

DATA SCIENCE 1

VORLESUNG 2 - EINFÜHRUNG IN PYTHON

PROF. DR. CHRISTIAN BOCKERMANN

HOCHSCHULE BOCHUM

WINTERSEMESTER 2024/2025

- 1 Einführung in Python
 - Einstieg in Python
 - Datentypen und Operatoren
 - Listen, Tuple und Sequenzen
 - Mengen und Hash-Tabellen
 - Zeichenketten: str
 - Kontrollstrukturen: if, for, while
- 2 Funktionen und Module in Python
 - Eigene Funktionen
 - Python Module
- 3 Spaß mit Listen/Sequenzen

Einführung in Python

Literatur zu Python + Programmierung



Buch ist in der Bibliothek (auch online) verfügbar [Inden(2021)].
(VPN Verbindung erforderlich!)

Literatur



MICHAEL INDEN.

EINFACH PYTHON.

dpunkt.verlag, 2021.

ISBN 9783969106464.

Relevante Kapitel in *Einfach Python*:

- Kapitel 2: Grundlagen, Variablen, Schleifen
- Kapitel 3: Zeichenketten/Strings
- Kapitel 5 bis 5.2.2: Listen und Mengen (5.2.1 optional)

Python ist eine interpretierte Skript-Sprache

- Programme in Text-Dateien (Skripte) mit Endung `.py`
- Skripte werden zeilenweise abgearbeitet

Beispiel: Datei `HelloWorld.py`

```
print("Hallo, Welt!")
```

Python ist eine interpretierte Skript-Sprache

- Programme in Text-Dateien (Skripte) mit Endung `.py`
- Skripte werden zeilenweise abgearbeitet

Beispiel: Datei `HelloWorld.py`

```
print("Hallo, Welt!")
```

Starten durch Aufruf des Interpreters mit dem Skript:

```
# python3 HelloWorld.py
```

Python Skripte sind Folgen von Ausdrücken

- Variablenzuweisung, Funktionsaufrufe
- Funktions- oder Klassendefinition
- Kontrollstrukturen (**if**, **for**,...)

Beispiel:

```
a = 42
b = 21
c = a + b
d = someFunc(c)    # Funktionsaufruf
xs = [0, 1, 2, 3]  # eine Liste
```


Python erlaubt **Kommentare** im Code

```
# Pythagoras aus der Schule  
a = 3.0  
b = 4.0  
c2 = a*a + b*b      # c2 = c zum Quadrat  
# wie berechnen wir die Wurzel aus c2?
```

Python erlaubt **Kommentare** im Code

```
# Pythagoras aus der Schule  
a = 3.0  
b = 4.0  
c2 = a*a + b*b      # c2 = c zum Quadrat  
# wie berechnen wir die Wurzel aus c2?
```

- Kommentare starten mit Raute (**#**)
- Kann an beliebiger Stelle starten
- Kommentare sind **wichtig um Code zu verstehen**

Python ist eine **dynamisch typisierte** Sprache, d.h.

- Variablen haben einen Typ
- der Typ einer Variablen wird von Python zur Laufzeit ermittelt
- Typ wird *nicht* explizit vom Benutzer festgelegt

Beispiel:

```
a = 42           # Variable a ist ein int
b = 42.0        # b ist vom Typ float
c = a * b       # Welchen Typ hat c?
xs = [a, b]     # xs ist vom Type list
```

Die Funktion `type(x)` gibt den Typ von `x` zurück.



◀ Probieren Sie es im Notebook aus!

Text	str
Numerische Werte	int, float, complex
Sequenzen	list, tuple, range
Maps	dict
Mengen	set, frozenset
Bool'sche Werte	bool
Binäre Daten	bytes, bytearray, memoryview

Abbildung: Datentypen in Python (Auszug)

Python bietet Operatoren für Grundrechenarten:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division (Gleitkomma)
%	Modulo Operator (Rest)
//	Division (ganzzahlig)
**	Potenzieren

Spaß mit Variablen:

```
a = 3.0  
b = 4.0  
c2 = a**2 + b**2  
  
c = c2 ** 0.5   # Wurzelziehen => hoch 0.5
```

Weitere Beispiele:

```
m = 8 % 3  
x = 8 / 3  
y = 8 // 3
```

Welche Werte ergeben sich für **m**, **x** und **y**?



◀ Probieren Sie es im Notebook aus!

Relationale Operatoren für den Vergleich von Werten:

>	größer als
<	kleiner als
==	gleich
!=	ungleich
>=	größer oder gleich
<=	kleiner oder gleich

Vergleichsoperatoren liefern einen Wahrheitswert (**bool**)

```
erg = 1 < 3  
print(erg)
```

Vergleich von Variablen:

```
a = 3.0  
b = 4.0  
x = a >= b  
print(x)
```



◀ Probieren Sie es im Notebook aus!

Vergleichsoperatoren können mit **and**, **or** und **not** benutzt werden:

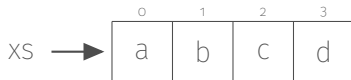
```
a = 3.0  
print( a > 2.0 and a < 5.0 )
```

Der Datentyp `list`

Listen sind unverselle Datenstruktur:

```
xs = ['a', 'b', 'c', 'd'] # Definition einer Liste
```

Was genau passiert dabei?

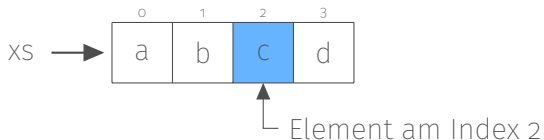


Der Datentyp `list`

Listen sind unverselle Datenstruktur:

```
xs = ['a', 'b', 'c', 'd'] # Definition einer Liste
```

Was genau passiert dabei?



```
y = xs[2] # y enthaelt jetzt den Buchstaben 'c'
```

Eine Liste ist eine universelle Datenstruktur:

```
xs = [1,2,3,4]      # Definition einer Liste
ys = [4,5,6,7]      # noch eine Liste

# Einige Operatoren funktionieren auch fuer Listen:
zs = xs + ys
ws = xs * 2
```

Eine Liste ist eine universelle Datenstruktur:

```
xs = [1,2,3,4]      # Definition einer Liste
ys = [4,5,6,7]      # noch eine Liste

# Einige Operatoren funktionieren auch fuer Listen:
zs = xs + ys
ws = xs * 2
```

Was ergibt die folgende Anweisung?

```
zs = xs - ys
```



Probieren Sie es im Notebook aus!

Wie wählen wir Teile aus einer Liste aus?

Zugriff auf einzelne Elemente:

```
xs = [1, 2, 3, 4, 5, 6]  
a = xs[3] # Welchen Wert hat a ?
```

Zugriff auf Teil-Listen (*slicing*):

```
xs = [1, 2, 3, 4, 5, 6]  
xxs = xs[0:4] # Ein Teilbereich von xs
```



◀ Probieren Sie es im Notebook aus!

Die Funktion **len** berechnet die Länge einer Liste:

```
xs = ['a', 'b', 'c', 'd']  
laenge = len(xs)
```

Mit dem Operator **in** kann getestet werden, ob eine Liste ein bestimmtes Element enthält:

```
xs = [ 1, 2, 3, 4, 5 ]  
y = 3  
enthalten = y in xs  
  
# enthalten ist nun 'True'
```

Für Listen existieren noch weitere Funktionen, z.B.:

```
xs.append('e')      # Anhaengen eines weiteren Elements
xs.index('c')      # Berechnet den Index von 'c'
xs.remove('b')     # Entfernt ein Element
xs.sort()          # Sortiert die Liste
xs.reverse()       # Dreht die Reihenfolge um
```


Für Listen existieren noch weitere Funktionen, z.B.:

```
xs.append('e')      # Anhaengen eines weiteren Elements
xs.index('c')      # Berechnet den Index von 'c'
xs.remove('b')     # Entfernt ein Element
xs.sort()          # Sortiert die Liste
xs.reverse()       # Dreht die Reihenfolge um
```



Definieren Sie eine Liste **xs** im Notebook und geben Sie in einer Zelle **xs.<TAB>** ein!

Der Datentyp `tuple`

- Tupel fassen mehrere Elemente zusammen
- Tupel sind `immutable` - können nicht verändert werden
- Tupel haben daher feste Länge

Tupel können beliebige Objekte enthalten:

```
paar = (1, 'A')           # ein 2er Tupel  
dreier = (2, 'B', [1,2]) # 3er Tupel
```

Der Datentyp `tuple`

Tupel verhalten sich wie *nur lesbare* Listen

```
paar = (1, 'A')    # Definition des Tupels  
x = paar[0]       # x = 1  
y = paar[1]       # y = 'A'
```



◀ Probieren Sie es im Notebook aus!

Python stellt den Typ `range` für Folgen ganzer Zahlen bereit:

```
seq1 = range(4)      # erzeugt die Folge 0, 1, 2, 3
seq2 = range(1,4)   # erzeugt die Folge 1, 2, 3
seq3 = range(1,9,3) # erzeugt die Folge 1, 4, 7
```

Aber:

```
seq = range(1,4)
print(seq)
```

erzeugt als Ausgabe:

```
range(1,4)
```

Was ist der Vorteil von `range` gegenüber einer Liste?

- `range` Objekt speichert die Werte **start**, **stop** und **step**
- Alle Informationen können daraus berechnet werden

Was ist der Vorteil von `range` gegenüber einer Liste?

- `range` Objekt speichert die Werte **start**, **stop** und **step**
- Alle Informationen können daraus berechnet werden

Betrachten wir die folgenden Zeilen:

```
seq = range(1, 10000) # start=1, stop=5, step=1  
xs  = list(seq)      # Liste mit 10000 Elementen
```

Python's **dict** Typ ist zentraler Datentyp

- **dict** Objekte ordnen Objekte andere Objekte zu
- Entsprechen einer Menge von (**key,value**) Paaren
- Sie funktionieren wie Nachschlagewerke
- Listen erlauben Zugriff per Index - **dict** einen Zugriff über Objekte

```
m = { "firstname": "John", "lastname": "Doe" }  
name = m["firstname"]  
  
# aequivalent:  
name = m.get("firstname")
```

Betrachten wir das folgende `dict` Object:

```
m = { "name": "John", "lastname": "Doe", "age": 42 }
```

Was passiert dabei anschaulich?

	Key	Value
m →	age	42
	name	John
	lastname	Doe

Wie lassen sich **dict** Objekte erzeugen?

```
leer = dict()    # leere Tabelle

tuples = [(1, 'a'), (2, 'b')] # Liste von Paaren
fromTuples = dict(tuples)    # Tabelle mit 1='a', usw

# ueber Parameter fuer dict(..)
direkt = dict( name="John", age=42 )

# durch Zuweisung von Werten zu Schluesseln (keys)
person = dict()
person["name"] = "Doe"
person["firstname"] = "John"
person["age"] = 42
```

Python enthält den Datentyp **str** für Zeichenketten:

```
a = "42"  
b = 42  
  
print(a == b)
```

Python **str** Objekte sind wie Listen:

```
name = "Hello, world!"  
ello = name[1:4]  
  
if "world" in name:  
    print("Die Welt ist noch da!")
```

Der **str** Typ stellt eine Menge nützlicher Funktionen bereit:

```
name = "alice"  
name.capitalize()           # ergibt: "Alice"  
name.islower()              # ergibt: True  
name.replace('ice', 'ex')   # ergibt: "alex"
```

Der **str** Typ stellt eine Menge nützlicher Funktionen bereit:

```
name = "alice"  
name.capitalize()           # ergibt: "Alice"  
name.islower()              # ergibt: True  
name.replace('ice', 'ex')   # ergibt: "alex"
```

Wichtig:

- **str** Objekte sind **immutable** (unveränderbar)
- Funktionen wie **replace(..)** liefern einen neuen String zurück!

Python stellt **Kontrollstrukturen** für Programmablauf bereit

- Bedingte Anweisungen mit **if**, **else** oder **switch**
- Schleifen mit **for** oder **while**
- Bedingte Anweisungen in Block (Einrückung!)

Beispiel:

```
x = 30
if x > 10:
    print("x ist mehr als 10!")
else:
    print("x ist weniger als 10!")
```

Python verwendet **Einrückungen** um Code zu strukturieren

- Aufeinanderfolgende Zeilen mit Tiefe bilden Block
- Im Pythonumfeld werden die Blöcke **Suites** genannt
- Dadurch weniger Klammern

```
if x > 4:  
    print("Mehr als 4")  
    x = x + 4  
else:  
    print("Weniger als 5")
```

Python verwendet **Einrückungen** um Code zu strukturieren

- Aufeinanderfolgende Zeilen mit Tiefe bilden Block
- Im Pythonumfeld werden die Blöcke **Suites** genannt
- Dadurch weniger Klammern

```
if x > 4:  
    print("Mehr als 4")  
    x = x + 4  
else:  
    print("Weniger als 5")
```

Zusammenhängender
Block

Schleifen führen Code-Blöcke mehrfach aus

Beispiel:

```
i = 0
xs = list()
while i < 10:
    xs.append(i)
    i = i + 1
```

```
xs = list()
for i in range(10):
    xs.append(i)
```


for Schleife für *iterierbare* Elemente (Listen, Sequenzen)

Beispiel:

```
xs = [1, 2, 3, 4]
for x in xs:
    print(x)
```

Kontrollstrukturen ermöglichen in Python **List Comprehension**

- Mathematisch können Mengen kompakt definiert werden, z.B. eine Menge von Quadratzahlen

$$\{ x^2 \mid x \in \{1, 2, \dots, n\} \}$$

Die dazu passende Python Definition ist:

```
n = 10  
quadrate = [ x*x for x in range(1,n+1) ]
```

List Comprehension lässt sich mit Bedingungen kombinieren

Zum Beispiel um nur die *gerade* Zahlen auszuwählen:

```
zahlen = [1,2,3,4,5]  
geradeZahlen = [x for x in zahlen if x % 2 == 0]
```


Funktionen in Python werden mit **def** definiert

- Funktionen bestehen aus einem Namen, den Parametern (optional) und einem Code Block als Rumpf
- Dem Rumpf kann eine Beschreibung (**docstring**) vorangestellt werden

Funktionen in Python werden mit **def** definiert

- Funktionen bestehen aus einem Namen, den Parametern (optional) und einem Code Block als Rumpf
- Dem Rumpf kann eine Beschreibung (**docstring**) vorangestellt werden

Beispiel: $f(x) = x^3 + 5x^2 + 27$

```
def f(x):  
    """  
    Die ist ein Beschreibung der Funktion  
    """  
    return x ** 3 + 5 * x ** 2 + 27
```

Ein weiteres Beispiel - Betragsfunktion

```
def betrag(x):  
    """  
    Liefert den Betrag von x zurueck  
    """  
    if x >= 0:  
        return x  
    else:  
        return -x
```

Funktionsparameter können Standardwerte haben

```
def greet(name = 'Welt'):  
    """  
    Sagt Hallo zum angegebenen Namen bzw. zur Welt,  
    wenn kein Name angegeben wurde.  
    """  
    print( "Hallo, " + name + "!")  
  
# Aufruf der Funktion ohne und mit Parameter  
greet()  
greet("Data Science")
```



Probieren Sie es im Notebook aus!

Hilfsfunktionen mit `lambda`

- Oft kleine Funktionen für Eigenschaften (`True/False`)
- Kleine Operationen, nur an einer Stelle benutzt

`lambda` input : expression

Hilfsfunktionen mit `lambda`

- Oft kleine Funktionen für Eigenschaften (`True/False`)
- Kleine Operationen, nur an einer Stelle benutzt

`lambda` input : expression
Parameter Anweisung

Hilfsfunktionen mit **lambda**

- Oft kleine Funktionen für Eigenschaften (**True/False**)
- Kleine Operationen, nur an einer Stelle benutzt

lambda input : expression
Parameter Anweisung

Beispiele:

```
# lambda-Ausdruck, der gerade/ungerade testet:
```

```
lambda x : x % 2 == 0
```

```
# lambda Ausdruck, der testet ob es eine 4 ist:
```

```
lambda x : x == 4
```

Beispiel: Filtern mit *richtiger Funktion*

```
def istEineVier(x):  
    return x == 4
```

```
xs = [4, 1, 2, 1, 4, 7, 3, 8, 4, 5, 4, 4]  
vieren = filter( istEineVier, xs )
```

Elegant mit `lambda` Ausdruck:

```
xs = [4, 1, 2, 1, 4, 7, 3, 8, 4, 5, 4, 4]  
vieren = filter( lambda x : x == 4, xs )
```

Ein **Modul** stellen Klassen und Funktionen bereit

- Python enthält große **Standardbibliothek** mit vielen Modulen
- Module im Skript mit **import** einbinden

Beispiel: `time` enthält Funktionen für Umgang mit Zeit

```
import time

jetzt = time.time()
print(jetzt)
```

Ein **Modul** stellen Klassen und Funktionen bereit

- Python enthält große **Standardbibliothek** mit vielen Modulen
- Module im Skript mit **import** einbinden

Beispiel: `time` enthält Funktionen für Umgang mit Zeit

```
import time

jetzt = time.time()
print(jetzt)
```

**Vieles braucht man nicht selbst zu programmieren!
Es geht mehr um die Verbindung der richtigen Teile ; -)**

Ein Modul kann mit **import** eingebunden werden

```
import datascience  
  
# Erstelle ein neues Objekt der Klasse MyModel  
m = datascience.MyModel()
```

Mit **import .. as** läßt sich der Namesraum eines Moduls ändern:

```
import datascience as ds  
  
m = ds.MyModel()
```

^oDas Modul **datascience** aus dem Beispiel ist ein fiktives Modul.

Manchmal ist es hilfreich, Funktionen/Klassen in den globalen Namensraum zu importieren:

```
from datascience import MyModel  
  
m = MyModel()
```

Dann kann auf **MyModel** ohne den Modulnamen zugegriffen werden

Viel Theorie - wofür eigentlich?

- Bisher Auflistung von Datentypen und Beispielen
- Für sich betrachtet: recht langweilig

Im Folgenden Abschnitt:

- Wichtige Standard-Funktionen für allerlei Aufgaben
- Filtern, Zusammenbauen, usw.

Viel Theorie - wofür eigentlich?

- Bisher Auflistung von Datentypen und Beispielen
- Für sich betrachtet: recht langweilig

Im Folgenden Abschnitt:

- Wichtige Standard-Funktionen für allerlei Aufgaben
- Filtern, Zusammenbauen, usw.

Sammelsurium nützlicher *day-to-day* Funktionen!



Probieren Sie es im Notebook aus!

Filtern von Sequenzen

Die Funktion `filter` selektiert Elemente aus Sequenzen

- Haben wir bereits kennengelernt (Folie zu `lambda`)
- Sehr nützlich im Zusammenhang mit lambda-Funktionen

Beispiel

```
jobs = ["data Scientist", "data Engineer",  
        "data Analyst", "developer"]  
  
dataJobs = filter( lambda job : "Data" in job, jobs )
```

Verbinden von Listen/Sequenzen

Die Funktion `zip` verbindet Listen/Sequenzen

`xs`

x_1	x_2	x_3	x_4
-------	-------	-------	-------

`zip(xs, ys)`

(x_1, y_1)	(x_2, y_2)	(x_3, y_3)	(x_4, y_4)
--------------	--------------	--------------	--------------

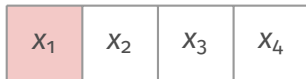
`ys`

y_1	y_2	y_3	y_4
-------	-------	-------	-------

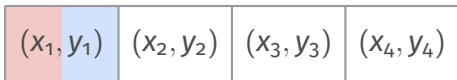
Verbinden von Listen/Sequenzen

Die Funktion `zip` verbindet Listen/Sequenzen

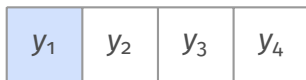
`xs`



`zip(xs, ys)`



`ys`



Zipper / "Reissverschlussfunktion"

Verbinden von Listen/Sequenzen

Die Funktion `zip` verbindet Listen/Sequenzen

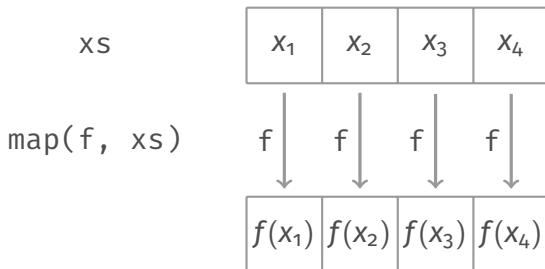
```
xs = [1, 2, 3]
ys = ['a', 'b', 'c']

zs = zip(xs, ys)
# zs ist nun ein zip Objekt (Sequenz)

ls = list(zs) # erzeugt eine Liste von Tupeln
# ls ist nun: [(1, 'a'), (2, 'b'), (3, 'c')]
```

Funktionen auf Listen/Sequenzen anwenden

Die Funktion `map` wendet eine Funktion auf Listen/Sequenzen an



Funktionen auf Listen/Sequenzen anwenden

Die Funktion `map` wendet eine Funktion auf Listen/Sequenzen an

```
xs = [1, 2, 3]

ys = map( lambda x : x + 1, xs )
# ys ist nun ein map Objekt (Sequenz)

ls = list(ys)
# ys ist nun: [2, 3, 4]
```