Object Oriented Programming

In all the programs we wrote till now, we have designed our program around functions i.e. blocks of statements which manipulate data. This is called the *procedure-oriented* way of programming. There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the *object oriented* programming paradigm. Most of the time you can use procedural programming, but when writing large programs or have a problem that is better suited to this method, you can use object oriented programming techniques.

Classes and objects are the two main aspects of object oriented programming. A **class** creates a new *type* where **objects** are **instances** of the class. An analogy is that you can have variables of type `int` which translates to saying that variables that store integers are variables which are instances (objects) of the `int` class.

> **Note for Static Language Programmers**
>
> Note that even integers are treated as objects (of the `int` class). This is unlike C++ and Java (before version 1.5) where integers are primitive native types.
>
> See `help(int)` for more details on the class.
>
> C# and Java 1.5 programmers will find this similar to the *boxing and unboxing* concept.

Objects can store data using ordinary variables that *belong* to the object. Variables that belong to an object or class are referred to as **fields**. Objects can also have functionality by using functions that *belong* to a class. Such functions are called **methods** of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object. Collectively, the fields and methods can be referred to as the **attributes** of that class.

Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called **instance variables** and **class variables** respectively.

A class is created using the `class` keyword. The fields and methods of the class are listed in an indented block.

The `self`

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object *itself*, and by convention, it is given the name `self` .

Although, you can give any name for this parameter, it is *strongly recommended* that you use the name `self` - any other name is definitely frowned upon. There are many advantages to using a standard name - any reader of your program will immediately recognize it and even specialized IDEs (Integrated Development Environments) can help you if you use `self` .

> **Note for C++/Java/C# Programmers**
>
> The `self` in Python is equivalent to the `this` pointer in C++ and the `this` reference in Java and C#.

You must be wondering how Python gives the value for `self` and why you don't need to give a value for it. An example will make this clear. Say you have a class called `MyClass` and an instance of this class called `myobject` . When you call a method of this object as `myobject.method(arg1, arg2)` , this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` - this is all the special `self` is about.

This also means that if you have a method which takes no arguments, then you still have to have one argument - the `self` .

Classes

The simplest class possible is shown in the following example (save as `oop_simplestclass.py`).

```
class Person:
    pass  # An empty block

p = Person()
print(p)
```

Output:

```
$ python oop_simplestclass.py
<__main__.Person instance at 0x10171f518>
```

How It Works

We create a new class using the `class` statement and the name of the class. This is followed by an indented block of statements which form the body of the class. In this case, we have an empty block which is indicated using the `pass` statement.

Next, we create an object/instance of this class using the name of the class followed by a pair of parentheses. (We will learn more about instantiation in the next section). For our verification, we confirm the type of the variable by simply printing it. It tells us that we have an instance of the `Person` class in the `__main__` module.

Notice that the address of the computer memory where your object is stored is also printed. The address will have a different value on your computer since Python can store the object wherever it finds space.

Methods

We have already discussed that classes/objects can have methods just like functions except that we have an extra `self` variable. We will now see an example (save as `oop_method.py`).

```
class Person:
    def say_hi(self):
        print('Hello, how are you?')

p = Person()
p.say_hi()
# The previous 2 lines can also be written as
# Person().say_hi()
```

Output:

```
$ python oop_method.py
Hello, how are you?
```

How It Works

Here we see the `self` in action. Notice that the `say_hi` method takes no parameters but still has the `self` in the function definition.

The `__init__` method

There are many method names which have special significance in Python classes. We will see the significance of the `__init__` method now.

The `__init__` method is run as soon as an object of a class is instantiated (i.e. created). The method is useful to do any *initialization* (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores both at the beginning and at the end of the name.

Example (save as `oop_init.py`):

```
class Person:
    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Swaroop')
p.say_hi()
# The previous 2 lines can also be written as
# Person('Swaroop').say_hi()
```

Output:

```
$ python oop_init.py
Hello, my name is Swaroop
```

How It Works

Here, we define the `__init__` method as taking a parameter `name` (along with the usual `self`). Here, we just create a new field also called `name`. Notice these are two different variables even though they are both called 'name'. There is no problem because the dotted notation `self.name` means that there is something called "name" that is part of the object called "self" and the other `name` is a local variable. Since we explicitly indicate which name we are referring to, there is no confusion.

When creating new instance `p`, of the class `Person`, we do so by using the class name, followed by the arguments in the parentheses: p = Person('Swaroop').

We do not explicitly call the `__init__` method. This is the special significance of this method.

Now, we are able to use the `self.name` field in our methods which is demonstrated in the `say_hi` method.

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 part, 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 its 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 `classmethod` or a `staticmethod` depending on whether we need to know which class we are part of. Since we refer to a class variable, let's use `classmethod`.

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 (i.e. a function that "wraps" around another function so that it can do something before or after the inner function), so applying the `@classmethod` decorator is the 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 for C++/Java/C# Programmers**
>
> All class members (including the data members) are *public* and all the methods are *virtual* in Python.

Inheritance

One of the major benefits of object oriented programming is **reuse** of code and one of the ways this is achieved is through the **inheritance** mechanism. Inheritance can be best imagined as implementing a **type and subtype** relationship between classes.

Suppose you want to write a program which has to keep track of the teachers and students in a college. They have some common characteristics such as name, age and address. They also have specific characteristics such as salary, courses and leaves for teachers and, marks and fees for students.

You can create two independent classes for each type and process them but adding a new common characteristic would mean adding to both of these independent classes. This quickly becomes unwieldy.

A better way would be to create a common class called `SchoolMember` and then have the teacher and student classes *inherit* from this class, i.e. they will become sub-types of this type (class) and then we can add specific characteristics to these sub-types.

There are many advantages to this approach. If we add/change any functionality in `SchoolMember`, this is automatically reflected in the subtypes as well. For example, you can add a new ID card field for both teachers and students by simply adding it to the SchoolMember class. However, changes in the subtypes do not affect other subtypes. Another advantage is that you can refer to a teacher or student object as a `SchoolMember` object which could be useful in some situations such as counting of the number of school members. This is called **polymorphism** where a sub-type can be substituted in any situation where a parent type is expected, i.e. the object can be treated as an instance of the parent class.

Also observe that we reuse the code of the parent class and we do not need to repeat it in the different classes as we would have had to in case we had used independent classes.

The `SchoolMember` class in this situation is known as the **base class** or the **superclass**. The `Teacher` and `Student` classes are called the **derived classes** or **subclasses**.

We will now see this example as a program (save as `oop_subclass.py`):

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Initialized SchoolMember: {})'.format(self.name))

    def tell(self):
        '''Tell my details.'''
        print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")


class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Initialized Teacher: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Salary: "{:d}"'.format(self.salary))


class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Initialized Student: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Marks: "{:d}"'.format(self.marks))
```

```
t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)

# prints a blank line
print()

members = [t, s]
for member in members:
    # Works for both Teachers and Students
    member.tell()
```

Output:

```
$ python oop_subclass.py
(Initialized SchoolMember: Mrs. Shrividya)
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"25" Marks: "75"
```

How It Works

To use inheritance, we specify the base class names in a tuple following the class name in the class definition (for example, `class Teacher(SchoolMember)`). Next, we observe that the `__init__` method of the base class is explicitly called using the `self` variable so that we can initialize the base class part of an instance in the subclass. This is very important to remember- Since we are defining a `__init__` method in `Teacher` and `Student` subclasses, Python does not automatically call the constructor of the base class `SchoolMember`, you have to explicitly call it yourself.

In contrast, if we have not defined an `__init__` method in a subclass, Python will call the constructor of the base class automatically.

While we could treat instances of `Teacher` or `Student` as we would an instance of `SchoolMember` and access the `tell` method of `SchoolMember` by simply typing `Teacher.tell` or `Student.tell`, we instead define another `tell` method in each subclass (using the `tell` method of `SchoolMember` for part of it) to tailor it for that subclass. Because we have done this, when we write `Teacher.tell` Python uses the `tell` method for that subclass vs the superclass. However, if we did not have a `tell` method in the subclass, Python would use the `tell` method in the superclass. Python always starts looking for methods in the actual subclass type first, and if it doesn t find anything, it starts looking at the methods in the subclass s base classes, one by one in the order they are specified in the tuple (here we only have 1 base class, but you can have multiple base classes) in the class definition.

A note on terminology - if more than one class is listed in the inheritance tuple, then it is called **multiple inheritance**.

The `end` parameter is used in the `print` function in the superclass's `tell()` method to print a line and allow the next print to continue on the same line. This is a trick to make `print` not print a `\n` (newline) symbol at the end of the printing.

Summary

We have now explored the various aspects of classes and objects as well as the various terminologies associated with it. We have also seen the benefits and pitfalls of object-oriented programming. Python is highly object-oriented and understanding these concepts carefully will help you a lot in the long run.

Next, we will learn how to deal with input/output and how to access files in Python.