CLASS AND OBJECT VARIABLES

We have already discussed the functionality part of classes and objects (i.e. methods), now let us learn about the data part. The data parts, i.e. fields, are nothing but ordinary variables that are *bound* to the **namespaces** of the classes and objects. This means that these names are valid within the context of these classes and objects only. That's why they are called *name spaces*.

There are two types of *fields* - class variables and object variables which are classified depending on whether the class or the object *owns* the variables respectively.

Class variables are shared - they can be accessed by all instances of that class. There is only one copy of the class variable and when any one object makes a change to a class variable, that change will be seen by all the other instances.

Object variables are owned by each individual object/instance of the class. In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance.

An example will make this easy to understand (save as `oop_objvar.py`):

```
class Robot:
    """Represents a robot, with a name."""

    # A class variable, counting the number of robots
    population = 0

    def __init__(self, name):
        """Initializes the data."""
        self.name = name
        print "(Initializing {})".format(self.name)

        # When this person is created, the robot
        # adds to the population
        Robot.population += 1

    def die(self):
        """I am dying."""
        print "{} is being destroyed!".format(self.name)
```

```
        Robot.population -= 1


        if Robot.population == 0:
            print "{} was the last one.".format(self.name)
        else:
            print "There are still {:d} robots working.".format(
                Robot.population)


    def say_hi(self):
        """Greeting by the robot.


        Yeah, they can do that."""
        print "Greetings, my masters call me {}.".format(self.name)


    @classmethod
    def how_many(cls):
        """Prints the current population."""
        print "We have {:d} robots.".format(cls.population)


droid1 = Robot("R2-D2")
```

```
droid1.say_hi()

Robot.how_many()


droid2 = Robot("C-3PO")

droid2.say_hi()

Robot.how_many()


print "\nRobots can do some work here.\n"


print "Robots have finished their work. So let's destroy them."

droid1.die()

droid2.die()


Robot.how_many()
```

Output:

```
$ python oop_objvar.py

(Initializing R2-D2)

Greetings, my masters call me R2-D2.

We have 1 robots.
```

```
(Initializing C-3PO)
Greetings, my masters call me C-3PO.
We have 2 robots.

Robots can do some work here.

Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3PO is being destroyed!
C-3PO was the last one.
We have 0 robots.
```

How It Works

This is a long example but helps demonstrate the nature of class and object variables. Here, `population` belongs to the `Robot` class and hence is a class variable. The `name` variable belongs to the object (it is assigned using `self`) and hence is an object variable.

Thus, we refer to the `population` class variable as `Robot.population` and not as `self.population`. We refer to the object variable `name` using `self.name` notation in

the methods of that object. Remember this simple difference between class and object variables. Also note that an object variable with the same name as a class variable will hide the class variable!

Instead of Robot.population, we could have also used self.__class__.population because every object refers to it's class via the self.__class__ attribute.

The how_many is actually a method that belongs to the class and not to the object. This means we can define it as either a class method or a static method depending on whether we need to know which class we are part of. Since we refer to a class variable, let's use class method.

We have marked the how_many method as a class method using a decorator.

Decorators can be imagined to be a shortcut to calling a wrapper function, so applying the @classmethod decorator is same as calling:

```
how_many = classmethod(how_many)
```

Observe that the *init* method is used to initialize the Robot instance with a name.

In this method, we increase the population count by 1 since we have one more robot being added. Also observe that the values of self.name is specific to each object which indicates the nature of object variables.

Remember, that you must refer to the variables and methods of the same object using the `self` **only**. This is called an **attribute reference**.

In this program, we also see the use of **docstrings** for classes as well as methods. We can access the class docstring at runtime using `Robot.__doc__` and the method docstring as `Robot.say_hi.__doc__`

In the `die` method, we simply decrease the `Robot.population` count by 1.

All class members are public. One exception: If you use data members with names using the *double underscore prefix* such as `__privatevar`, Python uses name-mangling to effectively make it a private variable.

Thus, the convention followed is that any variable that is to be used only within the class or object should begin with an underscore and all other names are public and can be used by other classes/objects. Remember that this is only a convention and is not enforced by Python (except for the double underscore prefix).

NOTE	*Note for C++/Java/C# Programmers* All class members (including the data members) are *public* and all the methods are *virtual* in Python.

Source: http://www.swaroopch.com/notes/python/