Programmierung mit Python für Programmierer

Kapitel 4 - Objektorientierte Programmierung

Autor: Dr. Christian Heckler

Vorbemerkungen

- Verwendete Literatur: siehe Referenzen
- Verwendete Symbole:
  - ❗: Beispielprogramm
  - ❗: Weitere Erläuterungen im Kurs
  - ❓: Übung

Motivation

- „Programmieren im Kleinen“ vs. „Programmieren im Großen“
- „Divide et impera“ („Teile und herrsche“)

Wiederholung - Datentypen

- Bisher: Verarbeitung von Daten verschiedenen Typs, z.B.
  - ganze Zahlen, Gleitkommazahlen
  - Wahrheitswerte
  - Zeichenketten
  - Listen, Tupel, Dictionaries
- Ein *Datum* („Wert“) hat also einen *Typ* („Datentyp“)
- Ein Datentyp legt fest:
  - die Menge der möglichen Werte (z.B. Zeichenketten)
  - die möglichen Operationen auf diesen Werten (z.B. `+`, `-`, `append`)
  - die Bedeutung der Operationen
    - die Bedeutung der `+`-Operation ist z.B. bei Zahlen
    - eine andere als bei Zeichenketten

Datentypen 2

- Jeder Wert hat einen eindeutigen Typ.
- Dieser kann mit der `type()`-Funktion bestimmt werden.
- Variablen können Werte verschiedenen Typs zugewiesen werden:

```
x = 1
type(x)
x = "Hallo Python"
type(x)
```

Datentypen 3

- Ein Datentyp modelliert also ein Konzept der „realen Welt“.
- Z.B. das Konzept der Zahlen, der Zeichenketten …
- Sie können verwendet werden, ohne die interne Implementierung zu kennen.
- Es reicht, die möglichen Operationen und deren Bedeutung zu kennen.
- Frage: Kann man eigene, neue Datentypen schreiben, um ein „Konzept der realen Welt“ darzustellen?

Objektorientierte Programmierung

- Idee: Definition eigener Datentypen zur Modellierung der zu implementierenden Fachlichkeit.
- Die interne Implementierung wird vor dem Anwender verborgen (*Datenkapselung*, *Information Hiding*)
- Beispiele:

  Bankanwendung

  Konto, Kontoauszug, Überweisung

  Web-Shop

  Einkaufswagen, Rechnung, Bestellung

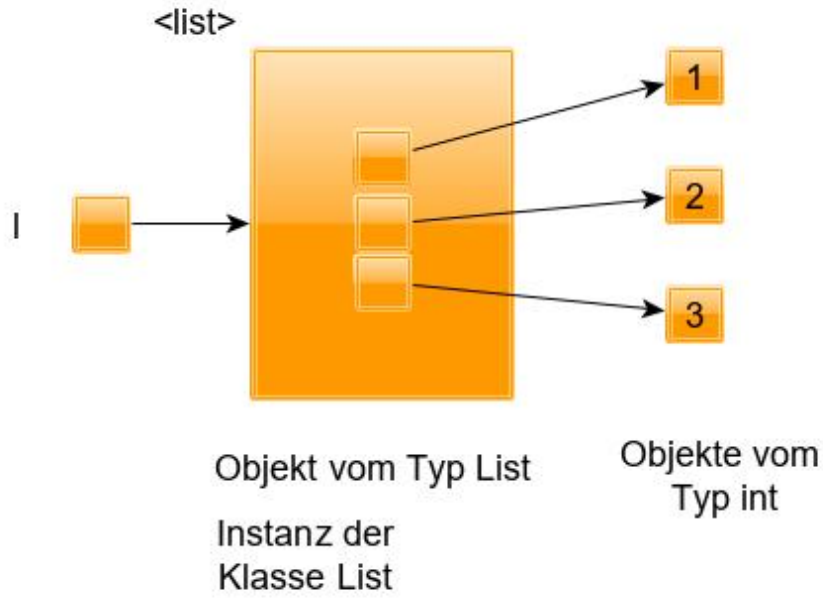
- Ein weiteres wichtiges Konzept von OO ist das Konzept der „Vererbung“, das im nächsten Kapitel vorgestellt wird.

Nomenklatur

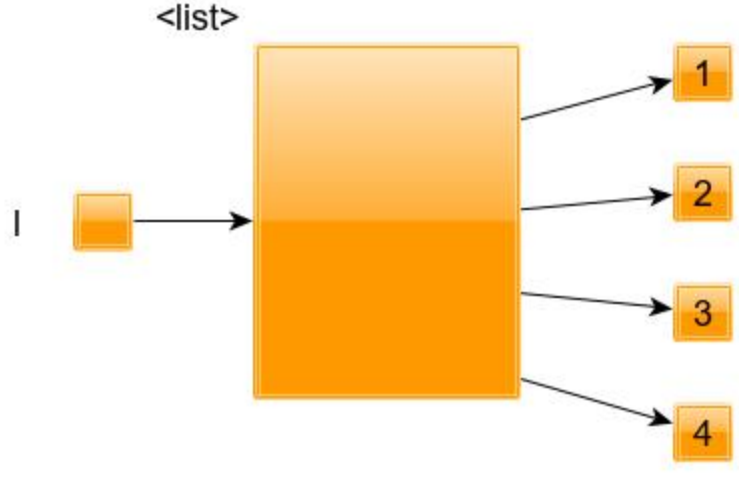
- Ein Datentyp wird *Klasse* genannt.
- Ein konkreter Wert einer Klasse wird *Objekt* genannt oder *Instanz einer Klasse*
  - Die Zeichenkette `"Hallo"` ist ein also Objekt (Instanz) der Klasse String.
  - Die Zahl `3` ist ein Objekt der Klasse int.
- Eine Funktion / Operation, die für ein Objekt definiert ist, wird *Methode* genannt:
  - Bsp: Wenn die Variable `l` einen Wert (Objekt) vom Typ Liste enthält, kann man mit der `append`-Methode für `l` (z.B. `l.append("Python")` einen Wert (Objekt) zu der Liste hinzufügen).

Illustration an Hand der Klasse `list`

```
l = [1,2,3]
type(l)
```




```
l.append(4)
# append ist eine Methode von list
```



Objektidentitäten und Gleichheit

- Jedes Objekt hat
  - eine Identität (siehe Funktion `id`)
  - und einen Zustand
- Objekte können gleich sein, obwohl sie nicht identisch sind:

```
l1 = [1,2,3,4]
l2 = l1
print(id(l1) == id(l2)) # True

l2 = [1,2,3,4]
print(id(l1) == id(l2)) # False
print(l1 == l2) # True
```

Einschub - Objektorientierung und Python

- In Python entspricht jeder Datentyp einer Klasse im Sinne der OO (s. auch letzte Folie).
- Das ist nicht in allen Programmiersprachen so. In vielen (Compiler-) Programmiersprachen sind Zahlen sogenannte *primitive* Datentypen. Zahlenwerte haben keine Identität und es wird auch keine Referenzen auf einen Zahlenwert gebildet.
- Daher hört man schon mal die Aussage: „In Python ist alles ein Objekt.“

Einschub - Objektorientierter Entwurf

- Die „Kunst“ besteht nicht in der Programmierung einer Klasse (wie wir gleich sehen werden).
- Beim Entwurf eines größeren Softwaresystems besteht die Schwierigkeit darin,
  - das System in geeignete Klassen zu zerlegen und
  - die Verantwortlichkeiten der Klassen zu definieren.
- Dabei helfen *Entwurfsprinzipien* und sogenannte *Entwurfsmuster*, siehe z.B.
  - https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs
  - https://de.wikipedia.org/wiki/Entwurfsmuster

Beispiel: Konto-Klasse

Ziel: Programmierung einer Klasse `Konto`, so dass in einem Programm beispielsweise folgendes möglich ist (die Idee einer Konto-Klasse ist [EK] entnommen):

- Erzeugung von Objekten vom Typ `Konto` und Zuweisung an Variablen

```
konto1 = Konto("Maier", 4711, 100.00)
konto2 = Konto("Müller", 4712,10.00)
```

- Verwaltung einer Liste von Konto-Objekten

```
kontoliste = [konto1, konto2]
```

- Einzahlen eines Betrages auf ein Konto durch Aufruf einer *Methode*

```
konto1.einzahlen(100)
```

Attribute einer Klasse

- Um die gewünschte Funktionalität zur Verfügung zu stellen, müssen in einem Kontoobjekt beispielsweise die folgenden Daten verwaltet werden:
  - Eine Zeichenkette für den Kontoinhaber.
  - Eine ganze Zahl für die Kontonummer.
  - Eine Gleitkommazahl für den Kontostand.
- Ein Wert, der innerhalb einer Klasse verwaltet wird, nennt man *Attribut*.
- Ein Objekt vom Typ Konto hat also drei Attribute
  - Kontoinhaber
  - Kontonummer
  - Kontostand

Kontobeispiel

Die zu schreibende Kontoklasse sollte also folgendermaßen aussehen:

Attribute

- Kontoinhaber
- Kontonummer
- Kontostand

Methoden

- Erzeugung eines Kontos für einen Kontoinhaber mit einer Kontonummer
- Einzahlen eines Betrages
- Auszahlen eines Betrages
- Anzeige des Kontostandes

Einschub: Implementierung ohne OO

- Wie könnte eine Implementierung aussehen ohne OO?
  - Modellierung eines Kontos als Tupel.
  - Modellierung eines Kontos als Dictionary.
- Wie sähen in diesem Falle die Methoden / Funktionen (einzahlen, ueberweisen) aus?
- Nachteil: die Funktionen sind getrennt von den Daten. (Natürlich könnte man alle Funktionen, die auf den entsprechenden Datentypen arbeiten, in einem Modul bündeln).
- Idee OO: Daten (Attribute) und Funktionen (Methoden) an einer Stelle zusammenführen (Klasse).

Einschub: Grundsätzliches zu Python und OO

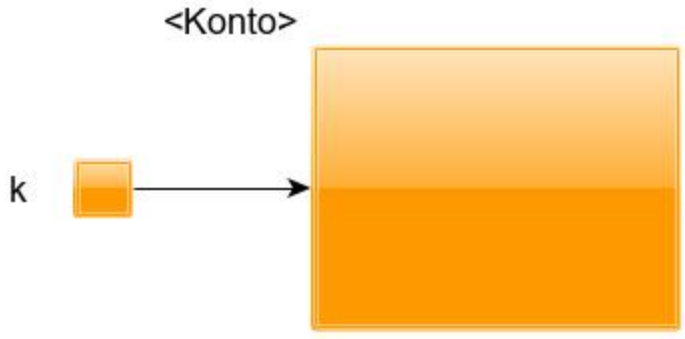
- In einer Klasse werden die Attribute und Methoden definiert.
- Alle Objekte der Klasse besitzen dann diese Attribute und Methoden.
- Python (als dynamische Sprache):
  - Variablen werden durch die Zuweisung eines Wertes „deklariert“
  - Man kann also nicht „statisch“ in einer Klasse die Attribute der Klasse definieren
  - Statt dessen können Attribute dynamisch zu einem Objekt zugefügt werden.
  - Damit alle Objekte einer Klasse dieselben Attribute haben, sollten diese also in einer Methode erzeugt werden.

Beispielimplementierung

Beispielimplementierung: ❗ `konto.py`

Illustration - Erzeugung eines Objektes 1

Schritt 1: Erzeugung eines "leeren" Objektes



Schritt 2: Aufruf von `k.__init__("Fritz", 4611, 100.00)`
(entspricht: `Konto.__init__(k, "Fritz", 4611, 100.00)`)

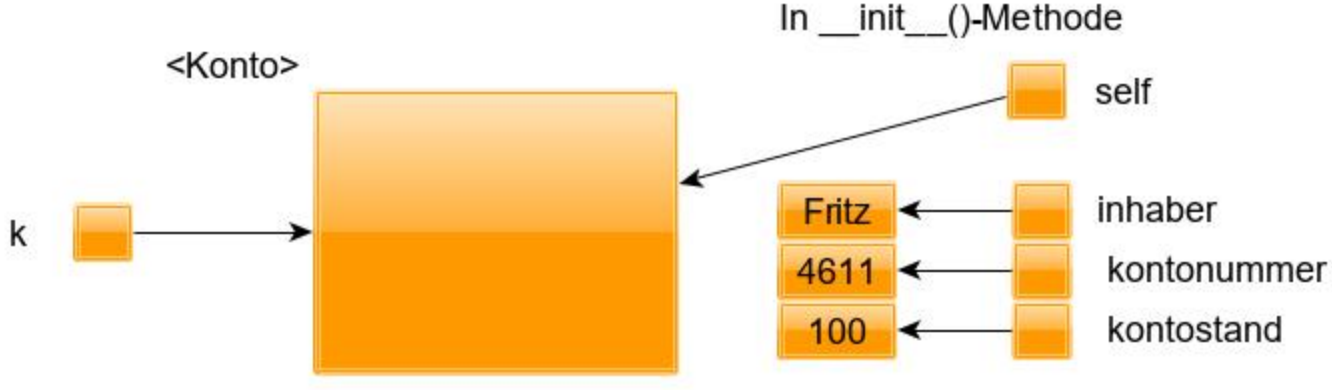
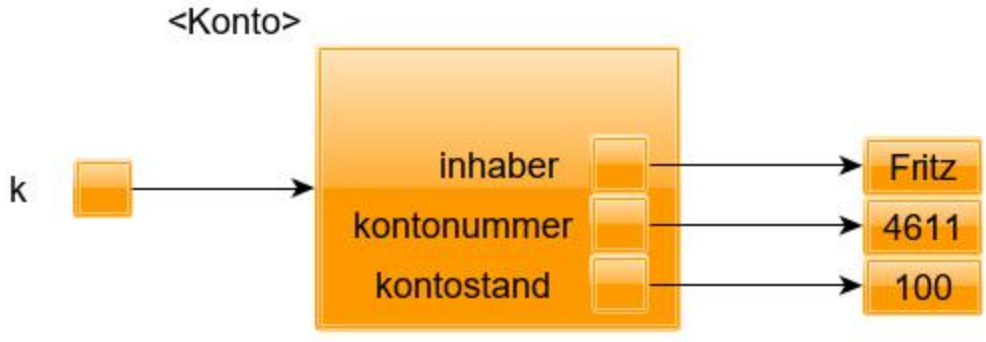


Illustration - Erzeugung eines Objektes 2

Schritt 3: Die Zuweisungen in der `__init__()` - Methode bewirken das Anlegen der Attribute in dem Objekt.



Die minimale Klasse

- Wie definiert man eine Klasse?

```
class Konto:
    pass
```

- Erstellung von Objekten von diesem Typ:

```
mein_konto = Konto()
dein_konto = Konto()
```

- Mit diesen Objekten kann man noch nicht viel anfangen. Es fehlen noch die Attribute und die Methoden.

Methoden

- Bsp.: Die Konto-Klasse soll eine Methode `einzahlen` bekommen.

```
class Konto:
    def einzahlen(self, betrag):
        print(„Methode einzahlen von Konto“)
k = Konto()
k.einzahlen(10) # Interpretiert wie Konto.einzahlen(k, 10)
```

- Eine Methode ist also eine Funktion,
  - die innerhalb einer Klassendefinition definiert ist,
  - die über ein Objekt aufgerufen wird,
  - bei der der erste Parameter die Referenz dieses Objektes ist (wird üblicherweise `self` genannt).
  - Beim Aufruf entfällt self (wird von Python wie oben umgesetzt)

Methoden 2

- Aufruf über ein konkretes Objekt (`k.einzahlen(10)`).
- In einer Methode ist `self` eine Referenz auf das Objekt.
- In einer Methode wird eine andere Methode über die Referenz self aufgerufen, z.B.:

```
class Konto:
    def f1(self):
        print(“Methode f1 von Konto“)
    def f2(self):
        print(“Methode f2 von Konto“)
        self.f1()
```

Attribute

- Attribute können dynamisch einem Objekt hinzugefügt werden.
- Wie in Python üblich durch die Zuweisung eines Wertes.
- Dies kann außerhalb der Klassendefinition über ein konkretes Objekt erfolgen:

```
class Konto:
    pass
konto = Konto()
konto.konto_nummer = "Girokonto Schmitz"
```

- **Nachteil:**
  Nur das Objekt `konto1` hat nun das Attribut `konto_nummer`. Die Idee der OO ist aber, dass in der Klasse die Funktionalität und damit der Aufbau aller Objekte beschrieben ist und somit alle Objekte diegleichen Attribute haben.

Attribute zu Objekten hinzufügen

- Eine Methode hat das aufrufende Objekt als Parameter (`self`).
- So kann also auch in einer Methode ein Attribut zu einem Objekt zugefügt werden.
- Beispiel: Methode in Konto-Klasse:

```
    def set_kontonummer(self, kntnr):
        self.kontonummer = kntnr
```

- Über self kann man dann natürlich auch auf ein Attribut zugreifen

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

- Empfehlung: Alle Attribute in einer Methode bei der Objekterzeugung erzeugen. Man sieht dann auf einen Blick, welche Attribute die Klasse besitzt (was ja der Sinn einer Klasse ist).

Die Methode `__init__()`

- Was passiert beim Erzeugen eines Objekts (`k1 = Konto()`)?
- Python erzeugt zuerst ein Objekt (ohne Attribute).
- Dann wird die Methode `__init__()` der Klasse für das Objekt aufgerufen, wenn die Methode existiert.
- Daher empfiehlt es sich, dort die Attribute zu erzeugen (und gegebenenfalls mit einem sinnvollen Anfangswert zu belegen).
- `__init__()` ist eine sog. „magische Methode“, weil sie per Namenskonvention automatisch von Python aufgerufen wird.

Klassenattribute

- „Normale“ Attribute existieren pro Objekt und haben pro Objekt einen anderen Wert, z.B. `konto_stand`.
- Ein *Klassenattribut* existiert pro Klasse einmal. Anders formuliert: Ist für alle Objekte der Klasse gleich.
- Ist damit unabhängig von einem konkreten Objekt.
- Kann über die Klasse oder ein Objekt angesprochen werden.
- Beispiel: Attribut anzahl, in dem man die Anzahl der erzeugten Objekte zählt.
- Definition in der Klasse außerhalb von Methode

Klassenattribute - Beispiel

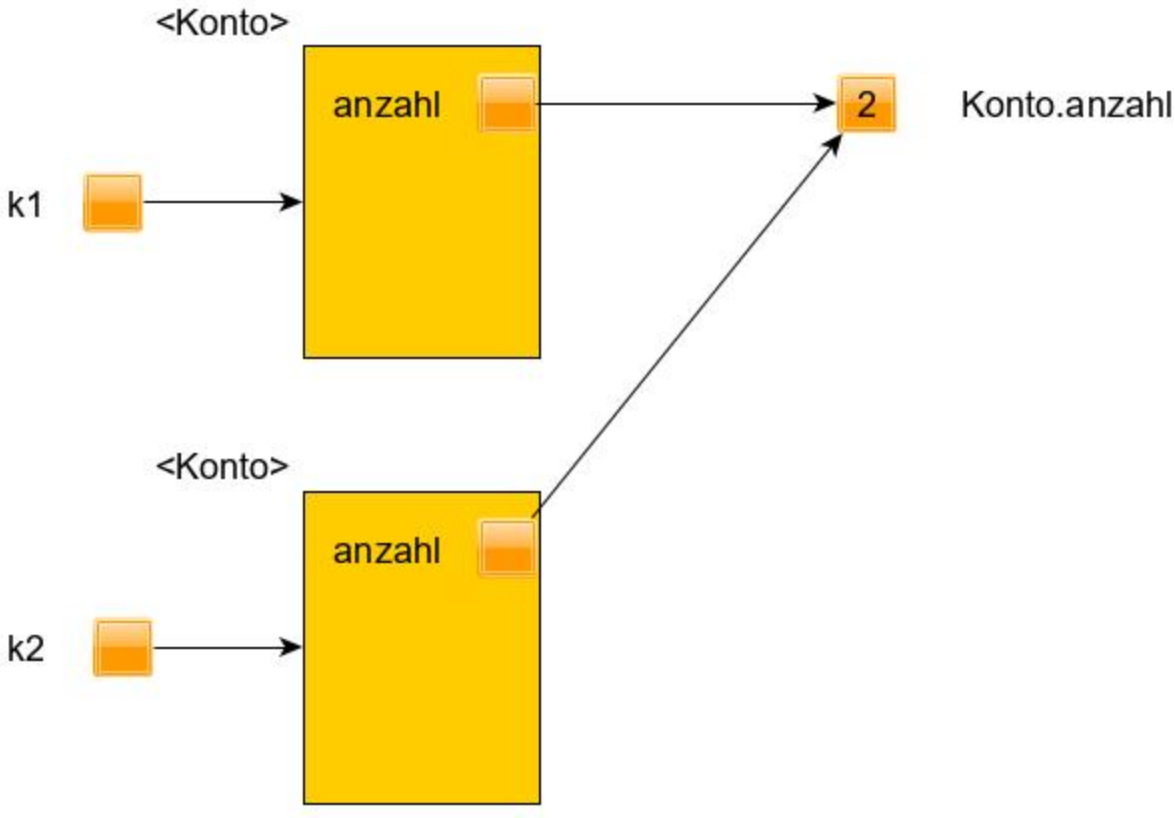
```
class Konto:
    anzahl = 0

    def __init__(self):
        Konto.anzahl += 1

if __name__ == "__main__":
    k1 = Konto()
    print(str(Konto.anzahl))

    k2 = Konto()
    print(str(Konto.anzahl))
```

Klassenattribute - Veranschaulichung



Statische Methoden

- Eine Klassenmethode ist nur über die Klasse, nicht über Objekte zugreifbar.
- Sie kann nur auf Klassenattribute zugreifen.
- Definition wie „normale“ Methode in Klasse aber ohne self-Parameter.
- Bsp.: ❗ `statische_methode.py`

Wdh.: Idee der OO

- Ein Objekt bietet eine Funktionalität an.
- Die interne Implementierung ist dem Anwender egal.
- Bsp: Liste: Die interne Implementierung ist für den Benutzer „weg-gekapselt“.
- Benutzt der Anwender nicht die internen Details des Objekts, kann die Implementierung des Objekts beliebig geändert werden.
- Ein Klasse besteht also aus:
  - einem öffentlichen Teil (öffentliche Schnittstelle) und
  - einem privaten Teil (interne Implementierung)
- Das gilt auch für das Konto-Beispiel, wobei dort die interne Implementierung (in dem Fall) trivial war.

Zugriffsrechte

- Es sollte also möglich sein, den Zugriff von außen auf die „internen“ Attribute und Methoden zu verhindern.
- Zugriffsrechte:
  - **private**: nur innerhalb von Klassenmethoden zugreifbar
    - Name beginnt mit zwei Unterstrichen
  - **public**: von außen und von innen zugreifbar
    - Name beginnt nicht mit Unterstrichen
  - **protected**: bei Vererbung wichtig (s. später)
    - Name beginnt mit einem Unterstrich
- Beispiel: Kontoklasse: Das Attribut kontostand sollte nicht von außen zugreifbar sein, sondern nur über Methoden verändert werden: ❗ `private.py`

OO und Wiederverwendung

- Auch die Verwendung von Klassen trägt zur einfacheren Wiederverwendung von (fremdem) Code bei.
- Man muss „nur“ die öffentliche Schnittstelle der Klasse kennen.

Properties 1

- Auf (interne) Attribute sollte möglichst nur innerhalb von Methoden zugegriffen werden.
- Das gilt auch für Attribute, auf die man von außen vielleicht doch zugreifen will.
- Klassisch: getter- und setter-Methoden.
- Warum? Weil man dort zusätzliches Verhalten implementieren könnte (aber nicht muss).
- Beispiel: ❗ `properties_1.py`, `properties_2.py`
- getter und setter sind Ballast, wenn eben nichts weiteres getan wird, als auf die Attribute zuzugreifen.

Properties 2

- Vermeidung dieses Balastes:
  - Zunächst keine getter und setter programmieren und direkt auf das Attribut zugreifen.
  - Braucht man doch getter und setter, so schreibt man diese und definiert in der Klasse properties: `kontoinhaber = property(get_kontoinhaber, set_kontoinhaber)`
- Der Zugriff „von außen“ bleibt dann unverändert.
- Beispiel: ❗ `properties_3.py`
- Hinweis: Wenn möglich sollte man den Zugriff auf Attribute vermeiden – auch über getter und setter (Setzen nur im Konstruktor).
  - Bsp: Kontostand
  - Bsp: Kontoinhaber

Die Methoden `__str__()` und `__repr__()`

- Darstellung eines Objekts als String
- Wird für ein Objekt `k` die Funktion `str` aufgerufen (`str(k)`) sucht der Interpreter nach einer Methode `__str__()` der Klasse und ruft diese auf (`k.__str__()`)
- Analog `repr(k)`
- Besitzt eine Klasse nur `__repr__()`, so wird immer diese verwendet.
- `str`: Darstellung für den Endanwender
- `repr`: Stringrepräsentation im Sinne von `eval(repr(k))`
- ❗ `repr.py`

Operator Overloading

- „Magische Methoden“: Werden von Python „automatisch“ aufgerufen, z.B. `__init__()`, `__str__()`
- Auch für die Operatoren (z.B. `+`) gibt es magische Methoden.
- Das nennt man Operator-Overloading
- Magische Methoden: s. Seite 235
- `x + y`: Aufruf von `x.__add__(y)`
- Was tun, wenn man folgendes möchte: `5 + k` (`k` referenziert ein Konto-Objekt)?
- `5` ist ein `int` und `int` hat keine `+`-Methode zur Addition mit Konto
- Lsg: In der Klasse Konto eine Methode `__radd__()` definieren.
- `5 + k` führt zu: `k.__radd__(5)`

Beispiel Operator Overloading

- ❗ `zeitspanne.py`
- Klasse repräsentiert eine Zeitspanne, die in Stunden und Minuten angegeben wird.
- Die interne Repräsentation besteht aus den Minuten
- Bei der Addition werden also die Minuten addiert.
- Addition einer Zahl (`int`): Muss in `add` abgefragt werden.
- Weiterer Anwendungsfall von `__radd__()`

Magische Methoden `hash` und `equals`

- Erinnerung: Will man Objekte einer eigenen Klasse als Schlüssel eines Dictionaries verwenden, müssen die Methoden sinnvoll überschrieben werden (s. `hashable.py` ❗):
  - Hash: `hash()` : Gibt ein Integer zurück.
  - Equals: `eq()` : Vergleicht ein Objekt mit einem anderen.
- Es sollte gelten:
  - Wenn o1 == o2, dann o1.*hash*() == o2.*hash*()
  - Das umgekehrte muss nicht unbedingt gelten.

Hash und Equals 2

- Es sollte nicht möglich sein, ein bestehendes Objekt so zu ändern, dass sich der Hash-Code ändert. Man findet dann nämlich das Objekt nicht mehr in dem Dictionary.
- Das ist der Grund, warum bei den eingebauten Typen nur die „unveränderlichen“ Typen als Schlüssel erlaubt sind.
- Umgekehrt ist das ein Grund, warum „wesentliche“ Attribute nicht von außen zugreifbar sein sollen.
- Soll eine Klasse nicht als Schlüssel verwendet werden können:

```
class NotHashable:
    __hash__ = None
```

- (siehe ❗ `no_hash.py`)

Iteratoren

- Wenn es möglich sein soll, mit einer for-Schleife über ein Objekt einer Klasse schleifen zu können, muss die Klasse die magische Methode `__iter__()` implementieren, die einen Iterator zurückgibt.
- Ein Iterator ist ein Objekt, das
  - die Methode *iter*() implementiert
  - und eine Methode *next*(),
    - die das nächste Element zurückgibt
    - und eine StopIteration-Exception wirft, wenn kein Element mehr verfügbar
- s. ❗ `iterator.py`

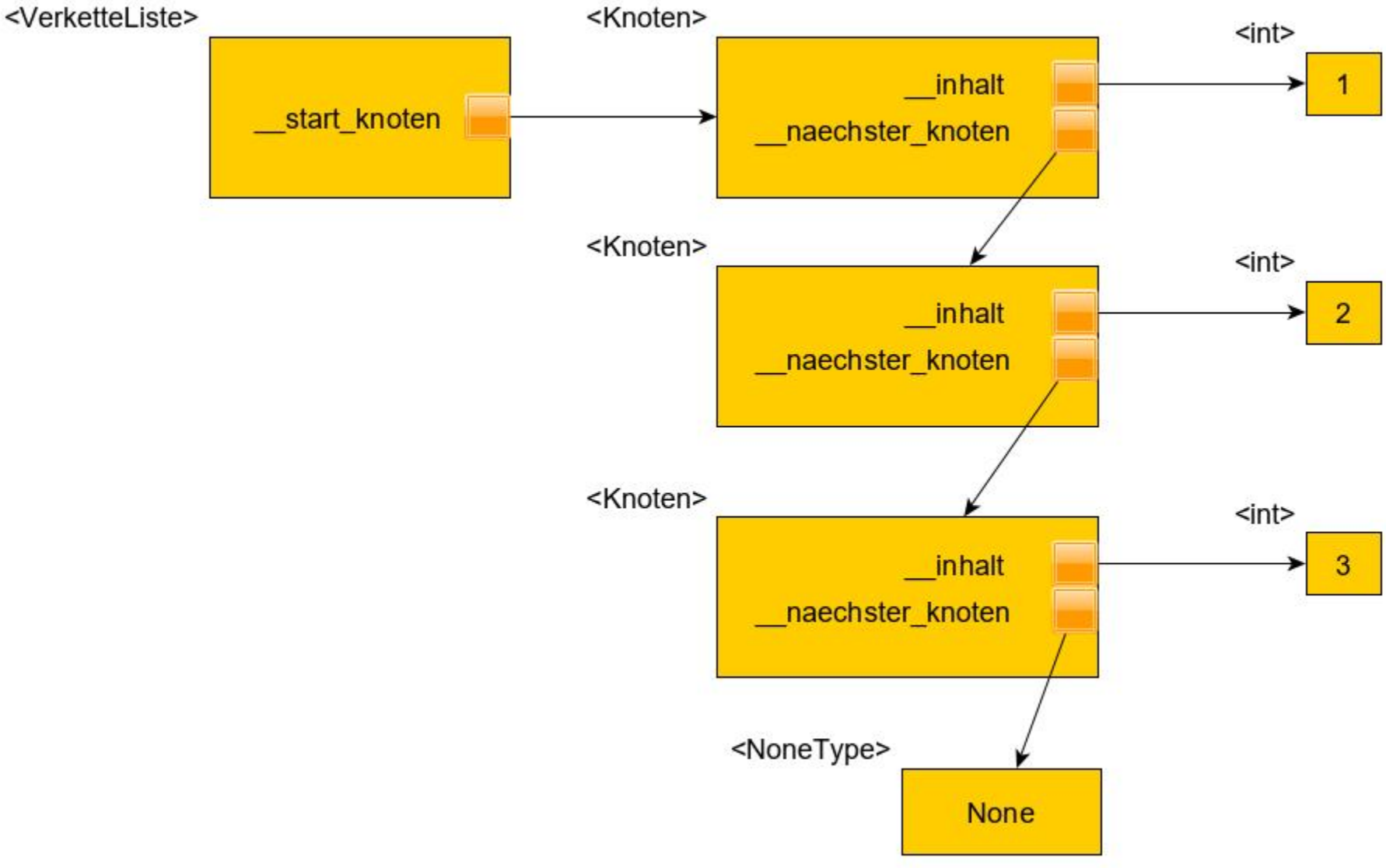
Verkettete Liste

- Implementierung einer eigenen Listenklasse ohne Verwendung von
  - List
  - Tupel
  - Dictionary
- Stichwort: „Verkettete Liste“
- Benötigt wird eine „Hilfsklasse“: `Knoten` mit zwei Attributen
  - `inhalt`: Das Objekt, das in der Liste in diesem Knoten verwaltet wird
  - `naechster_knoten`: Knoten, der das nächste Element der Liste enthält
- Die Klasse Liste hat als Attribut dann (zumindest) den `startknoten` der Liste, also den Knoten, der das erste Element der Liste enthält.

Übung Verkettete Liste 2

- Zu Beginn ist die Liste leer. Das Attribut `__startknoten` wird also auf `None` gesetzt.
- Einfügen eines neuen Elementes in die Liste:
  - Erzeugung eines neuen Objekts vom Typ `Knoten`.
  - Das Inhaltsattributs des neuen Knotens wird auf das einzufügende Element gesetzt.
  - Der Nachfolgeknoten des neuen Knotens ist `None`
  - Der Nachfolgeknoten des bisher letzten Knotens in der Liste wird auf den neuen Knoten gesetzt

Illustration Verkettete Liste



Referenzen

- [Ste] Ralph Steyer: Programmierung Grundlagen, Herdt-Verlag
- [Kle] Bernd Klein: Einführung in Python 3, Hanser-Verlag
- [Kof] Kofler: Python – Der Grundkurs, Rheinwerk Computing
- [EK] Johannes Ernesti, Peter Kaiser: Python 3 – Das umfassende Handbuch, Rheinwerk Computing
- [Mat] Eric Matthes: Python Crashkurs – Eine praktische, projektbasierte Programmiereinführung, dpunkt.verlag
- [Swe] Sweigart: Eigene Spiele programmieren: Python lernen