

Informatik I

15. Objekt-orientierte Programmierung: Einstieg

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

03. Dezember 2013

Informatik I

03. Dezember 2013 — 15. Objekt-orientierte Programmierung: Einstieg

15.1 Motivation

15.2 OOP: Erste Schritte in Python

15.3 Vererbung

15.1 Motivation

- Was ist OOP?
- Welche Konzepte sind wichtig?
- Vor- und Nachteile

Was ist das?

- ▶ OOP ist ein **Programmierparadigma** (Programmierstil) – es gibt noch weitere.
- ▶ Es ist die Art und Weise an ein Problem 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 (im OOP-Sinne)

- ▶ Objekte sind überall!
- ▶ Können von uns als solche wahrgenommen werden.
- ▶ Objekte haben
 - ▶ in der realen Welt: **Zustand** und **Verhalten**
 - ▶ in OOP modelliert durch: **Attributwerte** bzw. **Methoden**

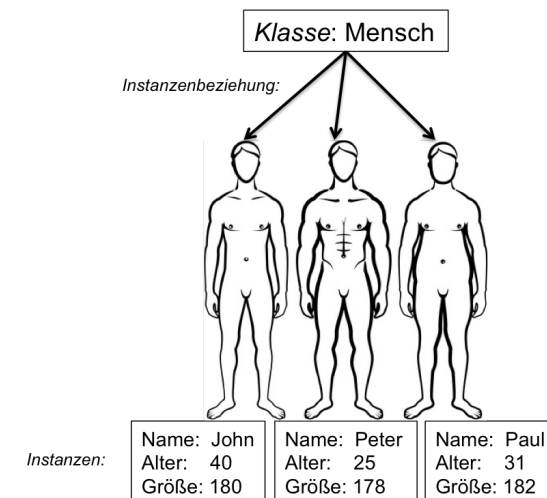
Objekte in OOP

- ▶ Der Zustand eines realen Objekts wird mit Hilfe von Attributwerten repräsentiert. Beispiel: Der *Kontostand* eines Kontos wird im Attribut guthaben als Zahl gespeichert.
- ▶ Verhalten wird durch Methoden realisiert. Beispiel: Entsprechend einem *Abhebe-Vorgang* verringert ein Aufruf der Methode abheben den Betrag, der unter dem Attribut guthaben gespeichert ist.
- ▶ Methoden sind die Schnittstellen zur Interaktion zwischen Objekten.
- ▶ Normalerweise wird der interne Zustand versteckt (**Datenkapselung**).

Klassen und Objekte (1)

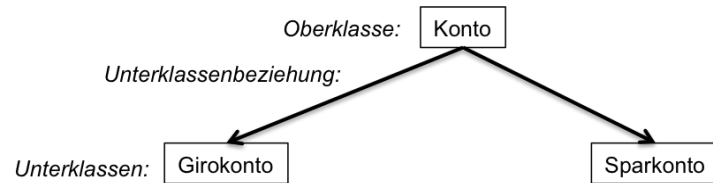
- ▶ Eine **Klasse**
 - ▶ ist der „Bauplan“ für bestimmte Objekte;
 - ▶ enthält die Definition der Attribute und Methoden;
 - ▶ macht alleine praktisch gar nichts.
- ▶ Ein **Objekt**
 - ▶ wird dem „Bauplan“ entsprechend erzeugt
 - ▶ ist dann ein Element/eine **Instanz** der Klasse

Klassen und Objekte (2)



Klassenhierarchien

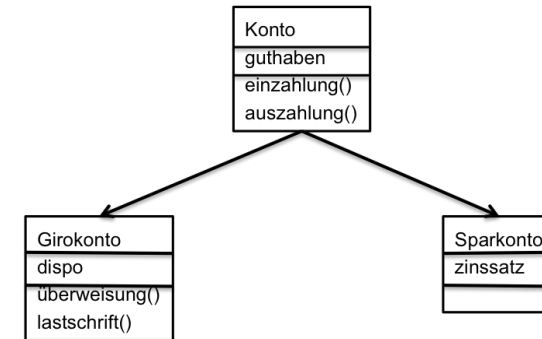
- ▶ Verschiedene Arten von Klassen können oft in einer **Generalisierungshierarchie** angeordnet werden:



- ▶ Man spricht von:
 - ▶ Superklasse, Oberklasse, Elternklasse und Basisklasse (für die obere Klasse)
 - ▶ Subklasse, Unterklasse, Kindklasse bzw. abgeleitete Klasse (für die unteren Klassen)

Vererbung

- ▶ Unterklassen **erben** Attribute und Methoden von der Oberklasse
- ▶ ... und können neue Attribute und Methoden **eingeführen**
- ▶ ... und können Attribute und Methoden der Oberklasse **überschreiben**



Vorteile von OOP

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

Nachteile von OOP

Bei folgenden Punkten hat der OOP-Ansatz Schwächen:

- ▶ **Formulierung**: Die natürliche Sprache hat keine feste Bindung von Substantiv (entspr. dem Objekt) und Verb (entspr. der Methode).
- ▶ **Klassenhierarchie** ist in der realen Welt nicht immer so klar (Kreis-Ellipse-Problem).
- ▶ **Transparenz**: Kontrollfluss nicht im Quelltext
- ▶ **Ausführungseffizienz**: OOP-Anwendungen benötigen häufig mehr Ressourcen (Laufzeit, Speicher, Energie) als prozedurale Formulierungen.
- ▶ **Programmier-effizienz**: Kleine Anwendungen sind oft schneller prozedural programmiert.

15.2 OOP: Erste Schritte in Python

- Klassendefinition
- Instanzenerzeugung
- Methoden
- Ein Beispiel: Der Kreis

Definieren von Klassen

- ▶ Neue Klassen werden mit der **class**-Anweisung eingeführt (Konvention: **CamelCase**-Namen).
 - ▶ Beachte: Wie bei Funktionsdefinitionen mit `def` werden die `class`-Anweisung **ausgeführt**. D.h. wenn man sie in einer bedingten Anweisung unterbringt, werden sie u.U. nicht ausgeführt!
- ▶ Sehen ähnlich aus wie Typen (und sind tatsächlich solche)

Python-Interpreter

```
>>> class MyClass:
...     pass # nur notwendig für leere Klasse!
...
>>> MyClass
<class '__main__.MyClass'>
>>> int
<class 'int'>
```

Erzeugung von Instanzen

- ▶ Eine **Instanz** einer Klasse wird erzeugt, indem man die Klasse wie eine Funktion aufruft.
- ▶ Alle erzeugten Instanzen sind untereinander nicht-identisch und ungleich!

Python-Interpreter

```
>>> class MyClass:
...     pass
...
>>> instance1 = MyClass()
>>> instance1
<__main__.MyClass object at 0x101ac51d0>
>>> instance2 = MyClass()
>>> instance1 is instance2
False
>>> instance1 == instance2
False
```

Instanzen sind dynamische Strukturen/Records

- ▶ Instanzen verhalten sich wie Records/Strukturen, denen man *dynamisch* neue **Attribute** zuordnen kann.

Python-Interpreter

```
>>> class Circle:
...     pass
...
>>> my_circle = Circle()
>>> my_circle.radius = 5
>>> 2 * 3.14 * my_circle.radius
31.4
```

- ▶ D.h. man kann jeder Instanz dynamisch neue Attribute zuordnen – jede Instanz stellt einen eignen **Namensraum** dar, auf den wir mit der Punktnotation zugreifen.
- ▶ Wie wir für alle Instanzen einer Klasse die selben Attribute erzeugen, sehen wir gleich.

Methoden-Definition

- ▶ **Methoden** werden als Funktionen innerhalb von Klassen definiert (mit `def`).
- ▶ Den ersten Parameter einer Methode nennt man per Konvention **self** – dies ist *die Instanz/das Objekt*.

Python-Interpreter

```
>>> class Circle:
...     def area(self):
...         return (self.radius * self.radius *
...                 3.14159)
...
>>> c = Circle()
```

Methoden-Aufrufe

- ▶ Methoden können aufgerufen werden:
 - ▶ über den Klassennamen (dann muss das `self`-Argument angegeben werden), oder
 - ▶ **normal** über den Instanzen-Namen (dann wird die Instanz implizit übergeben).

Python-Interpreter

```
>>> class Circle:
...     def area(self):
...         return (self.radius * self.radius *
...                 3.14159)
...
>>> c = Circle(); c.radius = 1
>>> Circle.area(c)
3.14159
>>> c.area()
3.14159
```

Konstruktion von Instanzen

- ▶ Um für alle Instanzen einer Klasse die gleichen Attribute zu haben, werden diese normalerweise in der `__init__`-Methode eingeführt, die bei der Erzeugung der Instanz aufgerufen wird.

Python-Interpreter

```
>>> class Circle:
...     def __init__(self, rad):
...         self.radius = rad
...
>>> circle = Circle(22)
>>> circle.radius
22
>>> circle.radius = 1
>>> circle.radius
1
```

Die `__init__`-Methode

- ▶ Die spezielle Methode mit dem Namen `__init__` wird aufgerufen, wenn die Instanz erzeugt wird. In dieser Methode „erzeugt“ man die Attribute durch Zuweisungen.
- ▶ **Beachte:** Alle Attribute sind öffentlich zugreifbar!
- ▶ **Beachte:** Auch bei Methoden-Definitionen sind benannte und Default-Parameter möglich!
- ▶ **Beachte:** Attributnamen und Parameternamen von Methoden gehören zu verschiedenen Namensräumen.

Python-Interpreter

```
>>> class Circle:
...     def __init__(self, radius=1):
...         self.radius = radius
...
>>> circle = Circle(5)
```

Ein Kreis ...

circle.py

```
class Circle:
    def __init__(self, x=0, y=0, radius=1):
        self.x = x
        self.y = y
        self.radius = radius

    def area(self):
        return self.radius * self.radius * 3.14

    def size_change(self, percent):
        self.radius *= (percent / 100)

    def move(self, xchange=0, ychange=0):
        self.x += xchange
        self.y += ychange
```

Kreise bearbeiten

Python-Interpreter

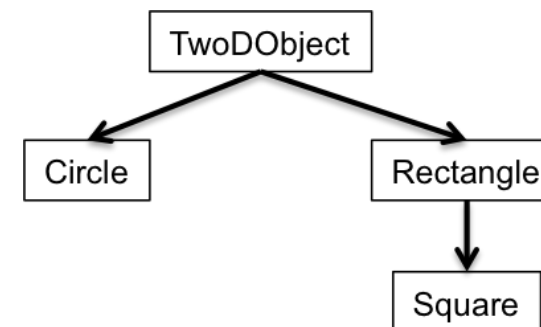
```
>>> c = Circle(x=1, y=2, radius=5)
>>> c.area()
78.5
>>> c.size_change(50)
>>> c.area()
19.625
>>> c.move(xchange=10, ychange=20)
>>> (c.x, c.y)
(11, 22)
```

15.3 Vererbung

- 2D-Objekte
- Überschreiben und Erweitern
- Quadrate und Rechtecke
- Datenkapselung
- Noch einmal Quadrate
- Klassenvariablen

Weitere Objekte

- ▶ Wir wollen jetzt noch andere geometrische Figuren einführen, wie Rechtecke, Quadrate, Dreiecke, Ellipsen, ...
- ▶ Diese haben **Gemeinsamkeiten** (alle haben eine Position in der Ebene) und es gibt **Unterschiede** (Kreis: Radius, Rechteck: Seiten)
- ▶ So etwas kann gut in einer **Klassenhierarchie** dargestellt werden



2D-Objekte

- ▶ Allen Objekten gemeinsam ist, dass sie eine Position in der Ebene haben
- ▶ Diese will man ggfs. auch drucken und verändern können.

geoclasses.py (1)

```
class TwoDObject:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def move(self, xchange=0, ychange):
        self.x += xchange
        self.y += ychange

    def print_pos(self):
        print(self.x, self.y)
```

Wo steckt der Fehler?

Überschreiben versus Erweitern

- ▶ Bei der Vererbung kommen weitere Attribute und Methoden hinzu.
- ▶ Vorhandene Methoden können **überschrieben** werden (Beispiel: `__init__`).
- ▶ Oft ist es besser, sie zu **erweitern** – und von der Funktionalität der Methode in der Superklasse Gebrauch zu machen.

geoclasses.py (3)

```
class Circle1(TwoDObject):
    def __init__(self, x=0, y=0, radius=1):
        self.radius = radius
        TwoDObject.__init__(self, x, y)
```

- ▶ **Beachte:** Hier wird die Methode über den Klassennamen aufgerufen.

Ein Kreis ist ein 2D-Objekt

- ▶ Jetzt können wir Kreise als eine Spezialisierung von 2D-Objekten einführen und die **zusätzlichen** und **geänderten** Attribute und Methoden angeben:

geoclasses.py (2)

```
class Circle(TwoDObject):
    def __init__(self, x=0, y=0, radius=1):
        self.radius = radius
        self.x = x
        self.y = y

    def area(self):
        return self.radius * self.radius * 3.14

    def size_change(self, percent):
        self.radius *= (percent / 100)
```

Super!

- ▶ Es wird explizit die Methode der aktuellen Superklasse aufgerufen. Wenn sich die Hierarchie ändert (z.B. auch nur der Name der Superklasse), muss beim Methodenaufruf nachgebessert werden.
- ▶ Stattdessen automatisch die Superklasse bestimmen:

geoclasses.py (4)

```
class Circle2(TwoDObject):
    def __init__(self, x=0, y=0, radius=1):
        self.radius = radius
        super().__init__(x, y)
```

- ▶ **Beachte:** Die Parameterkonventionen müssen bekannt sein oder man muss mit `**kwlist` arbeiten.
- ▶ Tatsächlich ist `super()` umstritten: <http://rhettinger.wordpress.com/2011/05/26/super-considered-super/>

Ein Rechteck ist auch ein 2D-Objekt

- ▶ Und weiter geht es mit Rechtecken

geoclasses.py (5)

```
class Rectangle(TwoDObject):
    def __init__(self, x=0, y=0, height=1, width=1):
        self.height = height
        self.width = width
        super().__init__(x, y)

    def area(self):
        return height * width

    def size_change(self, percent):
        self.height *= (percent / 100)
        self.width *= (percent / 100)
```

Wo steckt der Fehler?

Spielen mit 2D-Objekten

Python-Interpreter

```
>>> t = TwoDObject(10,20)
>>> c = Circle(11,22,5)
>>> r = Rectangle(100,100,20,20)
>>> c.print_pos()
(11,22)
>>> c.move(89,78); c.print_pos()
(100,100)
>>> t.area()
AttributeError: 'TwoDObject' object has no attribute 'area'
>>> r.area()
400
>>> r.size_change(50); r.area()
100
```

Ein Quadrat ist ein Rechteck

- ▶ Ein Quadrat ist ein Spezialfall eines Rechtecks (jedenfalls im mathematischen Sinne).

geoclasses.py (6)

```
class Square(Rectangle):
    def __init__(self, x=0, y=0, side=1):
        super().__init__(x, y, side, side)
```

- ▶ Was allerdings, wenn wir eine Square-Instanz ändern und z.B. dem height-Attribut einen neuen Wert zuweisen?
- Die Instanz ist **kein Quadrat** mehr!
- ▶ Allerdings haben wir hier auch auf **interne** (?) Attribute zugegriffen.
 - ▶ Was ist mit der Datenkapselung in Python?
 - ▶ Und würde Datenkapselung hier wirklich helfen?

Datenkapselung in Python

- ▶ Die Idee der **Datenkapselung** ist, dass die interne Implementation *nicht sichtbar* ist und nur über Methoden zugegriffen wird.
- ▶ In anderen OOP-Sprachen existieren Konzepte der Beschränkung wie **private** (sichtbar nur innerhalb der Klasse), **protected** (sichtbar in allen Subklassen), **public** (für jeden sichtbar).
- ▶ Python ist da liberal und vertraut darauf, dass die Nutzer **vernünftig** sind – was das Debuggen z.B. erheblich vereinfacht:
 - ▶ Attribute, die **nicht mit Unterstrich** beginnen, sind für alle sichtbar und modifizierbar.
 - ▶ Attribute, die mit **einem Unterstrich** beginnen, sind intern und sollten außerhalb nicht benutzt werden.
 - ▶ Attribute, die mit **zwei Unterstrichen** beginnen, sind *nicht direkt sichtbar*, da der Klassenname intern mit eingefügt wird (Namens-Massage).

Pythons beschränkt zugreifbare Attribute

`geoclasses.py` (7)

```
class TwoDObject1:
    def __init__(self, x=0, y=0):
        self.__x = x
        self._y = y
```

Python-Interpreter

```
>>> td = TwoDObject1(1,2)
>>> td._y
2
>>> td.__x
AttributeError: 'TwoDObject1' object has no attribute '__x'
>>> td._TwoDObject1__x
1
```

Ein Quadrat ist ein Quadrat ist ein ...

- ▶ Ändern wir die Klassendefinitionen so ab, dass alle Instanzvariablen einen oder zwei Unterstriche als erstes Zeichen haben (also nicht geändert werden sollen), so kann nur die Methode `size_change` die Attribute ändern.
- Ein als Quadrat eingeführtes Quadrat bleibt immer Quadrat!
- ▶ Was, wenn man *Höhe* und *Breite* separat über Methoden ändern könnte: `stretch_height` und `stretch_width`?
- Das **Kreis-Ellipsen-Problem** ist identisch mit dem *Quadrat-Rechteck-Problem*.
- ▶ Verschiedene Lösungen sind denkbar. M.E. die „vernünftigste“ ist, die beiden Methoden so zu **überschreiben**, dass jeweils auch der andere Wert geändert wird.

Rechtecke und Quadrate in friedlicher Koexistenz (1)

`geoclasses.py` (8)

```
class RectangleStretch(TwoDObject):
    def __init__ ...

    def stretch_height(self, percent):
        self.__height *= (percent / 100.0)

    def stretch_width(self, percent):
        self.__width *= (percent / 100.0)

class SquareStretch(RectangleStretch):
    def __init__ ...

    def stretch_height(self, percent):
        super().stretch_height(percent)
        super().stretch_width(percent)
```

Rechtecke und Quadrate in friedlicher Koexistenz (2)

- ▶ Jetzt wird bei jedem Aufruf von `stretch_height` und `stretch_width` dafür gesorgt, dass die jeweils andere Seite auch geändert wird.
- ▶ **Beachte:** Es kann jetzt auch ein Rechteck geben, das gleiche Höhe und Breite hat! Es ist dann aber nur *zufällig* ein Quadrat. Ein als Quadrat erzeugtes Objekt wird *immer* ein Quadrat sein.
- ▶ Alternative Möglichkeit zu der betrachteten: Eine Instanz könnte sich je nachdem, ob die Seiten gleichlang sind oder nicht, **dynamisch** als Instanz von Rechteck oder Quadrat einordnen.
- ▶ Weitere Alternative: Rechtecke sind Subklassen von Quadraten, da sie mehr Eigenschaften besitzen.
- ▶ Die Verwirrung entsteht, da die Objekte ja nicht nur statische, unveränderliche Eigenschaften haben, sondern sich ändern: Wie **verhält** sich ein Quadrat?

Klassenvariablen

- ▶ Können auch Klassen Attribute besitzen?
- Variablen, die innerhalb des Klassenkörpers eingeführt werden, heißen **Klassenattribute** (oder **statische Attribute** und sind (auch) in allen Instanzen (zum Lesen) sichtbar.
- ▶ Innerhalb von Methoden müssen sie über den Klassennamen angesprochen werden.

`geoclasses.py` (9)

```
class TwoDObjectCount:

    counter = 0

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        TwoDObjectCount.counter += 1
```

Klassenvariablen: Lesender und schreibender Zugriff

Python-Interpreter

```
>>> TwoDObjectCount.counter
0
>>> t1 = TwoDObjectCount()
>>> TwoDObjectCount.counter
1
>>> t2 = TwoDObjectCount()
>>> t3 = TwoDObjectCount()
>>> TwoDObjectCount.counter
3
>>> t1.counter
3
>>> t1.counter = 111 # Neues Objekt-Attr. erzeugt!
>>> TwoDObjectCount.counter
3
```