Classes and Objects

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/Python

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.

Overview of OOP Terminology (Cont.)

- 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.

Overview of OOP Terminology (Cont.)

- 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.

Overview of OOP Terminology (Cont.)

- Instance variable
  - A variable that is defined inside a method and belongs only to the current instance of a class.
- Inheritance
  - The transfer of the characteristics of a class to other classes that are derived from it.

Overview of OOP Terminology (Cont.)

- 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.

Overview of OOP Terminology (Cont.)

- Object
  - A unique instance of a data structure that is 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

- 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
```

Creating Classes (Cont.)

- 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.

Creating Classes (Cont.)

```
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)
```

Creating Classes (Cont.)

- The variable `empCount` is a class variable whose value is shared among all the instances of a in 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 create a new instance of this class.

Creating Classes (Cont.)

- 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.
- Do not need to include it when call the methods.

Creating Instance Objects

- To create instances of a class, 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

- 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)
```

Accessing Attributes (Cont.)

```
1  class Employee:
2      """Common base class for all employees"""
3      empCount = 0
4
5      def __init__(self, name, salary):
6          self.name = name
7          self.salary = salary
8          Employee.empCount += 1
9
10     def displayCount(self):
11         print ("Total Employee %d" % Employee.empCount)
12
13     def displayEmployee(self):
14         print ("Name : ", self.name,  ", Salary: ", self.salary)
15
16 # This would create first object of Employee class
17 emp1 = Employee ("Zara" , 2000)
18 emp2 = Employee ("Manni", 5000)
19 emp1.displayEmployee()
20 emp2.displayEmployee()
21 print ("Total Employee %d" % Employee.empCount)
```

Console | Problems

<terminated> demo.py [/usr/bin/python3.6]

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

Accessing Attributes (Cont.)

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

```
emp1.salary = 7000
# Add an 'salary' attribute.
emp1.name = 'xyz'  # Modify 'age' attribute.
del emp1.salary  # Delete 'age' attribute.
```

Accessing Attributes (Cont.)

- Instead of using the normal statements to access attributes, 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.

Accessing Attributes (Cont.)

```
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'
```

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.

Built-In Class Attributes (Cont.)

```
13      def displayEmployee(self):
14          print ("Name : ", self.name,  ", Salary: ", self.salary)
15
16 emp1 = Employee("Zara", 2000)
17 emp2 = Employee("Manni", 5000)
18 print ("Employee.__doc__:", Employee.__doc__)
19 print ("Employee.__name__:", Employee.__name__)
20 print ("Employee.__module__:", Employee.__module__)
21 print ("Employee.__bases__:", Employee.__bases__)
22 print ("Employee.__dict__:", Employee.__dict__ )
23
```

Console Problems

<terminated> demo.py [/usr/bin/python3.6]

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', '__doc__': 'Common base class for all employees', 'empCount':
```

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***.

Destroying Objects (Garbage Collection) (Cont.)

- Python's garbage collector runs during program execution.
- 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.

Destroying Objects (Garbage Collection) (Cont.)

- 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.

Destroying Objects (Garbage Collection) (Cont.)

```
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>
```

Destroying Objects (Garbage Collection) (Cont.)

- Normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space.
- 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.

Destroying Objects (Garbage Collection) (Cont.)

```
1  class Point:
2      def __init__( self, x=0, y=0):
3          self.x = x
4          self.y = y
5      def __del__(self):
6          class_name = self.__class__.__name__
7          print (class_name, "destroyed")
8
9  pt1 = Point()
10 pt2 = pt1
11 pt3 = pt1
12 print (id(pt1), id(pt2), id(pt3));   # prints the ids of the obejcts
13 del pt1
14 del pt2
15 del pt3
```

Console | Problems

<terminated> demo.py [/usr/bin/python3.6]

```
140478876712800 140478876712800 140478876712800
Point destroyed
```

Destroying Objects (Garbage Collection) (Cont.)

- Note – Ideally, should define your classes in a separate file, then should import them in your main program file using `import` statement.
- In the previous example, assuming definition of a `Point` class is contained in `point.py` and there is no other executable code in it.

```
import point
p1 = point.Point()
```

Class Inheritance

- Instead of starting from a scratch, 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 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 (Cont.)

- 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 (Cont.)

```
1  class Parent:        # define parent class
2      parentAttr = 100
3      def __init__(self):
4          print ("Calling parent constructor")
5
6      def parentMethod(self):
7          print ('Calling parent method')
8
9      def setAttr(self, attr):
10         Parent.parentAttr = attr
11
12     def getAttr(self):
13         print ("Parent attribute :", Parent.parentAttr)
14
15 class Child(Parent): # define child class
16     def __init__(self):
17         print ("Calling child constructor")
18
19     def childMethod(self):
20         print ('Calling child method')
21
```

Class Inheritance (Cont.)

```
1 from demo import Parent, Child
2
3 c = Child()            # instance of child
4 c.childMethod()        # child calls its method
5 c.parentMethod()       # calls parent's method
6 c.setAttr(200)         # again call parent's method
7 c.getAttr()            # again call parent's method
8
```

Console Problems

<terminated> test.py [/usr/bin/python3.6]

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

Class Inheritance (Cont.)

- 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

- Can always override parent class methods.
- One reason for overriding parent's methods is that may want special or different functionality in subclass.

Overriding Methods (Cont.)

```
1 class Parent:          # define parent class
2     def myMethod(self):
3         print ('Calling parent method')
4
5 class Child(Parent): # define child class
6     def myMethod(self):
7         print ('Calling child method')
8
9 c = Child()           # instance of child
10 c.myMethod()         # child calls overridden method
11
```

Console Problems

<terminated> test.py [/usr/bin/python3.6]

```
Calling child method
```

Base Overloading Methods

- `__init__ ( self [,args...] )`
  - Constructor (with any optional arguments)
  - Sample Call : `obj = className(args)`
- `__del__( self )`
  - Destructor, deletes an object
  - Sample Call : `del obj`
- `__repr__( self )`
  - Evaluatable string representation
  - Sample Call : `repr(obj)`

Base Overloading Methods (Cont.)

- `__str__( self )`
  - Printable string representation
  - Sample Call : `str(obj)`
- `__cmp__ ( self, x )`
  - Object comparison
  - Sample Call : `cmp(obj, x)`

Overloading Operators

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

```
1  class Vector:
2      def __init__(self, a, b):
3          self.a = a
4          self.b = b
5
6      def __str__(self):
7          return 'Vector (%d, %d)' % (self.a, self.b)
8
9      def __add__(self,other):
10         return Vector(self.a + other.a, self.b + other.b)
11
12 v1 = Vector(2,10)
13 v2 = Vector(5,-2)
14 print (v1 + v2)
15
```

Console Problems

<terminated> test.py [/usr/bin/python3.6]

```
Vector (7, 8)
```

Data Hiding

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

Data Hiding (Cont.)

```
1  class JustCounter:
2      __secretCount = 0
3
4      def count(self):
5          self.__secretCount += 1
6          print (self.__secretCount)
7
8  counter = JustCounter()
9  counter.count()
10 counter.count()
11 print (counter.__secretCount)
12
```

Console Problems

<terminated> test.py [/usr/bin/python3.6]

```
Traceback (most recent call last):
  File "/home/instructor/PythonHome/0830/test.py", line 11, in <module>
1
2
    print (counter.__secretCount)
AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

Data Hiding (Cont.)

- Python protects those members by internally changing the name to include the class name.
- Can access such attributes as **`object._className__attrName`**.
- If you would replace your last line as following, then it works :

```
print (counter._JustCounter__secretCount)
```

Lab

```
1 class MyClass:
2     var = '안녕하세요'
3     def sayHello(self):
4         print(self.var)
5
6 obj = MyClass()     # MyClass 인스턴스 객체 생성
7 print(obj.var)      # '안녕하세요'가 출력됨
8 obj.sayHello()      # '안녕하세요'가 출력됨
9
```

Lab (Cont.)

```
1  class MyClass:
2      var = '안녕하세요!!'
3      def sayHello(self):
4          param1 = '안녕'
5          self.param2 = '하이'
6          print(param1)          # '안녕'이 출력됨
7          print(self.var)          # '안녕하세요'가 출력됨
8
9  obj = MyClass()
10 print(obj.var)                  # '안녕하세요'가 출력됨
11 obj.sayHello()
12 #obj.param1
```

Lab (Cont.)

```
1  class MyClass:
2      def sayHello(self):
3          print('안녕하세요')
4
5      def sayBye(self, name):
6          print('%s! 다음에 보자!' %name)
7
8  obj = MyClass()
9  obj.sayHello()          # '안녕하세요'가 출력됨
10 obj.sayBye('철수')      # '철수! 다음에 보자!'가 출력됨
11
```

Lab (Cont.)

```
1 class MyClass:
2     def __init__(self):
3         self.var = '안녕하세요!'
4         print('MyClass 인스턴스 객체가 생성되었습니다')
5
6 obj = MyClass()    # 'MyClass 인스턴스 객체가 생성되었습니다'가 출력됨
7 print(obj.var)      # '안녕하세요'가 출력됨
8
```

Lab (Cont.)

```
1 class MyClass:
2     def __del__(self):
3         print('MyClass 인스턴스 객체가 메모리에서 제거됩니다')
4
5 obj = MyClass()
6 del obj          # 'MyClass 인스턴스 객체가 메모리에서 제거됩니다'가 출력됨
7
```

Lab (Cont.)

```
1  class Add:
2      def add(self, n1, n2):
3          return n1+n2
4
5  class Calculator(Add):
6      def sub(self, n1, n2):
7          return n1-n2
8
9  obj = Calculator()
10 print(obj.add(1, 2))    # 3이 출력됨
11 print(obj.sub(1, 2))    # -1이 출력됨
12
```