Objektorientierte Programmierung – Teil 2

Im ersten Teil zur objektorientierten Programmierung gab es noch einige Sachen, die unerwähnt blieben und trotzdem wichtig sind. So erlaubt uns diese Programmierung noch ein paar weitere Feinheiten bei der Verwendung von Klassen sowie auch Beziehungen zwischen den Klassen zu definieren.

Kapselung von Daten

Häufig ist es sinnvoll, dass die Daten innerhalb eines Objekts (vorgegeben durch die Klasse) nicht von jeder x-beliebiger Stelle direkt verändert werden können. Angenommen wir haben ein Objekt Kunde mit dem Attribut Geburtsdatum. So bietet es sich an, dass die Änderung dieses Attributes nur durch eine intere Methode der Klasse durchgeführt wird. Dies hätte dann den Vorteil, dass zum Beispiel immer geprüft werden kann, ob es sich um ein gültiges Datum handelt. Dies wäre nicht möglich, wenn der Wert des Attributes durch alle möglichen Eingabequellen direkt verändert werden kann.

Mehr Sicherheit bietet daher die Möglichkeit, dass Attribute als *public* (öffentlich), *protected* (gesichert) oder *private* (privat) deklariert werden können:

- **public**
  Diese Variablen besitzen keine Unterstriche vor dem Namen und können sowohl innerhalb des Objekt als auch von außen gesehen und verändert werden.
- **protected**
  Variablen, die einen einzigen Unterstrich vor ihrem Namen haben, sind protected. Das bedeutet, dass man diese zwar auch von innen und außen heraus sehen und bearbeiten kann, aber es laut Entwickler nicht sollte. Der Entwickler möchte damit zeigen, dass es sich um nicht veränderbare Werte (Konstanten) handelt.
- **private**
  Diese Variablen besitzen genau zwei Unterstriche vor ihrem Namen. Sie können nur innerhalb des Objekts verwendet werden und sind nicht einmal von außen zu sehen.

Hier ein kleines Beispiel:

```
class Konto:
    def __init__(self, inhaber, kontostand, kontonummer):
        self.__inhaber = inhaber
        self.__kontostand = kontostand
        self.__kontonummer = kontonummer

    def gib_kontostand(self):
        return self.__kontostand

    def auszahlen(self, betrag):
        if (betrag <= self.__kontostand):
            return True
            self.__kontostand -= betrag
        else:
            return False

    def einzahlen(self, betrag):
        self.__kontostand += betrag

konto = Konto('Max Mustermann', 500, 1234567890)
print('Ich habe ' + str(konto.gib_kontostand()) + ' Euro')
```

```
konto.__kontostand = 1000000
if konto.auszahlen(1000000):
    print('Ich bin Millionär!')
else:
    print('Ich bin immer noch kein Millionär...')
```

Ausgeführt wäre das Ergebnis dann:

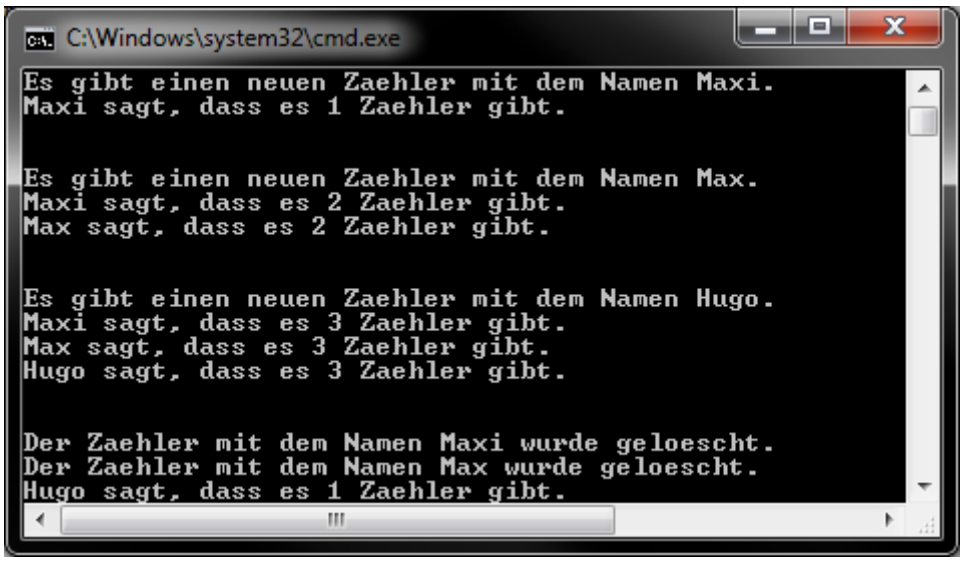
Statische Attribute

Bisher besaßen zwei Objekte der selben Klasse meist zwar das gleiche Attribut nur mit unterschiedlichen Werten. Für die Programmeirung ist es manchmal auch sinnvoll, wenn es ein Attribut gibt, dass bei allen Objekten immer den gleichen Wert besitzt. Solche Attribute nennt man *static* (statisch).

In dem folgenden Beispiel gibt es das Attribut objekt_zaehler. Dieses soll immer angeben, wie viele Objekte es von der Klasse gibt.

```
class Zaehler:
    zaehler = 0

    def __init__(self, name):
        self.name = name
        Zaehler.zaehler += 1
        print('Es gibt einen neuen Zaehler mit dem Namen ' + \
              self.name + '.')

    def nenne_anzahl(self):
        print(self.name + ' sagt, dass es ' + str(Zaehler.zaehler) \
              + ' Zaehler gibt.')

    def __del__(self):
        print('Der Zaehler mit dem Namen ' + self.name + \
              ' wurde geloescht.')
        Zaehler.zaehler -= 1

mein_zahler1 = Zaehler('Maxi')
mein_zahler1.nenne_anzahl()
print('\n')
mein_zahler2 = Zaehler('Max')
mein_zahler1.nenne_anzahl()
mein_zahler2.nenne_anzahl()
print('\n')
mein_zahler3 = Zaehler('Hugo')
mein_zahler1.nenne_anzahl()
mein_zahler2.nenne_anzahl()
mein_zahler3.nenne_anzahl()
print('\n')
del(mein_zahler1)
del(mein_zahler2)
mein_zahler3.nenne_anzahl()
```

Ausgeführt wäre das Ergebnis dann:



Vererbung

Betrachten wir nun erst einmal wieder das Beispiel mit den Pizzen. Es gab eine Klasse Pizza mit Standardteig und -zutaten. Nun gibt es bei vielen Lieferdiensten unterschiedliche Größen von Pizzen. Da wären zum Beispiel die kleinen Pizzen, die Normalgroßen oder Familienpizzen. Bei allen handelt es sich um Pizza und auf alle können die gleichen Zutaten enthalten, aber trotzdem gibt es Unterschiede wie zum Beispiel den Preis. Angenommen jede Zutat kostet bei einer kleinen Pizza 50 Cent, bei einer Normalen 1 Euro und bei einer Familienpizza 1,50 Euro. Die Berechnung des Preises lässt sich verschieden lösen:

1. Wir erweitern die Klasse *Pizza* um ein Attribut *groesse*. Anschließend kann in der Methode *preis_berechnen()* geprüft werden, welche Größe die Pizza hat und der Preis entsprechend ermittelt werden.
2. Wir nutzen Vererbung. Hierbei werden von der Klasse *Pizza* die Unterklassen *KleinePizza*, *NormalePizza* und *FamilienPizza* erzeugt. Dies hätte den Vorteil, dass kein weiteres Attribut notwendig ist und man kann einzelne Methoden unterschiedlich definieren und programmieren.

Der folgende Quellcode zeigt die Lösung 1:

```
class Pizza:
    def __init__(self):
        self.teig = 'Hefeteig'
        self.zutaten = ['Tomatensosse', 'Kaese']
        self.rand = 'ohne'
        self.groesse = 'normal'

    def bestellen(self):
        print('Pizza mit ' + self.teig + ', ' + self.rand + \
              '-Rand wird bestellt.')
        print('Zutaten-Liste:')
        print(self.zutaten)
        print('Preis in Euro:')
        print(str(self.berechne_preis()))

    def berechne_preis(self):
        if (self.groesse == 'klein'):
            return len(self.zutaten) * 0.5
        elif (self.groesse == 'normal'):
            return len(self.zutaten) * 1
```

```
        elif (self.groesse == 'familie'):
            return len(self.zutaten) * 1.5

meine_pizza = Pizza()
meine_pizza.zutaten.append('Salami')
meine_pizza.zutaten.append('Schinken')
meine_pizza.bestellen()
```

Der folgende Quellcode zeigt die Lösung 2:

```
class Pizza:
    def __init__(self):
        self.teig = 'Hefeteig'
        self.zutaten = ['Tomatensosse', 'Kaese']
        self.rand = 'ohne'

    def bestellen(self):
        print('Pizza mit ' + self.teig + ', ' + self.rand + \
              '-Rand wird bestellt.')
        print('Zutaten-Liste:')
        print(self.zutaten)
        print('Preis in Euro:')
        print(str(self.berechne_preis()))

    def berechne_preis(self):
        pass

class KleinePizza(Pizza):
    def berechne_preis(self):
        return len(self.zutaten) * 0.5

class NormalePizza(Pizza):
    def berechne_preis(self):
        return len(self.zutaten) * 1

class FamilienPizza(Pizza):
    def berechne_preis(self):
        return len(self.zutaten) * 1.5

meine_pizza = NormalePizza()
meine_pizza.zutaten.append('Salami')
meine_pizza.zutaten.append('Schinken')
meine_pizza.bestellen()
```

KleinePizza ist eine Unterklasse von Pizza.

Ähnlich zu Methoden wird die Mutterklasse bei der Definierung der Unterklassen übergeben. Eine Klasse kann auch von mehreren Klassen erben.

Methoden können in Unterklassen anders definiert bzw. programmiert werden. Alles andere bleibt wie in der Mutterklasse.

Generell lässt sich hier nun nicht sagen, welche Lösung besser ist. Wichtig für das Verständnis von Vererbung ist noch, dass die Unterklassen alle Attribute und Methoden der Mutterklasse besitzen. Der Vorteil ist aber, dass diese Unterklassen unterschiedlich voneinander erweitert oder verändert werden können.

Entwurfsmuster

Ein weiterer fundamentaler Vorteil, den die objektorientierte Programmierung mit sich bringt, ist die Verwendung von Entwurfsmustern. Hierbei handelt es sich um Lösungsansätze für wiederkehrende Entwurfsprobleme bei der Programmierung (besonders bei großen und komplexen Projekten). Es gibt sehr viele verschiedene solcher Entwurfsmuster für die unterschiedlichsten Probleme, die zum Teil auch untereinander kombiniert werden können. Eines, dass hier erwähnt werden soll, ist das *Model-View-Controller* (MVC) Entwurfsmuster.

Beim MVC werden bestimmte Bereiche eines größeren Programms voneinander getrennt, damit die Entwicklungs in größeren Teams erleichtert und mehr Übersicht über den zu entstehenden Quellcode ermöglicht wird. Programme werden beim MVC in das Model, das

View und dem Controller unterteilt ist. Dabei ist nicht ausgeschlossen, dass ein Programm nicht über mehrere Models, Views und/oder Controller verfügen kann.

- **Controller**
  Der Controller beinhaltet die Hauptlogik des Programms. Er steuert das ganze Programm und gibt Anweisungen an das Model und den View.
- **Model**
  Das Model dient als Datenquelle für das Programm. Das bedeutet, dass alle notwendigen Informationen in dieser Klasse enthalten sind und der Controller sich diese vom Model holt.
- **View**
  Der View ist für die visuellen Rückgaben an den Benutzer zuständig. Dies kann zum Beispiel das Zusammenbauen und Anzeigen einer grafischen Oberfläche sein. Änderungen an dieser grafischen Oberfläche werden dann meist vom Controller angestoßen und durch den View ausgeführt.

Miteinander verbunden bieten diese Klassen also eine Trennung von Daten, Programmlogik und Ausgabe. Wie bereits gesagt, kann dies den Vorteil haben, dass in einem größeren Projekt sich einzelne Entwickler zum Beispiel nur auf die Bereitstellung, Validierung und Verarbeitung von Daten befassen (also nur am Model arbeiten). Anders könnte es aber auch sein, dass sich einzelne Entwickler nur mit der Gestaltung der grafischen Oberfläche befassen (also nur am View arbeiten). Im letzten Fall könnte es dann auch einzelne Entwickler geben, die sich nur mit der Programmlogik befassen (also nur am Controller arbeiten). Wichtig bei größeren Projekten ist hier, dass alle Entwickler genau miteinander kommunizieren und das Programm auch in Gänze planen. Die letztliche Umsetzung kann dann in kleinere Gruppen unterteilt werden.