Table of Contents

Grundlagen: Objektorientierte Programmierung (OOP) verstehen und anwenden in Python

Öfters liest man solche Worte wie Programmierparadigma und komplex zum Thema objektorientierte Programmierung. Davon nicht erschrecken lassen!

Objektorientierte Programmierung ist nicht schwer und macht vieles einfacher! Also einfach auf das Abenteuer einlassen. Um das sperrige Wort objektorientierte Programmierung zu vermeiden, verwende ich die geläufige Abkürzung OOP.

Am Rande bemerkt: durch das Verständnis von OOP werden auch die Python Module und die komplette Sprache sehr viel einfacher verständlich.

OOP ganz einfach: Sprechen wir mal von Dingen (um die Materie greifbar zu machen)

- Dinge haben Eigenschaften wie z.B. Größe, Farbe und Alter
- Dinge können Aktionen machen/bzw. mit diesen Dingen gemacht werden wie z.B. knurren, schmusen und schlafen

Bisher hatten wir in Python entweder Daten (was wir bei unseren Dingen mit Eigenschaften beschrieben haben) oder wir haben irgendwelche Aufgaben erledigt mit Funktionen und Methoden (in die wir zeitweise verschiedene Daten reingekippt haben).

Was aber passiert, wenn wir Daten und Methoden miteinander verknüpfen? Dann haben wir schon objektorientierte Programmierung (OOP) bzw. den Kerngedanken begriffen.

Wir trennen uns von den unspezifischen Datenstrukturen wie Variablen, Listen und Tupeln und gehen hin zu Datenstrukturen, die ein Objekt (sprich ein Ding) beschreiben.

Schauen wir uns einmal ganz konkret (m)eine Katze an. Die ist orange, fett und frisst nur Lasagne, falls sie nicht schläft und heißt Garfield. Spaß beiseite, aber es kommt mit dieser Beschreibung schon relativ gut hin. Überleg einmal, welche Eigenschaften von Katzen einem einfallen und was Katzen so machen.

Eigenschaften:

- hat eine Farbe
- hat ein Alter
- hat einen Namen
- hat 4 Beine

Wir bauen uns also ein allgemeines Bild von einer Katze – einen Bauplan. Wir spielen mit Python Gott und schaffen einen allgemeinen Katzen-Zusammenbau-Plan. Das ist unsere Katzen-Klasse.

Und nun können wir virtuelle Katzen in beliebiger Anzahl erschaffen – sprich ganz viele Objekte, die grundlegend Gleich nach dem Bauplan aufgebaut sind, aber sich in Ihren Eigenschaften (Farbe, Alter, Name) unterscheiden und in der Ausprägung der Methoden.

Methoden in der OOP

Was sind nun Methoden? Nichts anderes wie in der bisherigen Programmierung die Funktionen! Nur nennt sich Methode bei der OOP, die wir aufrufen können.

Methoden (Funktionen) einer Katze wären z.B.:

- tut fressen
- tut schlafen
- tut schmusen
- tut fauchen
- tut krallen (manchmal)

Unsere Katze kann also, sobald die Methode „fressen()“ aufgerufen wird, den Futternapf leeren (oder die Maus verspeisen).

Je nach Objekt (nicht jede Katze ist gleich) tut (sprich wird eine Methode angewendet) eine Katze spielen, schmusen oder fauchen – muss aber nicht. Prinzipiell wäre es nach der Katzenklasse möglich.

Klasse	**Objekt**
allgemeiner Bauplan: Klasse Katze	Konkretes Tier: Objekt katze_sammy
	
Eigenschaften: • Farbe • Alter • Rufname	Objekt: • fast orange • 3 • Sammy
Methoden: • miauen • schlafen • fressen • schmusen	Methoden: • miauen() • schlafen() • fressen() • schmusen()
Allgemeine Beschreibung, die Blaupause (Klassen definieren Objekte)	Objekte haben konkrete Werte

Aus der Klasse können wir noch jede Menge weitere Objekte machen. „Machen“ hört sich nicht wirklich professionell an, daher spricht man bei der OOP von Instanzen erstellen bzw. instanziieren.

Wir haben also folgende Begriffe in der OOP:

- Klassen (die Blaupause)
- Objekte (aus Klassen erstellte Instanzen)

- Instanz (nichts anderes wie ein Objekt – Lateinische Begriffe hören sich einfach hipper an – die lateinische Bedeutung ist „abgeschlossene Einheit“)
- Eigenschaft (sprich Attribute – eine Beschreibung, wie das Objekt „ist“)
- Methoden (flapsig „Funktionen“ – was das Objekt tun kann)
- Vererbung (hoppla – noch nicht beschrieben)

Ein letzter Begriff aus der OOP: Vererbung

Ähnlich wie bei einem Erbfall bekommt der Erbende etwas vom Verblichenen. Allerdings muss bei der Programmierung nichts sterben.

Bleiben wir bei unserem Katzenbeispiel. Eine Katze ist schon sehr konkret (was man spätestens beim Einsatz der Krallen spürt). Hier können wir noch einen Schritt davor machen. Wir können uns eine allgemeine Klasse „Tier“ vorstellen. So ein Tier hat wie die Katze „Farbe, Alter, Bezeichnung“ und „frisst und schläft“ normalerweise. Es ist eine allgemeine Sichtweise.

Wir können nun eine Klasse „Tier“ erzeugen, was diese Eigenschaften und Methoden hat.

Sprich unsere Klasse „Katze“ kann von der Klasse „Tier“ diese Eigenschaften und Methoden erben und die Katzen-Klasse benötigt dann nur noch die fehlenden Eigenschaften und Methoden (nicht jedes Tier kann schmusen oder hat Krallen für das Gegenteil).

Man spart sich also Programmierarbeit. Zumal wir aus der Klasse Tier auch eine Klasse Hund erstellen können. Auch der Hund hat alle Eigenschaften und Methoden von der Klasse Tier.

Um einen letzten Begriff noch einzuführen. Passt etwas bei der Vererbung nicht, dann kann man es in der Realität einfach nicht annehmen oder wegwerfen. Beim Programmieren dagegen kann man diese Eigenschaft bzw. Methode „überschreiben“. Viele Möglichkeiten die dann beim Programmieren eine konkrete Welt als binäre Welt abbilden lassen.

Das soll soweit erst einmal als grundlegendes Verständnis der OOP reichen. Diese werden in den folgenden Kapiteln noch deutlich klarer, wenn wir diese konkret an Beispielen nutzen.

In den folgenden Kapiteln schauen wir uns also an, wie wir in Python Klassen aufbauen und daraus Instanzen bilden (sprich Objekte wie die Katze Sammy erstellen).

Klassen in Python

Nun starten wir mit Python und klassenorientierter Programmierung.

Wir definieren eine Klasse. Die Definition von Klassen muss vor dem Hauptprogramm im Code stehen. Zur Erinnerung aus dem letzten Kapitel: Klassen sind Baupläne, aus denen dann später die Objekte erstellt werden.

Starten wir einfach, denn am Beispiel wird die Funktion und die Vorteile klar.

Aus didaktischen Gründen basteln wir eine Katzen-Klasse (wer Allergien gegen Katzen hat, darf wahlweise auch eine Hunde-Klasse erstellen).

Also nennen wir das Kind beim Namen. In vielen Kursen und später bei gewohntem Umgang mit Klassen und objektorientierter Programmierung wird man den Klassennamen sehr kurz und aussagekräftig halten – dann würde man etwas in die Richtung `class Katze` festlegen. Zum Lernen werde ich den etwas sperrigeren Klassennamen `BauplanKatzenKlasse` einsetzen.

Zum Festlegen einer Klasse in Python benötigen wir das Schlüsselwort `class` und darauf folgt der Namen der Klasse.

Unsere erste Zeile für unsere Klasse lautet also:

```
class BauplanKatzenKlasse():
```

Was fällt auf? Der Klassenname startet mit einem Großbuchstaben! So erkennt man bereits am Namen, dass es sich um eine Klasse handeln muss! Sind es mehrere Worte, werden diese über die UpperCamelCase-Variante zusammengeschrieben. Wer ein deutsches Wort dafür benötigt – Binnenmajuskel :) – Großbuchstaben mitten im Wort wie bei den Kamelhöckern. Leerzeichen und Unterstriche werden auf keinen Fall genutzt! (Siehe Spickzettel zum Thema Konventionen und Schreibweisen)

Nach dem Klassennamen folgt eine öffnende und schließende runde Klammer und dann der Doppelpunkt, den man gerne einmal versehentlich vergisst.

Eine kleine Beschreibung gleich nach der Klassendefinition mitzugeben, macht sehr viel Sinn. Diese Beschreibung taucht bei der Nutzung von `help(Klassenname)` wieder auf. Diese Hilfen kennt man von allen Befehlen von Python, die man aufrufen kann über `help(befehl)`.

Diese Hilfen sind auch wichtig, wenn man mit mehreren Programmierern an einem Projekt arbeitet bzw. wenn man seine Klassen weitergeben möchte oder an einem schlechten Gedächtnis leidet.

Wie sieht das im Code aus? Nach der ersten Zeile mit `class` … starten wir in der nächsten Zeile eingerückt mit 3 doppelten Anführungszeichen – das nennt sich Docstring (Dokumentationsstring). Abgeschlossen wird der Hilfetext mit 3 doppelten Anführungszeichen.

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen
    Hilfetext ideal bei mehreren Programmierern in
```

```
    einem Projekt oder bei schlechtem Gedächtnis """
```

Um uns die Hilfe einmal ausgeben lassen zu können, geben wir noch in unserem Programm die folgende `help`-Zeile ein:

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen
    Hilfetext ideal bei mehreren Programmierern in
    einem Projekt oder bei schlechtem Gedächtnis """

print(help(BauplanKatzenKlasse))
```

Wir erhalten dann als Rückmeldung:

Help on class BauplanKatzenKlasse in module __main__:

class BauplanKatzenKlasse(builtins.object)
| Klasse für das Erstellen von Katzen
| Hilfetext ideal bei mehreren Programmierern in
| einem Projekt oder bei schlechtem Gedächtnis
|
| Data descriptors defined here:
|
| __dict__
| dictionary for instance variables (if defined)
|
| __weakref__
| list of weak references to the object (if defined)

Bisher haben wir nur den Rumpf für unsere Klasse erstellt. Im folgenden Kapitel wollen wir nun die Eigenschaften festlegen.

Initialisieren der Klasse

Eigenschaften einer Klasse müssen initialisiert werden. Wir wollen also die Einführung der Benennung unserer Eigenschaften, damit wir später darauf zugreifen können.

In unserer Klasse sollen folgende Eigenschaften vorhanden sein:

Eigenschaften:

- Farbe
- Alter
- Rufname

Bisher haben wir nur unseren Rumpf unserer Klasse „BauplanKatzenKlasse“ erstellt.

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen
    Hilfetext ideal bei mehreren Programmierern in
    einem Projekt oder bei schlechtem Gedächtnis """
```

Jetzt wollen wir unsere Eigenschaften einführen. Dazu wird ein neuer eingerückter Block erstellt, der immer den gleichen Aufruf hat: `def __init__(self, …):`. Folgend für unsere Katzen-Klasse:

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen
    Hilfetext ideal bei mehreren Programmierern in
    einem Projekt oder bei schlechtem Gedächtnis """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
```

Unserer Methode `__init__` wird immer mit 2 Unterstrichen am Anfang und am Ende geschrieben. In der Klammer kommt als erstes Argument immer „self“! Hier kommt ein wichtiges Prinzip zum Tragen, dass Klassen so stark macht.

Dazu müssen wir kurz vorgreifen und uns ein Objekt erstellen. Bauen wir unsere erste Katze mit dem Namen „Sammy“, die orange ist und 3 Jahre alt.

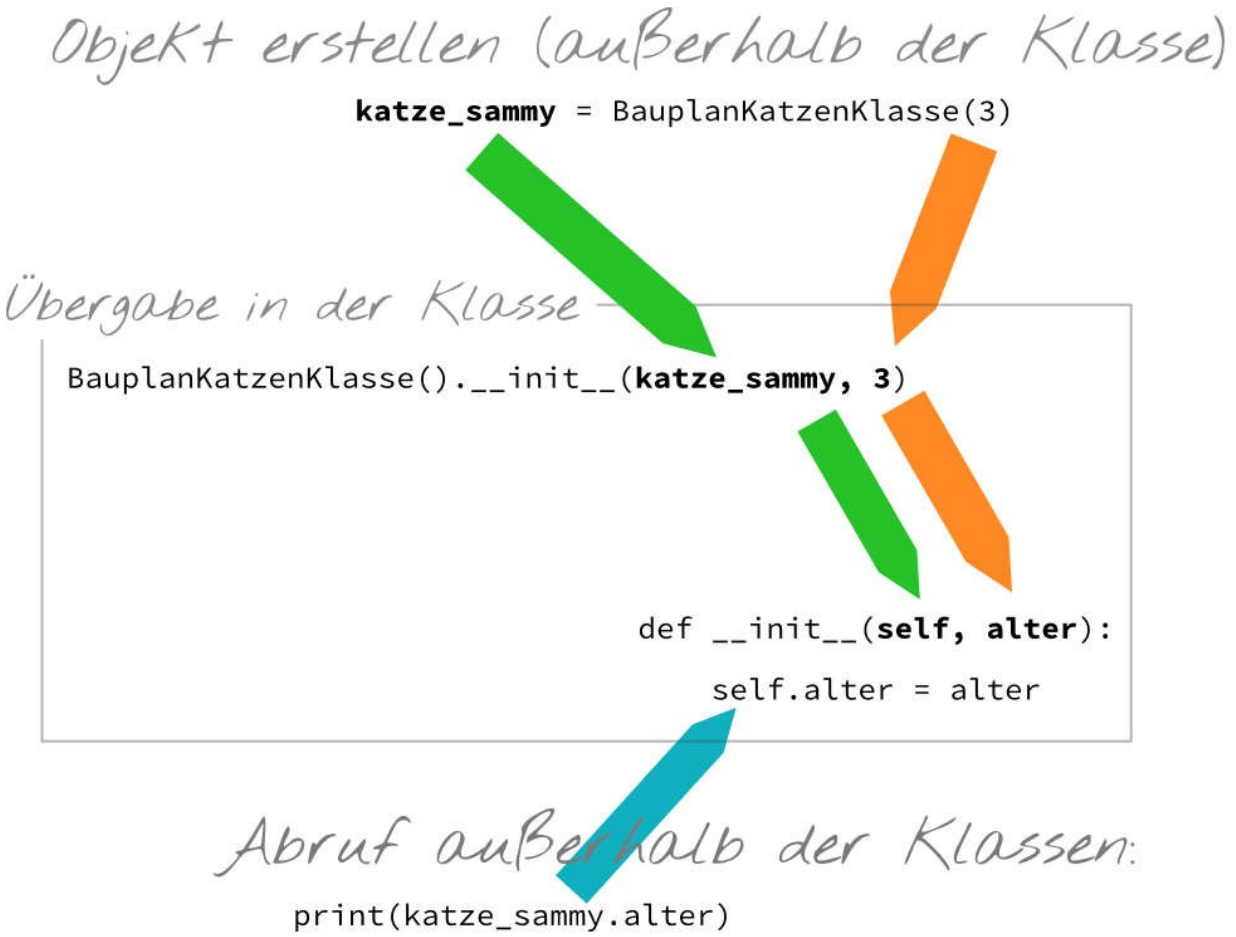
```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen
    Hilfetext ideal bei mehreren Programmierern in
    einem Projekt oder bei schlechtem Gedächtnis """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
```

`__self__` verstehen (WICHTIG!)

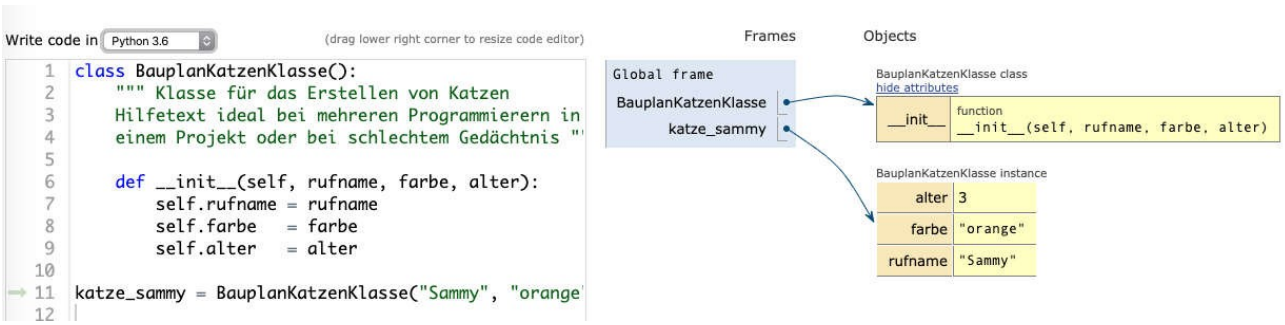
Was passiert nun da genau? Am besten ist dies an folgender Zeichnung nachzuverfolgen:

Wenn wir das Objekt „katze_sammy“ der Klasse „BauplanKatzenKlasse“ erstellen, wird der Objektname „katze_sammy“ als erstes Argument in die „__init__(self)“ übergeben. Rufen wir dann

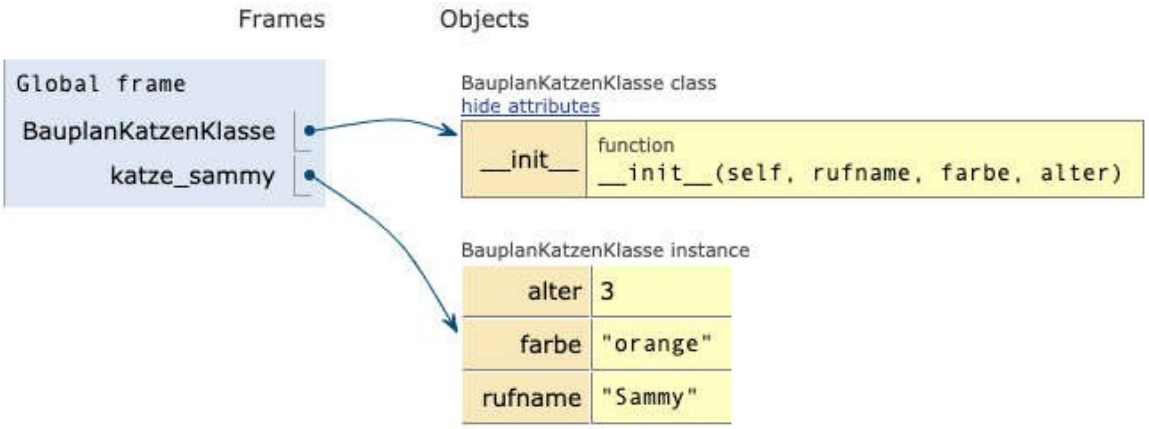
später Attribute der Klasse ab, machen wir das wieder über unseren Objektnamen „katze_sammy“, die über „self.alter“ auf den Wert von Alter zugreift.



Sehr schön sieht man auch die Auswirkungen unseren bisherigen Codes beim Live-Editor (python-online-lern-editor.htm):



und in groß:



Jetzt können wir das Alter über unsere Instanz (sprich Objekt) „katze_sammy“ abrufen:

```
katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
print(katze_sammy.alter)
```

Als Rückgabewert erhalten wir:

3

Hier sieht man schön die Gewichtigkeit von `self` sowohl beim „Befüllen“ unsere Klasse, wie beim Erzeugen eines Objekts und letztendlich auch immer beim Abruf von Attributen (wie im Beispiel das Alter oder den Rufnamen).

Zeit zum Üben!
Aufgabe: eine Klasse für Autos erstellen

Aus didaktischen Gründen basteln wir in dieser Aufgabe eine Klasse für Autos. Das liegt nicht daran, dass Autos irgendwie wichtig wären (in der aktuellen Diskussion zur Umwelt), sondern daran, dass sich jeder etwas unter Autos vorstellen kann und dies daher griffige Beispiele ergibt.

Wer mag, darf die folgenden Beispiele auch mit Fahrrädern oder was auch immer durchführen.

Erste Überlegung:

- wie benenne ich meine Klasse
- welche Eigenschaften (und später Methoden) sollen meine „Autos“ bekommen?

Bitte eine Klasse erstellen mit mindestens dem Wert „Farbe“ und ein Objekt erstellen und darüber die Farbe abrufen.

Lösung zur Übung Klasse für Auto erstellen

Die Aufgabe war, eine Klasse für Autos zu erstellen. Dabei sind die ersten Überlegungen:

- wie benenne ich meine Klasse
- welche Eigenschaften (und später Methoden) sollen meine „Autos“ bekommen?

Grundsätzlich kann man sich überlegen, ob die Benennung Auto denn so glücklich ist? Man kann bei den grundsätzlichen Überlegungen auch weiter verallgemeinern bzw. ähnliches suchen. Ähnlich wäre der Pkw (der PersonenKraftWagen oder in der Schweiz PW für PersonenWagen).

Es handelt sich um ein Fahrzeug mit eigenem Antrieb zum Personen befördern. Also kein Fahrrad, da diese keinen eigenen Antrieb haben. Weiter verallgemeinert wäre es ein Kraftfahrzeug. Wir könnten also für das Auto die Klasse „Pkw“ wählen.

```
class Pkw():
    """ Klasse für das Erstellen von Personenkraftwagen """
```

Im nächsten Schritt können wir uns überlegen, welche Eigenschaften wichtig sind. Dabei bestimmt unsere Anwendung die Auswahl der Eigenschaften. Für bestimmte Menschen ist zum Beispiel das Material des Interieurs wichtig. Wurzelholz oder kein Wurzelholz – das ist hier die Frage. Wir halten es allgemeiner und wir wollen die wichtigen Dinge als Eigenschaften. Uns interessiert neben

- der Farbe des Autos
- das Baujahr
- der aktuelle KM-Stand
- Anzahl Sitzplätze
- Marke

Also definieren wir diese Eigenschaften in unserer `__init__()`-Methode:

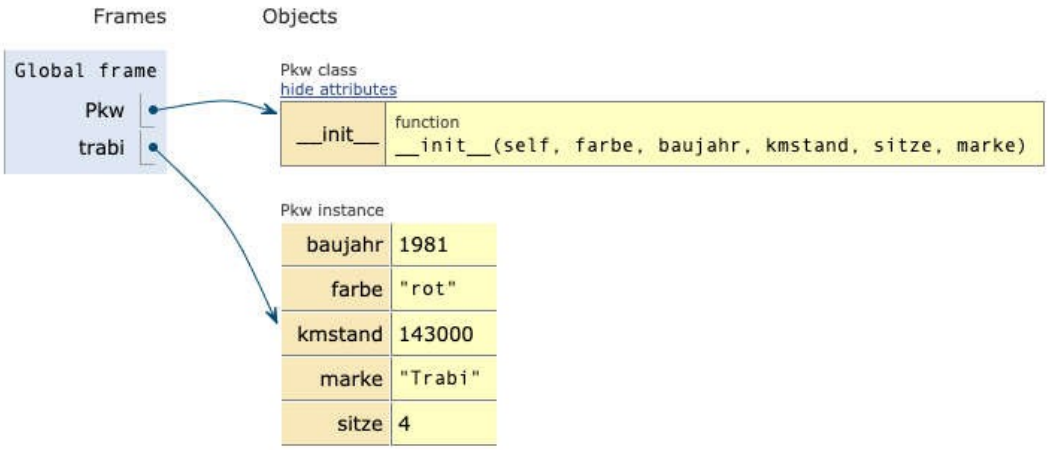
```
class Pkw():
    """ Klasse für das Erstellen von Personenkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        self.farbe   = farbe
        self.baujahr = baujahr
        self.sitze   = sitze
        self.marke   = marke
```

Aber hier ist wie gesagt die Anwendung wichtig und diese bestimmt die dafür benötigten Eigenschaften. Ganz außen vor ist gerade PS und Kraftstoffverbrauch.

Und jetzt können wir noch ein Objekt instanziieren – sprich wir basteln uns einen Trabi.

```
class Pkw():
    """ Klasse für das Erstellen von Personenkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        self.farbe   = farbe
        self.baujahr = baujahr
        self.kmstand = kmstand
        self.sitze   = sitze
        self.marke   = marke

trabi = Pkw("rot", 1981, 143000, 4, "Trabi")
```

Als Ergebnis haben wir ein Objekt, mit dem wir nun weiterarbeiten können:



Zum Abrufen der Farbe benötigen wir nach unserer erstellten Klasse mit dem Objekt „trabi“ nur den Aufruf:

```
print(trabi.farbe)
```

Instanz einer Klasse anlegen

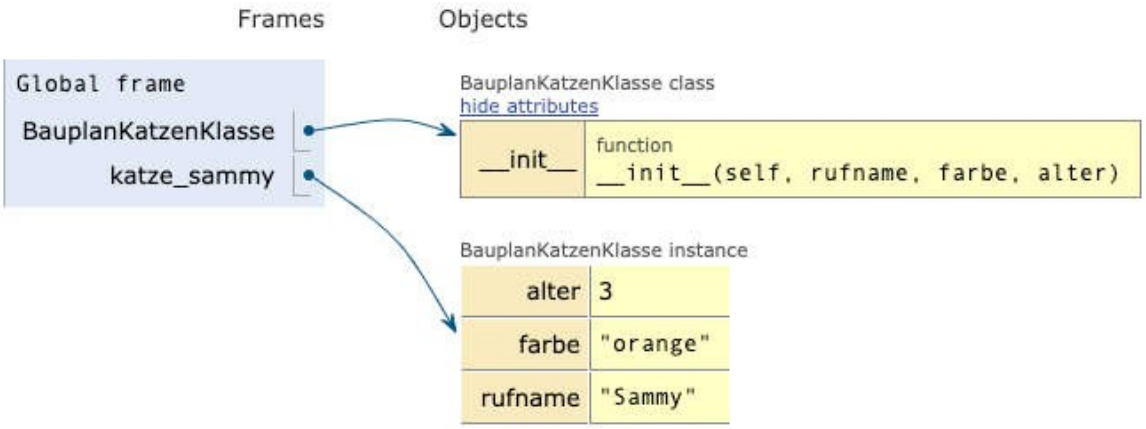
Im letzten Kapitel haben wir bereits eine Instanz unserer Klasse „BauplanKatzenKlasse“ angelegt. Hier nun die Feinheiten. Unser bisheriger Code der Klasse Katze:

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
```

Und zum Anlegen einer Instanz (sprich ein Objekt) hatten wir folgenden Aufruf:

```
katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
```

Extrem gut sieht man die internen Auswirkungen des Codes bei der Nutzung des Live-Editor (https://www.python-lernen.de/python-online-lern-editor.htm):



Wir haben die erste Instanz angelegt – unser erstes Objekt lebt.

Schauen wir uns weitere Möglichkeiten an.

Vorgabewerte für Eigenschaften festlegen

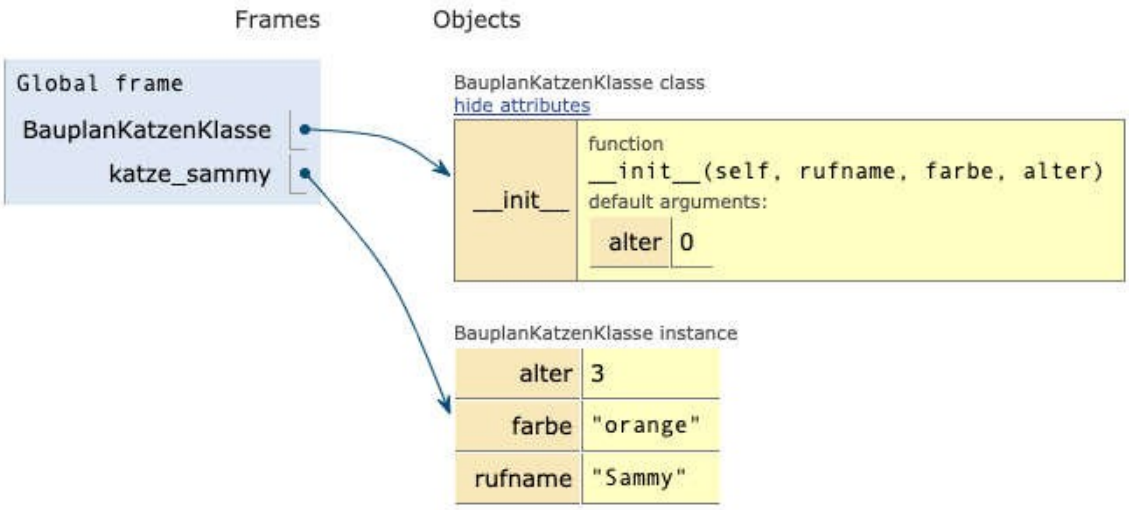
Beim Anlegen einer Klasse wäre es praktisch, auch bereits Vorgabewerte festlegen zu können. So könnte man sich ja vorstellen, dass man die Katzen immer „frisch geschlüpft“ bekommt. Somit hätten die als Alter einfach 0.

Genau das wollen wir zum Testen für das Alter als Vorgabewert hinterlegen.

Hier kommt wieder unsere `__init__()`-Methode zum Tragen. Wir können Eigenschaften mit Vorgabewert erweitern. Das kann man, muss aber nicht. Im Beispiel machen wir es nur für die Eigenschaft „alter“:

```
def __init__(self, rufname, farbe, alter=0):
```

Intern im Speicher ist nun beim Alter die 0 als Standard-Argument hinterlegt:



Erstellen wir unsere Instanz (Objekt) mit einer Angabe vom Alter, wird dieses genommen. Diese Angabe hat immer Vorrang.

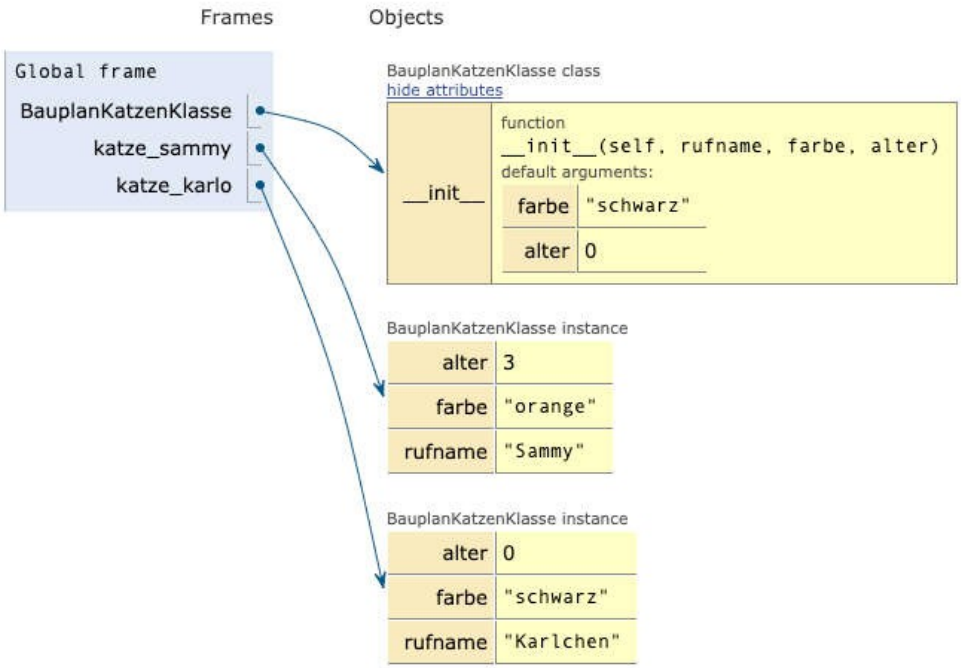
Tragen wir allerdings beim Initialisieren nichts ein, dann kommt der Vorgabewert zum Zug. So könnten wir auch noch festlegen, dass die meisten Katzen schwarz sind (das weiß man doch, wenn man dem Sprichwort glaub).

```
def __init__(self, rufname, farbe="schwarz", alter=0):
```

Erstellen wir nun ein neues Objekt „katze_karlo“ nur mit seinem Rufnamen, erhält dieser die Vorgabewerte zugewiesen:

```
katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
katze_karlo = BauplanKatzenKlasse("Karlchen")
```

Wir haben nun 2 Katzen:



Und natürlich sind genau deshalb in dieser Sekunde irgendwo auf der Welt (und natürlich in Ihrem Computer) zwei Katzen entstanden. „High five“ sage ich da nur!

Wichtig bei Vorgabewerten in Klassen

Sehr wichtig bei der Vergabe von Vorgabewerte ist die Platzierung innerhalb von `__init__`. Unsere Vorgabewerte müssen immer am Ende kommen!

Haben wir beispielsweise nur die Farbe als Vorgabewert und nicht das Alter, kommt eine Fehlermeldung. Folgender Code geht schief!

```
def __init__(self, rufname, farbe="schwarz", alter):
```

Als Fehlermeldung erhalten wir dann: „SyntaxError: non-default argument follows default argument“

Methoden bei Klassen erstellen und aufrufen bei Python

In unserer Katzen-Klasse haben wir bisher nur Eigenschaften und nur die Methode `__init__()`.

Bei der Festlegung, welche Methoden für unsere `BauplanKatzenKlasse` hatten wir notiert:

Eigenschaften:

- Farbe
- Alter
- Rufname

Methoden:

- miauen
- schlafen
- fressen
- schmusen

Also integrieren wir als Methode „miauen“. Eigentlich müssten wir für die Lautsprache von Katzen 10 weitere unterschiedliche Lautarten als Methode integrieren (man sollte seiner Katze mal genauer zuhören – im Gegensatz zum Hund, der nur 10 kann, kann die Katze bis 100 Einzellaute). Aber wir wollen nicht übertreiben. Unsere digitale Katze darf nur „miauen“ als Methode.

Unser bisheriger Code:

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
```

Wir bauen einen weiteren Bereich (Einrückung nicht vergessen!) mit `def tut_miauen()` ein. Auch hier ist extrem wichtig, dass „self“ als erstes Argument nicht zu vergessen! Sonst erhält man die Fehlermeldung „TypeError: tut_miauen() takes 0 positional arguments but 1 was given“

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter

    def tut_miauen(self):
        print("miau")

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
```

Um nun die Methode „tut_miauen()“ zu nutzen, wird nach dem Erzeugen des Objektes diese mit Objektnamen und Methode (verknüpft mit einem Punkt) und danach Klammern aufgerufen:

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter

    def tut_miauen(self):
        print("miau")

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
katze_sammy.tut_miauen()
```

Auf dem Bildschirm erhalten wir als Ausgabe:

miau

Werte beim Methodenaufruf übergeben

Wir wollen nun die Anzahl des Miauens noch mitgeben können bzw. wenn wir keine Anzahl mitgeben, soll nur ein „Miau“ kommen. Wie bereits von den Eigenschaften können wir auch bei den Methoden Vorgabewerte mitgeben. In diesem Fall soll 1 die Vorgabe sein:

```
    def tut_miauen(self, anzahl = 1):
        print(anzahl * "miau ")
```

Rufen wir nun die Methode auf, können wir die Anzahl übergeben:

```
katze_sammy.tut_miauen(3)
```

Es erscheinen ordnungsgemäß die 3 „Miaus“:

miau miau miau

beliebig viele Methoden in einer Klasse anlegen

Wir sind in der Anzahl der Methoden nicht begrenzt. Im folgenden Beispiel integrieren wir noch schlafen und die Anzahl der Minuten des Schlafens:

Das ist dasselbe, wie wir schon bei der Methode „tut_miauen“ programmiert haben. Wir machen noch eine Textausgabe, wie lange geschlafen wird:

```
    def tut_schlafen(self, dauer):
        print(self.rufname, " schläft jetzt ", dauer , " Minuten ")

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
katze_sammy.tut_schlafen(3)
```

Interessant wäre doch, wie lange die Katze insgesamt über den Tag weg schläft?

Dauer aufaddieren in Methode einer Klasse

Wir wollen mal die Schlafdauer aufsummieren. Dazu benötigen wir eine weitere Eigenschaft, die wir am Anfang mit 0 initialisieren. Wir nennen die Eigenschaft „schlafdauer“:

```
    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0
```

Jetzt müssen wir unsere Methode `tut_schlafen()` noch entsprechend erweitern. Auf den Wert der Eigenschaft „schlafdauer“ können wir einfach über das `self.schlafdauer` zugreifen und diesen Wert entsprechend erhöhen. Danach lassen wir uns die bisherige Gesamtdauer ausgeben

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen """
```

```
    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0

    def tut_miauen(self, anzahl = 1):
        print(anzahl * "miau ")

    def tut_schlafen(self, dauer):
        print(self.rufname, " schläft jetzt ", dauer , " Minuten ")
        self.schlafdauer += dauer
        print(self.rufname, " Schlafdauer insgesamt: ", self.schlafdauer, "
Minuten ")

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
katze_sammy.tut_miauen(3)
katze_sammy.tut_schlafen(3)
katze_sammy.tut_schlafen(6)
```

Wir erhalten nun als Ausgabe:

miau miau miau

Sammy schläft jetzt 3 Minuten

Sammy Schlafdauer insgesamt: 3 Minuten

Sammy schläft jetzt 6 Minuten

Sammy Schlafdauer insgesamt: 9 Minuten

Der coole Trick zum live Methoden und Klassen testen

Schöner wäre, wenn wir nicht bereits im Code unsere Aufrufe für die verschiedenen Methoden machen müssten, sondern live testen könnten. Das klappt, wenn wir unser Python-Programm in der Konsole mit einem Parameter aufrufen. Der Parameter verhindert, dass unser Programm sofort beendet wird und alle weiteren Eingaben werden ausgeführt, bis wir `exit()` eingeben.

Starten Sie in der Konsole das Programm mit dem Parameter „-i“ – die exakte Beschreibung von dem Trick gibt es im Kapitel www.python-lernen.de/trick-programm-ausfuehren-und-in-konsole-debuggen.htm

```
python -i BauplanKatzenKlasse.py
```

Im Folgenden sieht man die Möglichkeiten – erster Schritt: Das bestehende Programm wird abgearbeitet und gibt folgende Ausgaben.

Axels-MacBook-Pro:Python-lernen.de$ python3 -i BauPlanKatze.py

miau miau miau

Sammy schläft jetzt 3 Minuten

Sammy Schlafdauer insgesamt: 3 Minuten

Sammy schläft jetzt 6 Minuten

Sammy Schlafdauer insgesamt: 9 Minuten

>>>

Jetzt können wir „katze_sammy“ miauen lassen – dazu geben wir einfach nach den 3 Größerzeichen den Funktionsaufruf ein:

Axels-MacBook-Pro:Python-lernen.de$ python3 -i BauPlanKatze.py

miau miau miau

Sammy schläft jetzt 3 Minuten

Sammy Schlafdauer insgesamt: 3 Minuten

Sammy schläft jetzt 6 Minuten

Sammy Schlafdauer insgesamt: 9 Minuten

>>> katze_sammy.tut_miauen(5)

miau miau miau miau miau

>>>

Und dann ein Powernap von 10 Minuten:

Axels-MacBook-Pro:Python-lernen.de$ python3 -i BauPlanKatze.py

miau miau miau

Sammy schläft jetzt 3 Minuten

Sammy Schlafdauer insgesamt: 3 Minuten

Sammy schläft jetzt 6 Minuten

Sammy Schlafdauer insgesamt: 9 Minuten

>>> katze_sammy.tut_miauen(5)

miau miau miau miau miau

>>> katze_sammy.tut_schlafen(10)

Sammy schläft jetzt 10 Minuten

Sammy Schlafdauer insgesamt: 19 Minuten

>>>

Wir können auch neue Objekte anlegen! Erwecken wir eine neue Katze zum Leben über:

```
katze_soni = BauplanKatzenKlasse("Soni", "getigert", 2)
```

Als Rückmeldung kommt nichts, da beim Anlegen einer neuen Instanz (Objektes) dies nicht von unserem Programm kommentiert wird:

Axels-MacBook-Pro:Python-lernen.de$ python3 -i BauPlanKatze.py

miau miau miau

Sammy schläft jetzt 3 Minuten

Sammy Schlafdauer insgesamt: 3 Minuten

Sammy schläft jetzt 6 Minuten

Sammy Schlafdauer insgesamt: 9 Minuten

>>> katze_sammy.tut_miauen(5)

miau miau miau miau miau

>>> katze_sammy.tut_schlafen(10)

Sammy schläft jetzt 10 Minuten

Sammy Schlafdauer insgesamt: 19 Minuten

>>> katze_soni = BauplanKatzenKlasse("Soni", "getigert", 2)

>>>

Auch „katze_soni“ darf schlafen – hier sieht man, dass die Schlafdauer für alle Katzen getrennt aufaddiert wird:

```
Axels-MacBook-Pro:Python-lernen.de$ python3 -i BauPlanKatze.py
miau miau miau
Sammy  schläft jetzt  3  Minuten
Sammy  Schlafdauer insgesamt:  3  Minuten
Sammy  schläft jetzt  6  Minuten
Sammy  Schlafdauer insgesamt:  9  Minuten
>>> katze_sammy.tut_miauen(5)
miau miau miau miau miau
>>> katze_sammy.tut_schlafen(10)
Sammy  schläft jetzt  10  Minuten
Sammy  Schlafdauer insgesamt:  19  Minuten
>>> katze_soni = BauplanKatzenKlasse("Soni", "getigert", 2)
>>> katze_soni.tut_schlafen(5)
Soni  schläft jetzt  5  Minuten
Soni  Schlafdauer insgesamt:  5  Minuten
>>>
```

Zum Beende der Testsession einfach `quit()` eingeben.

Unbedingt testen – diese Möglichkeit macht richtig Spaß und so kann man auch hartnäckige Fehler einfacher finden.

Zeit zum Üben! Methoden für unsere PKW-Klasse

Wir hatten als letzte Übung eine Klasse „pkw“ erstellt. Dieser fehlen noch Methoden. Unser Auto soll können:

- hupen (ganz wichtige Methode!)
- fahren (hier die Anzahl der KM eingeben und aufaddieren)
- parken

- Kilometerstand ausgeben

Viel Spaß beim Umsetzen. Unser bisheriger Programmcode als Beispiellösung, an dem weitergearbeitet werden kann:

```
class Pkw():
    """ Klasse für das Erstellen von Personenkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        self.farbe   = farbe
        self.baujahr = baujahr
        self.kmstand = kmstand
        self.sitze   = sitze
        self.marke   = marke

trabi = Pkw("rot", 1981, 143000, 4, "Trabi")
```

Lösung zur Aufgabe: Methoden in PKW-Klasse einbauen

Hier der fertige Code. Es wurden auch die entsprechenden Anmerkungen als Docstrings gemacht. Dadurch kann die Hilfe entsprechend ausgegeben werden. Daher bitte bis zum Ende von diesem Kapitel lesen ;).

```
class Pkw():
    """ Klasse für das Erstellen von Personenkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        """ Eigenschaften farbe, baujahr, kmstand, Sitzplätze, Marke erfassen
"""
        self.farbe   = farbe
        self.baujahr = baujahr
        self.kmstand = kmstand
        self.sitze   = sitze
        self.marke   = marke

    def hupen(self):
        """ hier sollte noch eine MP3-Datei ausgegeben werden """
        print("Trööt")

    def fahren(self, km):
        """ wie viele KM gefahren werden, was dem Tachostand aufaddiert wird """
        self.kmstand += km
        print("Ich fahre ", km, " Kilometer")
        print("Insgesamt bin ich ", self.kmstand ," gefahren")

    def parken(self):
        """ neben fahren schon das größere Problem in Städten """
        print("Ich habe eine Parkplatz gefunden")

    def kilometerstand(self):
        """ Ausgabe des KM-Standes vom Tacho """
        print("Ich habe ", str(self.kmstand) ," auf dem Tacho")

trabi = Pkw("rot", 1981, 143000, 4, "Trabi")
trabi.hupen()
```

```
trabi.kilometerstand()
trabi.fahren(5)
trabi.kilometerstand()
trabi.parken()
```

Jetzt können wir uns die Hilfe-Funktion zu dieser Klasse ausgeben lassen und erhalten die von uns eingegebene Erklärung zur Klasse und Methoden:

```
….
trabi.kilometerstand()
trabi.parken()
print(help(Pkw))
```

Und wir bekommen diese Ausgabe:

Help on class Pkw in module __main__:

class Pkw(builtins.object)
| Pkw(farbe, baujahr, kmstand, sitze, marke)
|
| Klasse für das Erstellen von Personenkraftwagen
|
| Methods defined here:
|
| __init__(self, farbe, baujahr, kmstand, sitze, marke)
| Eigenschaften farbe, baujahr, kmstand, Sitzplätze, Marke erfassen
|
| fahren(self, km)
| weiviel KM gefahren werden, was dem Tachostand aufaddiert wird
|
| hupen(self)
| hier sollte noch eine MP3-Datei ausgegeben werden
|
| kilometerstand(self)
| Ausgabe des KM-Standes vom Tacho
|
| parken(self)
| neben fahren schon das größere Problem in Städten
|
| --
| Data descriptors defined here:
|
| __dict__
| dictionary for instance variables (if defined)
|
| __weakref__
| list of weak references to the object (if defined)

Zusätzlich können wir uns noch die enthaltenen Methoden der Klasse und des erzeugten Objekts anzeigen lassen über die Python-Anweisung `dir()`. Auch das packen wir ans Ende dazu:

```
….
trabi.kilometerstand()
trabi.parken()
print(help(Pkw))

print(dir(Pkw))
print(dir(trabi))
```

Für die Klasse `dir(Pkw)` erhalten wir folgende Informationen:

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'fahren', 'hupen', 'kilometerstand', 'parken']

Es sind alle von uns definierten Methoden von fahren, hupen bis parken aufgeführt.

Schauen wir uns die Ausgabe von dem Objekt mit `dir(trabi)` an, dann haben wir zusätzlich neben den Methoden auch die Eigenschaften aufgeführt:

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'baujahr', 'fahren', 'farbe', 'hupen', 'kilometerstand', 'kmstand', 'marke', 'parken', 'sitze']

Vererbung bei Klassen in Python

Warum sollte eine Klasse etwas vererben? Gibt es auch digitales Sterben, könnte man sich fragen? Ja und Nein – aber was auf jeden Fall leidet, ist die Übersicht, wenn Klassen immer größer und größer und dadurch unübersichtlicher werden. Und daher ist wichtig, rechtzeitig eine Erbfolge aufzubauen, bevor die Übersichtlichkeit stirbt!

Was passiert bei der Vererbung von Klassen?

Die erbende Klasse übernimmt alle Eigenschaften und Methoden der beerbten Klasse!

Bleiben wir bei unserem Beispiel mit Katzen. Bisher hatte unsere Klasse Katzen folgenden Aufbau:

Klasse

Eigenschaften:

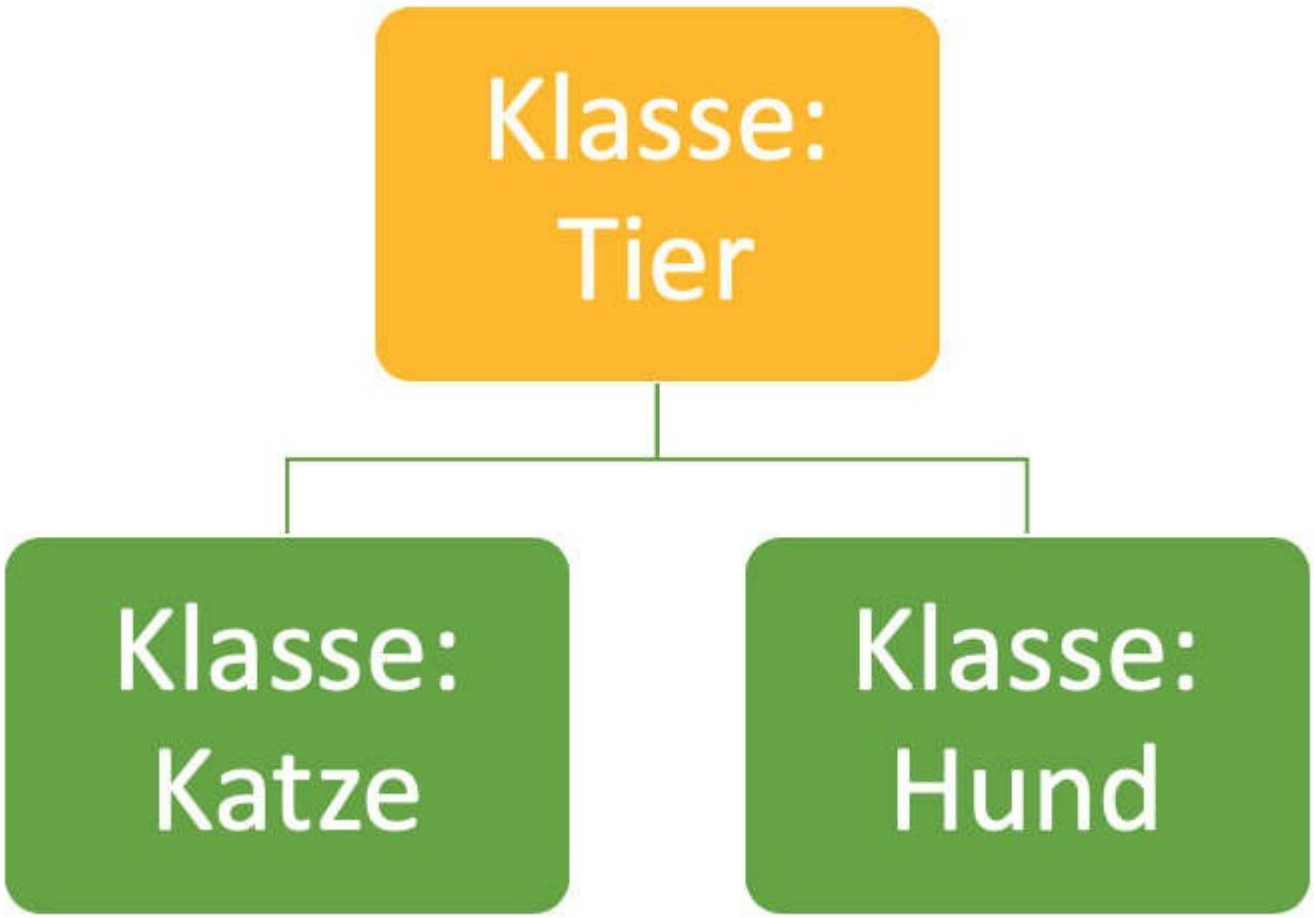
- Farbe
- Alter
- Rufname

Methoden:

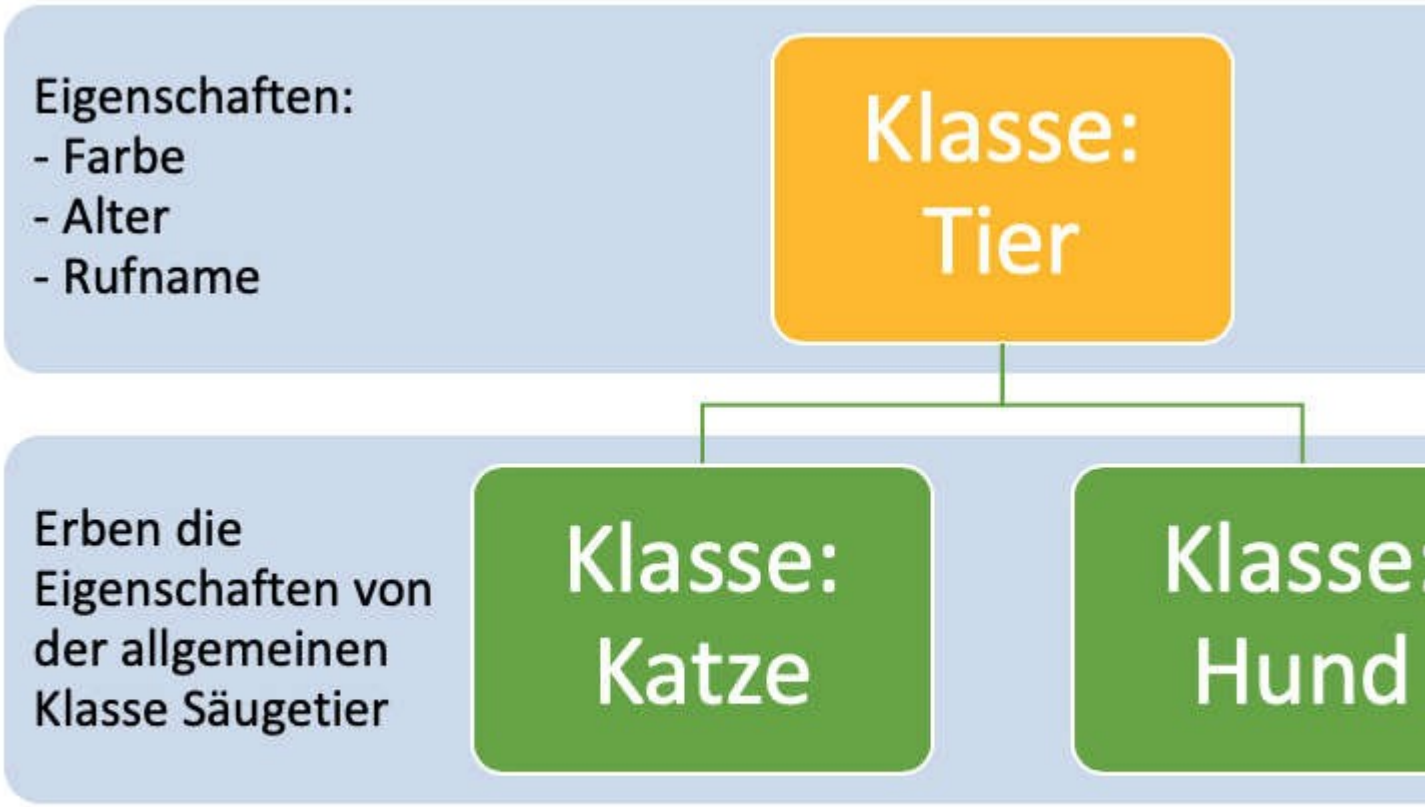
- miauen
- schlafen
- fressen
- schmusen

Jetzt wollen wir den Überbegriff zur Katze: Es handelt sich um ein Tier und genauer um ein Säugetier (was man als Mensch auch manchmal beim Milchtritt der Katze direkt abbekommt).

Um das Beispiel besser ausbauen zu können, nehmen wir noch die Konkurrenz zur Katze dazu, also Hunde. Wir haben also für beide den Überbegriff Säugetier – unsere neue „Über“-Klasse, die es zu beerben gilt!



Und nun können die Eigenschaften der Katze in die Klasse Säugetier umgebaut werden. Die Oberklasse Säugetier bekommt nun Eigenschaften der Unterklasse Katze, damit auch etwas vererbt werden kann. In der Unterklasse Katze werden die übernommenen Eigenschaften entfernt und somit haben wir eine sehr aufgeräumte Klasse was den Code sehr gut lesbar macht.



Unser bisheriger Code:

```
class BauplanKatzenKlasse():
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
print(katze_sammy.farbe)
```

Jetzt erstellen wir unsere allgemeine Klasse „Tier“, die alle Eigenschaften der Katze erhält, was die Klasse der Katze bereinigt (doppelt ist unnötig):

```
class Tier():
    """ Klasse für das Erstellen von Säugetieren """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0

class BauplanKatzenKlasse():
```

```
    """ Klasse für das Erstellen von Katzen """

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
print(katze_sammy.farbe)
```

Nach dem Starten des Python-Programms erhalten wir allerdings eine Fehlermeldung, dass unsere Katzen-Klasse keinen Konstruktor hat, der das Argument übernehmen könnte.

Bisher gibt es auch noch keine Verbindung zwischen den beiden Klassen. Schaffen wir die Verbindung!

Vererbung aktivieren zwischen 2 Klassen

Um die Vererbung und die Erbreihenfolge festzulegen, muss nur die erbende Klasse im `class`-Aufruf in Klammern die allgemeine Klasse notiert werden:

```
class BauplanKatzenKlasse(Tier):
```

Die `__init__`-Methode wird weiterhin benötigt! Allerdings wird bei der erbenden Klasse innerhalb von __init__ die Eigenschaften über `super()` aus der Eltern-Klasse (also die Klasse von der man erbt) „abgeholt“:

```
    def __init__(self, rufname, farbe, alter):
        super().__init__(rufname, farbe, alter)
```

Dieses erst einmal ominöse `super()` steht für „superclass“, sprich Oberklasse. Wir stellen damit eine Verbindung zwischen der Eltern-Klasse (Klasse, von der Kind erbt) und Kind-Klasse her.

Und nun funktioniert das auch!

Unsere Katzen-Klasse ist schlanker geworden, wobei wir noch keinen Code gespart haben, da der bisherige Code in die neue Klasse „Tier“ gewandert ist. Hier der bisher erstellte Code komplett:

```
class Tier():
    """ Klasse für das Erstellen von Säugetieren """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0

class BauplanKatzenKlasse(Tier):
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
print(katze_sammy.farbe)
```

Legen wir nun eine weitere Kind-Klasse (also noch eine Klasse, die von der Eltern-Klasse erbt) an, sieht man die Ersparnisse deutlich. Und der Code ist übersichtlicher! Bauen wir eine Klasse Hund. Nachdem wir nun schon mit Klassen umgehen können und den sperrigen Namen

„BauplanKatzenKlasse“ nicht wirklich schön ist, bekommt unsere Hunde-Klasse einen schöneren Namen, und zwar „Hund“. Zur Erinnerung: erkennbar ist die Klasse immer an der Schreibweise vom ersten Buchstaben in Großschreibung. Wir legen uns auch gleich ein neues Objekt „hund_bello“ zum Testen an, zum Testen, ob das alles funktioniert.

```
class Tier():
    """ Klasse für das Erstellen von Säugetieren """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0

class BauplanKatzenKlasse(Tier):
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)

class Hund(Tier):
    """ Klasse für das Erstellen von Hunden """

    def __init__(self, rufname, farbe, alter):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
print(katze_sammy.farbe)

hund_bello = Hund("Bello", "braun", 5)
print(hund_bello.farbe)
```

Methoden vererben in der OOP

Bisher haben wir uns nur die vererbten Eigenschaften angesehen. Genauso können wir die Methoden vererben. Als Erstes vererben wir unsere Methode „tut_schlafen()“. Schlaf benötigt jedes Tier und interessant ist bei unserer Methode, dass die Gesamtschlafdauer aufaddiert wird.

Unsere Klasse Tier wird nun über die Methode „tut_schlafen()“ erweitert.

```
class Tier():
    """ Klasse für das Erstellen von Säugetieren """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0

    def tut_schlafen(self, dauer):
        print(self.rufname, " schläft jetzt ", dauer , " Minuten ")
        self.schlafdauer += dauer
        print(self.rufname, " Schlafdauer insgesamt: ", self.schlafdauer, "
Minuten ")
```

Weder bei unserer Katzen-Klasse noch bei der Hund-Klasse müssen wir irgendetwas machen. Beide erben automatisch alle Methoden der Eltern-Klasse „Tier“, die sofort zur Verfügung stehen. Das können wir auch gleich testen über:

```
hund_bello.tut_schlafen(4)
katze_sammy.tut_schlafen(5)
hund_bello.tut_schlafen(2)
```

Der Hund schläft öfters, dass man schön sieht, dass für jede Instanz (also Bello und Sammy getrennt voneinander) automatisch ein Schlafkonto geführt wird:

```
hund_bello.tut_schlafen(4)
katze_sammy.tut_schlafen(5)
hund_bello.tut_schlafen(2)
```

Unser Ergebnis ist:

Bello schläft jetzt 4 Minuten
Bello Schlafdauer insgesamt: 4 Minuten

Sammy schläft jetzt 5 Minuten
Sammy Schlafdauer insgesamt: 5 Minuten

Bello schläft jetzt 2 Minuten
Bello Schlafdauer insgesamt: 6 Minuten

Wir integrieren noch die Methode `tut_reden()` in unsere Hauptklasse:

```
    def tut_reden(self, anzahl = 1):
        print(self.rufname, "sagt: ", anzahl * "miau ")
```

Unser kompletter Code:

```
class Tier():
    """ Klasse für das Erstellen von Säugetieren """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0

    def tut_schlafen(self, dauer):
        print(self.rufname, " schläft jetzt ", dauer , " Minuten ")
        self.schlafdauer += dauer
        print(self.rufname, " Schlafdauer insgesamt: ", self.schlafdauer, "
Minuten ")

    def tut_reden(self, anzahl = 1):
        print(self.rufname, "sagt: ", anzahl * "miau ")

class BauplanKatzenKlasse(Tier):
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)
```

```
class Hund(Tier):
    """ Klasse für das Erstellen von Hunden """

    def __init__(self, rufname, farbe, alter):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
hund_bello = Hund("Bello", "braun", 5)

katze_sammy.tut_reden(1)
hund_bello.tut_reden(3)
```

Attribute und Methoden in Klassen überschreiben

Unser Programm aus dem letzten Kapitel ist für die Katz (wortwörtlich). Der Hund zieht den Kürzeren, da er sich nicht korrekt mitteilen kann. Die geerbte Methode passt nicht!

Aber was ist mit Methoden, die so nicht passen? Unsere Methode „tut_miauen()“ mag ja noch für die Katze passen, ist aber beim Hund merkwürdig. Also bekommt die Elternklasse die Methode „tut_reden()“ (sorry, ein besserer Methodenname fällt mir gerade nicht ein). Bei der „Rede“ kommt noch als Ausgabe, wer da gerade redet:

Wir erhalten:

Sammy sagt: miau

Bello sagt: miau miau miau

Unser bisher entstandener kompletter Code:

```
class Tier():
    """ Klasse für das Erstellen von Säugetieren """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0

    def tut_schlafen(self, dauer):
        print(self.rufname, " schläft jetzt ", dauer , " Minuten ")
        self.schlafdauer += dauer
        print(self.rufname, " Schlafdauer insgesamt: ", self.schlafdauer, "
Minuten ")

    def tut_reden(self, anzahl = 1):
        print(self.rufname, "sagt: ", anzahl * "miau ")

class BauplanKatzenKlasse(Tier):
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
```

```
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)

class Hund(Tier):
    """ Klasse für das Erstellen von Hunden """

    def __init__(self, rufname, farbe, alter):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
hund_bello = Hund("Bello", "braun", 5)

katze_sammy.tut_reden(1)
hund_bello.tut_reden(3)
```

Als Ergebnis miaut nun unser Hund :(

Sammy sagt: miau

Bello sagt: miau miau miau

Das ist natürlich für den Hund frustrierend und führt langfristig zu Hundedepressionen. Dem wollen wir vorbeugen.

Methoden Überschreiben in der objektorientierten Programmierung

Wir können Methoden überschreiben. Passt eine geerbte Methode nicht, können wir diese in der Kindklasse einfach überschreiben. Unser Hund im Beispiel soll artgerecht bellen.

Also erzeugen wir in der Hund-Klasse eine Methode mit dem exakt gleichen Namen! Somit wird diese bei Aufruf ausgeführt und somit überschreibt diese die Methode der Elternklasse:

Unsere Hundeklasse:

```
class Hund(Tier):
    """ Klasse für das Erstellen von Hunden """

    def __init__(self, rufname, farbe, alter):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)

    def tut_reden(self, anzahl = 1):
        print(self.rufname, "sagt: ", anzahl * "WAU ")
```

Wird nun eine Unterhaltung zwischen Hund und Katze gehalten, läuft diese wie gewohnt ab:

```
katze_sammy.tut_reden(1)
hund_bello.tut_reden(3)
```

Mit dem Ergebnis:

Sammy sagt: miau

Bello sagt: WAU WAU WAU

Und der komplette Code:

```
class Tier():
    """ Klasse für das Erstellen von Säugetieren """

    def __init__(self, rufname, farbe, alter):
        self.rufname = rufname
        self.farbe   = farbe
        self.alter   = alter
        self.schlafdauer = 0

    def tut_schlafen(self, dauer):
        print(self.rufname, " schläft jetzt ", dauer , " Minuten ")
        self.schlafdauer += dauer
        print(self.rufname, " Schlafdauer insgesamt: ", self.schlafdauer, "
Minuten ")

    def tut_reden(self, anzahl = 1):
        print(self.rufname, "sagt: ", anzahl * "miau ")

class BauplanKatzenKlasse(Tier):
    """ Klasse für das Erstellen von Katzen """

    def __init__(self, rufname, farbe, alter):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)

class Hund(Tier):
    """ Klasse für das Erstellen von Hunden """

    def __init__(self, rufname, farbe, alter):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname, farbe, alter)

    def tut_reden(self, anzahl = 1):
        print(self.rufname, "sagt: ", anzahl * "WAU ")

katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
hund_bello = Hund("Bello", "braun", 5)

katze_sammy.tut_reden(1)
hund_bello.tut_reden(3)
```

Zeit zum Üben: Vererbung beim Auto und Methoden überschreiben

Unsere Klasse „Pkw“ aus der letzten Übung soll eine weitere Geschwisterklasse (nicht von der man erbt) bekommen, den „Lkw“. Alle bisher vorhandenen Eigenschaften und Methoden sollen in die Elternklasse „Fahrzeug“ verlagert werden.

Das Auto soll zusätzlich die Eigenschaft `kofferraumvolumen` bekommen.

Beim Lkw soll die Methode „parken()“ überschrieben werden. Es soll als Ausgabe kommen „auf Firmenhof abgestellt“.

Der Lkw soll eine zusätzliche Methode bekommen: aufladen()

Unser bisheriger Code aus der letzten Lösung, der jetzt erweitert werden soll.

```
class Pkw():
    """ Klasse für das Erstellen von Personenkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        """ Eigenschaften farbe, baujahr, kmstand, Sitzplätze, Marke erfassen
"""
        self.farbe   = farbe
        self.baujahr = baujahr
        self.kmstand = kmstand
        self.sitze   = sitze
        self.marke   = marke

    def hupen(self):
        """ hier sollte noch eine MP3-Datei ausgegeben werden """
        print("Tröööt")

    def fahren(self, km):
        """ wie viele KM gefahren werden, was dem Tachostand aufaddiert wird """
        self.kmstand += km
        print("Ich fahre ", km, " Kilometer")
        print("Insgesamt bin ich ", self.kmstand ," gefahren")

    def parken(self):
        """ neben fahren schon das größere Problem in Städten """
        print("Ich habe eine Parkplatz gefunden")

    def kilometerstand(self):
        """ Ausgabe des KM-Standes vom Tacho """
        print("Ich habe ", str(self.kmstand) ," auf dem Tacho")

trabi = Pkw("rot", 1981, 143000, 4, "Trabi")
trabi.hupen()
trabi.kilometerstand()
trabi.fahren(5)
```

Viel Spaß beim Umbauen.

Lösung Aufgabe Vererbung

In der Aufgabe zum Üben (und Testen) von Vererbung von Elternklassen und Überschreiben von Methoden sollten wir zu unserer Klasse „Pkw“ eine weitere Geschwisterklasse erstellen, den „Lkw“. Alle bisher vorhandenen Eigenschaften und Methoden sollen in die Elternklasse „Fahrzeug“ verlagert werden.

Beim Auto zusätzlich Eigenschaft: Kofferraumvolumen

Methode „parken()“ beim Lkw überschrieben mit Ausgabe: „auf Firmenhof abgestellt“.

Der Lkw soll eine zusätzliche Methode bekommen: aufladen()

Sortieren wir die Punkte, in welcher Reihenfolge diese am meisten Sinn ergeben:

1. Elternklasse „Fahrzeug“ erstellen
2. alle Eigenschaften und Methoden verlagern
3. Klasse Auto mit Eigenschaft Kofferraumvolumen erweitern

4. Klasse Lkw erzeugen
5. Lkw-Methode parken() überschreiben
6. Lkw-Methode aufladen() erzeugen

Gehen wir es an:

Punkt 1: Elternklasse „Fahrzeug“ erstellen

Das Erstellen einer neuen Klasse ist nicht wirklich ein Thema:

```
class Fahrzeug():
    """ Klasse für das Erstellen von Fahrzeugen """
```

Punkt 2: alle Eigenschaften und Methoden verlagern

Nach dem Verlagern müssen wir an die Vererbung denken (`class Pkw(Fahrzeug):`) und die __init__ in der Kindklasse entsprechend mit super() ausstatten.

```
class Pkw(Fahrzeug):
    """ Klasse für das Erstellen von Personenkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        super().__init__(farbe, baujahr, kmstand, sitze, marke)
```

Der gesamte Code (auch zum Testen, denn an der Ausgabe hat sich nichts geändert – aber funktionieren sollte es)

```
class Fahrzeug():
    """ Klasse für das Erstellen von Fahrzeugen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        """ Eigenschaften farbe, baujahr, kmstand, Sitzplätze, Marke erfassen
"""
        self.farbe   = farbe
        self.baujahr = baujahr
        self.kmstand = kmstand
        self.sitze   = sitze
        self.marke   = marke

    def hupen(self):
        """ hier sollte noch eine MP3-Datei ausgegeben werden """
        print("Trööt")

    def fahren(self, km):
        """ wie viele KM gefahren werden, was dem Tachostand aufaddiert wird """
        self.kmstand += km
        print("Ich fahre ", km, " Kilometer")
        print("Insgesamt bin ich ", self.kmstand ," gefahren")

    def parken(self):
        """ neben fahren schon das größere Problem in Städten """
        print("Ich habe eine Parkplatz gefunden")

    def kilometerstand(self):
        """ Ausgabe des KM-Standes vom Tacho """
        print("Ich habe ", str(self.kmstand) ," auf dem Tacho")
```

```
class Pkw(Fahrzeug):
    """ Klasse für das Erstellen von Personenkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        super().__init__(farbe, baujahr, kmstand, sitze, marke)

trabi = Pkw("rot", 1981, 143000, 4, "Trabi")
trabi.hupen()
trabi.kilometerstand()
trabi.fahren(5)
```

Punkt 3: Klasse Auto mit Eigenschaft Kofferraumvolumen erweitern

Wir können auch komplett eigenständige Eigenschaften in einer Kindklasse haben, die nicht in der Elternklasse vorkommen. Das sieht man schön am Kofferraumvolumen.

Diese müssen wir wie gewohnt in der `__init__` übertragen:

```
    def __init__(self, farbe, baujahr, kmstand, sitze, marke,
kofferraumvolumen):
```

Und natürlich beim Erzeugen des Objekts nicht vergessen:

```
trabi = Pkw("rot", 1981, 143000, 4, "Trabi", 20)
```

Und nun noch in Eigenschaft (`self` nicht vergessen) speichern, damit wir darauf zugreifen können:

```
class Pkw(Fahrzeug):
    """ Klasse für das Erstellen von Personenkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke,
kofferraumvolumen):
        super().__init__(farbe, baujahr, kmstand, sitze, marke)
        self.kofferraumvolumen = kofferraumvolumen
```

Jetzt können wir diese Eigenschaft abfragen

```
trabi = Pkw("rot", 1981, 143000, 4, "Trabi", 20)
print(trabi.kofferraumvolumen)
```

Eine von der Elternklasse unabhängige Eigenschaft. Auch andere Geschwisterklassen wissen von der Eigenschaft nichts, denn diese erben nur von der Elternklasse!

Punkt 4: Klasse Lkw erzeugen

Hier kommt bisher keine große Überraschung:

```
class Lkw(Fahrzeug):
    """ Klasse für das Erstellen von Lastkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        super().__init__(farbe, baujahr, kmstand, sitze, marke)
```

Und nun ein Lkw-Objekt erstellen:

```
lkw_plattnase = Lkw("orange", 1999, 50000, 3, "MAN")
print(lkw_plattnase.farbe)
```

Am Rande bemerkt: ein klasse Artikel über „Plattnasen“ und „Langhauber“: https://www.spiegel.de/auto/fahrkultur/warum-lastwagen-in-deutschland-und-den-usa-unterschiedlich-aussehen-a-855319.html

Punkt 5: Lkw-Methode parken() überschreiben

Und nun bekommt unsere Klasse Lkw die Methode „parken()“. Nachdem diese sich exakt gleich schreibt, wie die in der Elternklasse „Fahrzeug“ überschreibt diese die ursprüngliche Methode.

```
class Lkw(Fahrzeug):
    """ Klasse für das Erstellen von Lastkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        super().__init__(farbe, baujahr, kmstand, sitze, marke)

    def parken(self):
        print("auf Firmenhof abgestellt")

lkw_plattnase = Lkw("orange", 1999, 50000, 3, "MAN")
print(lkw_plattnase.farbe)
lkw_plattnase.parken()
```

Punkt 6: Lkw-Methode aufladen() erzeugen

Und nun noch eine zusätzliche Methode für den Lkw:

```
class Lkw(Fahrzeug):
    """ Klasse für das Erstellen von Lastkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        super().__init__(farbe, baujahr, kmstand, sitze, marke)

    def parken(self):
        print("auf Firmenhof abgestellt")

    def aufladen(self):
        print("habe fertig geladen")

lkw_plattnase = Lkw("orange", 1999, 50000, 3, "MAN")
print(lkw_plattnase.farbe)
lkw_plattnase.parken()
lkw_plattnase.aufladen()
```

Und unser kompletter Code:

```
class Fahrzeug():
    """ Klasse für das Erstellen von Fahrzeugen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        """ Eigenschaften farbe, baujahr, kmstand, Sitzplätze, Marke erfassen
"""
        self.farbe   = farbe
        self.baujahr = baujahr
        self.kmstand = kmstand
```

```
        self.sitze   = sitze
        self.marke   = marke

    def hupen(self):
        """ hier sollte noch eine MP3-Datei ausgegeben werden """
        print("Trööt")

    def fahren(self, km):
        """ wie viele KM gefahren werden, was dem Tachostand aufaddiert wird """
        self.kmstand += km
        print("Ich fahre ", km, " Kilometer")
        print("Insgesamt bin ich ", self.kmstand ," gefahren")

    def parken(self):
        """ neben fahren schon das größere Problem in Städten """
        print("Ich habe eine Parkplatz gefunden")

    def kilometerstand(self):
        """ Ausgabe des KM-Standes vom Tacho """
        print("Ich habe ", str(self.kmstand) ," auf dem Tacho")

class Pkw(Fahrzeug):
    """ Klasse für das Erstellen von Personenkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke,
kofferraumvolumen):
        super().__init__(farbe, baujahr, kmstand, sitze, marke)
        self.kofferraumvolumen = kofferraumvolumen

class Lkw(Fahrzeug):
    """ Klasse für das Erstellen von Lastkraftwagen """

    def __init__(self, farbe, baujahr, kmstand, sitze, marke):
        super().__init__(farbe, baujahr, kmstand, sitze, marke)

    def parken(self):
        print("auf Firmenhof abgestellt")

    def aufladen(self):
        print("habe fertig geladen")

lkw_plattnase = Lkw("orange", 1999, 50000, 3, "MAN")
print(lkw_plattnase.farbe)
lkw_plattnase.parken()
lkw_plattnase.aufladen()

trabi = Pkw("rot", 1981, 143000, 4, "Trabi", 20)
print(trabi.kofferraumvolumen)
trabi.hupen()
trabi.kilometerstand()
trabi.fahren(5)
```

Vererbung weiterdenken! Erben von Listen etc.

Wir haben bisher fleißig aus von uns erstellten Klassen geerbt. Aber Vererbung in Python kann man noch weiterdenken! Die daraus entstehenden Möglichkeiten wird man erst nach und nach erfassen (wirklich).

Hier einfach mal zum Kennenlernen. Aber Schritt für Schritt!

Der große Vorteil von objektorientierter Programmierung haben wir am Anfang vom Kapitel gesehen, dass wir benötigte Datenstrukturen passend zu unserem Objekt (was Realität abbildet) zusammenbauen können.

Einen Schritt zurück. Schauen wir uns den Datentyp Liste an.

```
datenliste = ["Hans", "Elke", "Sonja", "Kai"]
print(dir(datenliste))
```

Über die Anweisung `dir()` sehen wir die verfügbaren Methoden (es handelt sich bei Listen auch im Objekte, was man auch in der Anwendung sieht).

Das Ergebnis ist:

['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

Wir können zum Beispiel die Daten der Liste auslesen über die Methode `pop()`:

```
datenliste = ["Hans", "Elke", "Sonja", "Kai"]
print(datenliste)
print(datenliste.pop())
print(datenliste)
```

Die Anweisung `pop()` liest das letzte Element aus und wirft es aus der Liste:

['Hans', 'Elke', 'Sonja', 'Kai']

Kai

['Hans', 'Elke', 'Sonja']

Schauen wir uns eine Klasse an mit einer ähnlichen Datenstruktur. Wir basteln uns einen sportiven Jogger, der durch die Welt läuft und gelaufene Zeiten sammelt. Hier haben wir in unserer Klasse eine Liste mit dem Namen `gelaufene_zeiten` und einer Methode `zeiterfassung()`.

```
class Jogger():
    """ sportliche Klasse für das Verwalten von gelaufenen Zeiten """

    def __init__(self, altersklasse, zeit=[]):
```

```
        self.altersklasse = altersklasse
        self.gelaufene_zeiten  = zeit

    def zeiterfassen(self, zeiten):
        self.gelaufene_zeiten += zeiten
```

Wir sehen bei der `__init__`, dass unsere übergebene Zeit eine Liste ist (die eckigen Klammern verraten dies).

Jetzt erstellen wir die Jogger-Instanz – unser Objekt heißt „Laeufer_Hans“

```
class Jogger():
    """ sportliche Klasse für das Verwalten von gelaufenen Zeiten """

    def __init__(self, altersklasse, zeit=[]):
        self.altersklasse = altersklasse
        self.gelaufene_zeiten  = zeit

    def zeiterfassen(self, zeiten):
        self.gelaufene_zeiten += zeiten

Laeufer_Hans = Jogger("M40")
print(Laeufer_Hans.altersklasse)
Laeufer_Hans.zeiterfassen(["2:30"])
print(Laeufer_Hans.gelaufene_zeiten)
Laeufer_Hans.zeiterfassen(["2:40", "3:10"])
print(Laeufer_Hans.gelaufene_zeiten)
```

Als Ergebnis sehen wir die Altersklasse (das ist das Teil, damit Läufer sich untereinander vergleichen können – sprich unser Läufer ist zwischen 40 und 50 Jahre alt. Und seine gesammelten Zeiten:

```
M40
['2:30']
['2:30', '2:40', '3:10']
```

Lassen wir uns über `dir()` ausgeben, was so an Methoden und Eigenschaften unsere Klasse und unser Objekt hat:

```
print(dir(Jogger))
print(dir(Laeufer_Hans))
```

Als Ergebnis bekommen wir:

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'zeiterfassen']

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'altersklasse', 'gelaufene_zeiten', 'zeiterfassen']

Und das Ganze übersichtlicher:

Die Klasse Jogger hat:

- die Methode „zeiterfassung“

Das Objekt „Laeufer_Hans“ hat:

- die Methode „zeiterfassung“ (logisch, ist ja aus Klasse „Jogger“ entstanden)
- die Eigenschaft „altersklasse“
- die Eigenschaft „gelaufene_zeiten“

Wenn wir nun für die Eigenschaft „gelaufene_zeiten“ Methoden wie es die Datenstruktur „Liste“ anbietet gerne anwenden würden, müssten wir selber neue Methoden schreiben. Aber Faulheit siegt (zumindest, wenn man irgendwann auch mal Feierabend haben möchte).

Methoden von Datentyp „Liste“ erben

Also wollen wir die Methoden des Datentyps Liste („list“) erben. Unsere Klasse Jogger soll also ein Kindelement von „list“ werden.

```
class Jogger(list):
```

Lassen wir uns jetzt wieder die Möglichkeiten des Objekts über `dir()` ausgeben:

```
print(dir(Jogger))
```

Jetzt stehen uns neben „zeiterfassung()“ auch alle Methoden von „list” zur Verfügung:

['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__module__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort', 'zeiterfassen']

Und diese wenden wir nun auch an – unser kompletter Code:

```
class Jogger(list):
    """ sportliche Klasse für das Verwalten von gelaufenen Zeiten """

    def __init__(self, altersklasse, zeit=[]):
        self.altersklasse = altersklasse
        self.gelaufene_zeiten  = zeit

    def zeiterfassen(self, zeiten):
        self.gelaufene_zeiten += zeiten

Laeufer_Hans = Jogger("M40")
Laeufer_Hans.zeiterfassen(["2:30"])
Laeufer_Hans.zeiterfassen(["2:40", "3:10"])

# print(dir(Jogger))

print()
```

```
print("vor POP:")
print(Laeufer_Hans.gelaufene_zeiten)
print("POP:")
print(Laeufer_Hans.gelaufene_zeiten.pop())
print("nach POP:")
print(Laeufer_Hans.gelaufene_zeiten)
```

Und als Ergebnis erhalten wir wie erwartet nach der allgemeinen Methode `pop()`, die uns das Objekt „Liste“ zur Verfügung stellt:

vor POP:

['2:30', '2:40', '3:10']

POP:

3:10

nach POP:

['2:30', '2:40']

Am Rande: die Methode `pop()` würde auch so in der Klasse zu Verfügung stehen :). Hier sieht man auf jeden Fall, wie weit Klassen erben können und ich denke, das Wissen darum kann sehr hilfreich sein.

Variablen (Unterschied zu Eigenschaften) in Klassen nutzen

In der „normalen“ Programmierung spricht man viel über Variablen – was kann man nun in Klassen damit anstellen (es sind nicht die Eigenschaften gemeint!)

Unsere Oberklasse „Tier“ bekommt eine Variable gleich nach der Anlage der Klasse:

```
class Tier():
    """ Beispiel für Variablen in Klassen """
    tieranzahl = 0

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

    def anzahl_tiere(self):
        print(" aktuelle Anzahl: ", Tier.tieranzahl)
```

Zusätzlich können wir und die Anzahl der Tiere über die Methode `anzahl_tiere()` ausgeben lassen.

Unsere Kindklassen „Katze“ und „Hund“ greifen auf die Variable gleich am Anfang zu und lassen diese ausgeben:

```
class Tier():
    """ Beispiel für Variablen in Klassen """
    tieranzahl = 0
```

```
    def __init__(self, rufname):
        self.rufname = rufname

    def anzahl_tiere():
        print("aktuelle Anzahl: ", Tier.tieranzahl)

class Katze(Tier):
    """ Klasse für das Erstellen von Katzen """
    print("Aus Klasse Katze: ", Tier.tieranzahl)

    def __init__(self, rufname):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname)

class Hund(Tier):
    """ Klasse für das Erstellen von Hunden """
    print("Aus Klasse Hund: ", Tier.tieranzahl)

    def __init__(self, rufname):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname)

Tier.anzahl_tiere()
```

Als Ergebnis erhalten wir:

Aus Klasse Katze: 0

Aus Klasse Hund: 0

aktuelle Anzahl: 0

Man sieht hier sehr schön, dass der „Kopfbereich“ der Klassen auf jeden Fall ausgeführt werden. Somit erhalten wir sofort die Ausgabe „Aus Klasse Katze: 0“ und dasselbe beim Hund. Erst dann rufen wir die Methode `Tier.anzahl_tiere()` auf.

Wir könnten die Variable auf außerhalb direkt abfragen

```
print(Tier.tieranzahl)
```

Und natürlich können wir überall die Variable setzen (auch außerhalb der Klassen und die Anzahl innerhalb der Klasse würde sich ändern!). Einfach einmal probieren.

```
Tier.anzahl_tiere()
print(Tier.tieranzahl)
Tier.tieranzahl = 5
Tier.anzahl_tiere()
```

Nutzwert von Variablen in Klassen

Mit diesem Wissen können wir die Anzahl der generierten (sprich instanziierten) Objekte sehr einfach erfassen.

Jedes Mal, wenn wir ein neues Objekt erstellen, wird die `__init__` durchlaufen. Wenn wir nun in beide (sowohl bei Hund wie Katze) dieses Variable erhöhen, können wir jederzeit die Gesamtanzahl der aktuell erstellten Objekte (sprich wie viele Hunde und Katzen gibt es insgesamt) abfragen:

```
class Katze(Tier):
    """ Klasse für das Erstellen von Katzen """
    print("Aus Klasse Katze: ", Tier.tieranzahl)

    def __init__(self, rufname):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname)
        Tier.tieranzahl += 1
```

Und nun basteln wir Tiere – sprich wir erzeugen Objekte in Form von einer Katze und zwei Hunden:

```
Tier.anzahl_tiere()
katze_schnurri = Katze("Schnurri")
print(katze_schnurri.rufname)
Tier.anzahl_tiere()

hund_bello = Hund("Bello")
print(hund_bello.rufname)
Tier.anzahl_tiere()

hund_wedler = Hund("Wedler")
print(hund_wedler.rufname)
Tier.anzahl_tiere()
```

Als Ausgabe erhalten wir:

aktuelle Anzahl: 0

Schnurri

aktuelle Anzahl: 1

Bello

aktuelle Anzahl: 2

Wedler

aktuelle Anzahl: 3

Zur Sicherheit der komplette Code:

```
class Tier():
    """ Beispiel für Variablen in Klassen """
    tieranzahl = 0

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

    def anzahl_tiere():
        print("aktuelle Anzahl: ", Tier.tieranzahl)

class Katze(Tier):
    """ Klasse für das Erstellen von Katzen """
    # print("Aus Klasse Katze: ", Tier.tieranzahl)

    def __init__(self, rufname):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname)
```

```
        Tier.tieranzahl += 1

class Hund(Tier):
    """ Klasse für das Erstellen von Hunden """
    # print("Aus Klasse Hunde: ", Tier.tieranzahl)

    def __init__(self, rufname):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(rufname)
        Tier.tieranzahl += 1

Tier.anzahl_tiere()
katze_schnurri = Katze("Schnurri")
print(katze_schnurri.rufname)
Tier.anzahl_tiere()

hund_bello = Hund("Bello")
print(hund_bello.rufname)
Tier.anzahl_tiere()

hund_wedler = Hund("Wedler")
print(hund_wedler.rufname)
Tier.anzahl_tiere()
```

Eigenschaften vor Zugriff absichern

Im letzten Kapitel haben wir gesehen, wie einfach wir von außen auf Variablen (und auch Eigenschaften) zugreifen und diese beliebig ändern können.

Dies ist je nach Anwendung fatal!

Nehmen wir an, wir sind eine Bank (die fiktive Kontinentalbank) und verwalten Konten und Geld für unsere Kunden.

Also brauchen wir eine Klasse, die sowohl Konten (mit einer Kontonummer) und aktuelles Guthaben verwaltet.

```
class Konto:
    """ unsere kleines Bankprogramm zum Verwalten Konten/Geld """

    def __init__(self, kontonummer, kontostand=0):
        self.kontonummer = kontonummer
        self.kontostand  = kontostand

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

    def geld_einzahlen(self, betrag):
        self.kontostand += betrag

    def kontostand_anzeigen(self):
        print("aktueller Kontostand: ", self.kontostand)

kunde_schulz = Konto("000111555")
kunde_schulz.kontostand_anzeigen()
```

Jetzt kann der Kunde Geld einzahlen:

```
kunde_schulz = Konto("000111555")
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_einzahlen(150)
kunde_schulz.kontostand_anzeigen()
```

Bei Banken und Buchführung gibt es immer doppelte Buchführung – es darf nichts nicht nachvollziehbar sein. Wir realisieren dies in kleiner Form, in dem wir (wie im letzten Kapitel schon gezeigt) eine Bestandszahl mitführen, wie viel Geld insgesamt gerade in der Bank sein müsste und geben Einzahlungen, Auszahlungen und den Gesamtbestand aus:

```
class Konto:
    """ unsere kleines Bankprogramm zum Verwalten Konten/Geld """
    geldbestand = 0

    def __init__(self, kontonummer, kontostand=0):
        self.kontonummer = kontonummer
        self.kontostand  = kontostand

    def geld_abheben(self, betrag):
        print("Geld wird abgehoben:", betrag)
        self.kontostand -= betrag
        Konto.geldbestand -= betrag

    def geld_einzahlen(self, betrag):
        print("Geld wird eingezahlt:", betrag)
        self.kontostand += betrag
        Konto.geldbestand += betrag

    def kontostand_anzeigen(self):
        print("aktueller Kontostand: ", self.kontostand)
        print("aktueller Geldbestand der Bank: ", Konto.geldbestand, "\n")
```

Wir lassen den Kunden Schulz 2-mal einzahlen und einmal abheben:

```
kunde_schulz = Konto("000111555")
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_einzahlen(150)
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_einzahlen(250)
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_abheben(75)
kunde_schulz.kontostand_anzeigen()
```

Als Ergebnis erhalten wir:

aktueller Kontostand: 0
aktueller Geldbestand der Bank: 0

Geld wird eingezahlt: 150
aktueller Kontostand: 150
aktueller Geldbestand der Bank: 150

Geld wird eingezahlt: 250
aktueller Kontostand: 400
aktueller Geldbestand der Bank: 400

Geld wird abgehoben: 75
aktueller Kontostand: 325
aktueller Geldbestand der Bank: 325

So weit, so gut. Die Rechnung passt: 150 + 250 -75 = 325 (was auch der aktuelle Geldbestand der Bank zeigt).

Was passiert aber, wenn unser Kunde ein Schlitzohr ist? Er betätigt sich einfach eines direkten Zugriffs. Vor dem letzten Abheben erhöht er einfach einmal sein Kontostand um 1000.

```
kunde_schulz = Konto("000111555")
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_einzahlen(150)
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_einzahlen(250)
kunde_schulz.kontostand_anzeigen()
kunde_schulz.kontostand += 1000
kunde_schulz.geld_abheben(75)
kunde_schulz.kontostand_anzeigen()
```

Und schon haben wir ein massives Problem – die Kontrollzahl weicht von dem Kontostand ab:

aktueller Kontostand: 1325
aktueller Geldbestand der Bank: 325

Die Kontinentalbank leidet gewissermaßen an „Inkontinenz“, Geld verrinnt.

Das müssen wir verhindern!

Unser bisher kritischer Code komplett:

```
class Konto:
    """ unsere kleines Bankprogramm zum Verwalten Konten/Geld """
    geldbestand = 0

    def __init__(self, kontonummer, kontostand=0):
        self.kontonummer = kontonummer
        self.kontostand  = kontostand

    def geld_abheben(self, betrag):
        print("Geld wird abgehoben:", betrag)
        self.kontostand -= betrag
        Konto.geldbestand -= betrag

    def geld_einzahlen(self, betrag):
        print("Geld wird eingezahlt:", betrag)
        self.kontostand += betrag
        Konto.geldbestand += betrag

    def kontostand_anzeigen(self):
        print("aktueller Kontostand: ", self.kontostand)
        print("aktueller Geldbestand der Bank: ", Konto.geldbestand, "\n")

kunde_schulz = Konto("000111555")
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_einzahlen(150)
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_einzahlen(250)
kunde_schulz.kontostand_anzeigen()
```

```
kunde_schulz.kontostand += 1000
kunde_schulz.geld_abheben(75)
kunde_schulz.kontostand_anzeigen()
```

Eigenschaften (und auch Variablen) dürfen nicht von außen veränderbar sein. Auch hier hilft uns Python. Wir können die Sichtbarkeit von Eigenschaften und Variablen einschränken:

Sichtbarkeit von Eigenschaften/Variablen in der OOP

Wir können die „Nutzbarkeit“ von Variablen einschränken über folgende Schreibweisen:

public	standard, kann auch von Außerhalb der Klassen genutzt und geändert werden
protect	Schreibweise: _1Unterstrichamanfang kann in eigener Klasse und davon abgeleiteten Klassen verwendet werden
privat	Schreibweise: __2Unterstricheamanfang kann nur in innerhalb eigener Klasse genutzt werden

Das bedeutet, dass wir unsere Eigenschaften und Variablen als privat auszeichnen müssen.

Weil wir ein sicherheitsbewusster Programmierer sind, werden alle verwendeten Eigenschaften und Variablen als „privat“ mit den 2 Unterstrichen am Anfang gekennzeichnet.

```
class Konto:
    """ unsere kleines Bankprogramm zum Verwalten Konten/Geld """
    __geldbestand = 0

    def __init__(self, kontonummer, kontostand=0):
        self.__kontonummer = kontonummer
        self.__kontostand  = kontostand

    def geld_abheben(self, betrag):
        print("Geld wird abgehoben:", betrag)
        self.__kontostand -= betrag
        Konto.__geldbestand -= betrag

    def geld_einzahlen(self, betrag):
        print("Geld wird eingezahlt:", betrag)
        self.__kontostand += betrag
        Konto.__geldbestand += betrag

    def kontostand_anzeigen(self):
        print("aktueller Kontostand: ", self.__kontostand)
        print("aktueller Geldbestand der Bank: ", Konto.__geldbestand, "\n")
```

Ab jetzt ist weder das direkte Abfragen von Variablen/Eigenschaften noch das Setzen von Werten außerhalb der Klasse möglich. Ein Versuch endet mit einer Fehlermeldung:

```
kunde_schulz = Konto("000111555")
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_einzahlen(150)
kunde_schulz.__kontostand += 1000
```

Es kommt als Fehlermeldung: „AttributeError: 'Konto' object has no attribute '__kontostand'“

Wir können weder etwas verändern noch abfragen von außen.

Bank gerettet!

Was passiert, wenn wir mit einer Kindklasse auf die geschützten Werte zugreifen wollen?

Zugriff auf geschützte Eigenschaften über Kindklasse

Wir wollen eine Klasse erstellen, welche die Konto-Klasse beerbt. Wann kann das erforderlich sein? Seit ein paar Jahre gibt es das sogenannte „Guthabenkonto“. Diese wurde für insolvente bzw. überschuldete Menschen eingeführt und bei diesen Konten gibt es nicht die Möglichkeit der Überziehung.

Diese ist für uns natürlich die ideale Übung als Kindeklasse eines normalen Kontos.

Auszahlen finden nur statt, wenn der gewählte Betrag nicht das Konto in das Minus treiben würde.

Allerdings kommt nun als Erschwerung dazu, dass wir nicht mehr so einfach Zugriff auf alle Werte und Eigenschaften der Klasse „Konto“ haben, da diese als „__privat“ gekennzeichnet sind.

Aber einen Schritt nach dem anderen!

Wir erstellen unsere Kindklasse „Pluskonto“:

```
class Pluskonto(Konto):
    """ ein Konto, dass nicht überzogen werden kann """

    def __init__(self, kontonummer, kontostand=0):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(kontonummer, kontostand=0)

    def geld_abheben(self, betrag):
        print("Geld soll vom Pluskonto abgehoben werden:", betrag)
```

Jetzt wollen wir auf den Kontostand der Elternklasse „Konto“ zugreifen. Was bisher einfach funktionierte, geht durch die __privat gesetzten Eigenschaften nicht mehr. Aber wir benötigen ja nur den Wert um vergleichen zu können, ob das Konto ins Minus gehen würde und wollen diesen nicht verändert.

Also programmieren wir uns eine Methode in der Elternklasse, die uns diesen Wert liefert:

```
class Konto:
    """ unsere kleines Bankprogramm zum Verwalten Konten/Geld """
    __geldbestand = 0

    def __init__(self, kontonummer, kontostand=0):
        self.__kontonummer = kontonummer
        self.__kontostand  = kontostand

    def geld_abheben(self, betrag):
        print("Geld wird abgehoben:", betrag)
        self.__kontostand -= betrag
        Konto.__geldbestand -= betrag

    def geld_einzahlen(self, betrag):
        print("Geld wird eingezahlt:", betrag)
        self.__kontostand += betrag
        Konto.__geldbestand += betrag

    def kontostand_anzeigen(self):
        print("aktueller Kontostand: ", self.__kontostand)
```

```
        print("aktueller Geldbestand der Bank: ", Konto.__geldbestand, "\n")

    def kontostand_aktuell(self):
        return self.__kontostand
```

Die letzte Methode `kontostand_aktuell(self)` liefert uns den aktuellen Wert über `return` zurück:

Auf diesen können wir in der Kindklasse „Pluskonto“ zugreifen und zum Vergleich heranziehen:

```
    def geld_abheben(self, betrag):
        print("Geld soll vom Pluskonto abgehoben werden:", betrag)
        print("Maximal verfügbar ist gerade:", self.kontostand_aktuell())

        if self.kontostand_aktuell() - betrag >= 0:
            print("Auszahlen von Pluskonto: ", betrag)
        else:
            print("Sorry, Konto kann nicht überzogen werden!")
```

Jetzt müssen wir noch das Geld verbuchen!

Wir können von der Kindklasse auf die Methoden der Elternklasse über `super().geld_abheben()` zugreifen. Dies ergänzen wir innerhalb unserer if-Abfrage.

```
    def geld_abheben(self, betrag):
        print("Geld soll vom Pluskonto abgehoben werden:", betrag)
        print("Maximal verfügbar ist gerade:", self.kontostand_aktuell())

        if self.kontostand_aktuell() - betrag >= 0:
            print("Auszahlen von Pluskonto: ", betrag)
            super().geld_abheben(betrag)
        else:
            print("Sorry, Konto kann nicht überzogen werden!")
```

Der vollständige Code:

```
class Konto:
    """ unsere kleines Bankprogramm zum Verwalten Konten/Geld """
    __geldbestand = 0

    def __init__(self, kontonummer, kontostand=0):
        self.__kontonummer = kontonummer
        self.__kontostand  = kontostand

    def geld_abheben(self, betrag):
        print("Geld wird abgehoben:", betrag)
        self.__kontostand -= betrag
        Konto.__geldbestand -= betrag

    def geld_einzahlen(self, betrag):
        print("Geld wird eingezahlt:", betrag)
        self.__kontostand += betrag
        Konto.__geldbestand += betrag

    def kontostand_anzeigen(self):
        print("aktueller Kontostand: ", self.__kontostand)
        print("aktueller Geldbestand der Bank: ", Konto.__geldbestand, "\n")

    def kontostand_aktuell(self):
```

```
        return self.__kontostand

class Pluskonto(Konto):
    """ ein Konto, dass nicht überzogen werden kann """

    def __init__(self, kontonummer, kontostand=0):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(kontonummer, kontostand=0)

    def geld_abheben(self, betrag):
        print("Geld soll vom Pluskonto abgehoben werden:", betrag)
        print("Maximal verfügbar ist gerade:", self.kontostand_aktuell())

        if self.kontostand_aktuell() - betrag >= 0:
            print("Auszahlen von Pluskonto: ", betrag)
            super().geld_abheben(betrag)
        else:
            print("Sorry, Konto kann nicht überzogen werden!")

kunde_minderjaehrig = Pluskonto("0000935")
kunde_minderjaehrig.kontostand_anzeigen()
kunde_minderjaehrig.geld_einzahlen(200)
kunde_minderjaehrig.geld_abheben(101)
kunde_minderjaehrig.kontostand_anzeigen()
```

Sicherheit benötigt also ein wenig mehr Arbeit – ist aber problemlos machbar, wenn man weiß wie.

Klassen auslagern

Unsere erstellten Klassen benötigen Platz und wenn alles sich in einer Datei befindet, wird es unübersichtlich. Daher ist eine gute Vorgehensweise, die Klassen als Module auszulagern und einfach zu importieren.

Aus dem Bank-Beispiel aus dem letzten Kapitel machen wir ein Modul. Der Modulname ist „konto.py“. Die Benennung ist sehr wichtig, da wir beim Import die Datei in der Form `from konto import Konto` ohne Dateiendung „.py“ angeben!

Der Inhalt der Datei „konto.py“ – vorneweg der DOCstring für die Hilfe nicht vergessen:

```
""" Klasse Konto und Pluskonto zum verwalten, ein- und auszahlen von Bankkonten
"""

class Konto:
    """ unsere kleines Bankprogramm zum Verwalten Konten/Geld """
    __geldbestand = 0

    def __init__(self, kontonummer, kontostand=0):
        self.__kontonummer = kontonummer
        self.__kontostand  = kontostand

    def geld_abheben(self, betrag):
        print("Geld wird abgehoben:", betrag)
        self.__kontostand -= betrag
        Konto.__geldbestand -= betrag

    def geld_einzahlen(self, betrag):
        print("Geld wird eingezahlt:", betrag)
```

```
        self.__kontostand += betrag
        Konto.__geldbestand += betrag

    def kontostand_anzeigen(self):
        print("aktueller Kontostand: ", self.__kontostand)
        print("aktueller Geldbestand der Bank: ", Konto.__geldbestand, "\n")

    def kontostand_aktuell(self):
        return self.__kontostand

class Pluskonto(Konto):
    """ ein Konto, dass nicht überzogen werden kann """

    def __init__(self, kontonummer, kontostand=0):
        """ Initalisieren über Eltern-Klasse """
        super().__init__(kontonummer, kontostand=0)

    def geld_abheben(self, betrag):
        print("Geld soll vom Pluskonto abgehoben werden:", betrag)
        print("Maximal verfügbar ist gerade:", self.kontostand_aktuell())

        if self.kontostand_aktuell() - betrag >= 0:
            print("Auszahlen von Pluskonto: ", betrag)
            super().geld_abheben(betrag)
        else:
            print("Sorry, Konto kann nicht überzogen werden!")
```

Und jetzt stehen und 5 Varianten zum Import zur Verfügung:

```
# eine von den 5 wählen!
from konto import *

# oder
from konto import Konto

# oder
from konto import Pluskonto

# oder
from konto import Konto, Pluskonto

# oder
import konto
```

Warum gibt es da so viel Auswahl? Wir haben in unserem Modus sowohl die Klasse „Konto“ wie die Klasse „Pluskonto“.

from konto import Konto

Über die Anweisung `from konto import Konto` sagen wir, lade die Datei „konto.py“ und verwende die Klasse „Konto“. Die Kindklasse „Pluskonto“ steht nicht zur Verfügung und wir bekommen eine Fehlermeldung beim Aufruf der Klasse „Pluskonto“.

```
from konto import Konto

# funktioniert
kunde_minderjaehrig = Konto("0000935")
```

```
# FEHLERMELDUNG NameError: name 'Pluskonto' is not defined
kunde_minderjaehrig = Pluskonto("0000935")
```

from konto import Pluskonto

Über die Anweisung `from konto import Pluskonto` laden wir aus dem Modul „konto“ nur das Pluskonto und können auch dieses nutzen (auch wenn dieses Intern auf Konto zugreift):

```
from konto import Pluskonto

# FEHLERMELDUNG (und bricht dann ab, das Zugriff unten würde funktionieren
kunde_minderjaehrig = Konto("0000935")

# funktioniert
kunde_minderjaehrig = Pluskonto("0000935")
```

import konto

Über die Anweisung `import konto` laden wir das gesamte Modul, allerdings müssen wir mit dem Aufruf „Modulname.klassenname“ auf die Klassen zugreifen!

```
import konto

# Zugriff über "Modulname.klassenname"!
kunde_minderjaehrig = konto.Konto("0000935")

# bzw.
kunde_minderjaehrig = konto.Pluskonto("0000935")
```

from konto import *

Über die Anweisung `from konto import *` laden wir beide Modul, die wir dann nutzen können. Im Vergleich zum `import konto` (siehe Punkt davor) müssen wir keine weiteren Angaben machen!

```
from konto import *

# funktioniert
kunde_minderjaehrig = Konto("0000935")

# funktioniert
kunde_minderjaehrig = Pluskonto("0000935")
```

Macht man ungern um Namenskonflikte zu vermeiden und man sieht auch so nicht, welche Klassen eigentlich genutzt werden. Daher besser gleich folgende Variante!

from konto import Konto, Pluskonto

Über die Anweisung `from konto import Konto, Pluskonto` laden wir beide Modul, die wir dann nutzen können:

```
from konto import Konto, Pluskonto
```

```
# funktioniert
kunde_minderjaehrig = Konto("0000935")

# funktioniert
kunde_minderjaehrig = Pluskonto("0000935")
```

Fertiger Code in auszuführender Datei

Und hier nun unser fertiger Code mit der Variante für den Import beider Klassen:

```
from konto import Konto, Pluskonto

kunde_schulz = Konto("000111555")
kunde_schulz.kontostand_anzeigen()
kunde_schulz.geld_einzahlen(400)
kunde_schulz.geld_abheben(150)
kunde_schulz.kontostand_anzeigen()

kunde_minderjaehrig = Pluskonto("0000935")
kunde_minderjaehrig.kontostand_anzeigen()
kunde_minderjaehrig.geld_einzahlen(200)
kunde_minderjaehrig.geld_abheben(101)
kunde_minderjaehrig.kontostand_anzeigen()
```

Als Ausgabe erhalten wir:

aktueller Kontostand: 0
aktueller Geldbestand der Bank: 0

Geld wird eingezahlt: 400
Geld wird abgehoben: 150
aktueller Kontostand: 250
aktueller Geldbestand der Bank: 250

aktueller Kontostand: 0
aktueller Geldbestand der Bank: 250

Geld wird eingezahlt: 200
Geld soll vom Pluskonto abgehoben werden: 101
Maximal verfügbar ist gerade: 200
Auszahlen von Pluskonto: 101
Geld wird abgehoben: 101
aktueller Kontostand: 99
aktueller Geldbestand der Bank: 349

Wer nachrechnen will, das passt mit der doppelten Buchführung so :).