Classes in Python

Let us start with the basics of a class and play around with the different features of it in the next few chapters.

The simplest definition of a class in Python is

```
class Empty:
    pass
```

The above two lines with just three words creates a class.

What can we do with this class? A lot

We can create instances of this class, assign it to other names and compare them

```
>>> a = Empty()
>>> b = Empty()
>>> c = a
>>> a == c
True
>>> a == b
False
```

We can now add attributes and methods to this class. Yes. You read it right. You can add attributes and methods to a Python class at runtime. We will see later that you can even create a class at runtime.

Here is how you add an attribute to a python class at runtime

```
>>> Empty.greeting = 'An empty can do many things'
```

This attribute can then be accessed with class or instance references

```
>>> Empty.greeting
'An empty can do many things'
>>> a.greeting
'An empty can do many things'
>>> b.greeting
'An empty can do many things'
>>> type(a).greeting
'An empty can do many things'
```

This brings us to the question as to what type of attribute is greeting for the Empty class?

It is a static attribute. There is one copy of this attribute which is shared by the class and all its instances. Changing the value of this attribute at class level reflects when it is accessed through any of its instances

```
>>> Empty.greeting = 'There is a limit to everything'
>>> a.greeting
'There is a limit to everything'
>>> b.greeting
'There is a limit to everything'
```

However, the converse is not true. Setting a value for this attribute with instance reference does not change the value of this attribute in other instances or at class level

```
>>> a.greeting = 'And I am not the one'
>>> a.greeting
'And I am not the one'
>>> Empty.greeting
'There is a limit to everything'
>>> b.greeting
'There is a limit to everything'
```

What happens is, setting value for an attribute at instance level creates a new attribute for that instance with that same name.

Referencing an attribute through an object, fetches the value of the attribute attached to that object. Only when the attribute does not exist in that object, it is looked up at the class (type) of the object, which returns the value set at the class level.

We can remove an attribute attached to an object using the del statement. Deleting the 'greeting' attribute of instance 'a', deletes the instance level attribute. Now accessing 'a.greeting' returns the value of class level greeting attribute

```
>>> del a.greeting
>>> a.greeting
'There is a limit to everything'
```

In a similar fashion, methods can be added to a class at runtime. Let us create a function which returns a greeting message and assign this function to 'Empty' class

```
def sayHello():
    return 'Hello Caller'
```

```
>>> Empty.greet = sayHello
>>> Empty.greet()
'Hello Caller'
```

The above code assigns 'sayHello' function to the 'greet' variable of the 'Empty' class. The 'greet' can then be invoked as a method on 'Empty' class.

Now let us see what happens when we invoke the greet method on the objects we created for the 'Empty' class

```
>>> a.greet()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sayHello() takes 0 positional arguments but 1 was
given
>>>
```

Calling this method through an instance of the 'Empty' class throws an error. Two things to note from the error message thrown.

One the method name remains as 'sayHello' and does not get shown as 'greet'. 'greet' is only an alias to the function 'sayHello' that was defined separately. We can check this by seeing the IDs of sayHello and greet

```
>>> id(sayHello)
54790304
>>> id(Empty.greet)
54790304
>>>
```

Their IDs are same indicating that both point to the same object which is the function 'sayHello'. Now what happens when we delete the 'sayHello' function

```
>>> del(sayHello)
>>> Empty.greet()
'Hello Caller'
>>> a.greet()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sayHello() takes 0 positional arguments but 1 was
given
>>>
```

Even after deleting the 'sayHello', the function continues to remain and is accessible through 'Empty.greet'. The names 'sayHello' and 'greet' are just two aliases (references) to the function object defined and this function object remains as long as there remains at least one active reference to that object.

Now the second point to note from the error message thrown when accessing the function 'greet' through an instance of the 'Empty' object. The message reads 'sayHello() takes 0 positional arguments but 1 was given'. We clearly are not passing in any parameter when we call the 'greet' method through the object instance a.

Python automatically passes in the instance object reference as the first argument to the method invoked. That is the reason why all the instance methods of a class in python have 'self' as the first parameter. 'self' refers to itself, the instance object through which the method is invoked.

Note: The name 'self' itself is only a convention and can be any name.

Now let us add the 'self' parameter to the method and see what happens. We will define the 'myMessage' function for this purpose.

```
# myMessage.py
def myMessage(self):
    return 'Congrats... I am ' + str(id(self))
```

Now we will assign this function as 'greet' in 'Empty' class and access it through its instance 'a'

```
>>> Empty.greet = myMessage
>>> a.greet()
'Congrats... I am 93237936'
>>> id(a)
93237936
>>>
```

We can see from above output that the object instance 'a' and the object reference pointed to by the first parameter 'self' of the myMessage method are the same.

So all instance methods in Python will have the first parameter as 'self' which will be the reference to the object that invoked it.

Can we call the 'myMessage' function through class reference? Yes. But when invoked through class, the first parameter does not get passed automatically.

```
>>> Empty.greet()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: myMessage() missing 1 required positional
argument: 'self'
>>>
```

The error message says that the 'self' argument that is required to be passed is missing. We can call Empty.greet by passing in the first parameter

```
>>> Empty.greet('A new string')
'Congrats... I am 55030192'
>>> id(Empty)
93258720
>>> id(a)
93237936
>>> Empty.greet(a)
'Congrats... I am 93237936'
>>>
```

The above code shows that invoking a.greet() is the same as invoking Empty.greet(a)

In the above examples, we started with an Empty class – which had nothing in it. We went on to add attributes and methods to it, created instances of it and saw different ways of accessing the attributes and methods added. The approach we took is more ad hoc, trying to understand its behavior. Next we will look at the standard approach to defining classes

* All the source files used in the samples are available under the src directory of the current path