Python Objects and Classes

programiz.com/python-programming/class

Join our newsletter for the latest updates.

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

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

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

We can think of a 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 house's blueprint, 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 def keyword in Python, class definitions begin with a class keyword.

The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly 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 Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# Output: 10
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)

# Output: "This is a person class"
print(Person.__doc__)
```

Output

```
10
<function Person.greet at 0x7fc78c6e8160>
This is a person 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.

```
>>> harry = Person()
```

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

Attributes may be data or method. Methods of an object are corresponding functions of that class.

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

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# create a new object of Person class
harry = Person()

# Output: <function Person.greet>
print(Person.greet)

# Output: <bound method Person.greet of <__main__.Person object>>
print(harry.greet)

# Calling object's greet() method
# Output: Hello
harry.greet()
```

Output

```
<function Person.greet at 0x7fd288e4e160>
<bound method Person.greet of <__main__.Person object at 0x7fd288e9fa30>>
Hello
```

You may have noticed the `self` parameter in function definition inside the class but we called the method simply as `harry.greet()` 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, `harry.greet()` translates into `Person.greet(harry)`.

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 begin 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 get_data(self):
        print(f'{self.real}+{self.imag}j')


# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)

# Call get_data() method
# Output: 2+3j
num1.get_data()

# Create another ComplexNumber object
# and create a new attribute 'attr'
num2 = ComplexNumber(5)
num2.attr = 10

# Output: (5, 0, 10)
print((num2.real, num2.imag, num2.attr))

# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no attribute 'attr'
print(num1.attr)
```

Output

```
2+3j
(5, 0, 10)
Traceback (most recent call last):
  File "<string>", line 27, in <module>
    print(num1.attr)
AttributeError: 'ComplexNumber' object has no attribute 'attr'
```

In the above example, we defined a new class to represent complex numbers. It has two functions, `__init__()` to initialize the variables (defaults to zero) and `get_data()` 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 *num2* and read it as well. But this does not create that attribute for object *num1*.

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.

```
>>> num1 = ComplexNumber(2,3)
>>> del num1.imag
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.get_data
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'get_data'
```

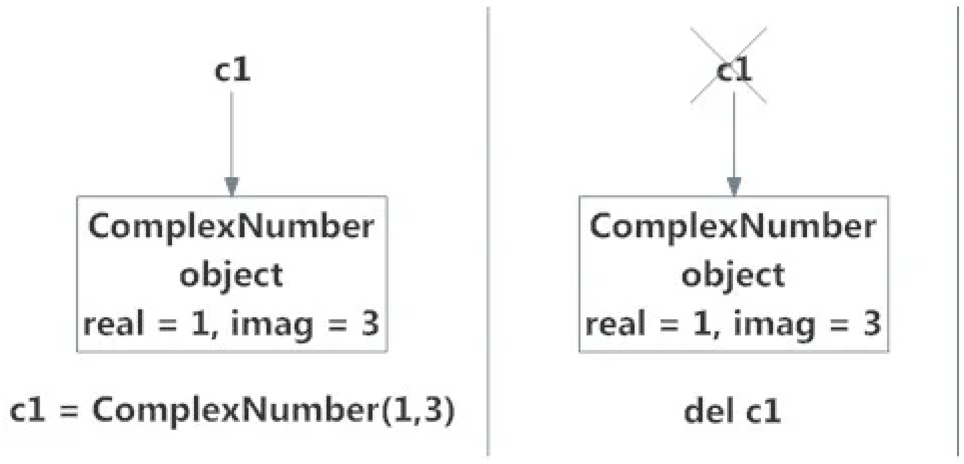
We can even delete the object itself, using the del statement.

```
>>> 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

Deleting objects in Python removes the name binding