Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-09 Dozent: Karsten Flügge

Themen des Tages

- Klassen (eigene Typen)
  - Konstruktion
  - Methoden
  - spezielle Methoden
  - private Felder
  - statische Methoden
  - Vererbung

minimal

```
class MyError(Exception): pass
```

Klassen

Klassen in Python sind eine grundlegende **Struktur**, die es ermöglicht, eigene **Daten(typen)** zu schaffen, die **spezifische Attribute** und **Methoden** haben. Man kann sich eine Klasse wie einen **Bauplan** vorstellen, der definiert, wie ein bestimmtes **Objekt** aussehen und funktionieren soll.

```
class Person:
    def __init__(self, name, age):   # initialisierung
        self.name = name
        self.age = age
    def send_email(self):pass
```

Anders als ein *dict* kann eine solche Person hier nur zwei Eigenschaften ("**Felder**") haben: **name + age**.

Die Struktur ist also (relativ) fest.

Person(name="John",age=30)

person = {"Name":"John", age:30, adresse:{}}

Klassen

Warum **Klassen?**

Modellierung eigener Konzepte

Abbildung der Welt

Erweiterung von vorhandenen Konzepten (*int* -> Fraction Bruch …)

Daten mit Datenbanken synchronisieren (z.B. Benutzer) SQL

Struktur! … Schema.

Abkapseln von Funktionalitäten (Sichtbarkeit)

Klassen Konstruktion

Klassen (Typ) Konstruktion (**Erzeugung von Objekten einer Klasse**)

Im sogenannten **constructor** werden **Attribute** initialisiert

```
class Person:
  # constructor Funktion zum Erzeugen eines Personen Objektes
    def __init__(self, name, age): # Initialisierung
        self.name = name # Attribut name wird über Parameter name gesetzt
        self.age = age   # 'speichern' des Parameters
       print("Hallo",name)


person = Person(name='John', age = 30) # Objekt wird in Variable gespeichert
print(person.name) # Abfrage von Feldern über "." erst mal ohne Klammern ()
print(person.age)
```

Die **__init__(self, …) Methode** ist eine Spezialfunktion (pro Klasse), welche bestimmt, wie ein Objekt nach der Erzeugung auszusehen hat. Der erste **Spezialparameter self** ist das Objekt welches wir modifizieren können. Ansonsten verhält es sich wie eine normale Funktion, mit all den Regeln die wir kennen.

Der Konstruktor erzeugt aus einer Klasse (Person) ein Objekt (person).
Zur Unterscheidung wird die **Klasse groß** geschrieben und eine **Instanz** / die zugehörige **Variable klein.**

Klassen Konstruktion

Klassen Konstruktion (**Erzeugung von Objekten einer Klasse**)

Instanz ≈ Ausstanzen der Schablone

Instanziieren ≈ Initiieren Initialisieren / Anlegen

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

person = Person('John', 30)
```

Der Konstruktor `__init__`:

- `__init__` ist eine spezielle Methode, die aufgerufen wird, wenn ein neues Objekt der Klasse erstellt wird.
- Sie wird oft verwendet, um die Anfangsattribute des Objekts zu **initialisieren ≈ anlegen**.

Selbstreferenz `self`:

- **`self`** ist ein Referenzparameter, der auf das aktuelle Objekt der Klasse zeigt.
- Es wird verwendet, um auf **Attribute** und **Methoden** des Objekts zuzugreifen.

Klassen Konstruktion

AUFGABE:

a) selbst Personen Klasse definieren und zwei Personen erzeugen und anzeigen:

```
class Person:
    def __init__(self, name, age):
        self.name = name # Attribut/Feld 'name' wird über Parameter 'name' gesetzt
        self.age = age

john = Person('John', age = 30)
person = Person('Alf', age = 12)
print(john.name) # Abfrage von Feldern/Attributen über "." erst mal ohne Klammern ()
print(john.age)
```

Doppelpunkte und **Einrückung beachten.** __init__ mit zwei*zwei Unterstrichen

Klassen **G**roß, individuen klein

Klassen Konstruktion

Unsere dir-Methode gibt wie immer alle Methoden und Attribute *der Klasse*

```
>>> dir(person)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__firstlineno__',
'__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__static_attributes__', '__str__',
'__subclasshook__', '__weakref__', 'age', 'name']
```

Auflisten aller Attribute eines Objektes:

```
vars(person)
{'name': 'John', 'age': 31}
```

```
john.__static_attributes__
('age', 'name')
```

Methoden vs Funktionen

```python
class Person:

  def __init__(self, name, age):
    self.name = name # Attribut name wird über Parameter name gesetzt
    self.age = age

  def happy_birthday(self):
    print("happy birthday", self.name)
    self.age += 1


john = Person("John", 20)

OHNE Einrückung wird Methode zur Funktion!

print(john.happy_birthday()) # Methodenaufruf SPO
print(happy_birthday(john))  # Funktionaufruf PO / PS
```

Methoden vs Funktionen

Aufgabe: verjünge Person um 10 Jahre!

```
class Person:

  def __init__(self, name, age):
    self.name = name # Attribut name wird über Parameter name gesetzt
    self.age = age

  def jungbrunnen(self):
    print("Jungbrunnen")
    self.age = self.age - 10

Aufruf über
john = Person('John', age = 30)
john.jungbrunnen()  mit Klammern (ohne self! nie mitgeben, immer interne Maschinerie)
print(john.age)    ohne Klammern
```

Argument ≠ Parameter ≠ Attribut

Argument ≠ Parameter ≠ Attribut

```
class Person:

  def __init__(self, name, alter):
    self.name = name # Attribut name wird über Parameter name gesetzt
    self.age = alter # Parameter speichern als Attribut
```

Argument ≠ Parameter ≠ Attribut können alle anders benannt sein

```
age_de_john = 20 # Variable als Argument:
john = Person('John', alter = age_de_john)
print(john.age)
```

Klassen Methoden

Funktionen die zu einer Klasse / einem Klassenobjekt gehören nennt man **Methoden.**
Wie beim bekannten constructor haben sie stets **self** als erstes **Spezialargument**.

```
class Person:

    # alle Daten werden in dieser init Methode gesetzt
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def zeige_Name(self):
        print(f"Mein Name ist {self.name}")
```

ALLE Methoden müssen eingerückt sein!

```
person = Person('John', 30)

person.zeige_Name() # das self wird nicht explizit mit übergeben!
```

Methoden von Objekten kennen wir schon von str, list, … ! z.B. "abc".upper()
selbst einem Konstruktor sind wir schon begegnet: str(1) int("1")

⚠️ die **Methoden** sind innerhalb der Klassen Deklaration **eingerückt** (also tab/spaces vor def…)!

Spezielle Methoden

Versucht man bisher so ein Personen Objekt mit print anzuzeigen bekommt man etwas recht technisches:

```
person = Person('John', 30)
print(type(person)) # <class '__main__.Person'>
print(person) # <__main__.Person object at 0x12b2f0a10> …
direkte Ausgabe noch nicht sehr schön / informativ ^^
```

Um diese **Ausgabe** zu **beeinflussen** gibt es **spezielle Methode**n (welche man stets am doppelten __ Unterstrich erkennen kann)

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

    def __str__(self): # wie soll dieses Objekt in ein String umgewandelt werden?
        return f'{self.name}, {self.age} years old'

person = Person('John', 30)
#str(person)
print(person) # John, 30 years old
```

Spezielle Methoden

Versucht man bisher so ein Personen Objekt mit print anzuzeigen bekommt man etwas recht technisches:

```
person = Person('John', 30)
print(type(person)) # <class '__main__.Person'>
print(person) # <__main__.Person object at 0x12b2f0a10> …
direkte Ausgabe noch nicht sehr schön / informativ ^^
```

Um diese **Ausgabe** zu **beeinflussen** gibt es **spezielle Methode**n (welche man stets am doppelten __ Unterstrich erkennen kann)

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

    def __repr__(self): # wie soll dieses Objekt repräsentiert werden (als String?)
        return f'{self.name}, {self.age} years old'

person = Person('John', 30)

print(person) # John, 30 years old
```

Lesen von Feldern

FELD = Attribut

Lesen von öffentlichen Feldern:

```
class Konto:
  def __init__(self, inhaber, saldo):
    self.inhaber = inhaber
    self.saldo = saldo # in Euro

mein_konto = Konto("Max Mustermann", 1000)
print(mein_konto.saldo) # 1000
```

Setzen von Feldern

Setzen von öffentlichen Feldern:

- Öffentliche Felder sind direkt zugänglich und können innerhalb und außerhalb der Klasse ohne Einschränkungen geändert werden.
- Sie werden normalerweise für Daten verwendet, die sicher von anderen Teilen des Programms gelesen und geschrieben werden können.

```
class Konto:
  def __init__(self, inhaber, saldo):
    self.inhaber = inhaber
    self.saldo = saldo

mein_konto = Konto("Max Mustermann", 1000)
mein_konto.saldo = 5000  # Direkter Zugriff auf öffentliches Feld

man kann Attribute also nachträglich ändern!
```

Setzen von Feldern

Setzen von beliebigen Attributen:

```
class Konto:
  def __init__(self, inhaber, saldo):
    self.inhaber = inhaber
    self.saldo = saldo

mein_konto = Konto("Max Mustermann", 1000)
mein_konto.saldo = 10000 # alten Kontostand überschreiben
mein_konto.memo = "Guter Kunde!!" # neues Attribut für mein eines Konto

man kann beliebig Attribute also nachträglich anlegen und ändern!
Fast so flexibel wie dict
Eher vermeiden und im Vorfeld überlegen welche Struktur meine Klasse haben soll
```

Klassen vs Dictionaries

Setzen/Anlegen von *beliebigen Attributen*:

```
class Konto:
  def __init__(self, inhaber, saldo):
    self.inhaber = inhaber
    self.saldo = saldo

mein_konto = Konto("Max Mustermann", 1000)
mein_konto.memo = "Guter Kunde!!"

mein_konto2 = {"inhaber":"Max Mustermann", "saldo": 1000, "memo":"JA!" }
mein_konto2["inhaber"]
# Unterschiede: Einträge Keys in Dict sind idR strings!

Klassen sind bessere dicts
```

Klassen Konstruktion

AUFGABE:

a) selbst Personen Klasse definieren und erste Person erzeugen und anzeigen:

b) erzeuge Methode happy_birthday(self) welche das Alter der Person um eins erhöht.

```
class Person:
  def __init__(self, name, age):
    self.name = name # Attribut name wird über Parameter name gesetzt
    self.age = age

  def happy_birthday(self):
    self.age += 1

john = Person("John", 20)
print(john.age)
john.happy_birthday() # Methoden MIT Klammern () aufrufen!
print(john.age)
```

Doppelpunkte und **Einrückung beachten.** __init__ mit zwei*zwei Unterstrichen

Klassen ≈ Typ

class ist **keyword** in python mit dem sich eine Klasse / Typ **definieren** läßt
type() ist eine eingebaute **Funktion**, mit welcher sich die Klasse von einem Objekt **abfragen** läßt

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

john = Person("John", 20)
print(type(john))
```

```
<class '__main__.Person'>  im Hauptmodul/namespace __main__
```

```
>>> type(3)
<class 'int'>
```

Schützen von Feldern

- **Private Felder** werden in Python durch einen Unterstrich _ oder doppelte Unterstriche __ vor dem Namen gekennzeichnet.
- **Ein Unterstrich _** wird als **Hinweis** verwendet, dass ein Attribut nicht von außerhalb der Klasse geändert werden sollte, obwohl es technisch möglich ist.
- **Doppelte Unterstriche __** bewirken eine 'Namensmangelung', die es **schwieriger** macht, von außerhalb der Klasse auf das Attribut zuzugreifen.

```
class Konto:
  def __init__(self, inhaber, saldo):
    self.__inhaber = inhaber
    self.__saldo = saldo

  def get_saldo(self):  # NUR LESEN
    return self.__saldo


mein_konto = Konto("Max Mustermann", 1000)
mein_konto.meine_ZusatzInfo = "super"
mein_konto.__saldo = 500  # von aussen kein Zugriff auf privates Feld

print(mein_konto.get_saldo()) # Puh, unverändert 1000!
```

statische Methoden

Statische Methoden gelten nicht für einzelne Objekte, sondern für die Klasse (oder die Gesamtheit aller Objekte).

```
class Person:

  def __init__(self, name, age):
    self.name = name
    self.age = age

  @staticmethod
  def json_to_person(json): # ≈ new
    return Person(json['name'], json['age'])

person = Person.json_to_person({'name': 'John', 'age': 30})

# hier als extra constructor
```

Statische Methoden werden durch die **@staticmethod annotation** deklariert.

statische Felder

Statische **Felder** gelten nicht für einzelne Objekte, sondern für die Klasse (oder die Gesamtheit aller Objekte).

```
class Person:

  count = 0 # gilt übergreifend für Klasse Person

  def __init__(self, name, age):
    self.name = name
    self.age = age
    Person.count += 1

person1 = Person('John', 30)
person2 = Person('Jane', 25)
person3 = Person('Jack', 40)

print(Person.count) # 3
```

Diese werden logischerweise nicht über ein Objekt (self oder person) sondern über die Klasse abgerufen
Person.count

statische Felder

Statische **Felder** gelten nicht für einzelne Objekte, sondern für die Klasse (oder die Gesamtheit aller Objekte).

```
class Person:

   count = 0  # global für ALLE Personen

   def __init__(self, name, age):
      self.name = name
      self.age = age
      Person.count += 1

   @staticmethod
   def counted():
      print(f'Es gibt {Person.count} Personen')

   @staticmethod
   def MenscheAufDerErde():
      return 800000000000

person1 = Person('John', 30)
person2 = Person('Jane', 25)
person3 = Person('Jack', 40)

print(Person.counted()) # 3
Person.MenscheAufDerErde() # pseudo Konstante
```

Diese werden logischerweise nicht über ein Objekt (self oder person) sondern über die Klasse abgerufen **Person.**count

statische Felder

Statische **Felder** gelten nicht für einzelne Objekte, sondern für die Klasse (oder die Gesamtheit aller Objekte).

```
class Person:

   count = 0 # gilt übergreifend für Klasse Person
   alle = [] # sammle per Hand alle Personen

   def __init__(self, name, age):
      self.name = name
      self.age = age
      Person.count += 1
       Person.alle.append(self)

person1 = Person('John', 30)
person2 = Person('Jane', 25)
person3 = Person('Jack', 40)

print(Person.count) # 3
print(Person.alle)
[...]
```

statische Felder

```python
class Person:
  count = 0  # gilt übergreifend für Klasse Person
  alle = []

  def __init__(self, name, age):
    self.name = name
    self.age = age
    Person.count += 1
    Person.alle.append(self)

  def __repr__(self): # 'representation'  fallback für __str__
    return f"Person(\"{self.name}\",{self.age})"
```

Spezialmethode **__repr__** zur 'korrekten' Darstellung der Objekte in Listen etc

```
[<__main__.Person object at 0x104d60550>, <__main__.Person object at 0x104d61310>, <__main__.Person object at 0x104d60cd0>]
???
```

```
besser:
[Person('John',30), Person('Jane',25), Person('Jack',40)]
```

Pseudo Konstanten

Statische **Felder** gelten nicht für einzelne Objekte, sondern für die Klasse (oder die Gesamtheit aller Objekte).

```
class Person:

    @staticmethod
    def MenscheAufDerErde():
        return 800000000000


Person.MenscheAufDerErde() # pseudo Konstante
```

Aufgaben

1. a (über)schreibe **__str__** Methode die Person schön zurückgibt (aber ohne Nebeneffekte!)

```
print(person)
```

1. b (über)schreibe **__repr__** Methode die Person formatiert zurückgibt:
   Person(name="…",alter=:…) Tipp: f"" oder als json **repr**esentation

```
print(person)   # __str__ oder __repr__ falls kein __str__
print([person]) # __repr__
```

Entferne die **__str__** Methode

```
print(person)
```

Die Spezialmethode __repr__ dient zur **repr**esentativen Darstellung des Objektes,
also sie sollte alle Informationen enthalten um das Objekt wieder herstellen zu können

2. füge statisches Feld **alle / personen_liste** in Person hinzu
   Bei jeder Konstruktion füge die neue Instanz (self) zur Liste hinzu

3. füge Feld "telefon" (Nummer) zum Konstruktor mit default=None hinzu
   nutze die Nummer in einer anrufen() Methode

Eigene Typen als Attribute

```
class Address:
  def __init__(self, street, zip, city):
    self.street = street
    self.zip = zip
    self.city = city

class Person:
  def __init__(self, name, age, address:Address, freund:Person=None):
    self.name = name
    self.age = age
    self.address = address
    self.freund = freund

addresse = Address("Hauptstr. 12", "12345", "Musterstadt")
person1 = Person("Hans", 35, addresse)
person2 = Person("Martina", 32, addresse, freund=person1)
```

Verschachtelte Klassen Hierarchie?

```
class Address:
  def __init__(self, street, zip, city):
    self.street = street
    self.zip = zip
    self.city = city

class Person:
  def __init__(self, name, age, address:Address, freund:Person=None):
    self.name = name
    self.age = age
    self.address = address
    self.freund = freund

addresse = Address("Hauptstr. 12", "12345", "Musterstadt")
person1 = Person("Hans", 35, addresse)
person2 = Person("Martina", 32, addresse, freund=person1)
```

Ausflug: Serialisierung

Serialisierung ≈ Abspeichern

- **per Hand** open(file).write() als csv Tabelle / über __repr__
- **import csv**
- **pickle** python: Keine Gedanken machen!!
- **Datenbank**: SQL (Adaptoren: @dataclass, ...) ORM object relational mapping

```
class Address:
  def __init__(self, street, zip, city):
    self.street = street
    self.zip = zip
    self.city = city

class Person:
  def __init__(self, name, age, address:Address, freund:Person=None):
    self.name = name
    self.age = age
    self.address = address
    self.freund = freund

addresse = Address("Hauptstr. 12", "12345", "Musterstadt")
person1 = Person("Hans", 35, addresse)
person2 = Person("Martina", 32, addresse, freund=person1)
```

Klassen Hierarchie

Wie in der richtigen Welt lässt sich Information besser strukturieren, indem man Unterklassen und Oberklassen betrachtet:
z.B.
Fahrrad, Auto, Zug < Fahrzeug Spezialtyp < Obertyp
Dreieck, Kreis, Viereck < Form Unterklasse < Oberklasse
Admin, User, Editor, Tester < Benutzer, Mitarbeiter < Person
Hund, Katze, Maus < Säugetier, Insekt < Tier < Lebewesen

```
class Vehicle:
  def honk(self):
    pass

class Car(Vehicle): # d.h. Car is a Vehicle
  def honk(self): # überschreiben der oberen Methode
    print('honk honk, hup hup')

class Bike(Vehicle):
  def honk(self):
    print('ding ding, ring ring')

Um eine Unterklassen von einer Oberklassen "erben" zu lassen gibt man sie in runden
klammern mit, z.B. Car(Vehicle)
```

Klassen Hierarchie Beispiele

Obst, Gemüse < Essbares, Getränke < Konsumierbar
Apfel, Banane < Obst (gemeinsam: Zuckergehalt, Farbe(n))
Fanta, Cola, Alkohol < Getränke
Bier, Sekt, Wein < Alkohol (gemeinsam: Promille)

Einkäufer, Lagerist, Buchhalter < Mitarbeiter (gemeinsam: Gehalt, Nummer)

Projekte, Gruppen, Stationen **Verschachtelung ≠ Vererbung**

Kind, Erwachsener, Mann, Frau < Mensch
als Unterklasse oder als Attribut?? oft stellt sich diese Frage (keine universelle Antwort):

```
class Mensch:
  geschlecht:str # "m" / "w"
  altersstufe:str # baby, jugendlich, erwachsen
```

ODER

```
class Männlich(Mensch): pass
class Junge(Kind, Männlich): pass
```

Klassen Hierarchie

Wie in der richtigen Welt lässt sich Information besser strukturieren, indem man Unterklassen und Oberklassen betrachtet: z.B.

Ganze < Prim < Natuerlich < Rational < Reell (real) < Zahl < Complex < Hyperreal

Graphit, Diamant, **Graphene**, Buckminster Fullerene < **Kohlenstoff**

Monde, Planeten(orbit), Sonnen(leuchtkraft), Kometen, Pulsare, Loecher < Himmelskoerper (Masse, Positon)

Hierarchie versus Verschachtelung

Nadel, Laub < Baum "Teilmenge"
Stamm, Wurzel, Blätter **Attribute, Aspekte** von Baum "Element, Teil" Komposition

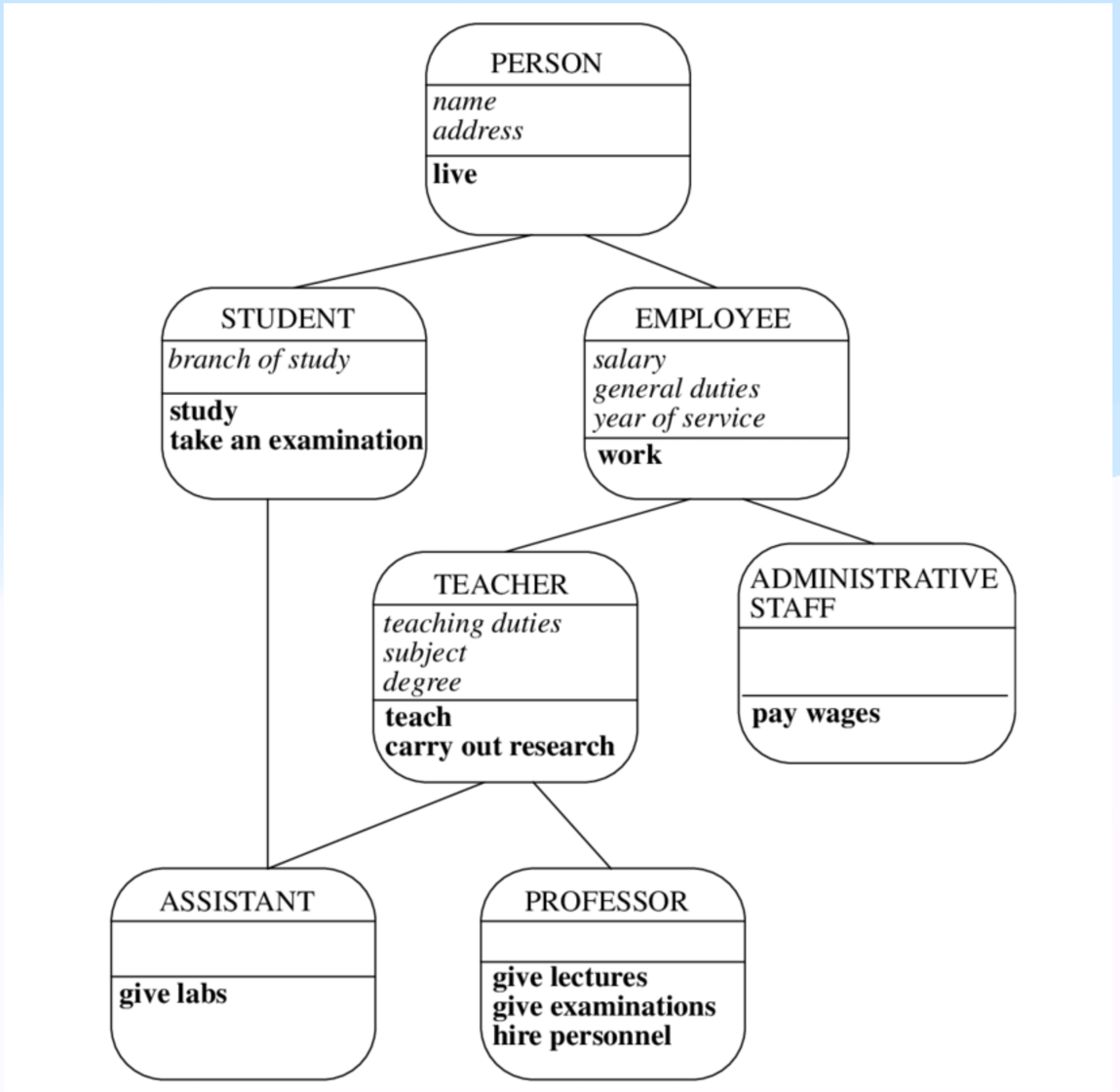
```
class Baum:
  def __init__(teile):
     self.root = teile["root"]

class Laubbaum(Baum):
   def blattart(): return "weich"
```

```
class Fussballspiel:
  def __init__():
     self.Mannschaften
     self.Torstand
     self.Schiri
     self.Karten # (rot ...)
```

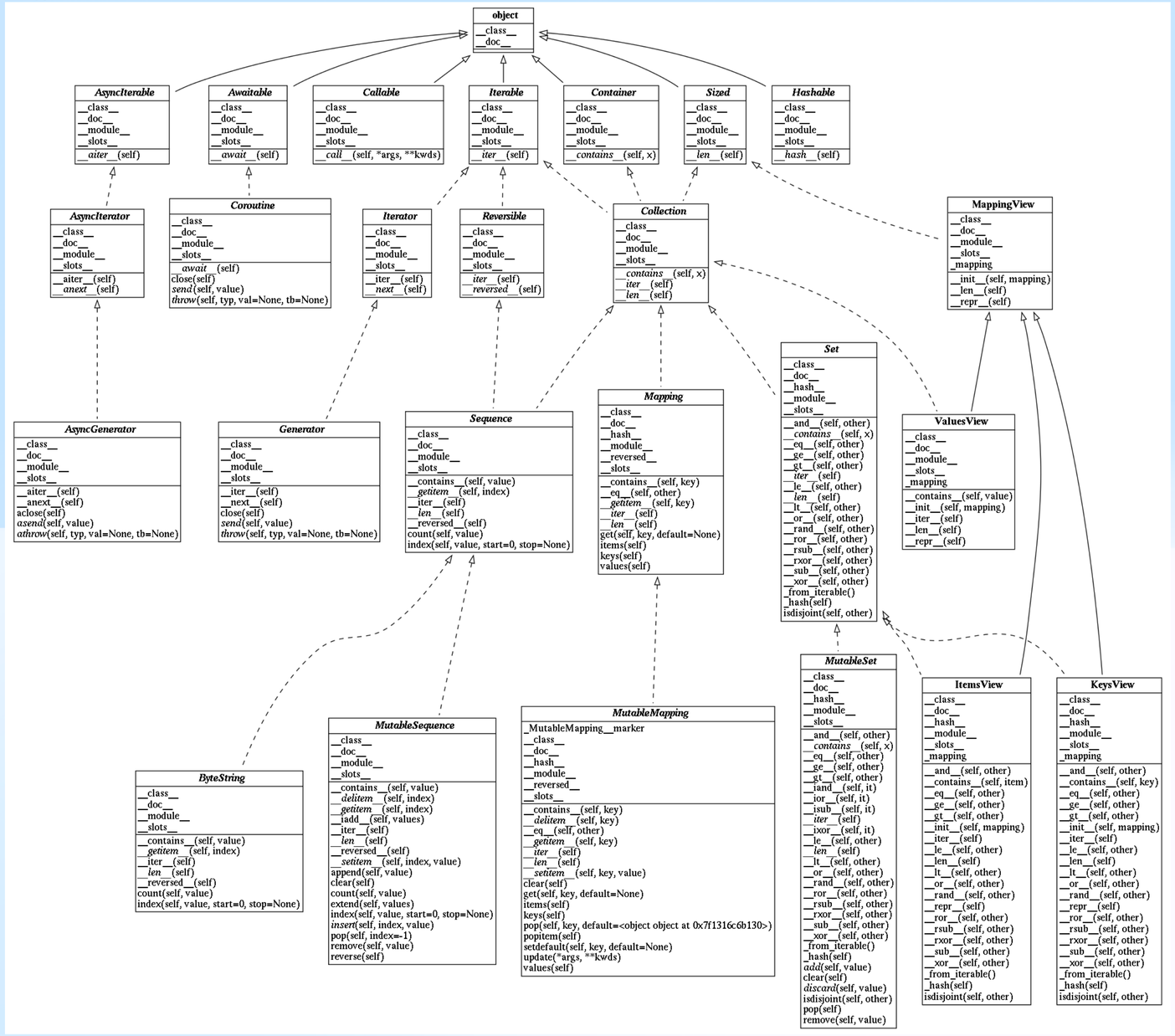
Hierarchie versus Verschachtelung

Real world example:

Attribute / Felder und
Methoden (fat)



Python Hierarchie



Vererbung von Methoden

```
class Vehicle:
  def honk(self):
     pass  # unspezifiziert 'abstrakt' virtual?

class Car(Vehicle):
  def honk(self): # overwrite
    print('hup hup')

class Bike(Vehicle):
  def honk(self):
    print('ring ring')

def letVehicleHonk(fahrzeug:Vehicle):
  fahrzeug.honk() # immer maximal spezifische Methode!

car = Car()
bike = Bike()

letVehicleHonk(car) # hup
letVehicleHonk(bike) # ring
```

Vererbung von Methoden

```
class Vehicle:

  def drive(self): # steht allen Unterklassen zur Verfügung
    print('brum brum')

class Car(Vehicle):
  def honk(self):
    super().drive() # von Oberklasse Vehicle
    print('honk honk, hup hup')

class Bike(Vehicle):
  def honk(self):
    print('ding ding, ring ring')

def letVehicleHonk(fahrzeug:Vehicle):
  fahrzeug.drive() # ein Fahrrad / ein Auto macht 'brum brum' über Vehicle Vererbung
  fahrzeug.honk() # immer maximal spezielle Methode!

car = Car()
bike = Bike()

letVehicleHonk(car) # hup
letVehicleHonk(bike) # ring
```

Zugriff auf Oberklasse in Methoden

Mit der **Spezialmethode** **super()** kann man auf die Oberklasse zugreifen (und auf deren Methoden)

```
class Vehicle:
   def drive(self): # steht allen Unterklassen zur Verfügung
      print('driving!')

class Car(Vehicle):
   def drive(self): # versteckt das Fahrzeug drive
      print('brum brum')

   def honk(self):
      # Vehicle.drive() # würde sich auf ALLE Fahrzeuge beziehen (ganze Flotte) müsste man implementieren
      super().drive() # zugriff auf nächste Oberklasse  'driving!'
      self.drive() # UNSER Fahren 'brum'
      print('honk honk, hup hup')

class Bike(Vehicle):
   def honk(self):
      print('ding ding, ring ring')

def letVehicleHonk(fahrzeug:Vehicle):
   fahrzeug.honk() # ein Auto macht extra 'brum brum'

car = Car()
bike = Bike()

letVehicleHonk(car)
letVehicleHonk(bike)
```

Tiefe Vererbung

Wie in der richtigen Welt lässt sich Information besser strukturieren,
indem man Unterklassen von Unterklassen betrachtet:
z.B.
Tandem < Fahrrad < Fahrzeug
Quadrat < Vieleck < Form
Schäferhund < Hund < Tier

```
class Vehicle:
  def honk(self):
    pass

class Car(Vehicle):
  def honk(self):
    print('honk honk, hup hup')

class Mazda(Car):
  def honk(self):
    print('tut tut, quietsch')
```

Tiefe Vererbung

ALLE Klassen erben implizit von Typ **object**

```
class Vehicle:
  def honk(self):
    pass

class Car(Vehicle):
  def honk(self):
    print('hup hup')

class Mazda(Car):
  def honk(self):
     super().honk() # direkte Oberklasse, also Car
    print('tut tut')

mein_mazda = Mazda()
mein_mazda.honk() # hup und tut
```

Car.honk(mein_mazda) greife explizit auf honk von Car Klasse zu

Tiefe Vererbung

```
class Vehicle(object):
  def honk(self):
    print("grr")

class Car(Vehicle):
  def honk(self):
    print('hup hup')

class Mazda(Car):
  def honk(self):
    super(Car, self).honk() # holt sich Oberklasse VON Car, also Vehicle
    print('tut tut')

mein_mazda = Mazda()
mein_mazda.honk()
```

so nicht:

```
super().super().
```

Tiefe Vererbung

Alle Klassen erben automatisch von object, damit erben sie alle folgenden Methoden:

```
>>> john=Person("john",23)
>>> dir(john)  #  von object geerbt:
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__firstlineno__',
'__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__static_attributes__', '__str__',
'__subclasshook__', '__weakref__', 'age', 'name']
```

viele sind vorgesehen (als 'stub') aber noch nicht implementiert, sondern sie werfen Fehler:

```
    alf > john
TypeError: '>' not supported between instances of 'Person' and 'Person'
```

```
>>> alf=Person("alf",31)
>>> alf2=Person("alf",31)
>>> alf == alf2   # ist definiert, aber nicht so wie wir denken?
False   # Müssten wir selbst überschreiben wenn sie gleich sein sollen
```

```
super().super().
```

Docstrings

```
class Person:
  """
  Eine Person hat Name und Alter
  """
  def __init__(self, name, age):
    self.name = name
    self.age = age

damit gibt

help(Person) dann den Text aus
```

Mehrfach Vererbung

In python kann eine Klasse von mehreren Klassen "erben" und damit dessen Funktionalitäten übernehmen:

```
class Drucker:
  def drucken(self):
    return "Druckt ein Dokument"

class Scanner:
  def scannen(self):
    return "Scannt ein Dokument"

class Multifunktionsgerät(Drucker, Scanner):
  pass
```

```
mfg = Multifunktionsgerät()
print(mfg.drucken())  # Ausgabe: Druckt ein Dokument
print(mfg.scannen())  # Ausgabe: Scannt ein Dokument
```

Vorteile der Mehrfachvererbung:

- Erhöhte Flexibilität in der Klassenstrukturierung und im Funktionsumfang.
- Fördert die Wiederverwendung von Code durch die Zusammenführung von Funktionen aus mehreren Klassen.

Nachteile und Herausforderungen:

- **Komplexität**: Die Vererbungshierarchie kann schwer zu verstehen und zu verwalten sein.
- **Diamond-Problem**: Ein klassisches Problem bei Mehrfachvererbung, bei dem eine Klasse auf zwei Wegen von derselben Basisklasse erbt, was zu Mehrdeutigkeiten führen kann.

Mehrfach Vererbung

In python kann eine Klasse von mehreren Klassen "erben" und damit dessen Funktionalitäten übernehmen:

```
class Drucker:
   def drucken(self):
      return "Druckt ein Dokument"

class Scanner:
   def scannen(self):
      return "Scannt ein Dokument"

class Multifunktionsgerät(Drucker, Scanner):
   pass

mfg = Multifunktionsgerät()
print(mfg.drucken())  # Ausgabe: Druckt ein Dokument
print(mfg.scannen())  # Ausgabe: Scannt ein Dokument
```

in Methode: `super(Drucker, self).tut()` # Zugriff auf spezielle Obermethoden

Vorteile der Mehrfachvererbung:

- Erhöhte Flexibilität in der Klassenstrukturierung und im Funktionsumfang.
- Fördert die Wiederverwendung von Code durch die Zusammenführung von Funktionen aus mehreren Klassen.

Nachteile und Herausforderungen:

- **Komplexität**: Die Vererbungshierarchie kann schwer zu verstehen und zu verwalten sein.
- **Diamond-Problem**: Ein klassisches Problem bei Mehrfachvererbung, bei dem eine Klasse auf zwei Wegen von derselben Basisklasse erbt, was zu Mehrdeutigkeiten führen kann.

Mehrfachvererbung

Wenn man Mehrfachvererbung hat und in beiden Elternklassen gibt es eine gleichnamige Funktion, ist es dann random welche genommen wird?

In Python folgt die Auflösung der **Method Resolution Order (MRO)**, die ebenfalls von der Reihenfolge der Klassen in der Vererbungsdefinition beeinflusst wird. Python verwendet das C3 Linearisierungsverfahren, um die MRO zu bestimmen, wobei die Reihenfolge so festgelegt ist, dass Kinderklassen vor ihren Elternklassen kommen und mehrere Basisklassen in der Reihenfolge gelistet sind, wie sie in der Klassendefinition erscheinen.

importieren von Klassen

```
# from Datei / Modul import Klasse  wie bei import von Funktionen

# in person.py oder Ordner modul
class Person: …

# in application.py :

# import person
from person import Person
person = Person('John', 30)

ODER

import module
person = module.Person('John', 30)
```

ORDNUNG

Module und Klassen dienen (auch) zum **Aufräumen** und **Ordnen** des Projektes
Warum? **separation of concerns, Entkopplung, file length, number of files in folder**

```
# from Datei / Modul import Klasse  wie bei import von Funktionen

# from mein_modul import *  # ALLES importieren, schlechte Angewohnheit

from mein_modul import xxx, meine_modul_funktion # nur was wir brauchen!
from mein_modul import MyClass # Variablen, Funktionen oder Klassen
```

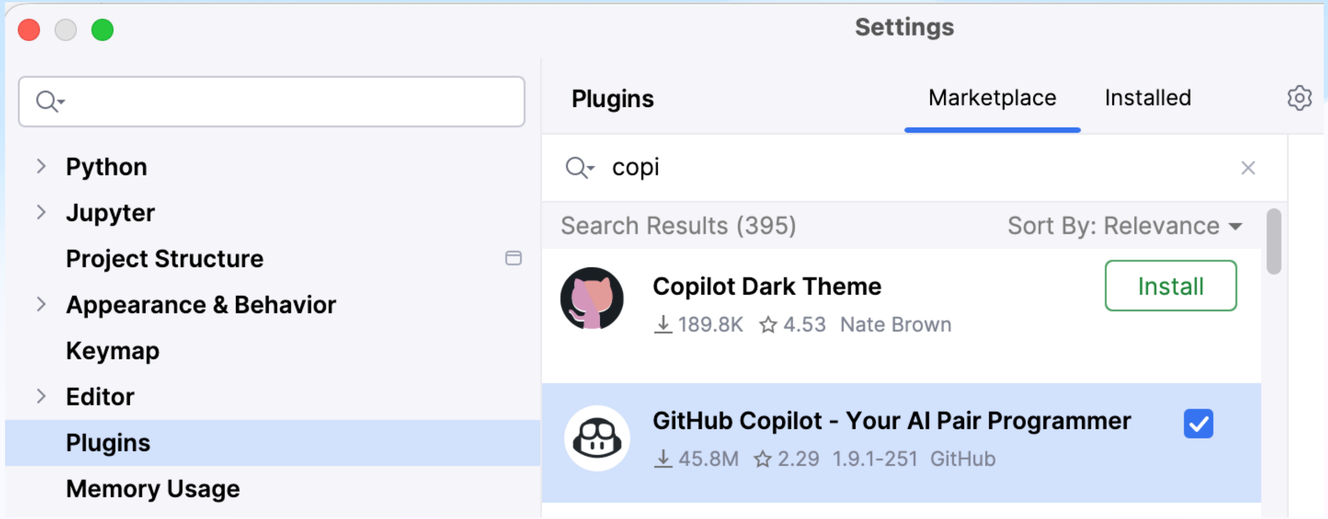
Offene Themen

try
lambda (signatur)
immutable types
list/dict comprehension
@dekaratoren
@dataclass
@classmethod

... Vertiefungen

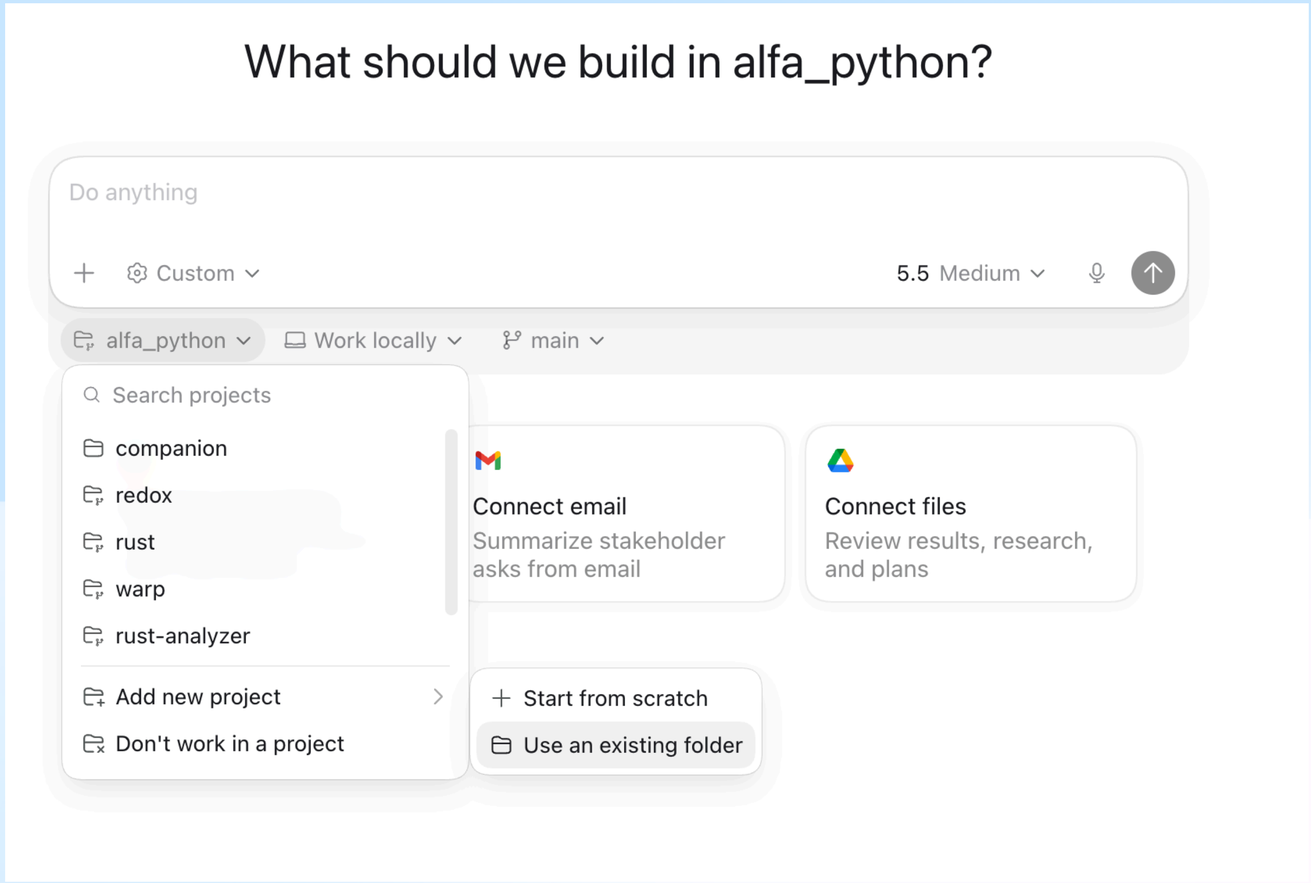
... spezielle module / Anwendungen

Settings -> Plugins -> GitHub Copilot



Offene Themen

`Codex Projekt Ordner auswählen / anlegen`



Aufgaben Klassen 2

Klassen Übungen
Leicht:
A
Definieren Sie eine Klasse Auto mit zwei Attributen: marke und modell.
Erstellen Sie ein Objekt der Klasse Auto und initialisieren Sie es mit Marke "Toyota" und Modell "Corolla".
Fügen Sie der Klasse Auto eine Methode zeige_info hinzu, die Marke und Modell des Autos ausdruckt.

B
Definieren Sie eine Klasse Buch mit den Attributen titel und autor.
Erstellen Sie ein Objekt der Klasse Buch und geben Sie dessen Attribute aus.

Mittel:
C
Erweitern Sie die Klasse Auto aus A um eine Methode alter, die das Alter des Autos basierend auf dem Produktionsjahr berechnet.

D
Definieren Sie eine Klasse Student, die Attribute wie name und matrikelnummer enthält und initialisiert.
Fügen Sie der Klasse Student eine Methode hinzu, die die Dauer bis zur Abschlussprüfung anzeigt, basierend auf dem aktuellen Studienjahr.
Erstellen Sie eine Klasse Person und definieren Sie einen Konstruktor, der Name und Alter annimmt.
Erstellen Sie eine Liste von Person-Objekten und implementieren Sie eine Methode, die die Namen aller Personen über 18 Jahre ausgibt.

Schwer:
E
Implementieren Sie in der Klasse Auto aus A eine Methode, die den Wert des Autos berechnet, der jedes Jahr um 5% abnimmt.

F
Fügen Sie der Klasse Student aus D Methoden hinzu, die den Status des Studenten (z.B. Vollzeit, Teilzeit) prüfen und entsprechend unterschiedliche Studiengebühren berechnen.
G
Erweitern Sie die Klasse Buch aus B um eine Methode, die die durchschnittliche Bewertung basierend auf einer Liste von Bewertungszahlen berechnet.
Implementieren Sie in der Klasse Person eine Methode, die das ideale Körpergewicht basierend auf Größe und Geschlecht berechnet (BMI).
Erstellen Sie eine Klasse Kurs, die eine Liste von Student-Objekten enthält und eine Methode zur Berechnung der Durchschnittsnote bietet.
Sehr schwer:
Implementieren Sie eine Vererbungshierarchie für Fahrzeuge, wobei Auto von einer generischen Klasse Fahrzeug erbt und spezielle Attribute hinzufügt.
Entwickeln Sie eine Klasse Universität, die verschiedene Fakultäten und deren Kurse verwaltet, inklusive Methoden zur Hinzufügung neuer Kurse und Studenten.
Schreiben Sie eine Methode für die Klasse Person, die sämtliche Verwandtschaftsverhältnisse in einem Stammbaum darstellen kann.
Entwerfen Sie eine Klasse Bibliothek, die verschiedene Ausleihmethoden für Bücher je nach Benutzerstatus (z.B. Student, Professor) implementiert.
Entwickeln Sie eine komplexe Klasse Wetterstation, die Wetterdaten sammelt, speichert und Analysen wie Durchschnittstemperaturen oder Niederschlagsmengen berechnet.

Vokabeln

Klassen
Konstruktor:
__init__ Initialisierung
self : Sonderparameter fürs Objekt
Feld = Attribut
eigene **Methoden**
Zugriff __geschuetzt
static ≈ übergreifend / global
__repr__ representation (oder __str__)
Hierarchie:
erben (Unterklasse)
super (Oberklasse)
überschreiben