



Übungslektion 5 - Klassen & Programmierkonzepte

Informatik II

17. März 2026

Willkommen!

Polybox



Passwort: jschul

Personal Website



https://jschultev.github.io/personal_website/

Heutiges Programm

Klassen

Programming Concept

Gruppenübung

Hausaufgaben

1. Klassen

- Klassen und Objekte
- Magische Methoden
- Vererbung

Klassen und Objekte

- Mit steigender Komplexität ist es wichtig, Programme zu strukturieren
- Klasse: Datentypdefinition mit Attributen und Methoden.
- Objekte: Instanzen einer bestimmten Klasse.

Student

- name
- grade
- sleep
- eat
- study

Klassen und Objekte

```
class Student:
    def __init__(self):
        self.name = ''
    def sleep(self, hours):
        print(self.name, 'slept for', hours, 'hours')
    def eat(self, food):
        print(self.name, 'ate', food)
    def study(self, hours, subject):
        print(self.name, 'studied', subject, 'for', hours, 'hours')
```

```
s = Student()
s.name = 'John'
s.sleep(7)
```

John slept for 7 hours

Übung 2: Klassen

- Was ist die Ausgabe?

```
class Dummy:  
    def __init__(self, n):  
        self.n = n
```

```
    def update(self, m):  
        self.n += m
```

```
d = Dummy(2)
```

```
d.update(3)
```

```
print(d.n)
```

d.n = 2

→ d.n += 3

⇒ d.n = 5

5

Übung 2: Klassen

- Was ist die Ausgabe?

```
class Dummy:  
  
    def __init__(self, n):  
        self.n = n  
  
    def method1(self, n):  
        return self.n * 2
```

```
d = Dummy(4) — d.n = 4  
res = d.method1(3) —— return 4 * 2 = 8  
print(res)
```

8

Übung 3: Definition einer Klasse

- Erstellen Sie eine Klasse **Rectangle** mit den folgenden Attributen und Methoden.
- Erstellen Sie ein Objekt mit Breite 4 und Höhe 5 und verwenden Sie die Methode **area**, um seine Fläche zu berechnen.

Rectangle

■ **width**

■ **height**

■ **area**

Übung 3: Definition einer Klasse

```
class Rectangle:  
    def __init__(self, width, height):  
        self.width = width  
        self.height = height  
    def area(self):  
        return self.width * self.height
```

Übung 3: Definition einer Klasse

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
rect = Rectangle(4, 5)
print(rect.area())
```

20

Magische Methoden

- Die sogenannten *magischen Methoden* definieren Standardoperatoren, die eine Klasse definieren kann
- Dafür muss lediglich die entsprechende Methode implementiert werden

Beispiele

- Vergleiche: `__eq__`, `__lt__`, `__gt__`, ...
- Arithmetik: `__add__`, `__sub__`, `__iadd__`, ...
- Output: `__str__`
- Länge: `__len__`

Magische Methoden auf ZF

Operator	Bedeutung	Magische Methoden	Operator	Bedeutung	Magische Methode
<	kleiner als	<code>__lt__</code>	+, +=	Addition	<code>__add__</code> , <code>__iadd__</code>
<=	kleiner gleich	<code>__le__</code>	-	Subtraktion	<code>__sub__</code>
>	größer als	<code>__gt__</code>	*	Multiplikation	<code>__mul__</code>
>=	größer gleich	<code>__ge__</code>	/	Division	<code>__truediv__</code>
==	gleich	<code>__eq__</code>	//	Ganzzahldivision	<code>__floordiv__</code>
!=	ungleich	<code>__ne__</code>	%	Modulo (Rest)	<code>__mod__</code>
			**	Exponentiation	<code>__pow__</code>

Beispiel zu den Magic Methods:

```
1 class class_name:
2     ...
3     def __lt__(self, other): # kleiner als
4         return ...
5
6     def __add__(self, other): # Addition
7         return ...
```

Loris Frey

Um gewisse Operationen für Klassen zu definieren, wird auf die **Magic Methods** zugegriffen. Einige Beispiele:

<	less than	<code>__lt__</code>
>=	greater than or equal	<code>__ge__</code>
==	equal to	<code>__eq__</code>
+	addition	<code>__add__</code>
**	exponentiation	<code>__pow__</code>
<code>print(...)</code>	printing	<code>__str__</code>

Max Schaldach

Beispiel: Output

- Die Standardausgabe für eigene Klassen ist nicht sehr hilfreich.

```
class Rectangle:
```

```
    ...
```

```
rect = Rectangle(m, n)
```

```
print(rect)
```

```
<__main__.Rectangle object at 0x0000020B02DF17C0>
```

Beispiel: Output

- Um dies zu verändern, können wir die magische Methode `__str__` implementieren.

```
class Rectangle:  
    def __str__(self):  
        return f'Rectangle: {self.width} x {self.height}'
```

```
rect = Rectangle(4, 5)  
print(rect)
```

```
Rectangle: 4 x 5
```

Übung 4: Magische Methoden

- Was ist die Ausgabe?

```
class Rectangle:
    ...

    def __add__(self, other):
        width = other.width + self.width
        height = other.height + self.height
        return Rectangle(width, height)

a = Rectangle(4, 5)
b = Rectangle(2, 3)
c = a + b
print(c)
```

Rectangle: 6 x 8

Übung 4: Magische Methoden

- Was ist die Ausgabe?

```
class Rectangle:  
    ...  
  
    def __add__(self, other):  
        width = other.width + self.width  
        height = other.height + self.height  
        return Rectangle(width, height)
```

```
a = Rectangle(4, 5)  
b = Rectangle(2, 3)  
c = a + 3  
print(c)
```

```
def __add__(self, other):  
    if isinstance(other, int):  
        width = self.width + other  
        height = ... height + other  
    if isinstance(
```

AttributeError: 'int' object has no attribute 'width'

Vererbung: Motivation

- Vererbung drückt eine Beziehung von Klassen aus.
- Eine Subklasse (erbende Klasse) beinhaltet dabei die Attribute und Methoden der Elternklasse (Basisklasse)

Beispiel

- Basisklasse: **Precipitation**
- Attribute: **amount**
- Methoden: **__str__**, **alarm**
- Subklassen: **Rain**, **Snow**

Beispiel

- Beziehung: Ein Quadrat **ist** ein Rechteck.

```
class Square(Rectangle):  
  
    def __init__(self, n):  
        Rectangle.__init__(self, width = n, height = n)  
  
s = Square(2)  
print(s)
```

```
Rectangle: 2 x 2
```

Übung 5: Vererbung

Was ist der Output von folgendem Code

```
a = Square(n = 4)
b = Rectangle(4, 2)
c = a + b
print(c, type(c))
```

```
Rectangle: 8 x 6 <class '__main__.Rectangle'>
```

Übung 5: Vererbung

Was ist der Output von folgendem Code

```
a = Square(4)
b = Square(2)
c = a + b
print(c, type(c))
```

```
Rectangle: 6 x 6 <class '__main__.Rectangle'>
```

Übung 5: Vererbung

Was ist der Output von folgendem Code

```
class Square(Rectangle):
    ...
    def __add__(self, other):
        if isinstance(other, Square):
            return Square(self.width + other.width)
        else:
            return other + self

a = Square(4)
b = Rectangle(3, 2)
c = a + b
print(c, type(c))
```

Rectangle: 7 x 6 <class '__main__.Rectangle'>

Übung 5: Vererbung

Was ist der Output von folgendem Code

```
class Square(Rectangle):
    ...
    def __add__(self, other):
        if isinstance(other, Square):
            return Square(self.width + other.width)
        else:
            return other + self

a = Square(n = 4)
b = Square(n = 2)
c = a + b
print(c, type(c))
```

Rectangle: 6 x 6 <class '__main__.Square'>

2. Programing Concept

- Compile vs. Interpret
- Static vs. Dynamic Typing
- Generic Programming
- Functional Programming
- Lambda Expression

Kompilieren vs. Interpretieren

Kompilierte Programmiersprachen (z.B. C / C++)

- Programmcode wird von einem Compiler (z.B. GCC) in Assemblercode kompiliert;
- Assemblercode wird ausgeführt; Der Programmcode sollte rekompiliert werden sofern Änderungen vorgenommen wurden.

Interpretierte Programmiersprachen (z.B. Python oder JavaScript)

- Der Interpreter liest den Programmcode und führt ihn direkt aus;
- Die Interpretation wird bei jeder Ausführung des Programmcodes wiederholt;

Kompilierte Sprachen haben üblicherweise eine bessere Performance.

Kompilieren vs. Interpretieren: Ein Beispiel

Führe ein C++-Programm aus, nachdem es modifiziert wurde.

```
Kompilieren  
Ausfuehren  
Modifizieren  
Kompilieren  
Ausfuehren
```

Führe ein Python-Programm aus, nachdem es modifiziert wurde.

```
Interpretieren und Ausfuehren  
Modifizieren  
Interpretieren und Ausfuehren
```

Statische vs. Dynamische Typisierung

Statische Typisierung (z.B. C / C++)

- Jedes Element hat einen vom Programmierer **definierten Typ**;
- Ob die gewählten Typen korrekt zusammenpassen, wird bei der Kompilation überprüft, sodass Kompilierungsfehler erzeugt werden, wenn dies nicht der Fall ist.

Dynamische Typisierung (z.B. Python)

- Elemente haben im Vorhinein **keinen festgelegten Typ**;
- Der Typ wird erst zur Laufzeit gewählt;
- Der Typ ist zur Laufzeit veränderlich;
- Abhängig vom Typ während der Ausführung kann es zu Laufzeit-Fehlern kommen;

Dynamic Typing: Ein Beispiel

Welche Outputs werden generiert?

```
def add_integers(l): # l has no type
    return sum(l[:2])
```

```
print(add_integers([1, 2, 3, 4, 5]))
print(add_integers([1, 2, 'a', 4, 5]))
```

9

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Generische Programmierung

Schreibe Code (Funktionen, Klassen), der so allgemein wie nur möglich einsetzbar ist.

Python unterstützt die generische Programmierung.

```
def add(x, y):  
    return x + y
```

Die Funktion **add** ist anwendbar für alle Typen, für die die "+"-Operation definiert ist.

Funktionale Programmierung

Funktionen können als Argumente an andere Funktionen weitergegeben werden.

```
def increase_one(n):  
    return n + 1  
  
def applyToAll(function, container):  
    return [function(elem) for elem in container]  
  
number = [1, 2, 3, 4, 5]  
# increase_one is passed as an argument  
print(applyToAll(increase_one, number))
```

Output: ?

[2, 3, 4, 5, 6]

Funktionale Programmierung: map

map: Eine eingebaute Python-Funktion, die eine Funktion auf jedes Element eines Iterable (z.B. list oder set) anwendet und ein neues Iterable mit den Resultaten zurückgibt.

```
def increase_one(n):  
    return n + 1  
  
new_number = map(increase_one, [1, 2, 3, 4, 5])  
print(new_number)  
print(list(new_number))
```

Output: ?

```
<map object at 0x7f8c6c6a5ab0>  
[2, 3, 4, 5, 6]
```

Funktionale Programmierung: map

Die als Input übergebene Funktion kann mehrere Argumente besitzen. In diesem Fall muss `map` die selbe Anzahl an Iterables als Argumente übergeben werden. Die Länge des zurückgegebenen Iterables ist gleich der Länge des **KÜRZESTEN** Input-Iterables.

```
def add(x, y):  
    return x + y  
  
number1 = [1, 2, 3, 4, 5, 6] # length: 6  
number2 = [7, 8, 9, 10, 11] # length: 5  
new_number = map(add, number1, number2)  
print(list(new_number))
```

Output: ?

```
[8, 10, 12, 14, 16]
```

Funktionale Programmierung: `filter`

filter: Eine eingebaute Python-Funktion, die aufgrund einer gegebenen Funktion Elemente aus einem Iterable (z.B. list oder set) filtert und diese gefilterten Elemente in einem neuen Iterable zurückgibt.

```
def is_even(n):  
    return n % 2 == 0  
  
even_number = filter(is_even, [1, 2, 3, 4, 5])  
print(even_number)  
print(list(even_number))
```

Output: ?

```
<filter object at 0x7f4604b89b70>  
[2, 4]
```

Filter gleich wie map...?

```
def is_even(n):  
    return n % 2 == 0  
  
even_number = map(is_even, [1, 2, 3, 4, 5])  
print(even_number)  
print(list(even_number))
```

Output: ?

```
<map object at 0x000001F48A2019C0>  
[False, True, False, True, False]
```

Filter gleich wie map...?

Definitionen:

- **map**: Eine eingebaute Python-Funktion, die eine Funktion auf jedes Element eines Iterable (z.B. list oder set) anwendet und **ein neues Iterable mit den Resultaten zurückgibt**.
- **filter**: Eine eingebaute Python-Funktion, die aufgrund einer gegebenen Funktion Elemente aus einem Iterable (z.B. list oder set) filtert und **diese gefilterten Elemente in einem neuen Iterable zurückgibt**.

Funktionale Programmierung: reduce

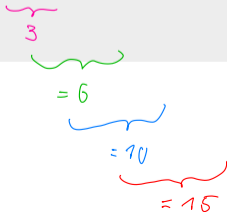
reduce: Eine eingebaute Python-Funktion im **functools**-Modul, das eine gegebene Funktion in kumulativer Weise auf die Elemente eines gegebenen Iterables anwendet und einen einzelnen Wert zurückgibt.

```
from functools import reduce
def add(x, y):
    return x + y

sum = reduce(add, [1, 2, 3, 4, 5])
print(sum)
```

Output: ?

15



Lambda-Ausdrücke

(Kleine) Funktionen ohne expliziten Namen. Notation:

```
lambda arguments : expression
```

Example:

```
lambda : print('Hello World') # no argument  
lambda x, y : x + y # two arguments x and y, return x + y
```

Lambda Expression

Verwende Lambda-Ausdrücke in **reduce**:

```
from functools import reduce

sum = reduce(lambda x, y : x + y, [1, 2, 3, 4, 5])
print(sum)
```

Output: ?

15

3. Gruppenübung

Liste von Objekten (Je nach Zeit)

```
names = ['Anne', 'Ben', 'Charles', 'David', 'Elena', 'Fiona']
```

Erstellen Sie eine Liste **students** von **Student**-Objekten mit den entsprechenden Namen von **names**.

Liste von Objekten (Je nach Zeit)

```
names = ['Anne', 'Ben', 'Charles', 'David', 'Elena', 'Fiona']
```

Erstellen Sie eine Liste **students** von **Student**-Objekten mit den entsprechenden Namen von **names**.

```
students = [Student() for name in names]
for student, name in zip(students, names):
    student.name = name
```

Liste von Objekten (Je nach Zeit)

```
names = ['Anne', 'Ben', 'Charles', 'David', 'Elena', 'Fiona']
```

Erstellen Sie eine Liste `students` von `Student`-Objekten mit den entsprechenden Namen von `names`.

```
students = [Student() for name in names]
for student, name in zip(students, names):
    student.name = name
```

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

Liste von Objekten

```
names = ['Anne', 'Ben', 'Charles', 'David', 'Elena', 'Fiona']
```

Lassen Sie alle Studierenden in **students** 9 Stunden schlafen, 6 Stunden Informatik lernen und dann eine Pizza essen.

Liste von Objekten

```
names = ['Anne', 'Ben', 'Charles', 'David', 'Elena', 'Fiona']
```

Lassen Sie alle Studierenden in **students** 9 Stunden schlafen, 6 Stunden Informatik lernen und dann eine Pizza essen.

```
for student in students:  
    student.sleep(9)  
    student.study(6, 'Informatik')  
    student.eat('pizza')
```

Gruppenübung 5: Departments

Definiere Klassen, kreierte Objekte und benutze "magische" Methoden.

- Definiere Klassen (Aufgabe 1 & 2);
- Definiere "magische" Funktionen (Aufgaben 3 & 5)
- Kreiere Objekte (Aufgabe 4);
- Verwende "magische" Methoden (Aufgabe 6).

Mehr dazu in der detaillierten Aufgabenbeschreibung auf Code Expert.

[https://expert.ethz.ch/enrolled/~~SS25~~/mavt2/codeExamples](https://expert.ethz.ch/enrolled/SS25/mavt2/codeExamples)
SS26

4. Hausaufgaben

Aufgabe 4: Klassen und Programmier-Konzepte

Einfach, Mittel, Schwer

- Immobilien
- Erdbeben-Datenbank
- Studenten und Professoren

KEINE HART-GEODETEN LÖSUNGEN

Fragen?

Feedback



https://jschultev.github.io/personal_website/Feedback

Kurzer Einschub

Es gibt einige Objekte (z.B range, map) in Python, die "Lazy" sind. Beim Initialisieren werden nur die Attribute gespeichert. Erst beim Aufrufe werden diese dann "ausgeführt". Das spart Speicher!

```
def f(l):  
    return 2  
a = range(1,6)  
b = map(f, a)  
print(a, ", " b)  
print(list(b))
```

Kurzer Einschub

Es gibt einige Objekte (z.B range, map) in Python, die "Lazy" sind. Beim Initialisieren werden nur die Attribute gespeichert. Erst beim Aufrufe werden diese dann "ausgeführt". Das spart Speicher!

```
def f(l):  
    return 2  
a = range(1,6)  
b = map(f, a)  
print(a, ", " b)  
print(list(b))
```

```
range(1, 6), <map object at 0x000001F48A1A88B0>  
[2, 2, 2, 2, 2]
```