Objektorientierte Programmierung

OOP Kennenlernen

Vorwort

Bei kleinen Programmen ist es immer fraglich, ob OOP die beste und einzige Lösung ist. Meist reicht es prozedural mit gut benannten Funktionen zu arbeiten oder diese in Module aufzuteilen.

Bei grösseren Programmen, oder gar (externen) Bibliotheken kann OOP auch sehr hilfreich sein. Daher nun eine kleine Einführung in das Thema.

Begriffe

- **`Vererbung`**: Beschreibt eine Beziehung (z.B.: ein Kreis ist eine Form)
- **`Komposition`**: Beschreibt eine "hat eine Beziehung" (z.B. ein Auto hat einen Motor)

Begriffe

- **`Klasse`**: ein Konzept oder ein Entwurf für ein reales Objekt in der Programmierung
- **`Instanz`**: konkretes Objekt einer Klasse, das im Programm verwendet wird

```
class Shape:
    pass
```

```
shape = Shape()
```

Klassen, Objekte und Methoden

Bei der objektorientierten Programmierung werden Klassen erzeugt, in denen die **Eigenschaften** und Methoden **von Objekten** festgelegt werden.

Methoden sind Funktionen, die nur auf Objekte der betreffenden Klasse angewendet werden können. Die Eigenschaften und die Methoden bilden zusammen die **Member** einer Klasse.

Klassen, Objekte und Methoden

Man kann verschiedene Objekte dieser Klassen erzeugen, den Eigenschaften unterschiedliche Werte zuweisen und die Methoden auf die Objekte anwenden.

Die Definitionen aus der Klasse und die zugewiesenen Werte begleiten die Objekte über ihren gesamten Lebensweg während der Dauer des Programms.

Klassen, Objekte und Methoden

Ein Beispiel: Es wird die eigene Klasse **`Car`** erschaffen, in der die Eigenschaften und die Methoden von Fahrzeugen bestimmt werden.

Ein Fahrzeug hat unter anderem die Eigenschaften **`Bezeichnung, Geschwindigkeit und Fahrtrichtung`**.

Ausserdem kann man ein Fahrzeug **`beschleunigen`** und **`lenken`**.

Innerhalb des Programms können viele unterschiedliche Fahrzeuge erschaffen und eingesetzt werden.

Klassen, Objekte und Methoden

Klassen können ihre Eigenschaften und Methoden ausserdem vererben.

Eine solche Klasse fungiert als **`Basisklasse`**, ihre Erben nennt man **`abgeleitete Klassen`**.

Damit kann die Definition von ähnlichen Objekten, die über eine Reihe von gemeinsamen Eigenschaften und Methoden verfügen, vereinfacht werden.

Klassen, Objekte und Methoden

Als Beispiel definieren wir nun die eigene Klasse Fahrzeug. Das Fahrzeug erhält zunächst nur die Eigenschaft velocity (geschwindigkeit) und die Methoden acceleration() (Beschleunigung) und output() (Ausgabe).

```
1 # 30-car.py
2
3 class Car:
4   velocity = 0
5   def acceleration(self, value):
6     self.velocity += value
7   def output(self):
8     print("Velocity:", self.velocity)
```

Klassen, Objekte und Methoden

Die Definition der eigenen Klasse wird eingeleitet vom Schlüsselwort **`class`**, gefolgt vom Namen der Klasse und einem Doppelpunkt.

Der Name einer eigenen Klasse sollte gemäss Konvention mit einem **`grossen Buchstaben`** beginnen.

`Die weiteren Zeilen der Definition werden eingerückt.`

Die Eigenschaft **`acceleration`** wird definiert und auf den Wert 0 gesetzt.

Klassen, Objekte und Methoden

Methoden werden wie Funktionen mithilfe des Schlüsselworts **`def`** definiert.

Methoden haben mindestens einen Parameter, **`nämlich eine Referenz auf das Objekt selbst`**.

Gemäss Konvention wird als Referenz **`self`** verwendet, es kann aber auch ein anderer Name gewählt werden.

Besondere Member

- Man kann eine besondere Methode mit dem festgelegten Namen **`__init__()`** als **Konstruktormethode** definieren, die ein Objekt zu Beginn seiner Lebensdauer mit Anfangswerten initialisiert.
- Man kann eine besondere Methode mit dem festgelegten Namen **`__str__()`** definieren, die die Eigenschaften eines Objekts ausgibt.

Besondere Member

- Die Eigenschaft **`__dict__`** stellt ein Dictionary mit den Namen und Werten der Eigenschaften bereit.
- Man kann eine besondere Methode mit dem festgelegten Namen **`__del__()`** als **Destruktormethode** definieren, die am Ende der Lebensdauer eines Objekts Aktionen auslöst, zum Beispiel das Schliessen einer offenen Datei.

Besondere Member

```
# 30-vehicle.py
class Vehicle:
    def __init__(self, speed):
        self.speed = speed

    def accelerate(self, value):
        self.speed += value

    def printout(self):
        print("Speed:", self.speed)

volvo = Vehicle(0)
opel = Vehicle(0)
volvo.printout()
volvo.accelerate(20)
volvo.printout()
opel.printout()
```

Besondere Member

Noch mehr besondere Member sieht man in 30-more-specials.py.
Bitte diese Datei anschauen und versuchen zu verstehen.

Operatorenmethoden

Operatormethoden werden für Objekte mithilfe von Operatoren aufgerufen.

Sie sind für die eingebauten Datentypen bereits vordefiniert.

Für Ihre eigenen Datentypen, also Ihre Klassen, kann man sie selbst definieren. (Muss man aber nicht!)

Operatorenmethoden

```python
# 30-operators.py

class Vehicle:
    def __init__(self, br, ac):
        self.brand = br
        self.acceleration = ac

    def __gt__(self, other):
        return self.acceleration > other.acceleration

    def __eq__(self, other):
        return self.acceleration == other.acceleration

    def __sub__(self, other):
        return self.acceleration - other.acceleration
```

Operatorenmethoden

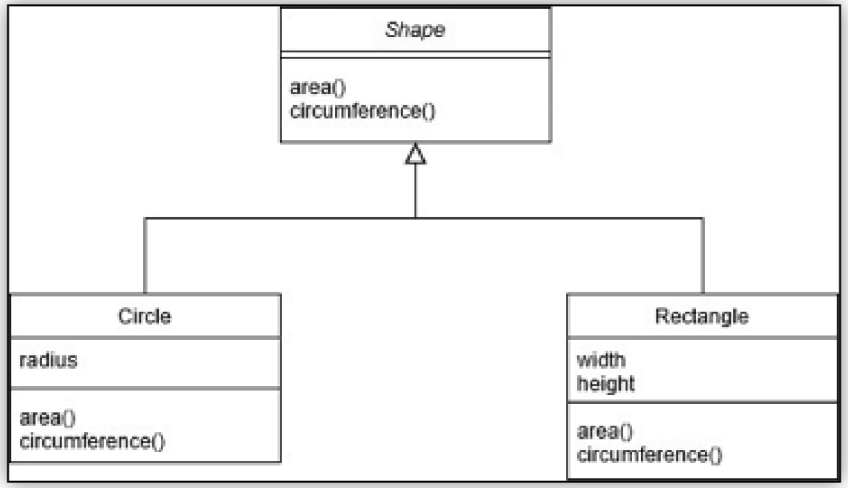
- >= führt zu __ge__(), greater equal
- < führt zu __lt__(), lower than
- <= führt zu __le__(), lower equal
- != führt zu __ne__(), not equal
- + führt zu __add__()
- * führt zu __mul__()
- / führt zu __truediv__()
- // führt zu __floordiv__()
- % führt zu __mod__()
- ** führt zu __pow__()

Vererbung

Eine Klasse kann ihre Eigenschaften und Methoden an eine andere Klasse vererben.

Dieser Mechanismus wird **`häufig angewendet`**.

Damit erzeugt man eine Hierarchie von Klassen, die die Darstellung von Objekten ermöglicht, die **`ähnliche Merkmale aufweisen`**.



Vererbung

Eine Klasse kann ihre Eigenschaften und Methoden an eine andere Klasse vererben.

Dieser Mechanismus wird häufig angewendet.

Damit erzeugt man eine Hierarchie von Klassen, die die Darstellung von Objekten ermöglicht, die **`ähnliche Merkmale aufweisen`**.

Vererbung

Die Klasse Vehicle:

```python
# 30-vehicle-car.py

class Vehicle:
    def __init__(self, br, ac):
        self.brand = br
        self.acceleration = ac

    def accelerate(self, value):
        self.acceleration += value

    def __str__(self):
        return f"{self.brand} {self.acceleration} km/h"
```

Vererbung

Die Klasse Car:

```python
# 30-vehicle-car.py

class Car(Vehicle):
    def __init__(self, br, ac, pa):
        super().__init__(br, ac)
        self.passenger = pa

    def __str__(self):
        return f"{super().__str__()} {self.passenger} passengers"

    def board(self, num):
        self.passenger += num

    def exit(self, num):
        self.passenger -= num
```

Vererbung

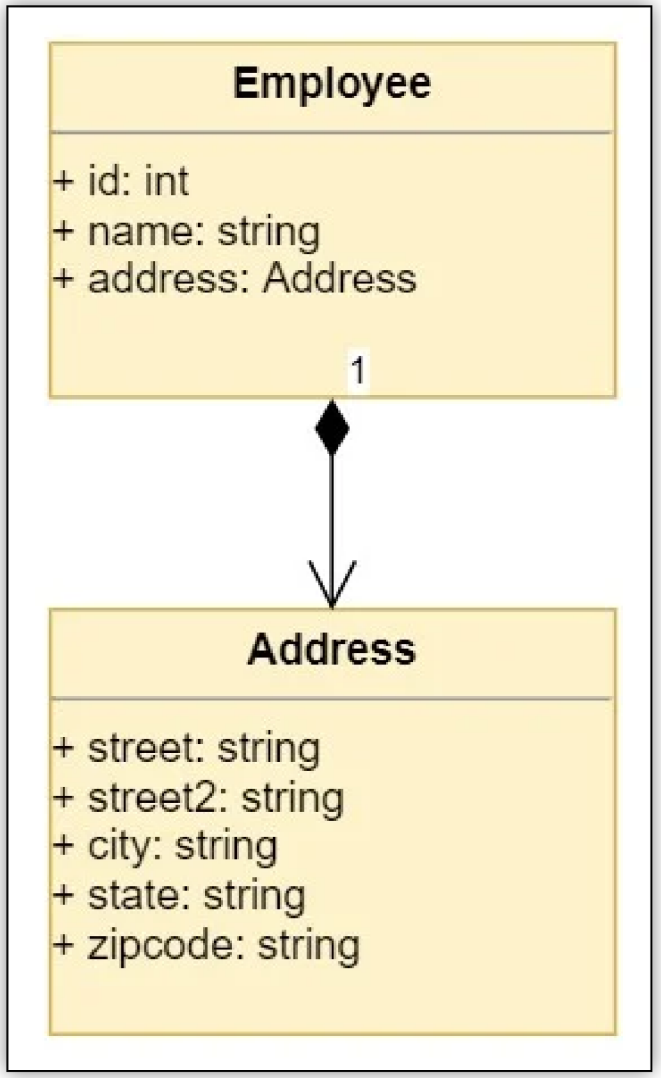
Zum Schluss noch der Aufruf:

```
1 # 30-vehicle-car.py
2
3 fiat = Car("Fiat Marea", 50, 0)
4 fiat.board(3)
5 fiat.exit(1)
6 fiat.accelerate(10)
7 print(fiat)
```

Komposition

Der Vollständigkeit halber soll auch noch kurz die Komposition ("has a" Beziehung") vorgestellt werden.

Diese wird weniger häufig verwendet als die Vererbung.



Komposition

Zuerst die Adresse

```python
# contacts.py
class Address:
    def __init__(self, street, city, state, zipcode, street2=''):
        self.street = street
        self.street2 = street2
        self.city = city
        self.state = state
        self.zipcode = zipcode

    def __str__(self):
        lines = [self.street]
        if self.street2:
            lines.append(self.street2)
        lines.append(f'{self.city}, {self.state} {self.zipcode}')
        return '\n'.join(lines)
```

Komposition

Danach die Klasse Employee (mit Address im Gepäck)

```python
# 30-employee.py

from contacts import Address

class Employee:
    def __init__(self, id, name, address):
        self.id = id
        self.name = name
        self.address = address

    def __str__(self):
        return f"I am: {self.name} id: {self.id} from: {self.address}"
```

Komposition

Dann der Aufruf (inkl. Addresse als Komposition)

```
# 30-employee.py

manager = Employee(1, 'Mary Poppins', 3000)
manager.address = Address(
    '121 Admin Rd',
    'Concord',
    'NH',
    '03301'
)

print(manager)
```

Mehrfachvererbung

Eine abgeleitete Klasse kann wiederum Basisklasse für eine weitere Klasse sein. Dadurch ergibt sich eine Vererbung über mehrere Ebenen.

Bei der Mehrfachvererbung kann eine Klasse ausserdem **`von mehr als einer Klasse erben`**.

Sie übernimmt in diesem Fall die Eigenschaften und **`Methoden aller Basisklassen`**.

Im folgenden Beispiel werden drei Klassen definiert:

Mehrfachvererbung

Die Klasse Date:

```
 1 # 30-multi-inherit.py
 2
 3 class Date:
 4     def __init__(self, d, m, y):
 5         self.day = d
 6         self.month = m
 7         self.year = y
 8
 9     def __str__(self):
10         return f"{self.day:02d}.{self.month:02d}.{self.year:d}"
```

Mehrfachvererbung

Die Klasse Time:

```
 1 # 30-multi-inherit.py
 2
 3 class Time:
 4     def __init__(self, h, m, s):
 5         self.hour = h
 6         self.minute = m
 7         self.second = s
 8
 9     def __str__(self):
10         return f"{self.hour:02d}:{self.minute:02d}:{self.second:02d}"
```

Mehrfachvererbung

Die Klasse Period:

```
1 # 30-multi-inherit.py
2
3 class Period(Date, Time):
4     def __init__(self, d, mo, y, h, mi, s):
5         Date.__init__(self, d, mo, y)
6         Time.__init__(self, h, mi, s)
7
8     def __str__(self):
9         return f"{Date.__str__(self)} {Time.__str__(self)}"
```

Mehrfachvererbung

Und schlussendlich der Aufruf

```
1 # 30-multi-inherit.py
2
3 d = Date(3, 1, 2022)
4 print(d)
5 u = Time(16, 5, 20)
6 print(u)
7 z = Period(9, 5, 2022, 9, 35, 8)
8 print(z)
```

Enumerationen

Enumerationen sind Aufzählungen von Konstanten.

Der Programmcode wird durch die Nutzung der Elemente einer Enumeration besser lesbar.

Python bietet seit der Version 3.4 innerhalb des Moduls enum unter anderem die Klasse IntEnum zur Erstellung einer Enumeration, deren Elemente als Konstanten für ganze Zahlen stehen.

Enumerationen

```python
# 30-enum.py

import enum

class Color(enum.IntEnum):
    red = 5
    yellow = 2
    blue = 4


x = 2
if x == Color.yellow:
    print("This is yellow")

print(Color.yellow)
print(Color.yellow * 10)
```

Übung: self, super, __init__

Schauen Sie sich
kurz folgendes Beispiel an:

```python
class Animal:
    def __init__(self, weight):
        self.weight = weight

    def info(self):
        print(f"I'm an abstract animal and I don't have a concept of weight.")

    def noise(self):
        print("I'm abstract, I don't make noises. Or maybe abstract noises...")


class Cat(Animal):
    def __init__(self, weight, color):
        super().__init__(weight)
        self.color = color

    def info(self):
        print(f"I'm a {self.color} cat that weights {self.weight} kg.")

    def noise(self):
        print("Meow")


class Dog(Animal):
    def __init__(self, weight, breed):
        super().__init__(weight)
        self.breed = breed

    def info(self):
        print(f"I'm a {self.breed} dog that weights {self.weight} kg.")

    def noise(self):
        print("Woof!")
```

Übung: self, super, __init__

Erstelle nun eine Datei namens shapes.py, und erledige folgendes:

- Erstellen Sie eine Shape-Klasse (**`Shape`**)
- Erstellen Sie eine Methode **`area`** und eine Methode **`circumference`** in der Shape-Klasse
- Erstelle eine Klasse **`Circle`** und eine Klasse **`Rectangle`**, die **`von Shape erben`**
- **`Circle`** braucht einen **`Radius`** für den Konstruktor (init)
- Rectangle **`braucht eine Breite und Höhe`** für den Konstruktor (init)
- Implementiere zudem Fläche und Umfang für Circle bzw. Shape

Ende

Das war alles für dieses Kapitel