Object-Oriented Programming

General Introduction

Though Python is an object-oriented language without fuss or quibble, we have so far intentionally avoided the treatment of object-oriented programming (OOP) in the previous chapters of our Python tutorial. We skipped OOP, because we are convinced that it is easier and more fun to start learning Python without having to know about all the details of object-oriented programming.

But even though we have avoided OOP, it has nevertheless always been present in the exercises and examples of our course. We used objects and methods from classes without properly explaining their OOP background. In this chapter, we will catch up on what has been missing so far. We will provide an introduction into the principles of object oriented programming in general and into the specifics of the OOP approach of Python. OOP is one of the most powerful tools of Python, but nevertheless you don't have to use it, i.e. you can write powerful and efficient programs without it as well.

Though many computer scientists and programmers consider OOP to be a modern programming paradigm, the roots go back to 1960s. The first programming language to use objects was Simula 67. As the name implies, Simula 67 was introduced in the year 1967. A major breakthrough for object-oriented programming came with the programming language Smalltalk in the 1970s.

OOP in Python

First-class Everything

Even though we haven't talked about classes and object orientation in previous chapters, we have worked with classes all the time. In fact, everything is a class in Python. Guido van Rossum has designed the language according to the principle "first-class everything". He wrote: "One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth." (Blog, The History of Python, February 27, 2009) This means that "everything" is treated the same way, everything is a class: functions and methods are values just like lists, integers or floats. Each of these are instances of their corresponding classes.

A Minimal Class in Python

We will design and use a robot class in Python as an example to demonstrate the most important terms and ideas of object orientation. We will start with the simplest class in Python.

```
class Robot:

    pass
```

We can realize the fundamental syntactical structure of a class in Python: A class consists of two parts: the header and the body. The header usually consists of just one line of code. It begins with the keyword "class" followed by a blank and an arbitrary name for the class. The class name is "Robot" in our case. The class name is followed by a listing of other class names, which are classes from which the defined class inherits from. These classes are called superclasses, base classes or sometimes parent classes. If you look at our example, you will see that this listing of superclasses is not obligatory. You don't have to bother about inheritance and superclasses for the time being. We will introduce them later.

The body of a class consists of an indented block of statements. In our case a single statement, the "pass" statement.

A class object is created, when the definition is left normally, i.e. via the end. This is basically a wrapper around the contents of the namespace created by the class definition.

It's hard to believe, especially for C++ or Java programmers, but we have already defined a complete class with just three words and two lines of code. We are capable of using this class as well:

```
class Robot:
    pass

x = Robot()
y = Robot()
y2 = y
print(y == y2)
print(y == x)
```

We have created two different robots x and y in our example. Besides this, we have created a reference y2 to y, i.e. y2 is an alias name for y. The output of this example program looks like this:

```
True
False
```

Attributes

Those who have learned already another object-oriented language, will have realized that the terms attributes and properties are usually used synonymously. Even in normal English usage the words "attribute" and "property" can be used in some cases as synonyms. Both can have the meaning "An attribute, feature, quality, or characteristic of something or someone". Usually an "attribute" is used to denote a specific ability or characteristic which something or someone has, like black hair, no hair, or a quick perception, or "her quickness to grasp new tasks". So, think a while about your outstanding attributes. What about your "outstanding properties"? Great, if one of your strong points is your ability to quickly understand and adapt to new situations! Otherwise, you would not learn Python!

Let's get back to Python: So far our robots have no attributes. Not even a name, like it is customary for ordinary robots, isn't it? So, let's implement a name attribute. "type designation", "build year" and so on are easily conceivable as further attributes as well.

Attributes are created inside of a class definition, as we will soon learn. We can dynamically create arbitrary new attributes for existing instances of a class. We do this by joining an arbitrary name to the instance name, separated by a dot ".". In the following example, we demonstrate this by created an attribute for the name and the build year:

```
class Robot:
    pass

x = Robot()
y = Robot()
```

```
 x.name = "Marvin"
 x.build_year = "1979"

 y.name = "Caliban"
 y.build_year = "1993"

 print(x.name)
 print(y.build_year)
```

```
Marvin
1993
```

As we have said before: This is not the way to properly create instance attributes. We introduced this example, because we think that it may help to make the following explanations easier to understand.

Attributes can be bound to class names as well. In this case, each instance will possess this name as well. Watch out, what happens, if you assign the same name to an instance:

```
>>> class Robot(object):
...     pass
...
>>> x = Robot()
>>> Robot.brand = "Kuka"
>>> x.brand
'Kuka'
>>> x.brand = "Thales"
>>> Robot.brand
'Kuka'
>>> y = Robot()
>>> y.brand
'Kuka'
>>> Robot.brand = "Thales"
>>> y.brand
'Thales'
>>> x.brand
'Thales'
```

Binding attributes to objects is a general concept in Python. Even function names can be attributed. You can bind an attribute to a function name in the same way, we have done so far:

```
def f(x):
     return 0
f.x = 42
```

```
print(f.x)
```

```
42
```

As we have already mentioned, we do not create instance attributes like this. To properly create instances we need methods. You will learn in the following subsection of our tutorial, how you can define methods.

The __init__ Method

We want to define the attributes of an instance right after its creation. __init__ is a method which is immediately and automatically called after an instance has been created. This name is fixed and it is not possible to chose another name. __init__ is one of the so-called magic methods, of which we will get to know some more details later. The __init__ method is used to initialize an instance. There is no explicit constructor or destructor method in Python, as they are known in C++ and Java. The __init__ method can be anywhere in a class definition, but it is usually the first method of a class, i.e. it follows right after the class header.

```
>>> class A:
...     def __init__(self):
...         print("__init__ has been executed!")
...
>>> x = A()
__init__ has been executed!
>>>
```

We add an __init__-method to our robot class:

```
class Robot:


    def __init__(self, name=None):
        self.name = name


    def say_hi(self):
        if self.name:
            print("Hi, I am " + self.name)
        else:
            print("Hi, I am a robot without a name")



x = Robot()
x.say_hi()
y = Robot("Marvin")
y.say_hi()
```

This little program returns the following:

```
Hi, I am a robot without a name
Hi, I am Marvin
```

Creating Classes

Example

Following is an example of a simple Python class –

```
class Employee:
   'Common base class for all employees'
   empCount = 0
   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee :" , Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)
```

- The variable *empCount* is a class variable whose value is shared among all the instances of a in this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts.

This would create first object of Employee class

```
emp1 = Employee("Zara", 2000)
```

This would create second object of Employee class

```
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows −

```
emp1.displayEmployee()

emp2.displayEmployee()

print ("Total Employee :", Employee.empCount)
```

Now, putting all the concepts together −

```
class Employee:

   'Common base class for all employees'

   empCount = 0


   def __init__(self, name, salary):

      self.name = name

      self.salary = salary

      Employee.empCount += 1


   def displayCount(self):

     print ("Total Employee :" , Employee.empCount)


   def displayEmployee(self):

      print ("Name : ", self.name,  ", Salary: ", self.salary)



#This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

#This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

print ("Total Employee :" , Employee.empCount)
```

When the above code is executed, it produces the following result −

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time −

```
emp1.age = 27  # Add an 'age' attribute.
emp1.name = 'xyz'  # Modify 'name' attribute.
del emp1.salary  # Delete 'salary' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions −

- The **getattr(obj, name[, default])** − to access the attribute of object.
- The **hasattr(obj,name)** − to check if an attribute exists or not.
- The **setattr(obj,name,value)** − to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** − to delete an attribute.

```
print(hasattr(emp1, 'salary'))   # Returns true if 'salary' attribute exists

print(getattr(emp1, 'salary'))    # Returns value of 'salary' attribute

setattr(emp1, 'salary', 7000) # Set attribute 'salary' at 7000

delattr(emp1, 'salary')    # Delete attribute 'salary'
```