Python - Objektorientiertes Programmieren

Jan Popko
Python Grundkurs

Business Trends Academy (BTA) GmbH
bta BUSINESS TRENDS ACADEMY
Nestorstraße 36
D-10709 Berlin
Tel.: +49 (0) 30 894 087 57
Fax: +49 (0) 30 895 429 94
Geschäftsführer:
Gabriele Fleischmann-Hahn
Maxi-Marlen Fleischmann
Hauptsitz des Unternehmens:
Nestorstraße 36, D-10709 Berlin
HRB 115251 B / Amtsgericht Berlin Charlottenburg
Steuer-Nr. 27/248/31179

Objektorientierung

Ein objektorientiertes Programm ist ein System von Objekten, die miteinander kommunizieren können. Wir haben schon mit Objekten gearbeitet, z. B. List oder Dictionary.

Jetzt geht es darum, eigene Klassen und zu definieren.

- **Eine Klasse ist ein Bauplan für Objekte**
- **Man kann sich eine Klasse als Fabrik vorstellen, die Objekte produziert**
- **Hier wird festgelegt, welche Eigenschaften und welches Verhalten ein Objekt hat**
- **Eine Klasse kann beliebig viele Objekte erzeugen**
- **Ein Objekt gehört jedoch immer zu genau einer Klasse**

Objektorientierung

Wenn man sich die Klasse als Fabrik vorstellt, so ist das daraus erzeugte Objekt etwas ganz Konkretes. Man nennt Objekte auch Instanzen. Nehmen wir beispielsweise die Klasse Geld.
Konkrete Geldobjekte sind: ein Geldschein, eine Münze, Kosten für einen Mietwagen…

Alle diese Objekte haben gewisse gemeinsame Merkmale, z. B. einen Betrag und eine Währungsangabe.

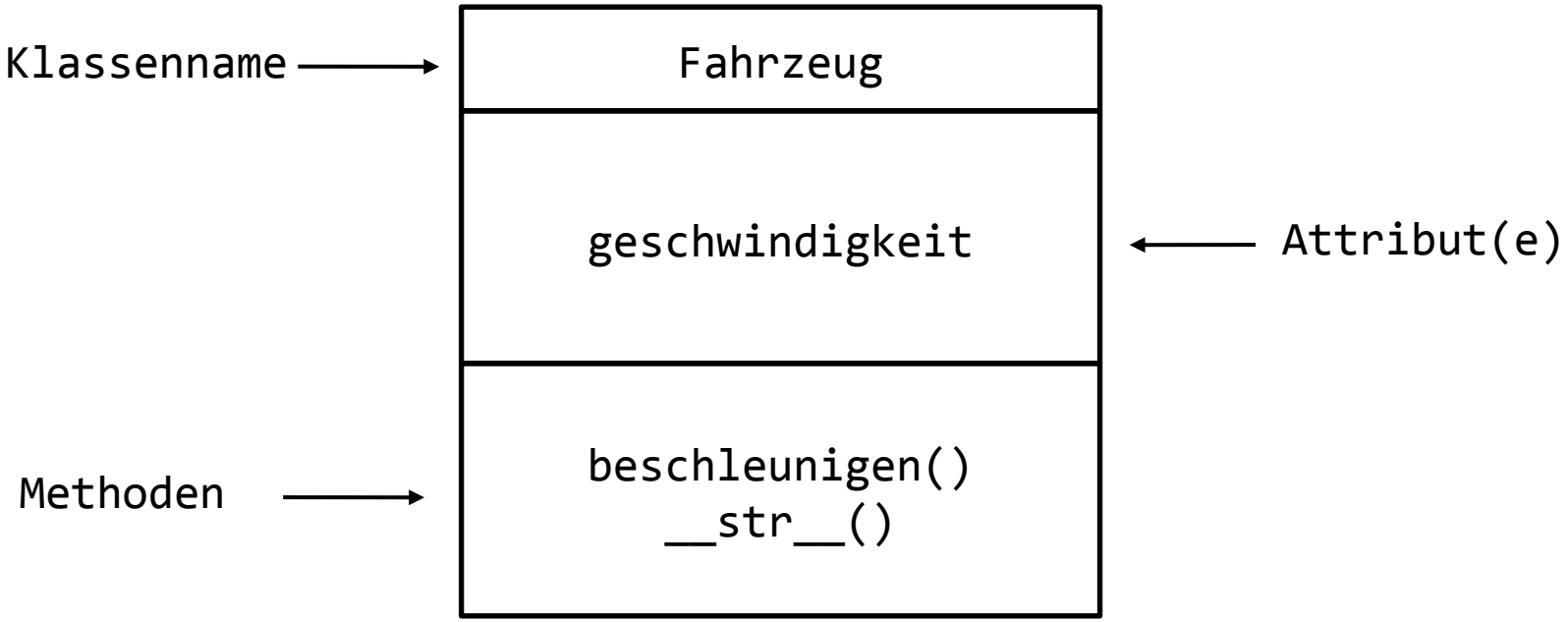
Diese Merkmale nennt man Attribute (Eigenschaften) eines Objekts.

Ein Attribut besteht aus einem Namen und einem Wert.

Objekte können außerdem Operationen ausführen. Man nennt diese Operationen Methoden.

Methoden sind nichts anderes als Funktionen, die innerhalb einer Klasse definiert sind.

Klasse in UML



Klasse in Python

Für unsere Klasse legen wir uns einen neuen Ordner namens **module** an. Die Klasse speichern wir unter dem Namen **fahrzeug.py** in dem Ordner module ab.

```
class Fahrzeug:

    # __init__ wird beim Erzeugen des Objekts aufgerufen und setzt
    # die Attribute, self ist immer das konkrete Objekt
    def __init__(self, x):
        self.geschwindigkeit = x

    def beschleunigen(self, x):
        self.geschwindigkeit += x

     # Liefert einen String, welcher das Objekt repräsentiert, wird das
     # Objekt mit der print() Methode ausgegeben, wird diese Methode
     # aufgerufen
    def __str__(self):
        return str(self.geschwindigkeit)
```

Hauptprogramm in Python

Nun benötigen wir ein Hauptprogramm. Wir erstellen uns eine neue Python-Datei mit dem Namen **mainFahrzeug.py.**

Diese Datei nicht mit in den module-Ordner legen, sondern einen Ordner darüber.

Hauptprogramm in Python

```
from module.fahrzeug import Fahrzeug

# neues Fahrzeug-Objekt namens fz1 wird erzeugt
fz1 = Fahrzeug(0)
# Zugriff auf das Attribut des Objekts
print(fz1.geschwindigkeit)
# Aufruf der Methode beschleunigen() des Objekts fz1 und
# Übergabe des Werts 100
fz1.beschleunigen(100)
print(fz1.geschwindigkeit)
fz1.beschleunigen(70)
print(fz1.geschwindigkeit)

# ruft die __str__() Methode des Objekts auf
print(fz1)
```

Sichtbarkeit

Die Attribute sind die Eigenschaften eines Objekts. Der Zugriff auf die Attribute kann gesteuert werden.

öffentliche Attribute
auf diese Attribute kann von überall aus zugegriffen werden

private Attribute

- stark private Attribute beginnen mit 2 Unterstrichen (__name)
  auf diese Attribute kann nur innerhalb der Klasse zugegriffen werden.

- schwach private Attribute beginnen mit einem Unterstrich (_age)

Sichtbarkeit

Fahrzeugklasse fahrzeug2.py mit privaten Attributen und getter-Methoden:

```
class Fahrzeug:

    # falls kein Startwert übergeben wird, wird die geschwindigkeit
    # auf 0 gesetzt, das Attribut geschwindigkeit wird auf privat gesetzt
    def __init__(self, geschwindigkeit = 0):
        self.__geschwindigkeit = geschwindigkeit

    # da man jetzt von außen nicht mehr auf das Attribut geschwindigkeit
    # zugreifen kann, benötigt man eine getter-Methode
    def getGeschwindigkeit(self):
        return self.__geschwindigkeit

    def beschleunigen(self, geschwindigkeit):
        self.__geschwindigkeit += geschwindigkeit

    # Liefert einen String, welcher das Objekt repräsentiert, wird das
    # Objekt mit der print() Methode ausgegeben, wird diese Methode
    # aufgerufen
    def __str__(self):
        return str(self.__geschwindigkeit)
```

Sichtbarkeit

Hauptprogramm:

```
from module.fahrzeug2 import Fahrzeug

# neues Fahrzeug-Objekt namens fz1 wird erzeugt
fz1 = Fahrzeug()
# Zugriff auf das Attribut des Objekts -> produziert einen Fehler
print(fz1.geschwindigkeit) #FEHLER!!!
print(fz1.getGeschwindigkeit())
# Aufruf der Methode beschleunigen() des Objekts fz1 und Übergabe
# des Werts 100
fz1.beschleunigen(100)
print(fz1.getGeschwindigkeit())
fz1.beschleunigen(70)
print(fz1.getGeschwindigkeit())

# ruft die __str__() Methode des Objekts auf
print(fz1)
```

Vererbung

Vererbung beschreibt die Beziehung zwischen einer Basisklasse und einer Unterklasse.

Die Unterklasse erbt **alle Attribute und alle Methoden** der Oberklasse.

Die Unterklasse hat meist **zusätzliche** Attribute und Methoden.

Wir wollen jetzt eine Klasse Lkw bauen, welche von der Basisklasse Fahrzeug (modul fahrzeug2) erbt.

```
class Lkw(Fahrzeug): # die Klasse Lkw erbt von der Klasse Fahrzeug
…
```

Vererbung

```
# Oberklasse wird eingebunden
from module.fahrzeug2 import Fahrzeug

class Lkw(Fahrzeug):
    # die __init__ Methode der Oberklasse wird in der __init__ Methode
    # der Unterklasse aufgerufen und ein neues Attribut beladung kommt hinzu
    def __init__(self, x = 0, y = 0):
        Fahrzeug.__init__(self, x)
        self.__beladung = float(y)

    def getBeladung(self):
        return self.__beladung

    def beladen(self, beladung):
        self.__beladung += beladung

   # auf geschwindigkeit kann nur über die Methode zugegriffen werden, weil das
   # Attribut privat gesetzt ist
    def __str__(self):
         return f"Geschwindigkeit :{self.getGeschwindigkeit()}  \nBeladung:
{self.__beladung} "
```

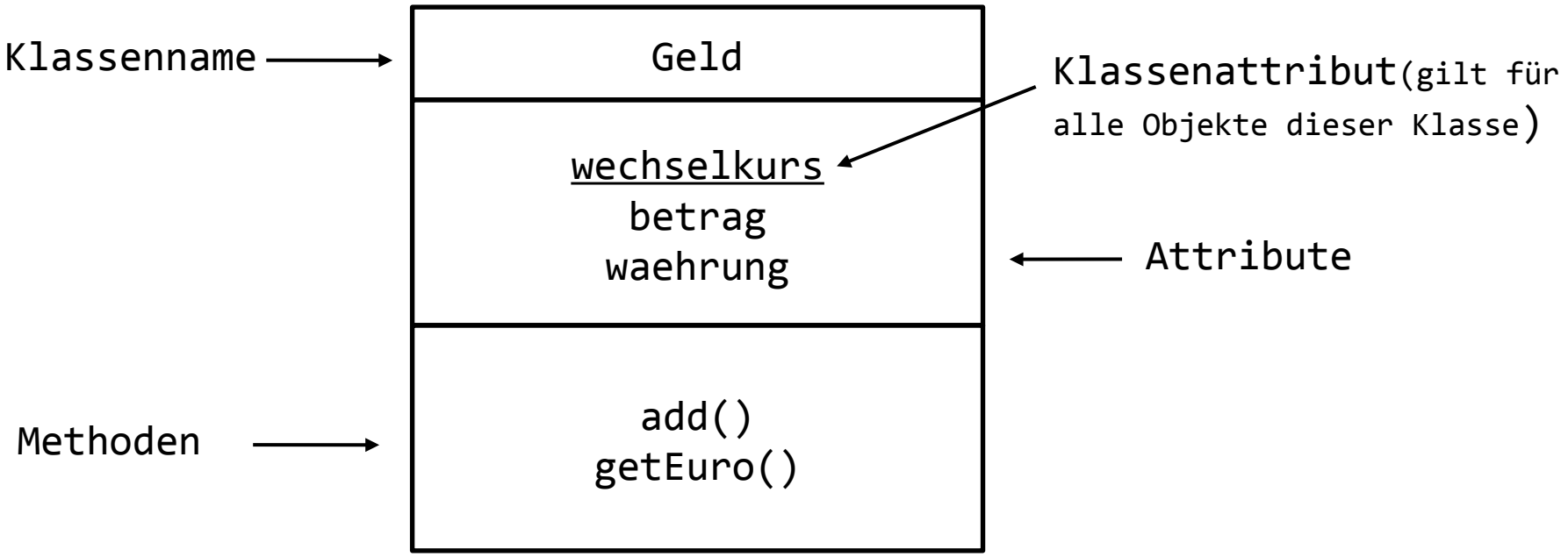
Vererbung

```
from module.lkw import Lkw

# neues Lkw-Objekt namens lkw wird erzeugt
lkw = Lkw()
# Zugriff auf die Methode getGeschwindigkeit() des Objekts
print(lkw.getGeschwindigkeit())
# Aufruf der Methode beschleunigen() des Objekts lkw und Übergabe
# des Werts 100
lkw.beschleunigen(100)
print(lkw.getGeschwindigkeit())
# Aufruf der Methode beladen() des Objekts lkw und Übergabe
# des Werts 100
lkw.beladen(5)
print(lkw.getBeladung())
lkw.beladen(7.5)

# ruft die __str__() Methode des Objekts auf
print(lkw)
```

Klasse in UML



Klasse in Python

Für unsere Klasse legen wir uns einen neuen Ordner namens **module** an. Die Klasse speichern wir unter dem Namen **geld.py** in dem Ordner module ab.

```
class Geld:

    wechselkurs={'USD':0.88, 'GBP':1.11, 'EUR':1.0}

    def __init__(self, waehrung, betrag):
        self.waehrung = waehrung
        self.betrag = float(betrag)

    def getEuro(self):
        return self.betrag*self.wechselkurs[self.waehrung]
```

Hauptprogramm in Python

Nun benötigen wir ein Hauptprogramm. Wir erstellen uns eine neue Python-Datei mit dem Namen **mainGeld.py. (nicht mit in den module-Ordner speichern)**

Hauptprogramm in Python

```
from module.geld import Geld

# neues Geld-Objekt namens mietwagen wird erzeugt
mietwagen = Geld('USD', 100)
# Zugriff auf die Attribute des Objekts
print(mietwagen.betrag)
print(mietwagen.waehrung)
# Zugriff auf die Methode des Objekts
print(mietwagen.getEuro())

# neues Geld-Objekt namens uebernachtung wird erzeugt
uebernachtung = Geld('GBP', 150)
# Zugriff auf die Attribute des Objekts
print(uebernachtung.betrag)
print(uebernachtung.waehrung)
# Zugriff auf die Methode des Objekts
print(uebernachtung.getEuro())
```

Sichtbarkeit

Geldklasse mit privaten Attributen und getter-Methoden:

```
class Geld:

    __wechselkurs={'USD':0.88, 'GBP':1.11, 'EUR':1.0}

    def __init__(self, waehrung, betrag):
        self.__waehrung = waehrung
        self.__betrag = float(betrag)

    def getWaehrung(self):
        return self.__waehrung

    def getBetrag(self):
        return self.__betrag

    def getEuro(self):
        return self.__betrag*self.__wechselkurs[self.__waehrung]
```

Sichtbarkeit

setter Methoden:

```
class Geld:
…
    def setWaehrung(self, neueWaehrung):
        if neueWaehrung in self.__wechselkurs.keys():
            alt = self.__wechselkurs[self.__waehrung]
            neu = self.__wechselkurs[neueWaehrung]
            self.__betrag = alt/neu * self.__betrag
            self.__waehrung = neueWaehrung
    else:
         print("Diese Währung ist nicht vorhanden")

    def setBetrag(self, neuerBetrag):
        self.__betrag = float(neuerBetrag)

…
```

Sichtbarkeit

Aufruf der setter Methoden:

```
from module.geld2 import Geld

# neues Geld-Objekt namens mietwagen wird erzeugt
mietwagen = Geld('USD', 100)
# Zugriff auf die Attribute des Objekts
print(mietwagen.getBetrag())
print(mietwagen.getWaehrung())
# Zugriff auf die Methode des Objekts
print(mietwagen.getEuro())

mietwagen.setBetrag(300)
mietwagen.setWaehrung('GBP')
print(mietwagen.getBetrag())
print(mietwagen.getWaehrung())
```

Vererbung

```
import time
from module.geld2 import Geld

class Konto(Geld):

    def __init__(self, waehrung, inhaber):
        Geld.__init__(self,waehrung, betrag = 0)
        self.__inhaber = inhaber

    def einzahlen(self,waehrung, neuerBetrag):
        self.setWaehrung(waehrung)
        einzahlung = self.getBetrag() + neuerBetrag
        self.setBetrag(einzahlung)

    def auszahlen(self, waehrung, neuerBetrag):
        self.setWaehrung(waehrung)
        self.setBetrag(self.getBetrag() - neuerBetrag)


    def __str__(self):
        return 'Konto von ' + self.__inhaber + \
               ':\nKontostand am ' + \
               time.asctime()+ ': '+ self.getWaehrung() + ' ' +\
               format (self.getBetrag(), '.2f')
```