



Felix Friedrich & Hermann Lehner

# Informatik II

Vorlesung am D-BAUG der ETH Zürich

Frühjahr 2020

# 1. Einführung

---

Ziele der Vorlesung

# Ziele der Vorlesung

- Verständnis des Entwurfs und der Analyse grundlegender Algorithmen und Datenstrukturen.
- Fähigkeit, korrekte und ausreichend effiziente Programme entwickeln, um eine klar formulierte Problemstellung zu lösen.

# Inhalte der Vorlesung

---

Software Engineering

Java To Python Introduction  
Python Datastructures

---

Datenstrukturen / Algorithmen

Begriff der Invariante, Kostenmodell, Landau Symbole

Algorithmenentwurf, Induktion, Divide & Conquer

Suchen, Sortieren

Wörterbücher: Hashing und Suchbäume, Balancierte Bäume

Dynamische Programmierung

Fundamentale Graphenalgorithmen

Kürzeste Wege, Maximaler Fluss

## 2. Von Java nach Python

---

Erstes Python Programm, Transfer Java → Python, Dynamische Datenstrukturen in Python

# Lernziele

- Erlernen einer neuen Programmiersprache (Python). Lernen, wie man von einer Programmiersprache zur anderen wechselt.
- Kennenlernen der wichtigsten Unterschiede zwischen Java und Python, sowohl aus syntaktischer wie auch aus semantischer Sicht.
- Erlernen der fundamentalen Datentypen von Python (list, set, dict und tuple) und der darauf anwendbaren Operationen.
- Gewöhnen an die neue Programmiersprache und Umgebung (Python), indem bekannte Algorithmen erneut implementiert werden.

# Erstes Java Programm

```
public class Hello {  
    public static void main (String[] args) {  
        System.out.print("Hello World!");  
    }  
}
```

# Erstes Python Programm

```
print("Hello World!")
```



# Kommentare

Kommentaren wird ein # vorangestellt:

```
# prints 'Hello World!' to the console  
print("Hello World!")
```

# Formattierung ist relevant: Anweisungen

- Whitespace (Space, Enter, Tab) ist relevant!
- Jede Zeile steht für eine Anweisung
- Also **genau eine** Anweisung pro Zeile
- Kommentare starten mit einem #

Beispiel-Program mit zwei Anweisungen:

```
# two print-statements
print("Hurray, finally ...")
print("... no Semicolons!")
```

# Formattierung ist Relevant: Blöcke

- Blöcke müssen eingerückt werden!
- Alle eingerückten Anweisungen gehören zum Block, Block endet, wo die Einrückung endet.
- Start eines Blocks ist markiert durch ein Doppelpunkt “:”

```
# in Python
while i > 0:
    x = x + 1 / i
    i = i - 1

print(x)
```

```
// in Java
while (i > 0) {
    x = x + 1.0 / i;
    i = i - 1;
}
System.out.print(x)
```

# Literale: Zahlen

- **integer**: 42, -5, 0x1b, 0o33, 7729684762313578932578932  
Beliebig genaue Ganzzahlen
- **float**: -0.1, 34.567e-4  
Wie **double** in Java, aber Präzision ist abhängig von der Plattform (CPU/Betriebssystem)
- **complex**: 2 + 3j, (0.21 - 1.2j)  
Komplexe Zahlen in der Form  $a+bj$ . Optional mit runden Klammern.

# Literale: Wahrheitswerte

- True
- False

# Literale: Zeichenketten

- 'a single quoted string\nand a second line'
- "a double quoted string\nand a second line"
- Multi-line strings (triple double quotes):

```
"""a multiline string  
and a second line"""
```

# Literale: Sequenzen

- **arrays:** Es gibt keine Arrays in Python
- **lists:** `[ 17, True, "abc" ]`, `[]`  
Mutierbare geordnete Sequenz von 0 oder mehr Werten mit beliebigen Typen.
- **tuples:** `(17, True, "abc")`, `(42, )`  
Nicht-änderbare geordnete Sequenz von 1 oder mehr Werten mit beliebigen Typen.

# Literale: Kollektionen

- **dicts:** { "a": 42, "b": 27, False: 0 }, {}

Mutierbarer Key-Wert Store. Keys und Werte mit beliebigen Typen.

- **sets:** {17, True, "abc"}, {42}

Mutierbare ungeordnete Sequenz von 0 oder mehr Werten mit beliebigen Typen. Keine Duplikate.



# Variablen

- Variablen werden automatisch erstellt bei der ersten Zuweisung
- Der Typ einer Variable wird bei Wertzuweisungen nicht geprüft. D.h. es können Werte von unterschiedlichen Typen einer Variable zugewiesen werden über Zeit.
- Zuweisung von Werten mittels Zuweisungsoperator: =
- Zuweisung an mehrere Variablen gleichzeitig mit Tupeln

```
a = "Ein Text"
print(a) # prints: Ein Text
a = 42
print(a) # prints: 42
```

```
x, y = 4, 5
print(x) # prints: 4
print(y) # prints: 5
```

# Variablen

- Variablen müssen immer erst zugewiesen werden, bevor diese ausgelesen werden können.

Angenommen, b wurde noch nie ein Wert zugewiesen.

```
a = b
```

Gibt folgenden Fehler

**NameError: name 'b' is not defined**

# Numerische und Boolean Operatoren

- Numerische Operatoren wie bei Java: `+`, `-`, `*`, `/`, `%`, `**`, `//`
- Achtung: `/` resultiert immer in einer Fließkommazahl
- `**`: Potenzierung,  $\mathbf{a**b = a^b}$ .
- `//`: Ganzzahl-Division, `5//2` ergibt `2`.
- Vergleichsoperatoren wie bei Java: `==`, `>=`, `<=`, `>`, `<`, `!=`
- Logische Operatoren: `and`, `or`, `not`
- Membership Operator: `in` Gibt an, ob ein Wert in einer Liste, einem Set oder einer Zeichenkette ist.
- Identitäts Operator: `is` Prüft, ob zwei Variablen auf das gleiche Objekt zeigen.

# Eingabe/Ausgabe

- Lesen von Eingaben mittels `input()`
- Übergabe eines Prompts möglich.
- Ausgabe mittels `print(...)`
- `print` nimmt eine oder mehrere Argumente entgegen und druckt diese mit einem Leerschlag dazwischen

```
name = input("What is your name: ")  
print("Hello", name)
```

# Eingabe/Ausgabe

- Eingabe wird immer als Zeichenkette gelesen.
- Um eine Zahl einzulesen, muss die Eingabe in eine Zahl konvertiert werden
- Es finden keine impliziten Konvertierungen statt
- Explizite Konvertierung mittels: `int()`, `float()`, `complex()`, `list()`,  
...

```
i = int(input("Enter a number: "))  
print("The", i, "th power of two is", 2**i)
```

# Bedingungen

- Keine Klammern nötig um den Test
- `elif` um einen weiteren Fall zu testen
- Einrückung beachten!

```
a = int(input("Enter a number: "))
if a == 42:
    print("Naturally, the answer")
elif a == 28:
    print("A perfect number, good choice")
else:
    print(a, "is just some boring number")
```

# While-Schleifen

Die bekannte Collaz-Folge

```
a = int(input("Enter a number: "))
while a != 1:
    if a % 2 == 0:
        a = a // 2
    else:
        a = a * 3 + 1
    print(a, end=' ')
```

# For-Schleifen

- For-Schleifen funktionieren anders als in Java
- Iteriert über die Elemente der angegebenen Menge

```
some_list = [14, 'lala', 22, True, 6]
total = 0;
for item in some_list:
    if type(item) == int:
        total += item
print("Total of the numbers is", total)
```



# For-Schleifen über einen Wertebereich

- Die Funktion `range(start, end, step)` erstellt eine Liste von Werten, beginnend mit `start`, bis zu `end` - exklusive. Schrittgrösse `step`.
- Schrittgrösse ist `1`, wenn das dritte Argument weggelassen wird.

```
# the following loop prints "1 2 3 4"
```

```
for i in range(1,5):  
    print(i, end=' ')
```

```
# the following loop prints "10 8 6 4 2"
```

```
for i in range(10, 0, -2):  
    print(i, end=' ')
```

# Methoden

## ■ Der Keks-Rechner revisited

```
def readInt(prompt, atleast = 1):
    """Prompt for a number greater 0 (or min, if specified)"""
    number = 0;
    while number < atleast:
        number = int(input(prompt))
        if (number < atleast):
            print("Too small, pick a number larger than", atleast)
    return number

kids = readInt("Kids: ")
cookies = readInt("Cookies: ", atleast=kids)
print("Each Kid gets", cookies // kids, "cookies.")
print("Papa gets", cookies % kids, "cookies.")
```

# Listen: Basisoperationen

- Elementzugriff (0-basiert): `a[2]` zeigt auf das dritte Element.
- Negative Indizes zählen von hinten!

```
a = [ 3, 7, 4]
```

```
print(a[-1]) # prints '4'
```

- Wert hinten hinzufügen: `a.append(12)`
- Test ob ein Element in einer Collection ist:

```
if 12 in a:
```

```
    print('12 is in the list, we just added it before')
```

- Anzahl Elemente in einer Collection: `len(a)`

# Listen: Slicing

- Slicing: Teilbereiche adressieren: `a[start:end]`
- `a` und/oder `b` sind positive oder negative Indizes.
- `end` ist nicht inklusive

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a[2:4])      # [3, 4]
print(a[3:-3])    # [4, 5, 6]
print(a[-3:-1])   # [7, 8]
print(a[5:])      # [6, 7, 8, 9]
print(a[:3])      # [1, 2, 3]
```

# Dictionaries

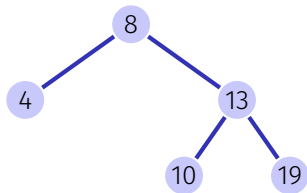
Dictionaries sind sehr wichtige primitive Datenstrukturen in Python

- Einfache und effiziente Möglichkeit, mehrere Datenfelder zu benennen und zusammenzufassen
- Aufbau von hierarchischen Datenstrukturen durch Verschachtelung
- Zugriff auf Elemente mittels `[]` Operator

```
record = { 'Firstname': 'Hermann', 'Lastname': 'Lehner',  
          'Salary': 420000, 'Mac User': True }  
record['Salary'] = 450000  
if record['Mac User']:  
    print('... one more thing!')
```

# Dynamische Datenstrukturen mit Dicts

```
tree = {  
    'key': 8,  
    'left' : {  
        'key': 4, 'left' : None, 'right': None  
    },  
    'right': {  
        'key': 13,  
        'left' : {  
            'key': 10, 'left' : None, 'right': None  
        },  
        'right': {  
            'key': 19, 'left' : None, 'right': None  
        }  
    }  
}
```



# Dynamische Datenstrukturen mit Dicts

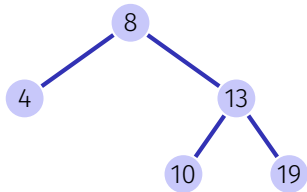
## ■ Arbeiten mit Dicts (Beispiele)

```
l = tree['left'] # assign left subtree to variable l
l['key'] = 6     # changes key from 4 to 6
```

```
if l['left'] is None: # proper way to test against None
    print("There is no left child here...")
else:
    print("Value of left subtree is", l['left']['key'])
```

# Dynamische Datenstrukturen mit Klassen

```
class Node:  
    def __init__(self, k, l=None, r=None):  
        self.key, self.left, self.right = k, l, r
```



```
# create the tree depicted on the right  
rightSubtree = Node(13, l=Node(10), r=Node(19))  
tree = Node(8, l=Node(4), r=rightSubtree)
```

```
# an example query  
print(tree.right.right.key) # prints: 19
```



# Module

Python hat eine extrem grosse Anzahl Libraries in Form von Modulen, die importiert werden können.

- Importieren eines ganzen Modules:

```
import math  
x = math.sqrt(4)
```

```
from math import *  
x = sqrt(4)
```

- Importieren von Teilen eines Modules:

```
from datetime import date  
t = date.today()
```