

Informatik I: Einführung in die Programmierung

12. Objekt-orientierte Programmierung: Einstieg und ein bisschen GUI

Albert-Ludwigs-Universität Freiburg



Peter Thiemann

18. Dezember 2018

1 Motivation



- Was ist OOP?
- Welche Konzepte sind wichtig?

Motivation

Was ist OOP?

Welche Konzepte
sind wichtig?

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

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 kooperierender Objekte
 - Zustand wird mit den Operationen darauf zusammengefasst

Motivation

Was ist OOP?

Welche Konzepte
sind wichtig?

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- 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?

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Der Zustand eines realen Objekts wird mit Hilfe von Attributwerten repräsentiert.
Beispiel: Der *Kontostand* eines Kontos wird im Attribut `balance` als Zahl gespeichert.
- Verhalten wird durch Methoden realisiert.
Beispiel: Entsprechend einem *Abhebe-Vorgang* verringert ein Aufruf der Methode `withdraw` den Betrag, der unter dem Attribut `balance` 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?

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Eine **Klasse**
 - ist der „Bauplan“ für bestimmte Objekte;
 - enthält die Definition der Attribute und Methoden;
- Ein **Objekt / Instanz der Klasse**
 - wird dem „Bauplan“ entsprechend erzeugt

Motivation

Was ist OOP?

Welche Konzepte
sind wichtig?

OOP: Die
nächsten
Schritte

Vererbung

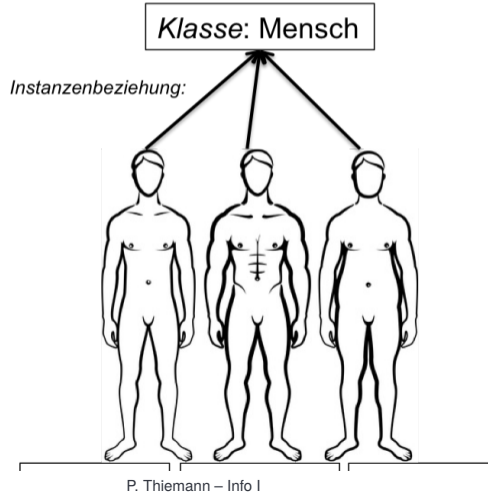
Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

Klassen und Objekte (2)



Motivation

Was ist OOP?

Welche Konzepte
sind wichtig?

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

2 OOP: Die nächsten Schritte



- Klassendefinition
- Methoden
- Ein Beispiel: Der Kreis

Motivation

OOP: Die nächsten Schritte

Klassendefinition

Methoden

Ein Beispiel: Der Kreis

Vererbung

Vererbung konkret

Vermischtes

Ein bisschen GUI

Zusammenfassung

Wiederholung: Definieren von Klassen

Beispiel: Geometrische Objekte



Kreis

Ein Kreis wird beschrieben durch seinen Mittelpunkt und seinen Radius.

Klassengerüst

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

Motivation

OOP: Die
nächsten
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der
Kreis

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung



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

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

    def area(self):
        return (self.radius * self.radius * math.pi)
```

- Der erste Parameter einer Methode heißt per Konvention **self**. Das ist der **Empfänger** des Methodenaufrufs, d.h. die Instanz, auf der die Methode aufgerufen wird.

Motivation

OOP: Die
nächsten
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der
Kreis

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Methoden können aufgerufen werden:

Python-Interpreter

```
>>> c = Circle(1, 0, 0)
```

```
>>> Circle.area(c)
```

```
3.14159
```

```
>>> c.area()
```

```
3.14159
```

- zur Not über den Klassennamen (dann muss das `self`-Argument angegeben werden), oder
- über eine Instanz, die dann implizit übergeben wird.
- Im *Normalfall* wird `c.area()` verwendet!

Motivation

OOP: Die
nächsten
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der
Kreis

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

Ein Kreis ...

circle.py

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

    def area(self):
        return self.radius * self.radius * math.pi

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

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

Motivation

OOP: Die
nächsten
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der
Kreis

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

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: Die
nächsten
Schritte

Klassendefinition
Methoden

Ein Beispiel: Der
Kreis

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Wir wollen jetzt noch weitere geometrische Figuren einführen, wie Kreissektoren, Rechtecke, Dreiecke, Ellipsen, Kreissegmente, ...
- Ein **Rechteck** wird beschrieben durch seinen Referenzpunkt (links oben) und seine Seitenlängen.
- Also

Motivation

OOP: Die
nächsten
Schritte

Klassendefinition
Methoden

Ein Beispiel: Der
Kreis

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

Klasse fürs Rechteck

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

    def area(self):
        return self.width * self.height

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

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



Motivation

OOP: Die
nächsten
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der
Kreis

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Die Bearbeitung des Referenzpunkts (x,y) ist bei `Circle` und `Rectangle` Objekten gleich.
 - Der Konstruktor behandelt sie gleich.
 - Die `move` Methode behandelt sie gleich.
- Gesucht: **Abstraktion**, mit der diese Gemeinsamkeit ausgedrückt werden kann, sodass der entsprechende Code für den `move` und die Initialisierung der Attribute nur einmal geschrieben werden muss.

Motivation

OOP: Die nächsten Schritte

Klassendefinition

Methoden

Ein Beispiel: Der Kreis

Vererbung

Vererbung konkret

Vermischtes

Ein bisschen GUI

Zusammenfassung

3 Vererbung



Motivation

OOP: Die
nächsten
Schritte

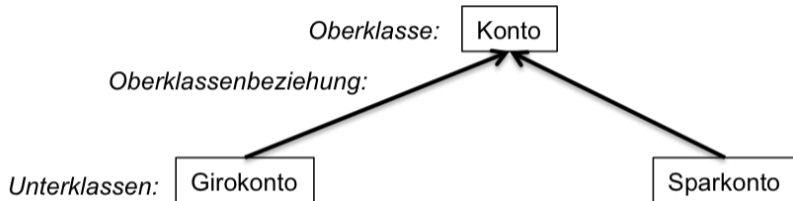
Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung



- Verschiedene Arten von Klassen können oft in einer **Generalisierungshierarchie** angeordnet werden.
- Terminologie:
 - Superklasse, Oberklasse, Elternklasse und Basisklasse (für die obere Klasse)
 - Subklasse, Unterklasse, Kindklasse bzw. abgeleitete Klasse (für die unteren Klassen)

Motivation

OOP: Die nächsten Schritte

Vererbung

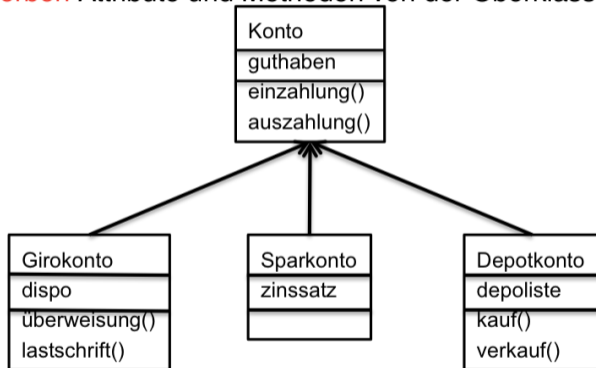
Vererbung konkret

Vermischtes

Ein bisschen GUI

Zusammenfassung

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



- ...und können neue Attribute und Methoden **introduce**
- ...und können Attribute und Methoden der Oberklasse **override**

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Vermischtes

Ein bisschen GUI

Zusammenfassung

4 Vererbung konkret



- 2D-Objekte
- Überschreiben und dynamische Bindung
- `__init__`
- `__str__`

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

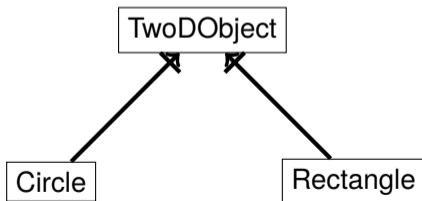
2D-Objekte
Überschreiben und dynamische Bindung
`__init__`
`__str__`

Vermischtes

Ein bisschen GUI

Zusammenfassung

- Wir fassen die **Gemeinsamkeiten** der Klassen (alle haben einen Referenzpunkt, der verschoben werden kann) in einer eigenen Klasse zusammen.
- Die **Unterschiede** werden dann in spezialisierten **Subklassen** (**Unterklassen**) implementiert.
- Daraus ergibt sich eine **Klassenhierarchie**:



- TwoDObject ist **Superklasse** von Circle und Rectangle.

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte
Überschreiben und dynamische Bindung
`__init__`
`__str__`

Vermischtes

Ein bisschen GUI

Zusammenfassung



- Allen geometrischen Figuren gemeinsam ist, dass sie einen Referenzpunkt besitzen, der verschoben werden kann, und dass sie eine Fläche besitzen.

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 = self.x + xchange
        self.y = self.y + ychange

    def area(self):
        return 0
```

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte

Überschreiben und
dynamische
Bindung

__init__
__str__

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

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, radius=1, x=0, y=0):
        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 = self.radius * (percent / 100)
```

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte
Überschreiben und
dynamische
Bindung
__init__
__str__

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Durch Vererbung kommen weitere Attribute und Methoden hinzu (hier: `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
- Geerbte Methoden können **überschrieben** werden (Beispiel: `area`), dadurch dass wir in der Subklasse eine neue Definition angeben.
- Auf einer `Circle` Instanz wird aufgerufen
 - `move` aus `TwoDObject`
 - `area` aus `Circle`
 - `__init__` aus `Circle`

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte
Überschreiben und
dynamische
Bindung
`__init__`
`__str__`

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung



- Das Verhalten eines Methodenaufrufs wie `obj.area()` bzw `obj.move()` wird erst zur Laufzeit des Programms bestimmt.
- Es **hängt ab vom (Laufzeit-) Typ** von `obj`.
 - Falls `obj` eine Instanz von `TwoDObject` ist, also falls `type(obj) == TwoDObject`, dann wird sowohl für `area` als auch für `move` der Code aus `TwoDObject` verwendet.
 - Falls `obj` eine Instanz von `Circle` ist, also falls `type(obj) == Circle`, dann wird für `area` der Code aus `Circle` und für `move` der Code aus `TwoDObject` verwendet.
- Dieses Verhalten heißt **dynamische Bindung** bzw **dynamic dispatch** und ist typisch für objekt-orientierte Sprachen.

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte

Überschreiben und dynamische Bindung

`__init__`

`__str__`

Vermischtes

Ein bisschen GUI

Zusammenfassung

Beispiel

Python-Interpreter

```
>>> t = TwoDObject(x=10, y=20)
>>> t.area()
0
>>> t.move(xchange=10, ychange=20)
>>> t.x, t.y
(20, 40)
>>> t.size_change(50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'TwoDObject' object has no attribute 'size_change'
>>> 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: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

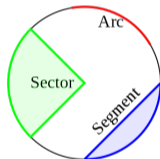
2D-Objekte
Überschreiben und
dynamische
Bindung
__init__
__str__

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Ein Kreissektor wird beschrieben durch einen Kreis und einen Winkel:



https://commons.wikimedia.org/wiki/File:Circle_slices.svg (public domain)

- Für Sektoren können wir eine Subklasse von `Circle` anlegen.

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte
Überschreiben und dynamische Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen GUI

Zusammenfassung

Kreissektor als Subklasse vom Kreis



```
class Sector (Circle):
    def __init__(self, angle= 180, radius=1, x=0, y=0):
        self.angle = angle
        self.radius = radius
        self.x = x
        self.y = y

    def area(self):
        return self.radius * self.radius * math.pi * self.angle / 360
```

Sector verwendet

- move von TwoDObject
- size_change von Circle
- area von Sector, aber ein Teil des Codes ist aus Circle **kopiert!**

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte

Überschreiben und
dynamische
Bindung

__init__
__str__

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?
- Über den Klassennamen könnte die überschriebene Methode der Superklasse explizit aufgerufen werden.

Expliziter Aufruf (fehleranfällig!)

```
class Sector1(Circle):
```

```
    ...
```

```
    def area():
```

```
        return Circle.area(self) * self.angle / 360
```

- **Fehlerquelle:** Wenn sich die Hierarchie ändert (z.B. auch nur der Name der Superklasse), muss beim Methodenaufruf nachgebessert werden.

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte

Überschreiben und
dynamische
Bindung

`__init__`

`__str__`

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Besser: Python bestimmt die Superklasse automatisch:

Verwendung von `super` (empfohlen)

```
class Sector1(Circle):  
    ...  
    def area():  
        return super().area() * self.angle / 360
```

- `super()` nur innerhalb von Methoden verwenden.
- `super().method(...)` ruft `method` auf dem Empfänger (also `self`) auf, aber tut dabei so, als ob `self` **Instanz der Superklasse** wäre.
- Im Beispiel wird `area` in `Circle` aufgerufen.

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte
Überschreiben und dynamische Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen GUI

Zusammenfassung

Python-Interpreter

```
>>> s = Sector (x=1, y=2, radius=5, angle=90)
>>> s.area()
19.634954084936208
>>> c = Circle (x=1, y=2, radius=5)
>>> c.area()
78.53981633974483
>>> assert s.area() * 4 == c.area()
>>> s.move(9,8)
>>> s.x, s.y
(10, 10)
>>> s.size_change(200)
>>> s.area()
78.53981633974483
```

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte

Überschreiben und
dynamische
Bindung

`__init__`

`__str__`

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

Die `__init__` Methode



- `__init__` wird ebenfalls in `Circle` und `Sector` überschrieben.
- Auch hier kann `super` verwendet werden.

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

class Sector2(Circle2):
    def __init__(self, angle= 180, radius=1, x=0, y=0):
        self.angle = angle
        super().__init__(radius, x, y)
    ...
```

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte
Überschreiben und
dynamische
Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung



- Falls die `__init__` Methode der Superklasse viele Parameter hat oder sich diese Parameter im Laufe der Zeit ändern können, dann ist es unschön, wenn diese Parameter im eigenen Code festgelegt werden.
- In Python gibt es die Möglichkeit, neben festen Parameterlisten auch beliebige Parameterlisten zu definieren, wobei die einzelnen Parameter durch ihren Namen (Schlüsselwort) angewählt werden können.
- Daher der Name **keyword parameter**.
- Für keyword parameter gibt es eine spezielle Syntax **`**kwargs`**, die sowohl in der formalen Parameterliste, wie auch im Funktionsaufruf verwendet werden kann.
- Hier und jetzt nur die Verwendung in `__init__`, Details später im Zusammenhang mit dem Datentyp **Dictionary**.

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte
Überschreiben und dynamische Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen GUI

Zusammenfassung

Praxistipp: `__init__` mit keyword parameter



```
class Circle3(TwoDObject):
    def __init__(self, radius=1, **kwargs):
        self.radius = radius
        super().__init__(**kwargs)
    ...
class Sector3(Circle3):
    def __init__(self, angle= 180, **kwargs):
        self.angle = angle
        super().__init__(**kwargs)
    ...
```

- **Beachte:**
 - jede Klasse benennt nur die Argumente des Konstruktors `__init__`, die lokal verwendet werden.
 - der Aufruf von `super().__init__` ist immer gleich!
- **Anmerkung:** `**kwargs` lässt sich mit jeder Funktion verwenden, aber nur an letzter Position der Parameterliste.

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte
Überschreiben und
dynamische
Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

Ein Rechteck ist auch ein 2D-Objekt

- Und weiter geht es mit Rechtecken

geoclasses.py (5)

```
class Rectangle(TwoDObject):
    def __init__(self, height=1, width=1, **kwargs):
        self.height = height
        self.width = width
        super().__init__(**kwargs)

    def area(self):
        return self.height * self.width

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

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte
Überschreiben und
dynamische
Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

Python-Interpreter

```
>>> c = Circle(5,11,22)
>>> r = Rectangle(100,100,20,20)
>>> c.x,c.y
(11,22)
>>> c.move(89,78); c.x,c.y
(100,100)
>>> t.area()
0
>>> r.area()
400
>>> r.size_change(50); r.area()
100
```

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte
Überschreiben und
dynamische
Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

- Wir haben das `__str__` Attribut bereits kennengelernt.
- Es definiert eine Funktion, die das Objekt in einen String umwandelt.
- `__str__` kann auch direkt als Methode in der entsprechenden Klasse definiert werden.

Aufgabe

Definiere eine `__str__` Methode in `TwoDObject`, die **auch für alle Subklassen korrekt funktioniert**.

Das heißt, sie liefert den Namen der Klasse, mit der die Instanz konstruiert wurde, und sämtliche Konstruktorparameter mit ihren Namen.

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte
Überschreiben und dynamische Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen GUI

Zusammenfassung



```
class TwoDObjectS:  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y  
  
    def __str__(self):  
        n = self.getName()  
        p = self.getParameters()  
        return n + "(" + p[1:] + ")"
```

Wunschdenken

- Die Methode `getName` liefert jeweils den Namen des Konstruktors.
- Die Methode `getParameters` liefert jeweils die Argumentliste als String.

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte
Überschreiben und
dynamische
Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

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

    def __str__(self):
        n = self.getName()
        p = self.getParameters()
        return n + "(" + p[1:] + ")"

    def getName(self):
        return "TwoDObject"

    def getParameters(self):
        return ",x=" + repr(self.x) + ",y=" + repr(self.y)
```

- In den Subklassen muss nur noch jeweils getName und getParameters überschrieben werden

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte

Überschreiben und dynamische Bindung

__init__

__str__

Vermischtes

Ein bisschen GUI

Zusammenfassung



```
class CircleS(TwoDObjectS):
    def __init__(self, radius=1, **kwargs):
        self.radius = radius
        super().__init__(**kwargs)

    def getName (self):
        return "Circle"

    def getParameters (self):
        return ",radius=" + repr(self.radius) + super().getParameters()
```

- der eigene Parameter zuerst, dann die Parameter der Superklasse durch den Aufruf von `super().getParameters()`

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte
Überschreiben und dynamische Bindung

`__init__`
`__str__`

Vermischtes

Ein bisschen GUI

Zusammenfassung



```
class SectorS(CircleS):
    def __init__(self, angle=180, **kwargs):
        self.angle = angle
        super().__init__(**kwargs)

    def getName (self):
        return "Sector"

    def getParameters (self):
        return ",angle=" + repr (self.angle) + super().getParameters()
```

- analog zu CircleS
- RectangleS würde genauso aussehen (selbst!)

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

2D-Objekte
Überschreiben und
dynamische
Bindung

__init__
__str__

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

■ Klassenvariablen

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Klassenvariablen

Ein bisschen
GUI

Zusammen-
fassung

- Auch Klassen können Attribute besitzen!

geoclasses.py (9)

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

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

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Vermischtes
Klassenvariablen

Ein bisschen GUI

Zusammenfassung

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
```

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes
Klassenvariablen

Ein bisschen
GUI

Zusammen-
fassung

6 Ein bisschen GUI



Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

**Ein bisschen
GUI**

Zusammen-
fassung

- Jede moderne Programmiersprache bietet heute eine oder mehrere APIs (Application Programming Interface) für GUIs (**Graphical User Interface**) an.
- Möglichkeit per Fenster und Mausinteraktion zu interagieren.
- In Python gibt es **tkinter** (integriert), **PyGtk**, **wxWidget**, **PyQt**, uvam.
- Wir wollen jetzt einen kleinen Teil von `tkinter` kennen lernen, um unsere Geo-Objekte zu visualisieren.

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Vermischtes

Ein bisschen GUI

Zusammenfassung

Hello World



Hello World

```
import tkinter as tk

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

- tkinter repräsentiert Bildschirmhalte durch einen Baum
- root wird das Wurzelobjekt, in das alle anderen Objekte hineinkommen.
- lab wird als **Label-Widget** innerhalb des root-Objekts erzeugt.
- Ein **Widget** ist eine (rechteckige) Fläche auf dem Schirm, auf der eine bestimmte Funktionalität implementiert ist.
- Das Label-Widget kann nur einen String als Text anzeigen.
- Dann wird lab in seinem Elternfenster positioniert.

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Vermischtes

Ein bisschen GUI

Zusammenfassung

Canvas erzeugen

```
import tkinter as tk
```

```
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 ein Widget, das wie eine Leinwand funktioniert, auf der verschiedene geometrische Figuren gemalt werden können.

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung



Motivation

OOP: Die nächsten Schritte

Vererbung

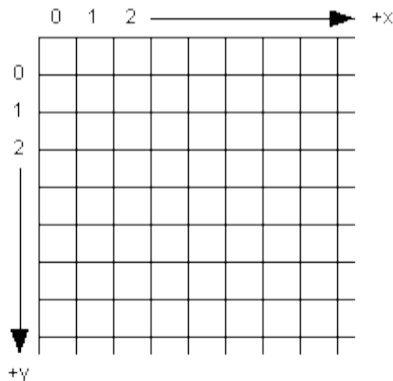
Vererbung konkret

Vermischtes

Ein bisschen GUI

Zusammenfassung

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



Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Vermischtes

Ein bisschen GUI

Zusammenfassung

- `canvas.create_line(x1, y1, x2, y2, **options)` zeichnet eine **Linie** von $(x1, y1)$ nach $(x2, y2)$.
- `canvas.create_rectangle(x1, y1, x2, y2, **options)` zeichnet ein **Rechteck** mit oberer linker Ecke $(x1, y1)$ und unterer rechter Ecke $(x2, y2)$.
- `canvas.create_oval(x1, y1, x2, y2, **options)` 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.

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

Geoclasses visuell

```
class TwoDObjectV:
    def __init__(self, cv=None, x=0, y=0):
        self.x = x
        self.y = y
        self.cv = cv
        self.index = 0

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

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

Geoclasses visuell

```
class CircleV(TwoDObjectV):
    def __init__(self, radius=1, **kwargs):
        self.radius = radius
        super().__init__(**kwargs)

    def draw(self):
        self.index = self.cv.create_oval(self.x-self.radius,
                                         self.y-self.radius,
                                         self.x+self.radius,
                                         self.y+self.radius)
```

Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

Zusammen-
fassung

7 Zusammenfassung



Motivation

OOP: Die
nächsten
Schritte

Vererbung

Vererbung
konkret

Vermischtes

Ein bisschen
GUI

**Zusammen-
fassung**



- **Objekt-orientierte Programmierung** ist ein **Programmierparadigma**
- Ein Objekt fasst Zustand (Attribute) und die Operationen darauf (Methoden) zusammen.
- **Klassen** sind die „Baupläne“ für die Objekte. Sie definieren Attribute und Methoden.
- **Methoden** sind Funktionen, die innerhalb der Klasse definiert werden. Der erste Parameter ist immer `self`, das **Empfängerobjekt**.
- **Attribute** werden innerhalb der `__init__`-Methode initialisiert.
- Klassen können in einer **Vererbungshierarchie** angeordnet werden.
- Subklassen **erben** Methoden und Attribute der Superklassen; Methoden der Superklassen können **überschrieben** werden.
- Der Aufruf von Methoden erfolgt durch **dynamische Bindung**.

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Vermischtes

Ein bisschen GUI

Zusammenfassung