

OOP

- Benutzerdefinierte Typen
- Klassen und Objekte
- Erstellung\Verwendung von benutzerdefinierten Typen bzw. Klassen in Python



Klassen und Objekte

Objektorientierte Programmierung (OOP) ist eine Methode zur
Modularisierung von Programmen

Die Basis von allem:

das Objekt (= Daten + Funktionalität)

Objekte

- beschreiben einen Gegenstand, Person etc. aus der realen Welt
- haben Felder/Attribute (Eigenschaften), die das Aussehen des Objekts beschreiben
- haben Methoden, die die Attribute verändern
- *sind Substantive in einem Text*

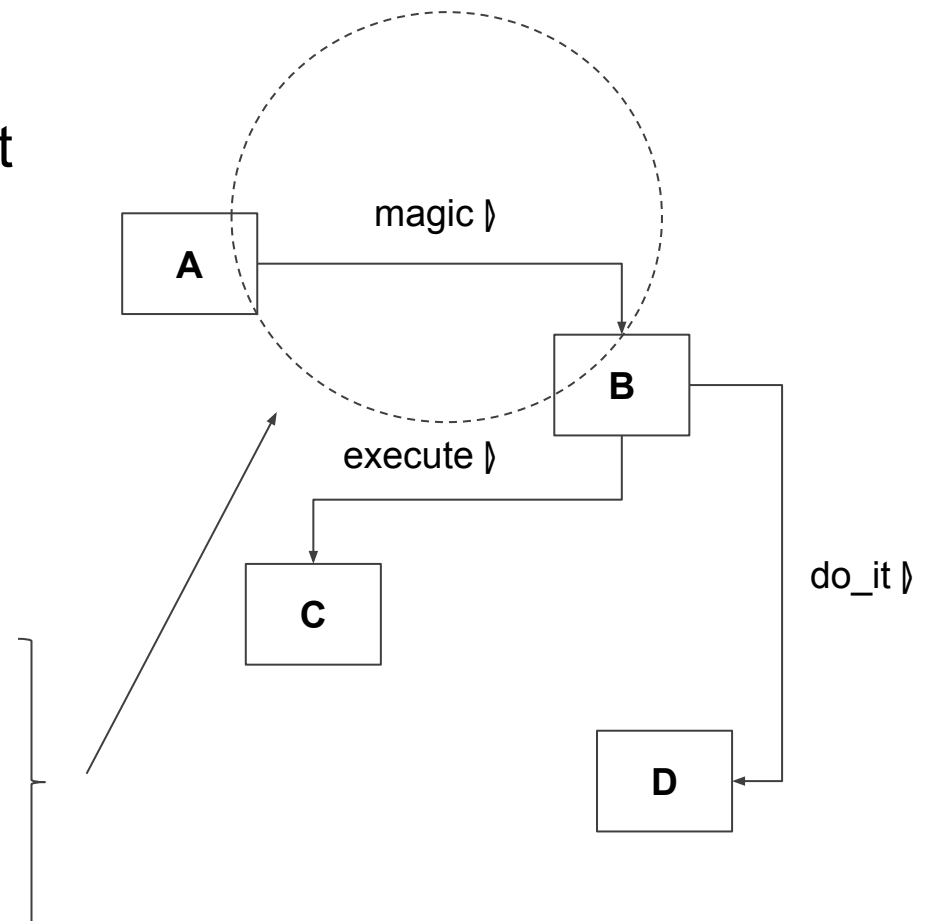
Beispiel

Schreibe eine Anwendung, welche das Spiel Hangman für die Konsole implementiert.

Es soll von einem **Spieler** beliebig oft spielbar sein. In jedem Schritt des **Spiels** soll die verbleibende Menge an Rateversuchen, sowie die bereits erratenen Buchstaben des Lösungswortes angezeigt werden.

Programme

- es gibt kein globaler Zustand
- der Zustand des Programms ist durch die Zustände aller Objekte beschrieben
- Objekte kommunizieren miteinander
- Beispiel:
 - Objekt **B** hat die Methode `magic`
 - Objekt **A** ruft `magic` auf (message passing)





just Code...

```
while True:
    print ("""
    1 - add
    2 - mul
    ...
    """)
    )
    opt = int(input("select?"))

    if opt == 1:
        a = int (input("a="))
        b = int (input("b="))
        number1 = (a,b)

        a = int (input("a="))
        b = int (input("b="))
        number2 = (a,b)

        rez = number[1]+...
    if opt == 2:
        a = int (input("a="))
        b = int (input("b="))
        number1 = (a,b)

        a = int (input("a="))
        b = int (input("b="))
        number2 = (a,b)

        rez = number[1]+...
```

Prozedurale Programmierung

```
def addition (r, total):  
    return rational(r.a*total.b + total.a+r.b,r.b*total.b)  
  
def multiplication(r, total):  
    ...  
  
def menu():  
    return """  
    1 - add  
    ...  
    """  
  
def main():  
    total = 0  
    while True:  
        print (menu())  
        opt = int(input("select?"))  
        ...
```

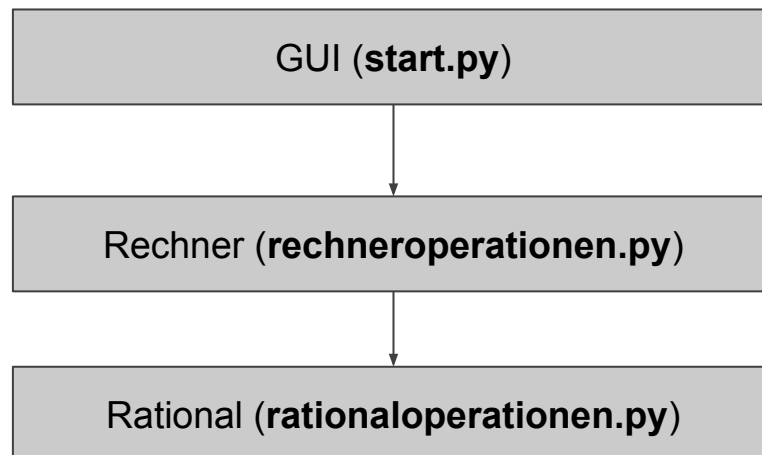
Modulare Programmierung

start.py

```
def menu():  
    return """  
    1 - add  
    ...  
    """  
  
def main():  
    while True:  
        print (menu())  
        opt = int(input("select?"))  
        ...
```

rationaloperationen.py

```
def add (r1, r2):  
    return rational(r1.a*r2.b +  
r2.a+r1.b,r1.b*r2.b)  
  
def mult (r1,r2):  
    return ...  
  
def to_string(r):  
    return "%i/%i" %(r.a,r.b)
```



rechneroperationen.py

```
def addition (r, total):  
    ...  
  
def multiplication(r, total):  
    ...
```



Modulare Programmierung

- Eine Anforderung wird in vielen kleinen Aufgaben/Tasks zerlegt
- Jede kleine Aufgabe/Task ist in sich abgeschlossen
- Jede kleine Aufgabe/Task ist in einer Datei abgelegt

classes
just. add. water.





OOP

- abstrahiert Gegenstände der realen Welt
- beschreibt und verändert Objekte
- versucht Daten und Funktionen eines Objekts in einer Struktur zu kapseln.
 - Die Daten beschreiben das Objekt.
 - Funktionen verändern die Attributwerte eines Objekts.

Prinzipien der OOP

Abstraktion

- Objekte der realen Welt werden nachgebildet

Datenkapselung

- das Objekt ist eine Black Box
- In dieser Black Box wird mit Hilfe von Felder das Objekt beschrieben
- Funktionen verändern die Felder eines Objekts

Beispiel

Ein Würfel

- Der Würfel ist rot -> Farbe
- Der Würfel hat eine Kantenlänge von 5 cm -> Länge
- *Der Würfel kann gedreht werden.*
- *Der Würfel kann neu eingefärbt werden.*

Beispiel

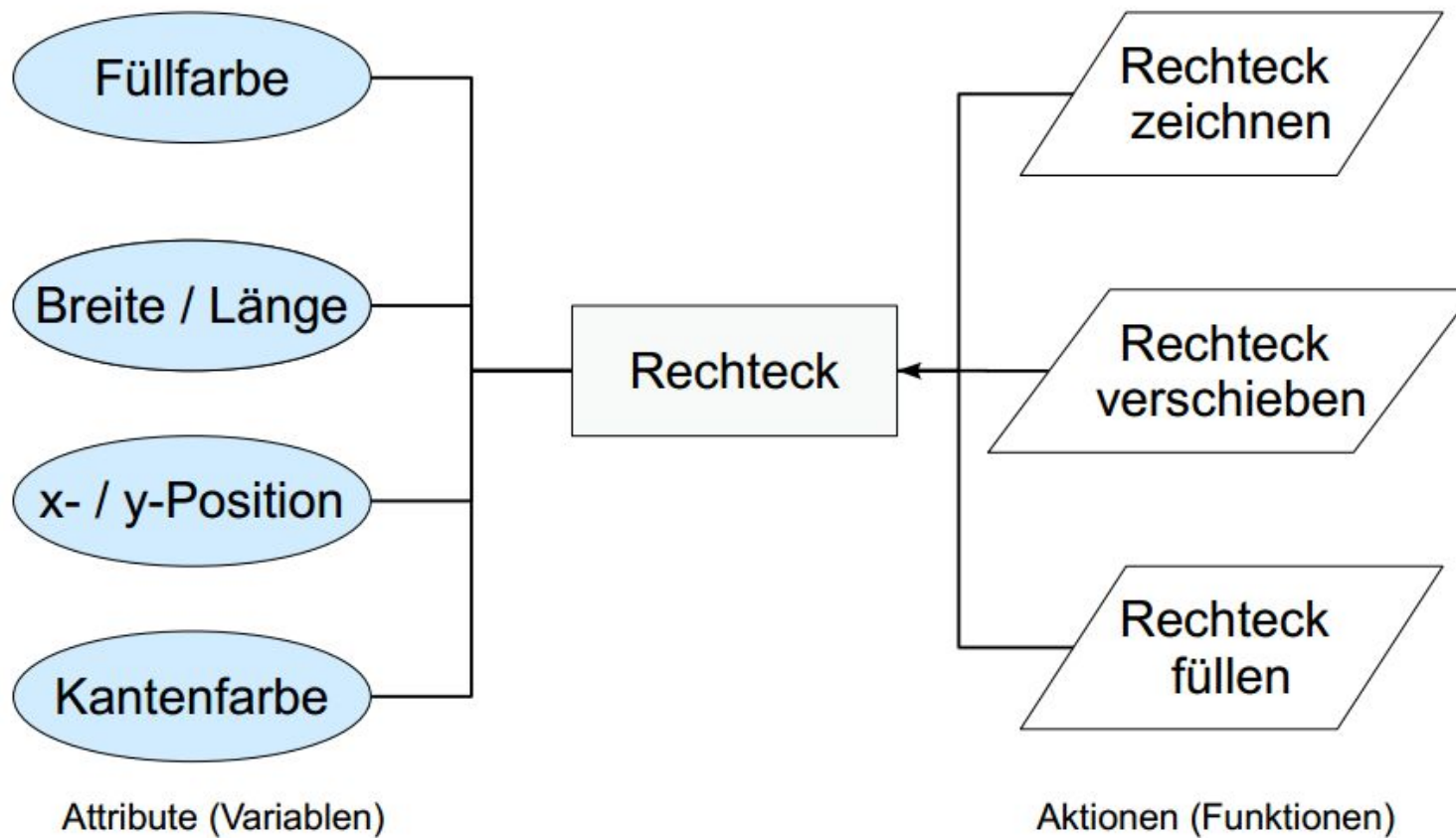
Welche Eigenschaften hat jeder Würfel?

- Farbe
- Kantenlänge

Welche Methoden hat jeder Würfel?

- Drehen
- Einfärben
- Zeichnen

Beispiel



Python

Everything is a object. In Python ist jedes Element ein Objekt.

Objekte:

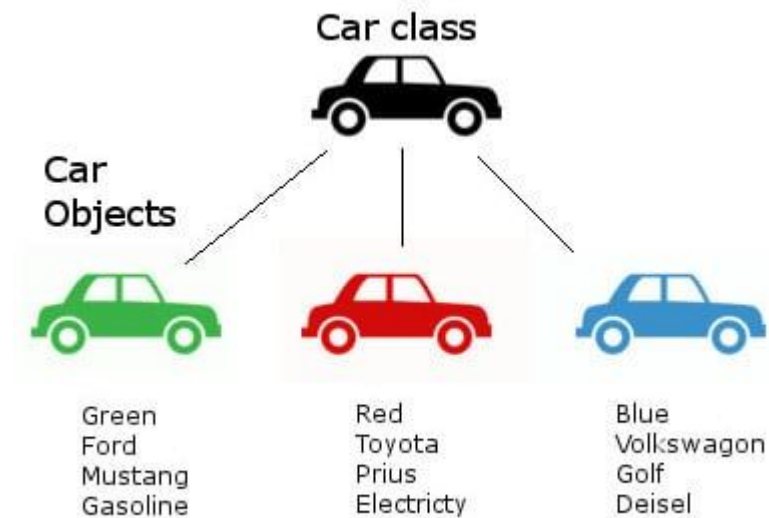
- können Felder und Funktionen haben
- sind an einen bestimmten Datentyp gebunden
- können an Funktionen übergeben werden
- werden mit Hilfe von Klassen beschrieben
- werden mit speziellen Methoden erzeugt und initialisiert

Beispiel

```
>>> #integer
>>> x = 1
>>> x.__add__(2) # x = 1 + 2
>>> 3
>>>
>>> #Listen
>>> l = [1, 2]
>>> l.__add__([2]) # l + [2]
[1, 2, 2]
>>> l
[1, 2]
>>>
```

Klassen

ein abstraktes Modell bzw. ein Bauplan für eine Reihe von ähnlichen Objekten



beschreiben Attribute (**Eigenschaften**) und Methoden (**Verhaltensweisen**) der Objekte.

Klassen

Definition:

- wird mit dem reservierten Wort `class` eingeleitet,
- danach kommt der Name der neuen Klasse,
- ein Doppelpunkt und wieder ein compound statement (**Einrückung!**)

```
class MyClass:  
    <statement 1>  
    ....  
    <statement n>
```

UML Beschreibung

- **Unified Modelling Language** zur Darstellung von Klassen
- Der Name der Klasse steht am oberen Rand
- Dem Namen folgen die Attribute der Klasse und darunter die Methoden
- In UML werden private Methoden und Attribute mit einem Minuszeichen und öffentliche Attribute und Methoden mit einem Pluszeichen gekennzeichnet.

Wuerfel
- farbe : string - kante: double
+get_Farbe() : string +get_Laenge() : float +set_Farbe() : string +set_Laenge(): float +drehen_Wuerfel() : void

Konstruktor

Konstruktor: eine Methode, die beim Erzeugen eines Objekts dieser Klasse aufgerufen wird.

```
x = MyClass()
```

Jedes Objekt hat einen eigenen Namespace. Namen darin heißen Attribute des Objekts.

```
class MyClass:  
    def __init__(self):  
        self.someData = []
```


Destruktor

Mit Hilfe von `del` RechteckBlau
wird automatisch der dazugehörige Destruktor
`def __del__()` aufgerufen

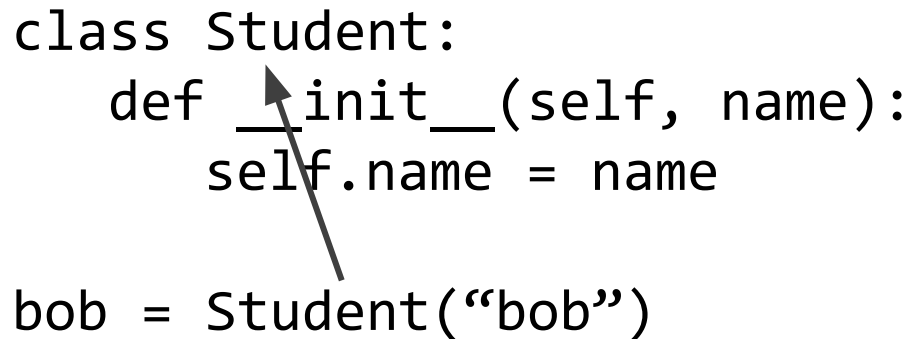
Die Methode wird implementiert, wenn zum Beispiel ...

- Netzwerkverbindungen getrennt werden müssen
- Dateien geschlossen werden müssen
- bestimmte Fehler abgefangen werden

Eine Instanz

```
class Student:
    def __init__(self, name):
        self.name = name

bob = Student("bob")
```

A diagram illustrating the creation of an object. An arrow points from the 'name' parameter in the 'def __init__' method to the 'name' argument in the 'Student("bob")' call. Another arrow points from the 'Student' class name to the 'Student' part of the same call. A third arrow points from the 'bob' variable to the 'Student' part of the call.

(Name des Objekts) (Name der Klasse)

- erweist auf ein bestimmtes Objekt einer bestimmten Kategorie
- ist ein Synonym für ein Objekt
- die Parameter sind im Konstruktor definiert



Felder/Attribute

- beschreiben den Zustand eines Objekts - Gegenstand, Person etc
- Jedes Objekt einer Klasse hat die gleiche Attribute
- Jedes Objekt einer Klasse unterscheidet sich aber in mindestens einem Attributwert von allen anderen Objekten

Felder/Attribute

```
self.n = a
```

```
n = a
```

```
class RationalNumber:
    """
        Abstract data type for rational numbers
        Domain: {a/b where a and b are integer numbers b!=0}
    """

    def __init__(self, a, b):
        """
            Creates a new instance of RationalNumber
        """
        self.n = a
        self.m = b

r1 = RationalNumber(1,3)  #create the rational number 1/3
```



Self I

- ist ein Platzhalter für den Aufrufer der Methode
- beantwortet die Frage „Wer hat die Methode aufgerufen?“
- ist meist das erste Argument einer Methode
- beschreibt die Instanz, die die Methode aufgerufen hat.



Methoden

- beschreiben das Verhalten eines Objekts
- lesen oder verändern Attributwerte
- beschreiben eine Schnittstelle nach außen
- werden innerhalb der Klasse definiert
- werden nur einmal für die Klasse im Speicher angelegt
- ergeben sich aus den Attributen und deren Nutzung in einer Klasse



```
def testCreate():
    """
    Test function for creating rational numbers
    """
    r1 = RationalNumber(1,3) #create the rational number 1/3
    assert r1.getNominator()==1
    assert r1.getDenominator()==3
    r1 = RationalNumber(4,3) #create the rational number 4/3
    assert r1.getNominator()==4
    assert r1.getDenominator()==3

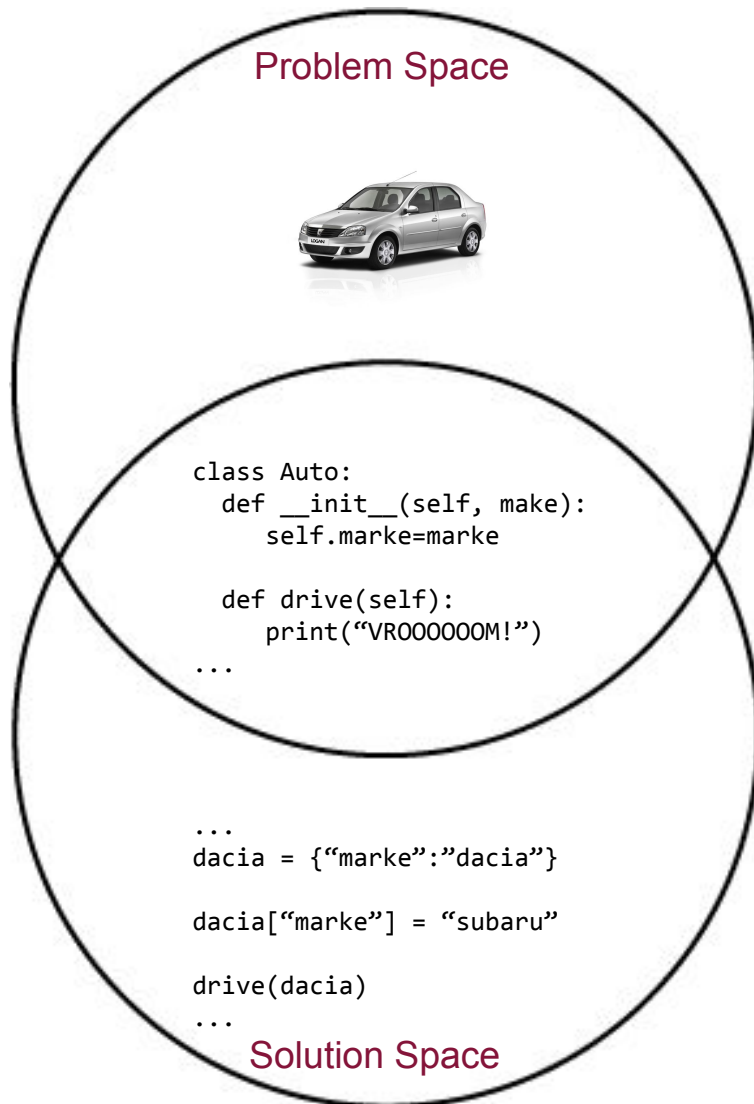
class RationalNumber:
    """
    Abstract data type rational numbers
    Domain: {a/b where a,b integer numbers, b!=0, greatest common divisor
    a, b =1}
    """
    def __init__(self, a, b):
        """
        Initialize a rational number
        a,b integer numbers
        """
        self.__nr = [a, b]

    def getDenominator(self):
        """
        Getter method
        return the denominator of the rational number
        """
        return self.__nr[1]

    def getNominator(self):
        """
        Getter method
        return the nominator of the method
        """
        return self.__nr[0]
```

- Die Methode wird mit der Instanz immer durch ein Punkt Verbunden.
- Falls die Methode nicht definiert ist, wird die Fehlermeldung „AttributeError“ ausgegeben.

Beispiele...Autos und Students



Beispiel

```
class Auto:
```

```
    def __init__(self):  
        self.kilometerstand = 0
```

```
    def drive(self):  
        self.kilometerstand += 1
```

```
dacia = Auto()
```

```
lada = Auto()
```

```
dacia.drive()
```

```
dacia.drive()
```

```
lada.drive()
```

Zustand

Verhalten

Objekte

dacia:

kilometerstand

2

lada:

kilometerstand

1



Self II

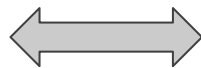
ist ein Platzhalter für den Aufrufer der Methode

```
class Auto:
    def __init__(self, farbe):
        self.kilometerstand = 0
        self.farbe = farbe

    def drive(self, km):
        self.kilometerstand += km
```

```
dacia = Auto("rot")
lada = Auto("blau")
```

```
dacia.drive(10)
lada.drive(200)
```



```
drive(dacia, 10)
drive(lada, 200)
```

Beispiel

- Implementiere eine Klasse **Auto**
- Jedes Auto hat für Felder/Attribute:
 - Marke als String
 - Modell als String
 - Farbe als String
 - Baujahr als Integer
- Implementiere eine Klasse **Statistics**
- die Klasse soll für eine Reihe von Autos berechnen:
 - die Anzahl von Autos mit einer eingegebenen Farbe
 - das durchschnittliche Baujahr für alle Autos einer Marke

Tests

Spezifikation