

DATA SCIENCE 1

TUTORIAL DAY

PROF. DR. CHRISTIAN BOCKERMANN

HOCHSCHULE BOCHUM

SOMMERSEMESTER 2023

Der Tutorial Day gibt Aufgaben zu den folgenden Themen

1. Python Listen, Strings (Aufgaben P1, P2, ...)
2. Pandas Series (Aufgaben S1, S2, ...)
3. Pandas DataFrame (Aufgaben D1, D2, ...)

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

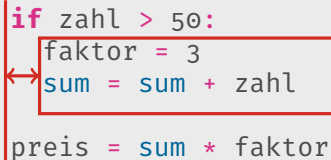
- Code-Blöcke durch Einrückung (Leerzeichen **oder** Tab)

```
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:
        ungerae.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:
        ungerae.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:
        ungerae.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:
        ungerae.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:
        ungerae.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!

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 Lists, 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 herausucht.
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 $n=10$ 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
```

Wiederholung: Pandas

Pandas: DataFrame

Ein DataFrame `df` ist eine Tabellenstruktur:

	a1	a2	a3
0	4	1	2
1	5	1	3
2	3	8	7

Pandas: DataFrame

Ein DataFrame `df` ist eine Tabellenstruktur:

	a1	a2	a3
0	4	1	2
1	5	1	3
2	3	8	7

Spalten-Index
`df.columns`

Zeilen-Index
`df.index`

Pandas: DataFrame

Ein DataFrame `df` ist eine Tabellenstruktur:

“Positionsindex”

	0	1	2
0	4	1	2
1	5	1	3
2	3	8	7

Spalten-Index
`df.columns`

Zeilen-Index
`df.index`

Ein DataFrame `df` mit anderen Indizes

```
df.index = ['A', 'B', 'C']  
df.columns = ['x1', 'x2', 'x3']
```

“Positionsindex”

		0	1	2
		x1	x2	x3
0	A	4	1	2
1	B	5	1	3
2	C	3	8	7

Spalten-Index
`df.columns`

Zeilen-Index
`df.index`

Einzelne Zeilen/Spalten sind **Series** Objekte

Zugriff auf **df** über verschiedene Elemente:

```
df[0:2]      # Zeilen mit Slicing
df[['a1', 'a2']] # Spalten durch Namensliste

# Zugriff mit Positionsindex:
df.iloc[zeilen, spalten]

# Zugriff mit Zeilen/Spalten-Index:
df.loc[zeilen, spalten]
```

Mit `df.iloc[...]` wird nach **Position** selektiert

```
df.iloc[ ZEILEN, SPALTEN ]
```

ZEILEN bzw. **SPALTEN** sind Zahlen, Listen von Zahlen, Slices

```
# Zelle in erster Zeile, dritter Spalte:
```

```
df.iloc[0,2]
```

```
# Die erste Zeile (als Series Objekt!)
```

```
df.iloc[0,:]
```

```
# Die erste Spalte:
```

```
df.iloc[:,0]
```

Selektieren mit `.iloc[...]`

Beim Slicing `a:b` gehört `a` dazu, `b` nicht mehr:

```
# die Zeilen 0 und 1:  
df.iloc[0:2,:]  
  
# die Spalten 0 und 1:  
df.iloc[:,0:2]
```

Was passiert bei `.loc[...]`?

```
df.loc['A':'B']
```

		0	1	2
		x1	x2	x3
0	A	4	1	2
1	B	5	1	3
2	C	3	8	7

Was passiert bei `.loc[...]`?

```
df.loc['A':'B']
```

		0	1	2
		x1	x2	x3
'A' → 0	A	4	1	2
1	B	5	1	3
2	C	3	8	7

Was passiert bei `.loc[...]`?

```
df.loc['A':'B']
```

		0	1	2
		x1	x2	x3
'A'	→ 0	4	1	2
'B'	→ 1	5	1	3
	2	3	8	7

Was passiert bei `.loc[...]`?

```
df.loc['A':'B']
```

		0	1	2
		x1	x2	x3
'A'	→ 0	4	1	2
'B'	→ 1	5	1	3
	2	3	8	7

Es werden **beide** Zeilen (A und B) selektiert!

Pandas Series

Die folgenden S-Aufgaben, beziehen sich auf Pandas **Series**.

Es geht um

- das Erstellen von Series Objekten,
- den Zugriff auf Elemente von Series Objekten,
- die Berechnung einfacher Werten.

Aufgabe S1

Betrachten Sie die Folgen

$4, 8, 7$ und $1, -3, -2, 5$

1. Definieren Sie eine Variable `series1` mit der ersten Folge $4, 8, 7$.
2. Definieren Sie eine Variable `series2` mit der zweiten Folge.

Aufgabe S2 - Zugriff auf Elemente

Sie haben in Aufgabe S1 die Series `series1` definiert

1. Definieren Sie eine Variable `a`, die das zweite Element der Series enthält.
2. Definieren Sie eine Variable `b`, die das letzte Element der Series enthält
(Erinnern Sie sich an *Slicing* und negative Indizes?).

Aufgabe S3 - Operationen mit Series

Sie haben in Aufgabe S1 die Series `series1` und `series2` definiert.

1. Addieren Sie die beiden Series Elemente und weisen Sie das Ergebnis der Variablen `series3` zu.

Benutzen Sie danach den Aufruf

```
print(series3)
```

um sich das Ergebnis anzuschauen.

Aufgabe S4 - Filtern

Über den Zugriffsoperator `s[...]` lassen sich Series Werte filtern:

1. Schreiben Sie eine Funktion `groessero`, die eine Series übergeben bekommt und eine Series mit den Werten zurückliefert, die größer oder gleich 0 sind.

Aufgabe S5 - Funktionen auf **Series** Objekten

1. Schreiben Sie eine Funktion **smax**, die den maximalen Wert einer Series zurückgibt.
2. Schreiben Sie eine Funktion **smin**, die den minimalen Wert zurückgibt.

Aufgabe S6 - Funktionen auf **Series** Objekten

Die Min-Max Normalisierung ist dadurch definiert, dass jeder Wert v durch

$$v' = \frac{v - \min}{\max - \min}$$

ersetzt wird.

1. Schreiben Sie eine Funktion `snorm(series)`, die aus einer Series eine normalisierte Series nach der Min-Max Normalisierung berechnet.

Pandas DataFrame

Die folgenden Aufgaben beziehen sich auf **DataFrame**

1. Erstellen von DataFrames
2. Selektieren von Zeilen/Spalten
3. Eigene Funktionen auf DataFrame Objekten

Aufgabe D1

Erstellen Sie den folgenden **DataFrame** und weisen Sie ihn der Variablen **df1** zu:

A	B
1	2
3	4

Aufgabe D2

Erstellen Sie ein `Series` Objekt mit den Werten 5 und 6 und fügen Sie es als neue Spalte "C" in den DataFrame `df1` ein. Der DataFrame sollte dann folgendermaßen aussehen:

A	B	C
1	2	5
3	4	6

Aufgabe D3

Erstellen Sie im `DataFrame df1` die Spalte "Z", die die Summe der Spalten "A", "B" und "C" enthält:

A	B	C	Z
1	2	5	8
3	4	6	13

Aufgabe D4

Teilen Sie im DataFrame **df1** alle Spalten ausser der Spalte "Z" durch die Werte der "Z"-Spalte.

Das Ergebnis sollte folgendermaßen aussehen:

A	B	C	Z
0.125	0.250	0.625	8
0.231	0.308	0.462	13

Selektieren von Zellen/Bereichen

Die Funktion `tutorial.toy_data()` liefert das folgende `DataFrame` Objekt zurück:

a1	a2	a3	a4
4	1	2	9
5	1	3	6
3	8	7	4

1. Definieren Sie die Variable `df_rot`, die den **rot** markierten Teil des DataFrames enthält.
2. Definieren Sie die Variable `df_blaue`, für den **blau** markierten Teil.

Selektieren von Zellen/Bereichen

1. Definieren Sie einen quadratischen DataFrame `qf`, indem Sie aus dem `toy_data()` DataFrame die ersten $n = 3$ Zeilen/Spalten selektieren.
2. Schreiben Sie eine Funktion `quadrat(df)`, die aus einem DataFrame den größten quadratischen Block extrahiert (von oben links beginnend).
3. Schreiben Sie eine Funktion `diag(df)`, die für einen DataFrame die Liste der Elemente auf der Hauptdiagonalen zurückliefert.

Mit `.apply(f, axis=1)` wird eine Funktion auf alle Zeilen angewandt:

```
# f gibt fuer jede Zeile 1 zurueck:  
def f(row):  
    # row ist ein Series Objekt!  
    return 1  
  
# f auf alle Zeilen anwenden  
series = df.apply(f, axis=1)
```

Betrachten Sie die folgende Tabelle:

Länge (cm)	Breite (cm)	Höhe (cm)	Entfernung (km)	Dienstleister
100	100	40	20	WPS
110	100	100	60	UPS
100	210	100	70	WPS
50	100	70	89	DHL

(Aufgabenblatt 4 aus Wirtschaftsinformatik 1)

Der Datensatz (ohne die Dienstleister-Spalte) ist als CSV-Datei im data Verzeichnis enthalten.

Aufgabe D7*

1. Laden Sie den Versand-Datensatz in einen DataFrame.
2. Schreiben Sie eine Funktion, die den Dienstleister pro Zeile berechnet.
3. Berechnen Sie die Spalte Dienstleister mit ihrer Funktion.

Regeln für die Dienstleister-Funktion:

- Paketvolumen $< 1m^3$, dann Dienstleister DHL
- $1m^3 \leq$ Paketvolumen $< 2m^3$, dann Dienstleister UPS
- Paketvolumen $\geq 2m^3$, dann Dienstleister WPS
- Wenn Entfernung $< 50km$, dann auf jeden Fall WPS

Selektieren von Zellen/Bereichen

Die Funktion `tutorial.sudoku()` liefert einen 4x4 DataFrame:

0	1	2	3
2	1	3	4
4	3	1	2
1	2	4	3
3	4	2	1

Beim Spiel **Sudoku**, müssen in jeder Zeile, jeder Spalte und jedem Block die Ziffern 1 bis 4 **genau einmal** vorkommen.

Aufgabe D8*

1. Schreiben Sie eine Funktion `sudoku_zeile`, die für eine Zeile eines 4x4 DataFrame `True` liefert, wenn es eine gültige Sudoku Zeile ist.
2. Schreiben Sie eine Funktion `sudoku_spalte`, die für eine Spalte eines 4x4 DataFrame `True` zurückgibt, wenn es eine gültige Sudoku Spalte ist.
3. Schreiben Sie eine Funktion `sudoku_block`, die für einen 2x2 Block eines DataFrame `True` zurückgibt, wenn es ein gültiger Sudoku Block ist.