Merkblatt Objektorientierung

May 7, 2020

1 Erste Schritte: Objektorientierung

Objektorientierung ist ein unglaublich mächtiges Konzept, welches es uns ermöglicht, Code sauber zu organisieren.

Wir haben schon mit Objekten gearbeitet, z. B. mit dem Listen-Objekt von Python, auf das wir Methoden wie die append()-Methode angewendet haben.

[3]:
```
students = ["Max", "Monika"]
students.append("Erik")

print(students)
```

```
['Max', 'Monika', 'Erik']
```

1.0.1 Eine Klasse definieren

Wir wollen eigene Objekte mit eigenen Methoden erzeugen. Dafür brauchen wir Klassen, das sind Baupläne für Objekte. Die entsprechend dieser Anleitungen erzeugten Objekte nennt man Instanzen dieser Klasse:

[2]:
```
# Wir definieren die Klasse Student mit der Methode name(),
# Klassennamen beginnen gemäß Konvention mit Großbuchstaben

class Student():

    # self ist ein Schlüsselwort, es fungiert gewissermassen
    # als Platzhalter für die jeweilige Instanz
    def name(self):
        print(self.firstname + " " + self.lastname)
```

1.0.2 Eine Instanz erstellen

Mittels dieser Klasse als Vorlage erstellen wir uns nun eine Student-Instanz und speichern sie in einer Variable:

[8]:
```
erik = Student()
```

[16]:
```
monika = Student()
monika.firstname = "Monika"
monika.lastname ="Müller"

print(monika.firstname)
print(monika.lastname)
```

```
Monika
Müller
```

1.0.3 Die Methode eines Objektes benutzen

Wie gewohnt funktioniert so auch der Zugriff auf die Methode des Objektes:

[17]:
```
erik.name()
```

```
Erik Mustermann
```

[19]:
```
monika.name()
```

```
Monika Müller
```

Insbesondere kann es auch weitere Objekte mit einer Methode desselben Namens geben:

[31]:
```
class Company():

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

c = Company()
c.legal_name = "Max Müller"
c.legal_type = "GmbH"

c.name()
```

```
Max Müller: GmbH
```

[32]:
```
def name_5x(v):
    for i in range(0,5):
         v.name()

name_5x(c)
name_5x(erik)
name_5x(monika)
```

```
Max Müller: GmbH
Max Müller: GmbH
Max Müller: GmbH
Max Müller: GmbH
```

```
Max Müller: GmbH
Erik Mustermann
Erik Mustermann
Erik Mustermann
Erik Mustermann
Erik Mustermann
Monika Müller
Monika Müller
Monika Müller
Monika Müller
Monika Müller
```

In der Funktion name_5x() wird jeweils die zum Objekt gehörige Methode name() ausgeführt. Dafür müssen die Objekte natürlich eine name()-Methode enthalten.

2 Objektorientierung: Constructor, Eigenschaften abändern

In den nächsten Abschnitten geht es darum, wie du:

- mit Hilfe eines Constructors Eigenschaften einer Klasse definieren kannst
- Eigenschaften einer Instanz abänderst.

Wir haben die bereits die Klasse Students ein wenig erweitert:

[13]:
```python
class Student():

    # Das ist unser sogenannter Constructor: hier definieren
    # wir die Variablen für die Klasse.
    #
    # self ist obligatorisch und bezieht sich immer auf das
    # Objekt, das gerade angelegt wird.
    #
    # bei der Erzeugung der Instanz taucht self aber nicht
    # als Parameter auf!
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

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

erik = Student("Erik", "Mustermann")
erik.name()
```

```
Erik Mustermann
```

Die Klassendefinition mit Constructor liefert also dieselben Ergebnisse wie die vorherige umständlichere Definition.

Wir ergänzen eine weitere Methode:

[1]:
```
class Student():

    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname
        # Hier initialisieren wir die neue Variable term (Eigenschaft)
        self.term = 1

         # Mit dieser Methode erhöhen wir die Variable term um 1
    def increase_term(self):
        self.term = self.term + 1

        # name() gibt nunmehr zusätzlich die Anzahl der Semester aus
    def name(self):
        print(self.firstname + " " + self.lastname +
              " (Semester: " + str(self.term) + ")")
```

[19]:
```
erik = Student("Erik", "Mustermann")
erik.name()
```

```
Erik Mustermann (Semester: 1)
```

[20]:
```
erik.increase_term()
erik.name()
```

```
Erik Mustermann (Semester: 2)
```

2.1 Objektorientierung: Private Eigenschaften und Methoden

Private Eigenschaften erlauben sauberes Kapseln von Eigenschaften und Methoden. Dadurch können wir Variablen und Methoden quasi vor "neugierigen Blicken" von außerhalb "schützen" - sehr wichtig, wenn wir später die Möglichkeit haben möchten, diese Variablen / Methoden noch anzupassen. Das geht nur, wenn unser Kollege von seinem Code aus darauf nicht zugreift:

[16]:
```
class Student():

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

    # Bei dieser Methode schränken wir ein, dass man
    # nicht mehr als 9 Semester erreichen
    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) + ")")


erik = Student("Erik", "Mustermann")
erik.increase_term()
erik.name()
```

```
Erik Mustermann (Semester: 2)
```

[17]:
```
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)
```

Trotzdem können wir von außen noch auf die Eigenschaft zugreifen und sie überschreiben:

[19]:
```
erik.term = 100
erik.name()
```

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

Wenn wir allerings zwei Unterstriche vor die Variable setzen, machen wir sie **privat**.

In Python gibt es auch die Konvention seitens der Programmierer, private Eigenschaften mit einem Unterstrich zu benennen, auch wenn sie dann technisch noch nicht privat sind. Dafür braucht es wirklich zwei Unterstriche zu Beginn ihres Namens:

[3]:
```
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

    # Die Methode ergänzen wir, um von außerhalb der Klasse
    # noch die term-Eigenschaft abfragen zu können
    def get_term(self):
        return self.__term

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

[5]:
```
erik = Student("Erik", "Mustermann")
# Punkt in Verbindung mit Unterstrich ist dabei als
# Warnung zu verstehen - unbedingt vermeiden!

# Zwei Unterschtriche (wie hier) ist aber komplett "privat",
# daher können wir so nicht auf das Attribut zugreifen:

print(erik.__term)
```

```
␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-5-6f05b6d74b22> in <module>()
         6 # daher können wir so nicht auf das Attribut zugreifen:
         7
   ----> 8 print(erik.__term)


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

Auch den Zugriff auf Methoden können wir einschränken:

[31]:
```
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) + ")")

    # Unserer private Funktion
    def __do_something(self):
        print("doSomething")
```

[34]:
```
test = Student("Tim", "Test")
test.name()
test.__do_something()
```

```
Tim Test (Semester: 1)
```

```
␣
↪---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-34-68818f291adb> in <module>()
  1 test = Student("Tim", "Test")
  2 test.name()
----> 3 test.__do_something()


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

2.2 In Python gibt es ein paar besondere Methoden, die unsere Klasse implementieren kann...

Damit kannst du dafür sorgen, dass:

- deine Klasse direkt ausgegeben werden kann
- du len(variable) berechnen kannst.

Die str-Funktion

[5]:
```
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("Mustermann", "+4912345678")
book.add("Müller", "+49123456789")

print(book)
```

```
PhoneBook({'Mustermann': '+4912345678', 'Müller': '+49123456789'})
```

Die repr-Methode

[12]:
```
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("Mustermann", "+4912345678")
```

```
book.add("Müller", "+49123456789")

print(book)
```

```
PhoneBook({'Mustermann': '+4912345678', 'Müller': '+49123456789'})
```

Die len-Methode

[13]:
```
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 Vererbung

Vererbung ist ein fundamentales Konzept der Objektorientierung, mit dem du Daten aufteilen und besser modellieren kannst.

Die Klasse Student kennen wir schon:

[4]:
```
class Student():
    def __init__(self, firstname, surname):
        self.firstname = firstname
        self.surname = surname

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

[7]:
```
student = Student("Monika", "Mustermann")
print(student.name())
```

```
Monika Mustermann
```

Wir wollen eine weitere Klasse definieren, die Ähnlichkeiten zu einer bereits bestehenden Klasse aufweist:

[29]:
```
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
```

[30]:
```
student = WorkingStudent("Max", "Müller", "ABCDEF GmbH")
print(student.name())
```

```
Max Müller
```

3.0.1 Eine Klasse mit Vererbung definieren

Wir können uns sparen, gleiche Instanzvariablen und Methoden ein weiteres Mal zu definieren - dank Vererbung. Dazu verweisen wir innerhalb einer Klassendefinition auf eine andere:

[44]:
```
# Als Parameter übergeben wir die Klasse, von der unsere neue Eigenschaften und␣
↪Methoden vererbt werden soll (Mutterklasse)
class WorkingStudent(Student):

    def __init__(self, firstname, surname, company):
        # Die alten Instanzvariablendefinitionen werden unten hinfällig
        # self.firstname = firstname
        # self.surname = surname

        # mit super() zeigen wir Python an, dass die init()-Methode der␣
↪Mutterklasse angewendet werden soll
        super().__init__(firstname, surname)
        self.company = company

    def name(self):
        # wieder verweisen wir mit super() auf die Methode der Mutterklasse,␣
↪die wir für die Klasse WorkingStudent überschreiben
        return super().name() + " (" + self.company + ")"
```

[39]:
```
student = WorkingStudent("Max", "Müller", "ABCDEF GmbH")
print(student.name())
```

```
Max Müller (ABCDEF GmbH)
```

[43]:
```
students = [
    WorkingStudent("Max", "Müller", "ABCDEF GmbH"),
    Student("Monika", "Mustermann"),
    Student("Erik", "Müller"),
    WorkingStudent("Franziska", "Mustermann", "XYZXYZ GmbH")
]

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

```
Max Müller (ABCDEF GmbH)
Monika Mustermann
Erik Müller
Franziska Mustermann (XYZXYZ GmbH)
```

Hier sehen wir, dass die verschiedenen name()-Methoden verschiedene Ausgaben liefern, obwohl wir mit demselben Namen auf sie zugreifen.

4 Typen von Variablen prüfen - die type() und isinstance() Funktionen

Wir benutzen für die Beispiele wieder die bekannten Student und WorkingStudent-Klassen:

[3]:
```
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 + ")"
```

[4]:
```
w_student = WorkingStudent("Max", "Müller", "ABCDEF GmbH")
student = Student("Monika", "Mustermann")
```

4.0.1 Den Typ überprüfen mit type()

Mit der **type()**-Funktion können wir den Typ eines Objektes feststellen:

[13]:
```
print(type(w_student))
print(type(student))
```

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

[16]:
```
if type(w_student) == Student:
    print("Diese Zeile wird nur für einen Student ausgegeben")

if type(student) == Student:
    print("Hier hingegen steht ein richtiger Student")
```

```
Hier hingegen steht ein richtiger Student
```

4.0.2 Checken, ob es sich um eine Instanz handelt mit isinstance()

Die Funktion **isinstance()** erhält zwei Parameter: die Variable und die Klasse bezüglich derer auf Zugehörigkeit der Variable geprüft werden soll. isinstance() liefert einen Bool zurück.

[19]:
```
print(isinstance(w_student, WorkingStudent))
print(isinstance(w_student, Student))

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

```
True
True
False
True
```

Da Student die Mutterklasse von WorkingStudent ist, ist w_student auch bezüglich Student eine Instanz.

Nützlich wird die Funktion, wenn wir nach Klassen filtern wollen, z. B. nur Instanzen von WorkingStudent ausgeben:

[24]:
```
students = [
    WorkingStudent("Max", "Müller", "ABCDEF GmbH"),
    Student("Monika", "Mustermann"),
    Student("Erik", "Müller"),
    WorkingStudent("Franziska", "Mustermann", "XYZXYZ GmbH")
]

for student in students:
    ## alternativ:
    ## if isinstance(student, WorkingStudent):
```

```
    if type(student) == WorkingStudent:
        print(student.name())
```

```
Max Müller (ABCDEF GmbH)
Franziska Mustermann (XYZXYZ GmbH)
```

4.1 Styleguide - Benennung von Klassen und Variablen

Grundsätzlich ist es in Python egal, wie wir eine Klasse / Variable benennen. Unser Programm wird so oder so funktionieren.

Aber: Für Python gibt es ein paar Style - Guides, wie wir "schönen" Code schreiben können. Da möchte ich in diesem Abschnitt die wichtigsten Punkte von durchgehen (https://www.python.org/dev/peps/pep-0008/).

Wie können (sollten) wir Variablen / Klassen / Funktionen überhaupt benennen, insbesondere wenn die Namen aus mehreren Wörtern bestehen sollen?

In Python verwendet man dazu nach Konvention:

- PascalCase (`IchBesteheAusMehrerenWoertern`)
- sneak_case (`ich_bestehe_aus_mehreren_woertern`)

Anders als in anderen Programmiersprachen benutzt man nicht:

- camelCase (`ichBesteheAusMehrerenWoertern`)

[3]:
```
# Klassennamen in PascalCase
# Das Beispiel ist aber zu lange ;-)
class IchBesteheAusMehrerenWoertern():
    def __init__(self):
        print("TEST")

    # Funktionsname in sneak_case
    def ich_bin_eine_funktion(self):
        print("asdf")

# Variablennamen auch in sneak_case; aber höchstens drei Wörter ;-)
ich_bestehe_aus_mehreren_woertern = IchBesteheAusMehrerenWoertern()

print(ich_bestehe_aus_mehreren_woertern)
```

```
TEST
<__main__.IchBesteheAusMehrerenWoertern object at 0x04E82E10>
```

[ ]:
```

```