# Python : Class, Object and Members

a simple Python program that creates a class with single method.

class Test:

    def fun(self):

        print("Hello")

obj = Test()

obj.fun()

Output:

Hello

# self in Python 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. It binds the attributes with the given arguments.

**Self is a convention and not a real python keyword**  
self is parameter in function and user can use another parameter name in place of it. But it is advisable to use self because it increase the readability of code.

This is similar to this pointer in C++ and this reference in Java.

A method with no argument still have default argument ‘self’

**The \_\_init\_\_ method**  
The \_\_init\_\_ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

class Person:

    # init method or constructor

    def \_\_init\_\_(self, name):

        self.name = name

    # Sample Method

    def say\_hi(self):

        print('Hello, my name is', self.name)

p = Person('Shweta')

p.say\_hi()

Output:

Hello, my name is Shweta

Here, we define the \_\_init\_\_ method as taking a parameter name

**Class and Instance Variables (Or attributes)**  
In Python, instance variables are variables whose value is assigned inside a constructor or method with self.

Class variables are variables whose value is assigned in class.

class CSStudent:

    # Class Variable

    stream = 'cse'

    # The init method or constructor

    def \_\_init\_\_(self, roll):

        # Instance Variable

        self.roll = roll

# Objects of CSStudent class

a = CSStudent(101)

b = CSStudent(102)

print(a.stream)  # prints "cse"

print(b.stream)  # prints "cse"

print(a.roll)    # prints 101

# Class variables can be accessed using class

# name also

print(CSStudent.stream) # prints "cse"

We can define instance variables inside normal methods also.

|  |
| --- |
| class CSStudent:        # Class Variable      stream = 'cse'        # The init method or constructor      def \_\_init\_\_(self, roll):            # Instance Variable          self.roll = roll        # Adds an instance variable      def setAddress(self, address):          self.address = address        # Retrieves instance variable      def getAddress(self):          return self.address    a = CSStudent(101)  a.setAddress("Bombay, Maharashtra")  print(a.getAddress()) |

Output :

Bombay, Maharashtra

## **Built-In Class Attributes**

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

**\_\_dict\_\_** − Dictionary containing the class's namespace.

**\_\_doc\_\_** − Class documentation string or none, if undefined.

**\_\_name\_\_** − Class name.

**\_\_module\_\_** − Module name in which the class is defined. This attribute is "\_\_main\_\_" in interactive mode.

**\_\_bases\_\_** − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

 let us try to access all these attributes −

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print ("Total Employee %d" % Employee.empCount)

def displayEmployee(self):

print ("Name : ", self.name, ", Salary: ", self.salary)

emp1 = Employee("Zara", 2000)

emp2 = Employee("Manni", 5000)

print ("Employee.\_\_doc\_\_:", Employee.\_\_doc\_\_)

print ("Employee.\_\_name\_\_:", Employee.\_\_name\_\_)

print ("Employee.\_\_module\_\_:", Employee.\_\_module\_\_)

print ("Employee.\_\_bases\_\_:", Employee.\_\_bases\_\_)

print ("Employee.\_\_dict\_\_:", Employee.\_\_dict\_\_ )

When the above code is executed, it produces the following result −

Employee.\_\_doc\_\_: Common base class for all employees

Employee.\_\_name\_\_: Employee

Employee.\_\_module\_\_: \_\_main\_\_

Employee.\_\_bases\_\_: (<class 'object'>,)

Employee.\_\_dict\_\_: {

'displayCount': <function Employee.displayCount at 0x0160D2B8>,

'\_\_module\_\_': '\_\_main\_\_', '\_\_doc\_\_': 'Common base class for all employees',

'empCount': 2, '\_\_init\_\_':

<function Employee.\_\_init\_\_ at 0x0124F810>, 'displayEmployee':

<function Employee.displayEmployee at 0x0160D300>,

'\_\_weakref\_\_':

<attribute '\_\_weakref\_\_' of 'Employee' objects>, '\_\_dict\_\_':

<attribute '\_\_dict\_\_' of 'Employee' objects>

}

Instead of using the normal statements to access attributes, you can use the following functions −

* The **getattr(obj, name[, default])** − to access the attribute of object.
* The **hasattr(obj,name)** − to check if an attribute exists or not.
* The **setattr(obj,name,value)** − to set an attribute. If attribute does not exist, then it would be created.
* The **delattr(obj, name)** − to delete an attribute.

hasattr(emp1, 'salary') # Returns true if 'salary' attribute exists

getattr(emp1, 'salary') # Returns value of 'salary' attribute

setattr(emp1, 'salary', 7000) # Set attribute 'salary' at 7000

delattr(emp1, 'salary') # Delete attribute 'salary'

## **Destroying Objects (Garbage Collection)**

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed as Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it is deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

However, 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.

## **Class Inheritance**

Instead of starting from a scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

### **Syntax**

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

class SubClassName (ParentClass1[, ParentClass2, ...]):

'Optional class documentation string'

class\_suite

class Parent: # define parent class

parentAttr = 100

def \_\_init\_\_(self):

print ("Calling parent constructor")

def parentMethod(self):

print ('Calling parent method')

def setAttr(self, attr):

Parent.parentAttr = attr

def getAttr(self):

print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class

def \_\_init\_\_(self):

print ("Calling child constructor")

def childMethod(self):

print ('Calling child method')

c = Child() # instance of child

c.childMethod() # child calls its method

c.parentMethod() # calls parent's method

c.setAttr(200) # again call parent's method

c.getAttr() # again call parent's method

When the above code is executed, it produces the following result −

Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200

In a similar way, you can drive a class from multiple parent classes as follows −

class A: # define your class A

.....

class B: # define your calss B

.....

class C(A, B): # subclass of A and B

.....

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

* The **issubclass(sub, sup)** boolean function returns True, if the given subclass **sub** is indeed a subclass of the superclass **sup**.
* The **isinstance(obj, Class)** boolean function returns True, if *obj* is an instance of class *Class* or is an instance of a subclass of Class

## **Overriding Methods**

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass

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

When the above code is executed, it produces the following result −

Calling child method

## **Base Overloading Methods**

The following table lists some generic functionality that you can override in your own classes −

|  |  |
| --- | --- |
| **Sr.No.** | **Method, Description & Sample Call** |
| 1 | **\_\_init\_\_ ( self [,args...] )**  Constructor (with any optional arguments)  Sample Call : *obj = className(args)* |
| 2 | **\_\_del\_\_( self )**  Destructor, deletes an object  Sample Call : *del obj* |
| 3 | **\_\_repr\_\_( self )**  Evaluatable string representation  Sample Call : *repr(obj)* |
| 4 | **\_\_str\_\_( self )**  Printable string representation  Sample Call : *str(obj)* |
| 5 | **\_\_cmp\_\_ ( self, x )**  Object comparison  Sample Call : *cmp(obj, x)* |

## **Overloading Operators**

Suppose you have created a Vector class to represent two-dimensional vectors. What happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *\_\_add\_\_* method in your class to perform vector addition and then the plus operator would behave as per expectation −

class Vector:

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

def \_\_str\_\_(self):

return 'Vector (%d, %d)' % (self.a, self.b)

def \_\_add\_\_(self,other):

return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)

v2 = Vector(5,-2)

print (v1 + v2)

When the above code is executed, it produces the following result −

Vector(7,8)

## **Data Hiding**

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then will not be directly visible to outsiders.

class JustCounter:

\_\_secretCount = 0

def count(self):

self.\_\_secretCount += 1

print (self.\_\_secretCount)

counter = JustCounter()

counter.count()

counter.count()

print (counter.\_\_secretCount)

When the above code is executed, it produces the following result −

1

2

Traceback (most recent call last):

File "test.py", line 12, in <module>

print counter.\_\_secretCount

AttributeError: JustCounter instance has no attribute '\_\_secretCount'

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object.\_className\_\_attrName*. If you would replace your last line as following, then it works for you −

.........................

print (counter.\_JustCounter\_\_secretCount)

When the above code is executed, it produces the following result −

1

2

2