Objektorientierte Programmierung

Ronja Düffel
WS2018/19

09. Oktober 2018

Überblick

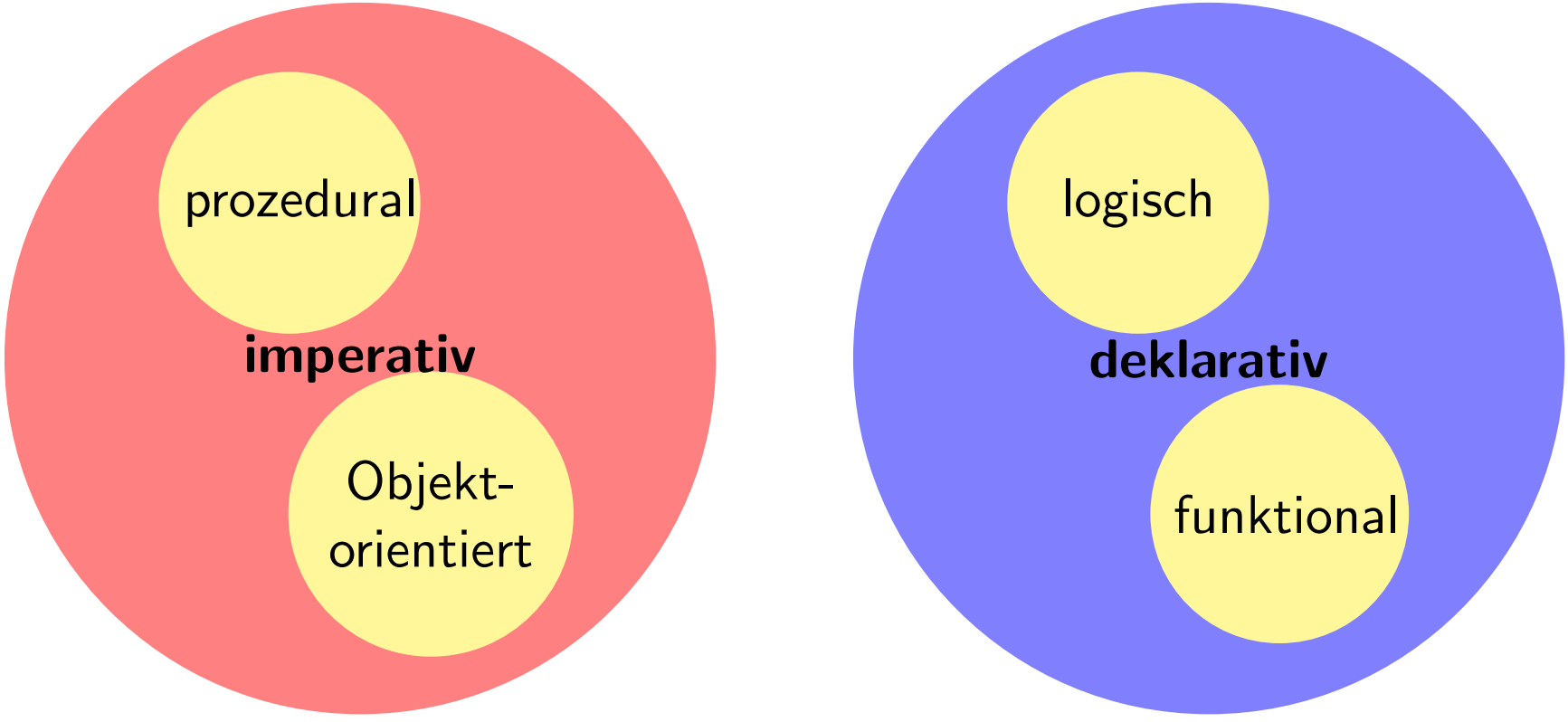
1. Was ist das?
2. Wie geht das?
3. Warum gibt es das?
4. Wie geht das in Python?

Überblick

1. Was ist das?
2. Wie geht das?
3. Warum gibt es das?
4. Wie geht das in Python?

Programmierparadigmen

Es gibt viele verschiedene höhere Programmiersprachen!



Was ist das?

- ein Programmierparadigma (Programmierstil)
- Art und Weise an ein Problem und dessen Lösung heranzugehen, es zu modellieren und somit auch zu programmieren
- bisher: **Prozedurale Programmierung**
  - Zerlegung in Variablen, Datenstrukturen und Funktionen
  - Funktionen operieren direkt auf Datenstrukturen
- Objektorientierung: Beschreibung eines Systems anhand des Zusammenspiels kooperierender Objekte

Objekte

- Objekte sind überall
- werden von uns als solche wahrgenommen
- Begriff eher unscharf $\Rightarrow$ kann auch abstrakter sein

In der realen Welt	OO-Programmierung
Zustand	Attribute
Verhalten	Methoden

Überblick

1 Was ist das?

2 Wie geht das?

3 Warum gibt es das?

4 Wie geht das in Python?

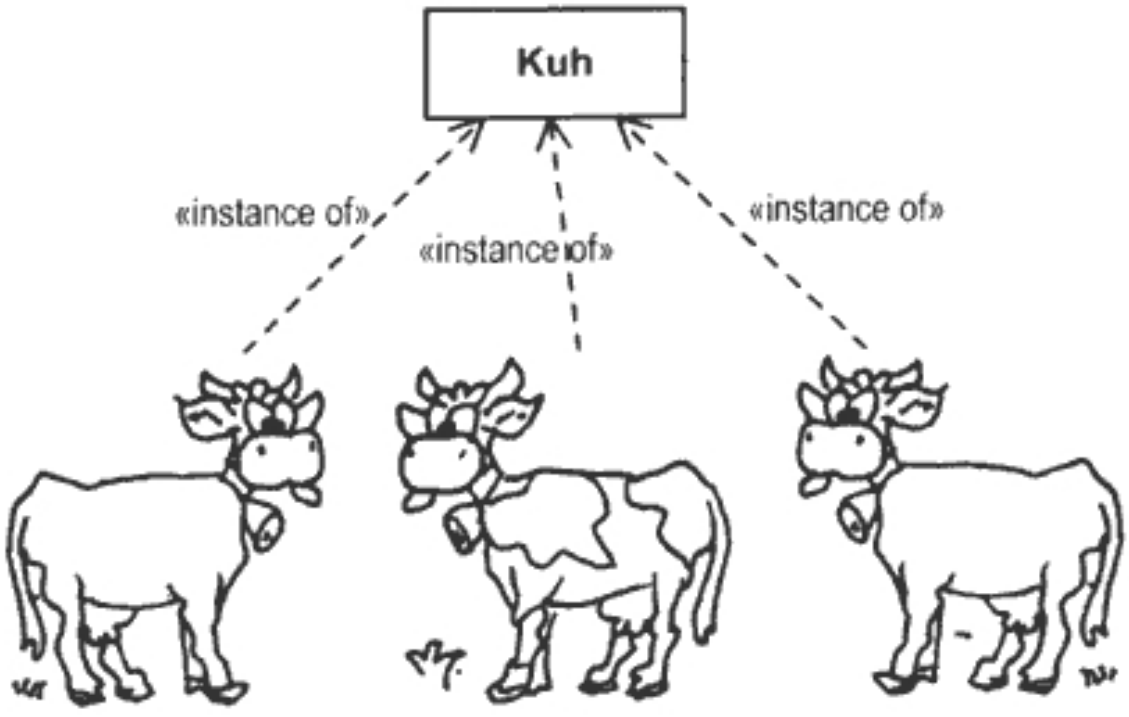
Objekte in OOP

- Zustand gespeichert in Attributwerten
- Verhalten festgelegt durch Methoden
- Interaktion mit anderen Objekten durch Methoden
- Zustand ist versteckt, nur über Methoden erreichbar
- Methoden definieren Schnittstelle, über die andere Objekte mit dem Objekt interagieren. (**Datenkapselung)**

Klassen und Objekte

- Klasse
  - definiert für eine Menge von Objekten deren Struktur (**Attribute**), Verhalten(**Methoden**) und Beziehungen
  - Bauplan für Objekt
  - Definition aller Attribute und Methoden
  - Besitzt Mechanismus zur Erzeugung eines Objekts
- Klasse allein macht noch nichts
- Objekt $\rightarrow$ ist konkrete Ausprägung (**Instanz**) der Klasse
- Jedem Objekt ist genau eine Klasse zugeordnet

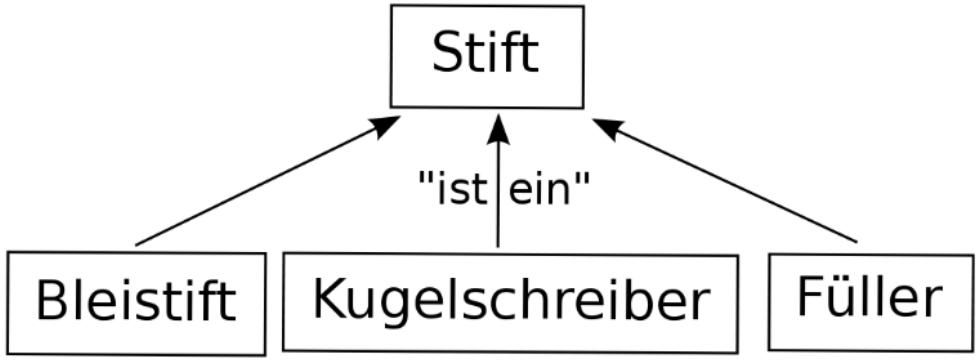
Objekte der Klasse Kuh



Klasse „Kuh“	**Objekt „Kuh Elsa“**
Name	Elsa Euter
Geburtsdatum	25. Mai 2015
Milchleistung	34l/Tag

Klassenhierarchie

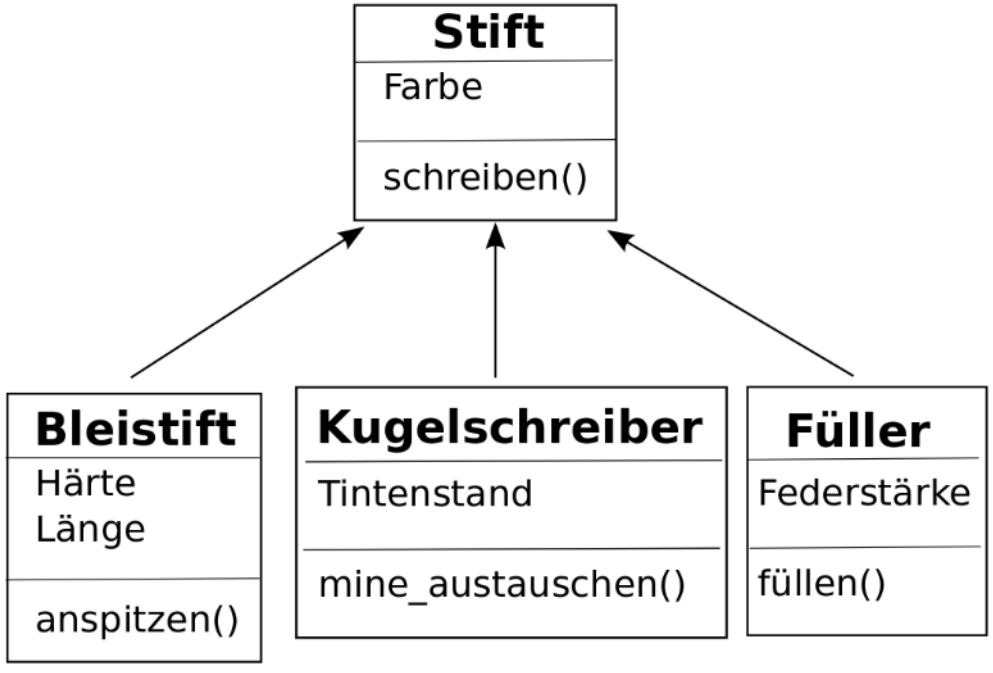
- Verschiedene Arten von Objekten haben häufig Gemeinsamkeiten
- “ist ein”-Beziehungen
- Beispiel:



- Superklasse ≡ Elternklasse ≡ Oberklasse ≡ Basisklasse
- Subklasse ≡ Kindklasse ≡ Unterklasse ≡ abgeleitete Klasse

Vererbung

- Kindklassen erben alle Attribute und Methoden von Elternklassen
- haben zusätzlich eigene Attribute und Methoden können Attribute und Methoden der Elternklasse *überschreiben*



abstrakte Klasse

- enthält nur leere Methoden
- kann keine Instanz erzeugen
- dient zur Zusammenfassung ähnlicher Klassen
- definiert gemeinsame Attribut- und Methodennamen
- zwingt alle Kindklassen Attribute und Methoden mit entsprechendem Namen zu haben

Überblick

1 Was ist das?

2 Wie geht das?

3 Warum gibt es das?

4 Wie geht das in Python?

Warum gibt es das?

Zunahme der Rechnerleistung

→ größere Programme

→ komplexere Software

→ größere Projekte

→ Modularität

Vorteil

- **Abstraktion:** Betrachtung der Objekte und ihrer Eigenschaften und Fähigkeiten, ohne Festlegung auf Implementierung
- **Datenkapselung:** Objekt interagiert nur über vordefinierte Methoden. Implementierung kann verändert werden, ohne dass ander Teile des Programms geändert werden müssen
- **Vererbung:** klarere Struktur und weniger Redundanz
- **Wiederverwendbarkeit:** Programme können einfacher erweitert und modifiziert werden. Klassen können auch in anderen Prorammen verwendet werden.

Nachteile

- **Formulierung:** natürliche Sprache hat keine feste Bindung von Substantiv (Objekt) und Verb (Methode).
- **Klassenhierarchie:** ist in der realen Wlt nicht immer so klar. (z.B. Kreis-Ellipse-Problem)
- **Transparenz:** Kontrollfluss nicht im Quelltext (besonders problematisch bei Parallelisierung)
- **Laufzeit- und Energieeffizienz:** OOP-Anwendungen benötigen häufig mehr Energie und längere Laufzeit

Überblick

1 Was ist das?

2 Wie geht das?

3 Warum gibt es das?

4 Wie geht das in Python?

Klassen in Python

- Klasse:

```
class KlassenName:
    def method1(self, ):
    def method2(self, ):
```

- Verwendung:
  - `obj1 = KlassenName()`
  - `obj1.method1()`

```
>>> class Person:
        pass

>>> a = Person()
>>> b = Person()
>>> a == b
False
```

```
>>> a
<__main__.Person object at 0x7fac6453eba8>
>>> b
<__main__.Person object at 0x7fac6453e550>
```

Konstruktor

- Konstruktor:
  erzeugt ein Objekt (Instanz) der Klasse
  ␣␣␣␣def __init__(self,):
  ␣␣␣␣␣␣␣␣. . .
- Beispiel:

```
1  class Konto:
2      # Konstruktor
3      def __init__(self, name, nummer):
4          self.inhaber = name
5          self.kontonummer = nummer
6          self.saldo = 0
7
8
9      def einzahlen(self, betrag):
10         self.saldo = self.saldo + betrag
```

Beispiel: Konto

```
1  from konto1 import Konto
2
3  k1 = Konto("Bob", 19836)
4  k2 = Konto("Alice", 39748)
5
6  k1.einzahlen(500)
7  k2.einzahlen(700)
8
9  print(k1.saldo, k1.inhaber, k1.kontonummer)
10 print(k2.saldo, k2.inhaber, k2.kontonummer)
```

```
500 Bob 19836
700 Alice 39748
>>>
```

Variablen/Attribute

- Klassenvariablen:
  - wird von allen Instanzen einer Klasse geteilt
  - mit `<KlassenName>.<AttributName>` innerhalb und außerhalb der Klasse erreichbar
- Objektvariable:
  - Existiert allein für dieses Objekt (Instanz der Klasse)
  - `<ObjektName>.<AttributName>` innerhalb der Klasse erreichbar (evtl. auch außerhalb).
- Destruktor:
  - löscht eine Instanz/Objekt der Klasse
  - nur notwendig, wenn beim Löschen tatsächlich etwas passieren muss

Beispiel: Klassenvariable

```
1  class Konto:
2      anzKont = 0 #Klassenvariable
3      # Konstruktor
4      def __init__(self, name, nummer):
5          self.inhaber = name
6          self.kontonummer = nummer
7          self.saldo = 0
8          Konto.anzKont = Konto.anzKont + 1
9
10
11     def einzahlen(self, betrag):
12         self.saldo = self.saldo + betrag
13
14     def __del__(self):
15         Konto.anzKont = Konto.anzKont - 1
```

Beispiel: Klassenvariable

```
1  from konto2 import Konto
2  print("Anzahl der Konten:", Konto.anzKont)
3
4  k1 = Konto("Bob", 19836)
5  k2 = Konto("Alice", 39748)
6
7  k1.einzahlen(500)
8  k2.einzahlen(700)
9
10 print(k1.saldo, k1.inhaber, k1.kontonummer)
11 print(k2.saldo, k2.inhaber, k2.kontonummer)
12 print("Anzahl der Konten:", Konto.anzKont)
13
14 del k1 #Bobs Konto löschen
15 print("Anzahl der Konten:", Konto.anzKont)
```

Beispiel: Klassenvariable Ausgabe

```
Anzahl der Konten: 0
500 Bob 19836
700 Alice 39748
Anzahl der Konten: 2
Anzahl der Konten: 1
>>>
```

public, protected, private

Name	Bezeichnung	Bedeutung
`name`	public	sowohl innerhalb einer Klasse, als auch von außen les- und schreibbar
`_name`	protected	von außen les- und schreibbar, Attribute und Methoden sollten aber nicht benutzt werden
`__name`	private	von außen weder sichtbar, noch nutzbar

Beispiel: private Attribute

```
1  class Konto:
2      anzKont = 0 #Klassenvariable
3      # Konstruktor
4      def __init__(self, name, nummer):
5          self.__inhaber = name
6          self.__kontonummer = nummer
7          self.__saldo = 0
8          Konto.anzKont = Konto.anzKont + 1
9
10
11     def einzahlen(self, betrag):
12         self.__saldo = self.__saldo + betrag
13
14     def __del__(self):
15         Konto.anzKont = Konto.anzKont - 1
```

Beispiel: private Attribute

```
1  from konto3 import Konto
2  print("Anzahl der Konten:", Konto.anzKont)
3
4  k1 = Konto("Bob", 19836)
5  k2 = Konto("Alice", 39748)
6
7  k1.einzahlen(500)
8  k2.einzahlen(700)
9
10 print(k1.__saldo, k1.__inhaber, k1.__kontonummer)
11 print(k2.__saldo, k2.__inhaber, k2.__kontonummer)
12 print("Anzahl der Konten:", Konto.anzKont)
13
14 del k1 #Bobs Konto löschen
15 print("Anzahl der Konten:", Konto.anzKont)
```

Beispiel: private Attribute

```
Anzahl der Konten: 0
Traceback (most recent call last):
  File "/home/ronja/Uni/Lernzentrum/Vorkurs/WS1819/Material/Folien/beikonto2.py"
, line 10, in <module>
    print(k1.__saldo, k1.__inhaber, k1.__kontonummer)
AttributeError: 'Konto' object has no attribute '__saldo'
>>>
```

Beispiel: Datenkapselung

```
1  class Konto:
2      anzKont = 0 #Klassenvariable
3      # Konstruktor
4      def __init__(self, name, nummer):
5          self.__inhaber = name
6          self.__kontonummer = nummer
7          self.__saldo = 0
8          Konto.anzKont = Konto.anzKont + 1
9      def einzahlen(self, betrag):
10         self.__saldo = self.__saldo + betrag
11     def getInhaber(self):
12         return(self.__inhaber)
13     def getSaldo(self):
14         return(self.__saldo)
15     def __del__(self):
16         Konto.anzKont = Konto.anzKont - 1
```

Beispiel: Datenkapselung

```
1  from konto4 import Konto
2  print("Anzahl der Konten:", Konto.anzKont)
3
4  k1 = Konto("Bob", 19836)
5  k2 = Konto("Alice", 39748)
6
7  k1.einzahlen(500)
8  k2.einzahlen(700)
9
10 print(k1.getSaldo(), k1.getInhaber())
11 print(k2.getSaldo(), k2.getInhaber())
12 print("Anzahl der Konten:", Konto.anzKont)
13
14 del k1 #Bobs Konto löschen
15 print("Anzahl der Konten:", Konto.anzKont)
```

Beispiel: Datenkapselung

```
Anzahl der Konten: 0
500 Bob
700 Alice
Anzahl der Konten: 2
Anzahl der Konten: 1
>>>
```

Prozedural vs Objektorientierung

prozedurale Programmierung	**objektorientierte Programmierung**
Ansammlung von Variablen, Datenstrukturen und Funktionen, bzw Unterprogrammen.	Datentypen (Klassen), welche Verhalten (Methoden) mit Daten (Attributen) verbinden.
Prozeduren oder Funktionen operieren direkt auf Datenstrukturen.	Instanz einer Klasse (Objekt) operiert auf seiner “eigenen” Datenstruktur.
Funktionen und Daten haben keinen Zusammenhalt.	Funktionen (Methoden) und Daten (Attribute) sind fest miteinander verbunden.

Einführungsveranstaltungen

- **Informatik:** 11.-12.10.2018 Beginn: Do 11:00 Uhr, Fr 12:00 Uhr
  Magnushörsaal, Robert-Mayer-Str. 11-15
- **Bioinformatik:** Fr, 12.10.2018, 12:00 Uhr
  Vor der Neuen Mensa, Campus Bockenheim
- **Wirtschaftsinformatik:** Do, 11.10.2018, 11:00 Uhr
  Hilbertraum (Raum 302), Robert-Mayer-Str. 6-8

Fragen?