

7 Objekte und Klassen

Je größer ein Programm wird, desto wichtiger ist es Ordnung zu halten. Eine Konzept, um die Übersicht besser zu behalten, ist die Modularisierung. Das heißt, dass wir unser Programm in einzelne, kleinere Komponenten aufteilen. Im Kapitel zu Funktionen haben wir bereits eine mögliche Variante der Modularisierung gesehen, indem die Ausführung von gewissen Codezeilen einer Funktion übergeben wurde.

In diesem Kapitel beschäftigen wir uns mit Objekten und Klassen⁸. Die Idee dahinter ist, dass unsere Welt aus Objekten besteht, wie z.B. Personen, Autos, Bäume, Häuser, Länder, Werkzeuge und Schuhe. Jedes dieser Objekte hat bestimmte Charakteristiken und kann andere Objekte beeinflussen. Dies ist eine sehr intuitive Beschreibung des Konzepts der objektorientierten Programmierung. Im folgenden sehen wir anhand einiger Beispiele eine konkrete Umsetzung in unserer Programmiersprache.

7.1 Klassen definieren und Objekte erstellen

Gehen wir von einem Programm aus, welches Informationen über Personen verwaltet. Jede einzelne Person kann im Programm durch ein Objekt repräsentiert werden. Ein solches Objekt kann somit eine vereinfachte Kopie einer Person aus der realen Welt darstellen. Mit dem Keyword `class` kann nun eine Klasse definiert werden, welche mir Objekte von dieser Klasse generieren. Die einfachste Weise dies zu machen, sieht folgendermaßen aus:

```
1 # Einfachste Art eine Klasse zu erstellen
2 class Person:
3     pass
```

Nun können Objekte (Instanzen) der Klasse `Person` erstellt werden.

```
>>> p1 = Person()
>>> p1
<__main__.Person object at 0x33c9210>
```

Man sieht das `p1` nun eine Instanz der Klasse `Person` ist. Es könnten theoretisch noch viele weitere Instanzen unserer Klasse `Person` erstellt werden.

```
>>> e1 = Person(); e2 = Person(); e3 = Person()
>>> e1;e2;e3
<__main__.Person object at 0x7f292a1f5dd8>
<__main__.Person object at 0x7f292a1f5c50>
<__main__.Person object at 0x7f292a1f5d30>
```

Wie am obigen Beispiel ersichtlich ist, wurden drei weitere Objekte von der Klasse `Person` erstellt.

⁸Die Verwendung von Objekten und Klassen wird objektorientierte Programmierung (kurz OOP) genannt. Für eine umfassende Beschreibung siehe z.B. unter <http://openbook.galileocomputing.de/oop/> nach.

7.2 Unterschied zwischen Klassen und Objekten

Zu Beginn sind die Unterschiede zwischen Klassen und Objekten vielleicht nicht ganz klar. Die folgenden Merkmale können einem beim Weiterlesen in diesem Kapitel helfen, die beiden Begriffe Klasse und Objekt voneinander zu unterscheiden.

- Eine Klasse ist eine Konstruktionsvorlage für Objekte, d.h. es ist wie eine Art Bauplan.
- Ein Objekt hingegen ist eine konkrete Umsetzung eines solchen Bauplans.

Instanzvariablen

Jede Person ist auf seine Art einzigartig und besitzt gewisse Eigenschaften (z.B. Name, Vorname, Geburtsdatum und Körpergewicht). Diese Variablen, welche wir nur innerhalb der Person p1 definieren wollen, können wir auf folgende Art erstellen:

```
>>> p1.name = "Müller"
>>> p1.vorname = "Kurt"
>>> p1.geb_datum = "03.02.01"
>>> p1.gewicht = 73.5
```

Variablen, welche zu einem Objekt gehören, werden Instanzvariablen genannt. Möchte man nun z.B. den Namen und den Vornamen der Person p1 wissen, so kann dieser mittels p1.name (bzw. p1.vorname) abgefragt werden.

```
>>> print(p1.name, p1.vorname)
Müller Kurt
```

Eine Eingabe in der Konsole wie z.B.

```
>>> print(name, vorname)
```

wird nicht den Namen „Müller Kurt“ unserer gewünschten Person p1 ausgeben, da diese nur im Namensbereich unseres Objektes p1 existieren.

Einer Instanz kann man beliebige Variablenamen zuordnen. Jedoch sollten sie schon irgendwie Sinn machen, weil es sonst im weiteren Programmverlauf nur zur Verwirrung führt.

Bemerkung

Die Instanzvariablen, welche wir hier erstellt haben (name, vorname, geb_datum und gewicht) charakterisieren die Instanz p1 der Klasse Person. Diese helfen verschiedene Instanzen der gleichen Klasse zu unterscheiden. Später werden wir auch noch Klassenvariablen sehen, welche nicht ein Objekt, sondern die Klasse selbst beschreibt.

7.2.1 Aufgaben

1. Wir haben folgendes Programm gegeben:

```
1 class Person:
2     pass
3
4 my_person1 = Person()
5 my_person2 = Person()
6 my_person1.vorname = "Patrick"
7 my_person2.name = "vonBerg"
```

Erkläre warum folgende Eingaben nicht funktionieren.

```
>>> print(my_person1.name)
>>> print(my_person2.vorname)
```

2. Erstelle eine Instanz der Klasse Person und nenne das Objekt dummy. Erstelle für dummy folgende Instanzvariablen: name = "Müller", vorname = "Jürg". Überlege zuerst was bei den folgenden Eingaben herauskommen sollte und kontrolliere es dann, indem du es laufen lässt. Interpretiere die Resultate.

```
1 print(dummy.vorname, dummy.name)
2 x = dummy
3 x.vorname = "Giovanni"
4 print(dummy.vorname, dummy.name)
```

3. Erkläre was im folgenden Programm gemacht wird:

```
1 class Person:
2     pass
3
4 class Auto:
5     pass
6
7 dummy = Person()
8 my_car = Auto()
9 my_car.marke = "Seat"
10 dummy.car = my_car
11 print(dummy.car.marke)
```

7.3 Die `__init__()` Methode

Von oben ist eigentlich klar, dass es vielleicht keinen Sinn macht, Personen ohne Identität (sprich ohne Instanzvariablen) herzustellen. Es wäre also nur logisch, wenn man bei der Herstellung eines Objekts, die Instanzvariablen, welche man auf sicher haben möchte, von Anfang an definieren könnte. In unserem Beispiel würde das heißen, dass wir der Person gleich bei der Herstellung eine Identität geben. Dies kann man machen, wenn in der Klasse die Funktion `__init__()` existiert. Die Funktion `__init__()` wird immer dann aufgerufen, wenn neue Objekte der Klasse instanziiert werden (z.B. `per = Person(...)`).⁹ Die gewünschte Belegung der Instanzvariablen können der Funktion `__init__()` einfach als Argumente übergeben werden. Auf unser Beispiel von oben angewandt, sieht es dann so aus:

```
1  # Neuer Bauplan für eine Person:
2  # Beim Erstellen eines Objekt der Klasse Person
3  # werden die Instanzvariablen direkt definiert.
4  class Person:
5      def __init__(self, name, vorname, geb_datum, gewicht):
6          self.name = name
7          self.vorname = vorname
8          self.geb_datum = geb_datum
9          self.gewicht = gewicht
```

Auf diese Weise kann keine Person ohne Identität erstellt werden und man kann sicherstellen, dass die geforderten Instanzvariablen (hier `name`, `vorname`, `geb_datum` und `gewicht`) auch sicher existieren. Erstellen wir nun eine neue Person:

```
>>> p2 = Person("Smith", "John", "04.04.04", 83)
>>> print(p2.name, p2.vorname, p2.geb_datum, p2.gewicht)
Smith John 04.04.04 83
```

Die Person `p2` wurde instanziiert und die geforderten Instanzvariablen sind garantiert belegt. Mit der `__init__()` Methode kommen wir der Idee des Bauplans aus Kapitel Unterschied zwischen Klassen und Objekten zum ersten Mal etwas näher, indem nun in der Klasse vorgeschrieben wird, wie ein Objekt der Klasse `Person` auszusehen hat.

Bemerkung

Das erste Argument `self` bei `__init__()` ist eine Referenz auf das Objekt. Auf diese Weise ist z.B. die Zuordnung in der Klasse `Person`

```
self.name = name
```

unmissverständlich. Das heißt `self.name` steht für die Instanzvariable des Objekts, welches erstellt wird und `name` steht für das Argument welches der Funktion `__init__()` übergeben wird. Natürlich kann man die Argumente auch anders benennen. Jedoch sollte klar ersichtlich sein, welches Argument zu welcher Instanzvariable gehört.

⁹Diejenigen die vielleicht mal in Java programmiert haben, könnten meinen, dass es sich bei `__init__()` um einen Konstruktor handelt. Jedoch ist die Sache in Python etwas anders. Der eigentliche Konstruktor wird implizit von Python gestartet und `__init__()` dient, wie der Name andeutet der Initialisierung der Attribute. Die `__init__()`-Methode wird jedoch unmittelbar nach dem eigentlichen Konstruktor gestartet und dadurch entsteht der Eindruck, als handle es sich um einen Konstruktor.

7.4 Methoden

Menschen sind nicht nur Träger von Merkmalen (Name, Vorname etc.), sondern besitzen auch ein Verhalten (z.B. „sich Vorstellen“ oder „Gewicht abnehmen“). Solche Verhaltensweisen können in Methoden/Funktionen innerhalb der Klasse definiert werden. Von oben haben wir gesehen, dass wir mit

```
>>> print(p2.name, p2.vorname, p2.geb_datum, p2.gewicht)
Smith John 04.04.04 83
```

auf die Instanzvariablen des Objekts zugreifen können. Jetzt möchte man aber nicht immer einen solch langen print-Befehl eingeben, sondern das Objekt (hier eine Person) soll sich gleich selber vorstellen. Dies kann auf folgende Weise innerhalb der Klasse Person realisiert werden:

```
1  # Neuer Bauplan für eine Person:
2  # Beim Erstellen eines Objekt der Klasse Person
3  # werden die Instanzvariablen direkt definiert.
4  class Person:
5      def __init__(self, name, vorname, geb_datum, gewicht):
6          self.name = name
7          self.vorname = vorname
8          self.geb_datum = geb_datum
9          self.gewicht = gewicht
10
11     def vorstellen(self):
12         text = "Hallo.\nIch heiße " \
13              + self.vorname + " " \
14              + self.name + ", wiege " \
15              + str(self.gewicht) + " kg und habe am " \
16              + self.geb_datum + " Geburtstag.\n" \
17              + "Nice to meet you."
18         print(text)
```


Wir haben also innerhalb der Klasse eine Funktion definiert. Das Argument `self` in der Klammer ist, wie schon bei der Funktion `__init__()`, eine Referenz auf das Objekt, auf welche diese Funktion angewendet wird.¹⁰

Jedes einzelne Objekt der Klasse Person hat nun die Möglichkeit auf die Funktion `vorstellen()` zu zugreifen.

Auf diese Weise bekommen wir, ohne viel Tipparbeit, gleich die Informationen der jeweiligen Personen, indem sie sich selber vorstellt. Wenden wir die Methode `vorstellen()` auf die beiden Personen `p1` und `p2` von oben an, so erhalten wir:

```
>>> p1.vorstellen()
Hallo.
Ich heiße Kurt Müller, wiege 73.5 kg und habe am 03.02.01 Geburtstag.
Nice to meet you.
```

¹⁰Man könnte auch einen anderen Namen als `self` verwenden (z.B. `this`). Denn über diesen Parameter erhält die Methode beim Aufruf eine Referenz auf das Objekt für welches sie aufgerufen wird. Es ist aber in Python üblich, dass der Name `self` verwendet wird.

	Name:	Klasse:	Datum:
	BS Scripte	BFS12 Python - 7	Imr

```
>>> p2.vorstellen()
```

```
Hallo.
```

```
Ich heiße John Smith, wiege 83 kg und habe am 04.04.04 Geburtstag.
```

```
Nice to meet you.
```

Hier sehen wir, dass die Funktion vorstellen(), je nach Objekt auf welches es angewandt wird, eine andere Ausgabe in der Konsole produziert. Das ist auch wünschenswert, denn jedes Objekt (hier jede Person) hat andere Eigenschaften (z.B. Name und Gewicht) und stellt sich dementsprechend auch anders vor.

Nun kann eine Person sich nicht nur vorstellen, sondern sie kann auch Gewicht verlieren, z.B. wenn sie Sport getrieben hat. Ein solches Verhalten können wir ebenfalls in der Klasse mit einer Funktion abnehmen() simulieren:

```

1  # Neuer Bauplan für eine Person:
2  # Beim Erstellen eines Objekt der Klasse Person
3  # werden die Instanzvariablen direkt definiert.
4  class Person:
5      def __init__(self, name, vorname, geb_datum, gewicht):
6          self.name = name
7          self.vorname = vorname
8          self.geb_datum = geb_datum
9          self.gewicht = gewicht
10
11     def vorstellen(self):
12         text = "Hallo.\nIch heiße " \
13             + self.vorname + " " \
14             + self.name + ", wiege " \
15             + str(self.gewicht) + " kg und habe am " \
16             + self.geb_datum + " Geburtstag.\n" \
17             + "Nice to meet you."
18         print(text)
19
20     def abnehmen(self, wie_viel):
21         print("Altes Gewicht:", self.gewicht, "kg")
22
23         # Das neue Gewicht in der Instanzvariable
24         # des Objektes speichern
25         self.gewicht = self.gewicht - wie_viel
26
27         print("Neues Gewicht:", self.gewicht, "kg")

```

Anders als die Funktion vorstellen() (welche nur Informationen auswirft) verändert die Funktion abnehmen() das Objekt, indem es die Instanzvariable gewicht des Objektes anpasst. Dessen muss man sich immer Bewusst sein. Ist die gewünschte Änderung des Objektes wirklich im Sinne meines Programms?

```
>>> p1.gewicht
73.5
>>> p1.abnehmen(3)
Altes Gewicht: 73.5 kg
Neues Gewicht: 70.5 kg
>>> p1.gewicht
70.5
```

Das obige Beispiel zeigt, dass das Objekt (hier die Person p1), nach dem Aufruf der Funktion `abnehmen()`, verändert wurde.

Bemerkung

Die Instanzvariablen der Klasse `Person` (`name`, `vorname`, `geb_datum` und `gewicht`) sowie die Methoden (`vorstellen()` und `abnehmen()`) sind nur Objekten derselben Klasse vorbehalten. Eine Eingabe wie

```
a = 1
a.vorstellen()
```

wird eine Fehlermeldung produzieren, da `a` hier ein Integer ist und somit die Funktion `vorstellen()` als Integer nicht kennt.

7.5 Public-, Protected- und Private Instanzvariablen

Manchmal macht es Sinn, dass gewisse Instanzvariablen nicht ohne Überprüfung einfach geändert werden können oder sie erst gar nicht gegen Außen sichtbar sein sollten. Nehmen wir als Beispiel folgende Eingabe:

```
>>> p1.gewicht = -20
```

`p1` ist eine Instanz der Klasse `Person`, wie wir es oben schon definiert haben. Hier wurde nun dem Gewicht der Person `p1` einen negativen Wert zugeordnet, was in der Realität gar nicht vorkommen kann.

Möchte man verhindern, dass die Instanzvariablen einer Klasse von außen ohne weiteres geändert oder gar gelesen werden kann, gibt es zwei Möglichkeiten, die Python einem zur Verfügung stellt.

1. Jedes Attribut, welches mit genau einem Unterstrich beginnt, ist `protected`. In diesem Fall kann das Attribut zwar immer noch gelesen und verändert werden, aber durch den Unterstrich wird kommuniziert, dass dies verboten oder nicht erwünscht ist.
2. Wenn der Name eines Attributes mit zwei Unterstrichen beginnt, wird der Zugriff von Außen durch Python weiter erschwert. In diesem Fall kann nur noch über `obj._classname__attribut` darauf zugegriffen werden. Ein solches Attribut wird `private` genannt.

Dazu sehen wir uns folgende Beispielklasse an¹¹:

```

1 class A():
2     def __init__(self):
3         self.__priv = "Ich bin privat"
4         self._prot = "Ich bin protected"
5         self.pub = "Ich bin öffentlich"

```

Wir haben hier also folgende Situation:

Name	Bezeichnung	Bedeutung
pub	public	Kann von Außen gelesen und geändert werden.
_prot	protected	Kann von Außen gelesen und geändert werden, jedoch sollte es nicht gemacht werden (Empfehlung vom Entwickler).
__priv	private	Kann von Außen nur gelesen werden, wenn der Klassenname angefügt wird.

Im folgenden Code-Snippet sehen wir sehr gut, wie sich die Entsprechenden Instanzvariablen verhalten:

```

>>> x = A()
>>> x.pub
'Ich bin öffentlich'
>>> x.pub = "Man kann meinen Wert ändern und das ist gut so"
>>> x.pub
'Man kann meinen Wert ändern und das ist gut so'
>>> x._prot
'Ich bin protected'
>>> x._prot = "Mein Wert kann aber sollte nicht von außen \
geändert werden!"
>>> x._prot
'Mein Wert kann aber sollte nicht von außen geändert werden!'
>>> x.__priv
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__priv'
>>> x._A__priv
'Ich bin privat'
>>> x._A__priv = "Auch ich kann verändert werden"x._A__priv
'Auch ich kann verändert werden'

```

Wir sehen also, dass wir auf die Instanzvariable `__priv` des Objekts `x` nur Zugriff erhalten, wenn wir `_A__priv` benutzen.

¹¹Beispiel aus http://www.python-kurs.eu/python3_klassen.php

7.5.1 Setter- und Getter-Methoden

Nun kann man sich fragen, wozu solche private Instanzvariablen gut sind, wenn sie von Außen sowieso nur umständlich verändert werden können. Sehen wir uns dazu die folgende Klasse an:

```
1 class Motorrad():
2     def __init__(self, marke, hubraum):
3         self.marke = marke
4         self.__hubraum = hubraum
```

Wir wollen nicht, dass ein Benutzer ein Motorrad mit negativem Hubraum modelliert. Aus diesem Grund haben wir hier die Instanzvariable `__hubraum` private gesetzt. Der Wert kann nun unter Beobachtung mit einer sogenannten setter() Methode geändert werden:

```
1 class Motorrad():
2     def __init__(self, marke, hubraum):
3         self.marke = marke
4         self.__hubraum = hubraum
5
6     def set_hubraum(self, kubik):
7         if (kubik <= 0):
8             print("Error: Negativer Wert für den Hubraum! \
9 Der Wert wurde nicht geändert")
10        else:
11            self.__hubraum = kubik
12            print("Hubraum wurde geändert.")
```

Auf diese Weise haben wir in Zeile 6 die Kontrolle, dass `__hubraum` keine negativen Werte annehmen kann. Dies bedeutet, dass die Veränderung der Instanzvariable `__hubraum` in eine Klassenmethode ausgelagert wurde, in welcher die Manipulation am Objekt und somit der Instanzvariable kontrolliert und überwacht werden kann.

Um nun auch noch den Wert abfragen zu können, erstellen wir zusätzlich noch eine getter() Methode. Dies kann dann so aussehen:

```
1 class Motorrad():
2     def __init__(self, marke, hubraum):
3         self.marke = marke
4         self.__hubraum = hubraum
5
6     def set_hubraum(self, kubik):
7         if (kubik <= 0):
8             print("Error: Negativer Wert für den Hubraum! \
9 Der Wert wurde nicht geändert")
10        else:
11            self.__hubraum = kubik
12            print("Hubraum wurde geändert.")
13
14    def get_hubraum(self):
15        return self.__hubraum
```

In der Python-Konsole können wir die Klasse nun testen, was dann wie folgt aussieht:

```
>>> toeff = Motorrad("Yamaha", 600)
>>> toeff.get_hubraum()
600
>>> toeff.set_hubraum(-300)
Error: Negativer Wert für den Hubraum! Der Wert wurde nicht geändert
>>> toeff.get_hubraum()
600
>>> toeff.set_hubraum(300)
Hubraum wurde geändert.
>>> toeff.get_hubraum()
300
```

Siehe auch

Der oben beschriebene Ansatz um Getter und Setter zu erstellen funktioniert in ähnlicher Art in vielen anderen Programmiersprachen auch. Ein erfahrener Python-Programmierer wird dies aber nicht so umsetzen, sondern sogenannte Decorators benutzen. Mit der `property()` Funktion können Properties wie in den hier beschriebenen Beispielen umgesetzt werden:
<https://docs.python.org/3/library/functions.html#property>

7.5.2 Aufgaben

- Wir erstellen ein Objekt der Klasse Motorrad:

```
>>> motrad = Motorrad("KTM", 950)
```

Überlege dir, welche der folgenden Eingaben valide sind und welche nicht. Begründe. Was wird jeweils auf der Konsole ausgegeben?

```
>>> motrad.set_hubraum(600)
>>> motrad.__hubraum += 400
>>> motrad.marke += 2
>>> motrad.__hubraum
>>> motrad.marke
>>> motrad.get_hubraum()
```

- Erneut erstellen wir ein Objekt der Klasse Motorrad.

- Erkläre und begründe was auf der Konsole ausgegeben wird. Worin besteht die Problematik?

```
>>> das_rad = Motorrad("Buell", -1200)
>>> das_rad.get_hubraum()
```

- Wie müsste man die Klasse Motorrad anpassen, um den unerwünschten Effekt von oben zu vermeiden? Vermeide dabei Codedublizität.

- Schreibe die Klasse Person von oben folgendermaßen um, so dass das Gewicht einer Person garantiert nie negativ gesetzt werden kann. Benutze dazu `setter()` und `getter()` Methoden.

4. Die Klasse Motorrad von oben wurde um die Zeile 17 erweitert.

```
1 class Motorrad():
2     def __init__(self, marke, hubraum):
3         self.marke = marke
4         self.__hubraum = hubraum
5
6     def set_hubraum(self, kubik):
7         if (kubik <= 0):
8             print("Error: Negativer Wert für den Hubraum! \
9 Der Wert wurde nicht geändert")
10        else:
11            self.__hubraum = kubik
12            print("Hubraum wurde geändert.")
13
14    def get_hubraum(self):
15        return self.__hubraum
16
17    hubraum = property(get_hubraum, set_hubraum)
18
19 if __name__ == "__main__":
20     motrad = Motorrad("Suzuki", 250)
21     print(motrad.hubraum)
22     motrad.hubraum = -40
23     print(motrad.hubraum)
24     motrad.hubraum += 50
25     print(motrad.hubraum)
```

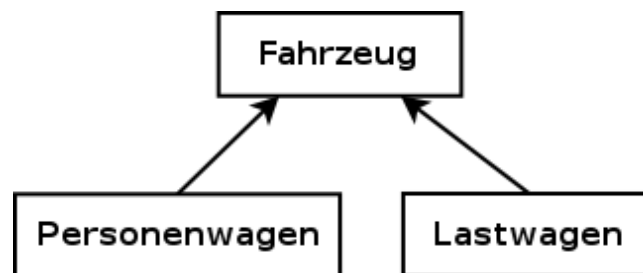
Lasse das Programm laufen und erkläre was in den Zeilen 20 - 25 passiert.

Informiere dich im Internet über das Thema Properties (z.B. auf der Seite http://www.python-kurs.eu/python3_properties.php).

7.6 Vererbung

Das Konzept der Vererbung erlaubt es uns spezialisiertere Klassen einer allgemeinen Klasse zu erstellen. Die spezialisierte Klasse soll dabei alle Eigenschaften der allgemeinen Klasse besitzen, so dass nur noch wenige Eigenschaften hinzugefügt werden müssen. Die Klasse, von welcher geerbt wird, nennt man Oberklasse, Superklasse oder Basisklasse und die Klasse, welche erbt, wird Unterklasse, abgeleitete Klasse oder Subklasse genannt.

So könnte z.B. die Klasse Fahrzeug eine Oberklasse der Unterklassen Personenwagen und Lastwagen sein. Jedes Fahrzeug ist durch die marke, den hubraum und die leistung charakterisiert. Bei den Personenwagen möchte man noch zusätzlich wissen, wie viele Sitzplätze es hat und beim Lastwagen wie schwer die Fracht sein darf.



Um im Programm zu deklarieren, dass die Klasse Personenwagen von der Klasse Fahrzeug erbt, setzt man beim Namen der Unterklasse einfach den Namen der Oberklasse in Klammern (Personenwagen(Fahrzeug)). In einem Programm könnte das folgendermaßen aussehen:

```

1 class Fahrzeug:
2     def __init__(self, marke, hubraum, leistung):
3         self.marke = marke
4         self.hubraum = hubraum
5         self.leistung = leistung
6
7     def get_infos(self):
8         return "Marke: " + self.marke + ", Hubraum: " + \
9             str(self.hubraum) + ", Leistung: " + str(self.leistung)
10
11 class Personenwagen(Fahrzeug):
12     pass
13
14 class Lastwagen(Fahrzeug):
15     pass
  
```

Testen wir die Klasse in der Konsole, dann könnte das so aussehen:

```

>>> pw = Personenwagen("Opel", 222, 100)
>>> lkw = Lastwagen("Mercedes", 5000, 300)
>>> print(pw.get_infos())
Marke: Opel, Hubraum: 222, Leistung: 100
>>> print(lkw.get_infos())
Marke: Mercedes, Hubraum: 5000, Leistung: 300
  
```

Wir sehen also, dass die beiden Unterklassen alle Eigenschaften, d.h. alle Instanzvariablen und alle Methoden der Oberklasse geerbt haben.

7.6.1 Überschreiben

Das obige Beispiel zeigt, dass die abgeleiteten Klassen die Eigenschaften der Oberklassen geerbt haben. Nun möchten wir aber noch die speziellen Eigenschaften der Unterklassen im Programm einbauen. Damit sind bei der Klasse Personenwagen die Anzahl Sitzplätze und bei der Klasse Lastwagen die Schwere der Fracht gemeint.

Um diese Eigenschaften zu implementieren führen wir in den Unterklassen eigene `__init__()` - Methoden ein. Dies hat zur Folge, dass wenn wir ein Objekt der Klasse Personenwagen erstellen, nun nicht mehr die `__init__()` - Methoden der Klasse Fahrzeug aufgerufen wird, sondern die der Klasse Personenwagen.


Um aber Code Dublizität zu vermeiden, können wir in der Unterklasse die `__init__()` - Methode der Oberklasse aufrufen. Dies geschieht mit dem Keyword `super()`.

Ebenfalls überschreiben wir die Methode `get_info()` und passen sie auf die entsprechende Unterklasse an.

```

1  class Fahrzeug:
2      def __init__(self, marke, hubraum, leistung):
3          self.marke = marke
4          self.hubraum = hubraum
5          self.leistung = leistung
6
7      def get_infos(self):
8          return "Marke: " + self.marke + ", Hubraum: " + \
9              str(self.hubraum) + ", Leistung: " + str(self.leistung)
10
11 class Personenwagen(Fahrzeug):
12     def __init__(self, marke, hubraum, leistung, anz_plaetze):
13         super().__init__(marke, hubraum, leistung)
14         self.anz_plaetze = anz_plaetze
15
16     def get_infos(self):
17         return super().get_infos() + ", Anzahl Plaetze: " + \
18             str(self.anz_plaetze)
19
20 class Lastwagen(Fahrzeug):
21     def __init__(self, marke, hubraum, leistung, last):
22         super().__init__(marke, hubraum, leistung)
23         self.last = last
24
25     def get_infos(self):
26         return super().get_infos() + ", Lastgewicht: " + \
27             str(self.last)
28
29 if __name__ == "__main__":
30     pw = Personenwagen("Opel", 222, 100, 5)
31     lk = Lastwagen("Mercedes", 5000, 300, 2000)
32     print(pw.get_infos())
33     print(lk.get_infos())

```

	Name:	Klasse:	Datum:
	BS Scripte	BFS12 Python - 7	Imr

Das Programm liefert folgenden Output:

```
>>>
```

```
Marke: Opel, Hubraum: 222, Leistung: 100, Anzahl Plaetze: 5
```

```
Marke: Mercedes, Hubraum: 5000, Leistung: 300, Lastgewicht: 2000
```

Bemerkung

Alternativ kann man auch über den Klassennamen anstelle von `super()` auf die Methoden der Oberklasse zugreifen. Zeile 13 von oben könnte dann folgendermassen aussehen:

```
Fahrzeug.__init__(self, marke, hubraum, leistung)
```

Dies ist aus folgendem Grund wichtig: In Python besteht auch die Möglichkeit einer Mehrfachvererbung, d.h. dass eine Unterklasse mehr als nur eine Basisklasse besitzen kann. Wir gehen hier aber nicht näher darauf ein.

7.7 Aufgaben

- Überlege dir, was die Python-Konsole ausgibt, wenn folgende Programme ausgeführt werden. Erkläre weshalb.

Klasse 1:

```
1 class Velo():
2     def __init__(self, farbe, alter):
3         self.farbe = farbe
4         self.alter = alter
5
6 v1 = Velo("gelb", 5)
7 v2 = Velo("rot")
```

Klasse 2:

```
1 class Velo():
2     def __init__(self, farbe, alter=0):
3         self.farbe = farbe
4         self.alter = alter
5
6 v1 = Velo("gelb", 5)
7 v2 = Velo("rot")
8 print(v1.alter)
9 print(v2.alter)
```

- Mache dich z.B. im Internet über das Thema Operatorüberladung schlau. Benutze das Wissen um für die Klasse Brueche von oben die Addition mittels des „+“-Operators zu definieren. Make das gleiche auch für den „*“-Operator und den „=-Operator.

3. In dieser Aufgabe wird eine Mitarbeiterdatenbank irgendeiner Firma simuliert.

(a) Schreibe eine Klasse `Mitarbeiter` mit den drei Instanzvariablen `vorname`, `nachname` und `lohn`. Achte darauf, dass bei der Instanzierung eines Mitarbeiters, die Instanzvariablen auch sicher belegt werden und der Lohn nicht kleiner als 3800 Euro ist.

(b) Erstelle in der Klasse `Mitarbeiter` Methoden mit folgenden Funktionalitäten:

i. `get_mitarbeiter_id()`

Diese Methode soll auf der Konsole folgendes ausgeben, wenn sie auf einen Mitarbeiter angewandt wird:

„Ich heiße Hans Mustermann. Mein Lohn bei dieser Firma beträgt 4000 Euro.“

ii. `lohn_erhoehen()`

Beim Ausführen dieser Methode, erhalte der jeweilige Mitarbeiter einen höheren Lohn. Wie viel mehr Lohn er erhält, kann als Argument der Methode übergeben werden.

iii. `lohn_senken()`

Hier wird, wie der Name schon sagt, der Lohn gesenkt. Um wie viel der Lohn gesenkt wird, kann wiederum als Argument der Methode übergeben werden. Zusätzlich gibt die Methode einen String zurück, welcher mitteilt, ob die Senkung erfolgreich war, oder nicht. Denn falls der Lohn bei der Senkung unter 3800 Euro fällt, so wird das Unterfangen abgebrochen. Dies könnte dann z.B. so aussehen:

```
>>> info = arbeiter.lohn_senken(200)
```

```
>>> print(info)
```

Erfolg: Der Lohn wurde um 200 Euro. gesenkt und beträgt nun 3900 Euro.

```
>>> info = arbeiter.lohn_senken(150)
```

```
>>> print(info)
```

Error: Der Lohn kann wegen der Mindestlohninitiative nicht gesenkt werden.

Er bleibt bei 3900 Euro.

Bitte mit der Gewerkschaft reden.

iv. `get_initialen()`

Die Methode gibt lediglich die Initialen des betroffenen Mitarbeiters zurück. Bei einem Mitarbeiter mit dem Namen Hans Müller wäre das also H.M. Baue diese Methode auch in der bereits erstellten Methode `get_mitarbeiter_id()` ein, um z.B. folgende Ausgabe zu erstellen:

„Ich heiße Hans Mustermann (alias H.M.). Mein Lohn bei dieser Firma beträgt 100 Euro.“

(c) Test dein Programm.

4. Brüche

- Definiere eine Klasse `Brueche`, welche eine Bruchzahl modelliert. Die Klasse soll die Instanzvariablen `zaehler` und `nenner` vom Typ `Integer` besitzen. Stelle immer sicher, dass der Nenner nicht den Wert 0 annehmen kann.
 - Füge der Klasse eine Methode mit dem Namen `print_wert()` hinzu, welche keine Parameter erwartet und den Wert des Bruches auf der Konsole ausgibt. Die Ausgabe könnte zum Beispiel so aussehen:
„Der Wert des Bruches beträgt 2/3.“
 - Füge der Klasse eine weitere Methode hinzu und zwar mit dem Namen `add()`. Diese Methode erhält als Argument ein Objekt derselben Klasse `Brueche` und addiert diesen mit dem Bruch-Objekt, auf welche die Methode angewandt wird. Speichere das Resultat in ein neues Objekt der Klasse `Brueche` und gib es mit dem `return` Statement zurück.
 - Das gleiche Prinzip wende nochmals für die Methode `multiply()` an.
 - Füge noch eine Methode `kuerzen()` hinzu, welche den referenzierten Bruch kürzt, falls möglich. Erweitere mit ihr die Methoden `add()` und `multiply()`.
 - Definiere noch eine letzte Methode `equal()`. Die Methode soll wahr zurückliefern, sofern der übergebene Bruch dem gleichen Wert wie dem aufgerufenen Bruch entspricht.
5. Python bietet uns ja Listen als Datenstruktur an, zusammen mit einer ganzen Sammlung an Methoden (z.B. `append()`, `pop()`). Jetzt kann es vorkommen, dass man manchmal bei den Listen (oder auch anderen Klassen) zusätzliche oder eingeschränkte Funktionalitäten bereitstellen möchte. Um zusätzliche Funktionalität anzubieten greifen wir einfach auf das Konzept der Vererbung zurück.

In dieser Aufgabe wollen wir mit Hilfe von Listen, eine eigene Datenstruktur implementieren, welche uns die Funktionalität eines Stacks zur Verfügung stellt. Ein Stack oder Stapelspeicher ist wie eine Liste ein Container, der aber nur Zugriff auf das zuletzt hinzugefügte Element gewährt. Für mehr Informationen siehe z.B. unter <https://de.wikipedia.org/wiki/Stapelspeicher> nach.

Erweitere nun die Klasse `Stapel`,

```
1 class Stapel:
2     def __init__(self):
3         self.inhalt = []
```

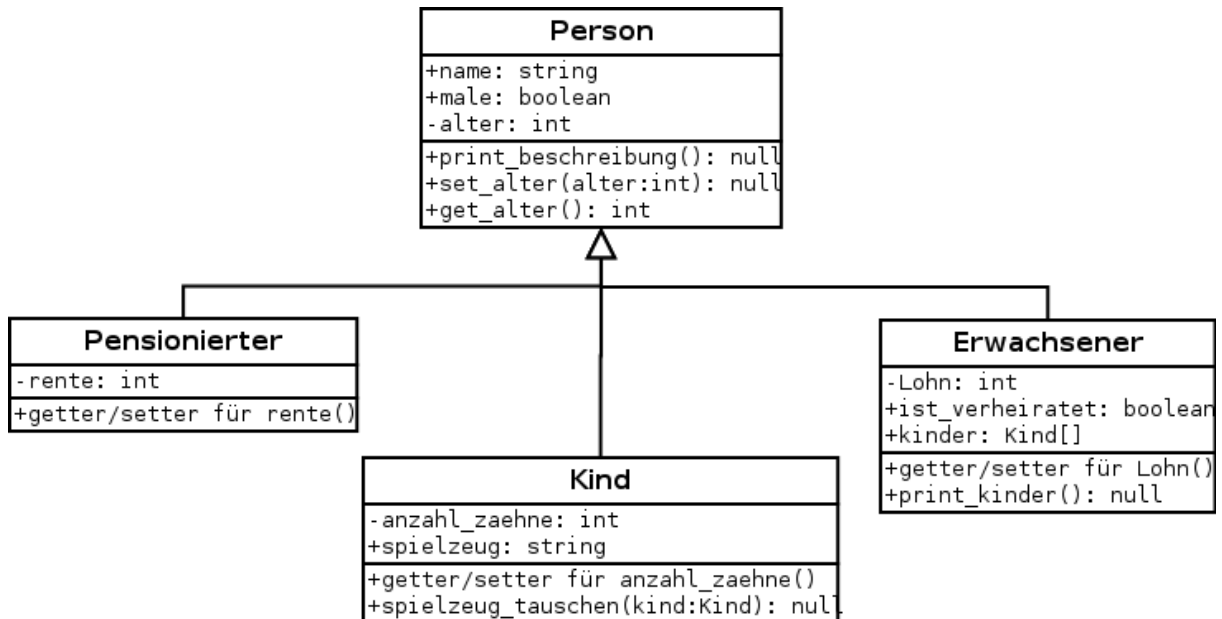
so dass sie folgende Funktionalitäten bereitstellt:

- Auf die Instanzvariable `inhalt` (Liste deren Inhalt den Inhalt des Stacks widerspiegelt) nicht außerhalb der Klasse zugegriffen werden kann.
- Ergänze die Klasse `Stapel` um die Funktion `add(element)` mit der man dem Stack ein neues Element hinzufügen kann.
- Ergänze die Klasse `Stapel` um die Funktion `is_empty()` welche `True` zurück gibt, wenn der Stack zur Zeit leer ist, sonst `False`.
- Ergänze die Klasse `Stapel` um die Funktion `get_last()` welche das zuletzt hinzugefügte Element des Stacks zurückgibt.
Beachte: Damit die Funktion funktioniert, muss sich mindestens ein Element in der Liste befinden.

- (e) Ergänze die Klasse Stapel um die Funktion `pop_last()` welche das zuletzt hinzugefügte Element des Stacks zurückgibt und es aus der Liste entfernt.
- (f) Ergänze die Klasse Stapel um die Funktion `clear_stack()` welche den Stack leert und die Elemente der Reihe nach (d.h. das älteste Element zuletzt) in der Konsole auf einer Zeile ausgibt. Vermeide Codeduplizität.

6. Vererbung

Gegeben sei das folgende Klassendiagramm:



Hinweise zum Diagramm:

- Der Pfeil bedeutet: „Erbt von“
- Im obersten Teilkästchen wird jeweils der Klassenname notiert. Anschließend folgen die Instanzvariablen und zum Schluss die Methoden.
- Instanzvariablen werden gemäß der folgenden Notation illustriert:
`<+/-> <name> : <type>`
wobei ein + für public und ein - für private steht.
- Analog bei Methoden:
`<+/-> <name>(<parameter>) : <return-type>`

Implementiere alle Klassen gemäß dem Klassendiagramm. Die Methode `print_beschreibung()` soll eine kurze Beschreibung ausgeben:

„Ich heiße Hans Muster, bin männlich, 70 Jahre alt und Pensionär(in).“

Achte darauf, dass das Alter eines Erwachsenen zwischen 18 und 61 Jahre beträgt. Kinder sind jünger als 18 Jahre und die Pensionierten älter als 61 Jahren.

Die Funktion `print_kinder()` in der Klasse Erwachsener gibt die Namen der Kinder auf der Konsole aus, falls sie überhaupt Kinder besitzt.

Vermeide Code Dublizität.