

8. Klassen

Die Definition und Verwendung von Klassen ist ein ganz essentieller Baustein der *objektorientierten Programmierung (OOP)*. Auch wenn einige Programmierer die OOP für eine moderne Errungenschaft halten, wissen wir bereits, dass ihre Wurzeln bis in die 1960er-Jahre zurückgehen. Die erste Programmiersprache, die Objekte verwendete, hieß bekanntlich *SIMULA*. Die Grundideen dieser Sprache stimmen noch immer mit denen von Python überein:

1. Alles ist ein Objekt
2. Objekte kommunizieren durch das Senden und Empfangen von Nachrichten
3. Objekte haben ihren eigenen Speicherbereich
4. Jedes Objekt ist ein Exemplar einer *Klasse*
5. Die *Klasse* beschreibt das Verhalten ihrer Exemplare

Insgesamt kann man die Grundideen als eine Zusammenfassung aller Daten und Operationen zu einer Einheit beschreiben. Dieses Prinzip wird auch *Datenkapselung* genannt, welches auch zum Schutz vor dem unmittelbaren Zugriff von Daten dient.

In der Terminologie der aktuellen OOP werden anstatt der Begriffe Daten und Operationen die Begriffe *Attribute* und *Methoden* verwendet, wobei *Methode* nur eine andere Bezeichnung für den Ihnen schon bekannten Begriff *Funktion* ist. Auf eine Kurzformel gebracht:

Objekt = *Attribute* + *Methoden*

Das Konzept der OOP verfolgt insbesondere drei Ziele:

- **Prinzip der Wiederverwendbarkeit:** Einmal definierte Klassen sollen auch in anderen Softwareprojekten wiederverwendet werden können.
- **Prinzip der Datenkapselung:** Durch die Aufteilung in übersichtliche Klassen soll die Komplexität reduziert werden. Softwareprojekte werden dadurch besser beherrschbar. Außerdem hat jede Klasse seine eigenen Variablen, ohne dass es zu Seiteneffekten bei der Programmausführung kommt
- **Bessere Wartbarkeit:** Wenn z.B. für bestimmte Methoden einer Klasse ein effektiverer Algorithmus mit besserer Laufzeit gefunden wurde, kann er einfach in seiner Klasse als gleichnamige Methode implementiert werden.

Fun Fact: Sie haben seit Beginn der Vorlesung druchgehend mit Klassen, Methoden und ihren Attributen gearbeitet (ohne es vielleicht explizit bemerkt zu haben).

8.1 Klassen als Container

A container is an object that contains other objects

In vergangenen Vorlesungen haben wir *Container* auch als Datenstrukturen bezeichnet. Dort wurden z.B. Werte zu einem kompakten `dict` zusammengefasst. Um auf die gepackten Daten zuzugreifen, wurde wiederum ein *Schlüssel* benötigt.

Klassen haben eine ähnliche Struktur: Sie sind Sammlungen von Attributen, zu denen man Werte speichern kann. Welche Attribute das sind, richtet sich danach, was modelliert werden soll. In unserem Beispiel wollen wir eine einfache Geld-Klasse darstellen:

 roboterklasse

Zur visuellen Beschreibung von OOP werden häufig Symbole der UML (*Unified Modeling Language*) verwendet:

 template

8.1.1 Definition von Klassen

Der syntaktische Aufbau einer Klassendefinition in Python lautet wie folgt:

```
classdef ::= class classname [inheritance] :  
          anweisungsfolge
```

```
inheritance ::= ([expression_list])  
classname ::= identifi er
```

Beispiele f r syntaktisch korrekte Kopfzeilen von Klassendefinitionen sind:

```
class MeineKlasse:  
class MeineKlasse(object):  
class MeineKlasse(Oberklasse):  
class MeineKlasse(Oberklasse1, Oberklasse2):
```

Hinweis: Klassennamen beginnen immer mit einem Gro buchstaben!

```
In [1]: # TODO einfachste Definition der Klasse "Geld"  
class Geld:  
        pass
```

8.1.2 Definition mit Konstruktor

Obige Implementierung ist zwar syntaktisch korrekt, aber v llig sinnlos. In der Praxis besteht der K rper einer Klassendefinition aus folgenden Komponenten:

- Definition der Klassenattribute (auch *statische* Attribute genannt) (falls vorhanden)
- Definition einer Konstruktormethode `__init__()`. Dort werden Objektattribute mit Anfangswerten belegt, wenn ein Objekt der Klasse instanziiert wird.
- Definition weiterer Methode (falls vorhanden)

```
In [3]: # TODO sinnvoller Definition der Klasse "Geld"
class Geld:
    # Klassenattribut
    wechselkurs = {'USD': 0.84998,
                   'GBP': 1.39480,
                   'EUR': 1.0,
                   'JPY': 0.007168}

    # Konstruktormethode
    def __init__(self, waehrung, betrag):
        self.waehrung = waehrung
        self.betrag = float(betrag)

    def getEuro(self):
        return self.betrag*self.wechselkurs[self.waehrung]

    def add(self, geld):
        summe_in_Euro = self.getEuro()+geld.getEuro()
        summe = Geld(self.waehrung, summe_in_Euro/self.wechselkurs[self.waehrung])
        return summe
```

Hinweis: Das Schlüsselwort `self` muss bei jeder Methodendefinition **immer** mitangegeben werden. Beim Zugriff auf ein Instanzattribut wird dieses **immer** vorangestellt. Die Konstruktormethode `__init__()` darf **keine** `return`-Anweisung enthalten!

8.2 Instanzen (Objekte)

Um ein Klassenobjekt zu erzeugen, muss man die Klasse zunächst aufrufen. Klassen sind dahingehend wie Funktionen und zählen zu den aufrufbaren Objekten (*callable*s). Als Übergabeargumente müssen die in der Parameterliste der Konstruktormethode geforderten Parameter angegeben werden, wobei man das erste Argument weglässt. Ein solches, durch Aufruf einer Klasse, erzeugtes Objekt nennt man dann auch **Instanz**.

```
In [5]: # TODO Erzeugen einer Instanz der Klasse Geld
fuffi = Geld("EUR", 50.0)
print(fuffi.betrag, fuffi.waehrung)
print(type(fuffi))
```

```
50.0 EUR
<class '__main__.Geld'>
```

Generieren wir nun einige `Geld`-Objekte und überprüfen die Wirkung der Methode `add()` entsprechend folgender *UML-Objektsymbole*:

objektsymbole

```
In [8]: # TODO Testen der Methode add()
hotelrechnung = Geld("USD", 123.45)
mietwagen = Geld("EUR", 527.30)

summe = hotelrechnung.add(mietwagen)
print(summe.betrag, summe.waehrung)
```

```
summe = mietwagen.add(hotelrechnung)
print(summe.betrag, summe.waehrung)
```


```
743.817538059719 USD
632.2300309999999999 EUR
```

Es ist auch möglich, Instanzen ohne Namen zu generieren. Diese nennt man *anonyme Objekte*:

```
In [10]: # TODO Anonymes Objekt als Instanz der Klasse Geld
ausgaben = Geld("EUR", 0)
ausgaben = ausgaben.add(Geld("USD", 200.0))
print(ausgaben.betrag, ausgaben.waehrung)
```

```
169.995999999999998 EUR
```

Das entsprechende UML-Diagramm sieht wie folgt aus:

 anonymesObjekt

8.3 Zugriff auf Attribute - Sichtbarkeit

Mit Attributen beschreibt man die Merkmale einer Klasse. Die *Objektattribute* beziehen sich dabei auf Eigenschafter individueller Instanzen und werden in Python durch Zuweisungen der Form `self.attribut = wert` innerhalb des Konstruktors erzeugt.

Klassenattribute (oder auch statische Attribute) sind dagegen Merkmale, die *alle* Instanzen einer Klasse besitzen. Diese sind von der Existenz einer Instanz unabhängig und werden durch eine Zuweisung der Form `attribut = wert` innerhalb der Klassendefinition (jedoch außerhalb einer Methode) definiert.

Der Zugriff aus Klassen- und Objektattribute kann eingeschränkt werden. Man spricht dann auch von *Sichtbarkeit* und unterscheidet zwischen öffentlichen und privaten Attributen.

8.3.1 Öffentliche Attribute

Öffentlich bedeutet, dass man "von außen" lesend und schreiben zugreifen kann. Der Syntax für einen Zugriff auf ein *Objektattribut* lautet `instanz.attribut`:

```
In [11]: # TODO einfacher Zugriff aus Objektattribut
hunni = Geld("EUR", 100)
print(hunni.betrag, hunni.waehrung)
```

```
100.0 EUR
```

Analog dazu kann man auch auf *Klassenattribute* zugreifen. Hierzu schreibt man lediglich den Klassennamen, anstatt des Instanznamen: `Klasse.attribut`

```
In [12]: # TODO Zugriff auf Klassenattribut
print(Geld.wechselkurs)
Geld.wechselkurs['USD'] = 0.5
print(Geld.wechselkurs)
```

```
{'USD': 0.84998, 'GBP': 1.3948, 'EUR': 1.0, 'JPY': 0.007168}
{'USD': 0.5, 'GBP': 1.3948, 'EUR': 1.0, 'JPY': 0.007168}
```

Die Verwendung öffentlicher Attribute wird insbesondere von Informatikern aus dem Software-Engineering **abgelehnt**, weil damit das Geheimnisprinzip verletzt wird.

Ein Objekt sollte so wenig wie möglich über seinen internen Aufbau verraten. Außerdem bergen sie das Risiko zu ungewollten Seiteneffekten und/oder inkonsistenten Zuständen.

Beispiel:

```
In [15]: # Negativbeispiel öffentlicher Attribute
preis = Geld("EUR", 100)
preis.waehrung = "DM"
preis.betrag = 200
# TODO welche Probleme treten auf?
print(preis.betrag, preis.waehrung)
print(type(preis.betrag))
preis.getEuro()
```

200 DM

<class 'int'>

```
-----
KeyError                                Traceback (most recent call last)
Cell In[15], line 8
      6 print(preis.betrag, preis.waehrung)
      7 print(type(preis.betrag))
----> 8 preis.getEuro()

Cell In[3], line 14, in Geld.getEuro(self)
     13 def getEuro(self):
--> 14     return self.betrag*self.wechselkurs[self.waehrung]

KeyError: 'DM'
```

8.3.2 Private Attribute

Um Attribute vor öffentlichem Zugriff zu schützen, sollte man sie mit einem oder zwei Unterstrichen definieren. Dadurch wird das Attribut abgeschirmt und man nennt es *privat*. Python unterscheidet hierbei zwischen schwacher (*protected*) und starker (das echte *private*) Privatheit:

- Stark private Attribute schreibt man z.B. `__privat`. Es ist nur möglich, innerhalb der Klassendefinition auf ein solches Attribut zuzugreifen.
- Schwach private Attribute schreibt man z.B. `_privat`. Durch eine `from ... import *`-Anweisung wird der Namen zwar nicht in den Namensraum aufgenommen und das Risiko einer Namenskollision verringert, allerdings kann man problemlos auf das Attribut zugreifen, wenn man seinen Namen kennt.

```
In [16]: # TODO Beispiel zu verschiedenen Attributzugriffen
class C:
    def __init__(self):
        self.__privat = "privat"
        self._privat = "schwach privat"
        self.privat = "oeffentlich"
```

```
c = C()
print(c.privat)
print(c._privat)
print(c.__privat)
```

oeffentlich
schwach privat

```
-----
AttributeError                                Traceback (most recent call last)
Cell In[16], line 11
      9 print(c.privat)
     10 print(c._privat)
--> 11 print(c.__privat)

AttributeError: 'C' object has no attribute '__privat'
```

Einen wirklichen Schutz von außen bieten aber selbst *stark private* Attribute nicht, denn diese werden eigentlich nur versteckt. Man kann auch diese Attribute sichtbar machen, indem man diesem zusätzlich den Klassennamen mit einem Unterstrich voranstellt (`_Klasse__attribut`):

```
In [17]: # TODO private Attribute aufdecken
print(c._C__privat)
```

privat

Sie sehen damit, dass Python (vor allem im Unterschied zu anderen Programmiersprachen) nicht sehr *restriktiv* ist. Generell ist *Restriktivität* in der Python-Community sehr verpönt, da jeder auf die eigene Vernunft und die der anderen vertraut. Alles andere sei *unpythonic*.

We are all consenting adults here - Guido Van Rossum

8.3.3 Erweitern der Definition der Klasse `Geld`

```
In [3]: # TODO Attribute umbenennen
class Geld:
    # Klassenattribut
    __wechselkurs = {'USD': 0.84998,
                     'GBP': 1.39480,
                     'EUR': 1.0,
                     'JPY': 0.007168}

    # Konstruktormethode
    def __init__(self, waehrung, betrag):
        self.__waehrung = waehrung
        self.__betrag = float(betrag)

    def getEuro(self):
        return self.__betrag*self.__wechselkurs[self.__waehrung]

    def add(self, geld):
        summe_in_Euro = self.getEuro()+geld.getEuro()
        summe = Geld(self.__waehrung,
                     summe_in_Euro/self.__wechselkurs[self.__waehrung])
        return summe
```

```

# TODO Erweiterung der Methoden
def getWaehrung(self):
    return self.__waehrung

def getBetrag(self):
    return self.__betrag

def setBetrag(self, neuerBetrag):
    self.__betrag = float(neuerBetrag)

def setWaehrung(self, neueWaehrung):
    if neueWaehrung in self.__wechselkurs.keys():
        alt = self.__wechselkurs[self.__waehrung]
        neu = self.__wechselkurs[neueWaehrung]
        self.__betrag = alt/neu * self.__betrag
        self.__waehrung = neueWaehrung

# TODO zu 8.3.4 Properties
betrag = property(getBetrag, setBetrag)
waehrung = property(getWaehrung, setWaehrung)

```

```

In [24]: # Erzeugen von Instanzen der erweiterten Klasse
preis = Geld("USD", 1000)
preis.setWaehrung("EUR")
print(preis.getBetrag(), preis.getWaehrung())

```

849.9799999999999 EUR

8.3.4 Properties

In Python kann man private Attribute so definieren, dass man auf sie von außen *scheinbar direkt* zugreifen kann, der Zugriff aber trotzdem von speziellen, in der Klasse definierten Methoden kontrolliert wird. Dazu definiert man am Ende der Klassendefinition mithilfe der `property()`-Funktion die sog. *properties* (Eigenschaften). Als Argumente der `property()`-Methode werden zuerst eine *get*- und optional auch eine *set*-Methode übergeben, die einen Lese- und/oder Schreibzugriff auf das private Attribut ermöglicht.

Der Syntax lautet:

```

property(fget=None, fset=None, fdel=None, doc=None)

```

wobei:

- `fget` : Funktion, die den Wert des Attributs zurückgibt
- `fset` : Funktion, die es erlaubt, den Wert des Attributs zu ändern
- `fdel` : Funktion, die den Löschvorgang beschreibt
- `doc` : Docstring als `String`

```

In [5]: # Erzeugen von Instanzen der Klasse mit properties
preis = Geld("EUR", 1000)
preis.waehrung = "USD"

```

```
print(preis.betrag, preis.waehrung)
print(preis._Geld__betrag)
```

```
1176.4982705475425 USD
1176.4982705475425
```

Wenn z.B. beim Aufruf der `property()`-Methode nur das erste Argument (`fget`) angegeben wird, wird damit auch keine Schreibmethode spezifiziert. So gibt es beim Versuch eines schreibenden Zugriffs einen `AttributeError`, wie das folgende Beispiel zeigt:

```
In [28]: class Const:
        def __init__(self, x):
            self.__x = x

        def getX(self):
            return self.__x

        x = property(getX)

k = Const(100)
print(k.x)
k.x = 101
```

```
100
```

```
-----
AttributeError                                Traceback (most recent call last)
Cell In[28], line 12
     10 k = Const(100)
     11 print(k.x)
--> 12 k.x = 101

AttributeError: property 'x' of 'Const' object has no setter
```

8.3.5 Dynamische Erzeugung von Attributen

In Python ist es möglich, während des Programmlaufs neue Attribute *dynamisch* zu erzeugen:

```
In [32]: # TODO Leere Klasse mit dynamisch erzeugten Attributen
        class Leer:
            pass

leer = Leer()
leer.atomzahl = 2
leer.element = "Wasserstoff"
print(leer.atomzahl, leer.element)
```

```
2 Wasserstoff
```

Hinweis: Dieses Feature ist eine potentielle Fehlerquelle! Wenn nämlich beim schreibenden Zugriff auf ein öffentliches Attribut der Attributname versehentlich falsch geschrieben wird, gibt es **keine** Fehlermeldung. Stattdessen wird einfach ein neues Attribut hinzugefügt.

8.4 Methoden

Der Syntax zur Definition von Methoden ist dem zur Definition von Funktionen sehr ähnlich:

```
def methode (self, arg1, arg2, ...)
```

Der Unterschied zu Funktionen liegt in folgenden Punkten:

- Methoden sind **keine** selbstständigen Objekte, sondern integraler Bestandteil einer Klasse
- In der Parameterliste der Methodendefinition bezeichnet der erste Parameter **immer** die Instanz. Der Name ist dabei beliebig, jedoch verwendet man üblicherweise `self`
- Beim Aufruf einer Methode muss immer der Name der Instanz, gefolgt von einem Punkt, vor den Methodennamen gestellt werden. Syntax: `instanz.methode(arg1, ...)`. Dabei ist der Methodenaufruf um eines kürzer, da das erste Argument `self` weggelassen wird.

Für Zugriffsoperationen auf (private) Attribute verwendet man sog. *setter*- und *getter*-Methoden. In UML-Klassensymbolen lässt man diese elementaren Verwaltungsmethoden meist weg

Genauso wie bei den Attributen gibt es auch private und öffentliche Methoden. Hierfür gilt der gleiche Syntax wie bei den Attributen, z.B. wäre `__machPrivateDinge()` eine stark private Methode, `_machDasNichtMitJedem()` eine schwach private Methode. Ohne Unterstrich bezeichnen die Namen öffentliche Methoden.

8.4.1 Polymorphismus - Überladen von Operatoren

Ein weiteres wichtiges Konzept der OOP ist der *Polymorphismus* (bzw. die *Polymorphie*). Dadurch ist es möglich, den gleichen Namen für gleichartige Operationen zu verwenden, die auf Objekte unterschiedlicher Klassen angewendet werden. Man spricht auch vom Überladen (*overloading*) einer Operation.

In Python können Operatoren und Standardfunktionen überladen werden, indem man bei einer neuen Klasse bestimmte Methoden mit vorgegebenen Namen definiert. In folgenden Tabellen sind einige dieser speziellen (*magic/Dunders*) Methoden aufgelistet:

Initialisierung:

Methode	Erläuterung
<code>__init__(self, [...])</code>	Initialisiert die Instanz

Stringrepräsentation:

Methode	Erläuterung
<code>__str__(self)</code>	Liefert einen <code>String</code> , der die Instanz repräsentiert
<code>__repr__(self)</code>	Liefert eine exakte <code>String</code> -Repräsentation einer Instanz, aus der ein Objekt erzeugt werden kann

```
In [1]: # Beispiel zur Unterscheidung von str und repr
import datetime
jetzt = datetime.datetime.now()
print("print:", jetzt, type(jetzt))
print("str:", str(jetzt), type(str(jetzt)))
print("repr:", repr(jetzt), type(repr(jetzt)))
neu = eval(repr(jetzt)) # evaluiert String-Repräsentation
print("neu:", neu, type(neu))
```

print: 2024-04-25 07:42:58.078521 <class 'datetime.datetime'>
str: 2024-04-25 07:42:58.078521 <class 'str'>
repr: datetime.datetime(2024, 4, 25, 7, 42, 58, 78521) <class 'str'>
neu: 2024-04-25 07:42:58.078521 <class 'datetime.datetime'>

Vergleichsoperatoren:

Methode	Erläuterung
<code>__eq__(self, other)</code>	<code>==</code>
<code>__ge__(self, other)</code>	<code>>=</code>
<code>__gt__(self, other)</code>	<code>></code>
<code>__le__(self, other)</code>	<code><=</code>
<code>__lt__(self, other)</code>	<code><</code>
<code>__ne__(self, other)</code>	<code>!=</code>
<code>__bool__(self)</code>	gilt das Objekt als <code>True</code> oder <code>False</code> ?

Arithmetische Operatoren:

Methode	Erläuterung
<code>__add__(self, other)</code> , <code>__radd__(self, other)</code>	<code>+</code>
<code>__iadd__(self, other)</code>	<code>+=</code>
<code>__truediv__(self, other)</code> , <code>__rtruediv__(self, other)</code>	<code>/</code>
<code>__itruediv__(self, other)</code>	<code>/=</code>
<code>__floordiv__(self, other)</code> , <code>__rfloordiv__(self, other)</code>	<code>//</code>
<code>__ifloordiv__(self, other)</code>	<code>//=</code>
<code>__mul__(self, other)</code> , <code>__rmul__(self, other)</code>	<code>*</code>
<code>__imul__(self, other)</code>	<code>*=</code>

Methode	Erläuterung
<code>__sub__(self, other)</code> , <code>__rsub__(self, other)</code>	-
<code>__isub__(self, other)</code>	-=
<code>__mod__(self, other)</code> , <code>__rmod__(self, other)</code>	%
<code>__imod__(self, other)</code>	%=
<code>__pow__(self, other)</code> , <code>__rpow__(self, other)</code>	**
<code>__ipow__(self, other)</code>	**=

Bitweise Operationen:

Methode	Erläuterung
<code>__lshift__(self, other)</code>	<<
<code>__ilshift__(self, other)</code>	<<=
<code>__rshift__(self, other)</code>	>>
<code>__irshift__(self, other)</code>	>>=
<code>__and__(self, other)</code>	&
<code>__iand__(self, other)</code>	&=
<code>__xor__(self, other)</code>	^
<code>__ixor__(self, other)</code>	^=
<code>__or__(self, other)</code>	
<code>__ior__(self, other)</code>	=

Sonstige Operationen:

Methode	Erläuterung
<code>__dict__</code>	Liefert das Dictionary, das die Attribute speichert.
<code>__slots__</code>	Liste welche die Attribute einer Klasse festlegt

```
In [6]: # TODO Beispiel Überladen der __str__-Methode
class Konto(Geld):

    def __str__(self):
        reVal = f"{self.betrag :.2f} "
        reVal += self.waehrung
        reVal += "\n"
        return reVal

konto = Konto("EUR", 1000)
print(konto)
print(preis)
```

1000.00 EUR

<__main__.Geld object at 0x00000240137D0890>