

Objektorientiertes Programmieren I

Symbolische Programmiersprache

Florian Fink – Folien von Annemarie Friedrich
Wintersemester 2020/2021

Centrum für Informations- und Sprachverarbeitung
LMU München

Imperatives Paradigma

Erst tu dies, dann tu das.

- Kontrollstrukturen definieren die Reihenfolge, in der die “Rechenschritte” (Programmschritte) ausgeführt werden.
 - Was ist ein Rechenschritt in Python?
 - Welche Kontrollstrukturen kennen Sie?
- Zustand des Programms ändert sich als eine Funktion der Zeit.
- Befehle können in Prozeduren (Funktionen) gruppiert werden.

Imperatives / Prozedurales Programmieren

```
1  def cels_to_fahr(c):
2      f = c*1.8 + 32
3      return f
4
5  def fahr_to_cels(f):
6      c = (f - 32) * 5/9
7      return c
8
9  print("Enter degrees.")
10 d = int(input(">>"))
11
12 print("C to F (1) or F to C (2)?")
13 option = None
14
15 while not (option == 1 or option == 2):
16     option = int(input(">>"))
```

```
1  if option == 1:  
2      f = cels_to_fahr(d)  
3      print(d, "C are", f, "F")  
4  else:  
5      c = fahr_to_cels(d)  
6      print(d, "F are", c, "C")
```

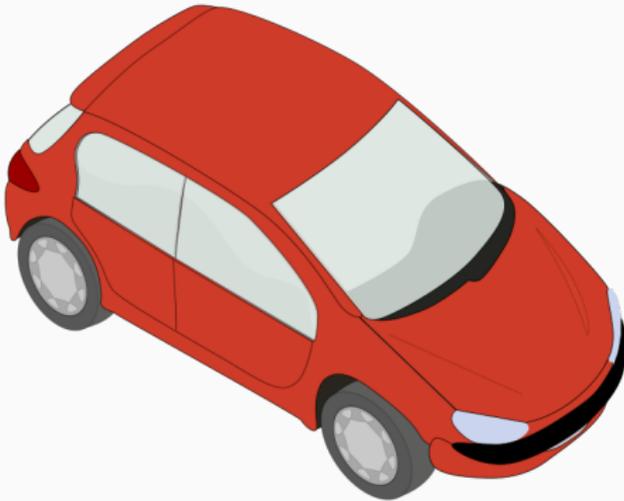
High-level-Überblick

- “Send messages between objects to simulate the temporal evolution of a set of real world phenomena.”
- Hilft die **Information** für ein zu modellierendes Phänomen zu gruppieren, und die dazugehörige **Funktionalität** zu organisieren.

- **Klassen** beschreiben Konzepte (Daten oder Operationen) des Anwendungsgebiets
- **Methoden** (= Prozeduren) sind an diese angegliedert
- **Instanzen** konkrete Realisierungen von Klassen (Klasseninstanzen)
- Beispiele: Java, C++, Smalltalk, Python, ...

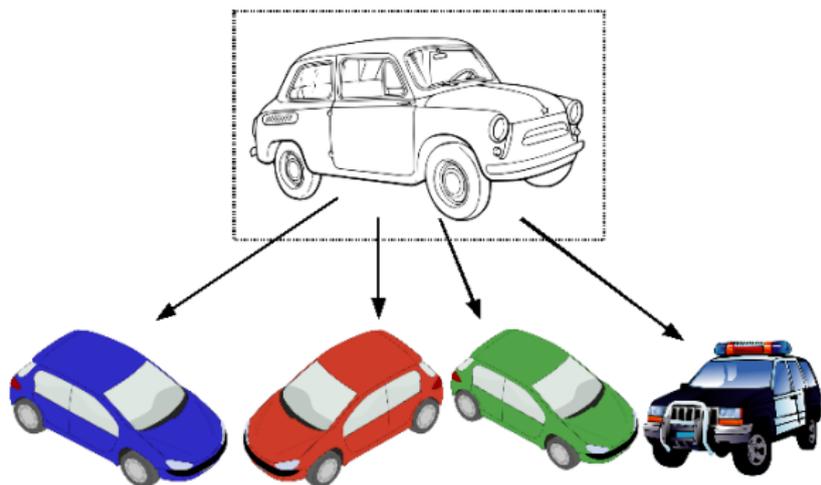
Objekte

- Real-life Objekte werden als **Software-Objekte** repräsentiert.
- Objekte kombinieren Charakteristika (**Attribute**) und Verhaltensweisen (**Methoden**).

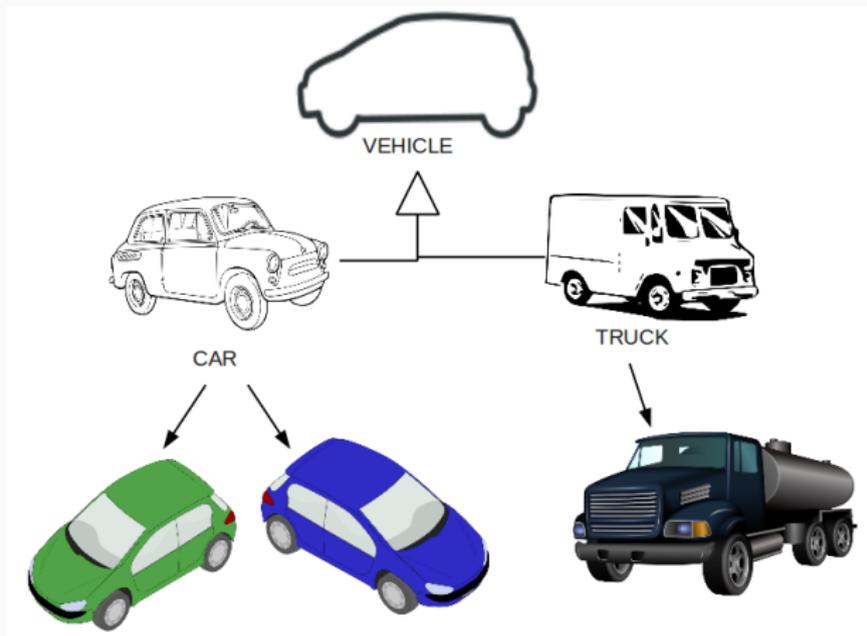


Car
ATTRIBUTES: make model color wheels seats autobody motor
METHODS: start drive stop honk

Klassen = Baupläne



- **Klassen** sind Baupläne/Designs für **Objekte**.
- Objekte mit Hilfe von Klassen erstellen: Wir **instanzieren** Objekte.
- Objekte sind **Instanzen** einer Klasse.
- Objekte einer Klasse haben die gleiche grundlegende Struktur, können sich aber in verschiedenen Aspekten unterscheiden.



- Verschiedene Objekte haben teilweise dasselbe Verhalten / dieselben Charakteristika.
- Vererbung → Code-Dopplungen vermeiden.

Details ...

... und Umsetzung in Python.

Software-Objekte repräsentieren Real-life-Objekte

Attribute \ Instanzen	annesAccount	stefansAccount
number	1	2
holder	'Anne'	'Stefan'
balance	200	1000

Attribute

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



Klassen = Baupläne für Objekte

```
1 class Account:
2     ''' a class for objects
3     representing an account '''
4     pass # skip
5
6 # Main part of the program
7 if __name__ == "__main__":
8     # Creating objects
9     annesAcc = Account()
10    stefansAcc = Account()
11    # Assigning attributes
12    annesAcc.holder = "Anne"
13    annesAcc.balance = 200
14    stefansAcc.holder = "Stefan"
15    stefansAcc.balance = 1000
16    # Accessing attributes
17    print(annesAcc.balance)
```

- Klassennamen beginnen mit Großbuchstaben
- Objekte werden erstellt, in dem "die Klasse aufgerufen wird"
- Zuweisung von / Zugriff auf Attribute und Methoden: *dot notation*

Methoden = Funktionen, die zu einer Klasse gehören

```
1 class Account:
2     # METHODS
3     def deposit(self, amount):
4         self.balance += amount
5
6 if __name__ == "__main__":
7     annesAcc = Account()
8     annesAcc.balance = 200
9     annesAcc.deposit(500)
```

Instanzmethoden

- operieren auf Objekten, die von dieser Klasse erstellt wurden
- Code manipuliert die Attribute des Objekts oder erlaubt den Zugriff auf diese
- Erster Parameter: `self` (Konvention)

Methoden = Funktionen, die zu einer Klasse gehören

```
1 class Account:
2     # METHODS
3     def deposit(self, amount):
4         self.balance += amount
5
6 if __name__ == "__main__":
7     annesAcc = Account()
8     annesAcc.balance = 200
9     annesAcc.deposit(500)
```

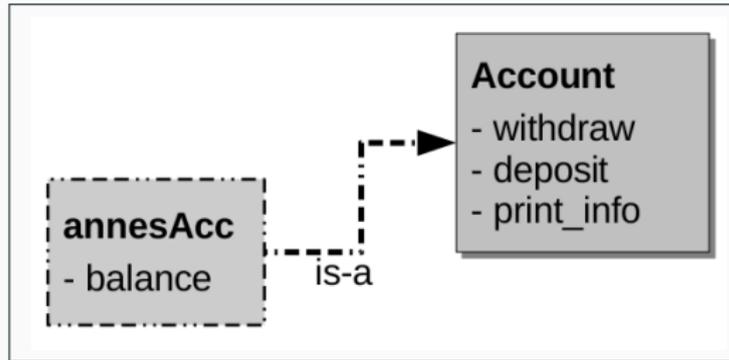
Instanzmethode

- wenn diese “von einem Objekt aus” aufgerufen werden (Zeile 9):
 - entsprechende Methode wird in der Klasse, von der das Objekt erstellt wurde, aufgerufen
 - das Objekt wird automatisch dem Parameter `self` zugewiesen
 - Zeile 9 ist äquivalent mit: `Account.deposit(annesAccount, 500)`

Methoden = Funktionen, die zu einer Klasse gehören

```
1  class Account:
2      # METHODS
3      def withdraw(self, amount):
4          self.balance -= amount
5      def deposit(self, amount):
6          self.balance += amount
7      def print_info(self):
8          print("Balance:", self.balance)
9
10  if __name__ == "__main__":
11      annesAcc = Account()
12      annesAcc.balance = 200
13      annesAcc.deposit(500)
14      annesAcc.withdraw(20)
15      annesAcc.print_info()
```

Objekte sind mit ihren Klassen “verlinkt”



- `annesAcc.deposit(500)`
- Python fängt im Objekt an, die Methode zu suchen (technisch gesehen können Methoden auch für individuelle Objekte definiert werden – praktisch werden sie immer in Klassen definiert*)
- Methode wird in Klasse, von der das Objekt erstellt wurde, gesucht

*Klassen sind auch Objekte in Python: aber dazu kommen wir später. ©

Attribute müssen existieren, wenn auf sie zugegriffen wird

```
1 class Account:
2     def print_info(self):
3         print("Balance:", self.balance)
4
5 if __name__ == "__main__":
6     stefansAcc = Account()
7     stefansAcc.print_info()
```

Warum wirft dieser Code einen Fehler?

Initialisierung / Konstruktor

```
1 class Account:
2     # CONSTRUCTOR
3     def __init__(self, num, person):
4         self.balance = 0
5         self.number = num
6         self.holder = person
7     # METHODS
8     ...
9
10 if __name__ == "__main__":
11     annesAcc = Account(1, "Anne")
12     stefansAcc = Account(2, "Stefan")
```

- `__init__(self)` wird automatisch nach Erstellen des Objekts aufgerufen
- **Konstruktor** / **Initialisierungs**-Methode weist Attributen des Objekts initiale / Default-Werte zu

```
1 annesAcc = Account(1, "Anne")
```

1. Neues Objekt der Klasse `Account` wird erstellt & und der Variable `annesAcc` zugewiesen
2. Die Initialisierungsmethode von `Account` wird aufgerufen
Variable `self` zeigt dabei auf das neue Objekt
technisch: `Account.__init__(annesAcc, 1, "Anne")`
3. In der Initialisierungsmethode wird das Objekt **initialisiert**
(Attribute werden auf die gegebenen oder auf Default-Werte gesetzt).

Regeln für gutes Klassendesign

1. Wie kann ich den Zustand meines Objekts beschreiben?
⇒ *Attribute*.
2. Was weiß ich über mein Objekt beim oder vor dessen Erstellen?
⇒ *Initialisierungsmethode*.
3. Welche Operationen, die den Zustand des Objekts ändern, werden auf dieses (im Lauf der Zeit) angewandt?
⇒ *Instanzmethode*.

Was sind die Antworten für das Konto-Beispiel?

Attribute nur in Instanzmethoden manipulieren

Schlechter Stil: `stefansAcc.balance = 1000`

Datenkapselung (data encapsulation)

- Attribute eines Objekts sollten vor Manipulationen “von außen” (= von Code, der das Objekt benutzt) “versteckt” sein
- Attribute eines Objekts sollten nur von Code, der *innerhalb* der Klasse definiert ist, modifiziert werden
- Dies stellt sicher, dass der Zustand des Objekts immer valide ist

Beispiel

- Konto mit Auflage: Kontostand darf nicht negativ sein
- Stefans Kontostand ist €1000, er will €1500 abheben
- Schalterbeamter zahlt aus und setzt Stefans Kontostand auf -€500 ⇒ Bankmanager sauer

Bankmanager glücklich machen

```
1 class Account:
2     # METHODS
3     def withdraw(self, amount):
4         if amount > self.balance:
5             amount = self.balance
6             self.balance -= amount
7         return amount
8     ...
9
10 if __name__ == "__main__":
11     stefansAcc = Account(2, "Stefan")
12     stefansAcc.deposit(1000)
13     cash = stefansAcc.withdraw(1500)
14     print("Oh no, I only got:", cash)
```

Attribut-Werte ändern: Setter-Methoden

```
1 class Account:
2     def set_holder(self, person):
3         self.holder = person
4
5 if __name__ == "__main__":
6     stefansAcc = Account(2, "Stefan")
7     stefansAcc.deposit(1000)
8     stefansAcc.set_holder("Andrea")
```

- Für jedes Attribut, das von außen geändert werden muss, eine **Setter-Methode** bereitstellen
- Erlaubt Validierung

Beispiel für Validierung in einer Setter-Methode:

```
1 def set_holder(self, person):
2     if (not type(person) == str):
3         raise TypeError
4     if not re.match("\w+(\ \w+)*", person.strip()):
5         raise ValueError
6     self.holder = person
```

1. Attributen Werte nur in Instanzmethoden (Setter-Methoden) oder im Konstruktor zuweisen.
2. Die Werte von Attributen nur über Instanzmethoden modifizieren.
3. Zugriff auf (Lesen von) Attributwerten mit `print(stefansAcc.balance)` ist okay.

Hinweis - eine weitere Möglichkeit in Python: properties

String-Repräsentation eines Objekts

```
1 class Account:
2     def __str__(self):
3         res = "*** Account Info ***\n"
4         res += "Account ID:" + str(self.number) + "\n"
5         res += "Holder:" + self.holder + "\n"
6         res += "Balance: " + str(self.balance) + "\n"
7         return res
8
9 if __name__ == "__main__":
10     annesAcc = Account(1, "Anne")
11     annesAcc.deposit(200)
12     print(annesAcc)
```

- **Magic methods*** = (magische) Methoden, die von Python in bestimmten Situationen automatisch aufgerufen werden
- hier: wird eine String-Repräsentation des Objekts benötigt?
z.B. `print(annesAcc)` oder `str(annesAcc)`

Vererbung in Python

```
1 class Animal:
2     def make_sound(self):
3         print('ROAR')
4     # Uses the make_sound method of its parent.
5 class Lion(Animal):
6     pass
7     # Overrides the make_sound method of its parent.
8 class Duck(Animal):
9     def make_sound(self):
10        print('QUACK')
11
12 if __name__ == "__main__":
13     lion = Lion()
14     lion.make_sound() # prints ROAR
15     duck = Duck()
16     duck.make_sound() # prints QUACK
```

- Klassen representieren Konzepte (Daten mit Operationen)
- Instanzen sind konkrete Realisierungen von Klassen
- Instanzen werden über den Klassenkonstruktor erzeugt
- Methoden ermöglichen die Kapselung von Attributen
- Attribute von Instanzen nur im Konstruktor oder mit Setter-Methoden verändern
- Setter-Methoden (und Konstruktoren) können zur Datenvalidierung verwendet werden
- Lesender Zugriff auf Attribute ist okay (keine Getter-Methoden)
- Vererbung um Code-Duplizierungen zu vermeiden (besser: Komposition)

Fragen?

-  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.
-  http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html