



TECHNISCHE
UNIVERSITÄT
DRESDEN

OBJEKTORIENTIERTE PROGRAMMIERUNG IN PYTHON

Folien: C. Statz, D. Pataky, C. Knoll

Sprecher: S. Brückner

Dresden, WiSe 2013/14

- Wunsch: Wiederverwendbarkeit schon implementierter Funktionalität
- Copy & Paste?
 - Häufige Quelle von Fehlern
(nicht alles notwendige geändert)
 - Spätere Änderungen an vielen Stellen notwendig → Aufwand

→ Problem lässt sich mit verschiedenen **Paradigmen** angehen

- Was ist ein Programmierparadigma?
 - Satz von Regeln für formale und strukturelle Code-Gestaltung
- Wozu dient es?
 - Soll bei Erstellung von **gutem** Code unterstützen
 - Zum Teil bestimmte Herangehensweise erzwingen
- Kein Dogma!
 - Anwendung abhängig vom konkreten Problem (u. Geschmack)
 - Anders als z.B. Java erzwingt Python kein bestimmtes Paradigma
 - Kombinationen möglich

Prozedurale Programmierung

```
def quadrat_zahl(x):  
    return x**2  
quadrat_zahl(7) # -> 49
```

Modulare Programmierung

```
from IPython import embed  
from numpy import array  
import sympy as sp
```

- Beschreibung eines komplexen Systems Zusammenspiel von **Objekten**
- Objekte bestehen aus Daten **und** zugehörigen Funktionen („Methoden“)
- Objekte sind **Instanzen** einer **Klasse**
d.h. ein Objekt ist die konkrete Variable, die Klasse ist der Datentyp

→ Abstraktes Konzept

- Genug Theorie für ganzes Semester
- Hier: wichtigste Begriffe und Prinzipien klären

Ein Ball mit Attributen und Methoden

- Der Ball hat z.B. die Eigenschaften (Attribute)
 - Radius
 - Material
 - ...
 - (Substantive)
- Der Ball stellt auch gewisse Aktionen (Methoden) bereit
 - Ball werfen (x,y,z-Ebene)
 - Ball rollen (x,y-Ebene)
 - ...
 - (Verben)



Der Ball ist eine spezielle Kugel

```
class Kugel():  
    """Eine Klasse als Bauplan einer Objekts"""  
  
    def __init__(self, radius, mittelpunkt=(0,0,0)):  
        """Konstruktor-Methode"""  
        self.radius = radius # z.B. in cm  
        self.mittelpunkt = mittelpunkt # als (x,y,z)-Tupel  
  
    def berechne_volumen(self):  
        r = self.radius  
        return (4/3)*pi*(r**3)
```

Auch andere Bälle sind Kugeln:

```
# Definition der Klasse
class Kugel():
    def __init__(self, radius, mittelpunkt=(0,0,0)):
        ...

# Instanziierung der Klasse
fussball = Kugel(radius=21)
tennisball = Kugel(3)
flummi = Kugel(1)

print "Flummi-Radius:", flummi.radius # Zugriff auf Attribut
V = tennisball.berechne_volumen() # Zugriff auf Methode
```

- Klasse `Kugel` also nur der Bauplan
 - Instanziierung: Erstellung von konkreten Objekten nach dem Bauplan
 - Eigener Speicherbereich für jedes Objekt
- Attribut-Werte sind unabhängig voneinander (jede `Kugel`-Instanz hat eigenen Radius, Mittelpunkt, ...)
- Jedes Objekt hat eindeutige Speicher-Adresse (auslesbar mit `id(...)`)

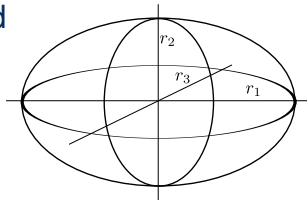
- Schaffung einer neuen Klasse basierend auf einer bestehenden
- Nur begrenzte Analogie zur biologischen Vererbung
- Typischer Fall: Vererbung vom Abstrakten zum Speziellen
- Darstellung: Basisklasse $\leftarrow$ Kindklasse („ $\leftarrow$ “ = „erbt von“)
- Bsp: Nahrungsmittel $\leftarrow$ Brot $\leftarrow$ Schwarzbrot
- Kindklasse hat alle Attribute/Methoden der Basisklasse
 - Können überschrieben werden
- Zusätzliche Attribute/Methoden in Kindklasse möglich

Wozu ist Vererbung gut?

- Gemeinsame Nutzung von Struktur und Code (Attribute und Methoden)
 - Reduziert Implementierungsaufwand
- Dokumentation der Ähnlichkeiten zwischen Klassen

Die Kugel ist ein spezieller Ellipsoid

```
class Ellipsoid():  
    def __init__(self, r1, r2, r3):  
        ...  
  
class Kugel(Ellipsoid):  
    def __init__(self, radius):  
        # Kugel ist ein Ellipsoid mit nur einem Radius  
        Ellipsoid.__init__(self, radius, radius, radius)
```



- Klasse `Kugel` wird hier von Klasse `Ellipsoid` abgeleitet
- Attribute und Methoden werden vererbt
- Konstruktor wird überschrieben, sodass er mit einem Radius auskommt.

```
class GeometrischesObjekt():
    ...
class Quader(GeometrischesObjekt):
    ...
class Ellipsoid(GeometrischesObjekt):
    ...
class Kugel(Ellipsoid):
    ...
tennisball = Kugel(3)
```

- Klasse `GeometrischesObjekt` ist hier Basisklasse (auch „Oberklasse“) aller weiteren Klassen
- Verdeutlichung mittels der python-Funktionen `isinstance(...)` und `issubclass(...)`

```
isinstance(tennisball, Kugel) # True
isinstance(tennisball, GeometrischesObjekt) # True
isinstance(tennisball, Quader) # False
issubclass(Kugel, Ellipsoid) # True
issubclass(Kugel, Quader) # False
```

Python-Besonderheit (1): `self`

- Methode = eine zu einem Objekt gehörende Funktion
 - Bei Ausführung einer Methode muss bekannt sein, zu welchem konkreten Objekt diese Methode gehört
- Übergabe der Instanz als implizites erstes Argument
- Bezeichnung: `self` (Konvention)

```
class KlasseA():  
    def m1(self):  
        print id(self)  
  
    def m1(x): # !! self vergessen  
        print x
```

```
k = KlasseA()  
k.m1() # kein explizites Argument (aber implizit das k)  
print id(k) # Uebereinstimmung
```

```
k.m2(23) # Fehler (Zu viele Argumente uebergeben)  
k.m2() # kein Fehler (= print k)
```

Besonderheit (2): alles ist ein Objekt

- Keine „primitiven Datentypen“ (wie in Java oder C++)
- Alles ist Objekt:
 - Zahlen (Instanzen von `int`, `float`, ...)
 - Zeichenketten (Instanzen von `str`, `unicode`, ...)
 - Auch Funktionen und Klassen sind Objekte

```
class KlasseA() :  
    pass
```

```
def funktion1() :  
    pass
```

```
type(KlasseA) # -> classobj  
type(funktion1) # -> function
```

- Schlüsselworte sind keine Objekte!

```
type(while) # Syntax-Fehler  
type(def) # Syntax-Fehler  
type(class) # Syntax-Fehler
```

Vorgestellte Begriffe

- Klasse
- Instanz (= Objekt)
- Attribut
- Methode
- Konstruktor
- Basisklasse
- Vererbung

- `class, isinstance(...),
issubclass(...), self,
id(...), type(...)`

Weitere Themen mit OOP-Bezug:

- Kapselung
- Polymorphie
- Ducktyping
- Mehrfachvererbung
- statische Methoden
- Meta-Klassen-Programmierung