Grundlagen Programmierung

(Begleitendes Jupyter-Notebook: Grundlagen_OOP)

Objektorientiertes Programmieren (OOP)

- **Klasse**: eine Kategorie, zu der viele ähnliche Objekte zugeordnet werden können
- **Objekt**: ein Gegenstand mit gewissen Merkmalen und Verhalten
- **Attribut**: Merkmale von Objekten oder Klassen heißen in Programmierung Attribute
- **Methode**: Gemeinsame Funktionen von Objekten einer Klasse
- **Vererbung**: eine Unterklasse (Sub-Class) erbt Attribute und Methoden von einer/mehreren Superklasse(n) (Super-Class)

In OOP-Paradigma versteht sich jeder Gegenstand im Universum als Objekt.

Jedes Objekt verfügt über gewisse Eigenschaften (Attribute) und Verhalten (Methoden).

Eine Ente beispielsweise hat zwei Flügel, hat Feder, fliegt und schwimmt.

Eine Klasse ist eine (Ober-) Kategorie, zu der viele ähnliche Objekte zugeordnet werden können.

Zum Beispiel die Klasse '**bird**' kann als eine Kategorie für alle Vögel betrachtet werden.

Objekte dieser Klasse haben *generell* Feder und Flügel, können auch *in der Regel* fliegen.

In der Informatik stellt der Begriff 'Klasse' eine Abstraktion dar, von der man **Objekte ableiten** kann.

Man sagt auch anstelle von Objekt eine **Instanz zu erzeugen**, oder **ein Objekt instanziieren**.

OOP in Python

Python unterstützt Objektorientierung. In Python ist *fast* alles ein Objekt.

Eine Klasse deklarieren

In Python kann eine Klasse mit Hilfe von Keyword 'class' definiert werden.

Den ersten Buchstaben eines Klassennamen schreibt man Groß:

```
class Book:
    pass
```

Anweisungen in einer Klassenstruktur müssen eingerückt sein. (Ähnlich wie bei Funktionen und Schleifen).

Ein Objekt erzeugen

```
b1 = Book( )
```

Datentyp (Datenklasse) mit type( ) ermitteln

```
>>> type(b1)
>>> <class '__main__'.Book>
```

b1 ist ein Objekt der Klasse "Book". Diese Klasse befindet sich in unserem Hauptprogramm.

Eine Klasse kann aus einem weiteren Modul in das Hauptprogramm mit import Befehl importiert werden.

Attribute deklarieren

Schauen wir uns folgendes Python-Dictionary an, sehen wir, dass drei Objekte c1, c2 und c3 jeweils zu einem weiteren Dictionary mit denselben Keys zugeordnet sind. Allerdings sind die Keys für jedes Objekt zu unterschiedlichen individuellen Werten (Values) zugeordnet. Diese Attributswerte sind dann in einem individuellen Dictionary für jedes Objekt gespeichert.

```
car = {
   'c1' : {'make':'mercedes', 'model':'s300', 'year':2020, 'color':'black' },
   'c2' : {'make':'bmw', 'model':'i3', 'year':2021, 'color':'blue' },
   'c3' : {'make':'audi', 'model':'q8', 'year':2019, 'color':'white' }
}
```

Die Konstruktor-Funktion __init__ in einer Python-Klasse, sorgt dafür, dass bei jedem Klassenaufruf (=Objektinstanziierung) alle vordefinierten Attribute jeweils einen Wert (Attributswert) zugewiesen bekommen.

```
class Car:
    # Konstruktorfunktion: verknüpft das Objekt mit Attributen
    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color
```

`self` bezieht sich auf das aktuelle Objekt aus der Klasse.
`self.make =` gegebener Wert zum Attribut 'make' zuordnen.

Ein Objekt erzeugen

```
c1 = Car('mercedes', 's300', 2020, 'black')
```

Instanzattribute einzeln auslesen

```
>>> print(c1.make)
mercedes

>>> print(c1.model)
s300
```

Zugriff auf alle Attribute eines Objektes

```
>>> print(c1.__dict__)
{'make': 'mercedes', 'model': 's300', 'year': 2020, 'color': 'black'}
```

Methoden

Methoden oder genauer gesagt Instanz-Methoden sind Funktionen, die das Verhalten eines Objektes darstellen. Eine Instanz-Methode bezieht sich immer auf ein Objekt der Klasse, kann aber auch weitere Argumente akzeptieren.

Getter Methode: liefert den aktuellen Wert eines Attributes zurück
Setter Methode: ersetzt den aktuellen Wert eines Attributes mit einem neuen Wert

```
class Fahrzeug:
    def __init__(self, hersteller, baujahr, geschwindigkeit = 0):
        self.hersteller = hersteller
        self.baujahr = baujahr
        self.geschwindigkeit = geschwindigkeit

    def beschleunigen(self, x):
        '''Erhöhe die Geschwindigkeit um x'''
        self.geschwindigkeit += x # Berechnung
        return

    def getGeschwindigkeit(self): # getter methode: ein Attribut zurückliefert
        '''Zeigt die aktuelle Geschwindigkeit an.'''
        return self.geschwindigkeit

    def setGeschwindigkeit(self, x): # setter Methode: den Wert direkt ersetzen
        self.geschwindigkeit = x
        return
```

Eine Methode kann selbst eine weitere Methode aufrufen:

```
class Fahrzeug:
    def __init__(self, hersteller, baujahr, geschwindigkeit = 0):
        self.hersteller = hersteller
        self.baujahr = baujahr
        self.geschwindigkeit = geschwindigkeit

    def beschleunigen(self, x):
        '''Ändert die Geschwindigkeit auf x'''
        self.geschwindigkeit += x # Berechnung
        return self.getGeschwindigkeit() # Methode innerhalb Methode aufrufen

    def getGeschwindigkeit(self): # getter methode: ein Attribut zurückliefert
        '''Zeigt die aktuelle Geschwindigkeit an.'''
        return self.geschwindigkeit

    def setGeschwindigkeit(self, x):
```

```
        self.geschwindigkeit = x  # keine Berechnung, sondern den Wert direkt ersetzen
        return
```

Im letzteren Beispiel ruft die Methode 'beschleunigen' als Rückgabewert die Methode 'getGeschwindigkeit', um gleich dann nach einer Beschleunigung den aktuellen Wert vom Attribut 'geschwindigkeit' zu zeigen. Dadurch erspart man sich einen expliziten Aufruf von Getter-Methode im Programm.

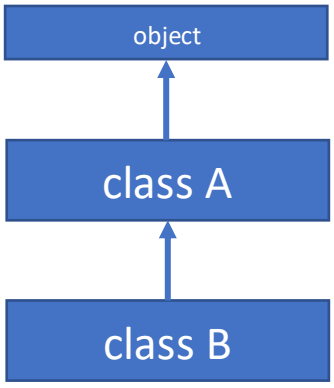
Kommunikation zwischen Objekten

Objekte einer Klasse (oder mehreren Klassen) können u.U. miteinander kommunizieren. Im folgenden Beispiel haben wir zwei Objekte aus derselben Klasse Roboter. Die Methode intro sorgt dafür, dass jedes Objekt auf das eigene Attribut 'name' zugreift. Die Methode hallo dagegen, greift auf das Attribut 'name' eines weiteren Objektes. Die Voraussetzung dabei ist natürlich, dass das zweite Objekt existiert und auch ein Attribut 'name' besitzt. Es ist dann egal, ob das zweite Objekt auch aus der Klasse des ersten Objektes abgeleitet worden ist, oder nicht.

```
class Roboter:
    def __init__(self, name):
        self.name = name

    def intro(self):
        '''Roboter stellt sich vor mit seinem Namen.'''
        print(f'Hallo, ich heiße "{self.name}"') # Zugriff auf das eigenes Attribut

    def hallo(self, other): # other: ein Objekt, welches ein Attribut 'name' besitzt.
        '''Roboter grüßt einen anderen Roboter!'''
        print(f'Hallo "{other.name}"') # Zugriff auf das Attribut des 'name' des Anderen

if __name__ == '__main__':
    xenon = Roboter('Xenon')
    marvin = Roboter('Marvin')
    xenon.intro() # stellt sich vor
    xenon.hallo(marvin) # grüßt den anderen
```

Vererbung

Eine Klasse kann ihre Struktur von einer oder mehreren Klassen übertragen bekommen.
In diesem Fall ist dann diese Klasse eine Unterklasse und die weiteren Klassen sind ihre Superklassen.



Drei wichtige Methoden:

isinstance(x, y) liefert True zurück, wenn x eine Instanz (ein Objekt) der Klasse y ist
issubclass(a, b) liefert True zurück, wenn a eine Unterklasse ist von b
mro(m) liefert eine Liste von Superklassen der Klasse m zurück

Besonderheiten bezüglich Unterklassen

Alle Objekte eine Unterklasse sind auch Instanzen der Superklasse(n).

```
class Person:
    def __init__(self, vname, name, alter):
        self.vname = vname
        self.name = name
        self.alter = alter

    def getName(self):
        return f'Der Vorname ist {self.vname}'

# Klasse Student als Unterklasse von Person
class Student(Person):
    def __init__(self, vname, name, alter, semester):
        Person.__init__(self, vname, name, alter)
        self.semester = semester # zusätzliches Attribut

    def studieren(self): # zusätzliche Methode
        return f'{self.vname.capitalize()} studiert im Semester {self.semester}'
```

Alle Attribute und Methoden in der Superklasse sind auch in der Unterklasse durch Vererbung dabei. In unserem Beispiel ist die Methode getName auch für Objekte der Klasse Student gültig. Im Gegenteil sind zusätzliche Methoden und Attribute in einer Unterklasse für die Objekte der Superklasse nicht vorhanden. D. h. in unserem Beispiel, das Attribut Semester oder die Methode studieren sind nur für Student-Objekte vorhanden.

Polymorphie (aus dem griechischen → Vielgestaltigkeit)
Wenn eine und dieselbe Methode zwei verschiedene Ergebnisse zurückliefert.

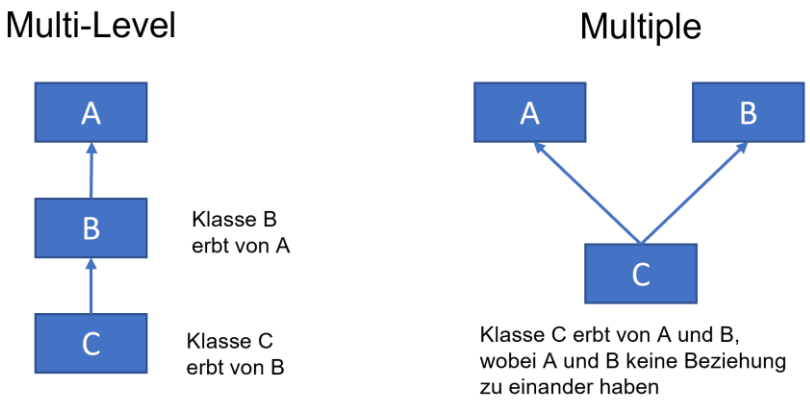
```
class Person:
    def __init__(self, vname, name, alter):
        self.vname = vname
        self.name = name
        self.alter = alter

    def getInfo(self):
        return f'Vorname: {self.vname.capitalize()}\nName: {self.name.capitalize()}\nAlter: {self.alter
}'


class Student(Person):
```

```
    def __init__(self, vname, name, alter, semester):
        Person.__init__(self, vname, name, alter)
        self.semester = semester # zusätzliches Attribut

    def studieren(self): # zusätzliche Methode
        return f'{self.vname.capitalize()} studiert im Semester {self.semester}'

    def getInfo(self):
        return f'Vorname: {self.vname.capitalize()}\nName: {self.name.capitalize()}\nAlter: {self.alter
}\nSemester: {self.semester}'
```

In diesem Beispiel liefert die Methode getInfo aus der Klasse Person nur den Vornamen, den Namen und das Alter einer Person zurück, wobei dieselbe Methode aus der Klasse Student noch das Semester dazu zurückliefert. Die Methode getInfo wurde aus der Superklasse in die Unterklasse übernommen und überschrieben (bzw. verändert), sodass wir ein zusätzliches Attribut, nämlich Semester damit abfragen können.

Multiple vs. Multi-Level Vererbung

Multi-Level-Vererbung ist, wenn Superklassen voneinander erben. Multiple ist, wenn es zwischen Superklassen keine Verwandtschaft gibt.



Beispiel Multiple-Vererbung

```
# oop_9.py
# Klasse Kunde als Unterklassen von Person und Konto
# ein Beispiel für Multiple-Vererbungsmodell
class Person:
    def __init__(self, vname, name):
        self.vname = vname
        self.name = name

    def show(self):
        return f'Vorname:{self.vname.capitalize()}\nName: {self.name.capitalize()}'

class Konto:
    def __init__(self, kontonr, guthaben = 0):
        self.kontonr = kontonr
```

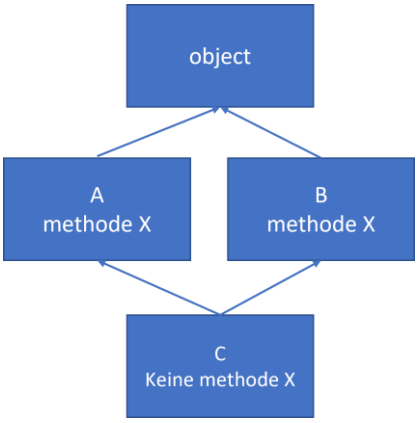
```
        self.guthaben = guthaben

    def kontostand(self):
        return f'Kontonummer: {self.kontonr}\nGuthaben: {self.guthaben}'

class Kunde(Person, Konto): # ein Kundenobjekt ist ein Personobjekt, das ein Kontoobjekt besitzt
    def __init__(self, vname, name, kontonr, guthaben):
        Person.__init__(self, vname, name) # Objekt mit Attributen und Methoden aus der Klasse 'Person'
        Konto.__init__(self, kontonr, guthaben) # zusätzliche Attribute und Methoden aus 'Konto'

k = Kunde('Jim', 'Bim', '12345678', 1000.00)
print(k.__dict__)
print(k.show()) # eine Methode aus der Superklasse Person
print(k.kontostand()) # # eine Methode aus der Superklasse Konto
```

Beispiel Multilevel-Vererbung

```
# oop_10.py
# Beispiel für Multilevel-Vererbungsmodell

class Opa:
    def opa_methode(self):
        print('Hallo von Opa.')

class Papa(Opa):
    def papa_methode(self):
        print('Hallo von Papa.')

class Kind(Papa):
    def kind_methode(self):
        print('Hallo ich bin Kind.')

# print(Kind.mro())

# Ein Kind-Objekt erzeugen
k = Kind()

# Die Methoden an k anwenden
k.kind_methode()
k.papa_methode()
k.opa_methode()
```

In diesem Beispiel erbt die Klasse Kind von Klasse Papa und die Klasse Papa erbt selbst von Klasse Opa. Bei solchen Vererbungsmodellen handelt es sich um ein Multi-Level-Vererbungsmodell.

Diamantenproblematik



Die Klasse C in diesem Diagramm ist eine Unterklasse von Superklassen A und B.
In dieser Klasse befindet sich keine Methode 'X'.
In der Superklasse A und B sind jeweils eine Methode 'X' definiert.
Wenn wir an einem Objekt der Klasse C diese Methode anwenden, dann wird zuerst die Methode in der Klasse A (die erste Klasse in der Reihenfolge) gesucht. Wenn die Methode da existiert, wird sie ausgeführt, ansonsten sucht der Interpreter in der zweiten Superklasse (B) nach ihr und führt sie von da aus.

```
# oop_11.py

class A:
    def __init__(self, name):
        self.name = name

    def show(self):
        return self.name

class B:
    def __init__(self, age):
        self.age = age

    def show(self):
        return self.age

class C(A, B):
    def __init__(self, name, age):
        A.__init__(self, name)
        B.__init__(self, age)

if __name__ == '__main__':

    obja = A('Objekt A')
    print(obja.show())
    objb = B(10)
    print(objb.show())
```

```
    objc = C('Objekt C', 20)
    print(objc.show())
```

Bei ähnlichen Fällen in Python-Klassen, wird oft die Methode in der Unterklasse überschrieben:

```
class C(A, B):
    def __init__(self, name, age):
        A.__init__(self, name)
        B.__init__(self, age)

    def show(self):
        return self.name, self.age
```

Im letzteren Beispiel überschrieben wir die Methode show, sodass sie ein Tupel aus beiden Attributen name und age zurückliefert.

Klassenattribute und Klassenmethoden

Sind Attribute und Methoden, die anhand Klassennamen (ohne dass man Objekte erzeugen muss) aufgerufen werden können. Natürlich können diese auch anhand Objekte der Klasse aufgerufen werden:

```
# oop_13.py

# Klassenattribute bzw. Klassenmethoden
class Car:

    make = 'Toyota Motors' # Ein Klassenattribut

    def __init__(self, model, year):
        self.model = model
        self.year = year

    def show(self):
        return f'Hersteller: {self.make}\nModell: {self.model}\nBaujahr:{self.year}'

    @classmethod # eine Methode als Klassenmethode definieren
    def getMake(cls):
        return cls.make


# Klassenattribut kann auch ohne Objekt aufgerufen werden
print(Car.make)
# Klassenmethode anhand Klasse aufrufen
print(Car.getMake())
# Klassenmethode bzw. -attribute anhand Objekt aufrufen
ca = Car('Aygo', 2018)
print(ca.getMake())
print(ca.make)
```

Private und Starkprivate Attribute

Manchmal wollen wir den einfachen Zugriff auf gewisse Attribute und Methoden (aus organisatorischen oder Sicherheitsgründen) verbieten. Dazu kann man diese privatisieren:

```
# oop_14.py
# Privat und Starkprivate Attribute und Methoden
class Person:
    def __init__(self, name, age, tel):
        self.name = name
        self._age = age # privatisiertes (geschütztes) Attribut
        self.__tel = tel # starkprivatisiertes Attribut

    def getTel(self):
        return self.__tel

p = Person('Jill', 27, 12345)
# Zugriff auf ein normales Attribut
print(p.name)
# Zugriff auf ein privatisiertes Attribut
print(p._age)
# Einfacher Zugriff auf starkprivatisiertes Attribut verweigert
# print(p.__tel) # Fehler
# Dazu muss man eine Gettermethode haben
print(p.getTel())
# Hack!
print(p._Person__tel) # _Person__tel ist der key aus dem p.__dict__ Dictionary!
# Wir sind die Guten ;) Solche Sachen nur für gute Zwecke …
# Der Sinn dahinter: den anderen Kollegen/innen zu sagen: Bitte nicht ändern! Do not touch
my code!
```