Chapter 7 :
Classes and Objects

Aviral Janveja

1 Object

Throughout this course, we have come across several data-types like integers, floats and strings. Any piece of data is basically an **object** in Python. In fact, this term serves a dual purpose in Python language :

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

This base object-type provides core functionality in the Python language, such as the ability to assign values to variables and so on. That 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 one can interact with it. For example, list objects are always defined using square brackets [ ] and we can manipulate them using functions such as append, pop and sorted.

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 and the methods.

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, within parenthesis, separated by a comma :

(3, 4)

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

Methods are functions that only work with this particular type. For example, you may define a distance method to calculate the euclidean distance, between two coordinate objects, but that method will naturally have no meaning for list objects.

3 Defining Our Own Type : Class Coordinate

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

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

3.1 Special Method __init__

First, we define a special method called __init__ within our class which is 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. They are basically conveying that - "The value of x that was passed in while initializing the object will be assigned to the x data attribute of the coordinate 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 lines print 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 method will also have the default **self** parameter, used to refer to the objects of this class. The next parameter called **other**, will refer to the other Coordinate object from which the distance will be calculated. 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 methods, we have used the **self.x** notation. This is because, we want to refer to the x-values of the created 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 example above.

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 an argument inside the parenthesis. We have been doing this already for other data-types, for example list1.extend(list2) for lists and so on.

3.4 Special Method __str__

Further, we can customize the **string representation** of our class objects with the help of __str__ method. Currently if we try to print out the class object **c**, by default 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 location in memory.

This is quite uninformative. What we might want instead, is to output the value of coordinate **c** as a string representation, which could be helpful while debugging for example. 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 :

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


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

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

The entire class that we have defined in the above section can be found here - Class_Coordinate

4 Encapsulation

In object oriented programming, encapsulation is achieved through the class definition, where data attributes and methods are enveloped together. This approach hides away the internal implementation details, while providing users with a simplified interface to interact with the class data and methods. Users need not really need to know how a class is implemented behind the scenes. They should just know, how to use the class.

Hence, an important aspect of encapsulation is **information hiding**. We need to make sure that class data, especially the data attributes are accessed and modified in a controlled manner, in order to prevent errors and unexpected changes.

4.1 Access Modifiers

Access modifiers are used to control the accessibility of class attributes. They define how variables and methods of a class can be accessed or modified by other parts of the program. In other object oriented programming languages like C++ and Java, information hiding is strictly enforced via access modifiers such as public, protected and private. Python on the other hand, supports information hiding through conventions and programming practices rather than enforced access modifiers :

- **Public** : Attributes accessible from anywhere, default in Python.
- **Protected** : Attributes should only be accessed within the class and the subclasses. Defined using a single underscore like **_age**.
- **Private** : Attributes should only be accessed within the class. Defined using double underscores before the variable name like **__age**.

Python applies internal name modification to private attributes, which changes the variable name to the following form in the background :

_ClassName__VariableName
For example : **_Animal__age**

This makes it harder, but not impossible to access or modify the variable from outside the class. Although these access restrictions are not enforced strictly as in other languages like Java, they allow developers to communicate the intended use of variables and methods.

So, how do we expose class attributes to users, while still controlling how they are accessed and modified? We can achieve that by adding **getter** and **setter** methods to our class. They allows us to control access and apply validation while modifying class attributes as shown in the following sections.

4.2 Getter

The getter method helps us **access** the data attributes in a controlled manner. Let us understand this through the Animal class example :

```
class Animal:
    def __init__(self, age):
        self.age = age
```

Consider the following case, where the class author wants to change the internal implementation. Let us say, they change the data attribute name from age to years :

```
class Animal:
    def __init__(self, age):
        self.years = age  # variable name changed

leo = Animal(3)
print(leo.age)  # Gives an error
```

In the above example, trying to access the age attribute directly from outside the class leads to an error, since the variable name has been changed internally. In order to prevent such errors, we can implement the getter method in our class as shown below :

```
class Animal:
    def __init__(self, age):
        self.years = age  # variable name changed

    def get_age(self):
        return self.years  # Getter modified accordingly

leo = Animal(3)
print(leo.get_age())  # No Error this time
```

As visible from the above example, using the getter method **get_age** prevents that error, since the class author has suitably edited the getter method to work correctly alongside the modified variable name. Hence, users accessing the data attribute from outside the class need not worry about it.

4.3 Setter

The setter method allows us to control the **modification** of data attributes. For example, we would like the animal age to be an integer value only :

```
class Animal:
    def __init__(self, age):
        self._age = age  # Protected variable

    def get_age(self):
        return self._age

leo = Animal("old")
print(leo._age)  # Misleading result : old
```

As shown above, making the age variable protected couldn't prevent misleading results, such as animal age being a string. Let us try making it private instead :

```
class Animal:
    def __init__(self, age):
        self.__age = age  # Private variable

    def get_age(self):
        return self.__age

leo = Animal("old")
print(leo.__age)  # Raises Error
```

Now, Python will raise an error if someone tries to access the attribute, let alone modify it. But this still doesn't address the main issue. How do we expose the **age** attribute to users, but still control how it is modified? For example, we want age to be an integer value only. The setter method will help us accomplish that :

```
class Animal:
    def __init__(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, new_age):
        if isinstance(new_age, int):
            self.__age = new_age
        else:
            raise TypeError("Animal age must
            be an integer")


leo = Animal(2)
print(leo.get_age())  # Output : 2

leo.set_age(3)
print(leo.get_age())  # Output : 3

leo.set_age("old")
print(leo.get_age())  # Raises Type Error as required
```

Before setting a new value, set_age ensures that it is an integer. In this way, you create a private attribute __age and still let users access and modify it in a controlled manner using get_age and set_age methods. The code is available here : Getter_Setter

4.4 Property Decorator

Python provides a more concise and readable way to use getters and setters through the built-in **property** function. This approach allows you to define getters and

setters, but still access them like regular attributes. For example, we could write **leo.age = 5** and still control the access and modification of age variable :

```
class Animal:
    def __init__(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, new_age):
        if isinstance(new_age, int):
            self.__age = new_age
        else:
            raise TypeError("Animal age must
            be an integer")

    age = property(get_age, set_age)
    # Using the in-built property function


leo = Animal(2)
print(leo.age)  # Accessing the attribute like regular

leo.age = 3  # Modifying the attribute like regular
print(leo.age)

leo.age = "old"
print(leo.age)  # Still raises Type Error as required
```

The above approach works fine. However, when classes and methods grow in size and complexity, it can get difficult to keep track of things with the above notation. Hence, Python provides the alternate **@decorator** syntax to define the same property function. This makes it easier to follow the property definition, keeping it right next to the methods that are associated with it, instead of declaring it all together at the end :

```
class Animal:
    def __init__(self, age):
        self.__age = age

    @property  # Using the @decorator syntax
    def age(self):
        return self.__age

    @age.setter  # The method names should be same
    def age(self, new_age):  # as the attribute name age
        if isinstance(new_age, int):
            self.__age = new_age
        else:
```

```
            raise TypeError("Animal age must
            be an integer")


leo = Animal(2)
print(leo.age)

leo.age = 3
print(leo.age)

leo.age = "old"
print(leo.age)
```

A decorator is a function that takes another function as parameter and lets you execute some code before and after that function, hence extending its functionality without having to modify the function body. The @decorator notation is merely an alternate syntax. Both the Property_Function.py and Property_Decorator.py implementations above are otherwise semantically equivalent.

5 Default Arguments

In Python, default arguments allow a function to have default values for parameters, if no argument is passed during the function call. These default values are defined when the function is declared. Here is a simple example :

```
def greet(name="Guest"):  # Default argument "Guest"
    print(f"Hello {name}!")


greet()  # Output: Hello Guest!
greet("Aviral")  # Output: Hello Aviral!
```

In this case, the function **greet( )** has a default argument. If no argument is provided when calling the function, **Guest** will be used by default. Otherwise, the provided argument will override the default.

6 Inheritance

Inheritance allows us to define a class that inherits all the data and methods from another class. **Parent class** is the class being inherited from, **Child class** is the class that inherits. Here is a simple example :

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")
```

```
class Dog(Animal):  # Inherits from Animal
    def __init__(self, name, breed=None):
        Animal.__init__(self, name)
        # Calling Animal's __init__ to
        inherit and save extra lines of code

        self.breed = breed
        # Initialize 'breed' attribute,
        Default value set to None

    def speak(self):
        print(f"{self.name}, the {self.breed}, barks.")
        # Overrides speak method of Animal


any_animal = Animal("Animal")
dog = Dog("Leo", "German Shepherd")

any_animal.speak()  # Output: Animal makes a sound.
dog.speak()  # Output: Leo, the German Shepherd, barks.
```

In this example, the **Dog class** inherits from the **Animal class** and overrides the speak method. The Dog object calls the speak method specific to the child class, while the Animal object calls the parent class method. The code can be viewed here : Inheritance.py

6.1 Super Function

Python also provides a super function that allows the child class to inherit data and methods from its parent class. While using the super function, you do not have to use the name of the parent class explicitly. Super will automatically do that for you. Hence, in the above example, instead of inheriting the init method from parent class like this :

```
class Dog(Animal):
    def __init__(self, name, breed=None):
        Animal.__init__(self, name)
        self.breed = breed
```

You can inherit like this, instead :

```
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed
```

This is a more flexible approach, especially when dealing with multiple inheritance. It automatically finds the correct method to call in the class hierarchy and hence, is more maintainable when working with complex inheritance structures.

7 Class Variables

So far we have seen instance variables. These variables are common across all instances of a class, but their value varies across different instances. On the other hand, class variables and their values are shared across all instances of a class. For example :

```
class Dog:
    species = "Canine"  # Class variable,
    shared by all instances

    def __init__(self, name):
    self.name = name  # Instance variable,
    unique to each instance


# Creating two instances
dog1 = Dog("Misty")
dog2 = Dog("Leo")

# Accessing instance variables
print(dog1.name)  # Output: Misty
print(dog2.name)  # Output: Leo

# Accessing class variable
print(dog1.species)  # Output: Canine
print(dog2.species)  # Output: Canine

# Changing class variable
Dog.species = "Wolf"

# Now all instances reflect the change
print(dog1.species)  # Output: Wolf
print(dog2.species)  # Output: Wolf
```

Hence, class variable values are shared by all instances and can be modified for all of them at once. Whereas, instance variables hold individual values unique to each instance. The code is uploaded here : Class_Variables.py