Chapter 8: Creating Your Own Type – Classes

What we will learn:

- ✓ Object-oriented programming
- ✓ What is a class
- ✓ How to create a class
- ✓ Assigning values to a class

What you need to know before:

- ✓ Data types
- ✓ Methods

Keywords: Object-oriented, Object, Attributes, Class, Abstract data type

8.1 Object-Oriented Programming (OOP)

We had a thorough discussion about data types in Chapter 3. It turns out that most applications in real life will require other data types that are not supplied by Python. Luckily, Python provides the facility to create your own data type. New data types created are often referred to as objects and the process is called object-oriented programming. In the Introduction, when we introduced Python, we stated that Python was a hybrid language and supported a multiple programming paradigm (a way of thinking about computation); it turns out that the ability to create classes and manipulate objects qualifies Python to be an object-oriented programming language. As an example, let us consider a student enrolment system used at a school or university. It would be a great idea if we could refer to a student and Python were to understand that we are talking about a person who has a name, address, telephone number, is registered for certain courses, and can do things such as walk, run, take an exam, to name a few. It turns out that these entities (students in this case) are referred to as objects.

An **object** is a thing or an event in our application. Objects can have data items, these are known as **attributes**. Objects also have behaviours which are called **methods**. Earlier in our students example, we classified a student as an object. The diagram below shows the student object with some attributes and behaviours:

Student

Attributes:
Name
Address
Telephone Number

Behaviour:
Can take course
Can talk
Can attend school

8.2 Why Do We Use Classes?

1) ***To bring the solution space closer to the problem space***. It is more natural to want to find out the name of a student than just referring to the string name. Computer programs are written to solve problems. Where the problem occurs is referred to as the problem space; for example, the problem space in our example above is a school and, in particular, in a course registration system. Since we are using the computer to solve this problem, then the solution space is in the computer. Using classes will give the programmer the ability to relate real-life entities (students) to the interpreter, bringing the solution space closer to the problem space.

2) ***To develop so-called abstract data types (ADT)***. Abstract data types are new data types that we can define numerous operations on. For example, if we develop a new ADT called "shapes" we can define interesting methods, such as `draw_self()`, or `colour_self()` which allow the shape to draw and colour itself.

3) ***For reuse of code.*** If we have a definition of a good ADT, then we can always import that ADT and use the methods that are already defined. Let us say we developed the student registration system mentioned above, and now we want to develop an extra-curricular club program that has student information then there is no need to redefine the student ADT that was developed earlier. Furthermore, other programmers can use ADT that was developed by other programmers.

4) ***To enforce data encapsulation.*** Data encapsulation is information hiding. It turns out that with classes, a programmer can use a new ADT without knowing the underlining implementation. This allows programmers to develop codes that other programmers can use by accessing the methods that are defined over these ADTs, without worrying about the intricacies of how this is actually accomplished. This also give the unique advantage, where if a new way of accomplishing a task comes about then the writer of the ADT can make this change and the programmer using this code will not struggle. We have seen this modular design in engineering and mechanics before, where the driver of a car is not necessarily concerned about how the pedal uses a lever to connect to the master cylinder which in turn applies the disk pads to the wheel to stop the car. Instead, he is only concerned with the fact that when he pushes on the brake pedal, this will have the effect of stopping the car. The designer of the car can choose whether to use drum brakes, air brakes, disc brakes or another braking system that is not discovered as yet; this will not affect the driver pushing on the pedal to stop the car. In a similar manner, if a more efficient data structure arrives then the designer of the ADT can implement this and all programmers benefit without breaking any code.

8.3 Data Types in Python

We have seen objects before. All data types in Python are objects. The implementation of an object is called a class.

```
>>> name = 'Patrick'
>>> print(type(name))
<class 'str'>
>>>
>>> number = 45
>>> print(type(number))
<class 'int'>
>>>
>>> z = [1,2,3]
>>> print(type(z))
<class 'list'>
>>>
```

From the code snippet on the previous page, we assign "Patrick" to name, and when we print the type() then we see get <class 'str'>. This means that "name" is an instance of a string class. Similarly, "number" is an instance of the integer class and "z" is an instance of the list class.

We have encountered string (a collection of characters) before. We also explored methods that are defined on string. Upper() and lower() are two such methods. It turns out that every string has these methods defined on them; this is the case because in the definition of the string class, upper() and lower() is defined.

```
>>> month = 'September'
>>> print(month.upper())
SEPTEMBER
>>> #now print the lower case of month
>>> print(month.lower())
september
>>>
```

To view all the methods that are defined on an object, we use the `dir()` method.

```
>>> dir(month)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__
init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith
', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'i
sdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper'
, 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', '
rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapca
se', 'title', 'translate', 'upper', 'zfill']
>>>
```

Notice that all of these methods are defined on all strings.

8.4 Defining a Class

We use the class keyword followed by the name to define a class.

Syntax:

```
class <class_name>:
      'class_docstring'
      <class_suite>
```

<class_name> is the name of the class following the same conventions as naming variables, discussed in Chapter 1.

'class_docstring' or documentation string explains what the class is about. This should give the user of the class enough information about the class being defined.

<class_suite> is the statement that defines the class. This will contain the methods that define the class and are invoked whenever we have instances of the class.

Example:

```
class Student:
    """ The Student class defines a student with
    attributes: name, age, form
    """
    #Attributes
    name = "not set"
    age = 1
    form = "7"

    #Functions
    def get_name(self):
        return self.name

    def get_age(self):
        return self.age

    def get_form(self):
        return self.form

    def set_name(self, new_name):
        self.name = new_name

    def set_age(self, new_age):
        self.age=new_age

    def set_form(self, new_form):
        self.form = new_form

    def say_hello(self):
        print("hello my name is", self.name, " I'm in form ", self.form)
```

The code snippet above will define the student class. The different sections are discussed below.

The class and docstring:

```
class Student:
    """ The Student class defines a student with
    attributes: name, age, form
    """
```

The keyword class is used, followed by the name of the class and colon.

The docstring or documentation string is used to build the documentation for any application that is using the class. This information also appears when we use the help facility to display the details of the student class.

The attributes:

```
#Attributes
name = "not set"
age = 1
form = "7"
```

This is the section to list the attributes on the class and initialise the default values.

The class functions:

```
#Functions
def get_name(self):
    return self.name

def get_age(self):
    return self.age

def get_form(self):
    return self.form

def set_name(self, new_name):
    self.name = new_name

def set_age(self, new_age):
    self.age=new_age

def set_form(self, new_form):
    self.form = new_form

def say_hello(self):
    print("hello my name is", self.name, " I'm in form ", self.form)
```

The class functions are used to access and change the attributes of the class. Notice the use of the word self. It turns out that we use self with the dot operator to refer to attributes of data items that belong to the class.

8.4.1 Using the Class

When we create a class, this is called an instance of the class. The following code will create an instance of student1 and use some of its functions.

```
#create a new Student
student1 = Student()

#Set the name of Student1 to John Doe
student1.set_name("John Doe")

#Use the accessor method to print the name
print(student1.get_name())

#Say hello from student1
print(student1.say_hello())
```

A few things to note above:

- We declare a variable and assign the name of the class as if it were a function
- Once we create an instance of the class we can now use the class methods

This is the output that we would get.

```
>>> ================================ RESTART ================================
>>>
John Doe
hello my name is John Doe  I'm in form  7
None
>>>
```

Notice that if we create a new instance of the student class and say “hello” we will get a different message.

```
>>> student2 = Student()
>>> print(student2.say_hello())
hello my name is not set  I'm in form  7
None
>>>
```

When we say “hello”, the default values are displayed.