

Informatik I: Einführung in die Programmierung

17. Objekt-orientierte Programmierung: Einstieg

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Bernhard Nebel

4. Dezember 2015

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

Motivation

Was ist OOP?

Welche Konzepte
sind wichtig?

Vor- und Nachteile

OOP: Erste
Schritte in
Python

Vererbung

Ein bisschen
GUI

Zusammen-
fassung

- 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

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

Vor- und Nachteile

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

- Objekte gibt es im realen Leben überall!
- Sie können von uns als solche wahrgenommen werden.
- Objekte haben
 - in der realen Welt: **Zustand** und **Verhalten**
 - in OOP modelliert durch: **Attributwerte** bzw. **Methoden**

Motivation

Was ist OOP?

Welche Konzepte
sind wichtig?

Vor- und Nachteile

OOP: Erste
Schritte in
Python

Vererbung

Ein bisschen
GUI

Zusammen-
fassung

- 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**).

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

Vor- und Nachteile

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

■ 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

Motivation

Was ist OOP?

Welche Konzepte
sind wichtig?

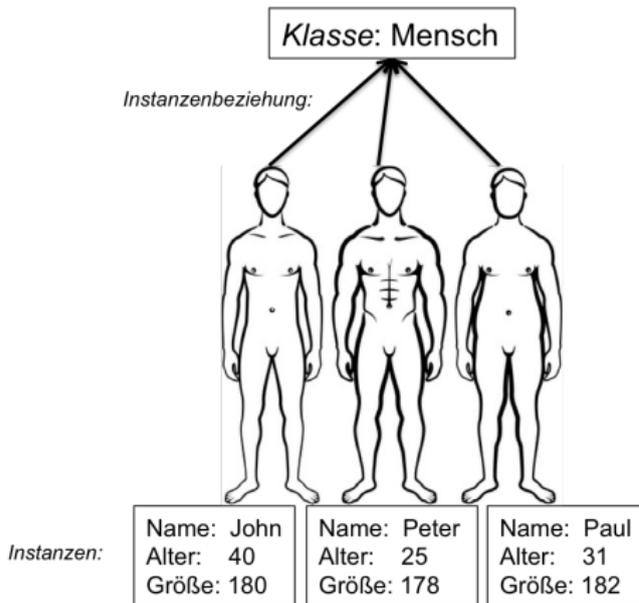
Vor- und Nachteile

OOP: Erste
Schritte in
Python

Vererbung

Ein bisschen
GUI

Zusammen-
fassung



Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

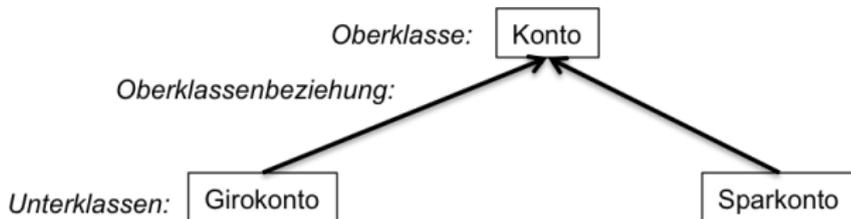
Vor- und Nachteile

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung



- 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)

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

Vor- und Nachteile

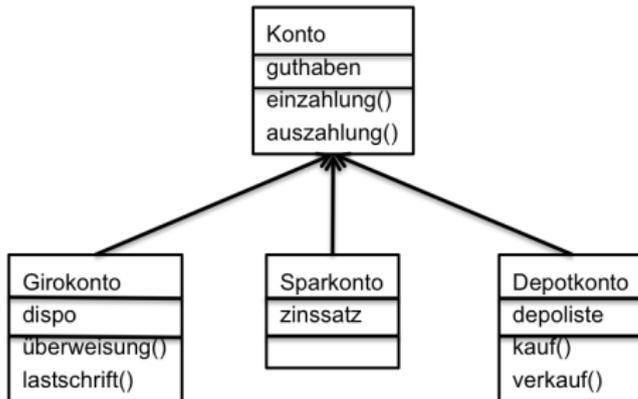
OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

- Unterklassen **erben** Attribute und Methoden von der Oberklasse



- ... und können neue Attribute und Methoden **einführen**
- ... und können Attribute und Methoden der Oberklasse **überschreiben**

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

Vor- und Nachteile

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

- **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.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

Vor- und Nachteile

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

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.
- **Programmiereffizienz:** Kleine Anwendungen sind oft schneller prozedural programmiert.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

Vor- und Nachteile

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

2 OOP: Erste Schritte in Python



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

Motivation

OOP: Erste Schritte in Python

Klassendefinition

Instanzenerzeugung

Methoden

Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung

Python-Interpreter

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

- 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)

Motivation

OOP: Erste Schritte in Python

Klassendefinition

Instanzen-

erzeugung

Methoden

Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung

- Eine **Instanz** einer Klasse wird erzeugt, indem man die Klasse wie eine Funktion aufruft.

Python-Interpreter

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

- Alle erzeugten Instanzen sind untereinander nicht-identisch und ungleich!

Motivation

OOP: Erste Schritte in Python

Klassendefinition

Instanzen-
erzeugung

Methoden

Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung



- 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 eigenen **Namensraum** dar, auf den wir mit der Punktnotation zugreifen.
- Wie wir für alle Instanzen einer Klasse die selben Attribute erzeugen, sehen wir gleich.

Motivation

OOP: Erste Schritte in Python

Klassendefinition

Instanzerzeugung

Methoden

Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung

- **Methoden** werden als Funktionen innerhalb von Klassen definiert (mit `def`).

Python-Interpreter

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

- Den ersten Parameter einer Methode nennt man per Konvention **self** – dies ist *die Instanz/das Objekt*.

Motivation

OOP: Erste Schritte in Python

Klassendefinition
Instanzen-
erzeugung

Methoden
Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung

- Methoden können aufgerufen werden:

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
```

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

Motivation

OOP: Erste Schritte in Python

Klassendefinition
Instanzen-
erzeugung

Methoden

Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung



- 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
```

Motivation

OOP: Erste Schritte in Python

Klassendefinition

Instanzerzeugung

Methoden

Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung



- Die spezielle Methode mit dem Namen `__init__` wird aufgerufen, wenn die Instanz erzeugt wird. In dieser Methode „erzeugt“ man die Attribute durch Zuweisungen.

Python-Interpreter

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

- **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.

Motivation

OOP: Erste Schritte in Python

Klassendefinition

Instanzen-
erzeugung

Methoden

Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung

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
```

Motivation

OOP: Erste Schritte in Python

Klassendefinition

Instanzen-
erzeugung

Methoden

Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung

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)
```

Motivation

OOP: Erste Schritte in Python

Klassendefinition

Instanzen-
erzeugung

Methoden

Ein Beispiel: Der Kreis

Vererbung

Ein bisschen GUI

Zusammenfassung

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

Motivation

OOP: Erste Schritte in Python

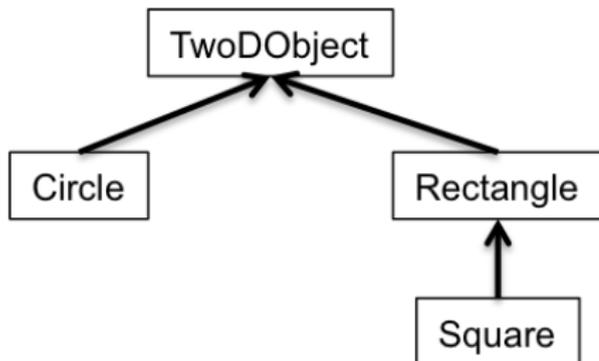
Vererbung

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

Ein bisschen GUI

Zusammenfassung

- 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



Motivation

OOP: Erste Schritte in Python

Vererbung

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

Ein bisschen GUI

Zusammenfassung

- 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=0):
        self.x += xchange
        self.y += ychange

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

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung



- 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)
```

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung



- 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.

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung

- Es wird explizit die Methode der aktuellen Superklasse aufgerufen. Wenn sich die Hierarchie ändert (z.B. auch nur der Name der Superklasse), muss beim Methodenaufwurf 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/>

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung



- 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)
```

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung



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
```

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung



- 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?

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung



- 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).

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung



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
```

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte
Überschreiben und Erweitern
Quadrate und Rechtecke

Datenkapselung
Noch einmal Quadrate
Klassenvariablen

Ein bisschen GUI

Zusammenfassung

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.

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung

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)
```

Motivation

OOP: Erste Schritte in Python

Vererbung

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

Ein bisschen GUI

Zusammenfassung

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* eine Quadrat. Ein als Quadrat erzeugtes Objekt wird *immer* ein Quadrat sein.
- Alternative Möglichkeit: 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?

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung

- Können auch Klassen Attribute besitzen?

geoclasses.py (9)

```
class TwoDObjectCount:
```

```
    counter = 0
```

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

```
        self.x = x
```

```
        self.y = y
```

```
        TwoDObjectCount.counter += 1
```

- Variablen, die innerhalb des Klassenkörpers eingeführt werden, heißen **Klassenattribute** (oder **statische Attribute**) und sind (auch) in allen Instanzen (zum Lesen) sichtbar.
- Zum Schreiben müssen sie über den Klassennamen angesprochen werden.

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung



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
```

Motivation

OOP: Erste Schritte in Python

Vererbung

2D-Objekte

Überschreiben und Erweitern

Quadrate und Rechtecke

Datenkapselung

Noch einmal Quadrate

Klassenvariablen

Ein bisschen GUI

Zusammenfassung

4 Ein bisschen GUI



Motivation

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

- Jede moderne Programmiersprache bietet auf den normalen Desktoprechnen heute ein oder mehrere **Graphical User Interfaces** an.
- In Python gibt es **tkinter** (integriert), **PyGtk**, **wxWidget**, **PyQt**, ...
- Möglichkeit per Fenster und Mausinteraktion zu interagieren.
- Wir wollen jetzt einen kleinen Teil von **tkinter** kennen lernen, um unsere Geo-Objekte zu visualisieren.

Motivation

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

Hello World

```
import tkinter as tk
import sys

root = tk.Tk()
lab = tk.Label(root, text="Hello World")
lab.pack()

if "idlelib" not in sys.modules:
    root.mainloop()
```

- `root` wird das Wurzelobjekt, in das alle anderen Objekte hineinkommen.
- `lab` wird ein **Label-Widget** innerhalb des `root`-Objekts erzeugt.
- Dann wird `lab` in seinem Elternfenster positioniert.
- Schließlich wird die Event-Schleife aufgerufen.

Motivation

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung



- IDLE selbst ist mit Hilfe von `tkinter` implementiert worden.
- Deshalb muss man etwas vorsichtig sein, wenn man `tkinter` in IDLE entwickelt.
- Man sollte nicht (noch einmal) `mainloop()` aufrufen (dafür sorgt das `if`-Statement)
- Man sollte das Programm nicht beenden, da sonst `tkinter` mit beendet wird.

Motivation

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

Canvas erzeugen

```
import tkinter as tk
import sys

root = tk.Tk()
cv = tk.Canvas(root, height=600, width=600)
cv.pack()
r1 = cv.create_rectangle(100, 100, 200, 150, fill='green')
o1 = cv.create_oval(400,400,500,500,fill='red',width=3)
```

- Ein Canvas ist eine Leinwand, auf der man „malen“ kann.
- Darauf kann man dann verschiedene geometrische Figuren erzeugen.

Motivation

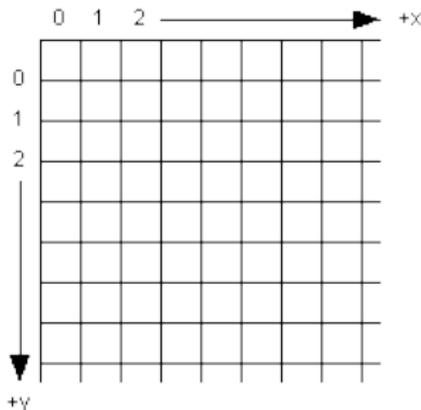
OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

- Im Unterschied zum mathematischen Koordinatensystem liegt der Nullpunkt bei Grafikdarstellungen immer **oben links**.



- Wie gewohnt gibt man **(x,y)**-Paare zur Bestimmung von Punkten an.

Motivation

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung



- `canvas.create_line(x1, y1, x2, y2, **kw)` zeichnet eine **Linie** von $(x1, y1)$ nach $(x2, y2)$.
- `canvas.create_rectangle(x1, y1, x2, y2, **kw)` zeichnet ein **Rechteck** mit oberer linker Ecke $(x1, y1)$ und unterer rechter Ecke $(x2, y2)$.
- `canvas.create_oval(x1, y1, x2, y2, **kw)` zeichnet ein **Oval** innerhalb des Rechtecks geformt durch obere linke Ecke $(x1, y1)$ und untere rechte Ecke $(x2, y2)$.
- Alle create-Methoden liefern den **Index** des erzeugten Objekts.
- `canvas.delete(i)` **löscht** Objekt mit dem Index i .
- `canvas.move(i, xdelta, ydelta)` **bewegt** Objekt um $xdelta$ und $ydelta$.
- `canvas.update()` erneuert die Darstellung auf dem Bildschirm (auch für andere Fenster möglich).

Motivation

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung



- Wenn wir annehmen, dass die Objektpositionen unserer geometrischen Objekte immer der **Schwerpunkt** ist, dann könnte man den Kreis wie folgt definieren.

Geoclasses visuell

```
class Circle(TwoDObject):
    def __init__(self, x=0, y=0, radius=1):
        self.radius = radius
        super().__init__(x, y)
        self.index = cv.create_oval(self.x-self.radius,
                                   self.y-self.radius,
                                   self.x+self.radius,
                                   self.y+self.radius)

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

Motivation

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung

5 Zusammenfassung



Motivation

OOP: Erste
Schritte in
Python

Vererbung

Ein bisschen
GUI

**Zusammen-
fassung**



- **Objekt-orientierte Programmierung** ist ein wichtiges **Programmierparadigma**
- Ein Problem wird zerlegt in seine Objekte und die Interaktionen zwischen den Objekten.
- **Klassen** sind die „Baupläne“ für die **Instanzen**.
- **Methoden** sind Funktionen, die innerhalb der Klasse definiert werden, und als ersten Parameter immer `self` besitzen sollten.
- Attribute werden innerhalb der `__init__`-Methode initialisiert.
- Klassen können in einer **Vererbungshierarchie** angeordnet werden.
- Es gibt auch **Klassenattribute**.

Motivation

OOP: Erste Schritte in Python

Vererbung

Ein bisschen GUI

Zusammenfassung