Classes and objects in python

In Python, we define a class using the keyword class which has following syntax

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

A class is a code template for creating objects. Objects have member variables and have behaviour associated with them.

class uses variables to store the data and to define the functionalities, we have to defined functions inside the class these functions are known as methods.

When you define methods, you will need to always provide the first argument to the method with a self keyword.

```
def method-name(self, argument):
     'Optional method documentation string'
     method-body
```

An object is created using the constructor of the class. This object will then be called the instance of the class. In Python we create instances in the following manner

```
Instance-name = class-name(arguments)
```

To access attributes and methods of the class using object we have to use . (dot) operator whenever an object calls its method, the object itself is passed as the first argument.

A specialized method __init__() can be defined inside the class that works as a constructor of the class and that method is called automatically whenever class is instantiated.

```
def __init__(self):
     method-body
```

An Example

```
class MyClass:
    "This is my first class"
    a = 10   # this is a class variable, have a common copy, shared among all
objects

    def __init__(self):
        print("I am equivalent to the constructor")

    def func(self):
        print("Inside func")

print("a =", MyClass.a)
obj = MyClass()
obj.func()
```

The above code will produce following output

```
a = 10
I am equivalent to the constructor
Inside func
```

Methods may call other methods by using method attributes of the self argument, like in the above example if we want to call func() from constructor then we need to write following code in the example

```
self.func()
```

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.

So for the class that we have defined above we are writing the following code

```
print("MyClass.__dict__",MyClass.__dict__)
print("MyClass.__doc__",MyClass.__doc__)
print("MyClass.__name__",MyClass.__name__)
print("MyClass.__module__",MyClass.__module__)
print("MyClass.__bases__",MyClass.__bases__)
```

The code will print following output

```
MyClass.__dict__ {'__module__': '__main__', '__doc__': 'This is my first
class', 'a': 10, '__init__': <function MyClass.__init__ at 0x000000F66A871378>,
'func': <function MyClass.func at 0x000000F66A8712F0>, '__dict__': <attribute
'__dict__' of 'MyClass' objects>, '__weakref__': <attribute '__weakref__' of
'MyClass' objects>}
MyClass.__doc__ This is my first class
MyClass.__name__ MyClass
MyClass.__module__ __main__
MyClass.__bases__ (<class 'object'>,)
```

Inheritance in Python

Python supports inheritance, it even supports multiple inheritance. A class can inherit attributes and behaviour methods from another class, called the superclass. A class which inherits from a superclass is called a subclass, also called heir class or child class.

```
class DerivedClassName(BaseClassName):
      .
      .
```

The syntax for multiple inheritance is

```
class DerivedClassName(BaseClassName1, BaseClassName1):
      .
      .
```

- issubclass(sub, sup): It is a boolean function returns true if the given subclass sub is indeed a subclass of the superclass sup.

An Example:

```
class Person:
      def __init__(self, first, last):
            self.firstname = first
            self.lastname = last

      def Name(self):
            return self.firstname + " " + self.lastname

class Employee(Person):
      def __init__(self, first, last, staffnum):
            Person.__init__(self,first, last)
```

```
            self.staffnumber = staffnum

      def GetEmployee(self):
            return self.Name() + ", " +  self.staffnumber

x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")

print(x.Name())
print(y.GetEmployee())
```

The above code will produce following output

```
Marge Simpson
Homer Simpson, 1007
```

One important point to remember is that overloading is not available in python although default arguments can be used to make a function work for different arguments.

Method overriding is available in python. To access method of super class in sub class that has been overridden, we have to use super() inside subclass.

An Example

```
class Person:
      def __init__(self, first, last):
            self.firstname = first
            self.lastname = last

      def Name(self):
            print("Inside Person.Name()")
            return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        Person.__init__(self, first, last)
        self.staffnumber = staffnum

    def Name(self):
        detail = super().Name() + ", " +  self.staffnumber
        print("Inside Employee.Name()")
        return detail

y = Employee("Homer", "Simpson", "1007")
print(y.Name())
```

The above code will produce following output

```
Inside Person.Name()
Inside Employee.Name()
Homer Simpson, 1007
```

Operator overloading

As we know that the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings means the same operator has different meaning according to the data type, This is what operator overloading is. It let an operator to have different meaning according to the context.

In python all operators has a specific method for itself. Whenever we want to overload an operator then we have provide definition of the method associated with that. Before moving forward let us take a look at the table that describe operator and respective function

Operator	Expression	Internally

Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Less than	p1 < p2	__lt__(self, other)
Less than or equal to	p1 <= p2	__le__(self, other)
Equal to	p1 == p1	__eq__(self, other)
Not equal to	p1 != p2	__ne__(self, other)
Greater than	p1 > p2	__gt__(self, other)
Greater than or equal to	p1 >= p2	__ge__(self, other)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()
[index]	__getitem__(self, index)	Index operator
In	__contains__(self, value)	Check membership
len	__len__(self)	The number of elements
Str	__str__(self)	The string representation
Del	__del(self)	The delete operator

An Example

```
class Point:
    def __init__(self, x=0, y=0):
        print("Inside __init__")
        self.x = x
        self.y = y

    def __str__(self):
        print("Inside overloaded __str__ operator")
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        print("Inside overloaded + operator")
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(2, 3)
p2 = Point(-1, 2)
print(p1 + p2)
```

The above code will produce following output

```
Inside __init__
Inside __init__
Inside overloaded + operator
Inside __init__
```

```
Inside overloaded __str__ operator
(1,5)
```