

# Chapter 7 : Objects and Classes

Aviral Janveja

## 1 Object

Throughout this course in Python, we have come across various examples of data like numbers, strings, booleans and so on. All of these instances of data belong to a particular data-type like integer, float or string.

In Python, the term object basically refers to any piece of data. Whether we take instances of integer data, float data or string data, they are all essentially objects. Therefore, the term **object** serves a dual purpose in Python language :

1. It refers to an instance of a particular data-type. For example,  $x = 5$ . Here 5 is an object of type integer.
2. Object is also the base of all data-types. For example, the integer data-type itself is built on top of the object data-type.

This base object data-type provides core functionality in Python language, such as the ability to assign values to variables. This is why it is often said that **everything in Python is an object** and every object has a type. The type of an object defines how it is represented internally within the Python language and the ways in which we can interact with it. For example, list objects are always defined using square brackets [ ] and we can manipulate them using various built-in methods such as insert, append, remove and sort.

## 2 Class

The next question is : how are these data-types defined behind the scenes and can we define our own type ?

**Answer** : Yes, we can define our own types through **classes**. Defining a class is like defining a blue print for your own data-type, similar to integers, floats and lists. The class definition looks like this :

```
class class_name:  
    # Define attributes here
```

We start with the **class** keyword followed by the **name** of our class. Inside the class, we define its attributes which are the data attributes and the procedural attributes that belong to the class.

## 2.1 Data

Data attributes define the data representation of this particular type. For example, a two dimensional coordinate object will be made up of two numbers, in a bracket, separated by a comma.

Coordinate object = (3,5)

One value for the x-coordinate and one value for the y-coordinate. We can further decide on whether these two numbers will be of type int or type float.

## 2.2 Methods

Procedural attributes, better known as methods are functions, that only work with this particular type. For example, you can define a distance method between two coordinate objects, but that method will have no meaning for list objects.

# 3 Defining Our Own Type : Class Coordinate

Let us continue with the class definition above and implement our own class.

```
class Coordinate:
    # Data and Methods
```

## 3.1 Special Method `__init__`

First, we define a method called `__init__` within our class. In Python `__init__` is a special method also known as the constructor. It is automatically called when a new object of the class is created. The `__init__` method allows you to **initialize** the data attributes of an object as soon as that object is created.

```
class Coordinate:

    def __init__(self, x, y):
```

While defining this method, a default parameter named **self** is always used. The **self** parameter refers to the object of the class itself and is used for assigning values to the data attributes of an object. The two underscores on each side of its name imply that the method is invoked automatically and used internally by Python, without needing to be called explicitly.

Any other parameters beyond **self** can be added, just like a normal function. In this particular case, we are going to initialize a coordinate object with two values, one for the x-coordinate and one for the y-coordinate.

```
class Coordinate:

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

Inside the `__init__` method we have two assignments as shown above. It basically says that the "x data attribute of the coordinate object will be assigned to whatever value of x was passed in, while creating that object". Same for the y data attribute.

## 3.2 Creating Coordinate Objects

We have created a simple class above and now we are ready to start creating objects of type `Coordinate` as follows :

```
class Coordinate:

    def __init__(self, x, y):
        self.x = x
        self.y = y

# Creating an object of type Coordinate
c = Coordinate(3, 4)
```

The following line of code calls the `init` method with `x = 3` and `y = 4` :

```
c = Coordinate(3, 4)
```

Notice, when we are creating an object here, we are only giving it 2 parameters, whereas in the `init` method, we have 3 parameters. This is okay, because implicitly, Python uses the `self` parameter to refer to the object `c` by default.

Below, we create another `Coordinate` object named **origin** whose values for `x` and `y` are both zero. So now, we have two `Coordinate` objects and we can access their data attributes using the dot-notation as shown below :

```
class Coordinate:

    def __init__(self, x, y):
        self.x = x
        self.y = y

c = Coordinate(3, 4)
origin = Coordinate(0, 0)

# Accessing data attributes using dot-notation
print(c.x) # Output = 3
print(origin.x) # Output = 0
```

The above line prints the `x`-Coordinates of both objects, which are 3 and 0 respectively.

## 3.3 Distance Method

Having defined the basic structure of our class above, let us add a distance method to our class definition.

Similar to `init`, this function will start with the default **self** parameter, used to refer to any object of the class. The next parameter called **other**, will refer to the other `Coordinate` object from which we want to calculate the distance. And inside, we are just going to implement the euclidean distance formula and return it, as shown below :

```

class Coordinate:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5

```

Notice that inside the method, we have used **self.x** notation. This is because, we want to find the difference between the x-values of the given objects, in order to calculate the euclidean distance between them. Hence, it is important to **note** that when working with classes, you always need to think about whose data attribute you are trying to access. As shown by the **self.x** and **other.x** notation used in the above example.

We have defined the distance method, now let us use it :

```

c = Coordinate(3, 4)
origin = Coordinate(0, 0)
print(c.distance(origin)) # Output = 5.0

```

Following the class definition, we create two coordinate objects as shown above. Then we use the dot-notation to call the distance method on one of the objects and give the other object as parameter in the parenthesis. We have been doing this so far already for other objects, for example, `list1.append` for lists and so on.

### 3.4 Special Method `__str__`

We can choose how our class object will be printed out with the help of `__str__` method . For example, if we try to print out the class object `c`, currently we get the following output :

```

print(c)

# Output : <__main__.Coordinate object at
0x0000016D6C17BFD0>

```

It basically says that, `c` is an object of type `Coordinate` at this particular memory location. This is quite uninformative. What we would want instead, is to output the `Coordinate` values of `c`. In order to achieve that, we need to define our own special method `__str__` that tells Python what to do, when we call `print` on an object of this type.

This method only takes the `self` parameter because you are just calling `print` on the object itself. It returns a string which outputs the `x` and `y` values of the `Coordinate` object within parenthesis, separated by a comma.

```

class Coordinate:

    def __init__(self, x, y):

```

```
self.x = x
self.y = y

def distance(self, other):
    x_diff_sq = (self.x-other.x)**2
    y_diff_sq = (self.y-other.y)**2
    return (x_diff_sq + y_diff_sq)**0.5

def __str__(self):
    return "(" + str(self.x) + ","
    + str(self.y) + ")"

print(c)    # Output : (3,4)
```