

# Objektorientiertes Programmieren II

Symbolische Programmiersprache

---

Benjamin Roth – Folien von Annemarie Friedrich  
Wintersemester 2017/2018

Centrum für Informations- und Sprachverarbeitung  
LMU München

## Recap: Software-Objekte repräsentieren Real-life-Objekte

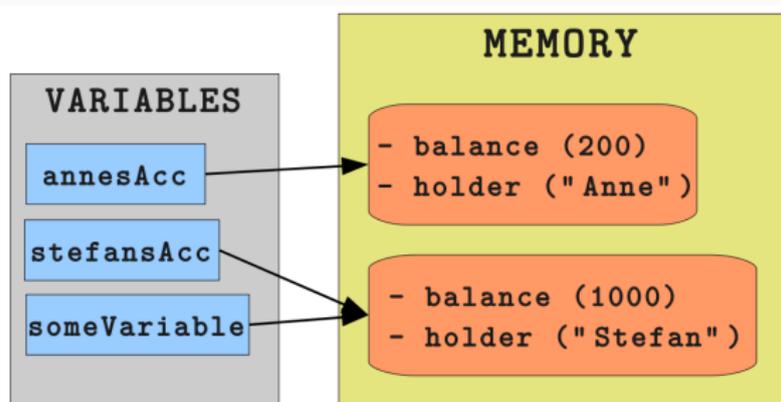
Attributes \ Object	annesAccount	stefansAccount
<b>number</b>	1	2
<b>holder</b>	'Anne'	'Stefan'
<b>balance</b>	200	1000

### Attribute

- beschreiben den *Zustand* des Objekts
- enthalten die *Daten* eines Objekts
- können sich im Laufe der Zeit verändern



## Recap: Zugriff auf Attribute mit der dot notation



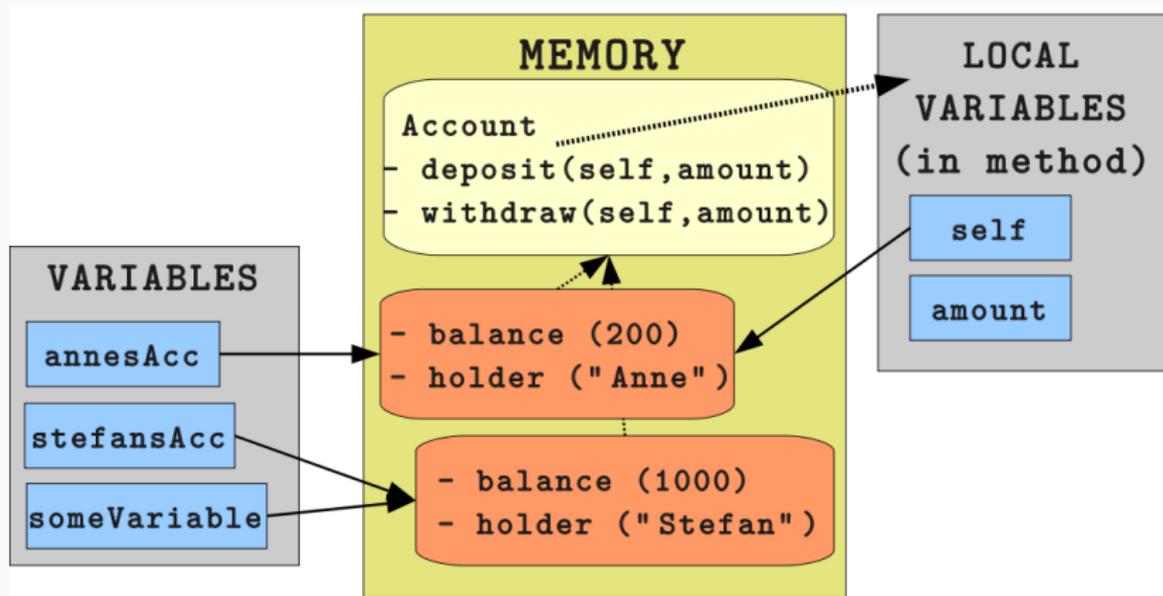
```
1 annesAcc = Account()
2 stefansAcc = Account()
3 annesAcc.balance = 200
4 stefansAcc.holder = "Stefan"
5 someVariable = stefansAcc
6 print(someVariable.holder)
7 someVariable.balance = 50
8 print(stefansAcc.balance)
```

## Recap: Methoden manipulieren die Daten eines Objekts

<b>Account</b>
+number +holder -balance
+__init__(self,num,holder) +__str__(self) +deposit(self,amount) +withdraw(self,amount)

```
1 class Account:
2     def __init__(self, num, holder):
3         self.num = num
4         self.holder = holder
5         self.balance = 0
6     def deposit(self, amount):
7         self.balance += amount
8     def withdraw(self, amount):
9         if self.balance < amount:
10            amount = self.balance
11            self.balance -= amount
12            return amount
13     def __str__(self):
14         return "[Account " + self.num \
15             + " " + self.holder + " " \
16             + self.balance + "]"
```

# Recap: Methoden werden "auf einem Objekt" aufgerufen



```
1 annesAcc.deposit(200)
2 stefansAcc.deposit(1000)
3 someVariable.withdraw(300)
```

## Instanzmethode

Das Objekt, auf dem die Methode aufgerufen wird, wird dem Parameter `self` zugewiesen.

## Recap: Konstruktor / Initialisierungsmethode

- wird gleich nach Erzeugen eines neuen Objekts aufgerufen (`self` zeigt auf das neue Objekt)
- `num` und `holder` sind lokale Variablen der Methode
- `self.num` und `self.holder` sind Attribute des Objekts
- **Verschiedene Namespaces  $\Rightarrow$  verschiedene Variablen!**
- $\rightarrow$  Tafel

```
1 class Account:
2     # Constructor
3     def __init__(self, num, holder):
4         self.num = num
5         self.holder = holder
6         self.balance = 0
7
8 annesAcc = Account(1, "Anne")
9 stefansAcc = Account(2, "Stefan")
```

# Python Modules

- **Modules** sind einfach Dateien mit Python-Code, können Funktionen, Variablen, Klassen oder ausführbaren Code definieren.
- Module gruppieren zusammengehörigen Code → Verständlichkeit
- Definition von Klasse / Funktion → Python erstellt Funktions-/Klassenobjekt
- Module sind auch Objekte, beinhalten Referenzen auf die Funktions-/Klassenobjekte, die das Modul definiert. Wir können diese Funktionen/Klassen in anderen Modulen (=Python-Dateien) **importieren**.
- Modulname = Dateiname ohne `.py`

```
1  # import all functions / classes from <modulename>
2  import modulename
3
4  # import specific function
5  from modulename import somefunction
6
7  # import specific class
8  from modulename import someclass
```

# Python Modules

- Wenn ein Modul importiert wird, sind dessen Funktionen/Klassen verfügbar.
- Ein Modul ausführen: `python3 someModule.py` (or F5 in IDLE)
- `someModule` kann auch Code aus anderen Dateien importieren.
- `if`-Statement (s.u.) checkt, ob das Modul als *main module* ausgeführt wird. ⇒ Tests für das Modul hierhin schreiben, beim Ausführen ist nur das *main module* relevant!

```
1  # imports
2
3  # some more function / class definitions
4
5  # main
6  if __name__ == "__main__":
7      # this code is executed when running THIS module
```

[http://www.tutorialspoint.com/python/python\\_modules.htm](http://www.tutorialspoint.com/python/python_modules.htm)

# Projektstrukturierung

- Klassen in separate Dateien.
- Klassendefinition z.B. in *accounts.py*
- Klassen in der Haupt-Anwendung importieren
- `from modulename import classname`
  - *modulename* = Dateiname, in dem die Klasse definiert ist (ohne *.py*)
  - *classname* = Name der Klasse

```
1  from accounts import Account
2
3  if __name__ == "__main__":
4      annesAcc = Account()
5      annesAcc.balance = 200
```

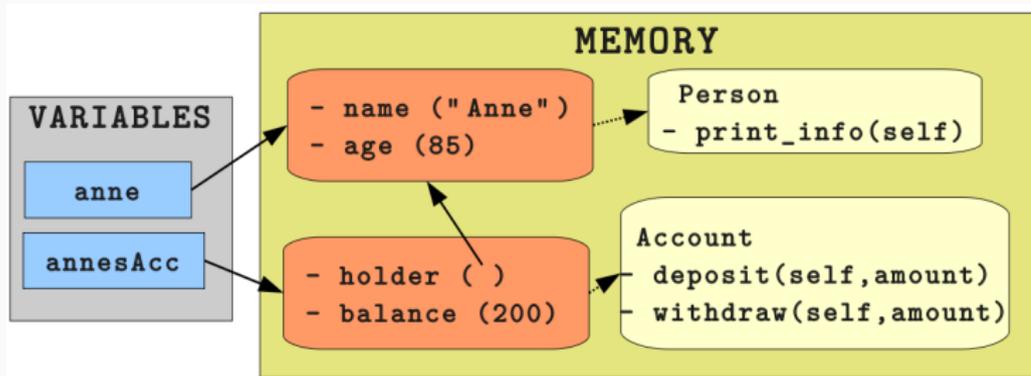
- Werte in Python haben *Typen*:
  - 1.5 hat Typ *float*
  - 'Stefan' hat Typ *str*
  - Der Typ des Instanzobjekts `stefansAcc` ist die Klasse, von dem es erstellt wurde

```
1 >>> stefansAcc = Account(2, "Stefan")
2 >>> type(stefansAcc)
3 <class '__main__.Account'>
```

- **Komposition:** Katze - Bein. Das Bein existiert nur, wenn die ganze Katze existiert.
- **Aggregation:** Vorlesung Python - Student. Der Student existiert auch, wenn es die Vorlesung nicht gibt.

# Komposition/Aggregation

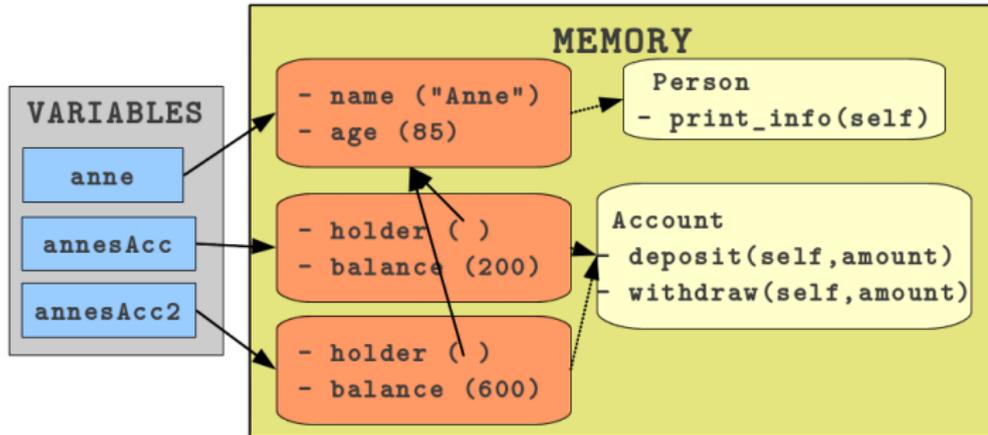
- Die Attribute eines Objekts können irgendeinen Typ haben
- Sie können auch selbst (komplexe) Objekte sein
- **Komposition** = komplexe Objekte werden aus mehreren Objekten zusammengebaut, die "enthaltenen" Objekte existieren nur innerhalb des komplexen Objekts
- **Aggregation** = keine exklusive Zugehörigkeit impliziert
- Zugriff mit *dot notation*: `annesAcc.holder.name`
- Diese Aufrufe nicht zu lang machen! (Lesbarkeit und Sicherheit)



# Aggregation: Beispiel

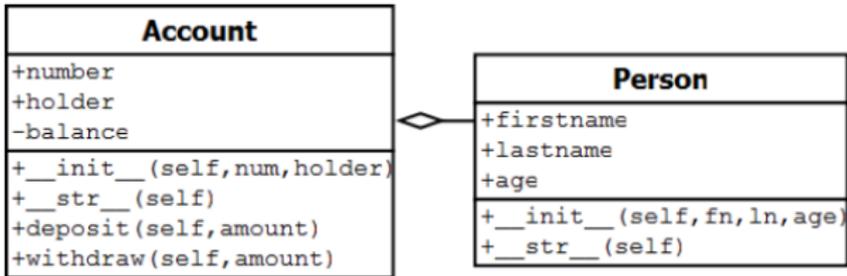
```
1 class Person:
2     def __init__(self, f, l, a):
3         self.firstname = f
4         self.lastname = l
5         self.age = a
6
7 class Account:
8     def __init__(self, person, num):
9         self.holder = person
10        self.num = num
11        self.balance = 0
12    def deposit(self, amount):
13        self.balance += amount
14
15 anne = Person("Anne", "Friedrich", 95)
16 annesAcc = Account(anne, 1)
17 annesAcc2 = Account(anne, 2)
```

# Aggregation: Shared References



- Aufpassen, wohin die Attribute zeigen:
- `annesAcc.holder.age += 1` ändert auch `annesAcc2.holder.age`
- Hier ok, aber aufpassen, damit keine Bugs erzeugt werden!

# Aggregation: UML Diagram



# References



Mark Lutz: *Learning Python*, Part VI, 4th edition, O'Reilly, 2009.



Michael Dawson: *Python Programming for the Absolute Beginner*, Chapters 8 & 9, 3rd edition, Course Technology PTR, 2010.