Python Programming

High performance. Delivered.

Module 11 - Classes & Objects

accenture

Strategy | Digital | Technology | Operations

Module Objectives

At the end of this module, you will be able to

- Python Classes & Objects
- Defining a Class in Python
- Creating an Object in Python
- Built-In Class Attributes
- Constructors in Python
- Deleting Attributes and Objects
- **Python Inheritance**
- **Method Overriding in Python**
- **Python Multiple Inheritance**
- **Multilevel Inheritance in Python**
- Python Operator Overloading
- Special Functions in Python
- Overloading the + Operator
- Overloading Comparison Operators in Python
- Data Hiding

Creating Classes

- The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows −

```
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.

Class Definition - Example

```
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
```

Class Definition (Contd)

- 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.

Creating Objects

- 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

- 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
```

Example:

```
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
```

OUTPUT:

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

Adding , Removing and Updaing attributes

- remove, or modify attributes of classes and objects at any time

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

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.

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 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.

Destroying Objects (Garbage Collection)- Example

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

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned 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.

Destroying Objects (Garbage Collection)- Example (Contd)

- This __del__() destructor prints the class name of an instance that is about to be destroyed

```
#!/usr/bin/python

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

```
OUTPUT
3083401324 3083401324 3083401324
Point destroyed
```

Class Inheritance

- Instead of starting from scratch, you can create a class by deriving it from a preexisting 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.

Class Inheritance - 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 Inheritance - Example

```
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
```

Class Inheritance – Example - OUTPUT

```
OUTPUT:
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Multiple Inheritance

- Multiple inheritance is possible in Python unlike other programming languages. A class can be derived from more than one base classes. The syntax for multiple inheritance is similar to single inheritance

Multiple Inheritance - Example

```
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

Multilevel Inheritance

- we can inherit form a derived class. This is also called multilevel inheritance. Multilevel inheritance can be of any depth in Python

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```

Special Functions in Python

- Class functions that begins with double underscore (__) are called special functions in Python. This is because, well, they are not ordinary. The __init__() function we defined above, is one of them. It gets called every time we create a new object of that class. There are a ton of special functions in Python.
- Using special functions, we can make our class compatible with built-in functions.

```
class Point:
    # previous definitions...

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
```

Overriding Methods

- You can always override your parent class methods. One reason for overriding parent's methods is because 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
```

```
OUTPUT:
Calling child method
```

Overloading Methods

- Following table lists some generic functionality that you can override in your own classes −

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 −

Overloading Operators - Example

```
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
```

```
OUTPUT:
Vector(7,8)
```

Overloading the + Operator

- To overload the + sign, we will need to implement __add__() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is sensible to return a Point object of the coordinate sum.

```
class Point:
    # previous definitions...

    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
```

- What actually happens is that, when you do p1 + p2, Python will call p1.__add__(p2) which in turn is Point.__add__(p1,p2).
- Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator Overloading Special Functions in Python

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Overloading Comparison Operators in Python

- Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well. Suppose, we wanted to implement the less than symbol < symbol in our Point class. Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

Overloading Comparison Operators - Example

```
class Point:
    # previous definitions...

    def __lt__(self,other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag
```

Overloading Comparison Operators in Python –(Contd)

- Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Comparision Operator Overloading in Python

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

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 are 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
```

```
OUTPUT:
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no
 attribute '__secretCount'
```

DATA Hiding - (Contd)

- 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
```

```
OUTPUT:
1
2
2
```

Questions