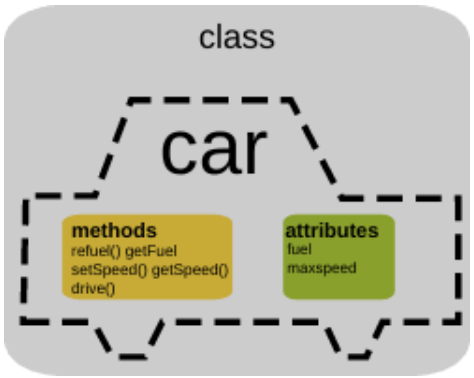
Objektorientierung in Python



Eine kompakte Zusammenfassung der Objektorientierung.

Basierend auf dem *Openbook Python 3* von Johannes Ernesti und Peter Kaiser, erschienen im Rheinwerk Verlag, sowie diversen Online-Tutorials.

Version 0.2
August 2022

Typesetting by LaTeX

Inhaltsverzeichnis

1 Object Orientated Programming (OOP)

Bei der **Objektorientierung** versucht man Bereiche der Wirklichkeit, die in einem Programm dargestellt werden soll, als sogenannte **Objekte** darzustellen.
Objekte haben **Attribute** (Eigenschaften) und **Methoden** (Funktionen die auf das Objekt wirken bzw. mit denen man mit dem Objekt interagieren kann).
Eine **Klasse** ist eine formale Beschreibung oder der **Bauplan** für ein **Objekt**. Es können aus einer Klasse beliebig viele Objekte **instanziiert** (erzeugt) werden.
Achtung: In der deutschsprachigen Literatur wird oftmals das Wort **Instanz** für ein Objekt verwendet. Beide Wörter meinen aber das Gleiche, also eine Instanz einer Klasse ist ein Objekt einer Klasse!

1.1 Klassen erzeugen

Klassen werden mit dem Keyword `class` erzeugt, gefolgt vom Namen der Klasse -> Achtung! Hier gibt es eine Abweichung zu bisherigen Namens-Konvention. Klassennamen werden groß und im `CamelCase`-Style geschrieben! Methoden und Attributnamen aber wie üblich klein und mit _ als Trennzeichen zwischen einzelnen Wörtern.

```
class BeispielKlasse:
        pass
```

1.1.1 init Methode

Die init-Methode ist eine spezielle Methode `__init__(self)`, die ein Objekt erzeugt bzw. instanziiert.
Das Spezielle daran ist ihr Name, er beginnt mit zwei Unterstrichen. Zudem ruft man sie nicht direkt über ihren Namen auf, sondern über den Klassennamen. Mehr zu speziellen Methoden gibt es im Kapitel 1.2 auf Seite 6.
Ebenfalls etwas seltsam erscheint der notwendige Parameter `self`. Er ist eine Referenz auf das zu erstellende Objekt (zum Vergleich: `self` entspricht in Java dem Keyword `this`).

oop-01.py:

```
'''
eine erste Beispielklasse
noch ohne Attribute
'''

class BeispielKlasse:
    def __init__(self):
        print("Hello from __init__")

my_first_object = BeispielKlasse()
print(my_first_object)
```

Liefert folgenden Output:

```
Hello from __init__
<__main__.BeispielKlasse object at 0x7f56be41c0d0>
```

1.1.2 Attribute

Man kann der init-Methode nun weitere Attribute übergeben, welche die Eigenschaften der Klasse bzw. eines Objektes beschreiben.
Als Beispiel wählen wir ein sehr einfaches Giro-Konto:

oop-02.py:

```
'''
ein einfaches Giro-Konto
mit Attributen aber
noch ohne Methoden
'''

class GiroKonto:
    def __init__(self, inhaber, kontonummer,
                 kontostand, overdraft = 0):
        self.inhaber = inhaber
        self.kontonummer = kontonummer
        self.kontostand = kontostand
        self.overdraft = overdraft


konto_1 = GiroKonto("MaSi",2712,927.75)
konto_2 = GiroKonto("AlDo",2713,75322.99,2000)

print(konto_1)
print(f"{konto_1.inhaber} hat "
      f"{konto_1.kontostand} Euro am Konto")
print(konto_2)
print(f"{konto_2.inhaber} hat "
      f"{konto_2.kontostand} Euro am Konto")
```

Liefert folgenden Output:

```
<__main__.GiroKonto object at 0x7f7238c8e0d0>
MaSi hat 927.75 Euro am Konto
<__main__.GiroKonto object at 0x7f7238af4970>
AlDo hat 75322.99 Euro am Konto
```

1.1.3 Methoden

Eine Methode unterscheidet sich äußerlich nur in zwei Aspekten von von einer Funktion:

- Sie ist eine Funktion, die innerhalb einer class-Definition definiert ist.
- Der erste Parameter einer Methode ist immer eine Referenz `self` auf die Instanz (Objekt), auf die sie aufgerufen wird.

Der Parameter `self` erscheint nur bei der Definition einer Methode. Beim Aufruf wird er nicht angegeben.
Beispiel mit Methode:

oop-03.py:

```
'''
ein einfaches Giro-Konto
mit Attributen und mit Methoden
aber noch ohne Kapselung
'''

class GiroKonto:
    def __init__(self, inhaber, kontonummer,
                 kontostand, overdraft = 0):
        self.inhaber = inhaber
        self.kontonummer = kontonummer
        self.kontostand = kontostand
        self.overdraft = overdraft

    #methoden
    def einzahlen(self, betrag):
       self.kontostand += betrag

    def auszahlen(self, betrag):
       self.kontostand -= betrag

    def get_kontostand(self):
        return self.kontostand

    def ueberweisen(self, ziel, betrag):
        if(betrag > self.kontostand + self.overdraft):
            # nicht gedeckt
            return False
        else:
            self.kontostand -= betrag
            ziel.kontostand += betrag
            return True



konto_1 = GiroKonto("MaSi",2712,927.75)
konto_2 = GiroKonto("AlDo",2713,75322.99,2000)

konto_1.einzahlen(1000)
#hacking ?!? :)
konto_1.kontostand = 1_000_000
print(f"{konto_1.get_kontostand()=}")
```

Liefert folgenden Output:

```
konto_1.get_kontostand()=1000000
```

1.1.4 Datenkapselung

Der kleine Kontostand-Hack im Beispiel `oop-03.py` von Seite 2 offenbart noch ein Problem: Man kann zur Zeit noch von außen direkt auf Attribute zugreifen! Dies ist in vielen Fällen nicht gewünscht, sondern meist nur über sogenannte `getter` oder `setter` Methoden oder über `properties` erlaubt. Diese bieten viele Vorteile, wie z.B. sanity checks -> bei einer Alterseingabe machen negative Zahlen wenig Sinn!
In manchen Programmiersprachen (z.B: Java) gibt es eine strenge Datenkapselung über Key-Words, in **Python gibt es diese strenge Kapselung nicht!**
Es gibt aber eine Namenskonvention, welche versucht dies abzubilden - je nach Zugriffsschutz werden ein oder zwei Unterstriche vorangestellt. Siehe Tabelle 1 auf Seite 4.

Namen	Bezeichnung	Bedeutung
`name`	public	Attribute ohne führende Unterstriche sind sowohl innerhalb einer Klasse als auch von außen schreib- und lesbar.
`_name`	protected	Man kann zwar auch von außen lesend und schreibend zugreifen, aber der Entwickler macht damit klar, dass man diese Member nicht benutzen sollte.
`__name`	private	Sind von außen nicht sichtbar und nicht benutzbar. (stimmt nicht ganz - es passiert im Hintergrund etwas, was man `name mangling` nennt. Der Name des Attributes wird von z.B. `__geek` auf `_classname__geek` verändert, aber dies sprengt den Rahmen dieses Kurses)

Tabelle 1: Datenkapselung

Beispiel mit Datenkapselung:

oop-04.py:

```
'''
ein einfaches Giro-Konto
mit privaten Attributen
'''

class GiroKonto:
    def __init__(self, inhaber, kontonummer,
                 kontostand, overdraft = 0):
        self.__inhaber = inhaber
        self.__kontonummer = kontonummer
        self.__kontostand = kontostand
        self.__overdraft = overdraft

    #methoden
    def einzahlen(self, betrag):
        if betrag > 0:
            self.__kontostand += betrag
            return True
        else:
            return False

    def get_kontostand(self):
        return self.__kontostand



konto_1 = GiroKonto("MaSi",2712,927.75)
konto_2 = GiroKonto("AlDo",2713,75322.99,2000)

konto_1.einzahlen(1000)
print(f"{konto_1.get_kontostand()=}")
print(konto_1.__kontostand)
```

Liefert folgenden Output:

```
konto_1.get_kontostand()=1927.75
Traceback (most recent call last):
  File "oop-04.py", line 32, in <module>
    print(konto_1.__kontostand)
AttributeError: 'GiroKonto' object has no attribute '__kontostand'
```

Diese Lösung funktioniert, ist aber nicht der von Python empfohlene Weg. Python empfiehlt generell mit öffentlichen Attributen zu arbeiten. Wenn für Attribute nur eingeschränkte Werte Sinn ergeben, soll auf sogenannte Properties zurückgegriffen werden. Siehe dazu Kapitel 1.1.5 auf Seite 5.

1.1.5 Property und @property Decorator

Wie im vorherigen Kapitel 1.1.4 auf Seite 3 schon erwähnt, gibt es zu der Möglichkeit mit `getter` und `setter` Methoden zu arbeiten, noch eine sogenannte **Property-Class**. Sie kann verwendet werden, um bestimmten Attributen nur bestimmte Werte zuzuordnen z.B. das Alter einer Person sollte nie negativ sein.

1.1.5.1 Property Class

Die Python property class gibt ein `property` Objekt zurück und hat die folgenden Syntax:
`property(fget=None, fset=None, fdel=None, doc=None)`
Die vier Parameter haben die folgende Bedeutung:

- `fget` eine Methode um den Wert des Attributes zu bekommen (getter method).
- `fset` eine Methode um den Wert des Attributes zu setzen (setter method).
- `fdel` eine Methode um das Attributes zu löschen.
- `fdoc` ist ein docstring also ein Kommentar zur Beschreibung.

Am einfachsten lässt sich die Verwendung von `property` anhand eines Beispiels erklären. oop-05.py:

```
'''
ein einfaches Beispiel
um properties zu erklaeren
'''

class Person:
    def __init__(self, name, age):
        #die attribute sind oeffentlich!
        self.name = name
        self.age = age

    #selbst definierte setter methode mit sanity check
    def set_age(self, age):
        if age <= 0:
            raise ValueError('The age must be positive')
        self._age = age

    #selbst definierte getter methode
    def get_age(self):
        return self._age

    #aufruf von property mit definierten Methoden
    age = property(fget=get_age, fset=set_age)


p1 = Person("Markus",10)
print(f"{p1.age=}")
p1.age = -3
```

Liefert folgenden Output:

```
p1.age=10
Traceback (most recent call last):
  File "oop-05.py", line 24, in <module>
    p1.age = -3
  File "oop-05.py", line 13, in set_age
    raise ValueError('The age must be positive')
ValueError: The age must be positive
```

Was hier passiert wirkt fast wie ein wenig Magie :)
Beim Zugriff auf das öffentliche Attribut `age` wird im Hintergrund immer die entsprechende *getter* oder *setter* Methode aufgerufen! Dabei ist es egal ob man versucht direkt auf das Attribut zuzugreifen mittels `p1.age` oder über den Konstruktor `p1 = Person("Future me",-2)`. Negative Werte für das Alter werden abgefangen!

1.1.5.2 @property Dekorator

Python wäre jedoch nicht Python, wenn es nicht noch einen eleganteren Weg gäbe. Dieser funktioniert mit Hilfe von Dekoratoren.
(Dekoratoren gehen eigentlich über den Umfang dieses Kurses hinaus, deshalb wird an dieser Stelle auf weiterführende Literatur zu diesem Thema verwiesen -> `https://www.python-kurs.eu/python3_dekorateure.php`)

Die Methode, die zum Abrufen eines Werts - also der Getter - verwendet wird, wird mit `@property` dekoriert, d.h. es steht direkt vor den Header der Methode. Der Name der Methode entspricht jetzt einfach dem Namen der Property, in unserem Fall also `age`. Die Methode, die als Setter fungieren muss, wird mit `@age.setter` dekoriert. Wenn die Property "foo"genannt worden wäre, müsste die Setter-Funktionalität mit `@foo.setter` dekoriert werden. Wieder passiert etwas bemerkenswertes: Man setzt einfach die Codezeile `self.age = age` in die `__init__`-Methode und die Property-Methode `age` wird verwendet, um die Grenzen der Werte zu überprüfen.

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

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value <= 0:
            raise ValueError('The age must be positive')
        self._age = value
```

1.2 Magic Methods

Es gibt in Python eine Reihe spezieller Methoden und Attribute, um Klassen besondere Fähigkeiten zu geben. Die Namen dieser Methoden und Attribute beginnen und enden jeweils mit zwei Unterstrichen. Im obigen Kapitel haben wir bereits eine dieser sogenannten Magic Methods kennengelernt, nämlich den **Konstruktor** namens `__init__`.
Der Umgang mit diesen Methoden und Attributen ist insofern *magisch*, als dass sie in der Regel **nicht direkt** mit ihrem Namen aufgerufen, sondern bei Bedarf implizit im Hintergrund **aufgerufen** werden. Der Konstruktor `__init__` wird beispielsweise immer dann aufgerufen, wenn ein neues Objekt einer Klasse erzeugt wird.
Es gibt eine ganze Reihe magischer Methoden und es würde den Umfang dieses Skriptes sprengen, deshalb werden hier nur ein paar einzelne Methoden vorgestellt.

- `__init__(self,...)`
  Der Konstruktor einer Klasse. Wird beim Erzeugen einer neuen Instanz (eines Objekts) automatisch aufgerufen.
- `__del__(self)`
  Der Finalizer einer Klasse. Wird beim Zerstören einer Instanz aufgerufen.
- `__str__(self)`
  Der Rückgabewert von `obj.__str__` gibt an, was str(obj) zurückgeben soll. Dies sollte nach Möglichkeit eine für den Menschen lesbare Repräsentation von obj in Form einer str-Instanz sein. Wird automatisch verwendet wenn man z.B: die `print()` Funktion mit `print(obj)` aufruft.
- `__repr__(self)`
  Der Rückgabewert von `obj.__repr__` ist sehr ähnlich zur `__str__(self)` Methode. Laut der Python-Doc ist `__repr__` die **offizielle** string Repräsentation eines Objektes und `__str__` die **informale**.

Für mehr Information siehe:
https://www.python-kurs.eu/python3_magische_methoden.php
oder
https://openbook.rheinwerk-verlag.de/python/21_007.html

Beispiel:

oop-06.py:

```
'''
Python Object ohne String-Repräsentation
'''

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

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value <= 0:
            raise ValueError('The age'
                ' must be positive')
        self._age = value


print(Person("MaSi",45))
```

Liefert folgenden Output:

```
<__main__.Person object at 0x7f52dbe280d0>
```

```
''
Python Object mit String-Repräsentation
'''

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

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value <= 0:
            raise ValueError('The age'
                ' must be positive')
        self._age = value

    #informale String Objektbeschreibung
    def __str__(self):
        return (f"{type(self)}:{self.name=}"
                f"{self.age=}")

print(Person("MaSi",45))
```

liefert folgenden Output:

```
class '__main__.Person'>:self.name='MaSi',self.age=45
```

1.2.1 if main ?

In manchen Python Programmen findet man das Konstrukt
`if __name__ == "__main__":`
Auch in diesem Fall wird so ein "magisches Attribut"verwendet, die Systemvariable `__name__`.
Dieses if-statement wird verwendet weil ein Python-File auf zwei Arten eingesetzt werden kann:

1. Als eigenständiges Programm, wenn wir z.B: ein File mit `python my_skript.py` ausführen.
2. Wenn wir das File als Module in einem anderen Programm importieren.

Im ersten Fall wird vom System die Variable `__name__` auf den Wert `"__main__"` gesetzt. Im zweiten Fall bekommt sie als Wert den Namen des Files, welches als Modul importiert wird.
So ist es nun möglich ein Python File sowohl als Modul als auch als ausführbares Programm zu verwenden. Dadurch ist es nicht nötig zwei Files für die zwei Fälle (Programm oder Modul) anlegen zu müssen.

```
'''
Python Programm das entweder direkt ausgeführt
wird oder als Modul importert wird.
'''

print("Dies wird immer ausgefuehrt")

if __name__ == "__main__":
    print("Dies nur wenn das File direkt ausgefuehrt wird")
else:
    print("Dies nur wenn das File als Modul importiert wird")
```

1.3 Klassenattribute, Klassenmethoden und statische Methoden

1.3.1 Klassenattribute

Bisher hatte jede Instanz (Objekt) einer Klasse ihre eigenen Attribute, die sich von denen anderer Instanzen unterschieden. Man bezeichnet dies als *nicht-statisch* oder *dynamisch*, da sie für jede Instanz einer Klasse dynamisch erstellt werden.
Manchmal möchte man aber auch Attribute, die für alle Instanzen gleich sind. Diese Attribute bezeichnet man als **statische** Attribute. Statische Attribute werden auch als Klassenattribute bezeichnet, weil sie Eigenschaften bezeichnen, die für die ganze Klasse gelten und nicht nur für einzelne Objekte der Klasse. Ein Klassenattribut existiert pro Klasse nur einmal, wird also nur einmal angelegt. Instanzattribute werden für jedes Objekt angelegt. Statische Attribute werden außerhalb der Instanzmethoden direkt im class-Block definiert. Außerdem muss jedem Klassenattribut ein Initialwert zugewiesen werden.
Beispiel:

oop-08.py:

```
'''
Python Object mit Klassenattribut
'''

class Person:

    #static or class variable
    species = "Human"

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

if __name__ == "__main__":
    p1 = Person("MaSi",45)
    p2 = Person("AlDo",40)
    print(f"{p1.species=}")
    print(f"{type(p2).species=}")
    print(f"{Person.species=}")
```

Liefert folgenden Output:

```
p1.species='Human'
type(p2).species='Human'
Person.species='Human'
```

Auf die Klassenattribute kann mittels Punktnotation entweder über den Klassennamen, oder über Instanzen (Objekte), zugegriffen werden.

Eine mögliche Anwendung einer Klassenvariable ist ein **Instanzzähler** - der mitzählt wie viele Objekte (Instanzen) einer Klasse es gibt.
In dem folgenden Beispiel wird wieder einer Magic Method aus Kapitel 1.2 von Seite 6 verwendet: `__del__` wird aufgerufen wenn eine Instanz gelöscht wird!
Beispiel:

oop-09.py:

```
'''
Python Object mit Klassenattribut
als Objekt-counter Anwendung
'''

class Person:

    #static or class variable
    counter = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age
        type(self).counter += 1

    def __del__(self):
        type(self).counter -= 1

if __name__ == "__main__":
    p1 = Person("MaSi",45)
    p2 = Person("AlDo",40)
    print(f"{Person.counter=}")
    #delete an object
    del p1
    print(f"{Person.counter=}")
```

Liefert folgenden Output:

```
Person.counter=2
Person.counter=1
```

Noch ein Wort zum **Zugriff** auf Klassenvariablen. Wenn es wie im folgenden Beispiel sowohl eine Instanzvariable als auch eine Klassenvariable mit dem selben Namen gibt, kommt es auf die Zugriffsart an. Verwendet man den Klassennamen bekommt man den Wert der Klassenvariable und verwendet man ein Objekt, bekommt man den Wert der Instanzvariable.

oop-10.py:

```
'''
Python Object mit Klassenattribut
'''

class Test:

    x = 10

    def __init__(self, x):
        self.x = x

t1 = Test(20)
print(f"{t1.x=}")
print(f"{Test.x=}")
```

Liefert folgenden Output:

```
t1.x=20
Test.x=10
```

1.3.2 Statische Methode

Bisher wurden nur Instanz-Methoden besprochen, die immer die Referenz `self` brauchen, um auf die Instanz (das Objekt) zuzugreifen.
Was aber, wenn man z.B. ein Klassenattribut kapseln möchte? Also nur über eine Methode darauf zugreifen möchte? Dazu braucht es Methoden die ohne Referenz auf ein Objekt (`self`) funktionieren - sogenannte **statische Methoden**. Schreibt man vor einer Methoden-Definition den Dekorator `@staticmethod`, kann `self` als Parameter weggelassen werden und die Methode ist nun sowohl über den Klassennamen als auch über eine Instanz aufrufbar.

oop-11.py:

```
'''
Python Object mit
statischer Methode
'''

class Test:

    #private class variable
    __counter = 0

    def __init__(self, x):
        self.x = x
        type(self).__counter += 1

    def __del__(self):
        type(self).counter -= 1

    @staticmethod
    def how_many():
        return Test.__counter

t1 = Test(20)
print(f"{Test.how_many()=}")
t2 = Test(30)
print(f"{t1.how_many()=}")
#the following will not work!
print(f"{Test.__counter=}")
```

Liefert folgenden Output:

```
Test.how_many()=1
Test.how_many()=2
Traceback (most recent call last):
  File "oop-11.py", line 26, in <module>
    print(f"{Test.__counter=}")
AttributeError: type object 'Test' has no
                attribute '__counter'
```

1.3.3 Klassenmethode

Klassenmethoden sind **nicht** an Instanzen gebunden sondern an die Klasse. Das erste Argument einer Klassenmethode ist eine Referenz auf die Klasse, d.h. das Klassenobjekt. Aufrufen kann man sie über den Klassennamen oder eine Instanz. Man kann sie z.B. verwenden um eine Instanz einer Klasse zu erzeugen - dann spricht man von `factory methods`. Weiters bieten sie bei der Vererbung (siehe Kapitel 1.4 auf Seite 12) Vorteile gegenüber statischen Methoden, weil man den Klassennamen nicht fest kodieren muss, sondern über die `cls`-Referenz zugreifen kann.
Erzeugt werden sie mit dem Dekorator `@classmethod` und als erstes Argument erhalten sie statt `self` eine Referenz auf die Klasse - `cls`.

oop-12.py:

```
'''
Python Object mit
Klassen-Methode
'''

class Test:

    #private class variable
    __counter = 0

    def __init__(self, x):
        self.x = x
        type(self).__counter += 1

    @classmethod
    def how_many(cls):
        return cls.__counter

t1 = Test(20)
print(f"{Test.how_many()=}")
```

Liefert folgenden Output:

```
Test.how_many()=1
```

Noch ein Beispiel:

oop-13.py:

```
'''
Klasse um einen Bruch in seiner
gekürzten Form darzustellen

Verwendet statische und Klassen Methoden
'''

class Bruch():
    def __init__(self,z,n):
        self.__z, self.__n = self.kuerze(z,n)

    def __str__(self):
        return str(self.__z)+'/'+str(self.__n)

    #größten gemeinsamer Teiler bestimmen
    @staticmethod
    def ggT(a,b):
        while b != 0:
            a,b = b,a%b
        return a

    @classmethod
    def kuerze(cls, zaehler, nenner):
        g = cls.ggT(zaehler, nenner)
        return (zaehler // g, nenner // g)
b1 = Bruch(8,24)
print(b1)
b2 = Bruch(6,8)
print(b2)
```

Liefert folgenden Output:

```
1/3
3/4
```

1.4 Vererbung

Vererbung bedeutet, dass man von bereits bestehenden Klassen neue Klassen ableitet, um diese um zusätzliche Funktionalität zu erweitern. Dabei übernimmt die abgeleitete Klasse alle Fähigkeiten von ihrer Superklasse, sodass sie zunächst eine Kopie dieser Klasse ist. Man sagt, die Superklasse vererbt ihre Fähigkeiten an eine Tochter-, Unter- oder Subklasse.
Die Syntax dafür lautet:
`class MySubClassName(SuperClassName)`
Vererbung ist einer der wichtigsten Pluspunkte der Objekt-Orientierten-Programmierung, weil sie es erlaubt, den bestehenden Code schnell zu erweitern ohne nochmals die volle Funktionalität schreiben zu müssen!
Am einfachsten zeigt sich dies anhand eines Beispiels:
Angenommen wir haben eine Klasse Person. Eine Person hat einen Namen und den kann sie mit der Methode vorstellen(self) auch ausgeben:

oop-14.py:

```
'''
Klasse um eine Person zu beschreiben
'''
class Person:
    def __init__(self, name):
        self.name = name

    def vorstellen(self):
        return f"Hi, ich bin {self.name}"
```

Angenommen man benötigt nun eine neue Klasse `Angestellter`. Ein Angestellter soll einen Namen und einen Jobtitel haben und sich vorstellen können. Bis auf den Jobtitel ist diese Funktionalität aber schon in der Klasse Person abgebildet! Wenn man diese Funktionalität nun nicht nochmals schreiben will, kann man sie einfach von der Klasse Person erben!
Mit dem Key-Word `super()` kann man auf die Methoden der Superklasse zugreifen!

oop-15.py:

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

    def vorstellen(self):
        return f"Hi, ich bin {self.name}"

class Angestellter(Person):
    def __init__(self, name, job_title):
        self.name = name
        self.job_title = job_title

sepp = Person("Sepp")
franz = Angestellter("Franz","Chef")
#kann Methode vorstellen aufrufen!
print(franz.vorstellen())
print(f"{type(franz)=}")
print(f"{isinstance(franz,Person)=}")
print(f"{isinstance(franz,Angestellter)=}")
print(f"{isinstance(sepp,Person)=}")
print(f"{isinstance(sepp,Angestellter)=}")
```

Liefert folgenden output:

```
Hi, ich bin Franz
type(franz)=<class '__main__.Angestellter'>
isinstance(franz,Person)=True
isinstance(franz,Angestellter)=True
isinstance(sepp,Person)=True
isinstance(sepp,Angestellter)=False
```

1.4.1 super()

Angenommen man möchte die Funktionalität der Methode `vorstellen(self)` in der Klasse `Angestellter` so erweitern, dass auch der Jobtitel Teil der Vorstellung sein soll.
Mit Hilfe des keywords `super()` kann man auf die Methoden und Attribute der Superklasse zugreifen!
oop-16.py

```
'''
Vererbung und super()
'''
class Person:
    def __init__(self, name):
        self.name = name

    def vorstellen(self):
        return f"Hi, ich bin {self.name}"

class Angestellter(Person):
    def __init__(self, name, job_title):
        #rufe init der super-klasse
        super().__init__(name)
        self.job_title = job_title

    def vorstellen(self):
        return f"{super().vorstellen()} und ich bin {self.job_title}"


franz = Angestellter("Franz","Chef")
#kann Methode vorstellen aufrufen!
print(franz.vorstellen())
```

Liefert folgenden output:

```
Hi, ich bin Franz und ich bin Chef
```

1.4.2 Mehrfachvererbung und MRO (Method Resolution Order)

Manche Programmiersprachen (z.B. Java) erlauben nur *single-inheritence*, also einfache Vererbung.
Dies wird oft mit dem DDD (Deadly Diamond of Death) begründet: Angenommen man würde von zwei Klassen (ClassOne und ClassTwo) erben, beide habe eine Methode `m()` aber mit unterschiedlicher Funktionalität. Welche der zwei Methoden `m()` würde die Subklasse nun erben?
Python hat eine Lösung für dieses Problem: **MRO** oder Method Resolution Order:
Bei der Syntax der Mehrfachvererbung **zählt die Reihenfolge** in der man die Super-Klassen angibt. Man erbt immer von der an erster Stelle stehenden Klasse und genauso kann mit dem keyword `super()` nur auf die an erster Stelle stehende Klasse zugreifen!
Beispiel:

oop-17.py:

```
'''
Mehrfachvererbung und MRO
'''
class A:
    def m(self):
        print("m() of A")

class B:
    def m(self):
        print("m() of B")

class C(A,B):
    pass
class D(B,A):
    pass

c = C()
c.m()
d = D()
d.m()
```

Liefert folgenden output:

```
m() of A
m() of B
```

Mit dem Magic-Parameter `__mro__` kann man die Vererbungskette einer Klasse anzeigen. Wenn man z.B. in der Subklasse eine Methode einer Superklasse aufruft die nicht in allen Superklassen definiert ist, so wird entlang der MRO alle Superklassen durchsucht bis die Methode gefunden wird. Beispiel:

oop-18.py:

```
'''
Mehrfachvererbung und MRO
'''
class A:
    def m(self):
        print("m() of A")

class B:
    def m(self):
        print("m() of B")
    def x(self):
        print("special method x of class B")

class C(A,B):
    pass

class D(B,A):
    pass

c = C()
print(f"{C.__mro__=}")
c.m()
c.x()
d = D()
print(f"{D.__mro__=}")
d.m()
```

Liefert folgenden output:

```
C.__mro__=(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
m() of A
special method x of class B
D.__mro__=(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
m() of B
```

2 File IO - Wiederholung

Info: Dieses Kapitel ist eine Wiederholung aus dem Anfänger-Skript welche um das **with**-keyword (Kapitel 2.5 auf Seite 18) sowie um **Serialization** (Kapitel 3.1 auf Seite 21) und **JSON** (Kapitel 3.2 auf Seite 21) erweitert wurde.

Bevor wir Dateien behandeln müssen wir kurz das Konzept von Datenströmen (*data streams*) behandeln.
Unter einem *data stream* versteht man eine kontinuierliche Folge von Daten. Dabei werden zwei Typen unterschieden: Von eingehenden Datenströmen (*downstreams*) können Daten gelesen und in ausgehende Datenströme (*upstreams*) geschrieben werden. Bildschirmausgaben, Tastatureingaben sowie Dateien und Netzwerkverbindungen werden als Datenstrom betrachtet.
Es gibt zwei Standarddatenströme, die wir bereits verwendet haben, ohne es zu wissen: Sowohl die Ausgabe eines Strings auf dem Bildschirm, als auch eine Benutzereingabe sind nichts anderes als Operationen auf den Standardeingabe- bzw. -ausgabeströmen *stdin* und *stdout*. Auf den Ausgabestrom *stdout* kann mit der eingebauten Funktion `print()` geschrieben und von dem Eingabestrom *stdin* mittels `input()` gelesen werden.
Einige Betriebssysteme erlauben es, Datenströme im *Text-* und *Binärmodus* zu öffnen. Der Unterschied besteht darin, dass im Textmodus bestimmte Steuerzeichen berücksichtigt werden. So wird ein im Textmodus geöffneter Strom beispielsweise nur bis zum ersten Auftreten des sogenannten `EOF`-Zeichens gelesen, das das Ende einer Datei (*end of file*) signalisiert. Im Binärmodus hingegen wird der vollständige Inhalt des Datenstroms eingelesen.
Als letzte Unterscheidung gibt es Datenströme, in denen man sich beliebig positionieren kann und solche, in denen das nicht geht. Eine Datei stellt zum Beispiel einen Datenstrom dar, in dem die Schreib-/Leseposition beliebig festgelegt werden kann. Beispiele für einen Datenstrom, in dem das nicht funktioniert, sind der Standardeingabestrom (*stdin*) oder eine Netzwerkverbindung.

2.1 Daten aus einer Datei lesen

Zunächst einmal muss die Datei zum Lesen geöffnet werden. Dazu verwenden wir die Built-in Function `open()`. Diese gibt ein sogenanntes Dateiobjekt (*file object*) zurück:

```
fobj = open("cd-sammlung.csv","r")
```

Als ersten Parameter von `open()` übergeben wir einen String, der den Pfad zur gewünschten Datei enthält. Es sind sowohl relative als auch absolute Pfade erlaubt! Ein absoluter Pfad identifiziert eine Datei ausgehend von der Wurzel im Dateisystembaum. [1] Ein relativer Pfad bezieht sich auf das aktuelle Arbeitsverzeichnis des Programms. Hier kann die Verknüpfung *»..«* für das übergeordnete Verzeichnis verwendet werden. Im Beispiel ist ein relativer Pfad angegeben, die Datei *cd-sammlung.csv* muss sich also im gleichen Verzeichnis befinden wie das Programm.
Der zweite Parameter ist ebenfalls ein String und spezifiziert den Modus, in dem die Datei geöffnet werden soll, wobei *"r"* für *»read«* steht und bedeutet, dass die Datei zum Lesen geöffnet wird. Das von der Funktion zurückgegebene Dateiobjekt verknüpfen wir mit der Referenz `fobj`. Sollte die Datei nicht vorhanden sein, wird ein `FileNotFoundError` erzeugt.

Nachdem `open()` aufgerufen wurde, können mit dem Dateiobjekt Daten aus der Datei gelesen werden. Nachdem das Lesen der Datei beendet worden ist, muss sie explizit durch Aufrufen der Methode `.close()` geschlossen werden:

```
fobj.close()
```

Nach Aufruf dieser Methode können keine weiteren Daten mehr aus dem Dateiobjekt gelesen werden.

[1] Unter Windows könnte ein absoluter Pfad folgendermaßen aussehen:
C:\Programme\TestProgramm\cd-sammlung.csv
und unter Linux:
/home/TestProgram/cd-sammlung.csv

Im nächsten Schritt möchten wir die Datei zeilenweise auslesen. Dies ist relativ einfach, da das Dateiobjekt zeilenweise iterierbar ist. Wir können also die altbekannte `for`-Schleife verwenden:

Zeilenweises Lesen des Files:

```
fobj = open("cd-sammlung.csv", "r")
for line in fobj:
    print(line)
fobj.close()
```

Zugegehöriger Output:

```
CD-Code,Title

CL01,Vivaldi Die vier Jahreszeiten

CL02,Bach Toccata und Fuge

MT01,Judas Priest Screaming for Vengeance

MT02,Metallica Black Album

GR01,Nirvana Nevermind

GR02,Green Day Dookie


```

Hm, wo kommen die extra Leerzeilen her? Sie sind im File so nicht enthalten!
Wenn wir das file nochmals im binary mode öffnen sehen wir am Ende jeder Zeile das Steuerzeichen `\n` welches eigentlich den Zeilensprung anzeigt.

Zeilenweises Lesen des Files im Byte Modus:

```
fobj = open("cd-sammlung.csv", "rb")
for line in fobj:
    print(line)
fobj.close()
```

Zugegehöriger Output:

```
b'CD-Code,Title\n'
b'CL01,Vivaldi Die vier Jahreszeiten\n'
b'CL02,Bach Toccata und Fuge\n'
b'MT01,Judas Priest Screaming for Vengeance\n'
b'MT02,Metallica Black Album\n'
b'GR01,Nirvana Nevermind\n'
b'GR02,Green Day Dookie\n'
```

Da wir aber die `print()`-Funktion zum Ausgeben verwenden, welche standardmäßig sowieso einen Zeilensprung anhängt, wirkt es als ob eine Leerzeile vorhanden wäre.
Mit Hilfe der `.strip()` Methode, kann man Teile eines Strings entfernen. Verwenden wir sie um das Steuerzeichen `\n` aus dem String zu entfernen:

Zeilenweises Lesen des Files
und Steuerzeichen entfernen:

```
fobj = open("cd-sammlung.csv", "r")
for line in fobj:
    print(line.strip("\n"))
fobj.close()
```

Zugegehöriger Output:

```
CD-Code,Title
CL01,Vivaldi Die vier Jahreszeiten
CL02,Bach Toccata und Fuge
MT01,Judas Priest Screaming for Vengeance
MT02,Metallica Black Album
GR01,Nirvana Nevermind
GR02,Green Day Dookie
```

Nun haben wir den gewünschten Output!
Mit der `.split()` Methode kann man nun den Zeilenstring noch weiter aufteilen, in dem als Trennzeichen das Komma angeben wird, und so den File-Inhalt weiterverarbeiten. Zum Beispiel in diesem kleinen CD-Code Abfrage Programm:

CD-Code Abfrage Programm:

```
fobj = open("cd-sammlung.csv", "r")
sammlung = {}
for line in fobj:
    line = line.strip("\n")
    mini_dict = line.split(",")
    sammlung[mini_dict[0]] = mini_dict[1]
fobj.close()

while(True):
    cd_code = input("Bitte CD-Code eingeben ->")
    if cd_code in sammlung:
        print("Der CD-Code verweist auf:", sammlung[cd_code])
    else:
        if (cd_code == "end"):
            print("bye!")
            break
        print("Unbekannter CD-Code")
```

Zugegehöriger Output:

```
Bitte CD-Code eingeben ->MT01
Der CD-Code verweist auf: Judas Priest Screaming for Vengeance
Bitte CD-Code eingeben ->GR01
Der CD-Code verweist auf: Nirvana Nevermind
Bitte CD-Code eingeben ->end
bye!
```

2.2 Daten in eine Datei schreiben

Um eine Datei zum Schreiben zu öffnen, verwenden wir ebenfalls die Built-in Function `open()`. Diese Funktion erwartet einen Modus als zweiten Parameter, der im letzten Abschnitt `"r"` für *»read«* sein musste. Analog dazu muss `"w"` (für *»write«*) angegeben werden, wenn die Datei zum Schreiben geöffnet werden soll. Sollte die gewünschte Datei bereits vorhanden sein, wird sie geleert. Eine nicht vorhandene Datei wird erstellt.
Es gibt außer `"r"` und `"w"` noch andere Möglichkeiten, siehe dazu Tabelle 2 auf Seite 18.
File zum Schreiben öffnen:

```
fobj = open("ausgabe.txt", "w")
```

und wie vorher gilt, dass die Datei nach dem Schreiben geschlossen werden muss:

```
fobj.close()
```

Als kleines Beispiel wollen wir die Zahlen von 1 bis 4 zeilenweise in ein File schreiben:

Zeilenweises Schreiben:

```
fobj = open("numbers.txt","w")
for i in range(1,5):
    fobj.write("{}\n".format(i))
fobj.close()
```

File-Inhalt:

```
1
2
3
4
```

Bei Schreiben müssen wir uns selbst um den Zeilenumbruch kümmern, deshalb ist in der `.write()` Methode explizit das Steuerzeichen `\n` angegeben.

2.3 Mehr zum File-object

2.3.1 Optionen bei der Erzeugung

Die Built-in Function `open()` öffnet eine Datei und gibt das erzeugte Dateiobjekt zurück.
Die ersten beiden Parameter haben wir in den vorangegangenen Abschnitten bereits besprochen. Dabei handelt es sich um den Dateinamen bzw. den Pfad zur zu öffnenden Datei und um den Modus, in dem die Datei zu öffnen ist.

Für den Parameter *mode* muss ein String übergeben werden, wobei alle gültigen Werte und ihre Bedeutung in Tabelle 2 aufgelistet sind:

Modus	**Beschreibung**
`"r"`	Die Datei wird ausschließlich zum Lesen geöffnet.
`"w"`	Wird zum Schreiben geöffnet. Bestehende Datei wird überschrieben.
`"a"`	Wird zum Schreiben geöffnet. Bestehende Datei gleichen Namens wird **nicht** überschrieben, sondern erweitert.
`"x"`	Wird zum Schreiben geöffnet. Wenn bereits vorhanden wird eine FileExistsError-Exception geworfen.
`"r+", "w+","a+", "x+"`	Die Datei wird zum Lesen und Schreiben geöffnet.
`"rb", "wb", "ab", "xb", "r+b", "w+b", "a+b", "x+b"`	Die Datei wird im Binärmodus geöffnet. Achtung: In diesem Fall werden bytes-Instanzen anstelle von Strings verwendet.

Tabelle 2: File modi

Es gibt noch weitere Parameter wie z.B. das verwendete File-Encoding, was aber den Umfang dieser Einführung sprengen würde. Für mehr Informationen bitte einen Blick auf die offizielle Python Docs werfen:
`https://docs.python.org/3/library/functions.html#open`

2.4 Methoden und Attribute eines File-objects

Die beim Öffnen angegebenen Parameter können über die Attribute `name`, `encoding`, `errors`, `mode` und `newlines` des resultierenden Dateiobjekts wieder gelesen werden.
Die folgende Tabelle 3 fasst die wichtigsten Methoden eines Dateiobjekts kurz zusammen:

Methode	**Beschreibung**
`close()`	Schließt ein bestehendes Dateiobjekt. Danach kann keine Lese- oder Schreiboperationen mehr durchgeführt werden.
`fileno()`	Gibt den Deskriptor der geöffneten Datei als ganze Zahl zurück.
`flush()`	Verfügt, dass anstehende Schreiboperationen sofort ausgeführt werden.
`isatty()`	True, wenn das Dateiobjekt auf einem Datenstrom geöffnet wurde, der nicht an beliebiger Stelle geschrieben oder gelesen werden kann.
`next()`	Liest die nächste Zeile der Datei ein und gibt sie als String zurück.
`read([size])`	Liest size Bytes der Datei ein oder weniger, wenn vorher das Ende der Datei erreicht wurde. Sollte size nicht angegeben sein, wird die Datei vollständig eingelesen. Die Daten werden abhängig vom Lesemodus als String oder bytes-String zurückgegeben.
`readline([size])`	Liest eine Zeile der Datei ein. Durch Angabe von size lässt sich die Anzahl der zu lesenden Bytes begrenzen.
`readlines([sizehint])`	Liest alle Zeilen und gibt sie in Form einer Liste von Strings zurück. Sollte *sizehint* angegeben sein, wird nur gelesen, bis ungefähr • Bytes gelesen wurden.
`seek(offset, [whence])`	Setzt die aktuelle Schreib-/Leseposition in der Datei auf offset.
`tell()`	Liefert die aktuelle Schreib-/Leseposition in der Datei.
`truncate([size])`	Löscht in der Datei alle Daten, die hinter der aktuellen Schreib-/Leseposition stehen, bzw. – sofern angegeben – alles außer den ersten size Bytes.
`write(str)`	Schreibt den String str in die Datei.
`writelines(iterable)`	Schreibt mehrere Zeilen in die Datei. Das iterierbare Objekt iterable muss Strings durchlaufen, möglich ist zum Beispiel eine Liste von Strings.

Tabelle 3: Methoden eines File-objects

2.5 with - Anweisung

Es gibt Operationen, die in einem bestimmten **Kontext** ausgeführt werden müssen und bei denen sichergestellt werden muss, dass der Kontext jederzeit korrekt de-initialisiert wird, beispielsweise auch, wenn eine Exception auftritt. Ein Beispiel für einen solchen Kontext ist ein **geöffnetes Dateiobjekt**: Es muss **sichergestellt** sein, dass die `close`-Methode des Dateiobjekts gerufen wird, selbst wenn zwischen dem Aufruf von open und dem der close-Methode des

Dateiobjekts eine **Exception** geworfen wurde.
Für diese Zwecke definiert Python sogenannte **Kontextobjekte**. Das sind Objekte, die über die Magic Members `__enter__` und `__exit__` verfügen, um den Kontext zu betreten bzw. zu verlassen. Speziell für die Verwendung von Kontextobjekten existiert die `with`-Anweisung.
Am folgenden Beispiel werden die Vorteile der `with`-Anweisung erläutert:
Zum sicheren Datei-Handling gehört auch das Exception-Handling. D.h. normalerweise schreiben wir folgenden, abgesicherten Code:
file-01.py

```
'''
Sicheres File-IO mit Hilfe von try/except/finally
'''
#vorsichtiges file oeffnen
try:
    f = open("datei.txt", "r")
    #vorsichtiges lesen
    try:
        print(f.read())
    except:
        print("Ups, could not read file!")
    finally:
        f.close()
except:
    print("could not open file!")
```

Der Nachteil dieser Schreibweise ist, dass sich der Programmierer darum kümmern muss, dass das Dateiobjekt korrekt de-initialisiert wird. Die `with`-Anweisung überträgt diese Verantwortung an das Objekt selbst und erlaubt eine kurze und elegante Alternative zu dem oben dargestellten Code:
file-02.py

```
'''
Komfortables File IO mit with
'''
#muss immer noch ueberpruefen ob
#das file existiert, aber um das
#Lesen und Schliessen kuemmert
#sich die with-Anweisung
try:
    with open("datei.txt", "r") as f:
        print(f.read())
except:
    print("could not open file!")
```

2.5.1 __enter__ und __exit__

Um zu verstehen, was bei einer `with`-Anweisung genau passiert, definieren wir im nächsten Beispiel eine eigene Klasse, die sich mit der `with`-Anweisung verwenden lässt. Eine solche Klasse wird Kontext-Manager genannt.
Die Klasse MeinLogfile ist dazu gedacht, eine einfache Logdatei zu führen. Dazu implementiert sie die Methode `eintrag`, die eine neue Zeile in die Logdatei schreibt. Die Klassendefinition sieht folgendermaßen aus:

```
'''
Einfache Klasse um mit Logfile zu arbeiten
Erklaert __enter__ und __exit__
'''
class MeinLogfile:
    def __init__(self, logfile):
        self.logfile = logfile
        self.f = None
    def eintrag(self, text):
        self.f.write(f"==>{text}\n")
    def __enter__(self):
        self.f = open(self.logfile, "w")
        return self
    def __exit__(self, exc_type, exc_value, traceback):
        self.f.close()
```

Die ersten beiden Methoden der Klasse sind leicht verständlich. Dem Konstruktor`__init__` wird der Dateiname der Logdatei übergeben, der intern im Attribut `self.logfile` gespeichert wird. Zusätzlich wird das Attribut `self.f` angelegt, das später das geöffnete Dateiobjekt referenzieren soll.
Die Methode `eintrag` hat die Aufgabe, den übergebenen Text in die Logdatei zu schreiben. Dazu ruft sie die Methode `write` des Dateiobjekts auf. Achtung: `eintrag` kann nur innerhalb einer with-Anweisung aufgerufen werden, da das Dateiobjekt erst in den folgenden *Magic Methods* geöffnet und geschlossen wird.
Nun zu den *Magic Methods*: `__enter__` und `__exit__` sind das Herzstück der Klasse und müssen implementiert werden, wenn die Klasse im Zusammenhang mit `with` verwendet werden soll. Die Methode `__enter__` wird aufgerufen, wenn der Kontext aufgebaut, also bevor der Körper der `with`-Anweisung ausgeführt wird. Die Methode bekommt keine Parameter, gibt aber einen Wert zurück. Der Rückgabewert von `__enter__` wird später vom Target-Bezeichner referenziert, sofern einer angegeben wurde. Im Falle unserer Beispielklasse wird die Datei `self.logfile` zum Schreiben geöffnet und mit `return self` eine Referenz auf die eigene Instanz zurückgegeben.
Die zweite *Magic Method* `__exit__` wird aufgerufen, wenn der Kontext verlassen wird, also nachdem der Körper der `with`-Anweisung entweder vollständig durchlaufen oder durch eine Exception vorzeitig abgebrochen wurde. Im Falle der Beispielklasse wird das geöffnete Dateiobjekt `self.f` geschlossen.

Die soeben erstellte Klasse MeinLogfile lässt sich folgendermaßen mit with verwenden:

```
with MeinLogfile("logfile.txt") as log:
    log.eintrag("Hallo Welt")
    log.eintrag("Na, wie gehts?")
```

3 Objekte mittels Json und/oder Serialization speichern

3.1 Serialization

Serialization wird benötigt sobald man versucht *"komplizierte"* Dinge wie Instanzen von selbst definierten Klassen in Files zu speichern (bisher wurden nur einfache Strings gespeichert).

Unter **Serialization** versteht man die Umwandlung einer Objekthierarchie in einen Byte-Stream (wird z.B. beim Schreiben in Files verwendet). **De-serialization** ist die umgekehrte Operation, bei der ein Byte-Stream (aus einer Binärdatei oder einem bytes-ähnlichen Objekt) wieder in eine Objekthierarchie umgewandelt wird.

In Python steht das Modul `pickle` für **Serialization** zur Verfügung. Details dazu findet man unter:
`https://docs.python.org/3/library/pickle.html?highlight=pickle#module-pickle`
Es stellt mehrere Funktionen (`dump`, `dumps`, `load`, `loads`, usw.) bereit um Daten zu *"pickeln"* und *"un-pickeln"*.
Im folgenden Beispiel wird mit Instanzen der Pickler bzw. Unpickler Klasse gearbeitet:
serial-02.py

```
'''
objekte mit pickle serialisieren
'''
import pickle

class TestKlasse:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"{type(self)}:{self.x=} and {self.y=}"

t1 = TestKlasse(12,"hello")
t2 = TestKlasse((1,2,3),["my","list",[1,2,3]])

with open("save.sav", "wb") as f:
        p = pickle.Pickler(f)
        p.dump(t1)
        p.dump(t2)

x1 = None
x2 = None
with open("save.sav", "rb") as f:
        u = pickle.Unpickler(f)
        x1 = u.load()
        x2 = u.load()

print(x1)
print(x2)
```

liefert folgenden output:

```
<class '__main__.TestKlasse'>:self.x=12 and self.y='hello'
<class '__main__.TestKlasse'>:self.x=(1, 2, 3) and self.y=['my', 'list', [1, 2, 3]]
```

Achtung:
-> Zugriff auf die Files erfolgt im Byte-Modus. Sie sind dadurch für Menschen nicht lesbar!
-> Das pickle Modul ist nicht sicher - es ist möglich Pickle-Files zu erstellen, die beim un-pickeln bösartigen Code ausführen! Kann durch HMAC (Keyed-Hashing for Message Authentication) abgesichert werden.

3.2 JSON File Format

Das ursprünglich für JavaScript entwickelte Datenformat **JSON** (JavaScript Object Notation) hat sich zu einem Quasi-Standard für den einfachen Datenaustausch zwischen Webanwendungen entwickelt und konkurriert damit gewisser-

maßen mit XML. Im Gegensatz zur Markup-Sprache XML speichert **JSON** Daten in Form von gültigem JavaScript-Code. Trotzdem gibt es **JSON-Parser** für alle verbreiteten Programmiersprachen, selbstverständlich auch für Python. Ein großer Vorteil von JSON ist, dass es auf Text basiert und relativ leicht lesbar ist.
Ein in JSON repräsentiertes Objekt kann aus den in Tabelle 4 aufgelisteten **Datentypen** zusammengestellt werden, für die es jeweils ein Gegenstück in Python gibt. Anbei ein Beispiel eines JSON Files:

JSON-Datentyp	**Notation**	**Korrespondierender Python-Datentyp**
Object|	`{}`	dict
Array|	`[]`	list
Number|	`12,42.17`	int, float
String|	`" "`	str
Value|	`true,false,null`	bool, bool, NoneType

Tabelle 4: JSON-Datentypen

JSON-Beispiel:

```
{
  "student": [

     {
        "id":"01",
        "name": "Tom",
        "lastname": "Price"
     },

     {
        "id":"02",
        "name": "Nick",
        "lastname": "Thameson"
     }
  ]
}
```

3.2.1 JSON and Python

In der Standardbibliothek existiert das Modul `json`, das Python-Instanzen ins JSON-Format serialisieren bzw. aus dem JSON-Format erstellen kann.
Ähnlich wie `pickle` bietet das Modul `json` die Funktionen `dump`, `dumps` bzw. `load` und `loads` an, um Daten zu speichern bzw. zu laden.
Ein JSON-Objekt wird in Python als Dictionary abgebildet. Auf die Eigenschaften kann über den Namen der Eigenschaft zugegriffen werden. Umgekehrt muss aus einer Instanz (Objekt) ein Dictonary gemacht werden, bevor es als JSON-File gespeichert werden kann.
Um die Lesbarkeit der entstehenden JSON-Files zu erhöhen, kann man den json-Methoden noch die Parameter `indent` und `separators` mitgeben. z.B:
`json.dump(meine_daten,file_name,indent=2,separators=(", ","="))`
Als Beispiel dazu siehe Kapitel 3.2.1.2 auf Seite 23.

3.2.1.1 Daten in JSON speichern

Die Daten in ein Listen-Format bringen und mittels `json.dump` in ein JSON-File speichern.
json-01.py

```
'''
daten in json speichern
'''
import json

my_list = [1,2,3,"numbers"]
my_dict = {1:"one", 2:"two", 3:"three"}
my_data = [my_list, my_dict]

with open("meine_daten.json","w") as f:
    json.dump(my_data,f)

x1 = None
with open("meine_daten.json","r") as f:
    x1 = json.load(f)

print(f"{x1[0]=}")
print(f"{x1[1]=}")
```

liefert folgenden output:

```
x1[0]=[1, 2, 3, 'numbers']
x1[1]={'1': 'one', '2': 'two', '3': 'three'}
```

3.2.1.2 Eine Instanz einer Klasse in JSON speichern

Es gilt wieder, dass man die Daten, also nun Instanzen einer Klasse, als Dictonary darstellen muss. Erst dann können sie mit `json.dump` als JSON serialisiert gespeichert werden.
Um Objekte in ein Dictonary-Format umzuwandeln, verwendet man wieder einmal ein *Magic Attribute*: `__dict__`
Für jede Instanz einer Klasse legt Python ein Dictonary an, welches die Parameter der Instanz speichert.
json-02.py

```
'''
Objekte in json speichern
'''
import json

class Student:
    def __init__(self,matr,vorname,nachname):
        self.matr = matr
        self.vorname = vorname
        self.nachname = nachname

s1 = Student(1,"Max","Muster")
s2 = Student(2,"Sepp","Sagmeister")

alle_studenten = [s1,s2]

#alle studenten ins dictionary format umwandeln
alle_studenten_dict = [s.__dict__ for s in alle_studenten]

with open("json_objects.json","w") as f:
    json.dump(alle_studenten_dict,f,indent=4)
```

`cat json_objects.json` liefert mit
`json.dump(alle_studenten_dict,f, indent=4)`
folgenden output:

```
[
    {
        "matr": 1,
        "vorname": "Max",
        "nachname": "Muster"
    },
    {
        "matr": 2,
        "vorname": "Sepp",
        "nachname": "Sagmeister"
    }
]
```

`cat json_objects.json` liefert mit
`json.dump(alle_studenten_dict,f, indent=2,separators=(", ","="))`
folgenden output:

```
[
  {
    "matr"=1,
    "vorname"="Max",
    "nachname"="Muster"
  },
  {
    "matr"=2,
    "vorname"="Sepp",
    "nachname"="Sagmeister"
  }
]
```

Um das Json-File wieder in ein Python Programm zu laden wird wieder `json.load()` verwendete, dass als Parameter eine File-Object benötigt:

```
x = None
with open("json_objects.json","r") as f:
    x = json.load(f)

print(x)
```

liefert den Output

```
[{'matr': 1, 'vorname': 'Max', 'nachname': 'Muster'},
 {'matr': 2, 'vorname': 'Sepp', 'nachname': 'Sagmeister'}]
```

Man bekommt also die gleiche Liste (welche die Objekte als Dictionary dargestellt hat) wieder zurück. Nun muss man sich noch gedanken machen, wie man aus den Dictionaries wieder Objekte der Klasse Student machen kann!
Dazu gibt es einige Möglichkeiten:

- types.SimpleNamespace und object_hook
- selbstgebaute Decoder-Klassen
- jsonpickle

3.2.1.3 Jsonpickle

Jsonpickle ist eine library um komplexe Python Objekte zu serialisieren und in Json File zu schreiben. Siehe `https://jsonpickle.github.io/`
Es ist eine sehr einfache Art Objekte einer selbstgeschriebenen Klasse ins Json-Format zu bringen, und auch wieder ins Objekt-Format in einem Python Programm.
Beispiel json.03.py

```
'''
Objekte mit jsonpickle speichern und laden
'''
import json
import jsonpickle

class Student:
    def __init__(self,matr,vorname,nachname):
        self.matr = matr
        self.vorname = vorname
        self.nachname = nachname

    def __repr__(self):
        return (f"{type(self)} matr={self.matr} vorname={self.vorname}"+
                f" nachame={self.nachname}")

s1 = Student(1,"Max","Muster")
s2 = Student(2,"Sepp","Sagmeister")
print(s1)
print(s2)
print()

alle_studenten = [s1,s2]

#encode objects in jsonpickle format
students_json = [jsonpickle.encode(student) for student in alle_studenten]

#dump list of pickeld student objects
with open("json_objects.json","w") as f:
    json.dump(students_json,f,indent=2)

#load list with student objects - still pickle encoded
with open("json_objects.json","r") as f:
    all_students_encoded = json.load(f)

#alle
all_students_decoded =[jsonpickle.decode(student) for student in all_students_encoded]
s1 = all_students_decoded[0]
s2 = all_students_decoded[1]
print(s1)
print(s2)
```

liefert folgenden output:

```
<class '__main__.Student'> matr=1 vorname=Max nachame=Muster
<class '__main__.Student'> matr=2 vorname=Sepp nachame=Sagmeister

<class '__main__.Student'> matr=1 vorname=Max nachame=Muster
<class '__main__.Student'> matr=2 vorname=Sepp nachame=Sagmeister
```