ETH zürich

Übungslektion 5 - Klassen & Programmierkonzepte

Informatik II

17. März 2026

Willkommen!

Polybox

Passwort: jschul

Personal Website

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

Heutiges Programm

Klassen

Programing 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)
```

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)
res = d.method1(3)
print(res)
```

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
```

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
<	kleiner als	__lt__
<=	kleiner gleich	__le__
>	grösser als	__gt__
>=	grösser gleich	__ge__
==	gleich	__eq__
!=	ungleich	__ne__

Operator	Bedeutung	Magische Methode
+, +=	Addition	__add__, __iadd__
-	Subtraktion	__sub__
*	Multiplikation	__mul__
/	Division	__truediv__
//	Ganzzahldivision	__floordiv__
%	Modulo (Rest)	__mod__
**	Exponentiation	__pow__

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	__lt__
>=	greater than or equal	__ge__
==	equal to	__eq__
+	addition	__add__
**	exponentiation	__pow__
print(…)	printing	__str__

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)
```

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)
```

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 Qudrat **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))
```

Übung 5: Vererbung

Was ist der Output von folgendem Code

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

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))
```

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))
```

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 in 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
Modizifieren
    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]))
```

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** is 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: ?

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: ?

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: ?

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 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: ?

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: ?

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: ?

3. Gruppenübung

Liste von Objekten (Je ch Zeit)

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

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

```
# TODO
```

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.

```
# TODO
```

Gruppenübung 5: Departments

Definiere Klassen, kreiere 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

4. Hausaufgaben

Aufgabe 4: Klassen und Programmier-Konzepte

Einfach, Mittel, Schwer

- Immobilien
- Erdbeben-Datenbank
- Studenten und Professoren

KEINE HART-GECODETEN LÖSUNGEN

Fragen?

Feedback

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

Add - On

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