Data Science 1

Tutorial Day

Prof. Dr. Christian Bockermann

Hochschule Bochum

Sommersemester 2025

Tutorial Day – Ablauf

- Keine neuen Inhalte
- Wiederholung der wichtigstens Punkte
- Praktische Übungen

Tutorial Day – Ablauf

- Keine neuen Inhalte
- Wiederholung der wichtigstens Punkte
- Praktische Übungen

Zeitplan

- Wiederholung + Python Basics
- PAUSE
- Python mit LEGO

Der Tutorial Day 1 enthält Aufgaben zu den folgenden Themen

1. Python Basics (Aufgaben B1,...)
2. Python Listen, Strings (Aufgaben P1, P2, ...)
3. Aufgaben zur Motor-Steuerung & Sensor-Messung (nach der Pause)

Wiederholung

Python

- Skript-Sprache, Text-Datei wird ausgeführt
- Alternativ: Zellen in Jupyter-Notebook

Einfach Datentypen

- Zahlen als **`int`** und **`float`**
- Texte als **`str`**

Python

- Skript-Sprache, Text-Datei wird ausgeführt
- Alternativ: Zellen in Jupyter-Notebook

Einfach Datentypen

- Zahlen als **`int`** und **`float`**
- Texte als **`str`**

Mengen von Daten

- Listen mit **`list`** und **`[]`**
- Mengen mit **`set`**

Python Code-Blöcke

- Code-Blöcke durch Einrückung (Leerzeichen **oder** Tab)

```
if zahl > 50:
  faktor = 3
  sum = sum + zahl

preis = sum * faktor
```

Python Code-Blöcke

- Code-Blöcke durch Einrückung (Leerzeichen **oder** Tab)

```
if zahl > 50:
    faktor = 3
    sum = sum + zahl

preis = sum * faktor
```

Python Code-Blöcke

- Code-Blöcke durch Einrückung (Leerzeichen **oder** Tab)

```
if zahl > 50:
    faktor = 3
    sum = sum + zahl

preis = sum * faktor
```

Python Code-Blöcke

```
zahlen = [1,2,3,4,5,6]
gerade = []
ungerade = []

for zahl in zahlen:
  if zahl % 2 == 0:
    gerade.append(zahl)
  else:
    ungerade.append(zahl)

print("Die geraden Zahlen:")
print(gerade)
```

Python Code-Blöcke

```
zahlen = [1,2,3,4,5,6]
gerade = []
ungerade = []

for zahl in zahlen:
  if zahl % 2 == 0:
    gerade.append(zahl)
  else:
    ungerade.append(zahl)

print("Die geraden Zahlen:")
print(gerade)
```

Python Code-Blöcke

```
zahlen = [1,2,3,4,5,6]
gerade = []
ungerade = []

for zahl in zahlen:
  if zahl % 2 == 0:
    gerade.append(zahl)
  else:
    ungerade.append(zahl)

print("Die geraden Zahlen:")
print(gerade)
```

Python Code-Blöcke

```
zahlen = [1,2,3,4,5,6]
gerade = []
ungerade = []

for zahl in zahlen:
  if zahl % 2 == 0:
    gerade.append(zahl)
  else:
    ungerade.append(zahl)

print("Die geraden Zahlen:")
print(gerade)
```

Python Code-Blöcke

```
zahlen = [1,2,3,4,5,6]
gerade = []
ungerade = []

for zahl in zahlen:
  if zahl % 2 == 0:
    gerade.append(zahl)
  else:
    ungerade.append(zahl)

print("Die geraden Zahlen:")
print(gerade)
```

if-Bedingung

```
zahl = 32

if zahl > 50:
    text = "viel"
else:
    text = "wenig"

print("Ganz schoen " + text + ".")
```

if-Bedingung

```
zahl = 32
text = "wenig"

if zahl > 50:
    text = "viel"

print("Ganz schoen " + text + ".")
```

Variante ohne **else**

for-Schleife

```
zahlen = [0, 1, 1, 2, 3, 5, 8]
sum = 0
anzahl = 0

for zahl in zahlen:
  sum = sum + zahl
  anzahl = anzahl +1

schnitt = sum / anzahl
```

for-Schleife

```
zahlen = [0, 1, 1, 2, 3, 5, 8]
sum = 0
anzahl = 0

for zahl in zahlen:
  sum = sum + zahl
  anzahl = anzahl +1

schnitt = sum / anzahl
```

Schleifen-Block durch Einrückung!

while-Schleife

- Wiederholung mit “dynamischer Bedingung”
- Führt Anweisungen aus, solange die Bedingung gilt
- Bedingung ist ein bool’scher Ausdruck

```
while anzahl < 10:
  print("Anzahl ist jetzt: ", anzahl)
  anzahl = anzahl + 1
```

Wieder: Schleifen-Block durch Einrückung!

Eigene Funktionen (Beispiel)

Eigene Funktionen mit def

```
def durchschnitt(zahlen):
    sum = 0
    for zahl in zahlen:
        sum = sum + zahl

    return sum / len(zahlen)
```

Eigene Funktionen mit `def`

```
def durchschnitt(zahlen):
    sum = 0
    for zahl in zahlen:
        sum = sum + zahl

    return sum / len(zahlen)
```

- **`return`** legt Funktionsergebnis fest
- Bei **`return`** endet die Funktion sofort

Eigene Funktionen mit def

```
def durchschnitt(zahlen):
  sum = 0
  if len(zahlen) == 0:
    return 0

  for zahl in zahlen:
    sum = sum + zahl

  return sum / len(zahlen)
```

- **return** legt Funktionsergebnis fest
- Bei **return** endet die Funktion sofort

Zur Wiederholung: Listen

Listen sind Folgen von Objekten (Zahlen, Tupeln, Strings,...)
Eine einfache Liste gemischter Objekte:

```
# Gemischte Liste:
#
gemischt = [ 1, 94, "Wort", 3.14159, ('a', 123)]
```

Zugriff auf Elemente einer Liste über Index:

```
pi = gemischt[3]
```

Zur Wiederholung: Listen

Listen sind Folgen von Objekten (Zahlen, Tupeln, Strings,...)
Eine einfache Liste gemischter Objekte:

```
# Gemischte Liste:
#
gemischt = [ 1, 94, "Wort", 3.14159, ('a', 123)]
```

Zugriff auf Elemente einer Liste über Index:

```
pi = gemischt[3]
```

Zugriff vom Ende der Liste aus:

```
letztes = gemischt[-1]  # ('a',123)
wort = gemischt[-3]
```

Teile von Listen: Slicing

Slicing `von:bis`, aber `bis` gehört nicht mehr dazu!

```
gemischt = [ 1, 94, "Wort", 3.14159, ('a', 123)]

anfang = gemischt[:3]   # => [1, 94]
ende = gemischt[-2:]    # [ 3.14159, ('a', 123)]

alles = gemischt[:]

wortListe = gemischt[2:3]   # => ["Wort"]
```

Teile von Listen: Slicing

Slicing `von:bis`, aber `bis` gehört nicht mehr dazu!

```
gemischt = [ 1, 94, "Wort", 3.14159, ('a', 123)]

anfang = gemischt[:3]   # => [1, 94]
ende = gemischt[-2:]    # [ 3.14159, ('a', 123)]

alles = gemischt[:]

wortListe = gemischt[2:3]   # => ["Wort"]
```

Das Ergebnis ist wieder eine Liste!!

Listen können mit **append** verlängert werden:

```
# leere Liste
liste = []

liste.append(1)  # 1 anhaengen
liste.append(94) # 94 hinzufuegen
```

Listen verarbeiten

Mit **for** Schleifen können Listen leicht verarbeitet werden:

```
# Liste von Zahlen
liste1 = [ 2, 5, 8, 3, 9, 4 ]

# leere Liste
liste2 = []

for zahl in liste1:
  if zahl % 2 == 0:
    liste2.append(zahl)

# liste2 enthaelt die geraden Zahlen aus liste1
```

Künstliche Liste (Sequenz) mit `range`

Mit **`range`** lassen sich Folgen natürlicher Zahlen erzeugen:

```
quadrate = []

for x in range(1, 100):
  quadrate.append( x*x )
```

Künstliche Liste (Sequenz) mit **range**

Mit **range** lassen sich Folgen natürlicher Zahlen erzeugen:

```
quadrate = []

for x in range(1, 100):
  quadrate.append( x*x )
```

Kurzform mit *list comprehension*

```
quadrate = [ x*x for x in range(1, 100) ]
```

Funktionen

Eigene Funktionen mit **def** definieren:

```
def brutto(preis):
  return preis * 1.19
```

Funktionen

Eigene Funktionen mit **def** definieren:

```
def brutto(preis):
  return preis * 1.19
```

```
nettoPreise = [ 10, 23, 28, 120 ]

bruttos = []
for preis in nettoPreise:
    bruttos.append( brutto(preis) )

# alternativ:
bruttos = [ brutto(x) for x in nettoPreise ]
```

Strings - Folgen von Buchstaben

Wörter, Texte usw. sind Folgen von Buchstaben - eigentlich wie unveränderbare Listen:

```
wort = "DataScience"
a = wort[1]
science = wort[4:]
```

Strings - Folgen von Buchstaben

Wörter, Texte usw. sind Folgen von Buchstaben - eigentlich wie unveränderbare Listen:

```
wort = "DataScience"
a = wort[1]
science = wort[4:]
```

```
# Zaehlen der 'a'
zaehler = 0
for buchstabe in wort:
  if buchstabe == 'a':
    zaehler = zaehler + 1
```

Notebooks für Übungen

- Notebook Server enthält `TutorialDay` Verzeichnis
- Drei Notebooks: Python, Series und DataFrame
- Für viele Aufgaben `tutorial` Modul mit Selbsttest

Demo Notebook-Server

Python Basics

Bei den ersten Aufgaben (B1 bis B4) geht es um Basics von Python, also Schleifen (**for**), Tupel und Bedingungen.

Aufgabe B1

- Definieren Sie eine Liste der Zahlen von 1 bis 10 in Python.
- Berechnen Sie daraus die Liste der Quadrate der Zahlen.
- Schreiben Sie eine Funktion `quadrate(zahlen)`, die eine Liste von Zahlen als Eingabe erhält und die Liste der Quadrate zurückgibt.
- Berechnen Sie die Liste der Quadrate von 1 bis 1000.

Aufgabe B2

- Die Funktion `tutorial.wortliste()` liefert eine Liste von Worten zurück. Schreiben Sie eine Schleife, die die Anzahl der Worte berechnet, die mit 'S' beginnen.
- Schreiben Sie eine eigene Funktion `worteMitS(xs)`, die eine Liste `xs` von Worten bekommt und die Liste der Worte, die mit dem Buchstaben `'S'` beginnen zurückgibt.

Python Listen, Strings

Die P-Aufgaben drehen sich um Python und Listen.

Die Funktion `tutorial.wortliste()` aus dem Tutorial-Modul liefert eine Liste von Wörtern:

```
words = tutorial.wortliste()
```

Diese Liste können Sie für die Aufgaben gerne als Test-Daten nutzen.

Aufgabe P1:

1. Schreiben Sie eine Funktion `last_word(ws)`, die eine Liste von Wörtern als Parameter bekommt und das letzte Wort aus der Liste zurückiefert.
2. Testen Sie Ihre Funktion in dem Sie der Variablen `letztesWort` das letzte Wort aus der Wortliste zuweisen.

Rufen Sie anschließend `tutorial.AufgabeP1()` auf, um Ihre Funktion zu testen!

Aufgabe P2:

1. Schreiben Sie eine Funktion `words_with(xs, w)`, die aus einer Liste von Wörtern die Wörter heraussucht, die die Zeichenfolge `w` enthalten.

Beispiel:

```
words = ["abc", "bcd", "cde"]

words_with(words, "cd")
# sollte  ['bcd', 'cde']  zurueckgeben
```

Rufen Sie anschließend `tutorial.AufgabeP2()` auf, um Ihre Funktion zu testen!

Aufgabe P3:

1. Schreiben Sie eine Funktion `max_words`, die aus einer Liste von Wörtern die längsten Wörter heraussucht.
2. Definieren Sie auch eine Funktion `min_words` für die kürzesten Wörter einer Liste.

Hinweis:
Falls es ein eindeutiges längstes/kürzestes Wort in der Liste gibt, soll Ihre Funktion eine 1-elementige Liste mit diesem Wort zurückgeben.

Aufgabe P4:

Mit `range(n)` kann man in Python Folgen erzeugen

1. Benutzen Sie `range` und `list` um eine Liste `n10` der Zahlen von 1 bis 10 zu erzeugen.
2. Schreiben Sie eine Funktion `alle_vorhanden`, die überprüft, ob eine Liste der Länge *n* alle Zahlen von 1 bis *n* enthält.

Beispiel:

```
xs = [1,2,3,4,5]
alle_vorhanden(xs, 5)  # ergibt True

xs2 = [1,1,2,4,2]
alle_vorhanden(xs2, 5) # ergibt False
```