Python - Classes and Objects

Python is an **object-oriented programming language**, which means that it is based on principle of OOP concept. The entities used within a Python program is an object of one or another class. For instance, numbers, strings, lists, dictionaries, and other similar entities of a program are objects of the corresponding built-in class.

In Python, a class named **Object** is the base or parent class for all the classes, built-in as well as user defined.

What is a Class in Python?

In Python, a **class** is a user defined entity (data types) that defines the type of data an object can contain and the actions it can perform. It is used as a template for creating objects. For instance, if we want to define a class for Smartphone in a Python program, we can use the type of data like RAM, ROM, screen-size and actions like call and message.

Creating Classes in Python

The **class** keyword is used to create a new class in Python. The name of the class immediately follows the keyword **class** followed by a colon as shown below –

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

- The class has a documentation string, which can be accessed via ClassName.__doc__.
- The class_suite consists of all the component statements defining class members, data attributes and functions.

Example

Following is the example of a simple Python class –

```
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
```

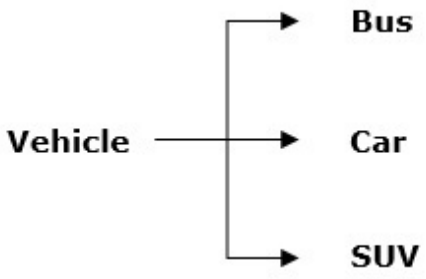
- The variable **empCount** is a class variable whose value is shared among all instances of a this class. This can be accessed as **Employee.empCount** from inside the class or outside the class.
- The first method **__init__()** is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is **self**. Python adds the **self** argument to the list for you; you do not need to include it when you call the methods.

Learn **Python** in-depth with real-world projects through our **Python certification course**. Enroll and become a certified expert to boost your career.

What is an Object?

An **object** is refered to as an instance of a given Python class. Each object has its own attributes and methods, which are defined by its class.

When a class is created, it only describes the structure of obejcts. The memory is allocated when an object is instantiated from a class.



In the above figure, **Vehicle** is the class name and **Car**, **Bus** and **SUV** are its objects.

Creating Objects of Classes in Python

To create instances of a class, you call the class using class name and pass in whatever arguments its **__init__** method accepts.

```
# This would create first object of Employee class
emp1 = Employee("Zara", 2000)
# This would create second object of Employee class
emp2 = Employee("Manni", 5000)
```

Accessing Attributes of Objects in Python

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

```
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

Now, putting all the concepts together –

</> Open Compiler

```
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)

# This would create first object of Employee class
emp1 = Employee("Zara", 2000)
```

```
# This would create second object of Employee class
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

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

```
Name :  Zara , Salary:  2000
Name :  Manni , Salary:  5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time –

```
# Add an 'age' attribute
emp1.age = 7
# Modify 'age' attribute
emp1.age = 8
# Delete 'age' attribute
del emp1.age
```

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

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

```
# Returns true if 'age' attribute exists
hasattr(emp1, 'age')
# Returns value of 'age' attribute
getattr(emp1, 'age')
# Set attribute 'age' at 8
setattr(emp1, 'age', 8)
# Delete attribute 'age'
delattr(empl, 'age')
```

Built-In Class Attributes in Python

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

SNo.	Attributes & Description
1	**__dict__** Dictionary containing the class's namespace.
2	**__doc__** Class documentation string or none, if undefined.
3	**__name__** Class name
4	**__module__** Module name in which the class is defined. This attribute is "__main__" in interactive mode.
5	**__bases__** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Example

For the above **Employee** class, let us try to access its attributes −

</> Open Compiler

```
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)
```

```
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__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Built-in Class of Python datatypes

As mentioned earlier, Python follows object-oriented programming paradigm. Entities like strings, lists and data types belongs to one or another built-in class.

If we want to see which data type belongs to which built-in class, we can use the Python type() function. This function accepts a data type and returns its corresponding class.

Example

The below example demonstrates how to check built-in class of a given data type.

</> Open Compiler

```
num = 20
print (type(num))
num1 = 55.50
print (type(num1))
s = "TutorialsPoint"
print (type(s))
dct = {'a':1,'b':2,'c':3}
print (type(dct))
```

```
def SayHello():
   print ("Hello World")
   return
print (type(SayHello))
```

When you execute this code, it will display the corresponding classes of Python data types –

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'dict'>
<class 'function'>
```

Garbage Collection(Destroying Objects) in Python

Python deletes unwanted 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 **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's 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.

```
# Create object <40>
a = 40
# Increase ref. count  of <40>
b = a
# Increase ref. count  of <40>
c = [b]

# Decrease ref. count  of <40>
del a
# Decrease ref. count  of <40>
b = 100
```

```
# Decrease ref. count  of <40>
c[0] = -1
```

You normally will not notice when the garbage collector destroys an unused instance and reclaims its space. But 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.

Example

The __del__() destructor prints the class name of an instance that is about to be destroyed as shown in the below code block –

</> Open Compiler

```
class Point:
   def __init__( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print (class_name, "destroyed")

pt1 = Point()
pt2 = pt1
pt3 = pt1
# prints the ids of the obejcts
print (id(pt1), id(pt2), id(pt3))
del pt1
del pt2
del pt3
```

On executing, the above code will produces following result –

```
135007479444176 135007479444176 135007479444176
Point destroyed
```

Data Hiding in Python

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 are not be

directly visible to outsiders.

Example

```
class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

Open Compiler

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

```
1
2
ERROR!
Traceback (most recent call last):
  File <main.py>", line 11, in <module>
AttributeError: 'JustCounter' object 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
```