8.94 - Leaflet - Object Orientation

July 23, 2018

1 First Steps: Object Orientation

Object Orientation is an incredibly powerful concept that allows us to organize code cleanly.

We have already worked with objects, such as the Python List object, to which we have applied methods such as the append() method.

```
In [1]: students = ["Max", "Monica"]
        students.append("Eric")

        print(students)
```

```
['Max', 'Monica', 'Eric']
```

1.0.1 Define a class

We want to create our own objects with our own methods. For this we need classes, these are building plans for objects. The objects created according to these instructions are called instances of this class:

```
In [4]: # We define the class Student with the method name(),
        # Class names start with capital letters according to convention

        class Student():

            # self is a keyword, it acts in a way
            # as placeholder for the respective instance
            def name(self):
                print(self.firstname + " " + self.lastname)
```

1.0.2 Create an instance

Using this class as a template, we now create a student instance and save it in a variable:

```
In [5]: eric = Student()
```

```
In [7]: monica = Student()
        monica.firstname = "Monica"
        monica.lastname ="Miller"

        print(monica.firstname)
        print(monica.lastname)
```

```
Monica
Miller
```

1.0.3 Use the method of an object

Access to the method of the object works as usual:

```
In [8]: eric.name()
```

```
        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-8-8684eabe9641> in <module>()
    ----> 1 eric.name()


        <ipython-input-4-d23e3c6c8e90> in name(self)
          7     # as placeholder for the respective instance
          8     def name(self):
    ----> 9         print(self.firstname + " " + self.lastname)


        AttributeError: 'Student' object has no attribute 'firstname'
```

```
In [10]: monica.name()
```

```
Monica Miller
```

In particular, there can also be other objects with a method of the same name:

```
In [11]: class Company():

             def name(self):
                 print(self.legal_name + ": " + self.legal_type)

         c = Company()
         c.legal_name = "John Doe"
```

```
        c.legal_type = "Evilcorp"

        c.name()
```

```
John Doe: Evilcorp
```

```
In [13]: def name_5x(v):
             for i in range(0,5):
                 v.name()

         name_5x(c)
         # name_5x(eric)
         name_5x(monica)
```

```
John Doe: Evilcorp
John Doe: Evilcorp
John Doe: Evilcorp
John Doe: Evilcorp
John Doe: Evilcorp
Monica Miller
Monica Miller
Monica Miller
Monica Miller
Monica Miller
```

In the function name_5x() the method name() belonging to the object is executed. Of course, the objects must contain a name() method. Eric would create an error, as he is not properly initialized.

2 Object orientation: constructor, change properties

The next sections are about:

- Use a constructor to define the properties of a class
- Change the properties of an instance.

We have already extended the Students class a little:

```
In [18]: class Student():

             # This is our so-called constructor:
             # Here we define the variables for the class.
             #
             # self is mandatory and always refers to the Object that is currently being creat
             #
             # When the instance is created, self does not as a parameter!
```

```
            def __init__(self, firstname, lastname):
                self.firstname = firstname
                self.lastname = lastname

            def name(self):
                print(self.firstname + " " + self.lastname)

        will = Student("Will", "Smith")
        will.name()
```

```
Will Smith
```

The class definition with Constructor therefore delivers the same results as the previous more cumbersome definition.

We are adding another method:

```
In [21]: class Student():

            def __init__(self, firstname, lastname):
                self.firstname = firstname
                self.lastname = lastname
                # Here we initialize the new variable term (property)
                self.term = 1

                 # With this method we increase the term variable by 1
            def increase_term(self):
                self.term = self.term + 1

                # name() now also outputs the number of semesters
            def name(self):
                print(self.firstname + " " + self.lastname +
                      " (Semester: " + str(self.term) + ")")


In [23]: eric = Student("Eric", "Johnson")
         eric.name()
```

```
Eric Johnson (Semester: 1)
```

```
In [24]: eric.increase_term()
         eric.name()
```

```
Eric Johnson (Semester: 2)
```

2.1 Object Orientation: Private Properties and Methods

Private properties allow clean encapsulation of properties and methods. This allows us to "protect" variables and methods from "curious looks" from outside - very important if we want to be able to adjust these variables/methods later. This is only possible if our colleague does not access it from his code:

```
In [26]: class Student():

             def __init__(self, firstname, lastname):
                 self.firstname = firstname
                 self.lastname = lastname
                 self.term = 1

             # With this method we limit that you can
             # not have more than nine semesters
             def increase_term(self):
                 if self.term >= 9:
                     return
                 self.term = self.term + 1

             def get_term(self):
                 return self.term

             def name(self):
                 print(self.firstname + " " + self.lastname +
                       " (Semester: " + str(self.term) + ")")


         eric = Student("Eric", "Johnson")
         eric.increase_term()
         eric.name()
```

```
Eric Johnson (Semester: 2)
```

```
In [27]: erik.increase_term()
         erik.increase_term()
         erik.increase_term()
         erik.increase_term()
         erik.increase_term()
         erik.increase_term()
         erik.increase_term()
         erik.increase_term()
         erik.increase_term()
         erik.increase_term()
         erik.name()
```

```
Erik Mustermann (Semester: 9)
```

Nevertheless, we can still access the property from outside and overwrite it:

```
In [28]: erik.term = 100
         erik.name()
```

```
Erik Mustermann (Semester: 100)
```

However, if we put two underscores before the variable, we make it **private**.

Python also has the programmers' convention of naming private properties with an underscore, even if they are technically not yet private. It really takes two underscores at the beginning of its name:

```
In [29]: class Student():

             def __init__(self, firstname, lastname):
                 self.firstname = firstname
                 self.lastname = lastname
                 self.__term = 1

             def increase_term(self):
                 if self.__term >= 9:
                     return
                 self.__term = self.__term + 1

             # We'll add this method to be able to access it from outside
             def get_term(self):
                 return self.__term

             def name(self):
                 print(self.firstname + " " + self.lastname +
                       " (Semester: " + str(self.__term) + ")")
```

```
In [33]: eric = Student("Eric", "Johnson")
         # Point in conjunction with underscore is understood as warning
         # avoid at all costs!

         # Two underscores (like here) is completely "private",
         # That's why we can't access the attribute like this:

         print(eric.__term)
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-33-d39afcdd0f3e> in <module>()
  6 # That's why we can't access the attribute like this:
```

```
      7
----> 8 print(eric.__term)


AttributeError: 'Student' object has no attribute '__term'
```

We can also restrict access to methods:

```
In [38]: class Student():

             def __init__(self, firstname, lastname):
                 self.firstname = firstname
                 self.lastname = lastname
                 self.__term = 1

             def increase_term(self):
                 if self.__term >= 9:
                     return
                 self.__term = self.__term + 1

             def get_term(self):
                 return self.__term

             def name(self):
                 print(self.firstname + " " + self.lastname +
                       " (Semester: " + str(self.__term) + ")")

             # Our private function
             def __do_something(self):
                 print(self.firstname[0] + " " + self.lastname)

In [40]: test = Student("Adam", "Test")
         test.name()
         test.__do_something()

Adam Test (Semester: 1)


---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-40-58c6a04a0084> in <module>()
  1 test = Student("Adam", "Test")
  2 test.name()
----> 3 test.__do_something()
```

```
AttributeError: 'Student' object has no attribute '__do_something'
```

2.2 In Python there are a few special methods that our class can implement...

With that, you can make sure that:

- your class can be printed directly
- you can calculate the len(variable).

The str-method

```
In [44]: class PhoneBook():
             def __init__(self):
                 self.__entries = {}

             def add(self, name, phone_number):
                 self.__entries[name] = phone_number

             def get(self, name):
                 if name in self.__entries:
                     return self.__entries[name]
                 else:
                     return None

             def __str__(self):
                 return "PhoneBook(" + str(self.__entries) + ")"

         book = PhoneBook()
         book.add("Doe", "+12345678")
         book.add("Miller", "+123456789")

         print(book)
```

```
PhoneBook({'Doe': '+12345678', 'Miller': '+123456789'})
```

The repr-method

```
In [45]: class PhoneBook():
             def __init__(self):
                 self.__entries = {}

             def add(self, name, phone_number):
                 self.__entries[name] = phone_number

             def get(self, name):
                 if name in self.__entries:
```

```
                return self.__entries[name]
            else:
                return None

        def __str__(self):
            return "PhoneBook(" + str(self.__entries) + ")"

        def __repr__(self):
            return self.__str__()


book = PhoneBook()
book.add("Doe", "+12345678")
book.add("Miller", "+123456789")

print(book)
```

```
PhoneBook({'Doe': '+12345678', 'Miller': '+123456789'})
```

The len-method

```
In [46]: class PhoneBook():
             def __init__(self):
                 self.__entries = {}

             def add(self, name, phone_number):
                 self.__entries[name] = phone_number

             def get(self, name):
                 if name in self.__entries:
                     return self.__entries[name]
                 else:
                     return None

             def __len__(self):
                 return len(self.__entries)


book = PhoneBook()
book.add("Mustermann", "+4912345678")
book.add("Müller", "+49123456789")

print(len(book))
```

```
2
```

3 Inheritance

Inheritance is a fundamental concept of object orientation with which you can divide and better model data.

We already know the student class:

```
In [48]: class Student():
             def __init__(self, firstname, surname):
                 self.firstname = firstname
                 self.surname = surname

             def name(self):
                 return self.firstname + " " + self.surname
```

```
In [49]: student = Student("Monica", "Miller")
         print(student.name())
```

```
Monica Miller
```

We want to define another class that is similar:

```
In [50]: class WorkingStudent():

             def __init__(self, firstname, surname, company):
                 self.firstname = firstname
                 self.surname = surname
                 self.company = company

             def name(self):
                 return self.firstname + " " + self.surname
```

```
In [51]: student = WorkingStudent("John", "Johnson", "Evelcorp")
         print(student.name())
```

```
John Johnson
```

3.0.1 Define a class with inheritance

We can save ourselves the trouble of defining the same instance variables and methods again - thanks to inheritance. For this purpose, we refer to another class definition:

```
In [52]: # We pass the class from which we want to inherit as a parameter (parent class)
         class WorkingStudent(Student):

             def __init__(self, firstname, surname, company):
                 # The old instance variable definitions become obsolete below
```

```
                # self.firstname = firstname
                # self.surname = surname

                # with super() we indicate to Python that the init() method of the parent cla
                super().__init__(firstname, surname)
                self.company = company

            def name(self):
                # again we refer with super() to the method of the parent class, which we ove
                return super().name() + " (" + self.company +")"
```

```
In [54]: student = WorkingStudent("John", "Johnson", "Evilcorp")
         print(student.name())
```

```
John Johnson (Evilcorp)
```

```
In [56]: students = [
             WorkingStudent("Max", "Worker", "ABC"),
             Student("Monica", "Smartass"),
             Student("Eric", "Smartass"),
             WorkingStudent("Paula", "Worker", "XYZ")
         ]

         for student in students:
             print(student.name())
```

```
Max Worker (ABC)
Monica Smartass
Eric Smartass
Paula Worker (XYZ)
```

Here we see that the different name() methods return different outputs, although we access them with the same name.

4 Check types of variables - the type() and isinstance() functions

We will again use the well-known Student and WorkingStudent classes for the examples:

```
In [57]: class Student():
             def __init__(self, firstname, surname):
                 self.firstname = firstname
                 self.surname = surname

             def name(self):
                 return self.firstname + " " + self.surname
```

```
class WorkingStudent(Student):
    def __init__(self, firstname, surname, company):
        super().__init__(firstname, surname)
        self.company = company

    def name(self):
        return super().name() + " (" + self.company + ")"
```

```
In [59]: w_student = WorkingStudent("John", "Doe", "Evilcorp")
         student = Student("Monica", "Miller")
```

4.0.1 Check the type with type()

With the **type()** function we can determine the type of an object:

```
In [60]: print(type(w_student))
         print(type(student))
```

```
<class '__main__.WorkingStudent'>
<class '__main__.Student'>
```

```
In [61]: if type(w_student) == Student:
             print("I am a Worker.")

         if type(student) == Student:
             print("I am a real Student")
```

```
I am a real Student
```

4.0.2 Check if it is an instance with isinstance()

The function **isinstance()** gets two parameters: the variable and the class it will be checked against. isinstance() returns a boolean.

```
In [62]: print(isinstance(w_student, WorkingStudent))
         print(isinstance(w_student, Student))

         print(isinstance(student, WorkingStudent))
         print(isinstance(student, Student))
```

```
True
True
False
True
```

Since Student is the parent class of WorkingStudent, w_student is also an instance of Student.

This function is useful if we want to filter by classes, e.g. only output instances of WorkingStudent:

```
In [63]: students = [
             WorkingStudent("Max", "Worker", "ABC"),
             Student("Monica", "Smartass"),
             Student("Eric", "Smartass"),
             WorkingStudent("Paula", "Worker", "XYZ")
         ]

         for student in students:
             ## alternativ:
             ## if isinstance(student, WorkingStudent):
             if type(student) == WorkingStudent:
                 print(student.name())
```

```
Max Worker (ABC)
Paula Worker (XYZ)
```

4.1 Styleguide - Naming classes and variables

Basically it doesn't matter how we name a class / variable in Python. Our program will work either way.

*__But:__ For Python there are some style guides how we can write "nice" code. I would like to go through the most important points in this section (https://www.python.org/dev/peps/pep-0008/).

How can (should) we name variables / classes / functions, especially if the names should consist of several words?

In Python, this is done by convention:

- PascalCase (`SeveralWords`)
- sneak_case (`several_words`)

Unlike other programming languages, this is not used:

- camelCase (`severalWordswords`)

```
In [64]: # Class names in PascalCase
         # But this example is too long ;-)
         class SeveralWorldsButWaaaayToooooLoooong():
             def __init__(self):
                 print("TEST")

             # Function name in sneak_case
             def i_am_a_function(self):
                 print("asdf")

         # Variable names also in sneak_case; but at most three words ;-)
         several_worlds = SeveralWorldsButWaaaayToooooLoooong()

         print(several_worlds )
```

```
TEST
<__main__.SeveralWorldsButWaaaayTooooLoooong object at 0x000001A7078CB080>
```