Classes, Instances and Objects

Classes are a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that object to be made (class instances are simply objects that have all the methods and members described in the class definition. The newly created instance has its own place in memory). Python classes provide all the standard features of Object Oriented Programming, which will be covered in the coming tutorials.

Defining a class

In Python, classes are defined using the `class` keyword, as –

```
class Foo(object):

    def bar(self):

        print('This is a class')
```

Here, we are creating a class named `Foo`, which is a type of `object`, which is a built-in type.

The class `object` is like a factory for creating objects. To create an object of type `Foo`, you call it as if it were a function –

```
>>> blank = Foo()

>>> print(blank)

<__main__.Foo object at 0x7fd5d99a4f60>
```

The return value is a reference to a `Foo` object, which we assign to `blank`. Creating a new object is called instantiation, and the object is an instance of the class.

You can also assign values to class instances using a dot notation, as –

```
>>> blank.x = 5

>>> blank.y = 4
```

Here, we are assigning values to named elements of an object. These elements are called attributes. You can print the values of these attributes and pass them as parameters to other functions.

Another important thing to note is that objects are mutable too. To change the state of an object, you can simply make an assignment to one of its attributes. You can also write functions that receive objects as parameters and return/modify those objects.

Class Methods

Methods are essentially functions that are associated with a class. They are semantically the same as functions, but there is one major difference – methods are defined inside a class definition in order to make the relationship between the class and the method explicit.

Let's consider an example. Let's create a class named `Time` that stores the time of day.

```
class Time(object):

    '''Represents the time of day'''
```

Let's create a function that prints this time –

```
def print_time(time):

    print('%.2d:%.2d:%.2d', % (time.hour, time.min, time.sec))
```

To call this function, you have to pass a `Time` object as an argument –

```
>>> start = Time()

>>> start.hour = 9

>>> start.min = 45

>>> start.sec = 00

>>> print_time(start)

09:45:00
```

To make `print_time` a method, we simply move the function into the class definition. Notice the change in indentation –

```
class Time(object):

    '''Represents the time of day'''

    def print_time(time):

        print('%.2d:%.2d:%.2d' % (time.hour, time.min, time.sec))
```

Now, there are two ways to call `print_time`. The first (and less common) way is to use function syntax –

```
>>> Time.print_time(start)

09:45:00
```

The second (and more concise) way is to use method syntax –

```
>>>start.print_time()

09:45:00
```

In this use of dot notation, `print_time` is the name of the method, and start is the object the method is invoked on, which is called the subject. Inside the method, the subject is assigned to the first parameter, so in this case, `start` is assigned to `time`.

By convention, the first parameter of a method is called `self`, so it would be more common to write `print_time` like this –

```
class Time(object):

    def print_time(self):

        print('%.2d:%.2d:%.2d' % (self.hour, self.min, self.sec))

    def convert_time_to_seconds(self):

        self.print_time()

        total_seconds = self.sec

        total_seconds += self.hour * 60 * 60

        total_seconds += self.min * 60

        return total_seconds
```

The method `convert_time_to_seconds()` can also be invoked as `total=start.convert_time_to_seconds()`. Here, the parameter `self` tells the method to take its own instance as a parameter. `self` essentially represents the instance of a class. By using `self`, we can access the attributes and methods of that particular class.

Please note, however, that `self` isn't a keyword – it's just that we use it as a convention. It can be replaced by some other word, and your code would still work –

```
class Time(object):

    def print_time(time):

        print('%.2d:%.2d:%.2d' % (time.hour, time.min, time.sec))

    def convert_time_to_seconds(time):

        time.print_time()

        total_seconds = time.sec

        total_seconds += time.hour * 60 * 60

        total_seconds += time.min * 60

        return total_seconds
```

The reason for this convention is an implicit metaphor – in object-oriented programming, the objects are the active agents. A method invocation like `start.print_time()` says "Hey start! Please print yourself".

This change in perspective might be more polite, but it is not obvious that it is useful. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions, and makes it easier to maintain and reuse code.

The `init` method

The init method (short for "initialization") is a special method that gets invoked when an object is instantiated. Its full name is `__init__` (two underscore characters). An init method for our `Time` class might look like –

```
class Time(object):

    def __init__(self, hrs=0, minutes=0, seconds=0):

        self.hour = hrs

        self.min = minutes

        self.sec = seconds
```

In the above example, the line `self.hour = hrs` stores the value of parameter `hrs` as an attribute of `self`.

This method is what is commonly called a "constructor" in other languages – it allows you to assign values to class members as soon as an object is created.

In this example, the parameters are optional (as they have default values), so if you call `Time` with no arguments, you get the default values.

```
>>> time = Time()

>>> time.print_time()

00:00:00
```

By providing arguments, you can override these default values –

```
>>> time = Time(8, 34, 23)

>>> time.print_time()

08:34:23
```

The `__str__` method

__str__ is a special method, like __init__, that is supposed to return a string representation of an object. For example, here is a str method for Time objects –

```
# inside class Time

def __str__(self):

  return '%.2d:%.2d:%.2d' % (self.hour, self.min, self.sec)
```

When you print an object, Python invokes the str method –

```
>>> time = Time(9, 45)

>>> print(time)

09:45:00
```

Operator Overloading

By defining other special methods, you can specify the behavior of operators on user-defined types. For example, if you define a method named __add__ for the Time class, you can use the + operator on Time objects.

Changing the behavior of an operator so that it works with user-defined data types is called operator overloading. For every operator in Python there is a corresponding special method. For more details, check out the Python docs.

Type-Based Dispatch

The built-in function isinstance takes a value and a class object, and returns True if the value is an instance of the class. Here's an example –

```
>>> other = Time()

>>> isinstance(other, Time)

True
```

This can be used to check the type of parameters received, and dispatch the computation based on the type of the arguments. This operation is called a type-based dispatch.