



Feedback: Woche 5

- man sollte wissen
 - wie Python funktioniert (Objekte, Names, etc.)
 - grundlegende Sprachelemente (keywords, if, for, etc.)
 - Grunddatentypen (int, float, etc.)
 - sequentielle Datentypen (list, tuple, dict, etc.)
 - Funktionen
 - Test-First-Ansatz

man sollte in der Lage sein

- einfache Algorithmen in Python zu implementieren
- seinen Code testen zu können



Feedback: Woche 5

- ich hätte mir mehr Kommunikation erwünscht
- einige von euch sollten mehr üben
- proaktives Feedback
- nur das Labor zu implementieren und passiv beim Seminar Code zu schreiben reicht einfach nicht



Modulare Programmierung

- wir haben Anweisungen in Funktionen gruppiert
- Aufgaben in Funktionen aufgeteilt
- Jede Funktion hat
 - ein klares Ziel/eine klare Verantwortung
 - Dokumentation
 - sinnvolle Namen
 - ExceptionHandling

Modulare Programmierung

Bad Funktion

```
def my_funk(n):  
    l = []  
    for i in range(n):  
        l.append(i)  
  
    s = sum(l)  
  
    print(s)
```

Modulare Programmierung

Bad Funktion

```
def my_funk(n):  
    l = []  
    for i in range(n):  
        l.append(i)  
  
    s = sum(l)  
  
    print(s)
```

- mehrere Verantwortungen (input, output, summe)
- no Dokumentation
- unklare Namen



Modulare Programmierung

- eine Softwaretechnik, die das Ausmaß erhöht, in dem Software aus unabhängigen, austauschbaren Komponenten besteht
 - Jede solche Komponente erfüllt einen Aspekt innerhalb des Programms und enthält alles, was dazu erforderlich ist.
 - Module in Python
- Module sind daher
 - Unabhängig
 - Austauschbar
- Ermöglichen die Gruppierung von Funktionen
- Ermöglichen die einfachere Bereitstellung von Funktionen
- Helfen bei der Lösung von Namenskonflikten

Modulare Programmierung in Python

- ein Python Modul ist eine Datei die aus Anweisungen und Definitionen besteht
- **Name**: der Dateiname ist der Modulname mit .py ergänzt
- **Docstring**
 - Drei öffnende Anführungszeichen
 - ein kurze Beschreibung in einem Satz
 - eine Leerzeile
 - weitere Bemerkungen
 - abschließend drei Anführungszeichen in einer eigenen Zeile
- **Anweisungen**
 - ein Modul kann ausführbare Anweisungen wie auch Funktionsdefinitionen enthalten
 - diese Anweisungen dienen der Initialisierung des Moduls
 - keine “globale” Variablen/Namen
 - sie werden nur dann ausgeführt, wenn das Modul das erste mal importiert wird

die import-Anweisung

- Um ein Modul verwenden zu können, muss es zuerst importiert werden.
- Die Importanweisung:
 - a. Durchsucht den globalen Namespace nach dem Modul. Wenn das Modul existiert, ist es bereits importiert und man muss nichts weiter machen
 - b. Sucht nach dem Modul.
 - c. Im Modul definierte Variablen und Funktionen werden in eine neue Symboltabelle (einen neuen Namespace) eingefügt. Nur der Modulname wird zur aktuellen Symboltabelle hinzugefügt

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
```

```
> sqrt(2)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'sqrt' is not defined
```

```
> from math import sqrt
```

```
> sqrt(2)
```

```
1.4142135623730951
```

```
> 
```



die import-Anweisung

```
from math import sqrt  
sqrt(2)
```

```
import math  
math.sqrt(2)
```

```
from math import *  
sqrt(2)
```



Beispiel

Modul

`useful_functions.py`

mit folgenden Funktionen

- `add(a,b)`
- `mul(a,b)`
- `sub(a,b)`

der Modul-Suchpfad

```
import spam
```

Wenn das Modul *spam* importiert wird, sucht der Interpreter nach einer Datei mit Namen *spam.py*

- im aktuellen Verzeichnis
 - wo das Skript gespeichert ist
- in der Liste der Verzeichnisse, die durch die Umgebungsvariable **PYTHONPATH** spezifiziert wird
- in der Liste der Verzeichnisse, die durch die Umgebungsvariable **PYTHONHOME** spezifiziert wird.
 - Abhängig von der Installation
 - /usr/local/lib/python (Unix)
- wenn das Modul nicht gefunden wurde, wird die ImportError Ausnahme ausgelöst

__name__-Variable

- jedes Modul bekommt, wenn es importiert wird, automatisch eine `__name__` Variable zugewiesen
 - die den Namen des Moduls angibt
- wenn Python-Module importiert werden, dann werden sie einmalig vom Interpreter ausgeführt
 - d.h. alle darin aufgelisteten Definitionen und Funktionsaufrufe werden zum Zeitpunkt des Importierens einmalig aufgerufen
- möchte man einige Funktionen in einer Python-Datei nur dann ausführen, wenn die Datei als Skript aufgerufen wird

```
if __name__ == '__main__':  
    # execute this only if the current file is interpreted directly
```



Packages

- Packages sind eine Möglichkeit, um Modulen zu strukturieren
- **A.B** zeigt das **B** ein Modul in Package **A** ist
 - Auf dem Laufwerk stellen Folders Packages dar
 - **B.py** befindet sich in einem Folder **A**.
- Für den Import von Paketen gelten die gleichen Regeln wie für Module
- Jeder Folder, der ein Package darstellt, enthält eine **__init__.py** Datei
- **__init__.py** kann leer sein oder Initialisierungscode enthalten.



Relative und absolute Pfadangaben

- um Module aus der gleichen Verzeichnisebene zu importieren

```
from . import modulname
```

- ein Modul kann aus dem übergeordneten Verzeichnis importiert werden

```
from .. import modulname
```

- und aus dem nochmals übergeordneten Verzeichnis importiert

```
from ... import modulname
```

Empfehlung für Paket-Strukturen

```
my_project/  
|- main.py  
| ...  
| - tests/  
|   |- test_main.py  
|   | - ...  
| - entities/  
|   |- student.py  
|   | - ...  
| - business/  
|   |- controller.py  
|   | - ...  
| - repository/  
|   |- student_repository.py  
|   | - ...  
| - user_interface/  
|   |- console.py  
|   | - ...  
|- LICENSE  
|- README.rst  
|- requirements.txt  
|- setup.py
```




Exkurs: Module

Lass uns ein Python-Programm schreiben, welches eine Liste von Studenten verwaltet. Jeder Student hat als Attribute Name, Universität.

Funktionalität

- Studenten anlegen und finden
- Studentinfo ausgeben





Exkurs: Module

Neue Funktionalität

Sort: Studenten nach Name sortieren





OOP

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



wie werden unsere Programme aussehen

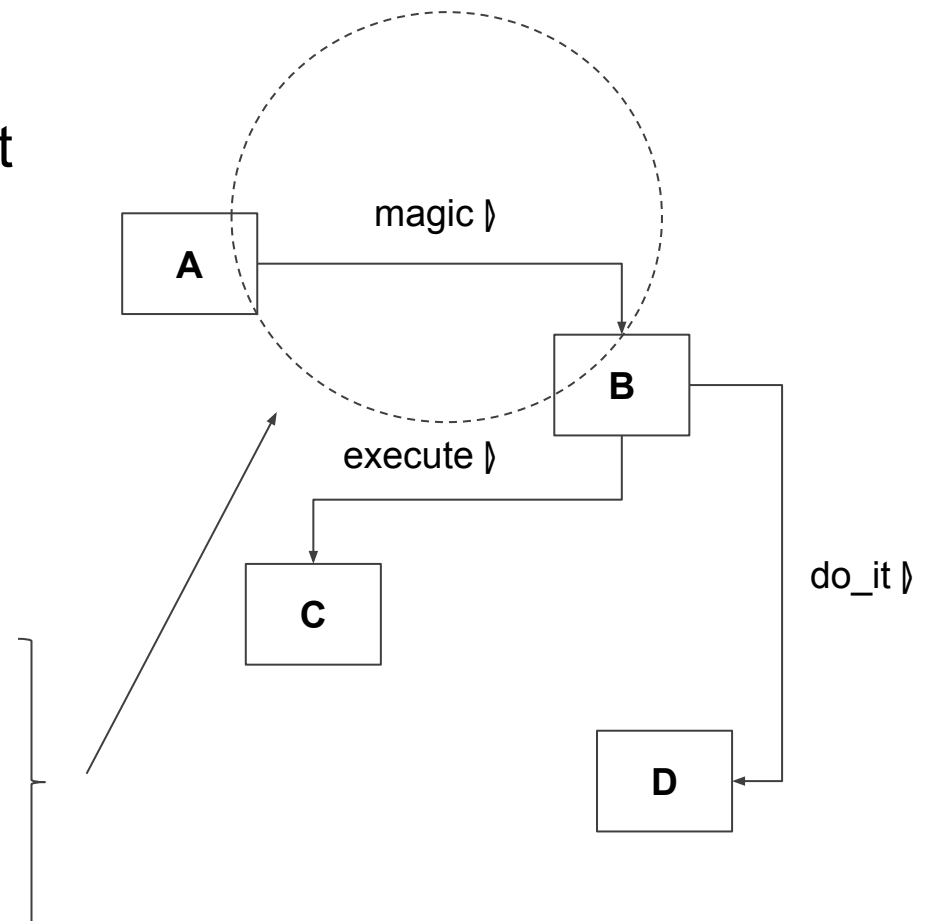
- Elementarer Bestandteil der Algorithmen-Entwicklung ist die Modellierung eines Ausschnitts aus der realen Welt
- Bisherige Modellierungsansätze: **Modulare Programmierung**
 - Die Lösung eines Problems wird in Funktionen modelliert
- Mit zunehmender Komplexität der Probleme führt dieser Ansatz allerdings zu immer komplexeren und unübersichtlicheren Programmen
- Der objektorientierte Ansatz bietet eine Lösung, komplexe Systeme in kleinere Komponenten zu zerlegen, die sich leichter beherrschen lassen

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 (einem Modul) abgelegt
 - und einer Funktion implementiert

wie werden unsere Programme aussehen

- 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?"))  
        ...
```

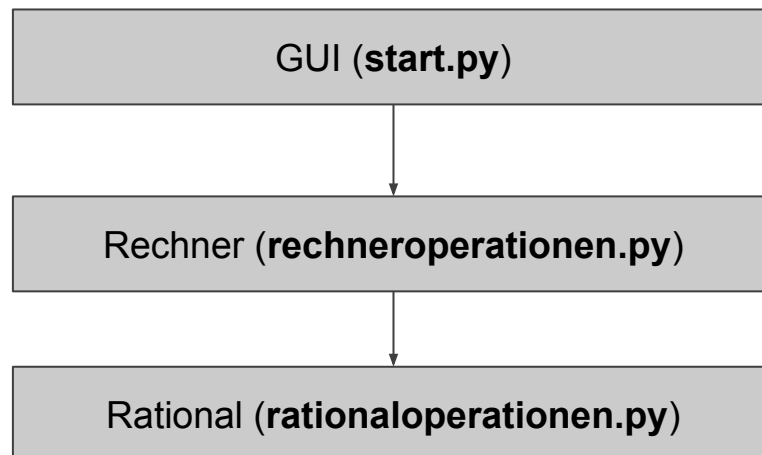
Modulare Programmierung

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):  
    ...
```

wie werden unsere Programme aussehen

- 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

classes
~~just. add. water.~~



OOP

- Objektorientierte Programmierung ist ein weiterer Ansatz, Problemstellungen der realen Welt zu modellieren
 - orientiert sich dabei an “Dingen” (Objekte, die gewisse Eigenschaften und ein gewisses Verhalten haben), die modelliert werden müssen
 - der OO Ansatz ein Konzept, komplexe Daten und Programmzustände zu modellieren
- Die tatsächlichen Algorithmen werden meist mit einer Mischung aus funktionalen und imperativen Konzepten notiert
- Die Algorithmen werden in der Regel mittels imperativer Konzepte implementiert, wogegen die Daten und deren Zustände OO modelliert sind

Klassen

- Die OO Sichtweise stellt sich die Welt als System von Objekten vor, die untereinander Botschaften austauschen
- **Objekte besitzen Eigenschaften (Attribute)**
 - Beispiel: ein Attribut eines Blumenhändlers ist der Ort an dem er sein Geschäft hat; weitere typische Attribute sind Name, Telefonnummer, Öffnungszeiten, Warenbestand, ...
- **Objekte können bestimmte Operationen (Methoden) ausführen**
 - Beispiel: ein Blumenhändler kann einen Lieferauftrag für Blumen entgegennehmen, Sträuße binden, Boten schicken, Blumen beim Großhandel einkaufen, ...
- Wenn ein Objekt eine geeignete Botschaft empfängt, wird eine zur Botschaft passende Operation gestartet (Methode aufgerufen)
- Der Umwelt (d.h. den anderen Objekten) ist bekannt welche Methoden ein Objekt beherrscht
- Allerdings weiß die Umwelt von den Methoden nur:
 - Was sie bewirken
 - Welche Daten sie als Eingabe benötigen

Klassen

- Die Umwelt weiß aber nicht, wie das Objekt funktioniert
 - d.h. nach welchen Algorithmen die Botschaften verarbeitet werden
- Das bleibt „privates“ Geheimnis des Objekts
 - z.B. hat ein Kunde keine Ahnung, wie Blumenhändler den Blumentransport bewerkstelligt
 - Kundens Aufgabe war einzig und allein, ein für sein Problem geeignetes Objekt zu finden und ihm eine geeignete Botschaft zu senden
 - Für einen Kunden ist zudem wichtig, zu wissen, wie er die Botschaft für den Blumenhändler formulieren muss
- Eine Methode ist die Implementierung eines Algorithmus



Klassen und Objekte

- Objekte kann man in Gruppen (Klassen) einteilen
- Eine Klasse ist eine Definition eines bestimmten Typs von Objekten
 - ein Bauplan indem die Methoden und Attribute beschrieben werden
- Nach diesem Schema können Objekte (Instanzen) einer Klasse erzeugt werden
- Ein Objekt ist eine Konkretisierung einer Klasse
- Alle Instanzen einer Klasse sind von der Struktur (Methoden/Attribute) her gleich
 - sie unterscheiden sich allein in der Belegung ihrer Attribute mit Werten

Klassen und Objekte

- Im Prinzip ist eine Klasse eine Sorte (Wertebereich)
 - Die Objekte der Klasse sind die „Literale“ also Werte der Sorte
- Andersrum kann auch eine Sorte wie int als eine Klasse von Objekten (eine Teilmenge der ganzen Zahlen) aufgefasst werden; das einzige Attribut ist der Wert, der durch das Literal dargestellt wird
- Jedes Objekt (Literal) aus int unterscheidet sich durch ein den Wert dieses Attributs
- Die „Klasse“ stellt verschiedene „Methoden“ bereit, die von allen Objekten „beherrscht“ werden
- Klassen heißen auch benutzereigene (abstrakte) Datentypen, da sie Mengen von Objekten mit Funktionalitäten zur Verfügung stellen

Zusammenfassung

- Eine Klasse definiert die Attribute und Methoden ihrer Objekte
- Der Zustand eines konkreten Objekts wird durch seine Attributwerte und Verbindungen (Links) zu anderen konkreten Objekten bestimmt
- Das mögliche Verhalten eines Objekts wird durch die Menge von Methoden beschrieben

Die Basis von allem:

das Objekt (= Daten + Funktionalität)



Prinzipien der OOP

Abstraktion

- zeigt nur die relevanten Details

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

Abstraktion

- Jedes erzeugte Objekt einer Klasse hat seine eigene Identität
- Beispiel: Lichtschalter in einem Haus
 - Alle Lichtschalter sind gleich zu bedienen.
 - Alle Lichtschalter sind gleich konstruiert.
 - Dennoch unterscheidet sich der Lichtschalter für die Flurbeleuchtung vom Lichtschalter fürs Badezimmer: Es ist nicht derselbe Lichtschalter
- Statt jeden einzelnen Lichtschalter neu zu modellieren/programmieren, abstrahiert man eine Klasse „Lichtschalter“, der alle Funktionalitäten nur einmal (für alle möglichen Lichtschalter) implementiert
- Abstraktion hilft, Details zu ignorieren, und reduziert damit die Komplexität des Problems. Dadurch werden komplexe Apparate und Techniken beherrschbar.

Kapselung

- Eine Klasse ist die Zusammenfassung einer Menge von Daten (Attributen) und darauf operierender Funktionen (Methoden)
- Die Attribute werden für jedes konkrete Objekt neu angelegt bzw. belegt
 - der Zustand der Objekte
- Die Methoden sind nur einmal realisiert / definiert und operieren bei jedem Aufruf auf den Daten eines bestimmten konkreten Objekts
 - das Verhalten der Objekte
- Kapselung bedeutet, dass (von gewollten Ausnahmen abgesehen) die Methoden die einzige Möglichkeit darstellen, mit einem konkreten Objekt zu kommunizieren und so Informationen über dessen Zustand zu gewinnen oder diesen zu verändern

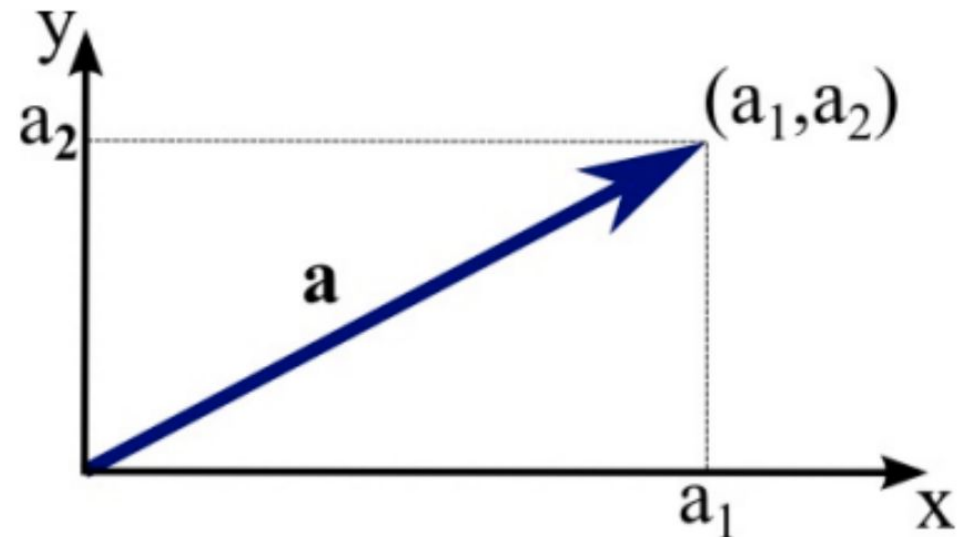
Kapselung

- Kapselung hilft, die Komplexität der Bedienung eines Objekts zu reduzieren
- Durch Kapselung werden die Implementierungsdetails von Objekten verborgen
- Dadurch können Daten nicht bewusst oder versehentlich verändert werden
- Kapselung ist daher auch ein wichtiger Sicherheitsaspekt: Ein direkter Zugriff auf die Daten wird unterbunden, der Zugriff erfolgt nur über definierte Schnittstellen, die bereitgestellten Methoden
- Beispiel: welche Attribute den Zustand eines Lichtschalters definieren kann uns egal sein, solange wir wissen, wie wir den Lichtschalter bedienen müssen; tatsächlich können wir den Zustand eines Lichtschalters nur über dessen „Schnittstelle“ verändern (indem wir den Schalter drücken oder drehen)



Beispiel - 2D Vector

- Attribute/Welche Eigenschaften hat jeder Vektor
 - Punkt
 - x
 - y
- Verhalten/Welche Methoden stellt jeder Vektor bereit
 - add
 - mul
 - sub
 - drehen



Beispiel - 2D Vector

```
import math

class Vector2D:
    """A two-dimensional vector with Cartesian coordinates."""

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        """Human-readable string representation of the vector."""
        return '{:g}i + {:g}j'.format(self.x, self.y)

    def __repr__(self):
        """Unambiguous string representation of the vector."""
        return repr((self.x, self.y))

    def dot(self, other):
        """The scalar (dot) product of self and other. Both must be vectors."""

        if not isinstance(other, Vector2D):
            raise TypeError('Can only take dot product of two Vector2D objects')
        return self.x * other.x + self.y * other.y

    def __sub__(self, other):
        """Vector subtraction."""
        return Vector2D(self.x - other.x, self.y - other.y)

    def __add__(self, other):
        """Vector addition."""
        return Vector2D(self.x + other.x, self.y + other.y)

    def __mul__(self, scalar):
        """Multiplication of a vector by a scalar."""

        if isinstance(scalar, int) or isinstance(scalar, float):
            return Vector2D(self.x*scalar, self.y*scalar)
        raise NotImplementedError('Can only multiply Vector2D by a scalar')
```

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
```

```
>>>
```

Objekt

Zustand

Verhalten

```
>>> #Listen
```

```
>>> l = [1, 2]
```

```
>>> l.__add__([2]) # l + [2]
```

```
[1, 2, 2]
```

```
>>> l
```

```
[1, 2]
```

```
>>>
```

Objekt

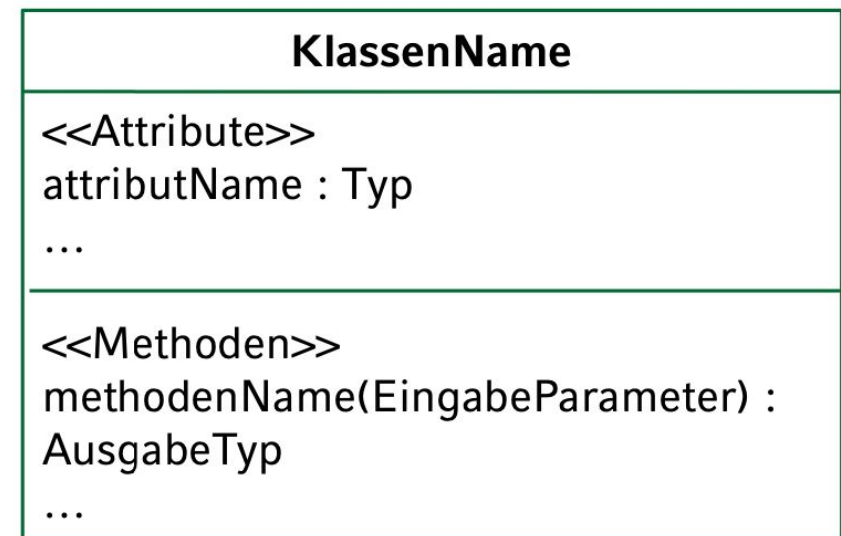
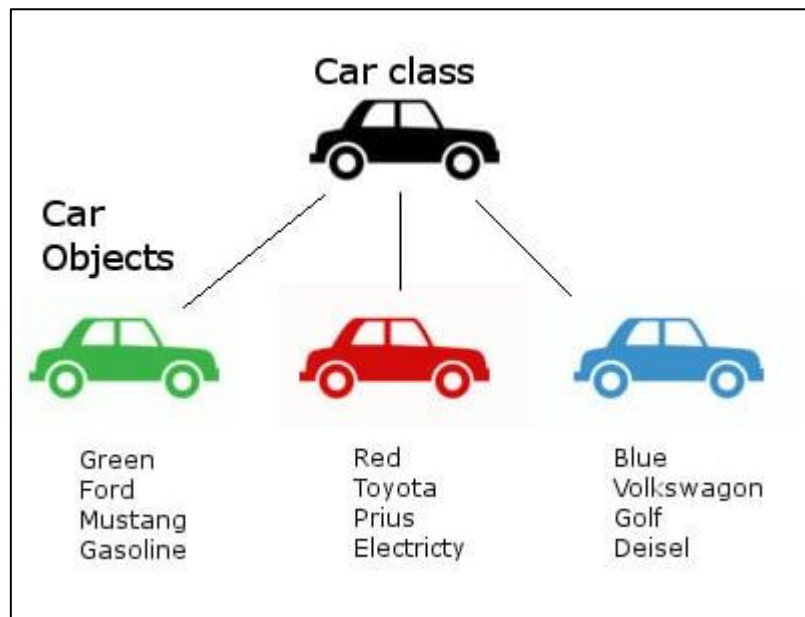
Zustand

Verhalten

das Objekt (= Daten + Funktionalität)

Klassen

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



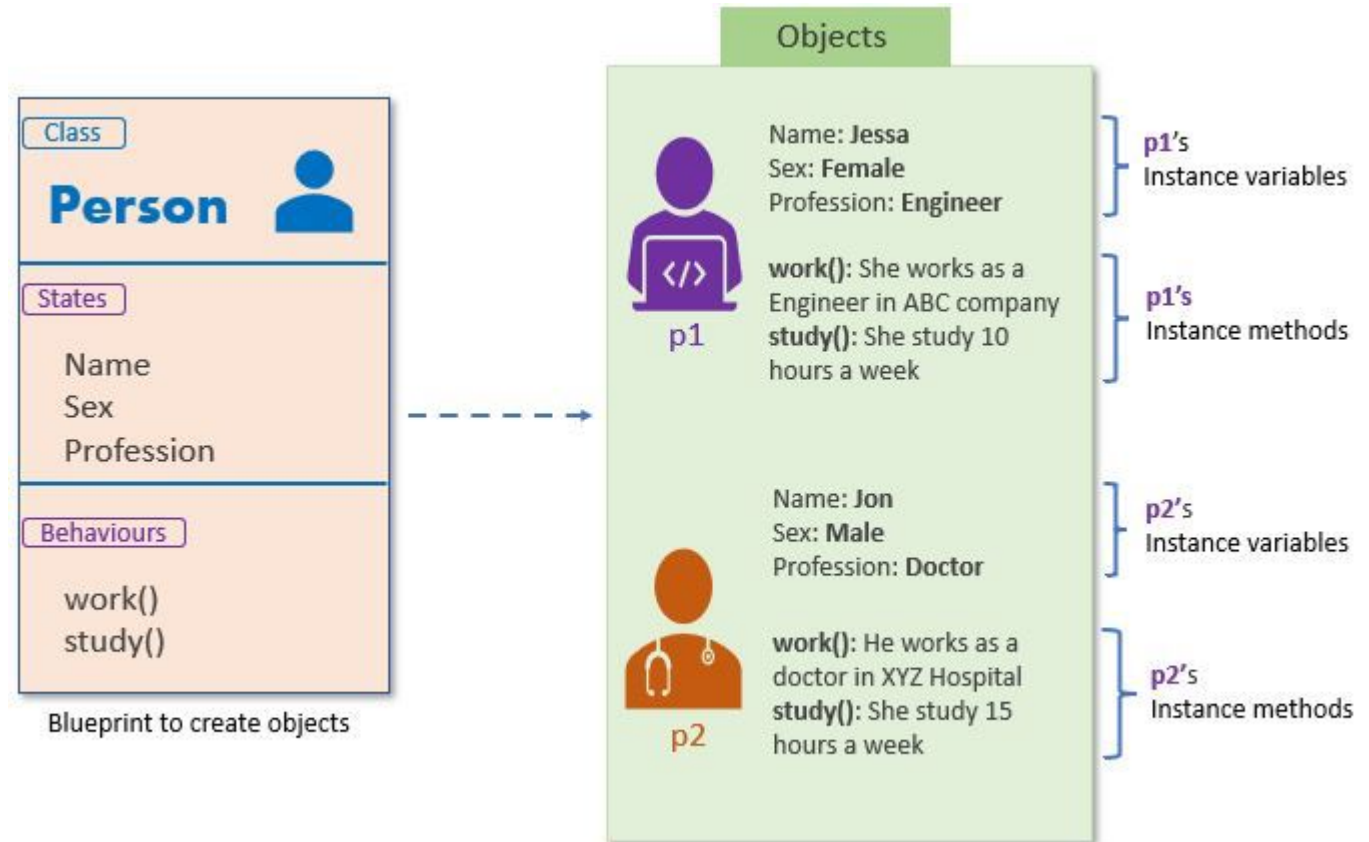
Klassen beschreiben

- Attribute (**Eigenschaften**)
- und Methoden (**Verhaltensweisen**) der Objekte

Klassendefinition in UML

Klassen

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

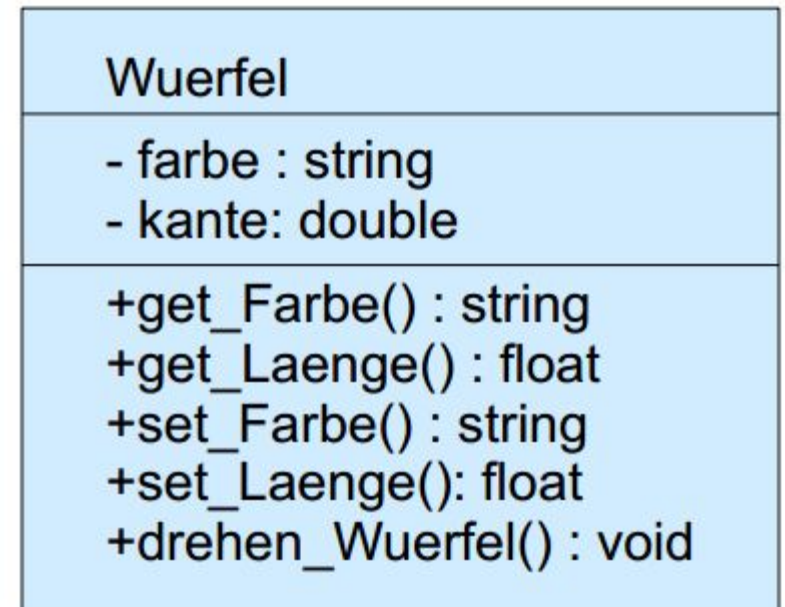


Klassen beschreiben

- Attribute (**Eigenschaften**)
- und Methoden (**Verhaltensweisen**) der Objekte

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.



Klassen

Definition in Python

- 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>
```

Konstruktor

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

```
x = MyClass()
```

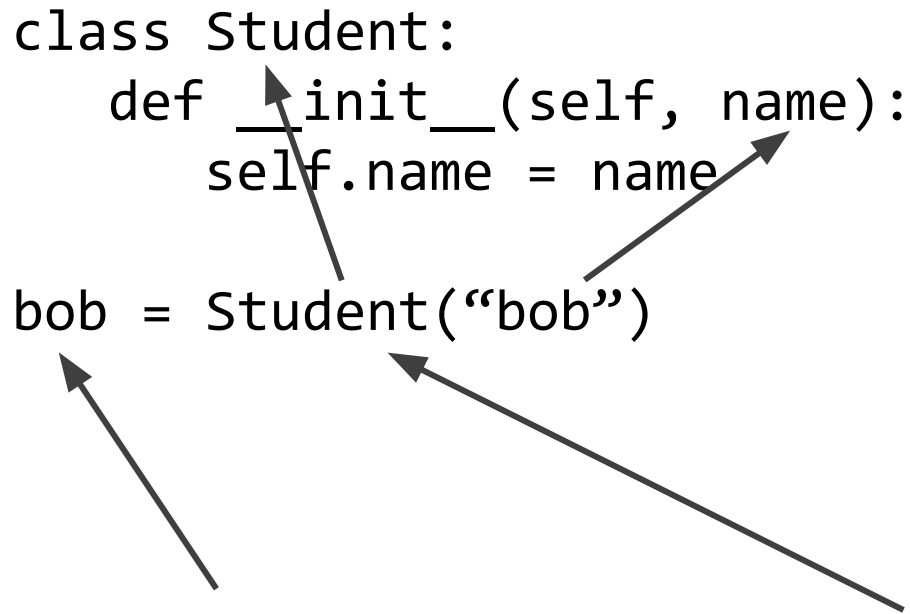
Jedes Objekt hat einen eigenen Namespace. Namen darin heißen Attribute des Objekts.

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


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 'Student' class name in the code to the 'Student' part of the 'Student' class definition. Another arrow points from the 'Student' part of the 'Student' class definition to the 'Student' part of the 'Student' constructor call 'Student("bob")'. A third arrow points from the 'bob' variable in the line 'bob = Student("bob")' to the 'Student' part of the constructor call. A fourth arrow points from the 'bob' variable to the 'bob' variable in the line 'bob = Student("bob")'.

(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

Definition mit self innerhalb des Konstruktors

- `self.attr = val`

```
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]

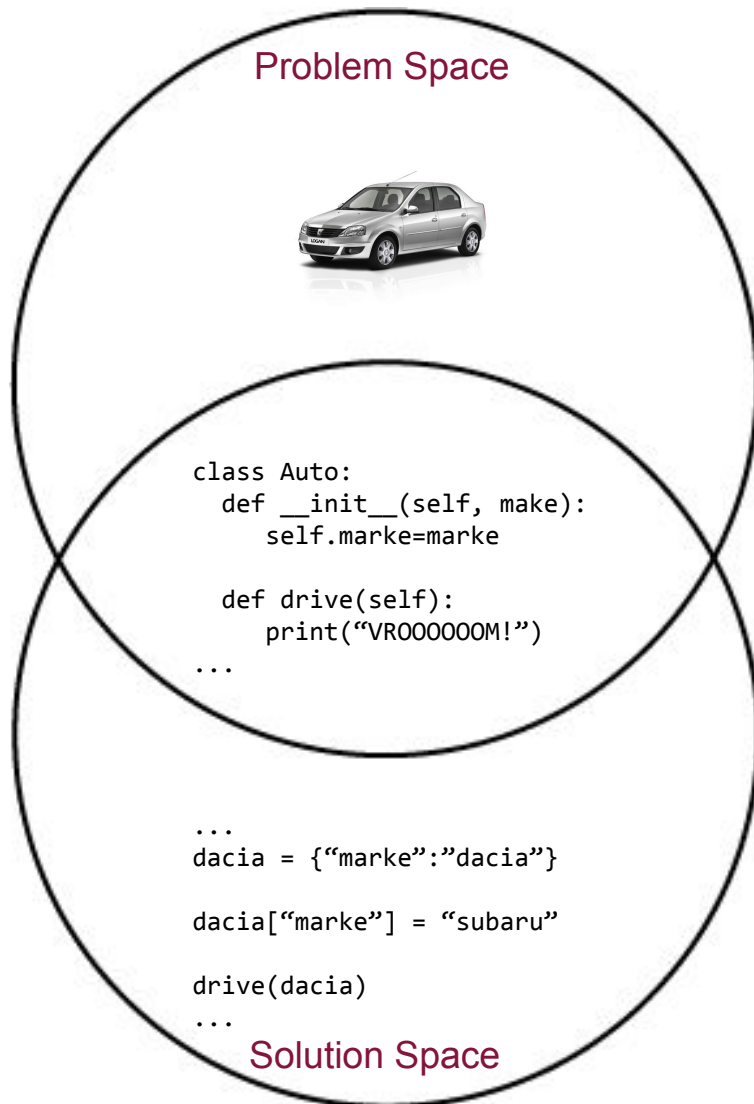
    def getDenominator(self):
        """
        Getter method
        return the denominator of the rational number
        """
        return self.__nr[1]

    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

`self` 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)
```

äquivalent zu...

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

Implizit als erster Argument gegeben

Beispiel

- Implementiere eine Klasse **Die**
 - Attribute: sides
 - Methods: roll()
- Implementiere eine Klasse **Player**
 - Attribute: dice
 - Methods: play()

Spezifikation

