python provides its value automatically. The self argument refers to the object itself. That is, the object that has called the method. This means that even if a method that takes no arguments, should be defined to accept the self. Similarly, a function defined to accept one parameter will actually take two- self and the parameter, so on and so forth.**

Since, the class methods uses self, they require an object or instance of the class to be used. For this reason, they are often referred to as *instance methods.*

Example:

```
class ABC():
    var = 10
    def display(self):
        print("In class method.....")
obj = ABC()
print(obj.var)
obj.display()
```

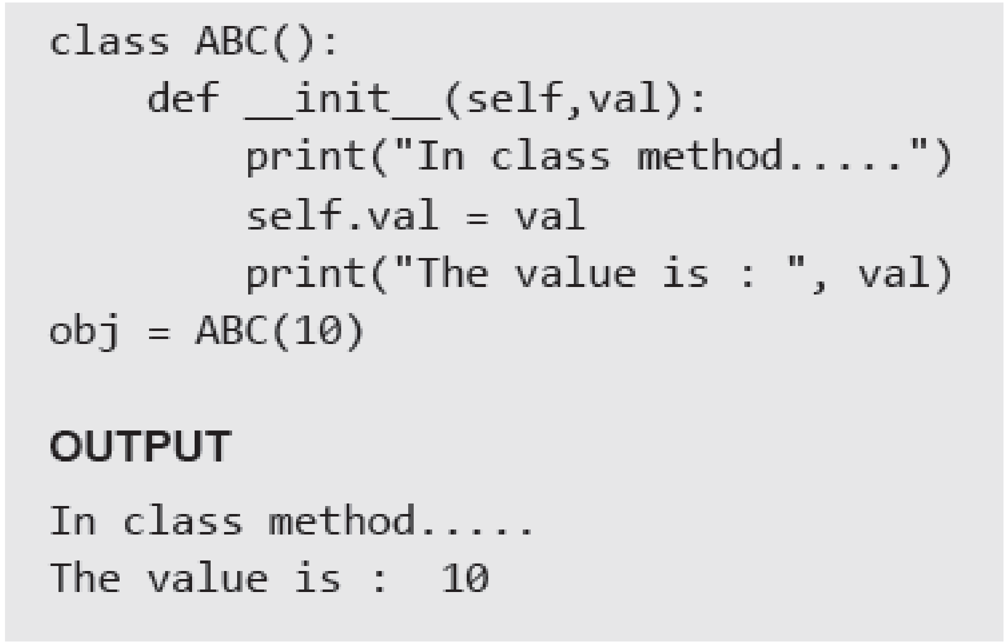
OUTPUT

```
10
In class method.....
```

The __init__() Method (The Class Constructor)

The __init__() method has a special significance in Python classes. The __init__() method is automatically executed when an object of a class is created. The method is useful to initialize the variables of the class object. Note the __init__() is prefixed as well as suffixed by double underscores.

Example:



```
class ABC():
    def __init__(self,val):
        print("In class method.....")
        self.val = val
        print("The value is : ", val)
obj = ABC(10)
```

OUTPUT

```
In class method.....
The value is :  10
```

Class Variables And Object Variables

Basically, these variables are of two types- class variables and object variables. *Class variables* are owned by the class and *object variables* are owned by each object. What this specifically means can be understood using following points.

- If a class has n objects, then there will be n separate copies of the object variable as each object will have its own object variable.
- The object variable is not shared between objects.
- A change made to the object variable by one object will not be reflected in other objects.

If a class has one class variable, then there will be one copy only for that variable. All the objects of that class will share the class variable.

- Since there exists a single copy of the class variable, any change made to the class variable by an object will be reflected to all other objects.

Class Variables And Object Variables - Example

```
class ABC():
    class_var = 0    # class variable
    def __init__(self,var):
        ABC.class_var += 1
        self.var = var  # object variable
        print("The Object value is : ", var)
        print("The value of class variable is : ", ABC.class_var)
obj1 = ABC(10)
obj2 = ABC(20)
obj3 = ABC(30)
```

OUTPUT

```
The Object value is :  10
The value of class variable is :  1
The Object value is :  20
The value of class variable is :  2
The Object value is :  30
The value of class variable is :  3
```

Programming Tip: Class variable must be prefixed by the class name and dot operator

The __del__() Method

The __del__() method does just the opposite work. The __del__() method is automatically called when an object is going out of scope. This is the time when object will no longer be used and its occupied resources are returned back to the system so that they can be reused as and when required. You can also explicitly do the same using the del keyword.

Example:

```
class ABC():
    class_var = 0    # class variable
    def __init__(self,var):
        ABC.class_var += 1
        self.var = var  # object variable
        print("The Object value is : ", var)
        print("The value of class variable is : ", ABC.class_var)
    def __del__(self):
        ABC.class_var -= 1
        Print("Object with value %d is going out of scope"%self.var)
obj1 = ABC(10)
obj2 = ABC(20)
obj3 = ABC(30)
del obj1
del obj2
del obj3
```

OUTPUT

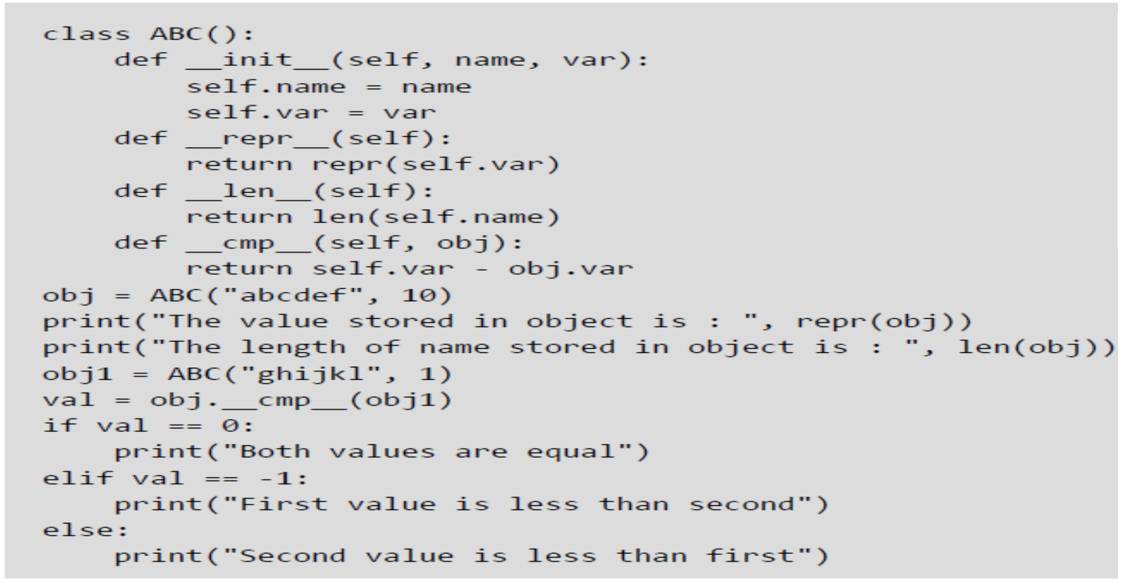
```
The Object value is :  10
```

For C++/ Java Programmers: In C++ and Java, all members are private by default but in Python, they are public by default

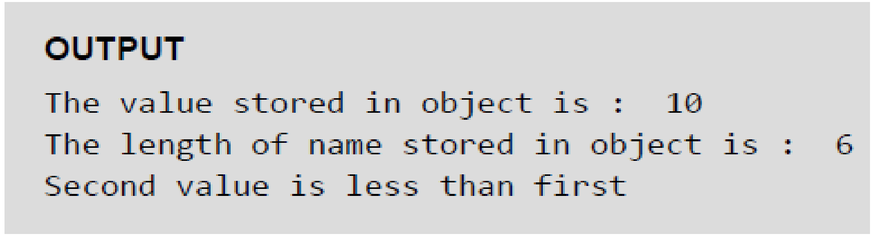
Other Special Methods

- **__repr__(): __ The __repr__() function is a built-in function with syntax repr(object). It returns a string representation of an object. The function works on any object, not just class instances.**
- **__cmp__():The __cmp__() function is called to compare two class objects.**
- **__len__(): The __len__() function is a built-in function that has the syntax, len(object). It returns the length of an object.**

Example:



```
class ABC():
    def __init__(self, name, var):
        self.name = name
        self.var = var
    def __repr__(self):
        return repr(self.var)
    def __len__(self):
        return len(self.name)
    def __cmp__(self, obj):
        return self.var - obj.var
obj = ABC("abcdef", 10)
print("The value stored in object is : ", repr(obj))
print("The length of name stored in object is : ", len(obj))
obj1 = ABC("ghijkl", 1)
val = obj.__cmp__(obj1)
if val == 0:
    print("Both values are equal")
elif val == -1:
    print("First value is less than second")
else:
    print("Second value is less than first")
```



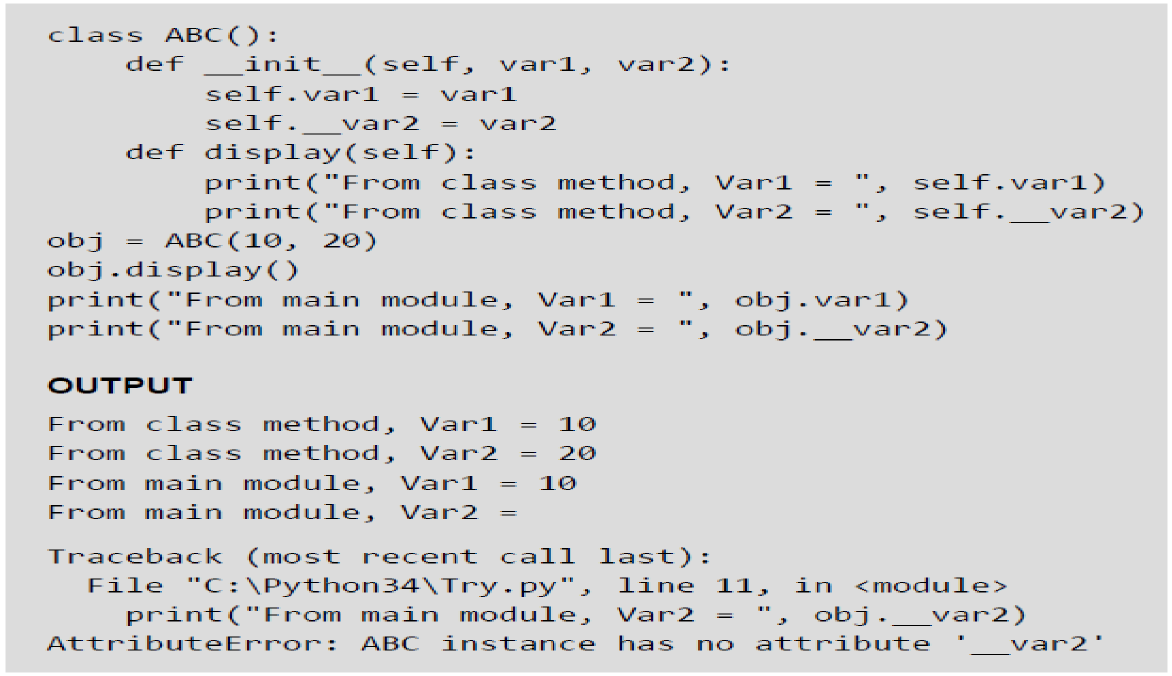
OUTPUT

```
The value stored in object is :  10
The length of name stored in object is :  6
Second value is less than first
```

Public and Private Data Members

Public variables **are those variables that are defined in the class and can be accessed from anywhere in the program, of course using the dot operator.** ***Private variables,*** **on the other hand, are those variables that are defined in the class with a double score prefix (__). These variables can be accessed only from within the class and from nowhere outside the class.**

Example:



```
class ABC():
    def __init__(self, var1, var2):
        self.var1 = var1
        self.__var2 = var2
    def display(self):
        print("From class method, Var1 = ", self.var1)
        print("From class method, Var2 = ", self.__var2)
obj = ABC(10, 20)
obj.display()
print("From main module, Var1 = ", obj.var1)
print("From main module, Var2 = ", obj.__var2)
```

OUTPUT

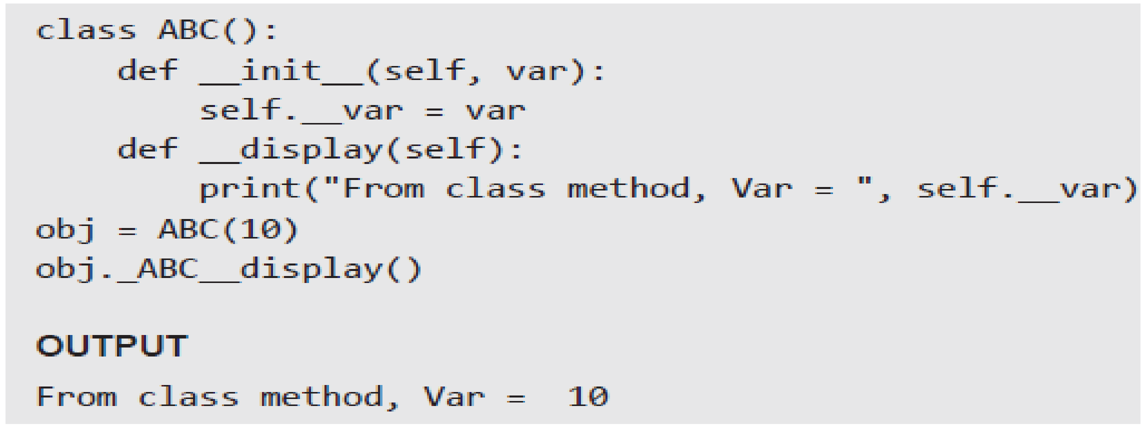
```
From class method, Var1 = 10
From class method, Var2 = 20
From main module, Var1 = 10
From main module, Var2 =

Traceback (most recent call last):
  File "C:\Python34\Try.py", line 11, in <module>
    print("From main module, Var2 = ", obj.__var2)
AttributeError: ABC instance has no attribute '__var2'
```

Private Methods

Like private attributes, you can even have private methods in your class. Usually, we keep those methods as private which have implementation details. So like private attributes, you should also not use private method from anywhere outside the class. However, if it is very necessary to access them from outside the class, then they are accessed with a small difference. A private method can be accessed using the object name as well as the class name from outside the class. The syntax for accessing the private method in such a case would be.

objectname._classname__privatemethodname

Example:



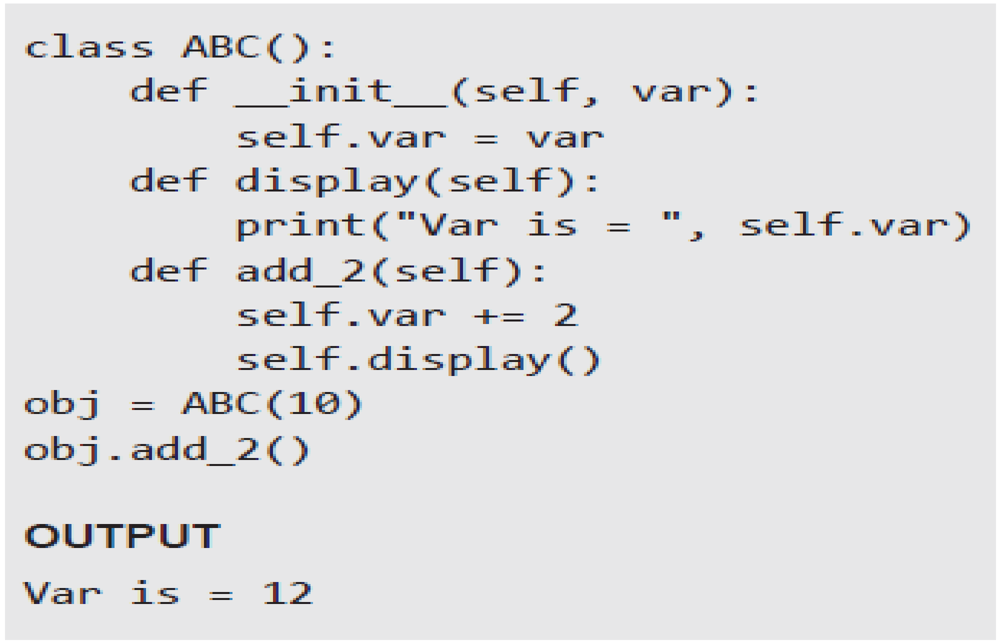
```
class ABC():
    def __init__(self, var):
        self.__var = var
    def __display(self):
        print("From class method, Var = ", self.__var)
obj = ABC(10)
obj._ABC__display()
```

OUTPUT

```
From class method, Var =  10
```

Calling a Class Method from Another Class Method

Example:



```
class ABC():
    def __init__(self, var):
        self.var = var
    def display(self):
        print("Var is = ", self.var)
    def add_2(self):
        self.var += 2
        self.display()
obj = ABC(10)
obj.add_2()
```

OUTPUT

```
Var is = 12
```

Built-in Functions To Check, Get, Set And Delete Class Attributes

hasattr(obj,name): The function is used to check if an object possess the attribute or not.

getattr(obj, name[, default]): The function is used to access or get the attribute of object. Since getattr() is a built-in function and not a method of the class, it is not called using the dot operator. Rather, it takes the object as its first parameter. The second parameter is the name of the variable as a string, and the optional third parameter is the default value to be returned if the attribute does not exist. If the attribute name does not exist in the object's namespace and the default value is also not specified, then an exception will be raised. Note that, getattr(obj, 'var') is same as writing obj.var. However, you should always try to use the latter variant.

setattr(obj,name,value): The function is used to set an attribute of the object. If attribute does not exist, then it would be created. The first parameter of the setattr() function is the object, the second parameter is the name of the attribute and the third is the new value for the specified attribute.

delattr(obj, name): The function deletes an attribute. Once deleted, the variable is no longer a class or object attribute.

Built-in Functions - Example

```
class ABC():
    def __init__(self, var):
        self.var = var
    def display(self):
        print("Var is = ", self.var)
obj = ABC(10)
obj.display()
print("Check if object has attribute var .....", hasattr(obj,'var'))
getattr(obj,'var')
setattr(obj,'var', 50)
print("After setting value, var is : ", obj.var)
setattr(obj,'count',10)
print("New variable count is created and its value is : ", obj.count)
delattr(obj,'var')
print("After deleting the attribute, var is : ", obj.var)
```

OUTPUT

```
Var is =  10

Check if object has attribute var ..... True
After setting value, var is :  50
New variable count is created and its value is :  10
After deleting the attribute, var is :
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 15, in <module>
    print "After deleting the attribute, var is : ", obj.var
AttributeError: ABC instance has no attribute 'var'
```

Built-in Class Attributes

.__dict__: The attributes gives a dictionary containing the class's or object's (with whichever it is accessed) namespace.

.__doc__: The attribute gives the class documentation string if specified. In case the documentation string is not specified, then the attribute returns None.

.__name__:The attribute returns the name of the class.

.__module__: The attribute gives the name of the module in which the class (or the object) is defined.

.__bases__: Used in inheritance to return the base classes in the order of their occurrence in the base class list.

Example:

```
class ABC():
    def __init__(self, var1, var2):
        self.var1 = var1
        self.var2 = var2
    def display(self):
        print("Var1 is = ", self.var1)
        print("Var2 is = ", self.var2)
obj = ABC(10, 12.34)
obj.display()
print("object.__dict__ - ", obj.__dict__)
print("object.__doc__ - ", obj.__doc__)
print("class.__name__ - ", ABC.__name__)
print("object.__module__ - ", obj.__module__)
print("class.__bases__ - ", ABC.__bases__)
```

OUTPUT

```
Var1 is =  10
Var2 is =  12.34
obj.__dict__ -  {'var1': 10, 'var2': 12.34}
obj.__doc__ -  None
```

Garbage Collection (Destroying Objects)

Python performs automatic garbage collection. This means that it deletes all the objects (built-in types or user defined like class objects) automatically that are no longer needed and that have gone out of scope to free the memory space. The process by which Python periodically reclaims unwanted memory is known as *garbage collection*.

Python's garbage collector runs in the background during program execution. It immediately takes action (of reclaiming memory) as soon as an object's reference count reaches zero. For example,

```
var1 = 10      # Create object var1
var2 = var1    # Increase ref. count  of var1 - object assignment
var3 = [var2]  # Increase ref. count  of var1 - object used in a list
var2 = 50      # Decrease ref. count of var1 - reassignment
var3[0] = -1   # Decrease ref. count  of var1 - removal from list
del var1       # Decrease ref. count  of var1 - object deleted
```

Class Methods

Class methods **are little different from these ordinary methods. First, they are called by a class (not by instance of the class). Second, the first argument of the classmethod is cls not the self.**

Class methods are widely used for factory methods, which instantiate an instance of a class, using different parameters than those usually passed to the class constructor.

Example:

```
class Rectangle:
        def __init__(self,length, breadth):
                self.length = length
                self.breadth = breadth

        def area(self):
                return self.length * self.breadth
        @classmethod
        def Square(cls,side):
                return cls(side,side)
S = Rectangle.Square(10)
print("AREA = ", S.area())
```

OUTPUT

```
AREA =  100
```

Static Methods

Any functionality that belongs to a class, but that does not require the object is placed in the static method. Static methods are similar to class methods. The only difference is that a static method does not receive any additional arguments. They are just like normal functions that belong to a class.

A static method does not use the self variable and is defined using a built-in function named staticmethod. Python has a handy syntax, called a *decorator*, to make it easier to apply the staticmethod function to the method function definition. The syntax for using the staticmethod decorator.

```
@staticmethod
def name (args...) :
    statements
```

Static Methods — Example

```
class Choice:
        def __init__(self, subjects):
                self.subjects = subjects
```

```
        @staticmethod
        def validate_subject(subjects):
                if "CSA" in subjects:
                        print("This option is no longer available.")
                else:
                        return True
subjects = ["DS","CSA","FoC","OS","ToC"]
if all(Choice.validate_subject(i) for i in subjects):
        ch = Choice(subjects)
        print("You have been allotted the subjects : ", subjects)
```

OUTPUT

```
This option is no longer available.
```