Python Objects and Class

In this article, you'll learn about the core functionality of Python, Python objects and classes. You'll learn what a class is, how to create it and use it in your program.

Table of Contents

What are classes and objects in Python?

Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stress on objects.

Object is simply a collection of data (variables) and methods (functions) that act on those data. And, class is a blueprint for the object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

Defining a Class in Python

Like function definitions begin with the keyword def, in Python, we define a class using the keyword class.

The first string is called docstring and has a brief description about the class. Although not mandatory, this is recommended.

Here is a simple class definition.

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores (__). For example, __doc__ gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class MyClass:
    "This is my second class"
    a = 10
    def func(self):
        print('Hello')
# Output: 10
print(MyClass.a)
# Output: <function MyClass.func at 0x0000000003079BF8>
print(MyClass.func)
# Output: 'This is my second class'
print(MyClass.__doc__)
```

When you run the program, the output will be:

```
10

<function 0x7feaa932eae8="" at="" myclass.func="">

This is my second class
```

Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

```
>>> ob = MyClass()
```

This will create a new instance object named `ob`. We can access attributes of objects using the object name prefix.

Attributes may be data or method. Method of an object are corresponding functions of that class. Any function object that is a class attribute defines a method for objects of that class.

This means to say, since `MyClass.func` is a function object (attribute of class), `ob.func` will be a method object.

```python
class MyClass:
    "This is my second class"
    a = 10
    def func(self):
        print('Hello')
# create a new MyClass
ob = MyClass()
# Output: <function MyClass.func at 0x000000000335B0D0>
print(MyClass.func)
# Output: <bound method MyClass.func of <__main__.MyClass object at
0x000000000332DEF0>>
print(ob.func)
# Calling function func()
# Output: Hello
ob.func()
```

You may have noticed the `self` parameter in function definition inside the class but, we called the method simply as `ob.func()` without any arguments. It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, `ob.func()` translates into `MyClass.func(ob)`.

In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

For these reasons, the first argument of the function in class must be the object itself. This is conventionally called `self`. It can be named otherwise but we highly recommend to follow the convention.

Now you must be familiar with class object, instance object, function object, method object and their differences.

Constructors in Python

Class functions that begins with double underscore (__) are called special functions as they have special meaning.

Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

```
class ComplexNumber:
    def __init__(self,r = 0,i = 0):
        self.real = r
        self.imag = i
    def getData(self):
        print("{0}+{1}j".format(self.real,self.imag))
# Create a new ComplexNumber object
c1 = ComplexNumber(2,3)
# Call getData() function
# Output: 2+3j
c1.getData()
# Create another ComplexNumber object
# and create a new attribute 'attr'
c2 = ComplexNumber(5)
c2.attr = 10
# Output: (5, 0, 10)
print((c2.real, c2.imag, c2.attr))
# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no attribute 'attr'
c1.attr
```

In the above example, we define a new class to represent complex numbers. It has two functions, `__init__()` to initialize the variables (defaults to zero) and `getData()` to display the number properly.

An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute `attr` for object `c2` and we read it as well. But this did not create that attribute for object `c1`.

Deleting Attributes and Objects

Any attribute of an object can be deleted anytime, using the del statement. Try the following on the Python shell to see the output.

```
>>> c1 = ComplexNumber(2,3)
>>> del c1.imag
>>> c1.getData()
```

```
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'


>>> del ComplexNumber.getData
>>> c1.getData()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'getData'
```

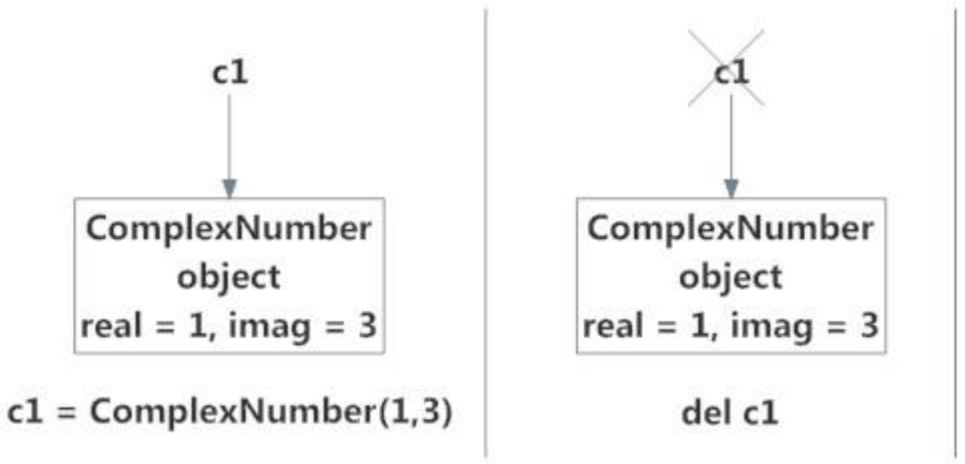
We can even delete the object itself, using the del statement.

```
>>> c1 = ComplexNumber(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not defined
```

Actually, it is more complicated than that. When we do `c1 = ComplexNumber(1,3)`, a new instance object is created in memory and the name `c1` binds with it.

On the command del c1, this binding is removed and the name `c1` is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.

This automatic destruction of unreferenced objects in Python is also called garbage collection.


c1
ComplexNumber object real = 1, imag = 3
c1 = ComplexNumber(1,3)
c1
ComplexNumber object real = 1, imag = 3
del c1

Python Inheritance

Inheritance enable us to define a class that takes all the functionality from parent class and allows us to add more. In this article, you will learn to use inheritance in Python.

Table of Contents

What is Inheritance?

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new class with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Python Inheritance Syntax

```
class BaseClass:

  Body of base class

class DerivedClass(BaseClass):

  Body of derived class
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called `Polygon`defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i
in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides, `n` and magnitude of each side as a list, `sides`.

Method `inputSides()` takes in magnitude of each side and similarly, `dispSides()` will display these properly.

A triangle is a polygon with 3 sides. So, we can created a class called `Triangle` which inherits from `Polygon`. This makes all the attributes available in class `Polygon` readily available in `Triangle`. We don't need to define them again (code re-usability). `Triangle` is defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, class `Triangle` has a new method `findArea()` to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()


>>> t.inputSides()
Enter side 1 : 3
 Enter side 2 : 5
Enter side 3 : 4


>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0


>>> t.findArea()
The area of the triangle is 6.00
```

We can see that, even though we did not define methods like `inputSides()` or `dispSides()`for class `Triangle`, we were able to use them.

If an attribute is not found in the class, search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Method Overriding in Python

In the above example, notice that `__init__()` method was defined in both classes, `Triangle`as well `Polygon`. When this happens, the method in the derived class overrides that in the base class. This is to say, `__init__()` in `Triangle` gets preference over the same in `Polygon`.

Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling `Polygon.__init__()` from `__init__()` in `Triangle`).

A better option would be to use the built-in function `super()`.
So, `super().__init__(3)` is equivalent to `Polygon.__init__(self,3)` and is preferred.
You can learn more about the super() function in Python.

Two built-in functions `isinstance()` and `issubclass()` are used to check inheritances.
Function `isinstance()` returns `True` if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class `object`.

```
>>> isinstance(t,Triangle)
True


>>> isinstance(t,Polygon)
True


>>> isinstance(t,int)
False


>>> isinstance(t,object)
True
```

Similarly, `issubclass()` is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False


>>> issubclass(Triangle,Polygon)
True


>>> issubclass(bool,int)
True
```

Python Multiple Inheritance

In this article, you'll learn what is multiple inheritance in Python and how to use it in your program. You'll also learn about multilevel inheritance and the method resolution order.

Table of Contents

Multiple Inheritance in Python

Like C++, a class can be derived from more than one base classes in Python. This is called multiple inheritance.

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

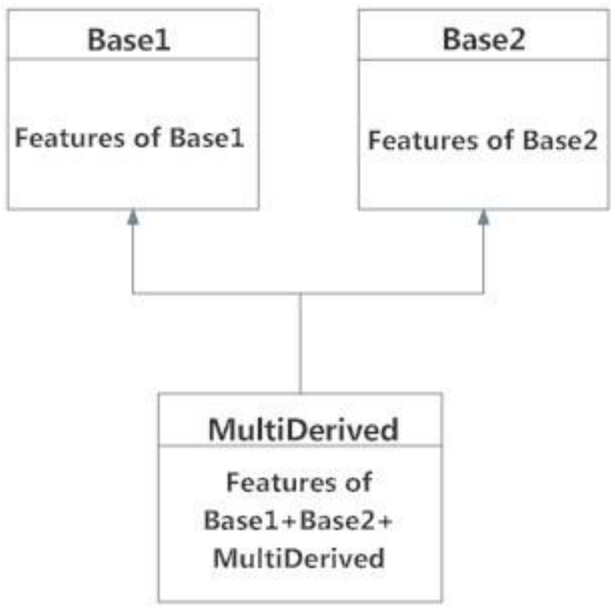
Example

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

Here, `MultiDerived` is derived from classes `Base1` and `Base2`.



The class `MultiDerived` inherits from both `Base1` and `Base2`.

Multilevel Inheritance in Python

On the other hand, we can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class.

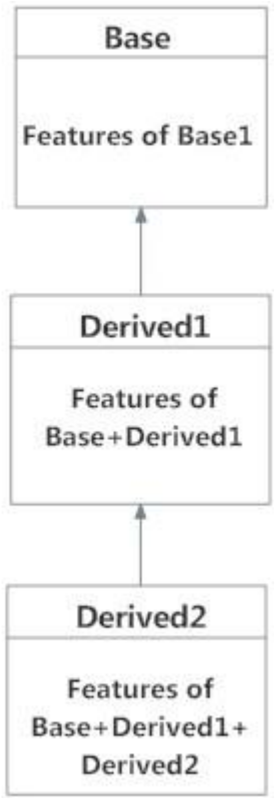
An example with corresponding visualization is given below.

```
class Base:
    pass


class Derived1(Base):
    pass


class Derived2(Derived1):
    pass
```

Here, `Derived1` is derived from `Base`, and `Derived2` is derived from `Derived1`.


Base
Features of Base1
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

Method Resolution Order in Python

Every class in Python is derived from the class `object`. It is the most base type in Python.

So technically, all other class, either built-in or user-defines, are derived classes and all objects are instances of `object` class.

```
# Output: True
print(issubclass(list,object))
# Output: True
print(isinstance(5.5,object))
# Output: True
print(isinstance("Hello",object))
```

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching same class twice.

So, in the above example of `MultiDerived` class the search order is [`MultiDerived`, `Base1`, `Base2`, `object`]. This order is also called linearization of `MultiDerived` class and the set of rules used to find this order is called **Method**

Resolution Order (MRO).

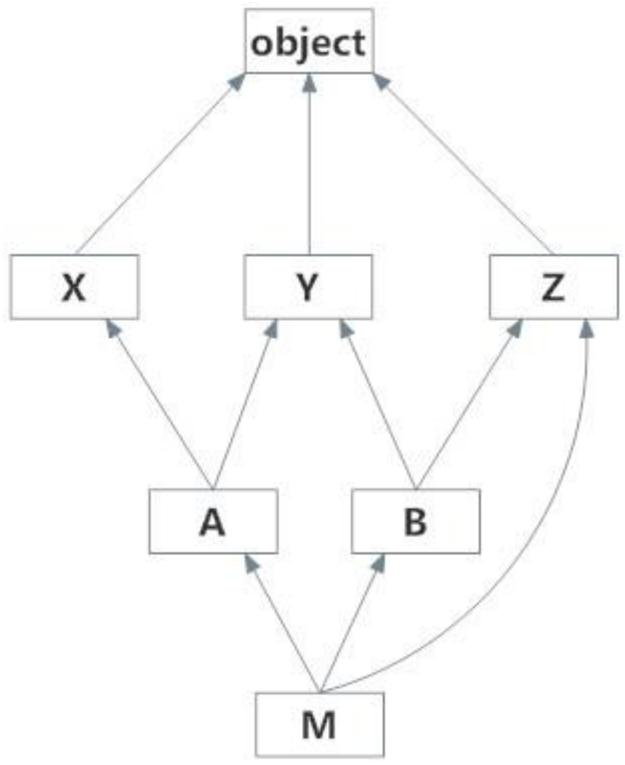
MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents and in case of multiple parents, the order is same as tuple of base classes.

MRO of a class can be viewed as the `__mro__` attribute or `mro()` method. The former returns a tuple while latter returns a list.

```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>)


>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>]
```

Here is a little more complex multiple inheritance example and its visualization along with the MRO.



```
class X: pass
class Y: pass
class Z: pass
class A(X,Y): pass
class B(Y,Z): pass
class M(B,A,Z): pass
# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
# <class '__main__.A'>, <class '__main__.X'>,
# <class '__main__.Y'>, <class '__main__.Z'>,
# <class 'object'>]
print(M.mro())
```

Refer to this, for further discussion on MRO and to know the actual algorithm how it is calculated.

Python Operator Overloading

You can change the meaning of an operator in Python depending upon the operands used. This practice is known as operating overloading.

Table of Contents

What is operator overloading in Python?

Python operators work for built-in classes. But same operator behaves differently with different types. For example, the `+` operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.

This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

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

Now, run the code and try to add two points in Python shell.

```
>>> p1 = Point(2,3)

>>> p2 = Point(-1,2)

>>> p1 + p2

Traceback (most recent call last):

...

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Whoa! That's a lot of complains. `TypeError` was raised since Python didn't know how to add two `Point` objects together.

However, the good news is that we can teach this to Python through operator overloading. But first, let's get a notion about special functions.

Special Functions in Python

Class functions that begins with double underscore __ are called special functions in Python. This is because, well, they are not ordinary. The `__init__()` function we defined above, is one of them. It gets called every time we create a new object of that class. There are a ton of special functions in Python.

Using special functions, we can make our class compatible with built-in functions.

```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

That did not print well. But if we define `__str__()` method in our class, we can control how it gets printed. So, let's add this to our class.

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

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
```

Run

Powered by DataCamp

Now let's try the `print()` function again.

```
>>> p1 = Point(2,3)
>>> print(p1)
(2,3)
```

That's better. Turns out, that this same method is invoked when we use the built-in function `str()` or `format()`.

```
>>> str(p1)
'(2,3)'

>>> format(p1)
'(2,3)'
```

So, when you do `str(p1)` or `format(p1)`, Python is internally doing `p1.__str__()`. Hence the name, special functions.

Ok, now back to operator overloading.

Overloading the + Operator in Python

To overload the `+` sign, we will need to implement `__add__()` function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is sensible to return a `Point` object of the coordinate sum.

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

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

    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
```

Now let's try that addition again.

```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> print(p1 + p2)
(1,5)
```

What actually happens is that, when you do `p1 + p2`, Python will call `p1.__add__(p2)` which in turn is `Point.__add__(p1,p2)`. Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator Overloading Special Functions in Python

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)
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__()

Overloading Comparison Operators in Python

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.

Suppose, we wanted to implement the less than symbol `<` symbol in our `Point` class.

Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

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

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

    def __lt__(self,other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag
```

Try these sample runs in Python shell.

```
>>> Point(1,1) < Point(-2,-3)
True

>>> Point(1,1) < Point(0.5,-0.2)
False

>>> Point(1,1) < Point(1,1)
False
```

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Comparision Operator Overloading in Python

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)