

# Geschichte

## Python: Die Geschichte

- Ab 1989 von dem niederländischen Programmierer Guido van Rossum entwickelt
- Python 1.0 (Erste Vollversion): Januar 1994
- Python 2.0: Oktober 2000
- Python 3.0: Dezember 2008
- Python 2.7.18 (Letzte 2.x Version, keine weitere Unterstützung): April 2020

## Python: Zur Geschichte

Mehr zur Entwicklungsgeschichte von Python erfahren Sie in dem Vorwort, das Guido van Rossum zur ersten Auflage von *Programming Python* (Mark Lutz, O'Reilly, 1996) geschrieben hat. Sie können es frei zugänglich unter <https://www.python.org/doc/essays/foreword> finden.

Kurzes Video von Guido van Rossum zur Geschichte und Zukunft von Python:  
<https://www.youtube.com/watch?v=J0Aq44Pze-w>

## Python: Der Name

Python wurde nach der britischen Komikergruppe Monty Python benannt, deshalb finden sich immer wieder Bezüge darauf in der Python-Dokumentation und in Tutorien.



Quelle: pixabay.com

## Warum Python?

Warum nicht?

## Warum Python?

- Ein Ziel bei der Entwicklung war es, Python möglichst einfach und übersichtlich zu gestalten, was es zu einer guten Sprache für Programmieranfänger macht.
- Python hat eine sehr klare Syntax, was die Lesbarkeit erleichtert.

## Warum Python?

- Wie viele andere Sprachen auch kann Python durch Pakete erweitert werden. Es ist deshalb sehr flexibel und leistungsfähig.

- Es gibt eine sehr große und aktive Nutzergemeinde und Python ist eine der wichtigsten Programmiersprachen in Forschung und Wirtschaft.

## Python 2 oder Python 3?

- Seit dem Erscheinen von Python 3 Ende 2008 wurden zwei Versionen von Python weiterentwickelt: Python 2 und Python 3. Die Entscheidung, welche Version man verwenden sollte, hing oft davon ab, ob benötigte Pakete schon für Python 3 erhältlich waren.
- Die Zeit für Python 2 ist abgelaufen: Im April 2020 erschien das letzte Sicherheitsupdate.
- Wir werden uns deshalb auf Python 3 konzentrieren.



# Einführung in die Programmierung mit Python

Installationsanleitungen Windows .....	1
Installationsanleitungen Linux .....	2
Installationsanleitungen macOS .....	2

## Installationsanleitungen Windows

### Python

Browser: <https://www.python.org/downloads/>

- Herunterladen des Installers über den Download-Button
- Nachdem Download fertig, den heruntergeladenen Installer starten

Im Installer-Dialog:

- „Add Python 3.x to PATH“ ankreuzen
- Install Now klicken
- Warten bis Installation abgeschlossen.

Python ist nun installiert. Sie können im Startmenü dann nach Python suchen und entweder die Anwendung „IDLE“ starten oder einfach in der Eingabeaufforderung **python** tippen, um in die interaktive Python-Konsole zu gelangen.

### Visual Studio Code

Browser: <https://code.visualstudio.com>

- Herunterladen des Installers für Ihr Betriebssystem
- Nachdem Download fertig, den heruntergeladenen Installer starten

Im Installer-Dialog die gewünschten Optionen anpassen, die Voreinstellungen sollten aber passen.

Visual Studio Code ist nun installiert und sollte über das Startmenü auffindbar sein.

Starten Sie Visual Studio Code und installieren Sie die Erweiterung „Python“. Sofern Sie das Tool gerne in Deutsch verwenden würden, installieren Sie auch noch die Erweiterung „German Language Pack“.

# Installationsanleitungen Linux

## Python

Python 3 ist bei Ubuntu ab Version 16.04 bereits vorinstalliert, d.h. hier ist nichts zu tun. Das Kommando um Python auszuführen ist

```
python3
```

## Visual Studio Code

Den Software-Store von Ubuntu öffnen, nach Visual Studio Code suchen und installieren.

Danach wie oben bei Windows beschrieben die Erweiterungen „Python“ und bei Bedarf „German Language Pack“ installieren.

# Installationsanleitungen macOS

## Python

Browser: <https://www.python.org/downloads/macos/>

- Falls Sie einen Intel-basierten Mac haben (das sind alle Modelle bis etwa 2020), so laden Sie im Abschnitt *Stable Releases* den aktuellsten Installer herunter, der „**Intel**“ im Namen hat. Ansonsten laden Sie den aktuellsten Installer herunter, der „**universal2**“ im Namen hat.
- Nachdem Download fertig, den heruntergeladenen Installer öffnen

Im Installer-Dialog:

- Klicken Sie sich durch den Installer (akzeptieren Sie alles)
- Warten bis Installation abgeschlossen.

Python ist nun installiert. Sie können die Anwendung „IDLE“ (einfaches Python-Entwicklungstool) starten oder einfach im Terminal **python3** tippen, um in die interaktive Python-Konsole zu gelangen.

## Visual Studio Code

Browser: <https://code.visualstudio.com>

- Herunterladen des Installers für macOS (das ist ein Zip-Archiv)
- Nachdem Download fertig, das Zip-Archiv öffnen
- Im Finder dann die entpackte Datei „Visual Studio Code“ in den Programme-Ordner verschieben.

Visual Studio Code ist nun installiert.

Starten Sie Visual Studio Code und installieren Sie die Erweiterung „Python“. Nach der Installation der Erweiterung sehen Sie bei der Python-Erweiterung ein Zahnrad-Icon für Einstellungen. Klicken Sie das Zahnrad und dann auf „Extension Settings“. Scrollen Sie bis zur Einstellung „Python: Python Path“ und stellen Sie sicher, dass da „**python3**“ steht, und nicht nur „**python**“. Das stellt sicher, dass die soeben installierte Version 3 von Python verwendet wird und nicht die ggf. ebenfalls installierte Version 2.

Sofern Sie das Tool gerne in Deutsch verwenden würden, installieren Sie auch noch die Erweiterung „German Language Pack“.

# Werkzeuge

## Editoren: Wichtige Funktionen

Grundsätzlich kann man Python-Code in jedem Texteditor schreiben, selbst in so einfachen Anwendungen wie etwa Notepad in Windows. Sie bekommen hierbei aber nur ein Grundmaß an Funktionalität. Besser sind hier Editoren, die einige Zusatzfunktionen anbieten wie etwa:

- Codevervollständigung (*Code Completion*): Man tippt die ersten Buchstaben etwa eines Funktionsnamens, und der Editor zeigt dann einen oder mehrere Vorschläge an.
- Syntaxhervorhebung (*Syntax Highlighting*): Unterschiedliche syntaktische Elemente haben unterschiedliche Farben.
- Automatischer Zeileneinzug (*Auto Indentation*)

## Python: Warum Einrückungen wichtig sind

Viele Programmiersprachen verwenden etwa Klammern, um Code in Blöcken zu organisieren. Solche Blöcke können beispielsweise Schleifen sein, in denen man die gleiche Operation für jeden Bestandteil einer Sammlung von Werten ausführen lässt. Hier ist ein Beispiel in R: Es gibt die Zahlen von eins bis zehn nacheinander aus:

```
for(i in 1:10){  
  print(i)  
}
```

Da R Klammern verwendet um den Code zu strukturieren, könnte man das ganze Stück auch linksbündig (und/oder in einer einzigen Zeile) schreiben und es würde immer noch funktionieren:

```
for(i in 1:10){  
  print(i)  
}  
  
for(i in 1:10) {print(i)}
```

Für Python jedoch sind solche Einrückungen unerlässlich, denn sonst kann es die Rang- und Reihenfolge der verschiedenen Codesegmente nicht korrekt bewerten. Auch muss die Anzahl der Einrückungen pro Code-Level konstant bleiben. Normalerweise verwendet man vier Leerstellen; Sie sollten hier den Tabulator vermeiden:

```
for i in range(1, 11):  
    print(i)
```

1  
2  
3  
4  
5  
6  
7  
8

Dadurch, dass Blöcke eingerückt und weniger Klammern verwendet werden, wird der Code aber auch leichter lesbar.

## Editoren: Beispiele

Wenn Sie sich Python von der Python-Seite heruntergeladen und installiert haben, dann haben Sie auch IDLE (**I**ntegrated **L**earning and **D**evelopment **E**nvironment), einen (sehr) einfachen Editor, mit dem Sie Python-Code schreiben und ausführen lassen können.

Weitere gute Werkzeuge sind:

- [Notepad++](#): Klein, aber leistungsfähig. Kostenlos, allerdings nur Windows.
- [Google Colab](#): Webbasiert, also auch auf Tablets nutzbar. Kostenlose Version bietet schon gute Funktionalität zur Erstellung von Notebooks. Google-Konto erforderlich.
- [Sublime Text](#): Sehr leistungsfähig. Zeitlich unbegrenzter Test möglich. Windows, macOS, Linux.
- [Atom](#): Sehr leistungsfähig. Kostenlos. Windows, macOS, Linux.

## Editoren: Beispiele

- [Jupyter Notebook / JupyterLab](#): Besonders für die Arbeit mit Jupyter Notebooks geeignet, kann aber auch viele andere Sprachen unterstützen. Läuft im Webbrowser, aber meist lokale Installation. Kann über den Paketemanager pip installiert werden. (Folien für den Kurs wurden mit Jupyter Notebook erstellt.)
- [Spyder](#): Sehr leistungsfähig. Auf Python spezialisiert und in Python geschrieben. Kostenlos. In der Forschung beliebt. Kann über den Paketemanager pip installiert werden.
- [PyCharm](#): Sehr leistungsfähig. Auf Python spezialisiert. Kostenlose Community Edition oder kostenlose Studentenlizenz für PyCharm Professional.
- [Microsoft Visual Studio Code](#): **Unsere Empfehlung**. Sehr leistungsfähig mit sehr guter Python-Unterstützung durch Extensions. Kostenlos für Windows, macOS, Linux und keine Registrierung erforderlich. Auch für Jupyter Notebooks gut geeignet.

# Rechnen mit Python

## Arithmetische Operatoren: Python als Taschenrechner

Arithmetische Operatoren erlauben die Durchführung grundlegender Rechenoperationen selbst in der Python-Konsole:

Operation	Operator	Beispiel	Ergebnis	Erklärung
Addieren	+	1 + 1	2	
Subtrahieren	-	1-1	0	
Multiplizieren	*	2 * 2	4	
Dividieren	/	4 / 2	2	
Potenzieren	**	5 ** 2	25	5 hoch 2
Teilungsrest (Modulus)	%	7 % 3	1	Welcher Rest bleibt, wenn man 7 durch 3 teilt?
Ganzzahldivision	//	10 // 3	3	10 wird durch 3 geteilt und das Ergebnis zur nächst kleineren ganzen Zahl abgerundet

## Arithmetische Operatoren: Python als Taschenrechner

Überlegen Sie sich jetzt ein paar Beispiele und geben Sie diese in der Konsole ein.

## Hello World!

Seit der ersten öffentlichen Verwendung in dem 1978 erschienenen Programmierhandbuch *Programmieren in C (Programming in C)* von Brian Kernighan und Dennis Ritchie ist es eine Tradition geworden, dass das erste Programm, das man in einer neuen Sprache schreibt, die Worte "Hello World!" ausgibt.

Übrigens: Auf der Seite <http://helloworldcollection.de> finden Sie "Hello World!"-Programme in mehreren hundert Sprachen.

Versuchen Sie es also! Starten Sie eine Python-Konsole, geben dann den Text `print("Hello World!")` ein und drücken dann die Eingabetaste:

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

## Hello World! - Was passiert hier?

In dieser kurzen Zeile sehen wir bereits einige wichtige Aspekte von Python und auch anderen Programmiersprachen sowie einen der augenfälligsten Unterschiede zwischen Python 2 und Python 3:

- Es gibt Funktionen. Funktionen erlauben es Ihnen, Code für eine bestimmte Aufgabe zu schreiben und diesen dann immer wieder zu verwenden. Viele Funktionen haben Parameter, d.h., Sie können Objekte wie etwa die Phrase "Hello World!" oder eine Zahl als Argumente an die Funktion weitergeben und von der bearbeiten lassen. Man kann die bereits in Python integrierten Funktionen nutzen (wie hier `print()`) oder eigene Funktionen schreiben.

## Hello World! - Was passiert hier?

- Es gibt verschiedene Datentypen. "Hello World!" ist eine Zeichenkette (character string), also eine Abfolge von Buchstaben, Leerzeichen, Satzzeichen, oder auch Ziffern. Es steht deshalb in Anführungszeichen, während etwa eine Zahl wie 12345 ohne Anführungszeichen auskommt.
- Beachten Sie folgenden Unterschied zwischen Python 2 und 3: In Python 3 ist `print` eine Funktion und der auszugebende Text steht in Klammern: `print("Hello World!")`. In Python 2 war `print` eine Anweisung und verwendete deshalb keine Klammern: `print "Hello World!"`

# Variablen

## Variablen

Variablen sind ein wichtiges Werkzeug, wenn man mit Python (und anderen Sprachen arbeitet):

```
a = 1
```

In diesem Fall haben wir ein sog. Objekt (*object*) erzeugt (dazu später mehr) und einer **Variable** (*variable*) zugewiesen. Eine Variable ist ein Verweis auf einen gespeicherten Wert. Das kann ein einzelnes Zeichen oder ein ganzer Datensatz sein. Sie können sich den Wert der Variable anzeigen lassen, indem Sie einfach deren Namen in die Konsole eingeben:

```
>>> a = 1  
>>> a  
1
```

## Variablen

Der Wert der Variable kann jederzeit geändert werden:

```
>>> a = 2  
>>> a  
2
```

## Variablen

Man kann dem gleichen Objekt eine zweite Variable zuweisen:

```
>>> b = a  
>>> b  
2
```

## Variablen

Hier wird kein zweites Objekt erzeugt, sondern nur ein zweiter Verweis. Sie können das sehen, wenn Sie sich die ID des Objektes mit der **id()**-Funktion anzeigen lassen:

```
>>> id(a)  
140717129051168  
>>> id(b)  
140717129051168
```

Diese IDs werden sich von Computer zu Computer und Sitzung zu Sitzung unterscheiden.

## Variablen

Was passiert, wenn Sie jetzt den Wert von b ändern und die ID von b ausgeben lassen?

```
>>> b = 300
```

## Variablen

```
>>> b = 300
>>> b
300
>>> id(b)
1524496151568
```

Python hat jetzt ein neues Objekt mit einer neuen ID erzeugt.

## Variablen

Was passiert, wenn Sie die folgenden Zeilen eingeben und dann die IDs testen:

```
>>> a = 300
>>> b = 300
>>> c = 30
>>> d = 30
```

## Variablen

```
>>> a = 300
1524496152240
>>> b = 300
1524496152176
>>> c = 30
140717129045728
>>> d = 30
140717129045728
```

Was ist hier passiert?

## Variablen

Als Sie Python gestartet haben, hat Python für einen kleinen Zahlenbereich bereits Objekte kreiert (ganze Zahlen von -5 bis 256) und verwendet sie dann bei der Ausführung Ihres Codes. Python kann dadurch etwas effizienter arbeiten.

## Variablennamen: Regeln

Variablennamen können folgende Zeichen beinhalten:

- Buchstaben
- Zahlen
- Unterstrich (\_)

# Variablenamen: Regeln

Geben Sie die folgenden Zeilen in die Konsole ein. Was passiert?

```
>>> test = "ab"  
>>> _test = "ab"  
>>> 12 = "ab"
```

# Variablenamen: Regeln

```
>>> test = "ab"  
>>> _test = "ab"  
>>> 12 = "ab"  
    File "<stdin>", line 1  
SyntaxError: can't assign to literal
```

# Variablenamen: Regeln

Es bestehen zwei wichtige Einschränkungen: Variablennamen dürfen nur mit einem Unterstrich oder einem Buchstaben beginnen.

Ein weiterer wichtiger Punkt: Python achtet bei Variablennamen auf Groß- und Kleinschreibung.  
Geben Sie folgende Zeilen ein:

```
>>> a = 1  
>>> a  
>>> A
```

# Variablenamen: Regeln

```
>>> a = 1  
>>> a  
1  
>>> A  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'A' is not defined
```

# Variablenamen: Format

Aus technischer Sicht steht es Ihnen frei, wie Sie Variablennamen gestalten, solange Sie Namen, die aus mehreren Teilen (Wörtern, Zahlen) bestehen, zusammenschreiben. So können Sie etwa eine Variable, die sich auf die Durchschnittstemperatur für Mai 2020 bezieht, folgendermaßen schreiben:

DURHSCHNITTTEMPERATURMAI2020

# Variablenamen: Format

Das ist allerdings schwer zu lesen. Bessere Alternativen sind:

- Der erste Buchstabe in jedem Wort wird groß-, der Rest kleingeschrieben (Pascal Case):

DurchschnittTemperaturOktober2019

- Wie bei Pascal Case, nur wird der erste Buchstabe des Namens kleingeschrieben (Camel Case):

durchschnittTemperaturOktober2019

## Variablennamen: Format

Der Style Guide for Python Code (PEP, oder Python Enhancement Proposal 8, <https://www.python.org/dev/peps/pep-0008/>) empfiehlt, dass Variablennamen (und auch die Namen von Funktionen) immer klein geschrieben und Wörter durch einen Unterstrich getrennt werden (snake case):

durchschnitt\_temperatur\_oktober\_2019

Die Wahl bleibt letztendlich Ihnen überlassen, solange Sie konsistent sind. Wenn Sie mit anderen an einem Projekt arbeiten, fragen Sie, ob es interne Richtlinien gibt.

## Variablennamen: Schlüsselwörter (*keywords*)

Es gibt eine Anzahl von Schlüsselwörtern (*keywords*), die Sie nicht als Variablennamen verwenden dürfen, da sie schon eine bestimmte Rolle in Python spielen. Die Liste kann sich bei einer neuen Python-Version ändern.

Die aktuelle Liste finden Sie mit **help("keywords")**.

## Variablennamen: Schlüsselwörter (*keywords*)

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

## Variablennamen: Was noch zu beachten ist

- Verwenden Sie klare und eindeutige Namen:

```
durchschnitt_temperatur_mai_2020 statt xy
```

- Sie können zwar die Namen von existierenden Funktionen als Variablennamen benutzen, sollten es aber nicht tun, um Verwirrungen zu vermeiden:

```
a = range(1,6) # Weist a die Zahlen von 1 bis 5 zu
sum = sum(a)
```

# Datentypen

## Zu einem anderen Thema:

Geben Sie den folgenden Code in die Konsole ein:

```
>>> 1 + 1
>>> 1 + "a"
>>> "a" + "b"
>>> 3 * "abc"
```

Welche Ergebnisse bekommen Sie?

```
>>> 1 + 1
2
>>> 1 + "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> "a" + "b"
'ab'
>>> 3 * "abc"
'abcabcabc'
>>>
```

## Datentypen

Der Grund für den Fehler ist, dass es in Python verschiedene Datentypen gibt, mit denen man unterschiedliche Sachen machen kann und die etwa von arithmetischen Operatoren unterschiedlich behandelt werden.

In diesem Fall haben Sie versucht, den Operator `+` mit zwei verschiedenen Datentypen (**int** [integer] und **str** [string]) zu verwenden, was dieser nicht erlaubt.

## Datentypen: Boolesche Werte

Ein *bool* (*boolean*) kann nur einen von zwei Werten haben: *True* (Wahr) oder *False* (Falsch). *True* wird mit 1 gleichgesetzt und *False* mit 0. Dieser Typ kommt etwa zum Einsatz, wenn Sie testen, ob der Wert einer Variablen eine bestimmte Eigenschaft hat:

```
a = "Hallo"

a.isalpha() # Beinhaltet a nur Buchstaben?
```

True

```
a.isalnum() # Beinhaltet a nur alphanumerische Zeichen?
```

```
True
```

```
a.isdigit() # Beinhaltet a nur Zahlen?
```

```
False
```

## Datentypen: Zeichenketten

Zeichenketten, oder *character strings/strings*, sind Aneinanderreihungen von Zeichen (Buchstaben, Zahlen, Satzzeichen, Leerstellen, usw.) von beliebiger Länge. Sie werden bei der Definition immer in Anführungszeichen (normalerweise doppelt) gesetzt:

```
>>> a = "Hello World!"
```

## Datentypen: Zeichenketten

Wenn eine Zeichenkette ein Anführungszeichen enthalten soll, haben Sie zwei Möglichkeiten:

- Sie setzen vor das Anführungszeichen einen *backslash*. Dieser funktioniert als sog. Escape-Zeichen (*escape character*):

```
>>> a = "Sie sagt \"Hallo\""
```

- Wenn Sie ein doppeltes Anführungszeichen in die Zeichenkette einfügen wollen, umschließen Sie die ganze Kette mit einfachen Anführungszeichen und umgekehrt:

```
>>> a = 'Sie sagt "Hallo"'
```

## Datentypen: Integer

Ein *integer* ist eine ganze Zahl, also eine Zahl ohne Nachkommastellen. Geben Sie die folgende Zeile ein, um der Variable a den Wert 1 zuzuweisen und lassen Sie sich dann mit **type()** den Datentyp anzeigen.

```
>>> a = 1  
>>> type(a)  
<class 'int'>
```

## Datentypen

Was passiert, wenn Sie stattdessen 1.5 eingeben und sich dann den Typ anzeigen lassen?

```
>>> a = 1.5  
>>> type(a)  
<class 'float'>
```

## Datentypen: Gleitkommazahl

1.5 ist eine sog. Gleitkommazahl (*float*, oder *floating-point number*), d.h., eine Zahl, die auch Nachkommastellen hat. Bitte beachten Sie, dass Python nicht--wie im Deutschen üblich--ein Komma verwendet, um Nachkommastellen zu signalisieren, sondern einen Punkt.

## Datentypen: Gleitkommazahl und wissenschaftliche Notation

Gleitkommazahlen lassen sich auch dazu verwenden, um sehr große oder sehr kleine Zahlen darzustellen. Geben Sie die folgenden Zeile ein:

```
1 / 5**25
```

## Datentypen: Gleitkommazahl und wissenschaftliche Notation

Das Ergebnis wird Ihnen in der Exponentialdarstellung, oder traditionellen wissenschaftlichen Notation dargestellt:

```
3.3554432e-18
```

Dies entspricht  $3.3554432 \times 10^{-18}$

## Datentypen: Komplexe Zahl

Komplexe Zahlen seien hier nur der Vollständigkeit halber erwähnt.

```
>>> a = 2 + 2j  
>>> type(a)  
<class 'complex'>
```

$j$  ist eine imaginäre Zahl mit der Eigenschaft  $j^2 = -1$ . Die Verwendung von  $j$  stammt aus den Ingenieurwissenschaften, wohingegen Mathematiker  $i$  verwenden, um den Imaginärteil zu kennzeichnen.

## Vorsicht bei Berechnungen

Welches Ergebnis erwarten Sie bei den folgenden Berechnungen?

```
>>> 1.1 + 2.2  
>>> 1 / 10 + 2 / 10  
>>> 1 / 10 + 2 / 10 == 0.3
```

## Vorsicht bei Berechnungen

Welches Ergebnis bekommen Sie, wenn Sie die folgende Zeilen in die Konsole eingeben?

```
>>> 1.1 + 2.2
>>> 1 / 10 + 2 / 10
>>> 1 / 10 + 2 / 10 == 0.3
```

## Vorsicht bei Berechnungen

```
3.300000000000003
0.3000000000000004
False
```

## Vorsicht bei Berechnungen

Computer speichern und bearbeiten Zahlen im Binärformat, das nur 0 und 1 kennt. So wird etwa die Zahl 7 als 111 dargestellt:

$$7_{10} = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 111_2$$

Das funktioniert meist ganz gut, sorgt aber etwa bei Zahlen mit unendlich vielen Nachkommastellen (etwa 1/3, also 0,333333...) oder bei bestimmten Brüchen für Probleme, da diese nur annähernd im Binärformat dargestellt und gespeichert werden können.

Hier eine gute Seite für die Umrechnung von Zahlen in verschiedene Formate:

<https://www.matheretter.de/rechner/zahlenkonverter>

## Zur Zahlendarstellung

Seit kurzem (Python 3.6) können Sie Unterstriche verwenden, um Ziffern zu gruppieren und große Zahlen dadurch leichter lesbar zu machen. Das heißt, 1000000 (eine Million) kann auch so geschrieben werden: 1\_000\_000.

## Zur Zahlendarstellung

Zahlen werden in Python normalerweise im Dezimalformat dargestellt. Indem man einer Zahl eines von mehreren Präfixen voranstellt, kann auch ein anderes Format verwendet werden:

Präfix	System	Beispiel	Entspricht im Dezimalsystem
0b	Binär	0b1111011	123
0o (Null o)	Oktal	0o173	123
0x	Hexadezimal	0x7B	123

## Zur Zahlendarstellung

Sie können diese alternativen Formate auch bei Berechnungen verwenden:

```
>>> 0xA * 0b1010 # 10 (Hexadezimal) * 10 (Binär)
100
```

# Kommentare

## Kommentare

Wenn Sie in Ihrem Python-Code Erklärungen für andere Nutzer--oder Erinnerungsstützen für sich selbst--einfügen möchten, können Sie das mit Kommentaren (*comments*) machen. Diese Kommentare werden bei der Ausführung Ihres Codes ignoriert; Kommentare sind daher auch nützlich, wenn Sie verschiedene Versionen etwa einer Funktion ausprobieren möchten. Die jeweils nicht benötigten Versionen können dann auskommentiert werden.

## Kommentare

Kommentare beginnen immer mit "#". Man kann einen Kommentar entweder an das Ende einer Zeile Code oder, was vorzuziehen ist, in seine eigene Zeile schreiben.

OK:

```
a = 1 # Das ist ein Kommentar.
```

Besser:

```
# Das ist auch ein Kommentar.
```

## Kommentare

Es gibt in Python nicht, wie in manchen anderen Sprachen, sogenannte Blockkommentare, mit denen man längere Codeblöcke auf einmal auskommentieren kann. Stattdessen müssen alle Zeilen individuell kommentiert werden. Viele Editoren erlauben es Ihnen aber, Blöcke auszuwählen und dann alle darin enthaltenen Zeilen auf einmal auszukommentieren und die "#" dann auch wieder zu entfernen.

In Microsoft Visual Studio Code können Sie etwa folgendermaßen vorgehen:

- Zeilen auswählen
- Bearbeiten -> Zeilenkommentar umschalten

Oder

- Zeilen auswählen
- Strg + #

## Docstrings

Später im Kurs werden wir noch *Docstrings* (*documentation strings*) kennenlernen. Diese dienen zur Dokumentation von Funktionen, Methoden, Klassen und Modulen (was das ist, werden wir später

noch sehen), während Kommentare eher zur Erklärung von kleinen Code-Fragmenten dienen. *Docstrings* beginnen und enden mit drei doppelten Anführungszeichen (""""") und können über mehrere Zeilen gehen.

Beispiel anhand einer Funktion "addieren", die zwei Zahlen addiert:

```
def addieren(a, b):
    """Addiert zwei Zahlen und gibt die Summe zurück."""
    return a + b
```

## Docstrings

Wenn Sie dann später mehr über diese Funktion wissen möchten, können Sie sich die Docstrings anzeigen lassen.

```
help(addieren)

Help on function addieren in module __main__:

addieren(a, b)
    Addiert zwei Zahlen und gibt die Summe zurück.
```

```
print(addieren.__doc__)

Addiert zwei Zahlen und gibt die Summe zurück.
```

# Unicode

## Unicode

Zum Anfang des Kurses haben wir bereits etwas über die Unterschiede zwischen Python 2 und 3 gesprochen. Ein weiterer wichtiger Unterschied ist, dass Python 3 standardmäßig *Unicode* für die Übersetzung von Bytes in Schriftzeichen und umgekehrt verwendet.

Was ist Unicode?

Lange Geschichte...

## Unicode

- Alle Daten, also auch die Zeichen, die Sie auf Ihrem Bildschirm sehen, werden als Bits (0 oder 1) gespeichert.
- Anfangs war Speicherplatz extrem teuer und man versuchte--wann immer möglich--Speicher zu sparen.\*

\* Aus diesem Grund wurden auch Jahreszahlen oft nur zweistellig gespeichert, was zum Ende des 20. Jahrhunderts einer der Gründe für die aufkommende Angst vor dem Jahr-2000-Problem (*Millenium Bug, Y2K-Bug*) war. Der befürchtete weitgehende Zusammenbruch digitaler Infrastrukturen blieb jedoch aus.

## Unicode

- Zeichensatztabellen (*codepages*), in denen Zeichen und ihre Bytewerte gespeichert werden, hatten ursprünglich nur wenige Zeichen. ASCII (*American Standard Code for Information Interchange*) etwa verwendete anfangs 7 bits, d.h., es können  $2^7$ , oder 128, Zeichen dargestellt werden. Dazu gehören auch Steuerzeichen wie Tabulator.
- Spätere Tabellen wurden auf 8 bit (auch eine erweiterte Form von ASCII) vergrößert, um Sonderzeichen wie etwa die deutschen Umlaute berücksichtigen zu können.
- Da sich im Laufe der Zeit immer mehr Tabellen entwickelten, um mehr Sprachen abdecken zu können, musste man immer angeben, welche Tabelle in einer Datei verwendet wurde.

## Unicode

- Jedem einzelnen Zeichen immer mehr Speicher zur Verfügung zu stellen hilft auf die Dauer auch nicht, da dadurch in den meisten Fällen Speicher verschwendet wird.
- Unicodes Lösung: Eine einzelne Tabelle mit jedem bekannten Zeichen und unterschiedliche langen Byte-Sequenzen.

## Unicode

- Unicode verwendet verschiedene Formate, um Zeichen zu kodieren. Das gebräuchlichste ist UTF-8, welches bis zu 7 Bytes pro Zeichen verwendet (für die ASCII-Zeichen benötigte man nur 1 Byte, oder 8 bit).
- Seit Python 3 werden Zeichenketten automatisch als Unicode gespeichert. Sie können also auch Zeichen wie Schriftzeichen aus verschiedenen asiatischen Sprachen oder Emojis direkt in Zeichenketten verwenden.

## Unicode

Unicode-Homepage:

<https://home.unicode.org/>

Gute englischsprachige Zusammenfassung zu Unicode/Unicode in Python:

<https://docs.python.org/3.3/howto/unicode.html>

Suchmaschine für Unicode-Zeichen:

<https://unicode-table.com/de/>

# Datenstruktur Liste

## Datenstrukturen: Listen

Bisher haben wir nur jeweils einen einzelnen Wert gespeichert. Pythons eingebaute Datenstrukturen erlauben es Ihnen aber auch, mehrere Werte auf einmal abzuspeichern. Ein einfaches Beispiel sind Listen (*lists*):

```
a = ["apfel", "banane", "banane", "kirsche"]
b = [1, 2, 4, 5, 8, 17]
```

## Datenstrukturen: Listen

Listen können mehrere Datentypen enthalten:

```
a = [1, 2, 4, "apfel", "kirsche", "banane"]
```

Und sogar andere Listen:

```
b = [1, 2, [3, 4, 5]]
```

## Datenstrukturen: Listen

Sie können über den Indexwert eines Wertes, d.h. dessen Position in der Liste, auf diesen zugreifen.

Bitte beachten Sie, dass in Python--wie in fast allen anderen Programmiersprachen--die Zählung bei 0, nicht bei 1, beginnt:

```
a = [1, 2, 8, 17, 5, 4]
erster_wert = a[0]
letzter_wert = a[-1]
```

## Datenstrukturen: Listen

Es gibt auch eine Vielzahl von nützlichen Funktionen zu Listen, wie etwa **sum()**, mit der Sie Werte in einer Liste aufaddieren können:

```
a = [1, 2, 8, 17, 5, 4]
b = sum(a)
b
```

## Datenstrukturen: Listen

Auch können Sie etwa Werte in Listen zählen lassen:

```
a = ["apfel", "banane", "banane", "kirsche"]
a.count("banane")
```

2

## Datenstrukturen: Listen

Es gibt verschiedene Möglichkeiten, Listen sortieren zu lassen:

```
a = [1, 2, 8, 17, 5, 4]
a.sort()
a
```

[1, 2, 4, 5, 8, 17]

## Datenstrukturen: Listen

Beachten Sie, dass wir hier die sortierte Liste nicht einer neuen Variable zuweisen. **sort()** sortiert die Liste *in-place*, d.h., die Originalliste wird verändert. Wenn Sie die ursprüngliche Liste behalten und stattdessen eine neue, sortierte Liste erzeugen wollen, verwenden Sie **sorted()**:

```
c = sorted(a)
c
```

[1, 2, 4, 5, 8, 17]

## Datenstrukturen: Listen

Noch ein Punkt zur Sortierung:

Geben Sie die folgende Liste in der Konsole ein und lassen sie sortieren:

```
a = ["kirsche", "apfel", "Banane"]
```

Welches Ergebnis bekommen Sie?

## Datenstrukturen: Listen

Sie sollten folgendes Ergebnis bekommen:

```
a = ["kirsche", "apfel", "Banane"]
b = sorted(a)
b
```

['Banane', 'apfel', 'kirsche']

Beachten Sie: Die Funktion sortiert Großbuchstaben vor Kleinbuchstaben.

# Datenstrukturen: Listen

Wenn Sie möchten, dass die Werte in der Liste unabhängig von Groß/Kleinschreibung sortiert werden, verwenden Sie **sorted()** mit dem Parameter **key**. Mit **key** können Sie eine Funktion wählen, die den für die Sortierung verwendeten Wert generiert. Hier ist das **str.lower**, welche alle Groß- in Kleinbuchstaben umwandelt:

```
a = ["kirsche", "apfel", "Banane"]
b = sorted(a, key=str.lower)
b
```

```
['apfel', 'Banane', 'kirsche']
```

**key** wirkt sich hier nicht auf den in der Liste gespeicherten Wert aus, nur darauf, wie er während der Sortierung behandelt wird.

# Datenstrukturen: Listen

Wenn Sie zwei Listen zusammenfügen möchten, können Sie das mit dem **+**-Operator erreichen:

```
a = [1, 2]
b = [3, 4]
c = a + b
c
```

```
[1, 2, 3, 4]
```

# Datenstrukturen: Listen

Listen gehören zu den veränderbaren, oder mutablen (*mutable*) Datenstrukturen. Das bedeutet, dass sie nach ihrer Erzeugung verändert werden können. Im folgenden Beispiel wird das erste Element der Liste a ("banane") durch "birne" ersetzt:

```
a = ['banane', 'ananas', 'apfel', 'kirsche', 'apfel']
a[0] = 'birne'
a
```

```
['birne', 'ananas', 'apfel', 'kirsche', 'apfel']
```

# Datenstrukturen: Listen

Sie können auch mehrere Werte auf einmal ändern. Im folgenden Beispiel werden der zweite und dritte Wert der Liste durch zwei andere Werte ersetzt:

```
a = ['banane', 'ananas', 'apfel', 'kirsche', 'apfel']
a[1:3] = ['erbse', 'gurke']
a
```

```
['banane', 'erbse', 'gurke', 'kirsche', 'apfel']
```

# Datenstrukturen: Listen

Mit **append()** können Sie einzelne Werte an eine Liste anfügen. Falls mehrere Werte (etwa aus einer anderen Liste) dazukommen sollen, verwenden Sie **extend()**:

```
a = ['banane', 'ananas', 'apfel', 'kirsche', 'apfel']
a.append('erbse')
a
```

```
['banane', 'ananas', 'apfel', 'kirsche', 'apfel', 'erbse']
```

```
a.extend(['gurke', 'tomate'])
a
```

```
['banane', 'ananas', 'apfel', 'kirsche', 'apfel', 'erbse', 'gurke', 'tomate']
```

# Datenstrukturen: Listen

So wie Sie bestimmte Werte aus einer Liste auswählen können, so können Sie auch einzelne oder mehrere Werte aus einer Liste löschen:

```
a = ['banane', 'ananas', 'apfel', 'kirsche', 'apfel', 'erbse', 'gurke',
      'tomate']
del a[0] # Löscht ersten Wert. a hat jetzt 7 Werte.
a
```

```
['ananas', 'apfel', 'kirsche', 'apfel', 'erbse', 'gurke', 'tomate']
```

```
del a[1:3] # Löscht zweiten und dritten Wert. a hat jetzt 5 Werte.
a
```

```
['ananas', 'apfel', 'erbse', 'gurke', 'tomate']
```

```
del a[0:5:2] # Löscht ersten, dritten, fünften usw. Wert. a hat jetzt 2 Werte.
a
```

```
['apfel', 'gurke']
```

# Datenstruktur Tupel

## Datenstrukturen: Tupel

Im Vergleich zu Listen haben Tupel (*tuples*) zwar eine ähnliche Struktur, bieten aber weniger Flexibilität, da sie unveränderbar (*immutable*, *immutable*) sind. Tupel werden mit runden Klammern erzeugt:

```
a = ('banane', 'ananas', 'apfel', 'kirsche', 'apfel')
```

## Datenstrukturen: Tupel

Werte in einem Tupel können wieder über ihre Indexwerte adressiert werden:

```
a = ('banane', 'ananas', 'apfel', 'kirsche', 'apfel')
a[0]
```

```
'banane'
```

```
a[0:3]
```

```
('banane', 'ananas', 'apfel')
```

## Datenstrukturen: Tupel

Was passiert, wenn Sie versuchen, einen Wert aus dem Tupel zu löschen?

```
del a[0]
```

```
-----  
TypeError                                         Traceback (most recent call last)  
<ipython-input-26-c47830f0ae67> in <module>  
----> 1 del a[0]  
  
TypeError: 'tuple' object doesn't support item deletion
```

Operationen, welche ein Tupel verändern würden, führen zu Fehlermeldungen.

## Datenstrukturen: Tupel

Bei der Erzeugung eines Tupel können Sie die runden Klammern auch weglassen:

```
person = 'karl', 'theodor', 'mustermann'
```

Das Zuweisen von Werten zu einem Tupel heißt  *Tuple Packing*. Beim  *Tuple Unpacking* können Sie dann jeden der Werte im Tupel einer Variablen zuweisen und so dann gezielt aufrufen. Auf diese

Weise kann man auch in einer Zeile mehrere Variablen erzeugen. Auch beim *Tuple Unpacking* können die runden Klammern weggelassen werden.

```
vorname, patenname, nachname = person  
vorname
```

```
'karl'
```

# Strings

## Strings

Wir befassen uns in diesem Video mit *strings* oder *Zeichenketten*, die Sie bereits aus dem Video zu Datentypen kennen. Python bietet eine Vielzahl von Werkzeugen, um mit Zeichenketten zu arbeiten.

## Strings: Konkatenation mit *join()*

Konkatenation (*concatenation*) bedeutet soviel wie Verketten oder Zusammenfügen und bezeichnet den Prozess, in dem man einzelne Zeichen (z.B. Buchstaben) oder Zeichenketten/*strings* (z.B. Wörter) zu längeren Ketten zusammensetzt. Python bietet dazu mehrere Möglichkeiten. So können Sie die Funktion **join()** verwenden, um Elemente etwa aus einer Liste oder einem Tupel zu verbinden:

```
a = ["Das", "ist", "eine", "Zahl"]
" ".join(a)
```

```
'Das ist eine Zahl'
```

Im Beispiel verwenden wir ein Leerzeichen (in den vorangestellten Anführungszeichen) als Verbindung zwischen den Listenelementen. Sie können hier auch kein oder mehrere Zeichen verwenden.

Auch ist es möglich, Variablenwerte einzufügen:

```
wert = "Das"
a = [wert, "ist", "eine", "Zahl"]
" ".join(a)
```

```
'Das ist eine Zahl'
```

Man muss aber darauf achten, dass der Variablenwert eine Zeichenkette ist. Wenn Sie stattdessen etwa eine Zahl verwenden, bekommen Sie eine Fehlermeldung. Sie müssten die Zahl dann in Anführungszeichen setzen, um sie in eine Zeichenkette umzuwandeln:

```
wert = 42
a = [wert, "ist", "eine", "Zahl"]
" ".join(a)
```

```
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-71-b1efef9e370a5> in <module>
      1 wert = 42
      2 a = [wert, "ist", "eine", "Zahl"]
----> 3 " ".join(a)

TypeError: sequence item 0: expected str instance, int found
```

```
wert = "42"
a = [wert, "ist", "eine", "Zahl"]
" ".join(a)
```

```
'42 ist eine Zahl'
```

## Strings: Konkatenation mit +

Ein anderer Ansatz verwendet den Operator `+`, den wir schon für die Addition von Zahlen genutzt haben. Bei Zeichenketten hat er aber einen anderen Effekt. Hier werden die einzelnen Elemente aneinandergefügten:

```
a = "ist"  
b = "Satz"  
c = "Das " + a + " ein " + b + ". "  
print(c)
```

```
Das ist ein Satz.
```

Man kann mit der Methode leicht vorgegebenen Text und Zeichenketten aus Variablen kombinieren und Sie können diesen Ansatz in vielen Codebeispielen sehen. Allerdings hat er auch einige Nachteile:

- Man kann beim Programmieren leicht Anführungs- und Pluszeichen vergessen und der Code läuft dann nicht mehr. Auch die korrekte Platzierung von Leerzeichen wird hier erschwert.
- Wenn Ihre Variablen Zahlenwerte enthalten, möchten Sie diese vielleicht für die Ausgabe formatieren, um beispielsweise weniger Nachkommastellen anzuzeigen. Bei der Konkatenation mit `+` wird das umständlich.
- Auch hier gilt die schon im Zusammenhang mit `join()` erwähnte Beschränkung, dass nur Zeichenketten verwendet werden können:

```
wert = 42  
a = wert + " ist eine Zahl."  
print(a)
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-74-fc6d31b4b8e0> in <module>  
      1 wert = 42  
----> 2 a = wert + " ist eine Zahl."  
      3 print(a)  
  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

## Strings: Formatierung mit `format()`

Python bietet hier auch bessere Alternativen an. Mit der Funktion `format()` werden geschweifte Klammern `{}` als Platzhalter in längeren Zeichenketten eingesetzt, die dann durch die Werte ersetzt werden, die man an die Funktion übergibt:

```
"Heute ist {}, der {}.{}.{}.format("Mittwoch", 3, 10, 1990)
```

```
'Heute ist Mittwoch, der 3.10.1990.'
```

Wir können jetzt Zeichenketten und Zahlen beliebig vermischen.

Die Reihenfolge, in der die Werte dabei eingesetzt werden, richtet sich dabei erst einmal danach, wie

sie an **format()** übergeben wurden. Sie können jedem Platzhalter auch einen Namen geben und die Werte dann dadurch adressieren:

```
"Heute ist {wochentag}, der {tag}.{monat}.{jahr}.".format(wochentag="Mittwoch",
tag=3, monat=10, jahr=1990)
```

```
'Heute ist Mittwoch, der 3.10.1990.'
```

Soll das Ergebnis selbst geschweifte Klammern enthalten, verwenden Sie doppelte Klammern:

```
"Das sind {g} {k} {{}} und das sind {r} {k} ().".format(g="geschweifte",
r="runde", k="Klammern")
```

```
'Das sind geschweifte Klammern {} und das sind runde Klammern ().'
```

Wie Sie sehen, kann ein Platzhalter auch mehrere Male verwendet werden, wenn Sie ihm einen Namen zugewiesen haben.

Wenn Sie einen der eingefügten Werte noch ändern müssen (etwa eine Zahl runden), kann dies als Teil des Formatierungsprozesses geschehen. Dies geschieht mithilfe von Pythons *Format Specification Mini-Language*. Hier etwa wird eine Zahl (vom Typ *float*) mit fünf Nachkommastellen auf zwei Nachkommastellen gerundet, um einen Eurobetrag angeben zu können. Sie müssen dann die formatierte Version nicht separat abspeichern:

```
a = 36.85739
"Die durchschnittlichen Kosten betragen {:.2f} Euro.".format(a)
```

```
'Die durchschnittlichen Kosten betragen 36.86 Euro.'
```

Mehr Informationen zur *Format Specification Mini-Language* finden Sie auf <https://docs.python.org/3/library/string.html#format-specification-mini-language>.

## Strings: Formatierung mit *f-strings*

Seit Python 3.6 gibt es mit **f-strings** (*formatted string literals*, formatierte String-Literale) noch eine weitere Möglichkeit zur Zeichenkettenformatierung, die eine noch einfachere Syntax bietet. Man kann sie an einem vorangestellten kleinen (oder großen) *f* erkennen:

```
a = 42
b = f"Die Antwort lautet {a}."
print(b)
```

```
Die Antwort lautet 42.
```

Beachten Sie, dass die Klammern keinen *backslash* enthalten und--anders als bei *format()*--nicht leer sein dürfen.

Im vorherigen Beispiel haben wir nur einen Variablenamen in den geschweiften Klammern angegeben. Wir können hier aber auch Rechenoperationen durchführen:

```
a = f"Die Antwort lautet {6 * 7}."
print(b)
```

Die Antwort lautet 42.

Es ist möglich, innerhalb der Klammern Variablen zu definieren (`c := 42`) und dann auch an Funktionen zu übergeben `a(c)` (mehr dazu im Video zu Funktionen):

```
def a(x):
    return x * x

b = f"Die Antwort auf die Frage 'Was ist {c := 42} mal {c}?' lautet {a(c)}."
print(b)
```

Die Antwort auf die Frage 'Was ist 42 mal 42?' lautet 1764.

Dieselbe *Format Specification Mini-Language*, welche wir eben schon im Zusammenhang mit `format()` kennengelernt haben, kommt auch bei *f-strings* zum Einsatz:

```
a = 36.85739
b = f"Die durchschnittlichen Kosten betragen {a:.2f} Euro."
print(b)
```

Die durchschnittlichen Kosten betragen 36.86 Euro.

## Strings: Methoden

Neben `join()` und `format()` bietet Python noch eine Vielzahl anderer Methoden für die Arbeit mit Zeichenketten. Wenn eine Methode Änderungen an der Zeichenkette vornimmt, betrifft das nicht das Original, sondern es wird stattdessen eine Kopie erzeugt.

Eine vollständige Liste finden Sie auf <https://docs.python.org/3/library/stdtypes.html#string-methods>. Hier seien nur einige Beispiele genannt. Diese verwenden den folgenden Beispielsatz:

```
zeichenkette1 = "Ich habe schon 5 Programme mit Python geschrieben \
und es hat mir Spaß gemacht."
```

### capitalize()

Mit Ausnahme des ersten Zeichens wandelt **capitalize()** alle Zeichen in ihre kleingeschriebene Version um:

```
zeichenkette2 = zeichenkette1.capitalize()
zeichenkette2
```

'Ich habe schon 5 programme mit python geschrieben und es hat mir spaß gemacht.'

### casefold()

**casefold()** wandelt alle Zeichen in ihre kleingeschriebenen Versionen um. Dabei werden auch Zeichen, von denen es keine großgeschriebenen Formen gibt, in ihre Equivalente umgewandelt. Das deutsche ß etwa wird zu ss. Wenn Sie das nicht möchten, können Sie die Funktion **lower()** verwenden, die später noch beschrieben wird.

```
zeichenkette2 = zeichenkette1.casefold()  
zeichenkette2
```

```
'ich habe schon 5 programme mit python geschrieben und es hat mir spass gemacht.'
```

## count()

**count()** zählt, wie oft ein Zeichen oder eine Zeichenkette vorkommt. Beachten Sie, dass Klein- und Großbuchstaben getrennt behandelt werden. Wenn Sie also sicherstellen wollen, dass beide Versionen gefunden werden, sollten Sie die Zeichenkette erst mit **lower()** in Kleinbuchstaben umwandeln und dann zählen.

```
ergebnis = zeichenkette1.count("I")  
ergebnis
```

```
1
```

```
ergebnis = zeichenkette1.count("i")  
ergebnis
```

```
3
```

```
ergebnis = zeichenkette1.lower().count("i")  
ergebnis
```

```
4
```

**count()** akzeptiert noch zwei weitere Argumente (Index von Start- und Endpunkt), mit denen wir eine "Scheibe" (*slice*) aus der zu untersuchenden Zeichenkette herausnehmen und getrennt betrachten können:

```
ergebnis = zeichenkette1.lower().count("e", 4, 8)  
ergebnis
```

```
1
```

## find()

Mit **find()** finden Sie den Index der Stelle, an der die gesuchte Zeichenkette zum ersten Mal erscheint. Wenn nichts gefunden wird, gibt **find()** -1 zurück. Wie **count()** unterscheidet auch **find()** zwischen Klein- und Großschreibung.

```
ergebnis = zeichenkette1.find("i")  
ergebnis
```

```
28
```

```
ergebnis = zeichenkette1.find("I")  
ergebnis
```

```
0
```

```
ergebnis = zeichenkette1.lower().find("i")
ergebnis
```

```
0
```

Auch können Sie wieder einen kleinen Teil der Zeichenkette auswählen:

```
ergebnis = zeichenkette1.lower().find("n", 15, 37)
ergebnis
```

```
36
```

Hier wurde der Teil "5 Programme mit Python" ausgewählt und Python fand das *n* am Ende.

Wenn Sie nur feststellen möchten, ob--und nicht wo--eine Zeichenkette vorkommt, sollten Sie stattdessen den Operator **in** verwenden. Das ist beispielsweise bei *if*-Bedingungen oft der Fall:

```
if "Python" in zeichenkette1:
    print("Gefunden!")
```

```
Gefunden!
```

## index()

Funktioniert ähnlich wie **find()**, gibt aber einen *ValueError* aus, wenn nichts gefunden wird.

```
ergebnis = zeichenkette1.index("n", 15, 37)
ergebnis
```

```
36
```

```
ergebnis = zeichenkette1.index("x", 15, 37)
ergebnis
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-96-174af8c223f6> in <module>
----> 1 ergebnis = zeichenkette1.index("x", 15, 37)
      2 ergebnis

ValueError: substring not found
```

## isalnum()

Sind alle Zeichen in der Zeichenkette alphanumerisch? Für unseren Beispielsatz wird uns *False* angezeigt, da dort auch Leer- und Satzzeichen vorkommen. **isalnum()** und verwandte Funktionen werden oft zur Eingabeverifikation verwendet.

```
zeichenkette1.isalnum()
```

```
False
```

```
zeichenkette3 = "password123"
zeichenkette3.isalnum()
```

True

## isalpha()

Sind alle Zeichen in der Zeichenkette alphabetisch?

```
zeichenkette1.isalpha()
```

False

```
zeichenkette3 = "passwort"  
zeichenkette3.isalpha()
```

True

## islower()

Sind alle Zeichen, die kleingeschrieben werden können, auch kleingeschrieben?

```
zeichenkette1.islower()
```

False

```
zeichenkette3 = "passwort"  
zeichenkette3.islower()
```

True

## isnumeric()

Sind alle Zeichen numerische Zeichen? Wenn Sie bspw. ein Programm schreiben, bei dem man Zahlen eingeben und dann Berechnungen durchführen kann, können Sie **isnumeric()** verwenden, um die Eingaben vor der Weiterverarbeitung zu überprüfen.

```
zeichenkette1.isnumeric()
```

False

```
zeichenkette3 = "12345"  
zeichenkette3.isnumeric()
```

True

## isupper()

Sind alle Zeichen, die großgeschrieben werden können, auch großgeschrieben?

```
zeichenkette1.isupper()
```

False

```
zeichenkette3 = "PASSWORT"
zeichenkette3.isupper()
```

True

## lower()

**lower()** wandelt alle Zeichen in ihre kleingeschriebenen Versionen um. Anders als bei **casefold()** wird ein *B* nicht in *ss* umgewandelt.

```
zeichenkette2 = zeichenkette1.lower()
zeichenkette2
```

'ich habe schon 5 programme mit python geschrieben und es hat mir spaß gemacht.'

## lstrip()

**lstrip()** entfernt Zeichen am Anfang der Zeichenkette. Wenn keine Zeichen an **lstrip()** übergeben werden, sind dies *whitespace characters*, also bspw. Leerzeichen (wie im folgenden Beispiel), Tabulatoren, usw. Wenn Sie ein oder mehr Zeichen angeben, wird jedes dieser Zeichen entfernt bis zum ersten Zeichen, das Sie nicht angegeben haben. Die Reihenfolge, in der Sie die Zeichen angegeben haben, spielt dabei keine Rolle.

```
zeichenkette3 = "        Das ist ein Satz."
zeichenkette3.lstrip()
```

'Das ist ein Satz.'

```
zeichenkette3.lstrip(" aDs")
```

'ist ein Satz.'

## replace()

Wenn eine Zeichenkette die im ersten Argument angegebenen Zeichen enthält, werden sie mit **replace()** durch die Zeichen im zweiten Argument ersetzt. Falls ein drittes Argument angegeben wird, kann man bestimmen, wie oft diese Zeichen ersetzt werden dürfen. Im ersten Beispiel werden alle "h" durch "x" ersetzt, im zweiten Beispiel betrifft das nur die ersten zwei "h".

```
zeichenkette2 = zeichenkette1.replace("h", "x")
zeichenkette2
```

'Icx xabe scxon 5 Programme mit Pytxon gescxriben und es xat mir Spaß gemacxt.'

```
zeichenkette2 = zeichenkette1.replace("h", "x", 2)
zeichenkette2
```

'Icx xabe schon 5 Programme mit Python geschrieben und es hat mir Spaß gemacht.'

## rstrip()

**rstrip()** ist das Gegenstück zu **lstrip()** und entfernt alle (vorgegebenen) Zeichen am Ende der Zeichenkette.

```
zeichenkette3 = "Das ist ein Satz.      "
zeichenkette3.rstrip()
```

```
'Das ist ein Satz.'
```

```
zeichenkette3.rstrip("tzas .")
```

```
'Das ist ein'
```

## split()

Mit **split()** können Sie eine Zeichenkette, etwa einen Satz, in eine Liste mit den einzelnen Bestandteilen (z.B. Wörter) aufteilen. Sie können selbst das/die Trennungszeichen bestimmen, an dem die Zeichenkette gespalten wird. Wenn kein Trennungszeichen angegeben wird, werden ein oder mehr aufeinanderfolgende Leerzeichen verwendet. Mit dem zusätzlichen Parameter *maxsplit* geben Sie an, wie oft die Zeichenkette geteilt werden soll.

```
zeichenkette3 = "Das ist ein      Satz."
zeichenkette3.split()
```

```
['Das', 'ist', 'ein', 'Satz.']}
```

```
zeichenkette3 = "58002345006748008"
zeichenkette3.split("00")
```

```
['58', '2345', '6748', '8']
```

```
zeichenkette1.split(" ", maxsplit=2)
```

```
['Ich',
'habe',
'schon 5 Programme mit Python geschrieben und es hat mir Spaß gemacht.']}
```

## splitlines()

Mit **splitlines()** können Sie längere Zeichenketten in einzelne Zeilen aufspalten, die dann in einer Liste gespeichert werden. Ähnlich wie bei **split()** definieren Sie ein oder mehrere Trennungszeichen. Dies sind u.a. die Zeichen, welche in verschiedenen Betriebssystemen als Zeilenenden verwendet werden:

Betriebssystem	Zeichen
Unix	\n
Windows	\r\n
macOS	\n
Mac (alte Versionen)	\r

Die Liste der möglichen Trennungszeichen finden Sie unter  
<https://docs.python.org/3/library/stdtypes.html#str.splitlines>.

Das folgende Beispiel ist sehr kurz, aber Sie können die Funktion auch für einen aus einer Datei geladenen Text verwenden.

```
a = "Das ist\r\nnein Satz.\nDas ist ein\nzweiter Satz."
a.splitlines()
```

```
['Das ist', 'ein Satz.', 'Das ist ein', 'zweiter Satz.']}
```

## strip()

**strip()** funktioniert wie eine Kombination aus **lstrip()** und **rstrip()**, d.h., es entfernt Zeichen am Anfang und Ende einer Zeichenkette.

```
zeichenkette3 = "      Das ist ein Satz.      "
zeichenkette3.strip()
```

```
'Das ist ein Satz.'
```

```
zeichenkette3 = "www.uni-tuebingen.de"
zeichenkette3.strip("ew.d")
```

```
'uni-tuebingen'
```

## swapcase()

**swapcase()** vertauscht Groß- und Kleinschreibung.

```
zeichenkette1.swapcase()
```

```
'iCH HABE SCHON 5 PROGRAMME MIT PYTHON GESCHRIEBEN UND ES HAT MIR SPASS GEMACHT.'
```

## title()

Mit **title()** (von *title case*) wird das erste Zeichen in jedem Wort groß-, der Rest kleingeschrieben.

```
zeichenkette1.title()
```

```
'Ich Habe Schon 5 Programme Mit Python Geschrieben Und Es Hat Mir Spaß Gemacht.'
```

## upper()

**upper()** verwandelt alle Zeichen in Großbuchstaben.

```
zeichenkette1.upper()
```

```
'ICH HABE SCHON 5 PROGRAMME MIT PYTHON GESCHRIEBEN UND ES HAT MIR SPASS GEMACHT.'
```

# Strings: Funktionen

Hier noch zwei Funktionen:

## len()

Die Funktion **len()** zeigt Ihnen, wie lange eine Zeichenkette ist, d.h., wie viele Zeichen sie enthält.

```
len(zeichenkette1)
```

78

## str()

**str()** wandelt Pythonobjekte in Zeichenketten um.

```
a = 42  
str(a)
```

'42'

## Strings: Index

Wie bei Listen kann man auch bei Zeichenketten über deren Index auf einzelne Bestandteile zugreifen: bei den Listen auf Listenelemente, bei Zeichenketten auf einzelne Zeichen.

Wenn man von vorne zählt, beginnt man bei 0 und zählt hoch. Wenn man von hinten zählt, beginnt man bei -1 und zählt herunter.

Index von vorne	0	1	2	3	4	5
Zeichenkette	P	y	t	h	o	n
Index von hinten	-6	-5	-4	-3	-2	-1

```
zeichenkette3 = "Python"  
zeichenkette3[0]
```

'P'

```
zeichenkette3[-1]
```

'n'

Weiterhin können wir auch mehrere Zeichen auswählen, indem wir einen Start- und Endpunkt angeben. Beim Endpunkt nennen wir tatsächlich den Index des Zeichens *nach* dem letzten Zeichen, das wir auswählen wollen.

```
zeichenkette3[2:4]
```

'th'

```
zeichenkette3[-4:-2]
```

```
'th'
```

Schließlich ist es auch möglich, eine Schrittweite (*step*) anzugeben und damit nicht jedes Element auszuwählen, sondern bspw. jedes zweite, jedes dritte, usw. Im folgenden Beispiel beginnen wir mit dem zweiten Zeichen (Indexwert 1), gehen bis zum Ende der Zeichenkette (Indexwert 6) und wählen dann jedes zweite Zeichen, also Buchstabe 2, 4 und 6.

```
zeichenkette3[1:6:2]
```

```
'yhn'
```

Da wir bis zum Ende der Zeichenkette gehen, können wir den zweiten Indexwert weglassen.

```
zeichenkette3[1::2]
```

```
'yhn'
```

Falls wir mit dem ersten Zeichen ("P") beginnen, kann auch der erste Wert wegfallen.

```
zeichenkette3[::2]
```

```
'Pto'
```

Auch bei der Schrittweite können wir negative Werte angeben. In diesem Fall wird die Reihenfolge der einzelnen Zeichen umgekehrt, d.h. die Zeichenkette beginnt mit dem ursprünglich letzten Buchstaben.

```
zeichenkette3[:::-1]
```

```
'nohtyP'
```

Man könnte hiermit bspw. leicht einen Palindromtester programmieren. Ein Palindrom ist ein Text, der vorwärts und rückwärts gleich geschrieben wird. Damit das funktioniert, müssen wir erst Leerzeichen entfernen und alle Buchstaben in Kleinbuchstaben umwandeln. Dann können wir testen, ob beide Versionen (vor- und rückwärtsgeschrieben) gleich sind. Gibt Python *True* aus, handelt es sich bei der Zeichenkette um ein Palindrom.

```
zeichenkette3 = "Erika feuert nur untreue Fakire".replace(" ", "").lower()
zeichenkette3 == zeichenkette3[::-1]
```

```
True
```

Man kann das auch noch erweitern. Wenn Sie die Materialien zu Kontrollstrukturen durchgearbeitet haben, versuchen Sie mal, noch Satzzeichen aus der Zeichenkette zu entfernen, ohne nach jedem Satzzeichen einzeln zu suchen.

## Literatur

- Offizielle Dokumentation: <https://docs.python.org/3/tutorial/inputoutput.html>
- Format Specification Mini-Language: <https://docs.python.org/3/library/string.html#format-specification-mini-language>

- Python F-Strings Number Formatting Cheat Sheet: <https://cheatography.com/brianallan/cheat-sheets/python-f-strings-number-formatting/>
- Python String Methods: <https://docs.python.org/3/library/stdtypes.html#string-methods>

# Datenstruktur Menge

## Zurück zu Datenstrukturen: Mengen

Mengen, oder *sets*, haben die Eigenschaft, dass in ihnen jedes Element nur einmal vorkommen darf. Es gibt zwei verschiedene Methoden, um *sets* zu erzeugen:

- **set()** erzeugt eine mutable Menge, d.h., eine Menge, die nachträglich verändert werden kann. Hier wird ein neues *set* mit Werten aus einem *tuple* befüllt, daher die doppelten runden Klammern.

```
a = set(('a', 'b', 'c'))  
a
```

```
{'a', 'b', 'c'}
```

## Datenstrukturen: Mengen

- **frozenset()** erzeugt eine immutable Menge, d.h., sie kann nachträglich nicht mehr verändert werden.

```
a = frozenset(('a', 'b', 'c'))  
a
```

```
frozenset({'a', 'b', 'c'})
```

## Datenstrukturen: Mengen

Was passiert, wenn Sie versuchen, mehrmals den gleichen Wert in ein *set* zu packen?

```
a = (1, 2, 2, 3, 4, 5, 5, 5, 5)  
b = set(a)  
b
```

```
{1, 2, 3, 4, 5}
```

Jeder Wert, der mehrfach vorhanden ist, wird auf eine Erwähnung reduziert.

## Datenstrukturen: Mengen

Ein anderes Experiment: Was passiert, wenn Sie versuchen, einmal ein Tupel und dann eine Liste in ein Set einzufügen? Ein Beispiel:

```
a = set((1, (3, 4))) # Zweites Element ist ein Tupel  
a
```

```
{(3, 4), 1}
```

```
a = set((1, [3, 4])) # Zweites Element ist eine Liste  
a
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-39-dcc7e88513a8> in <module>  
----> 1 a = set((1, [3, 4])) # Zweites Element ist eine Liste  
      2 a  
  
TypeError: unhashable type: 'list'
```

## Datenstrukturen: Mengen

Der Grund für die Fehlermeldung ist, dass *sets* keine mutablen Elemente, wie etwa Listen, enthalten können; man könnte sonst nicht sicherstellen, dass es später zu keinen duplizierten Werten kommt. *Tuples* verursachen keine Probleme, da sie immutabel sind.

Dafür sind *sets* selbst aber mutabel, wenn sie mit **set()** und nicht mit **frozenset()** erzeugt wurden.

```
a = set((1, 2, 3, 4))  
a.add(5)  
a
```

```
{1, 2, 3, 4, 5}
```

## Operatoren und Methoden für Mengen

Operator	Beispiel	Erklärung	Werte	Äquivalente Methode
in	x in a	Ist Wert x in Menge a?	True/False	
not in	x not in a	Ist Wert x nicht in Menge a?	True/False	
<=	a <= b	Ist a eine Teilmenge von b?	True/False	a.issubset(b)
<	a < b	Ist a eine echte Teilmenge von b, d.h., es gibt mindestens ein Element in b, das nicht in a ist	True/False	
>=	a >= b	Ist b eine Teilmenge von a?	True/False	a.issuperset(b)
>	a > b	Ist b eine echte Teilmenge von a?	True/False	

## Operatoren und Methoden für Mengen

Beispiel:

```
a = set([1, 2, 3, 4, 5, 6])  
b = 5  
if b in a:  
    print("b ist in a")
```

```
b ist in a
```

# Operatoren und Methoden für Mengen

Operator	Beispiel	Erklärung	
	a   b	Neue Menge, die alle Werte von a und b enthält	a.union(b)
&	a & b	Schnittmenge von a und b	a.intersection(b)
-	a - b	Menge mit allen Elementen aus a, außer denen, die auch in b sind	a.difference(b)
^	a ^ b	Menge mit Elementen, die entweder in a oder b sind, aber nicht in beiden	a.symmetric_difference(b)

# Operatoren und Methoden für Mengen

Beispiel:

```
a = set([1, 2, 3, 4, 5, 6])
b = set([5, 6, 7, 8, 9, 10])
c = a&b
c
```

```
{5, 6}
```

# Kontrollstrukturen

## Exkurs: Kontrollstrukturen

Bevor wir mit der nächsten Datenstruktur weitermachen, sollten wir über **Kontrollstrukturen** (*control structures*) sprechen.

Kontrollstrukturen bestimmen den Ablauf eines Programms. **Fallunterscheidungen** (*conditional statements*) führen abhängig davon, ob eine oder mehrere Bedingungen erfüllt sind oder nicht, unterschiedlichen Code aus. **Schleifen** (*loops*) führen den gleichen Code immer wieder aus.

Schleifen und Fallunterscheidungen können auch miteinander kombiniert werden.

## Kontrollstrukturen: Fallunterscheidung mit *if*

**if** testet, ob eine Bedingung erfüllt ist. Wenn ja, wird der unter **if** eingerückte Code ausgeführt. Wenn die Bedingung nicht erfüllt ist, geht Python zur nächsten nicht eingerückten Zeile weiter:

```
a = 1
if a == 1:
    print("a hat den Wert 1")
print("Hier gehts weiter!")
```

```
a hat den Wert 1
Hier gehts weiter!
```

```
a = 2
if a == 1:
    print("a hat den Wert 1")
print("Hier gehts weiter!")
```

```
Hier gehts weiter!
```

## Exkurs im Exkurs: Vergleichsoperatoren

Sie haben eben schon einen von Pythons Vergleichsoperatoren gesehen, das doppelte Gleichheitszeichen (==). Damit können Sie testen, ob die Werte auf beiden Seiten identisch sind. Hier ist eine Liste mit allen Vergleichsoperatoren:

Operator	Beispiel	Erklärung
==	1 == 2	Ist 1 gleich 2?
!=	1 != 2	Ist 1 ungleich 2?
<	1 < 2	Ist 1 kleiner als 2?
>	1 > 2	Ist 1 größer als 2?
<=	1 <= 2	Ist 1 kleiner oder gleich 2?

Operator	Beispiel	Erklärung
<code>&gt;=</code>	<code>1 &gt;= 2</code>	Ist 1 größer oder gleich 2?

## Exkurs im Exkurs: Logische Operatoren

Durch Nutzung mehrerer logischer Operatoren können Sie zwei oder mehr Vergleiche kombinieren:

and: `(a < b) and (c < d)` -> Ist 1 kleiner als 2 **und** gleichzeitig 3 kleiner als 4? (Beide müssen wahr sein.)

or: `(a < b) or (c < d)` -> Ist 1 kleiner als 2 **oder** 3 kleiner als 4? (Nur eines muss wahr sein.)

Mit **not** können Sie einen Vergleich umkehren: `not (a < b)` ist das gleiche wie `a >= b`

## Zurück zu Kontrollstrukturen und Fallentscheidungen

Jetzt wissen wir, wie wir mit Vergleichs- und logischen Operatoren mehrere Bedingungen in einer if-Anweisung testen können:

```
a = 7

if a > 6 and a < 10:
    print("a ist größer als 6 und kleiner als 10")
```

a ist größer als 6 und kleiner als 10

## Kontrollstrukturen: Fallunterscheidung mit *if/elif*

Oft will man aber nicht nur eine Bedingung testen. Wenn Sie eine zweite Bedingung hinzufügen möchten, dann können Sie das mit **elif** tun:

```
a = 5

if a > 6 and a < 10:
    print("a ist größer als 6 und kleiner als 10")
elif a > 4 and a <= 6:
    print("a ist größer als 4 und kleiner als oder gleich 6")
```

a ist größer als 4 und kleiner als oder gleich 6

## Kontrollstrukturen: Fallunterscheidung mit *if/elif*

Sie können beliebig viele Bedingungen mit **elif** anfügen, solange Sie am Anfang eine einzelne **if**-Bedingung stehen haben:

```
a = 1

if a > 6 and a < 10:
    print("a ist größer als 6 und kleiner als 10")
elif a > 4 and a <= 6:
    print("a ist größer als 4 und kleiner als oder gleich 6")
elif a > 2 and a <= 4:
```

```

    print("a ist größer als 2 und kleiner als oder gleich 4")
elif a > 0 and a <= 2:
    print("a ist größer als 0 und kleiner als oder gleich 2")

```

a ist größer als 0 und kleiner als oder gleich 2

## Kontrollstrukturen: Fallunterscheidung mit *if/elif/else*

Schließlich können Sie am Schluß noch mit **else** alle anderen Fälle abfangen:

```

a = -1
if a > 6 and a <= 10:
    print("a ist größer als 6 und kleiner als oder gleich 10")
elif a > 4 and a <= 6:
    print("a ist größer als 4 und kleiner als oder gleich 6")
elif a > 2 and a <= 4:
    print("a ist größer als 2 und kleiner als oder gleich 4")
elif a >= 0 and a <= 2:
    print("a ist größer als oder gleich 0 und kleiner als oder gleich 2")
else:
    print("Ihre Zahl liegt nicht zwischen 0 und 10")

```

Ihre Zahl liegt nicht zwischen 0 und 10

## Kontrollstrukturen: Fallunterscheidung mit *if/elif/else*

Ändern Sie jetzt einmal den Wert von "a" zu einer Zeichenkette (bspw. "abc"). Was geschieht?

```

a = "abc"
if a > 6 and a <= 10:
    print("a ist größer als 6 und kleiner als oder gleich 10")
elif a > 4 and a <= 6:
    print("a ist größer als 4 und kleiner als oder gleich 6")
elif a > 2 and a <= 4:
    print("a ist größer als 2 und kleiner als oder gleich 4")
elif a >= 0 and a <= 2:
    print("a ist größer als oder gleich 0 und kleiner als oder gleich 2")
else:
    print("Ihre Zahl liegt nicht zwischen 0 und 10")

```

```

-----
TypeError                                                 Traceback (most recent call last)
<ipython-input-16-86f8b407a51f> in <module>
      1 a = "abc"
----> 2 if a > 6 and a <= 10:
      3     print("a ist größer als 6 und kleiner als oder gleich 10")
      4 elif a > 4 and a <= 6:
      5     print("a ist größer als 4 und kleiner als oder gleich 6")

TypeError: '>' not supported between instances of 'str' and 'int'

```

## Kontrollstrukturen: Fallunterscheidung mit *if/elif/else*

Sie bekommen eine Fehlermeldung, da Sie versuchen, den Vergleichsoperator mit zwei verschiedenen Datentypen zu verwenden. Hier macht es Sinn, dass man zuerst testet, ob der

gewählte Wert eine Zahl ist. Wie Sie im folgenden Beispiel sehen können, kann man **if/elif/else** ineinander verschachteln (*nesting*). Wichtig dabei ist, dass für jede Einrückungsstufe die gleiche Anzahl von Leerzeichen verwendet wird (normalerweise 4), was aber normalerweise Ihr Editor für Sie übernimmt.

```
a = 4
if isinstance(a, (int, float)) == True:
    if a > 6 and a <= 10:
        print("Ihre Zahl ist größer als 6 und kleiner als oder gleich 10")
    elif a > 4 and a <= 6:
        print("Ihre Zahl ist größer als 4 und kleiner als oder gleich 6")
    elif a > 2 and a <= 4:
        print("Ihre Zahl ist größer als 2 und kleiner als oder gleich 4")
    elif a >= 0 and a <= 2:
        print("Ihre Zahl ist größer als oder gleich 0 und kleiner als oder
gleich 2")
    else:
        print("Ihre Zahl liegt nicht zwischen 0 und 10")
else:
    print("Bitte wählen Sie eine Zahl als Wert")
```

Ihre Zahl ist größer als 2 und kleiner als oder gleich 4

```
a = "abc"
if isinstance(a, (int, float)) == True:
    if a > 6 and a <= 10:
        print("Ihre Zahl ist größer als 6 und kleiner als oder gleich 10")
    elif a > 4 and a <= 6:
        print("Ihre Zahl ist größer als 4 und kleiner als oder gleich 6")
    elif a > 2 and a <= 4:
        print("Ihre Zahl ist größer als 2 und kleiner als oder gleich 4")
    elif a >= 0 and a <= 2:
        print("Ihre Zahl ist größer als oder gleich 0 und kleiner als oder
gleich 2")
    else:
        print("Ihre Zahl liegt nicht zwischen 0 und 10")
else:
    print("Bitte wählen Sie eine Zahl als Wert")
```

Bitte wählen Sie eine Zahl als Wert

## Fallunterscheidungen mit Bedingten Ausdrücken

Mit einem bedingten Ausdruck (*conditional expression*) können Sie eine **if/else**-Konstruktion (mit einer Bedingung) kürzen. Die beiden Beispiele geben Ihnen das gleiche Ergebnis:

```
a = 4
print("Zahl ist gerade" if a % 2 == 0 else "Zahl ist ungerade")
```

Zahl ist gerade

```
a = 1
if a % 2 == 0:
    print("Zahl ist gerade")
```

```
else:  
    print("Zahl ist ungerade")
```

Zahl ist ungerade

## Kontrollstrukturen: Schleifen

Mit Schleifen können Sie den selben Codeblock mehrmals ausführen lassen. Python bietet Ihnen zwei Wege, um Schleifen zu erzeugen: mit **while** und **for**.

Wenn Sie mit **while** arbeiten, legen Sie eine Bedingung fest; solange die Bedingung erfüllt ist, läuft die Schleife weiter. Erst wenn die Bedingung nicht mehr erfüllt ist, wird die Schleife beendet:

```
a = 1  
while a < 5:  
    print(a)  
    a = a + 1 # Ohne diese Zeile läuft die Schleife endlos weiter  
              # da a immer kleiner als 5 bleibt  
print("Fertig!")
```

1  
2  
3  
4  
Fertig!

## Kontrollstrukturen: Schleifen mit *while*

Sie können Schleifen mit **break** auch vorzeitig beenden:

```
a = 1  
while a < 5:  
    print(a)  
    if a == 3:  
        break  
    a = a + 1  
print("Fertig!")
```

1  
2  
3  
Fertig!

## Kontrollstrukturen: Schleifen mit *for*

Wenn Sie eine Schleife mit **for** bilden wollen, benötigen Sie ein iterierbares (*iterable*) Objekt, wie etwa eine Liste oder ein Tupel. Für jedes Element in diesem Objekt läuft die Schleife einmal. Wenn die Schleife für jedes Element durchgelaufen ist, endet sie.

```
a = ["banane", "kirsche", "apfel", "ananas"]  
for x in a:  
    print(x)
```

banane

```
kirsche  
apfel  
ananas
```

## Kontrollstrukturen: Schleifen mit *for*

Sie können genauso gut mit Zahlensequenzen arbeiten. Hier ein Beispiel einer mit **range()** erzeugten Sequenz aller geraden Zahlen von 20 bis 2:

```
a = range(20, 1, -2)  
for x in a:  
    print(x)
```

```
20  
18  
16  
14  
12  
10  
8  
6  
4  
2
```

## Kontrollstrukturen: Schleifen mit *for*

Wie bei **while**-Schleifen können Sie auch bei **for** Bedingungen einbauen:

```
a = ["banane", "kirsche", "apfel", "ananas"]  
for x in a:  
    if x == "apfel":  
        print(x.upper())  
    else:  
        print(x)
```

```
banane  
kirsche  
APFEL  
ananas
```

## Kontrollstrukturen: *pass*

Wenn Sie an einem Programm arbeiten, gibt es möglicherweise Stellen, an denen Sie vorerst nur einen Platzhalter benötigen. Dafür können Sie **pass** verwenden. Die Fallunterscheidung oder Schleife wird dann einfach ignoriert:

```
a = ["banane", "kirsche", "apfel", "ananas"]  
for x in a:  
    if x == "apfel":  
        print(x.upper())  
    elif x == "banane":  
        pass  
    else:  
        print(x)
```

kirsche

APFEL

ananas

# Datenstruktur Dictionary

## Datenstrukturen: Dictionary

Ein Dictionary ist eine Datenstruktur, in der Schlüssel-Wert-Paare (*key-value pairs*) gespeichert werden.

Dabei muss jeder Schlüssel einmalig sein, die Werte können sich aber wiederholen. Auch muss der Schlüssel durch einen immutablen Datentypen ausgedrückt werden, z.B. durch eine Zeichenkette, Integer, usw. Dictionaries werden in geschweifte Klammern gesetzt. Sie können das Dictionary in eine Zeile schreiben oder für jeden Eintrag eine neue Zeile beginnen, was bei längeren oder verschachtelten Dictionaries übersichtlicher ist.

```
a = {  
    "Klaus": 31,  
    "Stefanie": 64,  
    "Beate": 11,  
    "Martin": 71  
}  
a
```

```
{'Klaus': 31, 'Stefanie': 64, 'Beate': 11, 'Martin': 71}
```

## Dictionaries

Ein Dictionary kann auch selbst wieder Dictionaries beinhalten. Wenn Sie bspw. für jede Person nicht nur das Alter, sondern auch den Wohnort speichern möchten, können Sie für jede Person ein Dictionary erzeugen, dort Alter und Wohnort speichern und das individuelle Dictionary als Wert mit dem Namen der Person als Schlüssel in einem übergeordneten Dictionary verwenden, wie im folgenden Beispiel:

```
b = {  
    "Klaus": {  
        "alter": 31,  
        "wohnort": "Tübingen"  
    },  
    "Stefanie": {  
        "alter": 64,  
        "wohnort": "Rottenburg"  
    },  
    "Beate": {  
        "alter": 11,  
        "wohnort": "Reutlingen"  
    }  
}  
b
```

```
{'Klaus': {'alter': 31, 'wohnort': 'Tübingen'},  
 'Stefanie': {'alter': 64, 'wohnort': 'Rottenburg'},
```

```
'Beate': {'alter': 11, 'wohnort': 'Reutlingen'}}}
```

## Dictionaries: Elemente auswählen

Anders als etwa bei einer Liste oder einem Tupel verwenden Sie bei Dictionaries keine Indexwerte, sondern die Schlüssel, um auf einzelne Elemente zuzugreifen:

```
b["Stefanie"]
```

```
{'alter': 64, 'wohnort': 'Rottenburg'}
```

```
b["Stefanie"]["wohnort"]
```

```
'Rottenburg'
```

## Dictionaries: Schleifen

Da Dictionaries iterabel sind, können sie auch in Schleifen (normalerweise **for**-Schleifen) eingesetzt werden. Das folgende Beispiel verwendet das einfachere Dictionary, das Sie zu Beginn gesehen haben und gibt für jedes Element den Schlüssel aus:

```
for x in a:  
    print(x)
```

```
Klaus  
Stefanie  
Beate  
Martin
```

## Dictionaries: Schleifen

Sie können sich auch den Wert des Elements, oder Wert und Schlüssel gleichzeitig ausgeben lassen:

```
for x in a:  
    print(a[x])
```

```
31  
64  
11  
71
```

```
for x in a:  
    print("{} ist {} Jahre alt.".format(x, a[x]))
```

```
Klaus ist 31 Jahre alt.  
Stefanie ist 64 Jahre alt.  
Beate ist 11 Jahre alt.  
Martin ist 71 Jahre alt.
```

## Dictionary: Funktionen

Es gibt verschiedene Funktionen, welche die Arbeit mit Dictionaries erleichtern. Im folgenden Beispiel wird **items()** verwendet; Sie können hiermit sowohl Schlüsseln als auch Werten Variablenamen zuweisen:

```
for name, alter in a.items():
    print("{} ist {} Jahre alt.".format(name, alter))
```

```
Klaus ist 31 Jahre alt.
Stefanie ist 64 Jahre alt.
Beate ist 11 Jahre alt.
Martin ist 71 Jahre alt.
```

## Dictionary: Verschachtelt in Schleifen

Um mit verschachtelten Dictionaries zu arbeiten, können Sie verschachtelte Schleifen verwenden.

Zur Erinnerung noch einmal das Dictionary:

```
b = {
    "Klaus": {
        "alter": 31,
        "wohnort": "Tübingen"
    },
    "Stefanie": {
        "alter": 64,
        "wohnort": "Rottenburg"
    },
    "Beate": {
        "alter": 11,
        "wohnort": "Reutlingen"
    }
}
b
```

```
{'Klaus': {'alter': 31, 'wohnort': 'Tübingen'},
 'Stefanie': {'alter': 64, 'wohnort': 'Rottenburg'},
 'Beate': {'alter': 11, 'wohnort': 'Reutlingen'}}
```

## Dictionary: Verschachtelt in Schleifen

Und hier die verschachtelte Schleife:

```
for k, v in b.items():
    print(k)
    for x, y in v.items():
        print(y)
```

```
Klaus
31
Tübingen
Stefanie
64
Rottenburg
Beate
11
Reutlingen
```

## Dictionary

Im obigen Beispiel können Sie auch in der ersten Schleife über die Schlüssel auf individuelle Werte zugreifen:

```
for k, v in b.items():
    print("{} wohnt in {} und ist {} Jahre alt.".format(k, v["wohnort"],
v["alter"]))
```

Klaus wohnt in Tübingen und ist 31 Jahre alt.  
Stefanie wohnt in Rottenburg und ist 64 Jahre alt.  
Beate wohnt in Reutlingen und ist 11 Jahre alt.

# Module und Pakete

## Module und Pakete installieren und importieren

Python ist u.a. deshalb so beliebt, da es auf eine sehr große Bibliothek von Modulen und Paketen zugreifen kann, mit denen sich seine Funktionalität enorm ausbauen lässt. Module sind einzelne Pythondateien, während Pakete zumeist mehrere Module enthalten. Um diese Erweiterungen verwenden zu können, müssen sie--falls sie nicht Teil der Standardinstallation sind--aber erst installiert und dann in allen Fällen importiert werden.

Nehmen wir als Beispiel das Paket *pandas*, welches wir in einem späteren Video noch für die Analyse tabellarischer Daten verwenden werden und das separat installiert werden muss. Für diese Installation können wir *pip* verwenden. *pip* ist ein Paketverwaltungsprogramm und wir benötigen es zum Installieren, Verwalten und Entfernen von Paketen und Modulen in Python. Es wird meist zusammen mit Python auf Ihrem Computer installiert.

Um *pandas* zu installieren, öffnen Sie auf Ihrem Computer das Terminal (macOS/Linux) oder die Eingabeaufforderung/PowerShell (Windows). Geben Sie den folgenden Befehl ein und drücken danach die Eingabetaste:

```
pip install pandas
```

*pip* sollte jetzt den Installationsprozess starten. Im Fall von *pandas* kann das durchaus etwas länger dauern, da neben *pandas* noch mehrere andere Pakete installiert werden müssen, die *pandas* benötigt, um funktionieren zu können. Diese werden als *dependencies* bezeichnet. *pip* benachrichtigt Sie dann, wenn der Installationsprozess entweder abgeschlossen oder fehlgeschlagen ist.

Jetzt können Sie *pandas* mit dem Schlüsselwort **import** in Ihr Programm importieren. Um Ihren Code übersichtlich zu gestalten, sollten Sie diese Importe immer gleich zu Beginn durchführen:

```
import pandas
```

Wenn Sie Module/Pakete auf diese Weise importieren, wird ein sog. *Namensraum (namespace)* geschaffen. Das verhindert, dass ein Element aus dem importierten Modul oder Paket, welches den gleichen Namen wie ein bereits existierendes Element hat, diese überschreibt. Deshalb müssen Sie dann vor den Namen der importierten Elemente noch den Namen des Moduls/Pakets setzen:

```
a = pandas.DataFrame()
```

Es ist auch möglich, ein Modul/Paket oder Teile davon unter einem anderen Namen zu importieren, etwa wenn der eigentliche Name sehr lang ist oder oft erwähnt wird. Sie können diese Namen selbst wählen, solange sie sich von denen anderer installierter Pakete unterscheiden. Manchmal haben sich aber im Laufe der Zeit quasi standardisierte Alternativnamen herausentwickelt. Dies ist etwa bei *pandas* der Fall, das normalerweise als *pd* importiert wird:

```
import pandas as pd
```

Es gibt in Python auch einen globalen Namensraum (*global namespace*). Den verwenden die Funktionen, die Sie bisher verwendet haben. Sie können Module/Pakete auch teilweise oder komplett in diesen globalen Namensraum einbinden:

```
from pandas import *
```

```
from pandas import DataFrame
```

```
from pandas import DataFrame as datafr
```

Sie müssen bei dieser Methode den Namen des Moduls/Pakets nicht mehr extra nennen:

```
from pandas import *
a = DataFrame()
```

```
from pandas import DataFrame
a = DataFrame()
```

```
from pandas import DataFrame as datafr
a = datafr()
```

Es gibt allerdings einen entscheidenden Nachteil:

Wenn im globalen Namensraum bereit ein Element mit diesem Namen existiert, dann wird es mit dem importierten Element überschrieben und das Original steht nicht mehr zur Verfügung. Solche Fehler sind dann oft schwer aufzufinden. Besonders problematisch ist das im ersten Beispiel:

```
from pandas import *
a = DataFrame()
```

Hier wird das gesamte Paket in den globalen Namensraum kopiert und man kann keine Elemente ausschließen, die möglicherweise Konflikte verursachen könnten.

# Jupyter Notebook

## Jupyter Notebook

Jupyter Notebook bezeichnet sowohl eine Computeranwendung als auch ein Dokumentenformat, die sich in den letzten Jahren zu beliebten Werkzeugen, besonders für den Einsatz von Python in der Wissenschaft, entwickelt haben.

Wenn wir über das Dokumentenformat sprechen, bezeichnet Jupyter Notebook eine Datei, die in einem interaktiven Dokument formatierten Text, Computercode und Visualisierungen enthalten kann. Diese Notebooks können nicht nur mit der Anwendung Jupyter Notebook erstellt werden, sondern auch mit Editoren wie MS Visual Studio Code--der Notebooks über eine Erweiterung unterstützt--und Onlinediensten wie Google Colaboratory (<https://colab.research.google.com>).

Notebooks können auch Code in anderen Sprachen beinhalten (der Name setzt sich aus den drei Programmiersprachen **Julia**, **Python** und **R** zusammen, andere Sprachen sind auch möglich), sie lassen sich leicht mit anderen teilen und ermutigen zum Experimentieren.

## Jupyter Notebook

Das Computerprogramm Jupyter Notebook läuft, wie auch sein bereits erhältlicher Nachfolger JupyterLab, als eine Anwendung in Ihrem Browser. In diesem Fall werden aber keine Daten mit einem externen Server ausgetauscht. Stattdessen startet Jupyter Notebook zu Beginn jeder Sitzung einen Server auf Ihrem Computer: daher auch das Befehlszeilenfenster, das Sie sehen, kurz bevor der Browser geöffnet wird und das Sie nicht schließen dürfen. Alle Operationen finden also auf Ihrem Computer statt!

## Was ist ein Notebook?

Ein Notebook besteht aus einzelnen Zellen (oder *cells*), die ähnlich einer Folie in einer Präsentation funktionieren. Zellen dienen dazu, das Notebook zu strukturieren und erlauben es Ihnen, entweder das ganze Notebook oder nur Teile ausführen zu lassen. Sie können nachträglich auch leicht verschoben werden. Für jede Zelle können Sie einen von drei Inhaltstypen wählen:

- Code: Python-Code, der in der Zelle ausgeführt werden kann.
- Markdown: Eine einfache Markup-Sprache, mit der man leicht Text formatieren kann. Kann auch auf vielen Webplattformen wie Wordpress verwendet werden. Diese Folien wurden in Markdown geschrieben.
- Raw NBConvert: Damit können Zellen beim Export in bestimmte Formate unterdrückt werden.

## Jupyter Notebook installieren

Sie können Jupyter Notebook wie ein Python-Paket installieren, und zwar mit dem Befehl:

```
pip install notebook
```

Der Prozess wird wahrscheinlich länger als bei anderen Paketen laufen, da viele Zusatzpakete installiert werden müssen.

Wenn die Installation abgeschlossen ist, können Sie folgenden Befehl verwenden, um Jupyter Notebook zu starten:

```
jupyter notebook
```

## Jupyter Notebook Extensions installieren

Jupyter Notebook Extensions bieten viele zusätzliche Funktionen für Jupyter Notebook, die von einfachen Ergänzungen wie etwa Zeilennummern bis hin zur Verwendung von Notebooks für Präsentationen reichen.

Zur Installation benötigen Sie zwei Befehle:

```
pip install jupyter_contrib_nbextensions
```

```
jupyter contrib nbextension install --user
```

Nach der Installation können Sie die einzelnen Erweiterungen auf der Startseite von Jupyter Notebook unter dem Reiter *Nbextensions* aktivieren. Dort finden Sie auch Beschreibungen für jede Erweiterung.

# Markdown

## Markdown

Markdown ist eine Auszeichnungssprache (**markup language**)--wie etwa auch HTML--die es erlaubt, mit sehr wenig Aufwand Dokumente zu strukturieren und sie dann mit der passenden Software in verschiedene Formate umzuwandeln.

Mit Jupyter Notebook können Sie solche Markdown-Dokumente erstellen und in eine Vielzahl von Formaten exportieren (etwa PDF, Webpräsentationen, LaTeX für professionellen Drucksatz und viele mehr).

## Markdown: Formatierung

Syntax	Darstellung
*kursiv*, _kursiv_	<i>kursiv, kursiv</i>
**fett**, __fett__	<b>fett, fett</b>
~~durchgestrichen~~	<del>durchgestrichen</del>

## Horizontale Linie

Mit \*\*\* fügen Sie eine horizontale Linie ein:

---

## Überschriften

Sie erstellen eine Überschrift, indem Sie dem Text ein bis sechs # voranstellen. Die wichtigsten Überschriften haben nur ein #; je weiter sie untergeordnet sind, desto mehr # haben sie.

## # Überschrift 1

## ## Überschrift 2

## ### Überschrift 3

## #### Überschrift 4

## ##### Überschrift 5

## ###### Überschrift 6

## Formeln

Sie können eine Formel in den Text integrieren, indem Sie jeweils ein \\$ an den Anfang und das Ende der Formel setzen:

Syntax: \$A = \pi \* r^2\$

Darstellung: \$A = \pi \* r^2\$

## Links

Links können folgendermaßen eingefügt werden:

[Link zu Python Seite](https://www.python.org/)

Der Text in eckigen Klammern wird dann im Dokument dargestellt und kann angeklickt werden. In den runden Klammern steht die URL:

[Link zu Python Seite](https://www.python.org/)

## Bilder

Markdown für Bilder funktioniert fast so wie bei Links, nur dass Sie vorher noch ein Ausrufezeichen benötigen:

![Optionale Bildunterschrift](schlange.jpg)



## Listen

Markdown bietet Ihnen zwei verschiedene Arten von Listen (geordnet und ungeordnet), die Sie auch schon von Textverarbeitungsprogrammen kennen.

## Geordnete Listen: Markup

Bei geordneten Listen müssen Sie die Nummerierung nicht selbst vornehmen. Sie können immer die gleiche Zahl eingeben; solange danach ein Punkt folgt, erfolgt bei der Konvertierung in das Zielformat die korrekte Nummerierung. Bei den Einrückungen müssen Sie darauf achten, dass Sie immer die gleiche Anzahl von Leerzeichen verwenden.

1. Nummer 1
1. Nummer 2
  1. Nummer 2.1
  1. Nummer 2.2
    1. Nummer 2.2.1
    1. Nummer 2.3
1. Nummer 3

## Geordnete Listen: Darstellung

1. Nummer 1
2. Nummer 2
  - A. Nummer 2.1
  - B. Nummer 2.2
    - a. Nummer 2.2.1
  - C. Nummer 2.3
3. Nummer 3

## Ungeordnete Listen: Markup

Bei ungeordneten Listen können Sie das gleiche Zeichen verwenden (Minus funktioniert gut), solange Sie immer die gleiche Anzahl von Leerzeichen pro Stufe verwenden.

- Stufe 1
  - Stufe 2
    - Stufe 3
  - Stufe 2
- Stufe 1

## Ungeordnete Listen: Darstellung

- Stufe 1
  - Stufe 2
    - Stufe 3
  - Stufe 2
- Stufe 1

## Tabellen

Tabellen lassen sich mit Markdown sehr leicht erstellen. Sie müssen dabei vor allem 2 Sachen beachten:

- Spalten werden mit | getrennt. Um den Markdowntext lesbarer zu machen, können Sie zwischen Text und | auch Leerzeichen einfügen.
- Nach der ersten Reihe mit den Spaltennamen müssen Sie eine Zeile einfügen, in der für jede Spalte mindestens ein Minus steht. Diese werden dann auch wieder mit | getrennt.

## Tabellen: Markup

```
|Name Spalte 1|Name Spalte 2|Name Spalte 3|
|-|-|-|
|Reihe 1, Spalte 1|Reihe 1, Spalte 2|Reihe 1, Spalte 3|
|Reihe 2, Spalte 1|Reihe 2, Spalte 2|Reihe 2, Spalte 3|
```

## Tabellen: Darstellung

Name Spalte 1	Name Spalte 2	Name Spalte 3
Reihe 1, Spalte 1	Reihe 1, Spalte 2	Reihe 1, Spalte 3
Reihe 2, Spalte 1	Reihe 2, Spalte 2	Reihe 2, Spalte 3

## Tabellen: Linksbündig, zentriert, rechtsbündig

Wenn Sie in der zweiten Zeile Doppelpunkte einfügen, können Sie auch bestimmen, ob der Text in einer Tabelle linksbündig (vor dem Minus), zentriert (auf beiden Seiten) oder rechtsbündig (hinter dem Minus) dargestellt werden soll.

## Tabellen: Markup

```
| Linksbündiger Text | Zentrierter Text | Rechtsbündiger Text |
| :- | :- | -: |
| abc | abc | abc |
| abc | abc | abc |
```

## Tabellen: Darstellung

Linksbündiger Text	Zentrierter Text	Rechtsbündiger Text
abc	abc	abc
abc	abc	abc

## Codebeispiele im Text

Wenn Sie in den laufenden Text Codebeispiele integrieren wollen--etwa um zu demonstrieren wie man die **sum()**-Funktion verwendet `a = sum(1,2,3,4)`--können Sie das folgendermaßen tun:

```
`a = sum(1,2,3,4)`
```

Beachten Sie, dass der Code nicht in einfache Anführungszeichen gefasst ist, sondern in Backticks / Backquotes (oder im Deutschen "rückwärts geneigtes Hochkomma", auf der Tastatur rechts neben dem Fragezeichen).

## Codebeispiele

Besonders für längere Codebeispiele können Sie diese in drei Backticks einschließen. Wenn Sie die Sprache spezifizieren, können Sie auch Extrafunktionen wie etwa Syntaxhervorhebung nutzen:

```
```python
def f(x):
    """a docstring"""
    return x**2
```
```

```
def f(x):
    """a docstring"""
    return x**2
```

# Funktionen

## Funktionen

Sie haben bereits mit vordefinierten Funktionen in Python gearbeitet:

```
a = [1, 2, 3, 4, 5]
sum(a)
```

15

```
len(a)
```

5

```
min(a)
```

1

Python erlaubt Ihnen aber auch, selbst Funktionen zu schreiben und diese dann in Ihren Programmen anzuwenden. Sie müssen dabei vor allem darauf achten, dass Sie eine selbstgeschriebene Funktion erst nützen können, nachdem Sie diese definiert haben.

Es bietet sich deshalb an, Funktionsdefinitionen entweder an den Anfang eines Programms zu setzen oder sie gleich in eine separate .py-Datei zu schreiben und diese dann am Anfang des Programms wie ein Paket zu importieren. Wenn die Datei, in der Sie Ihre eigenen Funktionen speichern, etwa *funktionen.py* heißt, importieren Sie diese mit `import funktionen`.

## Nutzung von Funktionen

Funktionen sind sehr nützlich, um Wiederholungen im Code zu vermeiden. Anstatt bestimmte Stellen mehrmals zu wiederholen, ist es oft einfacher und effizienter, Code in eine Funktion auszulagern. Um eine Funktion möglichst vielseitig einsetzen zu können, sollte man nicht mehrere Aufgaben in einer Funktion kombinieren, sondern stattdessen unterschiedliche Funktionen schreiben.

## Funktionen definieren

Die Definition einer Funktion beginnt wie folgt:

- Das Schlüsselwort **def**.
- Der Name der Funktion.
- Die Parameter der Funktion in runden Klammern. Eine Funktion muss nicht unbedingt Parameter haben, aber die Klammern werden immer benötigt.
- Ein Doppelpunkt.

Für Funktionsnamen gelten die gleichen Regeln wie bei Variablennamen, also:

- Die folgenden Zeichen sind erlaubt: Buchstaben, Zahlen, Unterstrich
- Funktionsnamen dürfen nicht mit einer Zahl beginnen

Laut dem [Style Guide](#) sollen Funktionsnamen kleingeschrieben und einzelne Worte im Namen durch Unterstriche getrennt werden.

Diese Elemente werden alle in eine Zeile geschrieben. In den nächsten Zeilen folgt--eingerückt--der Funktionskörper mit dem Code, den die Funktion ausführen soll. Mit `return` kann dann das Ergebnis wieder ausgegeben werden.

```
def funktionsname (parameter1, parameter2):
    funktionskörper
```

## Beispiel

```
def multiplizieren (wert_a, wert_b):
    produkt = wert_a * (wert_b + 2)
    return produkt

ergebnis = multiplizieren (wert_a = 42, wert_b = 21)
ergebnis
```

966

Wenn Sie die Funktion aufrufen, werden die Argumente--also die eigentlichen Werte, die Sie in Klammern angegeben haben (42 und 21)--innerhalb der Funktion den Parametern der Funktion *a* und *b* zugewiesen. Das Produkt der beiden Zahlen wird dann der Wert der Variable *produkt*; dieser Wert wird von der Funktion zurückgegeben, als Wert in der Variable *ergebnis* gespeichert und steht ab diesem Zeitpunkt dem Programm zur Verfügung.

Übrigens: Beim Aufrufen einer Funktion bezeichnet man die in Klammern angegebenen Werte als Argumente. In der Funktionsdefinition heißen sie dann Parameter.

Was passiert, wenn Sie jetzt den Namen der Variable *produkt* eingeben, um sich den Wert anzeigen zu lassen?

## Lokale Variablen

Sie sollten folgende Fehlermeldung bekommen: `name 'produkt' is not defined`

Python kann die Variable *produkt* nicht finden, da sie--wie auch *wert\_a* und *wert\_b*--eine sog. *lokale Variable* (*local variable*) ist. Das heißt, dass sie nur innerhalb der Funktion `multiplizieren()` existiert. Wenn Sie Zugriff auf den Wert einer Variable haben möchten, müssen Sie ihn mit `return()` wieder ausgeben. Wenn Sie mehrere Werte ausgeben möchten, können Sie eine dafür geeignete Datenstruktur verwenden (z.B. ein Tupel oder, wie im folgenden Beispiel, eine Liste):

```
def multiplizieren (wert_a, wert_b):
    produkt = wert_a * (wert_b + 2)
    summe = produkt + produkt
    return [summe, produkt]
```

```
ergebnis = multiplizieren(wert_a = 42, wert_b = 21)
ergebnis
```

```
[1932, 966]
```

## return

*return None* beendet eine Funktion, ohne einen Wert zurückzugeben. Im folgenden Beispiel wird die Funktion vorzeitig beendet, wenn eine Bedingung (eine der beiden Zahlen ist 0) zutrifft:

```
def multiplizieren(wert_a, wert_b):
    if wert_a == 0 or wert_b == 0:
        return None
    else:
        produkt = wert_a * (wert_b + 2)
        summe = produkt + produkt
        return summe

ergebnis = multiplizieren(wert_a = 42, wert_b = 0)
ergebnis
```

## Parameter

Sehen wir uns wieder unsere Funktion an:

```
def multiplizieren(wert_a, wert_b):
    produkt = wert_a * (wert_b + 2)
    return produkt

ergebnis = multiplizieren(wert_a = 42, wert_b = 21)
ergebnis
```

```
966
```

Hier haben wir beim Aufrufen der Funktion nicht nur die Werte der Argumente angegeben, sondern auch deren Namen. Dadurch könnten wir deren Reihenfolge ändern und würden dennoch das gleiche Ergebnis bekommen.

```
def multiplizieren(wert_a, wert_b):
    produkt = wert_a * (wert_b + 2)
    return produkt
```

```
ergebnis = multiplizieren(wert_a = 42, wert_b = 21)
ergebnis
```

```
966
```

```
ergebnis = multiplizieren(wert_b = 21, wert_a = 42)
ergebnis
```

Wir können allerdings die Namen auch weglassen, müssen jetzt aber auf die korrekte Reihenfolge der Argumente achten, da wir sonst unterschiedliche Ergebnisse bekommen:

```
ergebnis = multiplizieren(42, 21)
ergebnis
```

966

```
ergebnis = multiplizieren(21, 42)
ergebnis
```

924

Es ist auch möglich, beide Methoden miteinander zu kombinieren. Sie müssen dabei aber darauf achten, dass Sie erst diejenigen Argumente aufführen, welche durch ihre Position spezifiziert werden (*positional argument*), und erst danach die Argumente mit einem Schlüsselwort (*keyword argument*):

```
ergebnis = multiplizieren(42, wert_b = 21)
ergebnis
```

966

```
ergebnis = multiplizieren(wert_a = 42, 21)
ergebnis
```

```
File "<ipython-input-16-bee75674236d>", line 1
    ergebnis = multiplizieren(wert_a = 42, 21)
                                         ^
SyntaxError: positional argument follows keyword argument
```

Python bietet auch die Möglichkeit, Funktionen flexibler zu gestalten, indem mehrere Argumente an einen Funktionsparameter übergeben werden können. Wenn die Argumente dabei über ihre Position zugewiesen werden, stellt man dem betreffenden Parameter in der Parameterliste der Funktion einen Stern (\*) voran:

```
def multiplizieren(*zahlen):
    y = 1
    for x in zahlen:
        y *= x
    return y

ergebnis_multiplizieren = multiplizieren(5, 6, 7, 8, 9)
print(ergebnis_multiplizieren)
```

Wenn es hingegen um Argumente mit Schlüsselwörtern geht, verwendet man zwei Sterne (\*\*) vor dem Parameternamen. Die Argumente werden als ein Dictionary an die Funktion übergeben:

```
def multiplizieren(**zahlen):
    y = 1
    for x in zahlen:
        y *= zahlen[x]
    return y
```

```
ergebnis_multiplizieren = multiplizieren(a=5, b=6, c=7, d=8, e=9)
print(ergebnis_multiplizieren)
```

Im folgenden Beispiel wird noch ein zweiter Parameter eingefügt, der nur ein Argument annehmen kann. Wenn man die Funktion aufruft, wird der erste Wert aus der Argumentenliste dem Parameter *a* zugewiesen, alle anderen gehen an *zahlen*:

```
def multiplizieren_addieren(a, *zahlen):
    b = 1
    for x in zahlen:
        b = b * x
        c = b + a
    return c

ergebnis_multiplizieren_addieren = multiplizieren_addieren(100, 5, 6, 7, 8, 9)
print(ergebnis_multiplizieren_addieren)
```

Beim Schreiben einer Funktion können Sie auch Standardwerte definieren, die zum Zuge kommen, wenn beim Funktionsaufruf kein anderer Wert für einen Parameter angegeben wurde:

```
def multiplizieren_addieren(*zahlen, a=100):
    b = 1
    for x in zahlen:
        b = b * x
        c = b + a
    return c

ergebnis_multiplizieren_addieren = multiplizieren_addieren(5, 6, 7, 8, 9)
print(ergebnis_multiplizieren_addieren)
```

Wenn man im Funktionsaufruf dann einen anderen Wert festsetzt, wird der Standardwert überschrieben:

```
ergebnis_multiplizieren_addieren = multiplizieren_addieren(5, 6, 7, 8, 9, a=50)
print(ergebnis_multiplizieren_addieren)
```

## Docstrings

Zur Dokumentation von Funktionen können Sie die in einem vorhergehenden Video erwähnten Docstrings verwenden:

```
def multiplizieren_addieren(*zahlen, a=100):
    """Multipliziert mehrere Zahlen mit b und addiert dann a."""
    b = 1
    for x in zahlen:
        b = b * x
        c = b + a
    return c
```

# Debugging

## Debugging

Mit *Debugging* bezeichnet man das Suchen und Beseitigen von Fehlern (oder *bugs*) in einem Computerprogramm und es ist ein wichtiger Teil des Programmierprozesses. Sie sollten bei jedem Projekt Zeit für die Fehlerbehebung einplanen.

Ein wichtiges Hilfsmittel sind dabei bspw. die *Tracebacks* und *Exceptions*, die Python produziert und die in einem separaten Video behandelt wurden.

Viele Texteditoren und IDEs bieten zusätzliche Funktionen für das *Debugging* an, die Ihre Arbeit wesentlich erleichtern können. Im Folgenden sehen wir uns an, wie Sie Microsofts *Visual Studio Code* zum *Debugging* verwenden können.

Als Beispiel können wir das folgende Programm verwenden, welches mit einer *for*-Schleife durch eine Liste geht und für jeden der darin enthaltenen Werte erst den Wert selbst druckt, den Wert dann verdoppelt und dann noch das Ergebnis der Multiplikation anzeigt. Innerhalb der Schleife wird auch eine Funktion (*testfunction*) aufgerufen, die den Wert quadriert und kubiert und beide Ergebnisse dann als Teil eines Satzes ausgibt.

Kopieren Sie dazu den untenstehenden Code in eine neue Python-Datei, die Sie mit Visual Studio Code erstellt haben.

```
a = [1, 2, 3, 4, 5]

def testfunction(x):
    a = 2
    b = 3
    c = x ** a
    d = x ** b
    print("x ** a = {}".format(c))
    print("x ** b = {}".format(d))

for x in a:
    print("x = {}".format(x))
    testfunction(x)
    y = x * 2
    print("y = {}".format(y))

print("Done!")
```

Die *Debugging*-Funktionen von VS Code erlauben es uns, die Ausführung des Programms notfalls Zeile für Zeile zu beobachten. Dadurch können wir nachverfolgen, wie sich etwa Variablenwerte ändern und zu welchem Zeitpunkt bestimmte Bedingungen erfüllt sind.

Zwei Hinweise:

- Die folgenden Beispiele wurden mit Version 1.58.1 von VS Code getestet.

- Die von VS Code angebotenen *Debugging*-Funktionen können sich von Sprache zu Sprache unterscheiden. Falls Sie also schon mit anderen Sprachen in VS Code gearbeitet haben, kann es sein, dass manche Funktionen für Python nicht zur Verfügung stehen.

```

1 a = [1, 2, 3, 4, 5]
2
3 def testfunction(x):
4     a = 2
5     b = 3
6     c = x ** a
7     d = x ** b
8     print("x ** a = {}".format(c))
9     print("x ** b = {}".format(d))
10
11 for x in a:
12     print("x = {}".format(x))
13     testfunction(x)
14     y = x * 2
15     print("y = {}".format(y))
16
17 print("Done!")

```

The screenshot shows the Visual Studio Code interface with a Python file named `test.py` open. The code performs some basic calculations and prints the results. A tooltip for the 'Ausführen und Debuggen' (Run and Debug) button is displayed on the left sidebar. The bottom status bar indicates Python 3.9.4 64-bit.

Sie können den *Debugger* mithilfe der *f5*-Taste starten. VS Code öffnet daraufhin ein Menü, aus dem Sie eine *Debug*-Konfiguration auswählen können. Sie können hier gleich die erste Option nehmen (*Python File Debug the currently active Python file*). Damit können Sie mit der zurzeit in VS Code aktiven Datei arbeiten. Andere Optionen dienen zum *Debugging* von Programmen auf Servern (im Vgl. zu lokalen Daten) oder Webanwendungen, die mit verschiedenen *Frameworks* entwickelt wurden (die letzten vier Optionen).

The screenshot shows the Visual Studio Code interface with a dropdown menu for selecting a debug configuration. The option 'Python File Debug the currently active Python file' is highlighted. The main code editor window shows the same Python script as in the previous screenshot. The bottom status bar indicates Python 3.9.4 64-bit.

Das Programm startet jetzt. Allerdings läuft es bis zum Ende durch, ohne das der Debugger weitere Informationen ausgibt:

```

1 a = [1, 2, 3, 4, 5]
2
3 def testfunction(x):
4     a = 2
5     b = 3
6     c = x ** a
7     d = x ** b
8     print("x ** a = {}".format(c))
9     print("x ** b = {}".format(d))
10
11 for x in a:
12     print("x = {}".format(x))
13     testfunction(x)
14     y = x * 2
15     print("y = {}".format(y))
16
17 print("Done!")

```

Werte werden bis zum Ende ausgegeben

## Haltepunkte / Breakpoints

Das liegt daran, dass Sie noch Haltepunkte (oder Breakpoints) bestimmen müssen. Sie können zwischen drei Arten von Haltepunkten wählen:

| Typ   | Erklärung   | Klick  |
|---|---|--------|
| Haltepunkt  | Das Programm wird an dieser Stelle angehalten   | Links  |
| Bedingter Haltepunkt<br>( <i>Conditional Breakpoint</i> ) | Das Programm wird an dieser Stelle angehalten, wenn eine Bedingung erfüllt ist                                | Rechts |
| Protokollpunkt  | Das Programm wird nicht angehalten. Stattdessen wird eine Nachricht in der <i>Debugging-Konsole</i> angezeigt | Rechts |

Klicken Sie dazu in den weißen Bereich links neben den Zeilenummern. Normale Haltepunkte können durch einen Linksklick erzeugt werden, die anderen beiden über ein mit einem Rechtsklick geöffnetes Auswahlmenü.

## Haltepunkt erstellen

AUSFÜHREN UND D... > AUSFÜHREN

**Ausführen und Debuggen**

Öffnen Sie zum Anpassen von "Ausführen und debuggen" einen Ordner, und erstellen Sie eine launch.json-Datei.

Alle Konfigurationen für das automatische Debuggen anzeigen.

HALTEPUNKTE

- Raised Exceptions
- Uncaught Except...
- test.py** C:\User... 12

```
C: >
1 a = [1, 2, 3, 4, 5]
2
3 < def testfunction(x):
4     a = 2
5     b = 3
6     c = x ** a
7     d = x ** b
8     print("x ** a = {}".format(c))
9     print("x ** b = {}".format(d))
10
11 < for x in a:
12     print("x = {}".format(x))
13     testfunction(x)
14     y = x * 2
15     print("y = {}".format(y))
16
17 print("Done!")
```

PROBLEME AUSGABE TERMINAL DEBUGGING-KONSOLE

Windows PowerShell  
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.  
Lernen Sie das neue plattformübergreifende PowerShell kennen – <https://aka.ms/pscore6>

PS >

## Bedingten Haltepunkt erstellen

Der bedingte Haltepunkt wird nur aktiviert, wenn die Bedingung (hier `y > 5`) erfüllt ist.

11     **for** x **in** a:

● 12        print("x = {}".format(x))

13        testfunction(x)

14        y = x \* 2

15        **print("y = {}".format(y))**

Haltepunkt hinzufügen

**Bedingten Haltepunkt hinzufügen...**

Protokollpunkt hinzufügen ...

BUGGING-

Ausdruck     y > 5

15        **print("y = {}".format(y))**

## Protokollpunkt erstellen

Beim Erreichen des Protokollpunkts wird der voreingestellte Text in der *Debugging-Konsole* ausgegeben. Dieser Text kann auch Variablenwerte enthalten, hier z.B. den Wert von y: Der Wert von y ist {y}.

The screenshot shows a code editor with the following Python code:

```
11  for x in a:  
● 12      print("x = {}".format(x))  
13      testfunction(x)  
14      y = x * 2
```

A red dot at line 12 indicates a breakpoint. A tooltip for the variable `y` is displayed, containing the text "Der Wert von y ist {y}" and a "Protokollpunkt hinzufügen ..." button. Below the code, a message in the console says "Wert von y ist {y}" and shows the current value of `y` as `y = x * 2`.

## Bedienelemente

### Schaltfläche Funktion



Debugging starten / zum nächsten Haltepunkt gehen [F5]



Weiter zur nächsten Zeile (Prozedurschritt) [F10]



Weiter zur nächsten Zeile, geht bei einem Funktionsaufruf in die Funktion (Einzelschritt) [F11]



Untergeordnete Struktur (z.B. Funktion) wieder verlassen [Umschalttaste + F11]



Debugging erneut starten [Strg + Umschalttaste + F5]



Debugging enden [Umschalttaste + F5]

## Prozedurschritt

AUSFÜHREN UND D... ⋮

VARIABLEN

- Locals
  - > special variables
  - > function variable...
  - > a: [1, 2, 3, 4, 5]
  - x: 1
- > Globals

test.py x

```
C: > Users > Markus > Desktop > dateien > kurs_videos > python_ss_2021 > test.py > ...
1 a = [1, 2, 3, 4, 5]
2
3 def testfunction(x):
4     a = 2
5     b = 3
6     c = x ** a
7     d = x ** b
8     print("x ** a = {}".format(c))
9     print("x ** b = {}".format(d))
10
11 for x in a:
12     print("x = {}".format(x))
13     testfunction(x)
14     y = x * 2
15     print("y = {}".format(y))
16
17 print("Done!")
```

PROBLEME AUSGABE TERMINAL DEBUGGING-KONSOLE

Lernen Sie das neue plattformübergreifende PowerShell kennen - <https://aka.ms/pscore6>

PS C:\Users\Markus\Desktop\dateien\kurs\_videos\python\_ss\_2021> & 'C:\Users\Markus\AppData\Local\Programs\Python\Python39\python.exe' 'c:\Users\Markus\.vscode\extensions\ms-python.python-2021.6.944021595\pythonFiles\lib\python\debugpy\launcher' '58505' '--' 'c:\Users\Markus\Desktop\dateien\kurs\_videos\python\_ss\_2021\test.py'

x = 1

Zeile 13, Spalte 1 Leerzeichen: 4 UTF-8 CRLF Python ⚡ Q

## Einzelschritt

AUSFÜHREN UND D... ⋮

VARIABLEN

- Locals
  - x: 1
- > Globals

test.py x

```
C: > Users > Markus > Desktop > dateien > kurs_videos > python_ss_2021 > test.py > testfunction
1 a = [1, 2, 3, 4, 5]
2
3 def testfunction(x):
4     a = 2
5     b = 3
6     c = x ** a
7     d = x ** b
8     print("x ** a = {}".format(c))
9     print("x ** b = {}".format(d))
10
11 for x in a:
12     print("x = {}".format(x))
13     testfunction(x)
14     y = x * 2
15     print("y = {}".format(y))
16
17 print("Done!")
```

PROBLEME AUSGABE TERMINAL DEBUGGING-KONSOLE

Lernen Sie das neue plattformübergreifende PowerShell kennen - <https://aka.ms/pscore6>

PS C:\Users\Markus\Desktop\dateien\kurs\_videos\python\_ss\_2021> & 'C:\Users\Markus\AppData\Local\Programs\Python\Python39\python.exe' 'c:\Users\Markus\.vscode\extensions\ms-python.python-2021.6.944021595\pythonFiles\lib\python\debugpy\launcher' '58727' '--' 'c:\Users\Markus\Desktop\dateien\kurs\_videos\python\_ss\_2021\test.py'

x = 1

Zeile 4, Spalte 1 Leerzeichen: 4 UTF-8 CRLF Python ⚡ Q

# Comprehensions

## Listen-Abstraktion (*list comprehensions*)

Wie Sie wissen, kann man über eine Schleife Werte in einer Liste ausgeben oder manipulieren. Im folgenden Beispiel haben wir eine Liste *a* mit sechs Elementen. In einer *for*-Schleife gehen wir durch die Liste, quadrieren jeden Wert und speichern den neuen Wert *c* in einer separaten Liste (*b*):

```
a = [1, 2, 3, 4, 5, 6]
b = []
for x in a:
    c = x**2
    b.append(c)
b
```

```
[1, 4, 9, 16, 25, 36]
```

Mithilfe von *list comprehensions* (Listen-Abstraktionen) können wir die *for*-Schleife vermeiden und den Code kompakter gestalten:

```
a = [1, 2, 3, 4, 5, 6]
b = [x**2 for x in a]
b
```

```
[1, 4, 9, 16, 25, 36]
```

Ein *list comprehension* wie `[x**2 for x in a]` besteht aus mindestens drei Teilen:

- `x**2`: Ein Ausdruck beschreibt die Operation, die durchgeführt werden soll
- `for x`: Ein Wert, der sich bei jedem Durchgang ändert und der in der Operation verwendet werden kann
- `in a`: Ein iterierbares Objekt, das alle Werte enthält, die bearbeitet werden sollen

Wir können jetzt weitere Elemente hinzufügen, wie z.B. Bedingungen. Im folgenden Beispiel gehen wir von derselben Liste aus, verwenden dann aber nur ungerade Zahlen:

```
a = [1, 2, 3, 4, 5, 6]
b = [x**2 for x in a if x % 2 == 1]
b
```

```
[1, 9, 25]
```

*List comprehensions* können nicht nur Listen als Ausgangsmaterial verwenden, sondern auch mit anderen iterierbaren Objekten arbeiten, wie z.B. *sets* oder ein *range*-Objekt, d.h. eine mit der Funktion *range()* erzeugte Zahlenfolge:

```
a = set([1, 2, 3, 4, 5, 6]) # ein set
```

```
b = [x**2 for x in a]
b
```

```
[1, 4, 9, 16, 25, 36]
```

```
b = [x**2 for x in range(1, 7)] # ein range-Objekt
b
```

```
[1, 4, 9, 16, 25, 36]
```

Wir können auch mehrere iterierbare Objekte in einem *list comprehension* verwenden. Das folgende Beispiel berechnet Potenzen, wobei die Werte aus Liste a die Basis und die Werte aus Liste b den Exponenten bilden. Dabei wird für jede Basis jeweils der Potenzwert mit den Exponenten 2, 3 und 4 berechnet; dann folgt die nächste Basis:

```
a = [1, 2, 3, 4, 5, 6]
b = [2, 3, 4]
c = [x**y for x in a for y in b]
c
```

```
[1, 1, 1, 4, 8, 16, 9, 27, 81, 16, 64, 256, 25, 125, 625, 36, 216, 1296]
```

Dieser *list comprehension* kann auch in nur einer Zeile geschrieben werden:

```
a = [x**y for x in range(1, 7) for y in range(2, 5)]
a
```

```
[1, 1, 1, 4, 8, 16, 9, 27, 81, 16, 64, 256, 25, 125, 625, 36, 216, 1296]
```

Das Ganze kann natürlich auch mit verschachtelten *for*-Schleifen implementiert werden, ist dann allerdings auch länger:

```
a = []
for x in range(1, 7):
    for y in range(2, 5):
        a.append(x**y)
a
```

```
[1, 1, 1, 4, 8, 16, 9, 27, 81, 16, 64, 256, 25, 125, 625, 36, 216, 1296]
```

## Set Comprehensions

Ähnlich wie wir mit *list comprehensions* aus verschiedenen Datenstrukturen Listen erzeugen können, so können wir *set comprehensions* verwenden, um *sets* zu generieren. Der einzige Unterschied in der Syntax besteht darin, dass ein *set comprehension* in geschweifte Klammern eingefasst wird:

```
a = [1, 2, 3, 4, 5, 6]
b = {x**2 for x in a}
b
```

```
{1, 4, 9, 16, 25, 36}
```

# Dict Comprehensions

Schließlich gibt es *dict comprehensions* (oder *dictionary comprehensions*), mit denen man *dictionaries* erzeugen kann. Wie die eben besprochenen *set comprehensions* verwenden auch sie geschweifte Klammern. Beachten Sie den Doppelpunkt in `x:x`. Der Wert vor dem Doppelpunkt bildet im resultierenden *dictionary* den Schlüssel (*key*), nach dem Doppelpunkt kommt der Wert (*value*). Hier stellt die Zahl aus Liste a den Schlüssel dar. Dieselbe Zahl wird dann verdoppelt und als korrespondierender Wert in das *dictionary* eingefügt.

```
a = [1, 2, 3, 4, 5, 6]
b = {x:x * 2 for x in a}
b
```

```
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12}
```

Wir können auch verschiedene Arten von *comprehensions* kombinieren. Hier haben wir einen *dict comprehension*, mit dem wir durch Liste a gehen. Diese Zahlen bilden die Schlüssel im *dictionary* und die Basen für die erzeugten Potenzen. Mithilfe des *list comprehension* durchlaufen wir dann Liste b. Die erzeugte Liste stellt dann den Wert, oder *value*, dar.

```
a = [1, 2, 3, 4, 5, 6]
b = [2, 3, 4]
c = {x:[x**y for y in b] for x in a}
c
```

```
{1: [1, 1, 1],
2: [4, 8, 16],
3: [9, 27, 81],
4: [16, 64, 256],
5: [25, 125, 625],
6: [36, 216, 1296]}
```

# Virtuelle Umgebungen

## Virtuelle Umgebungen

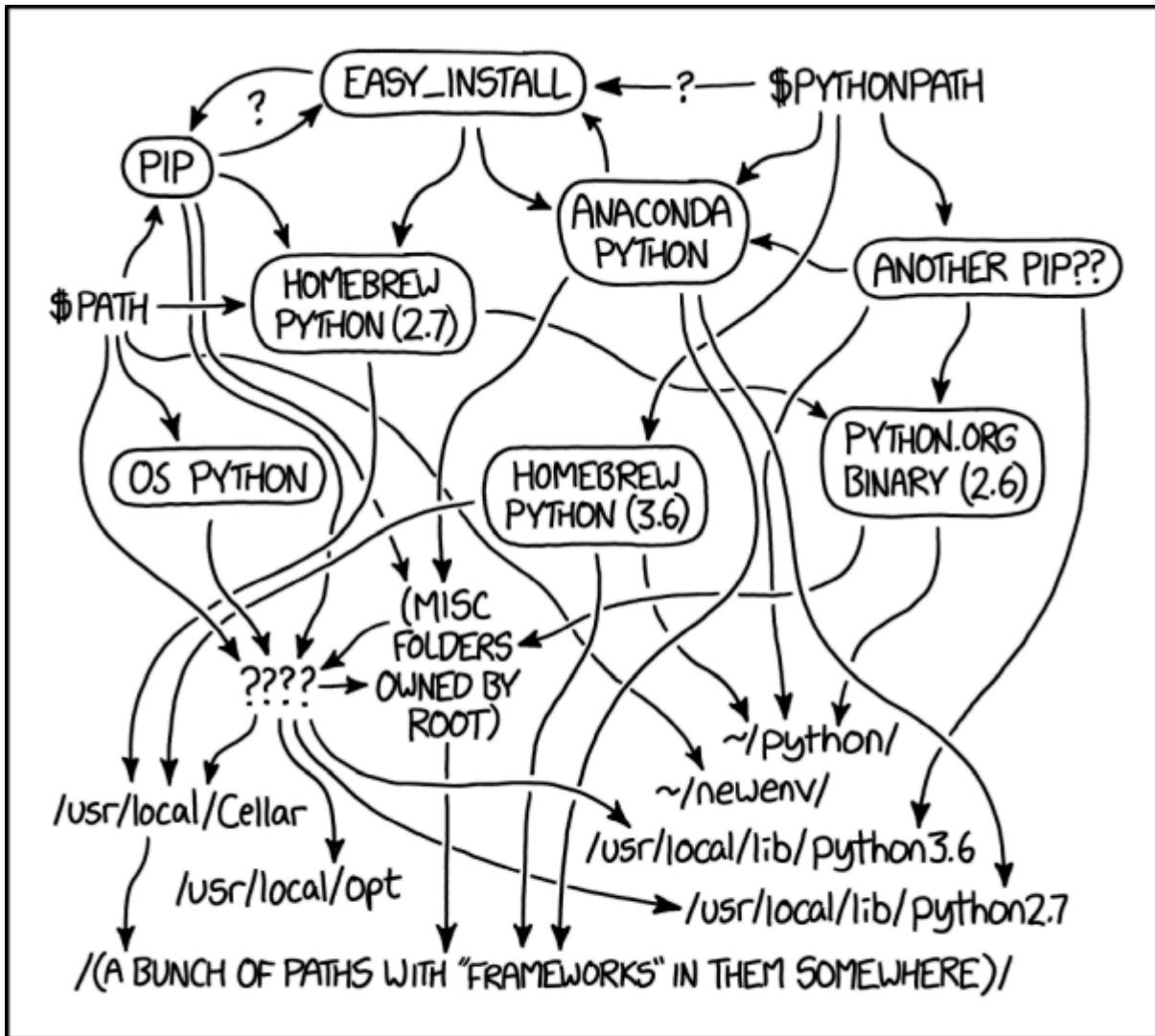
Wie Sie wissen, können Sie mithilfe von Paketen die Funktionsfähigkeit Ihrer Python-Installation je nach Bedarf erweitern. Pakete müssen nur einmal installiert, aber bei jeder Programmausführung neu geladen werden.

Was aber können Sie machen, wenn Sie für unterschiedliche Projekte auch unterschiedliche Versionen von einzelnen Paketen benötigen? Oder Sie möchten, dass für jedes Ihrer Projekte auch nur die notwendigen Pakete installiert werden?

Hier können **virtuelle Umgebungen** (*virtual environments*) helfen.

Wenn Sie Python auf Ihrem Computer installieren, wird eine globale Installation durchgeführt. Diese steht dann allen Ihren Projekten zur Verfügung, ebenso wie die Pakete, die Sie hier installieren. Mit einer virtuellen Umgebung können Sie einzelne Umgebungen voneinander abschotten und so Konflikte zwischen Projekten aufgrund von Versionsunterschieden bei Paketen vermeiden.

## Wie eine Pythonumgebung aussehen kann



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED  
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Quelle: <https://xkcd.com/1987/>

## Anaconda und virtuelle Umgebungen

Falls Sie Anaconda verwendet haben, um Python (und andere Programme) zu installieren, haben Sie bei der Installation bereits eine virtuelle Umgebung mit dem Namen "base" erzeugt. Wenn Sie jetzt statt der regulären Eingabeaufforderung den *Anaconda Prompt* verwenden, um ein Paket zu installieren, wird dieses standardmäßig innerhalb der virtuellen Umgebung installiert und steht deshalb auch nur dort zur Verfügung. Dies ist oft der Grund, warum Sie Fehlermeldungen wegen fehlender Pakete bekommen, obwohl Sie diese eigentlich installiert haben.

## virtualenv vs. venv

Es gibt mehrere Möglichkeiten, eine virtuelle Umgebung einzurichten. Häufig wird das Paket **virtualenv** verwendet, das Sie zusätzlich installieren müssen. Die nun offiziell empfohlene Lösung ist aber **venv**, welches mittlerweile Teil der Python-Standardinstallation ist.

Im Folgenden wird gezeigt, wie Sie **venv** verwenden, um unter Windows eine virtuelle Umgebung einzurichten und dann unter MS Visual Studio Code zu verwenden.

## Virtuelle Umgebung mit *venv* einrichten

Wir benötigen hier die Windows Eingabeaufforderung. Sie können diese z.B. starten, wenn Sie in der Textbox neben der Windows-Startschaltfläche "cmd" eingeben.

Wenn Sie die Eingabeaufforderung öffnen, befinden Sie sich in Ihrem Arbeitsverzeichnis (*working directory*). Sie können dann Dateien, die sich auch in diesem Ordner befinden, ohne Angabe eines Dateipfades, also nur über den Dateinamen öffnen. Das Arbeitsverzeichnis kann jederzeit geändert werden, indem Sie in einen anderen Ordner navigieren(dazu gleich mehr).

Der Dateipfad ist normalerweise `C:\Users\[Ihr Benutzername]`, z.B. `C:\Users\JaneDoe`.

Wenn Sie die Eingabeaufforderung starten, ist das Arbeitsverzeichnis meist auch Ihr Benutzerverzeichnis (*home directory*), also das Verzeichnis, in dem Ihre persönlichen Einstellungen gespeichert werden. Dieses Verzeichnis kann normalerweise nicht geändert werden. Auf einem Windows-PC finden Sie das Benutzerverzeichnis oft unter `C:\Users[Ihr Benutzername]`, z.B. `C:\Users\JaneDoe`.

Zuerst benötigen wir einen Ordner, in dem die virtuelle Umgebung angelegt werden kann. Wir können einen bereits vorhandenen Ordner nehmen oder einen neuen anlegen. Um zu einem bereits existierenden Ordner zu gelangen, können wir folgende Befehle in die Eingabeaufforderung eingeben:

| Befehl                       | Erklärung  |
|------------------------------|--|
| <code>cd ..</code>           | Damit gelangen wir in den jeweils übergeordneten Ordner  |
| <code>cd [ORDNERNAME]</code> | Damit gelangen wir in einen untergeordneten Ordner, z.B. mit <code>cd pythonkurs</code> kommen wir in den Ordner <i>pythonkurs</i> |

**cd** steht hier für "change directory" (Verzeichnis wechseln).

Wenn wir einen neuen Ordner erstellen möchten, dann navigieren wir mittels der eben gesehenen Befehle in einen übergeordneten Ordner. Dort können wir dann mittels `mkdir [Ordnername]` (z.B. `mkdir meine_virtuellen_umgebungen`) einen neuen Ordner für unsere virtuelle Umgebung erstellen und dann in diesen Ordner wechseln (im Beispiel mit `cd meine_virtuellen_umgebungen`).

Alternativ können wir auch von überall aus `mkdir` zusammen mit dem kompletten Verzeichnispfad verwenden, um nicht erst in den übergeordneten Ordner wechseln zu müssen: `mkdir C:\Users\JaneDoe\meine_virtuellen_umgebungen`

**mkdir** steht hier für "make directory" (Verzeichnis erstellen).

Dann erstellen wir dort mit *venv* eine virtuelle Umgebung, die wir hier "virtuelle\_umgebung" nennen:

```
python -m venv virtuelle_umgebung
```

`venv` erzeugt dann einen Ordner "virtuelle\_umgebung" und mehrere Unterordner. Der Ordner enthält eine Kopie des Python-Interpreters, der für diese Umgebung ausgewählt wurde, und alle Pakete, die darin installiert wurden.

Jetzt wechseln wir noch in den Ordner, in dem wir die virtuelle Umgebung angelegt haben:

```
cd virtuelle_umgebung
```

Die neue Umgebung muss dann nur noch aktiviert werden. Unter Windows funktioniert das mit dem folgenden Befehl:

```
Scripts\activate.bat
```

Wenn die virtuelle Umgebung aktiv ist wird ihr Name zu Beginn der Befehlszeile angezeigt:

```
(virtuelle_umgebung)
C:\Users\JaneDoe\meine_virtuellen_umgebungen\virtuelle_umgebung>
```

Wenn Sie nun die globale oder eine andere virtuelle Umgebung verwenden möchten, können Sie die aktuelle Umgebung folgendermaßen aktivieren:

```
Scripts\deactivate.bat
```

## Virtuelle Umgebungen und MS Visual Studio Code

Die neue virtuelle Umgebung kann nun auch in Visual Studio Code genutzt werden. Zuerst müssen wir festlegen, welcher Python-Interpreter genutzt werden soll. Der Prozess kann mit folgendem Shortcut gestartet werden:

| Windows                   | macOS                    |
|---------------------------|--------------------------|
| <code>Ctrl+Shift+P</code> | <code>Cmd+Shift+P</code> |

Damit öffnen wir die *Command Palette* von Visual Studio Code. Dort beginnen wir damit, den Befehl *Python: Select Interpreter* einzugeben und wählen ihn dann aus, wenn er im Menü angezeigt wird.

Visual Studio Code zeigt jetzt alle Python-Interpreter an, die es auf dem Computer finden konnte. Wir wählen jetzt den Interpreter aus, den wir verwenden möchten (z.B., den in unserer neuen virtuellen Umgebung). Wenn die Umgebung nicht angezeigt wird, schließen Sie Visual Studio Code und starten Sie das Programm erneut. Führen Sie jetzt den Befehl *Python: Select Interpreter* noch einmal aus.

## Virtuelle Umgebungen auf anderem Computer verwenden

Falls Sie die neue virtuelle Umgebung auf einem anderen Computer verwenden möchten, dann können Sie mit *pip* eine Anforderungsdatei (*requirements file*) erstellen. Diese führt alle installierten Pakete sowie deren Versionsnummern auf.

Während Ihre Umgebung aktiviert ist, geben Sie folgenden Befehl ein:

| Betriebssystem | Befehl                         |
|----------------|--------------------------------|
| Windows        | pip freeze > requirements.txt  |
| macOS          | pip3 freeze > requirements.txt |

*pip* erzeugt jetzt eine Anforderungsdatei namens "requirements.txt". Sie können diese Datei dann auf einen anderen Computer kopieren und--sobald Sie eine neue virtuelle Umgebung erstellt haben--die Pakete dort mit diesem Befehl installieren:

| Betriebssystem | Befehl                           |
|----------------|----------------------------------|
| Windows        | pip install -r requirements.txt  |
| macOS          | pip3 install -r requirements.txt |

Wenn Sie jetzt innerhalb dieser virtuellen Umgebung ein Paket installieren, steht dieses auch nur dort und nicht in der globalen Umgebung zur Verfügung.

Als Beispiel wird hier das NLTK (Natural Language Toolkit) mit **pip**, dem Paketeverwaltungssystem von Python, installiert:

```
(virt_u) C:\Users\meinkonto\test\virt_u>pip install nltk
```

# Tracebacks

## Tracebacks

Sie haben sicher schon einmal eine solche oder ähnliche Meldung bekommen:

```
a = range(1,11)
for x in a:
print(a)

File "<ipython-input-13-b6e9d9fc4cff>", line 3
    print(a)
    ^
IndentationError: expected an indented block
```

Diese Meldung zeigt Ihnen an, dass Ihr Code aufgrund einer fehlenden Einrückung nicht ausgeführt werden konnte. Es handelt sich dabei um einen sog. **traceback** (von *to trace back*, zurückverfolgen). Tracebacks helfen Ihnen dabei, Fehler in Ihren Programmen aufzuspüren.

Wenn der obige Code in einer Python-Datei gespeichert und dann über die Befehlszeile ausgeführt wird, führt er zu folgender Meldung:

```
C:\Users\benutzername\Desktop>python test.py
  File "C:\Users\benutzername\Desktop\test.py", line 3
    print(a)
    ^
IndentationError: expected an indented block
```

In Python werden Tracebacks von unten nach oben gelesen. In der jeweils untersten Zeile wird die Fehlerkategorie genannt (im obigen Fall handelt es sich um einen *IndentationError*, d.h. einen Einrückungsfehler). Daneben steht die Fehlermeldung ("expected an indented block"), die genauer beschreibt, warum der Code nicht ausgeführt werden konnte:

**IndentationError: expected an indented block**

Im Beispiel stellen die nächsten zwei Zeilen den Code dar, bei dessen Ausführung der Fehler bemerkt wurde; in diesem Fall ist es der Aufruf der *print*-Funktion. Der Zirkumflex (^) markiert dabei die Stelle, an welcher der Fehler zum ersten Mal aufgefallen ist. Das ist oft--aber nicht immer--auch die Stelle, wo der Fehler sitzt:

```
print(a)
^
```

Schließlich wird noch angegeben, in welcher Datei und welcher Zeile der Fehler gefunden wurde. Wenn Sie nur mit einer Datei und ohne importierte Module arbeiten, ist das vielleicht noch nicht so wichtig, aber bei zunehmender Komplexität Ihrer Projekte erleichtern diese Angaben das Aufspüren von Fehlerquellen:

**File "C:\Users\benutzername\Desktop\test.py", line 3**

Hier ein Beispiel. Speichern Sie die beiden folgenden Codesegmente in separaten Pythondateien mit

den angegebenen Dateinamen, aber im selben Ordner.

Datei 1: testmodul.py

```
def funktion1(a):
    for x in a:
        if x % 2 == 1:
            b = x * c
            print(b)
```

Datei 2: programm.py

```
from testmodul import funktion1

a = range(1,11)

funktion1(a)
```

Starten Sie jetzt die Datei programm.py in der Befehlszeile:

```
C:\Users\benutzername\Desktop>python programm.py
```

Sie sollten folgende Fehlermeldung bekommen:

```
Traceback (most recent call last):
  File "C:\Users\benutzername\Desktop\programm.py", line 5, in <module>
    funktion1(a)
  File "C:\Users\benutzername\Desktop\testmodul.py", line 4, in funktion1
    b = x * c
NameError: name 'c' is not defined
```

Wir können hier sehen, wie der Fehler von Datei zu Datei verfolgt wird. Es beginnt mit der zuerst aufgerufenen Datei *programm.py*. Diese importiert dann die Datei *testmodul.py* und ruft in Zeile 5 die darin definierte Funktion *funktion1* auf.

Hier kommt es nun zum Fehler. In Zeile 4 wird eine Variable (*c*) aufgerufen, die wir noch nicht definiert hatten. Dies verursacht einen *NameError*, wie er am Ende des Traceback angezeigt wurde. Wir wissen jetzt also, in welcher Zeile und in welcher Datei wir den Fehler beheben können.

```
Traceback (most recent call last):
  File "C:\Users\benutzername\Desktop\programm.py", line 5, in <module>
    funktion1(a)
  File "C:\Users\benutzername\Desktop\testmodul.py", line 4, in funktion1
    b = x * c
NameError: name 'c' is not defined
```

# Exceptions

## Exceptions

Dieser Prozess, in dem solche Meldungen an übergeordnete Teile eines Programms weitergegeben werden, nennt man *Exception Handling* oder Ausnahmebehandlung und die Fehlermeldungen werden als *Exceptions* bezeichnet.

*Exceptions* sind Objekte mit Methoden und Attributen, die alle von der Klasse *BaseException* erben. Es ist auch möglich, eigene *Exception*-Klassen zu entwickeln; diese sollten dann von *Exception* erben.

## Häufig vorkommende Exceptions

Zu den häufig vorkommenden *Exceptions* gehören zuerst einmal syntaktische Fehler oder *SyntaxErrors*. Ein solcher Fehler wird bspw. angezeigt, wenn wir--wie im folgenden Beispiel--in der ersten Zeile einer Funktionsdefinition den Doppelpunkt am Ende vergessen:

```
def funktion1(x)
    print(x)

File "<ipython-input-2-051492fbcaf>", line 1
    def funktion1(x)
    ^
SyntaxError: invalid syntax
```

Dazu gehören auch *IndentationError* (wenn wir in einer Zeile nicht die nötige Anzahl von Einrückungen haben) oder *TabError* (wenn wir in einer Datei sowohl Leerzeichen als auch Tabulatoren für Einrückungen verwenden).

Die *Exception TypeError* wird gezeigt, wenn wir eine Operation durchführen wollen, die für einen Datentypen--oder eine Kombination von Datentypen--nicht erlaubt ist, wie hier die Addition einer Zahl und einer Zeichenkette:

```
a = 1 + "a"

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-3-1a82a5d81a3c> in <module>
      1 a = 1 + "a"
-----
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Im nächsten Fall haben wir einen *ValueError*. Hier verwenden wir zwar den korrekten Datentyp, aber der Wert selbst ist ungültig. Aber halt, wie können wir eine Zeichenkette an *int()* übergeben? *int()* akzeptiert auch Zahlen in Form einer Zeichenkette, z.B. `a = int("2")`.

```
a = int("two")
print(a)
```

```
-----
ValueError: invalid literal for int() with base 10: 'two'
```

```
ValueError                                                 Traceback (most recent call last)
<ipython-input-4-f5e5d55ba1eb> in <module>
----> 1 a = int("two")
      2 print(a)

ValueError: invalid literal for int() with base 10: 'two'
```

Ein *KeyError* bedeutet, dass wir versucht haben, mit einem nichtvorhandenen Schlüssel auf Elemente eines Dictionary zuzugreifen:

```
a = {"zutat1": "butter", "zutat2": "eier", "zutat3": "zucker"}
print(a["zutat4"])
```

```
-----
KeyError                                                 Traceback (most recent call last)
<ipython-input-5-824e3d127ac5> in <module>
      1 a = {"zutat1": "butter", "zutat2": "eier", "zutat3": "zucker"}
----> 2 print(a["zutat4"])

KeyError: 'zutat4'
```

Schließlich seien noch *IndexError* erwähnt. Diese werden ausgegeben, wenn wir ein Element aus einer Datenstruktur (z.B. einer Liste) auswählen wollen, das nicht existiert. Hier haben wir eine Liste mit fünf Elementen und wir versuchen, auf ein sechstes Element zuzugreifen:

```
a = ["null", "eins", "zwei", "drei", "vier"]
print(a[5])
```

```
-----
IndexError                                              Traceback (most recent call last)
<ipython-input-6-f65b6e5c12d1> in <module>
      1 a = ["null", "eins", "zwei", "drei", "vier"]
----> 2 print(a[5])

IndexError: list index out of range
```

Hier können Sie eine Liste und Beschreibungen aller zurzeit in Python 3 vordefinierten *Exceptions* finden: <https://docs.python.org/3/library/exceptions.html>

Übrigens müssen wir nicht darauf warten, bis in einem Programm ein Fehler auftritt, um eine *Exception* sehen zu können, sondern wir können sie mit dem Schlüsselwort **raise** direkt aufrufen:

```
raise NameError("Hello, World!")
```

```
-----
NameError                                              Traceback (most recent call last)
<ipython-input-7-8f3a51407ece> in <module>
----> 1 raise NameError("Hello, World!")

NameError: Hello, World!
```

## Mit *Exceptions* arbeiten

Wir haben jetzt gesehen, wie wir mithilfe von *Exceptions* in einem *Traceback* Fehler in unseren Pythonprogrammen aufspüren und beheben können. Allerdings können wir unseren Programmen auch beibringen, selbst auf solche Meldungen zu reagieren und entweder andere Meldungen anzuzeigen oder, abhängig vom Fehlertyp, anderen Code auszuführen.

Wenn dann bei der Programmausführung bspw. ein Problem auftritt, kann dieses möglicherweise vom Programm selbst gelöst werden und es ist dann nicht nötig, einen *Traceback* auszugeben.

Im folgenden Beispiel wird getestet, ob der Code innerhalb des **try**-Blocks ausgeführt werden kann. Wenn ja, wird der Wert von `1 / x` der Variable *ergebnis* zugewiesen und dann von der Funktion *funktion1* zurückgegeben.

Verursacht die Zeile `ergebnis = 1 / x` aber einen *TypeError*, wird stattdessen aber der Code im **except**-Block ausgeführt; *ergebnis* bekommt dann den Wert "Bitte wählen Sie eine Zahl als Wert aus." und dieser wird dann ausgegeben.

```
def funktion1(x):
    try:
        ergebnis = 1 / x
    except TypeError:
        ergebnis = "Bitte wählen Sie eine Zahl als Wert aus."
    return ergebnis

funktion1("a")
```

```
'Bitte wählen Sie eine Zahl als Wert aus.'
```

Wenn nötig, können wir auch mehrere **except**-Blöcke einfügen. Hier testen wir zusätzlich, ob es einen "ZeroDivisionError" gibt:

```
def funktion1(x):
    try:
        ergebnis = 1 / x
    except TypeError:
        ergebnis = "Bitte wählen Sie eine Zahl als Wert aus."
    except ZeroDivisionError:
        ergebnis = "Bitte wählen Sie nicht 0 als Wert."
    return ergebnis

funktion1(0)
```

```
'Bitte wählen Sie nicht 0 als Wert.'
```

Nach den **try**- und **except**-Blöcken können wir einen optionalen **else**-Block einfügen. Dieser Block wird nach dem Code im **try**-Block ausgeführt, wenn es keine *Exception* gab. Der Grund, warum der Code nicht gleich im **try**-Block ausgeführt wird, ist, dass man den **try**-Block so kurz wie möglich halten will. Dann kann man sicherstellen, dass auch nur die Zeile, die man testen will, einen möglichen Fehler verursacht hat.

Im Beispiel wird das Ergebnis im **else**-Block mit 2 multipliziert, falls es keine *Exception* gab.

```
def funktion1(x):
    try:
        ergebnis = 1 / x
    except TypeError:
        ergebnis = "Bitte wählen Sie eine Zahl als Wert aus."
    except ZeroDivisionError:
        ergebnis = "Bitte wählen Sie nicht 0 als Wert."
    else:
        ergebnis *= 2

funktion1(2)
```

```
ergebnis = 2 * ergebnis
return ergebnis

funktion1(4)
```

0.5

Schließlich können wir am Ende noch einen (auch optionalen) **finally**-Block anhängen; darin befindet sich dann Code, der auf jeden Fall ausgeführt werden soll, egal ob es eine *Exception* gab oder nicht:

```
def funktion1(x):
    try:
        ergebnis = 1 / x
    except TypeError:
        ergebnis = "Bitte wählen Sie eine Zahl als Wert aus."
    except ZeroDivisionError:
        ergebnis = "Bitte wählen Sie nicht 0 als Wert."
    else:
        ergebnis = 2 * ergebnis
    finally:
        print("Ende der Funktion!")
    return ergebnis

funktion1("a")
```

```
Ende der Funktion!
'Bitte wählen Sie eine Zahl als Wert aus.'
```

## Weitere Informationen

- Dokumentation zu *Errors and Exceptions*: <https://docs.python.org/3/tutorial/errors.html>
- Ernesti, Johannes und Peter Kaiser. *Python 3: Das umfassende Handbuch*. 5., aktualisierte Auflage. Bonn: Rheinwerk, 2017. Frei zugänglich unter: <http://openbook.rheinwerk-verlag.de/python/index.html>.
  - Kapitel 22: Ausnahmebehandlung

# Objektorientierung

## Objektorientierte Programmierung (OOP)

Objektorientierte Programmierung (*object-oriented programming*) ist ein sogenanntes Programmierparadigma, was Wikipedia<sup>1</sup> als einen "fundamentale[n] Programmierstil" definiert. Ein solches Paradigma legt bestimmte Prinzipien fest, welche die Softwareentwicklung leiten. Es gibt mehrere verschiedene Paradigmen, wie etwa die imperitative oder die deklarative Programmierung. Programmiersprachen können ein oder mehrere Paradigmen unterstützen.

1 <https://de.wikipedia.org/wiki/Programmierparadigma>

## Python und OOP

Python unterstützt objektorientierte Programmierung. Ohne es wahrscheinlich zu wissen, haben Sie in Python schon mit Objekten gearbeitet, denn alles, was Sie in Python erzeugen, ist ein Objekt.

Ein Objekt enthält eine Beschreibung von dessen Charakteristiken, oder **Attributen** (*attributes*), und der **Methoden** (*methods*), die auf das Objekt angewandt werden können. All das wird in einer **Klasse** (*class*) festgelegt. Methoden sind wie Funktionen--mit denen Sie schon vertraut sind--die innerhalb einer Klasse definiert werden.

## Klassen / Instanzen

Eine Klasse selbst ist noch kein Objekt, sondern lediglich eine Anleitung oder ein Rezept für dessen Erstellung. So wäre etwa eine Klasse "Brot" mit einem Backrezept vergleichbar. Erst durch das **Instanziieren** (*instantiation*) wird ein Objekt erzeugt, oder, um bei dem Vergleich zu bleiben, ein Brot gebacken. Das Objekt (das Brot) bezeichnet man dann als eine **Instanz** der Klasse "Brot."

Wie bei einem Backrezept auch kann man beliebig viele Instanzen erzeugen (Brote backen), die sich auch in ihren Attributwerten unterscheiden können (etwa durch unterschiedliche Zutaten und/oder Dosierungen), aber von ihrer Grundstruktur her immer noch zu der gleichen Klasse gehören.

## Vererbung

Wenn wir genauer zwischen einzelnen Brotsorten unterscheiden müssen, etwa weil unterschiedliche Arbeitsschritte erforderlich sind, können wir die Klasse "Brot" weiter unterteilen, u.a. in eine Klasse "Brötchen." "Brot" wird jetzt zu einer **Basisklasse** (*superclass*), welche die Attribute und Methoden enthält, die auf alle Brotsorten zutreffen.

"Brötchen" stellt jetzt eine **Unterklasse** (*subclass*) von "Brot" dar und übernimmt--oder "erbt"--erst einmal diese Attribute und Methoden. Deshalb spricht man auch von **Vererbung** (*inheritance*). In der Klassendefinition von "Brötchen" können wir dann Attribute und Methoden direkt übernehmen oder modifizieren; die Modifikationen stehen dann nur in dieser Klasse zur Verfügung.

# Beispiel: Konto

Am Beispiel eines Bankkontos lässt sich gut erklären, wie man ein in der realen Welt existierendes Objekt oder Konzept mit Python modellieren kann. Es wird daher oft in Tutorien und auch im folgenden Beispiel verwendet.

Eine Bank bietet meist mehrere Kontotypen an. Jeder dieser Typen ist normalerweise nicht auf einen Kunden maßgeschneidert, sondern wird von mehreren Kunden genutzt. Jeder Kontotyp kann dann in einer eigenen Klasse definiert werden.

Ein einzelnes Konto ist eine Instanz der jeweiligen Kontoklasse. Alle Konten haben zwar bestimmte Attribute (wie etwa "Kontonummer" oder "Inhaber") gemein, aber deren Werte unterscheiden sich, da die Konten verschiedene Kontonummern und Inhaber haben.

## Wie sieht das in Python aus?

Die Definition einer Klasse beginnt mit dem Schlüsselwort **class** und dem Namen der Klasse:

```
class Konto:  
    pass
```

Wie Sie schon wissen, macht *pass* hier nichts und ist nur ein Lückenfüller, bis wir unsere Klasse weiter ausbauen; ohne *pass* würden wir hier eine Fehlermeldung bekommen.

## Konstruktor

Im Moment können wir aber mit unserer Klasse noch nicht viel anfangen. Zuerst müssen wir unter anderem die verschiedenen Attribute der Klasse definieren. Dies können wir mit einem **Konstruktor** (*constructor*) erreichen. Der Konstruktor ist ein wichtiger Teil jeder Klasse und wird als eine Methode mit dem Namen \_init\_ (zwei Unterstriche vor und nach dem Wort *init*) implementiert.

```
class Konto:  
    def __init__(self, inhaber, kontonummer):  
        pass
```

## Instanzattribute

Hier definieren wir zwei Attribute, *Inhaber* und *Kontonummer*. Diese sind sog. **Instanzattribute** (*instance attributes*), da ihre Werte nur in der jeweiligen Instanz gelten.

```
class Konto:  
    def __init__(self, inhaber, kontonummer):  
        self.Inhaber = inhaber # Instanzattribut  
        self.Kontonummer = kontonummer # Instanzattribut
```

## Klassenattribute

Im Gegensatz zu Instanzattributen treffen **Klassenattribute** (auch statiche Attribute genannt) auf alle Instanzen einer Klasse zu. Sehen wir uns folgendes Beispiel an, in dem wir außerhalb des Konstruktors noch einen Wert für einen maximalen Tagesumsatz festlegen:

```
class Konto:  
    MaximalerTagesumsatz = 2000 # Klassenattribut  
    def __init__(self, inhaber, kontonummer):  
        self.Inhaber = inhaber # Instanzattribut  
        self.Kontonummer = kontonummer # Instanzattribut
```

Für jedes Konto der Klasse "Konto" gilt jetzt, dass der maximale Tagesumsatz 2000 beträgt. Die Werte für Inhaber und Kontonummer können aber für jede Instanz anders ausfallen.

## Instanzen erzeugen

An dieser Stelle können wir eine erste Instanz unserer Klasse "Konto" erzeugen. Wir generieren hier ein Konto für "Max Mustermann" mit der Kontonummer "20385710932345". Beachten Sie, dass wir hier "MaximalerTagesumsatz" nicht mehr festlegen müssen, da er bereits als Klassenattribut definiert ist. Auch stehen die folgenden Zeilen nach der Klassendefinition. Wie wir schon bei Funktionen gesehen haben muss auch eine Klasse erst definiert werden bevor wir sie verwenden können.

```
mein_konto = Konto("Max Mustermann", 20385710932345)  
mein_konto.Inhaber
```

```
'Max Mustermann'
```

```
mein_konto.Kontonummer
```

```
20385710932345
```

```
mein_konto.MaximalerTagesumsatz
```

```
2000
```

## Methoden

Um unsere Klasse sinnvoll nutzen zu können, benötigen wir aber noch weitere Komponenten, in diesem Fall die zuvor erwähnten Methoden. In unserem Beispiel legen die Methoden fest, welche Aktionen man in einem Konto ausführen kann.

```
class Konto:  
    MaximalerTagesumsatz = 2000 # Klassenattribut  
    def __init__(self, inhaber, kontonummer):  
        self.Inhaber = inhaber # Instanzattribut  
        self.Kontonummer = kontonummer # Instanzattribut  
  
    def name_zeigen(self):  
        print("Name: {}".format(self.Inhaber))
```

Hier haben wir der Klasse eine erste Methode hinzugefügt, welche den Namen des Kontoinhabers anzeigt. `self` bezieht sich auf die Instanz, von der aus die Methode aufgerufen wurde und muss in der Methodendefinition immer erwähnt werden. Wenn wir später die Methode aufrufen, wird es nicht mehr benötigt.

Jetzt fügen wir noch zwei Methoden hinzu, welche jeweils die Kontonummer und den maximalen Tagesumsatz anzeigen (wir könnten diese Funktionalität natürlich auch in unsere erste Funktion integrieren):

```
class Konto:
    MaximalerTagesumsatz = 2000 # Klassenattribut
    def __init__(self, inhaber, kontonummer):
        self.Inhaber = inhaber # Instanzattribut
        self.Kontonummer = kontonummer # Instanzattribut

    def name_zeigen(self):
        print("Name: {}".format(self.Inhaber))
    def kontonummer_zeigen(self):
        print("Kontonummer: {}".format(self.Kontonummer))
    def maximaler_tagesumsatz_zeigen(self):
        print("Maximaler Tagesumsatz: {}".format(self.MaximalerTagesumsatz))
```

In diesem Beispiel erzeugen wir eine neue Instanz und verwenden die obigen Methoden, um unsere Werte anzeigen zu lassen.

```
c = Konto("Hans Mustermann", 12345)
c.name_zeigen()
c.kontonummer_zeigen()
c.maximaler_tagesumsatz_zeigen()
```

```
Name: Hans Mustermann
Kontonummer: 12345
Maximaler Tagesumsatz: 2000
```

Wir haben jetzt die Grundstruktur einer möglichen Basisklasse für unser allgemeines Konto. Wir können davon auch Unterklassen für verschiedene spezialisierte Kontotypen ableiten, welche Methoden und Attribute von der Basisklasse erben (*inheritance*). Das folgende Beispiel zeigt eine Unterklasse für ein Girokonto, welches einen niedrigeren maximalen Tagesumsatz hat:

```
class Girokonto(Konto):
    MaximalerTagesumsatz = 1000
    def __init__(self, inhaber, kontonummer):
        pass
```

Was passiert jetzt, wenn wir versuchen, eine Instanz dieser Klasse zu erzeugen?

```
d = Girokonto("Martina Mustermann", 54321)
d.Inhaber
```

```
-----
AttributeError                                     Traceback (most recent call last)
Input In [4], in <cell line: 2>()
    1 d = Girokonto("Martina Mustermann", 54321)
----> 2 d.Inhaber
```

```
AttributeError: 'Girokonto' object has no attribute 'Inhaber'
```

Der Grund für die Fehlermeldung ist, dass die Unterklasse nicht mehr auf den Konstruktur der Basisklasse--in welchem etwa das Attribut "Inhaber" definiert ist--zugreifen kann; wir haben ihn ja mit dem Konstruktur der Unterklasse überschrieben. Um den Konstruktur der "Konto"-Klasse weiterhin nutzen zu können, fügen wir dem "Girokonto"-Konstruktur eine Zeile hinzu.

```
class Girokonto(Konto):
    MaximalerTagesumsatz = 1000
    def __init__(self, inhaber, kontonummer):
        super().__init__(inhaber, kontonummer)
```

Damit zeigen wir Python, dass wir die beiden Attribute "inhaber" und "kontonummer" aus dem Konstruktur der **superclass** übernehmen wollen.

```
d = Girokonto("Martina Mustermann", 54321)
d.Inhaber
```

```
'Martina Mustermann'
```

Auch die Methoden werden von der Basisklasse übernommen. Wenn wir möchten, können wir sie jedoch entweder durch neue Methoden ergänzen oder mit gleichnamigen Versionen überschreiben, welche an den neuen Kontotyp angepasst sind und dann innerhalb der Klasse "Girokonto" zur Verfügung stehen:

```
class Girokonto(Konto):
    MaximalerTagesumsatz = 1000
    def __init__(self, inhaber, kontonummer):
        super().__init__(inhaber, kontonummer)
    def kontonummer_zeigen(self):
        print("Nummer des Girokontos: {}".format(self.Kontonummer))
```

```
d = Girokonto("Martina Mustermann", 54321)
d.name_zeigen()
d.maximaler_tagesumsatz_zeigen()
d.kontonummer_zeigen()
```

```
Name: Martina Mustermann
Maximaler Tagesumsatz: 1000
Nummer des Girokontos: 54321
```

## Statische Methoden

Statische Methoden (*static methods*) beziehen sich nicht auf eine spezielle Instanz und können direkt aufgerufen werden, ohne dass zuvor eine Instanz erzeugt wird. Sie werden erst normal definiert und danach mit der Funktion *staticmethod()* in eine statische Methode umgewandelt:

```
class X:
    def anzeigen():
        print("Das ist eine statische Methode!")
    anzeigen = staticmethod(anzeigen)
```

```
X.anzeigen()
```

Das ist eine statische Methode!

## Klassenmethoden

Klassenmethoden können sowohl über die Instanz als über den Klassennamen aufgerufen werden:

```
class A:  
    # Der Name "cls" ist kein Schlüsselwort, sondern eine Python-Konvention für  
    # das erste Argument  
    # einer Klassenmethode  
    def anzeigen(cls):  
        print("Das ist Klasse {}".format(cls))  
    anzeigen = classmethod(anzeigen)  
  
A.anzeigen()  
  
X = A()  
X.anzeigen()
```

Das ist Klasse <class '\_\_main\_\_.A'>  
Das ist Klasse <class '\_\_main\_\_.A'>

## Decorators

Die eben gezeigte Methode, Klassen- und statische Methoden zu erzeugen, funktioniert zwar. In der Praxis hat sich aber der Gebrauch von **Decorators** durchgesetzt, um den Code etwas überschaubarer zu gestalten. Hier noch einmal die gleichen Methoden, welche nun aber einen Decorator mit einem vorangestellten "@" nutzen:

```
class X:  
    @staticmethod  
    def anzeigen():  
        print("Das ist eine statische Methode!")  
  
X.anzeigen()
```

Das ist eine statische Methode!

```
class A:  
    @classmethod  
    def anzeigen(cls): # Der Name "cls" ist kein Schlüsselwort, sondern eine  
    # Python-Konvention  
        print("Das ist Klasse {}".format(cls))  
  
A.anzeigen()  
  
X = A()  
X.anzeigen()
```

Das ist Klasse <class '\_\_main\_\_.A'>  
Das ist Klasse <class '\_\_main\_\_.A'>

# Magic Methods / Magic Attributes

Bei *Magic Methods* und *Magic Attributes* handelt es sich nicht um Harry Potters erste Programmierversuche. Es sind vielmehr Methoden und Attribute, die Sie normalerweise nicht durch ihren Namen aufrufen müssen, sondern die bei Bedarf im Hintergrund ausgeführt werden.

Sie haben schon eine *Magic Method* kennengelernt, `__init__`. Sie wird automatisch bei der Instantiierung ausgeführt.

Die Namen aller *Magic Methods* und *Magic Attributes* beginnen und enden mit einem doppelten Unterstrich.

Ein anderes Beispiel ist `__call__`. Damit kann man Klassen erstellen, deren Instanzen man wie eine Funktion nutzen kann:

```
class Potenz:
    def __init__(self, exponent):
        self.Exponent = exponent
    def __call__(self, basis):
        return basis ** self.Exponent

hoch5 = Potenz(5)
hoch5(3)
# Beispiel nach Ernesti und Kaiser, S. 372
```

243

## Weiterführende Lektüre zu Objektorientierter Programmierung

Dies war nur eine Einführung in objektorientierte Programmierung. Wenn Sie mehr darüber lernen möchten, finden Sie hier weitere Informationen:

Johannes Ernesti und Peter Kaiser. *Python 3. Das umfassende Handbuch*. 5., aktualisierte Auflage. Bonn: Rheinwerk, 2017. Kapitel 21. UB Signatur: 59 A 7577. [Allgemein sehr empfehlenswert. Ausführliche, aktuelle und gut verständliche Darstellung der objektorientierten Programmierung in Python.]

- eBook: <http://openbook.rheinwerk-verlag.de/python/>

Michael Weigend. *Python 3 lernen und professionell anwenden. Das umfassende Praxisbuch*. 8., erweiterte Auflage. Frechen: mitp, 2019. Kapitel 10. UB Signatur: inf P 9008. [Auch gute Erklärung.]

- eBook: <https://learning.oreilly.com/library/view/-/9783747500538/?ar>

Bernhard Lahres und Gregor Raýman. *Objektorientierte Programmierung. Das umfassende Handbuch*. 2. aktualisierte und erweiterte Auflage. Bonn: Galileo Press, 2009. UB Signatur: 49 A 8584. [Enthält Beispiele in mehreren Sprachen, u.a. auch Python.]

- eBook: <http://openbook.rheinwerk-verlag.de/oop/>

Auf Englisch:

*Python Object-Oriented Programming: Hands-on for Beginners [Updated for 2021]. Packt Publishing, 2021.*

- Serie kurzer Videos zur objektorientierten Programmierung in Python (insgesamt über drei Stunden)
- Zugang unter <https://www.oreilly.com/library/view/python-object-oriented-programming/9781801077613/?ar>
- "Universitaet Tuebingen" auswählen und dann mit universitären Nutzernamen und Passwort anmelden

# Dateien lesen und schreiben

## Dateien öffnen

Damit wir eine Datei lesen und/oder Daten darin speichern können, müssen wir sie zuerst mit der Funktion **open()** öffnen. Wenn wir bspw. eine Datei namens "staedte.txt" haben, die sich zusammen mit unserer Pythondatei in einem Verzeichnis befindet und deren Inhalt wir nur lesen, aber nicht verändern möchten, können wir sie folgendermaßen öffnen:

```
datei = open("staedte.txt", "r")
```

## Mode

Mit dem zweiten Parameter *mode* (im Beispiel "r") können wir festlegen, was wir mit der Datei machen möchten:

| Mode | Steht für           | Erklärung   |
|------|---------------------|---|
| r    | read (lesen)        | Daten werden nur gelesen.   |
| rb   | read binary         | Binärdaten werden nur gelesen. Nützlich, wenn man nicht mit Text, sondern bspw. mit Bilddaten arbeitet.                                 |
| w    | write (schreiben)   | Daten werden in die Datei geschrieben. Wenn sich dort bereits Daten befinden, werden diese überschrieben.                               |
| wb   | write binary        | Binärdaten werden in die Datei geschrieben.   |
| a    | append (hinzufügen) | Daten werden in die Datei geschrieben. Wenn sich dort bereits Daten befinden, werden die neuen Daten an das Ende der Datei geschrieben. |

Wenn wir uns nun mit *print()* den Inhalt der Datei anzeigen lassen möchten, bekommen wir folgendes Ergebnis:

```
print(datei)

<_io.TextIOWrapper name='staedte.txt' mode='r' encoding='cp1252'>
```

Die Funktion **open()** gibt ein sog. *TextIOWrapper*-Objekt zurück. Wir können aber über eine Schleife den Inhalt der Datei Zeile für Zeile ausgeben. Das zweite Argument (*end*) verhindert dabei, dass Python am Zeilenende eine zusätzliche Leerzeile einfügt (versuchen Sie es mal ohne den Parameter):

```
for i in datei:
    print(i, end=' ')
```

Stuttgart  
Karlsruhe  
Mannheim  
Freiburg  
Heidelberg  
Ulm

## Datei schließen

Wenn wir die Daten ausgelesen haben, müssen wir die Datei auch wieder mit **close()** schließen:

```
datei.close()
```

Es gibt aber auch eine andere Möglichkeit. Mit einer *with*-Anweisung können wir die Datei öffnen und einlesen, ohne sie dann explizit schließen zu müssen. Dies geschieht automatisch, sobald der Code innerhalb der *with*-Anweisung ausgeführt wurde:

```
with open("staedte.txt", "r") as daten:  
    for i in daten:  
        print(i, end="")
```

Stuttgart  
Karlsruhe  
Mannheim  
Freiburg  
Heidelberg  
Ulm  
Heilbronn  
Pforzheim

Denselben Ansatz können wir auch verwenden, wenn wir Daten in die Datei schreiben wollen. Hier fügen wir eine weitere Stadt zu unserer Liste hinzu. Beachten Sie das vorangestellte *\n*. Es ist ein Kontrollzeichen, das Sie schon aus dem Video zu regulären Ausdrücken kennen und das dafür sorgt, dass der neue Eintrag auch in eine neue Zeile geschrieben wird:

```
with open("staedte.txt", "a") as daten:  
    daten.write("\nReutlingen")
```

Wenn wir uns jetzt die Liste noch einmal ansehen, dann finden wir dort auch "Reutlingen":

```
with open("staedte.txt", "r") as daten:  
    for i in daten:  
        print(i, end="")
```

Stuttgart  
Karlsruhe  
Mannheim  
Freiburg  
Heidelberg  
Ulm  
Heilbronn  
Pforzheim  
Reutlingen

Wir können auch mehrere Dateien zur gleichen Zeit öffnen, um bspw. unsere Liste in ein zweites Dokument zu übertragen. Dazu lesen wir mit **readlines()** unsere Originaldatei Zeile für Zeile in eine Liste ein und schreiben die Listeneinträge wieder mit **writelines()** Zeile für Zeile in die neue Datei:

```
with open("staedte.txt", "r") as daten, open("staedte_neu.txt", "w") as
```

```
daten_neu:  
    staedtenamen = daten.readlines()  
    daten_neu.writelines(staedtenamen)
```

```
with open("staedte_neu.txt", "r") as daten:  
    for i in daten:  
        print(i, end="")
```

```
Stuttgart  
Karlsruhe  
Mannheim  
Freiburg  
Heidelberg  
Ulm  
Heilbronn  
Pforzheim  
Reutlingen
```

## Datei schreiben in *pandas*

Sie haben im Video zu *pandas* bereits gesehen, wie Sie mit diesem Paket Daten aus einer externen Datei einlesen und bearbeiten können. Es ist natürlich auch möglich, diese bearbeiteten Dateien wieder in eine Datei zu schreiben.

Als Beispiel nehmen wir die Datei "staedte\_neu.txt," die wir eben erstellt haben und jetzt noch einmal einlesen werden. Bisher haben wir noch keinen Namen für die erst einmal einzige Spalte. Den fügen wir mit der letzten Zeile hinzu:

```
import pandas as pd  
  
df = pd.read_csv("staedte_neu.txt", header=None, sep="\t")  
  
df.columns = ["stadt"]
```

```
df
```

### stadt

```
0    Stuttgart  
1    Karlsruhe  
2    Mannheim  
3    Freiburg  
4    Heidelberg  
5    Ulm  
6    Heilbronn  
7    Pforzheim  
8    Reutlingen
```

Wir fügen jetzt eine Spalte "bundesland" hinzu:

```
df["bundesland"] = "bw"
```

```
df
```

|   | stadt      | bundesland |
|---|------------|------------|
| 0 | Stuttgart  | bw         |
| 1 | Karlsruhe  | bw         |
| 2 | Mannheim   | bw         |
| 3 | Freiburg   | bw         |
| 4 | Heidelberg | bw         |
| 5 | Ulm        | bw         |
| 6 | Heilbronn  | bw         |
| 7 | Pforzheim  | bw         |
| 8 | Reutlingen | bw         |

Wenn wir das Ergebnis jetzt abspeichern möchten, können wir auf eine der von *pandas* bereitgestellten Schreibfunktionen zurückgreifen. In diesem Fall speichern wir das Ganze mithilfe der **to\_csv()**-Funktion und dem Argument **sep** als eine .tsv-Datei.

```
df.to_csv("staedte.csv", encoding='utf-8', index=False, sep="\t")
```

Wenn Sie mit einem Paket arbeiten, welches Funktionen zum Lesen und Schreiben von Dateien anbietet (wie etwa *pandas*), dann sollten Sie von diesen Gebrauch machen, da diese oft eine Vielzahl von Formaten unterstützen und Zusatzfunktionen, wie hier die Auswahl von Trennungszeichen, bereits zuverlässig implementiert sind.

# Module os und os.path

## os und os.path

Das Modul **os** und sein Untermodul **os.path** bieten eine Vielzahl von Funktionen, um mit dem Betriebssystem zu interagieren. Die teilweise großen Unterschiede zwischen den einzelnen Plattformen, auf denen Python laufen kann, werden dadurch (fast) ausgeglichen, so dass Sie sich kaum Gedanken darüber machen müssen, wie Ihr Code auf einem bestimmten Betriebssystem läuft.

Wie üblich importieren wir zuerst unsere Module:

```
import os
from os import path
```

Wenn wir eine Liste der Dateien und Ordner in einem bestimmten Verzeichnis benötigen, können wir das mit **os.listdir()** bekommen. Hier ist zu beachten, dass Windows in Dateipfaden keinen Schrägstriche zur Abgrenzung der einzelnen Ebenen verwendet, sondern einen *backslash*, der normalerweise als Escapezeichen dient. Wir benötigen deshalb einen doppelten *backslash*, damit das Zeichen richtig interpretiert wird:

```
os.listdir("C:\\\\Users\\\\Kurs\\\\Desktop")
```

```
['archiv', 'Datei_1.txt', 'desktop.ini', 'Ordner_1', 'test.txt']
```

Wenn nötig, können wir mit **os.mkdir()** und **os.makedirs()** neue Ordner erzeugen. **os.mkdir()** erstellt nur einen Ordner, dessen übergeordnete Ordner bereits existieren müssen, während **os.makedirs()** alle Ordner erstellt, die sich im Dateipfad befinden:

```
os.mkdir("C:\\\\Users\\\\Kurs\\\\Desktop\\\\Ordner_1")
```

```
os.makedirs("C:\\\\Users\\\\Kurs\\\\Desktop\\\\Ordner_1\\\\Ordner_2\\\\Ordner_3")
```

Zum Löschen von Dateien steht **os.remove()** zur Verfügung, wohingegen **os.rmdir()** einen einzelnen leeren Ordner löschen kann:

```
os.remove("C:\\\\Users\\\\Kurs\\\\Desktop\\\\Datei_1.txt")
```

```
-----
FileNotFoundException                                Traceback (most recent call last)
<ipython-input-26-4e7df5f45401> in <module>
----> 1 os.remove("C:\\\\Users\\\\Kurs\\\\Desktop\\\\Datei_1.txt")
```

```
FileNotFoundException: [WinError 2] Das System kann die angegebene Datei nicht finden: 'C:\\\\Users\\\\Kurs\\\\Desktop\\\\Datei_1.txt'
```

```
os.rmdir("C:\\\\Users\\\\Kurs\\\\Desktop\\\\Ordner_1")
```

Wenn ein Ordner Unterordner hat, können wir die gesamte Struktur mit der Funktion **rmtree()** aus dem Modul **shutil** entfernen:

```
import shutil  
shutil.rmtree("C:\\\\Users\\\\Kurs\\\\Desktop\\\\Ordner_1")
```

Ein Problem, das bei der Arbeit mit Dateien auftauchen kann, sind fehlende Berechtigungen, um eine Datei lesen, bearbeiten oder ausführen zu können. Mit **os.access()** können wir unsere Rechte für eine Datei abfragen. Mit dem zweiten Argument bestimmen wir, welcher Test durchgeführt werden soll:

- *F\_OK*: Existiert die Datei?
- *R\_OK*: Dürfen wir die Datei lesen?
- *W\_OK*: Dürfen wir die Datei verändern?
- *X\_OK*: Dürfen wir die Datei ausführen?

```
os.access("C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt", os.R_OK)
```

True

Manchmal benötigt man eine Liste aller Dateien, die sich in einem Ordner und dessen Unterverzeichnissen befinden. Hier kann die Funktion **os.walk()** helfen. Die Daten werden in *Tuples* gespeichert, die jeweils den Pfad ohne Dateinamen, den Ordner und den Dateinamen enthalten. Wir können die Daten dann in einer Schleife ausgeben.

Im folgenden Beispiel lassen wir die Funktion einen Ordner durchsuchen und dann den gesamten Dateipfad anzeigen. Dazu verwenden wir auch die Funktion **os.path.join()**, um den Pfad mit dem Dateinamen zu verbinden (dazu gleich mehr).

```
for path, dirs, files in os.walk("C:\\\\Users\\\\Kurs\\\\Desktop"):  
    for name in files:  
        print(os.path.join(path, name))
```

```
C:\\\\Users\\\\Kurs\\\\Desktop\\\\Datei_1.txt  
C:\\\\Users\\\\Kurs\\\\Desktop\\\\desktop.ini  
C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt  
C:\\\\Users\\\\Kurs\\\\Desktop\\\\archiv\\\\pruefsummen.txt  
C:\\\\Users\\\\Kurs\\\\Desktop\\\\archiv\\\\faustinetragdi01loepgoog_tif\\\\faustinetragdi01loep  
C:\\\\Users\\\\Kurs\\\\Desktop\\\\archiv\\\\faustinetragdi01loepgoog_tif\\\\faustinetragdi01loep  
C:\\\\Users\\\\Kurs\\\\Desktop\\\\archiv\\\\faustinetragdi01loepgoog_tif\\\\faustinetragdi01loep  
... und so weiter ...
```

Weitere nützliche Funktionen finden sich in einem Untermodul von **os**, nämlich **os.path**. Mit **os.path.join()** haben wir eben schon eine dieser Funktion gesehen. Es wäre in diesem Beispiel zwar

möglich gewesen, den Dateipfad und Dateinamen mithilfe der **format()**-Funktion zusammenzufügen. Allerdings gibt es zwischen manchen Betriebssystemen Unterschiede bei der Formatierung von Dateipfaden. **os.path.join()** hilft hier, Probleme zu vermeiden, indem es sich automatisch an das Betriebssystem anpasst, auf dem Python gerade läuft.

Wir können einen kompletten Dateipfad in den Pfad zum übergeordneten Verzeichnis einerseits und den Dateinamen andererseits trennen, und zwar mit **os.path.split()** verwenden:

```
os.path.split("C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt")
```

```
('C:\\\\Users\\\\Kurs\\\\Desktop', 'test.txt')
```

Wenn wir nur einen der beiden Teile benötigen, bekommen wir diese mit **os.path.dirname()** (für das übergeordnete Verzeichnis) und **os.path.basename()** (für den Dateinamen). Mit **os.path.splitext()** können wir die Dateiendung isolieren:

```
os.path.dirname("C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt")
```

```
'C:\\\\Users\\\\Kurs\\\\Desktop'
```

```
os.path.basename("C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt")
```

```
'test.txt'
```

```
os.path.splitext("C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt")
```

```
('C:\\\\Users\\\\Kurs\\\\Desktop\\\\test', '.txt')
```

Schließlich können wir noch mehrere Tests durchführen:

- **os.path.exists**: Existiert die Datei?
- **os.path.getsize**: Wie groß ist die Datei?
- **os.path.isfile**: Verweist der Pfad auf eine Datei?
- **os.path.isdir**: Verweist der Pfad auf ein Verzeichnis?

```
os.path.exists("C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt")
```

```
True
```

```
os.path.getsize("C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt")
```

```
19
```

```
os.path.isfile("C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt")
```

```
True
```

```
os.path.isdir("C:\\\\Users\\\\Kurs\\\\Desktop\\\\test.txt")
```

```
False
```



# Modul `pathlib`

## Warum `pathlib`?

Sie kennen schon eine Methode, in Python mit Dateien und Dateipfaden zu arbeiten, und zwar mit dem Modul `os` und seinem Untermodul `os.path`. So können Sie etwa Dateipfade aus mehreren Elementen zusammensetzen oder den Inhalt eines Verzeichnisses auslesen.

Wie aber würden Sie Dateien mit bestimmten Dateiendungen (wie etwa `.py` oder `.txt`) finden? Dafür bräuchten Sie, neben `os`, noch ein weiteres Paket (z.B. `glob`). Und wenn Sie diese Dateien dann in einen anderen Ordner verschieben möchten, würden Sie auf das Paket `shutil` zurückgreifen.

Es gibt aber seit Python 3.4 mit **`pathlib`** eine sehr gute Alternative, die nicht nur all diese Funktionen in einem Paket vereint, sondern auch noch einen weiteren Vorteil bietet: `pathlib` repräsentiert Dateipfade durch ein *Path*-Objekt, anstatt sie--wie in `os.path`--als Zeichenketten zu behandeln. Das macht es einfacher, mit Dateipfaden zu arbeiten und diese zu manipulieren.

Wie immer, wenn wir mit einem Paket arbeiten möchten, müssen wir dieses erst importieren:

```
import pathlib
```

Da wir hier nur mit einer der insgesamt sechs Klassen des Pakets--der *Path*-Klasse--arbeiten werden, könnten wir diesen Teil auch direkt importieren. Wenn wir dann eine der Methoden aus dem Paket aufrufen möchten, können wir das anfängliche `pathlib.` weglassen. Anstatt beispielsweise `pathlib.Path.cwd()` würden wir dann `Path.cwd()` verwenden:

```
from pathlib import Path
```

Mithilfe der eben erwähnten Methode `cwd()` können Sie sich übrigens das aktuelle Arbeitsverzeichnis (*current working directory*) anzeigen lassen:

```
pathlib.Path.cwd()
```

Abhängig von Ihrem Betriebssystem bekommen Sie dabei entweder einen Windows-Pfad [z.B. `WindowsPath('C:\Users\User123\Desktop')`] oder einen POSIX-Pfad [z.B. `PosixPath('/Users/User123/Desktop')`], wie man ihn etwa in macOS findet.

Das Benutzerverzeichnis finden Sie mit `home()`:

```
pathlib.Path.home()
```

## Dateipfade erzeugen

Es gibt jetzt mehrere Möglichkeiten, ein *Path*-Objekt zu erzeugen. So können Sie bspw. den Pfad als Zeichenkette angeben. Mit einem POSIX-Pfad funktioniert das ohne weiteres:

```
pfad = pathlib.Path("/Users/User123/Desktop/Python/cities.txt")
```

Wenn es sich um einen Windows-Pfad handelt, sollten Sie bedenken, dass der *backslash* (\), der in Windows für Dateipfade verwendet wird, in Python auch als Escape- oder Fluchtzeichen dient und Sie daher mit der folgenden Zeile eine Fehlermeldung bekommen:

```
pfad = pathlib.Path("C:\Users\User123\Desktop\Python\cities.txt")
```

Sie können dieses Problem umgehen, indem Sie den Pfad als sog. *raw string* angeben, in welcher der *backslash* wie jedes andere Zeichen behandelt wird und keine Sonderfunktion mehr hat. Dazu müssen Sie dem Pfad nur ein "r" voranstellen:

```
pfad = pathlib.Path(r"C:\Users\User123\Desktop\Python\cities.txt")
```

Alternativ können Sie einen Pfad auch aus einzelnen Bestandteilen zusammensetzen. *pathlib* verwendet dabei Schrägstriche, um die einzelnen Bestandteile aneinanderzufügen:

```
pfad = pathlib.Path.home() / "Desktop" / "Python" / "cities.txt"
```

Denselben Effekt erzielen Sie, wenn Sie die einzelnen Komponenten als Argumente an *joinpath()* übergeben:

```
pfad = pathlib.Path.home().joinpath("Desktop", "Python", "cities.txt")
```

## Pfadelemente auswählen

Nachdem Sie ein Pfad-Objekt erzeugt haben, können Sie direkt auf dessen einzelne Elemente zugreifen. So bekommen Sie bspw. mit *.name* das letzte Element des Pfades. Wenn Sie einen Pfad haben, der auf eine Datei verweist, ist das der Dateiname:

```
pfad = pathlib.Path(r"c:\Users\User123\Desktop\Python\cities.txt")
pfad.name
```

Die folgende Liste enthält nur eine kleine Auswahl an *Properties* und Methoden, die Ihnen zur Verfügung stehen. Eine vollständige Liste finden Sie unter

<https://docs.python.org/3/library/pathlib.html#methods-and-properties>.

| Property | Steht für   | Beispiel                           | Ergebnis  |
|----------|---|------------------------------------|---|
| drive    | Laufwerksbuchstabe, falls vorhanden                   | pfad.drive                         | c:  |
| root     | Stammverzeichnis, oberstes Verzeichnis im Dateisystem | pfad.root                          | / (macOS)   |
| anchor   | Kombination aus Laufwerk und Stammverzeichnis         | pfad.anchor                        | c: or c:\\ (Windows)<br>/ (macOS)                             |
| parents  | Übergeordnete Verzeichnisse                           | pfad.parents[0]<br>pfad.parents[1] | c:\Users\User123\Desktop\Python\<br>c:\Users\User123\Desktop\ |

| Property | Steht für | Beispiel | Ergebnis |
|----------|-----------|----------|----------|
|----------|-----------|----------|----------|

| <b>Property</b> | <b>Steht für</b>  | <b>Beispiel</b>  | <b>Ergebnis</b>                  |
|-----------------|---|--|----------------------------------|
| parent          | Das direkt übergeordnete Verzeichnis  | pfad.parent  | c:\Users\User123\Desktop\Python\ |
| suffix          | Dateierweiterung des letzten Pfadelements (falls vorhanden)                   | pfad.suffix  | .txt                             |
| suffixes        | Liste mit allen Dateierweiterungen des letzten Pfadelements (falls vorhanden) | datei =<br>pathlib.Path("cities.tar.gz") ["tar", "gz"]<br>datei.suffixes |                                  |
| stem            | Letztes Pfadelement ohne Dateierweiterung                                     | pfad.stem  | cities                           |

| <b>Methode</b> | <b>Steht für</b>   | <b>Beispiel</b>                             | <b>Ergebnis</b>                            |
|----------------|--|---|--|
| is_absolute()  | Ist ein Pfad absolut oder nicht?                         | pfad.is_absolute()                          | True                                       |
| match()        | Entspricht ein Teil des Pfades einem vorgegebenem Muster | pfad.match("cities")<br>pfad.match("towns") | True<br>False                              |
| with_suffix()  | Ändert die Dateinamenerweiterung                         | pfad.with_suffix(".rtf")                    | c:\Users\User123\Desktop\Python\cities.rtf |
| exists()       | Existiert der Pfad/die Datei?                            | pfad.exists()                               | True                                       |
| is_dir()       | Verweist der Pfad auf ein Verzeichnis?                   | pfad.is_dir                                 | False                                      |

| <b>Methode</b> | <b>Steht für</b>  | <b>Beispiel</b>  | <b>Ergebnis</b>                        |
|----------------|---|--|--|
| is_file()      | Verweist der Pfad auf eine Datei?                                 | pfad.is_file()   | True                                   |
| iterdir()      | Erzeugt für jede Datei in einem angegebenen Ordner ein Pfadobjekt | verzeichniss = pathlib.Path("c:\Users\Desktop")<br>verzeichniss.iterdir()        | Verzeichnis des Ordners wurde erstellt |
| mkdir()        | Erstellt an der angegebenen Stelle ein neues Verzeichnis          | pfad_dir =<br>pathlib.Path(r"c:\Users\User123\Desktop\test_dir")<br>pfad.mkdir() | Neues Verzeichnis wurde erstellt       |
| rename()       | Benennt Datei oder Verzeichnis um                                 | pfad.rename("places.txt")  | c:\Users\User123\Desktop\Python\place  |
| unlink         | Löscht Datei  | pfad.unlink  |  |



## Fallbeispiele

Hier nun ein paar Beispiele, wie man diese Properties und Methoden in Programmen verwenden kann:

### Verzeichnis erstellen und darin mehrere Textdateien erzeugen

```

# pathlib wird importiert
import pathlib
# Pfad wird festgelegt
path = pathlib.Path(r"C:\Users\User123\Desktop\test")
# Im angegebenen Pfad wird ein Ordner erstellt
pathlib.Path.mkdir(path)
# Start der for-Schleife. Wertebereich, der durchlaufen wird, sind die Zahlen
von 1 bis 10
for i in range(1, 11):
    # Pfad der zu erzeugenden Datei wird kreiert
    file_path = path / pathlib.Path("my_file_{}.txt".format(i))
    # Unter dem jeweiligen Pfad wird eine Datei erzeugt
    pathlib.Path.touch(file_path)

```

## Verzeichnis erstellen, darin Textdateien erzeugen und Text in Dateien schreiben

```

# pathlib wird importiert
import pathlib
# Pfad wird festgelegt
path = pathlib.Path(r"C:\Users\User123\Desktop\test")
# Im angegebenen Pfad wird ein Ordner erstellt
pathlib.Path.mkdir(path)
# Start der for-Schleife. Wertebereich, der durchlaufen wird, sind die Zahlen
von 1 bis 10
for i in range(1, 11):
    # Pfad der zu erzeugenden Datei wird kreiert
    file_path = path / pathlib.Path("my_file_{}.txt".format(i))
    # Unter dem jeweiligen Pfad wird eine Datei erzeugt
    pathlib.Path.touch(file_path)
    # Die eben erzeugte Datei wird zum Schreiben geöffnet
    with open(file_path, "w") as single_file:
        # Zeichenkette wird erzeugt. Hier werden dem Wert der Variable i führende
        Nullen
        # (leading zeros) vorangestellt, bis die Zahl zweistellig wird
        string = "This is string {:0>2}.".format(i)
        # Zeichenkette wird in die geöffnete Datei geschrieben. Falls weiter nichts in
        die Datei
        # geschrieben werden soll, wird diese automatisch geschlossen.
        single_file.write(string)

```

## Verzeichnis erstellen, darin Textdateien erzeugen, neuen Ordner erstellen, Dateien in neuen Ordner verschieben

```

# pathlib wird importiert
import pathlib
# Pfad für Ursprungsordner wird festgelegt
path1 = pathlib.Path(r"C:\Users\User123\Desktop\origin")
# Pfad für Zielordner wird festgelegt
path2 = pathlib.Path(r"C:\Users\User123\Desktop\destination")
# Ursprungsordner wird erstellt
pathlib.Path.mkdir(path1)
# Zielordner wird erstellt
pathlib.Path.mkdir(path2)

```

```

# Start der for-Schleife. Wertebereich, der durchlaufen wird, sind die Zahlen
von 1 bis 10
for i in range(1, 11):
    # Name der Datei wird kreiert
    name = "my_file_{}.txt".format(i)
    # Ursprungspfad wird kreiert
    origin_path = path1 / name
    # Zielpfad wird kreiert
    destination_path = path2 / name
    # Unter dem jeweiligen Pfad wird eine Datei erzeugt
    pathlib.Path.touch(origin_path)
    # Datei wird verschoben
    origin_path.rename(destination_path)

```

## Alle MP3-Dateien aus einem Verzeichnis löschen

```

# pathlib wird importiert
import pathlib
# Pfad wird festgelegt
path = pathlib.Path(r"C:\Users\User123\Desktop\AllMyFiles")
# Pfadobjekte für alle Dateien und Unterordner in diesem Ordner werden erzeugt
files = path.iterdir()
# Mithilfe einer for-Schleife sehen wir uns jedes gefundene Element an
for single_file in files:
    # 2 Tests:
    # - Handelt es sich um eine Datei (True) oder etwas anderes, z.B. einen Ordner
    # (False)?
    # - Ist die Dateiendung, nachdem man mit ".lower" alle Großbuchstaben im Pfad
    # durch
    #   Kleinbuchstaben ersetzt hat, gleich ".mp3" oder nicht?
    # ".lower" wird verwendet, damit man nicht noch auf ".MP3", ".Mp3", usw.
    # testen muss
    if single_file.is_file() == True and single_file.lower.suffix == ".mp3":
        # Wenn beide Bedingungen erfüllt sind, dann wird die Datei gelöscht
        single_file.unlink()

```

## Zur weiteren Lektüre

Offizielle Dokumentation zu *pathlib*. <https://docs.python.org/3/library/pathlib.html>

Trey Hunner. "Why you should be using *pathlib*." <https://treyhunner.com/2018/12/why-you-should-be-using-pathlib/>

Trey Hunner. "No really, *pathlib* is great." <https://treyhunner.com/2019/01/no-really-pathlib-is-great/>.

Geir Arne Hjelle. "Python 3's *pathlib* Module: Taming the File System."

<https://realpython.com/python-pathlib/>.

# Datenanalyse mit pandas

## Datenanalyse mit *pandas*

Python hat sich, nicht zuletzt durch seine große Paketebibliothek, zu einem der beliebtesten und leistungsfähigsten Werkzeuge für die Datenanalyse entwickelt. Wir werden uns im Folgenden mit dem Paket *pandas* beschäftigen, welches Pythons Standardlösung für die Arbeit mit tabellarischen Daten ist. Es bietet Funktionen zum Laden, Bearbeiten, Analysieren, Exportieren und Visualisieren solcher Datensätze und wir würden Ihnen dringend empfehlen, von diesen Fähigkeiten Gebrauch zu machen. Zwar können Sie viele dieser Aufgaben auch mit der Standardbibliothek erledigen. *pandas* erleichtert die Arbeit aber enorm und der damit entwickelte Code wird oft effizienter sein.

Da *pandas* nicht zur "Grundausstattung" von Python gehört, müssen Sie es erst installieren:

```
pip install pandas
```

Diese Installation kann etwas länger dauern, da Python möglicherweise noch mehrere Zusatzpakete installieren muss.

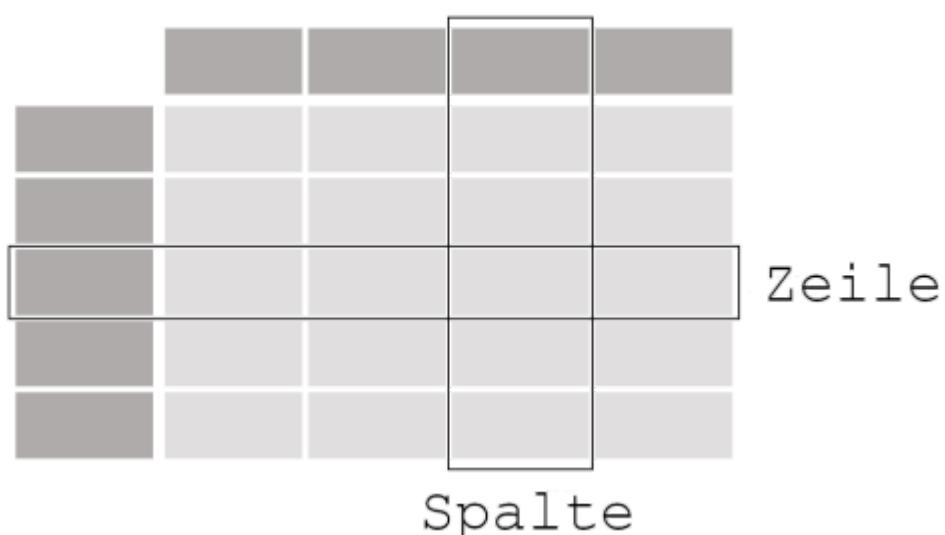
Anschließend können Sie es importieren. Gewöhnlich verwendet man dazu den Namen pd:

```
import pandas as pd
```

## Datentabellen

*pandas* verwendet eine als Datentabelle (*DataFrame*) genannte zweidimensionale Datenstruktur, die einer Tabelle in Excel gleicht und die aus Zeilen (eine Zeile pro Fall) und Spalten (eine Spalte pro Variable) besteht:

DataFrame



# Datentabellen

Datentabellen können auf verschiedene Weisen generiert werden. Eine häufig genutzte Möglichkeit besteht darin, ein Dictionary, welches mehrere Listen gleicher Länge enthält, zu einer Datentabelle zu konvertieren:

```
daten1 = {'stadt': ['Berlin', 'Hamburg', 'München'],
          'bundesland': ['Berlin', 'Hamburg', 'Bayern'],
          'einwohnerzahl': [3520031, 1787408, 1450381]}
df_stadt_1 = pd.DataFrame(daten1)
df_stadt_1
```

|   | stadt   | bundesland | einwohnerzahl |
|---|---------|------------|---------------|
| 0 | Berlin  | Berlin     | 3520031       |
| 1 | Hamburg | Hamburg    | 1787408       |
| 2 | München | Bayern     | 1450381       |

```
df_stadt_1
```

|   | stadt   | bundesland | einwohnerzahl |
|---|---------|------------|---------------|
| 0 | Berlin  | Berlin     | 3520031       |
| 1 | Hamburg | Hamburg    | 1787408       |
| 2 | München | Bayern     | 1450381       |

Wie Sie in der Tabelle sehen können, hat jede Zeile und jede Spalte eine eindeutige Bezeichnung: das ist der Index. Bei Spalten verwendet man meist Zeichenketten (wie hier "stadt", "bundesland", "einwohnerzahl"), bei Zeilen aufsteigende Zahlen. Wie wir schon in anderen Fällen (Index bei Listen und Zeichenketten) gesehen haben, beginnt man auch hier bei 0. Wenn wir später auf Spalten und Zeilen zugreifen wollen, verwenden wir diesen Index.

## Datentabellen: Einzelne Zeilen hinzufügen

Sie können jederzeit neue Zeilen hinzufügen; hier kommt eine neue Stadt hinzu. Die Daten für die neue Zeile werden zunächst in einem Dictionary gespeichert und dann in eine Datentabelle umgewandelt. Diese neue, einzelne Datentabelle wird dann an unseren Original-Datentabelle angehängt.

```
daten2 = {'stadt': 'Köln', 'bundesland': 'Nordrhein-Westfalen', \
          'einwohnerzahl': 1060582}
df2 = pd.DataFrame([daten2])
df_stadt_2 = pd.concat([df_stadt_1, df2])
df_stadt_2
```

|   | stadt  | bundesland | einwohnerzahl |
|---|--------|------------|---------------|
| 0 | Berlin | Berlin     | 3520031       |

|   | stadt   | bundesland          | einwohnerzahl |
|---|---------|---------------------|---------------|
| 1 | Hamburg | Hamburg             | 1787408       |
| 2 | München | Bayern              | 1450381       |
| 0 | Köln    | Nordrhein-Westfalen | 1060582       |

```
df_stadt_2
```

|   | stadt   | bundesland          | einwohnerzahl |
|---|---------|---------------------|---------------|
| 0 | Berlin  | Berlin              | 3520031       |
| 1 | Hamburg | Hamburg             | 1787408       |
| 2 | München | Bayern              | 1450381       |
| 0 | Köln    | Nordrhein-Westfalen | 1060582       |

Sie können hier sehen, dass unsere neue Stadt den Indexwert *0* hat. Dieser wurde aus der hinzugefügten Datentabelle übernommen.

Um stattdessen den gewünschten sequenziellen Index zu bekommen, übergeben wir *concat()* den Parameter **ignore\_index** mit dem Wert *True*. Damit wird der ursprüngliche Index der neuen Zeile ignoriert und stattdessen bekommt die Zeile den jeweils nächsten verfügbaren Indexwert:

```
daten2 = {'stadt': 'Köln', 'bundesland': 'Nordrhein-Westfalen', \
          'einwohnerzahl': 1060582}
df2 = pd.DataFrame([daten2])
df_stadt_2 = pd.concat([df_stadt_1, df2], ignore_index=True)
df_stadt_2
```

|   | stadt   | bundesland          | einwohnerzahl |
|---|---------|---------------------|---------------|
| 0 | Berlin  | Berlin              | 3520031       |
| 1 | Hamburg | Hamburg             | 1787408       |
| 2 | München | Bayern              | 1450381       |
| 3 | Köln    | Nordrhein-Westfalen | 1060582       |

## Datentabellen: Mehrere Zeilen hinzufügen

Wenn Sie mehrere Zeilen hinzufügen möchten, können Sie diese zuerst in eine Datentabelle umwandeln und dann mit **concat** ans Ende hängen:

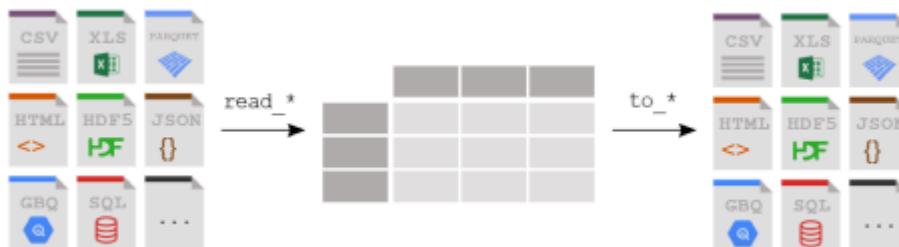
```
daten3 = {'stadt': ['Frankfurt', 'Stuttgart'],
          'bundesland': ['Hessen', 'Baden-Württemberg'],
          'einwohnerzahl': [732688, 623738]}

df3 = pd.DataFrame(daten3)
df_stadt_3 = pd.concat([df_stadt_2, df3], ignore_index=True)
df_stadt_3
```

|   | stadt     | bundesland          | einwohnerzahl |
|---|-----------|---------------------|---------------|
| 0 | Berlin    | Berlin              | 3520031       |
| 1 | Hamburg   | Hamburg             | 1787408       |
| 2 | München   | Bayern              | 1450381       |
| 3 | Köln      | Nordrhein-Westfalen | 1060582       |
| 4 | Frankfurt | Hessen              | 732688        |
| 5 | Stuttgart | Baden-Württemberg   | 623738        |

## Datentabelle: Daten laden

Sie können mit Pandas auch Daten aus einer externen Datei in eine Datentabelle laden. Dazu bietet Pandas verschiedene Methoden an, welche unterschiedliche Datenformate laden und auch wieder speichern können (z.B. Excel, JSON, SAS, Stata, SQL).



Im folgenden Beispiel verwenden wir die **read\_csv**-Methode. Diese lädt normalerweise Textdateien mit *comma-separated values* (CSV), d.h., Dateien, in denen die Werte der einzelnen Spalten durch Kommas voneinander getrennt werden. Man kann mit dem **sep**-Parameter aber auch ein anderes Zeichen auswählen; im Beispiel ist das ein Tabulatorzeichen.

Damit die Datei wie im folgenden Beispiel geladen wird, muss sie sich im selben Verzeichnis befinden wie die Python- oder Notebook-Datei mit dem ausgeführten Python-Code. Falls der Datensatz in einem anderen Verzeichnis gespeichert wurde, muss der Pfad angepasst werden.

```
df = pd.read_csv("spielerinnen_datentabelle_2019.txt", sep="\t")
```

```
df
```

|     | name                     | mannschaft | geburtsjahr | groesse | position |
|-----|--------------------------|------------|-------------|---------|----------|
| 0   | Michaela Abam            | Kamerun    | 1997        | NaN     | ST       |
| 1   | Ninon Abena              | Kamerun    | 1994        | NaN     | MF       |
| 2   | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153.0   | ST       |
| 3   | Olufolasade Adamolekun   | Jamaika    | 2001        | 173.0   | ST       |
| 4   | Yanara Aedo              | Chile      | 1993        | 154.0   | ST       |
| ... | ...                      | ...        | ...         | ...     | ...      |

|     | name             | mannschaft | geburtsjahr | groesse | position |
|-----|------------------|------------|-------------|---------|----------|
| 547 | Min-Ji Yeo       | Südkorea   | 1993        | 163.0   | ST       |
| 548 | Kumi Yokoyama    | Japan      | 1993        | 155.0   | ST       |
| 549 | Shelina Zadorsky | Kanada     | 1992        | 172.0   | AB       |
| 550 | Daniela Zamora   | Chile      | 1990        | 166.0   | ST       |
| 551 | Rui Zhang        | China      | 1989        | 172.0   | ST       |

552 rows × 5 columns

## Datentabellen anzeigen

Bei längeren Datensätzen können Sie sich mit der Methode **head()** auch nur die ersten paar Zeilen anzeigen lassen:

```
df.head()
```

|   | name                     | mannschaft | geburtsjahr | groesse | position |
|---|--------------------------|------------|-------------|---------|----------|
| 0 | Michaela Abam            | Kamerun    | 1997        | NaN     | ST       |
| 1 | Ninon Abena              | Kamerun    | 1994        | NaN     | MF       |
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153.0   | ST       |
| 3 | Olufolasade Adamolekun   | Jamaika    | 2001        | 173.0   | ST       |
| 4 | Yanara Aedo              | Chile      | 1993        | 154.0   | ST       |

Sie können auch mehr oder weniger Zeilen ausgeben, indem Sie die gewünschte Zeilenzahl an *head()* übergeben:

```
df.head(3)
```

|   | name                     | mannschaft | geburtsjahr | groesse | position |
|---|--------------------------|------------|-------------|---------|----------|
| 0 | Michaela Abam            | Kamerun    | 1997        | NaN     | ST       |
| 1 | Ninon Abena              | Kamerun    | 1994        | NaN     | MF       |
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153.0   | ST       |

Und mit **tail()** geben Sie die letzten Zeilen aus:

```
df.tail(3)
```

|     | name             | mannschaft | geburtsjahr | groesse | position |
|-----|------------------|------------|-------------|---------|----------|
| 549 | Shelina Zadorsky | Kanada     | 1992        | 172.0   | AB       |
| 550 | Daniela Zamora   | Chile      | 1990        | 166.0   | ST       |
| 551 | Rui Zhang        | China      | 1989        | 172.0   | ST       |

# Fehlende Werte

Es kann immer wieder vorkommen, dass in einem Datensatz Werte fehlen. In unserem Beispiel sind das die Größenangaben für die ersten zwei Spielerinnen, die als *NaN* markiert werden. *NaN* steht eigentlich für *Not a Number*, zeigt in *pandas* aber auch fehlende Werte in einer Datentabelle an:

```
df.head(3)
```

|   | name                     | mannschaft | geburtsjahr | groesse | position |
|---|--------------------------|------------|-------------|---------|----------|
| 0 | Michaela Abam            | Kamerun    | 1997        | NaN     | ST       |
| 1 | Ninon Abena              | Kamerun    | 1994        | NaN     | MF       |
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153.0   | ST       |

*NaN* wird von Python als Fließkommazahl (*float*) behandelt. Deshalb werden Zahlen in einer Spalte mit fehlenden Werten (wie hier "groesse") auch als Fließkommazahlen behandelt und, wenn nötig, umgewandelt.

Abhängig davon, was Sie mit Ihren Daten machen möchten, können fehlenden Werte Probleme verursachen. *pandas* bietet deshalb einige Funktionen, um mit diesem Problem umgehen zu können. Mit **fillna()** können Sie fehlende Werte durch einen vorgegebenen Wert ersetzen, wie hier durch "0":

```
df_beispiel = df.fillna(0)
df_beispiel.head(3)
```

|   | name                     | mannschaft | geburtsjahr | groesse | position |
|---|--------------------------|------------|-------------|---------|----------|
| 0 | Michaela Abam            | Kamerun    | 1997        | 0.0     | ST       |
| 1 | Ninon Abena              | Kamerun    | 1994        | 0.0     | MF       |
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153.0   | ST       |

Sie können auch mit **dropna()** Zeilen oder Spalten mit fehlenden Werten entfernen. Mit **dropna(axis=0)** entfernen Sie Zeilen, während **dropna(axis=1)** zum Entfernen von Spalten verwendet wird.

```
df_beispiel = df.dropna(axis=0)
df_beispiel.head(3)
```

|   | name                     | mannschaft | geburtsjahr | groesse | position |
|---|--------------------------|------------|-------------|---------|----------|
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153.0   | ST       |
| 3 | Olufolasade Adamolekun   | Jamaika    | 2001        | 173.0   | ST       |
| 4 | Yanara Aedo              | Chile      | 1993        | 154.0   | ST       |

```
df_beispiel = df.dropna(axis=1)
```

```
df_beispiel.head(3)
```

|   | name                     | mannschaft | geburtsjahr | position |
|---|--------------------------|------------|-------------|----------|
| 0 | Michaela Abam            | Kamerun    | 1997        | ST       |
| 1 | Ninon Abena              | Kamerun    | 1994        | MF       |
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        | ST       |

In unserem Beispiel werden wir Zeilen mit *NaN*-Werten entfernen:

```
df = df.dropna(axis=0)  
df.head(3)
```

|   | name                     | mannschaft | geburtsjahr | groesse | position |
|---|--------------------------|------------|-------------|---------|----------|
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153.0   | ST       |
| 3 | Olufolasade Adamolekun   | Jamaika    | 2001        | 173.0   | ST       |
| 4 | Yanara Aedo              | Chile      | 1993        | 154.0   | ST       |

## Datentypen ändern

Wenn *pandas* einen Datensatz importiert, versucht es, Spalten von vornherein den richtigen Datentyp zuzuweisen. Wenn das nicht richtig funktioniert oder wir möchten aus irgendeinem Grund den Datentyp ändern, dann können wir das mit der Methode **astype()** tun. Um herauszufinden, welche Datentypen wir in unserer Datentabelle haben, können wir das Attribut **dtypes** verwenden.

```
df.dtypes
```

```
name          object  
mannschaft    object  
geburtsjahr   int64  
groesse       float64  
position      object  
dtype: object
```

Wir sehen hier, dass bspw. das Geburtsjahr, eine vierstellige Zahl, als ein 64bit *integer* gespeichert wird. Der Datentyp *int64* umfasst einen *sehr* großen Wertebereich:

```
-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
```

Um diesen Wertebereich abdecken zu können, werden 64 Bit oder 8 Byte an Speicher benötigt. Wenn wir die Werte in der Spalte stattdessen als *int16* (16 Bit) speichern, benötigen wir nur noch 2 Byte.

Schauen wir uns an, wieviel Speicher unsere ursprüngliche Datentabelle benötigt:

```
df.info(memory_usage="deep", verbose=False)
```

```
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 550 entries, 2 to 551  
Columns: 5 entries, name to position
```

```
dtypes: float64(1), int64(1), object(3)
memory usage: 120.1 KB
```

Jetzt ändern wir den Datentyp in den Spalten "geburtsjahr" und "groesse" zu `int16`.

```
df = df.astype({"geburtsjahr": "int16", "groesse": "int16"})
df.dtypes
```

```
name          object
mannschaft    object
geburtsjahr   int16
groesse       int16
position      object
dtype: object
```

```
df.info(memory_usage="deep", verbose=False)
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 550 entries, 2 to 551
Columns: 5 entries, name to position
dtypes: int16(2), object(3)
memory usage: 113.6 KB
```

Sie können sehen, dass wir etwa 6KB eingespart haben. Dies scheint vielleicht nicht viel, aber bei sehr großen Datensätzen mit vielen Zahlenwerten kann es durchaus einen Unterschied bei der Verarbeitungsgeschwindigkeit und Speicherauslastung machen.

## Datentabellen erweitern

Wir können auch diesen Datensatz erweitern. Hier fügen wir eine Spalte "wm\_jahr" hinzu, in der das Jahr der Weltmeisterschaft aufgeführt wird. Das kann nützlich sein, falls man irgendwann auch andere Weltmeisterschaften berücksichtigen möchte:

```
df["wm_jahr"] = 2019
df.head(3)
```

|   |                          | name | mannschaft | geburtsjahr | groesse | position | wm_jahr |
|---|--------------------------|------|------------|-------------|---------|----------|---------|
| 2 | Gabrielle Aboudi Onguene |      | Kamerun    | 1989        | 153     | ST       | 2019    |
| 3 | Olufolasade Adamolekun   |      | Jamaika    | 2001        | 173     | ST       | 2019    |
| 4 | Yanara Aedo              |      | Chile      | 1993        | 154     | ST       | 2019    |

Anstatt einen festdefinierten Wert für alle Zeilen einzutragen, können wir die Werte auf der Basis bereits existierender Spalten berechnen.



Hier geben wir in einer neuen Spalte an, wie groß die jeweilige Person in Metern und nicht in Zentimetern ist:

```
df["groesse_meter"] = df["groesse"] / 100  
df.head(5)
```

|   | name                     | mannschaft | geburtsjahr | groesse | position | wm_jahr | groesse_meter |
|---|--------------------------|------------|-------------|---------|----------|---------|---------------|
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153     | ST       | 2019    | 1.53          |
| 3 | Olufolasade Adamolekun   | Jamaika    | 2001        | 173     | ST       | 2019    | 1.73          |
| 4 | Yanara Aedo              | Chile      | 1993        | 154     | ST       | 2019    | 1.54          |
| 5 | Lindsay Agnew            | Kanada     | 1995        | 175     | AB       | 2019    | 1.75          |
| 6 | Aitana                   | Spanien    | 1998        | 161     | MF       | 2019    | 1.61          |

Auch hätten wir gerne eine Spalte "alter2019", in der das Alter jeder Spielerin im Jahr der Weltmeisterschaft angegeben wird. Dafür berechnen wir die Differenz zwischen 2019 und dem Geburtsjahr der jeweiligen Spielerin:

```
df["alter2019"] = 2019 - df["geburtsjahr"]  
df.head(3)
```

|   | name                     | mannschaft | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|---|--------------------------|------------|-------------|---------|----------|---------|---------------|-----------|
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153     | ST       | 2019    | 1.53          | 30        |
| 3 | Olufolasade Adamolekun   | Jamaika    | 2001        | 173     | ST       | 2019    | 1.73          | 18        |
| 4 | Yanara Aedo              | Chile      | 1993        | 154     | ST       | 2019    | 1.54          | 26        |

## Datentabellen: Attribute

Datentabellen haben *Attribute (properties)*, die Eigenschaften der Datentabelle beschreiben. **ndim** gibt die Anzahl der Dimensionen an. So hat etwa unsere Datentabelle zwei Dimensionen (Spalten und Zeilen), während eine Serie nur eine Dimension hat.

```
df.ndim
```

2

Mit **shape** können Sie sich dann die Größe der verschiedenen Dimensionen (hier: Anzahl der Zeilen und Spalten) der Datentabelle anzeigen lassen:

```
df.shape
```

(550, 8)

**columns** gibt die Spaltennamen aus:

```
df.columns
```

```
Index(['name', 'mannschaft', 'geburtsjahr', 'groesse', 'position', 'wm_jahr',
       'groesse_meter', 'alter2019'],
      dtype='object')
```

**size** gibt an, wie viele Werte insgesamt vorhanden sind:

```
df.size
```

```
4400
```

## Datentabellen: Spalten auswählen



Wenn Sie aus einer Datentabelle eine individuelle Spalte auswählen möchten, können Sie dies tun, indem Sie den Spaltennamen in einfache eckige Klammern setzen:

```
df["geburtsjahr"].head(3)
```

```
2    1989
3    2001
4    1993
Name: geburtsjahr, dtype: int16
```

Das Ergebnis wird uns als Serie zurückgegeben.

Wenn wir stattdessen doppelte Klammern verwenden--die Spaltennamen also in einer Liste aufführen--bekommen wir eine Datentabelle:

```
df[["geburtsjahr"]].head(3)
```

### geburtsjahr

|   |      |
|---|------|
| 2 | 1989 |
| 3 | 2001 |
| 4 | 1993 |

In beiden Fällen können wir schon jetzt mit Hilfe von Aggregatfunktionen--welche bestimmte Eigenschaften unseres Datensatzes in einem Wert zusammenfassen--einfache Berechnungen durchführen:

| Funktion | Erklärung         | Beispiel                | Ergebnis |
|----------|-------------------|-------------------------|----------|
| min      | Der kleinste Wert | df["geburtsjahr"].min() | 1978     |

| Funktion | Erklärung  | Beispiel                  | Ergebnis    |
|----------|--|---------------------------|-------------|
| max      | Der größte Wert                                    | df["geburtstag"].max()    | 2003        |
| mean     | Das arithmetische Mittel (Mittelwert/Durchschnitt) | df["geburtstag"].mean()   | 1992.423... |
| median   | Der Median   | df["geburtstag"].median() | 1992.5      |
| count    | Anzahl der Werte in der Spalte                     | df["geburtstag"].count()  | 550         |
| sum      | Summe der Werte in der Spalte                      | df["geburtstag"].sum()    | 1095833     |
| std      | Standardabweichung der Werte in der Spalte         | df["geburtstag"].std()    | 4.127...    |

Wenn die Spaltennamen in einer Liste erscheinen, haben wir die Möglichkeit, noch weitere Spaltennamen angeben:

```
df[["name", "geburtstag", "groesse"]].head(3)
```

|   | name                     | geburtstag | groesse |
|---|--------------------------|------------|---------|
| 2 | Gabrielle Aboudi Onguene | 1989       | 153     |
| 3 | Olufolasade Adamolekun   | 2001       | 173     |
| 4 | Yanara Aedo              | 1993       | 154     |

Wenn wir jetzt eine der eben erwähnten Funktionen verwenden wollen, dann werden diese auf alle Spalten angewendet, mit denen diese Berechnungen durchgeführt werden können:

```
df[["name", "mannschaft", "geburtstag", "position", "wm_jahr",
     "groesse_meter"]].mean()
```

```
geburtstag      1992.423636
wm_jahr        2019.000000
groesse_meter   1.675800
dtype: float64
```

Ähnliches gilt für die Funktion **describe()**, mit der wir deskriptive Statistiken für unseren Datensatz erstellen können. Auch hier werden nur Spalten mit numerischen Werten berücksichtigt:

```
df.describe()
```

|              | geburtstag  | groesse    | wm_jahr | groesse_meter | alter2019  |
|--------------|-------------|------------|---------|---------------|------------|
| <b>count</b> | 550.000000  | 550.000000 | 550.0   | 550.000000    | 550.000000 |
| <b>mean</b>  | 1992.423636 | 167.580000 | 2019.0  | 1.675800      | 26.576364  |
| <b>std</b>   | 4.127806    | 6.484391   | 0.0     | 0.064844      | 4.127806   |
| <b>min</b>   | 1978.000000 | 148.000000 | 2019.0  | 1.480000      | 16.000000  |
| <b>25%</b>   | 1990.000000 | 163.000000 | 2019.0  | 1.630000      | 24.000000  |
| <b>50%</b>   | 1992.500000 | 168.000000 | 2019.0  | 1.680000      | 26.500000  |
| <b>75%</b>   | 1995.000000 | 172.000000 | 2019.0  | 1.720000      | 29.000000  |
| <b>max</b>   | 2003.000000 | 187.000000 | 2019.0  | 1.870000      | 41.000000  |

Wir können hier aber auch bestimmte Spalten oder nur eine Spalte auswählen:

```
df[["geburtsjahr", "groesse"]].describe()
```

|              | geburtsjahr | groesse    |
|--------------|-------------|------------|
| <b>count</b> | 550.000000  | 550.000000 |
| <b>mean</b>  | 1992.423636 | 167.580000 |
| <b>std</b>   | 4.127806    | 6.484391   |
| <b>min</b>   | 1978.000000 | 148.000000 |
| <b>25%</b>   | 1990.000000 | 163.000000 |
| <b>50%</b>   | 1992.500000 | 168.000000 |
| <b>75%</b>   | 1995.000000 | 172.000000 |
| <b>max</b>   | 2003.000000 | 187.000000 |

```
df["geburtsjahr"].describe()
```

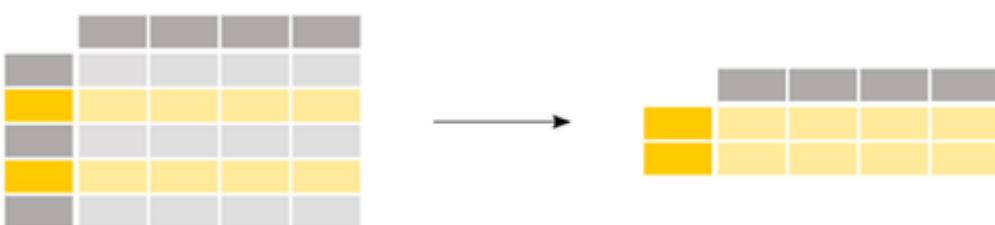
```
count      550.000000
mean     1992.423636
std       4.127806
min     1978.000000
25%    1990.000000
50%    1992.500000
75%    1995.000000
max     2003.000000
Name: geburtsjahr, dtype: float64
```

Statt der deskriptiven Statistiken können wir uns auch mit **value\_counts()** anzeigen lassen, wie oft jeder Wert in einer Spalte vorkommt:

```
df["geburtsjahr"].value_counts().head()
```

```
1993      64
1992      54
1990      48
1991      47
1994      42
Name: geburtsjahr, dtype: int64
```

## Datentabellen: Zeilen auswählen



Anstatt mit dem gesamten Datensatz zu arbeiten, können wir ihn nach verschiedenen Bedingungen filtern, d.h., wir arbeiten dann nur noch mit den Zeilen, welche diesen Bedingungen entsprechen.

Hier wählen wir die Zeilen aus, in denen der Wert für das Geburtsjahr größer oder gleich 1998 ist:

```
df[df["geburtsjahr"] >= 1998].head()
```

|    |  | name                   | mannschaft  | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|----|--|------------------------|-------------|-------------|---------|----------|---------|---------------|-----------|
| 3  |  | Olufolasade Adamolekun | Jamaika     | 2001        | 173     | ST       | 2019    | 1.73          | 18        |
| 6  |  | Aitana                 | Spanien     | 1998        | 161     | MF       | 2019    | 1.61          | 21        |
| 7  |  | Rasheedat Ajibade      | Nigeria     | 1999        | 167     | MF       | 2019    | 1.67          | 20        |
| 30 |  | Rosario Balmaceda      | Chile       | 1999        | 163     | ST       | 2019    | 1.63          | 20        |
| 40 |  | Lorena Benítez         | Argentinien | 1998        | 157     | MF       | 2019    | 1.57          | 21        |

Die Vergleichsoperatoren sind dieselben, die Sie bspw. schon von *if*-Bedingungen kennen:

| Operator | Bedeutung      |
|----------|----------------|
| >        | größer         |
| >=       | größer/gleich  |
| <        | kleiner        |
| <=       | kleiner/gleich |
| ==       | gleich         |
| !=       | ungleich       |

Die ursprüngliche Datentabelle wird dabei nicht überschrieben. Wenn wir die gefilterte Version aber später wiederverwenden möchten, sollten wir ihn unter einem anderen Namen speichern. Dann stehen uns beide Versionen zur Verfügung:

```
df_gefiltert = df[df["geburtsjahr"] >= 1998]
df_gefiltert.head(5)
```

|    |  | name                   | mannschaft  | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|----|--|------------------------|-------------|-------------|---------|----------|---------|---------------|-----------|
| 3  |  | Olufolasade Adamolekun | Jamaika     | 2001        | 173     | ST       | 2019    | 1.73          | 18        |
| 6  |  | Aitana                 | Spanien     | 1998        | 161     | MF       | 2019    | 1.61          | 21        |
| 7  |  | Rasheedat Ajibade      | Nigeria     | 1999        | 167     | MF       | 2019    | 1.67          | 20        |
| 30 |  | Rosario Balmaceda      | Chile       | 1999        | 163     | ST       | 2019    | 1.63          | 20        |
| 40 |  | Lorena Benítez         | Argentinien | 1998        | 157     | MF       | 2019    | 1.57          | 21        |

Im vorherigen Beispiel haben wir nur eine Bedingung verwendet. Wir können aber auch mehrere einsetzen. So suchen wir jetzt nach allen Spielerinnen der deutschen Mannschaft, die 1998 oder später geboren wurden.

```
df_gefiltert = df[(df["geburtsjahr"] >= 1998) & (df["mannschaft"] == "Deutschland")]
```

```
df_gefiltert.head(5)
```

|     |  | name          | mannschaft  | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|-----|--|---------------|-------------|-------------|---------|----------|---------|---------------|-----------|
| 68  |  | Klara Bühl    | Deutschland | 2000        | 172     | ST       | 2019    | 1.72          | 19        |
| 181 |  | Giulia Gwinn  | Deutschland | 1999        | 170     | MF       | 2019    | 1.70          | 20        |
| 368 |  | Lena Oberdorf | Deutschland | 2001        | 174     | MF       | 2019    | 1.74          | 18        |

Beachten Sie hier, dass jede einzelne Bedingung in Klammern gesetzt werden muss:

```
df[(df["geburtsjahr"] >= 1998) & (df["mannschaft"] == "Deutschland")]
```

Ein weiterer Punkt: In *pandas* werden logische UND-Verknüpfungen mit **&** vorgenommen. ODER-Verknüpfungen erfolgen mit **|** (senkrechter Strich, *Pipe*). Im folgenden Beispiel suchen wir nach Spielerinnen, die entweder in der deutschen Mannschaft spielten **oder** 1998 oder danach geboren wurden:

```
df[(df["geburtsjahr"] >= 1998) | (df["mannschaft"] == "Deutschland")]
```

## Spalten und Zahlen auswählen



Mithilfe des **loc**-Attributs können wir Zeilen über ihre Indexwerte selektieren. Sehen wir uns noch einmal unsere Datentabelle an:

```
df.head()
```

|   |  | name                     | mannschaft | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|---|--|--------------------------|------------|-------------|---------|----------|---------|---------------|-----------|
| 2 |  | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153     | ST       | 2019    | 1.53          | 30        |
| 3 |  | Olufolasade Adamolekun   | Jamaika    | 2001        | 173     | ST       | 2019    | 1.73          | 18        |
| 4 |  | Yanara Aedo              | Chile      | 1993        | 154     | ST       | 2019    | 1.54          | 26        |
| 5 |  | Lindsay Agnew            | Kanada     | 1995        | 175     | AB       | 2019    | 1.75          | 24        |
| 6 |  | Aitana                   | Spanien    | 1998        | 161     | MF       | 2019    | 1.61          | 21        |

Wenn wir jetzt die erste Zeile auswählen möchten, tun wir das bei *loc* nicht über die Zeilenposition (erste Zeile = 0), sondern über den Indexwert, der am linken Rand angezeigt wird (2):

```
df.loc[2]
```

```

name          Gabrielle Aboudi Onguene
mannschaft      Kamerun
geburtsjahr       1989
groesse          153
position           ST
wm_jahr          2019
groesse_meter     1.53
alter2019         30
Name: 2, dtype: object

```

Genauso können wir auch mehrere Zeilen und bestimmte Spalten auswählen. Hier etwa die erste Zeile und nur die Spalte "name":

```
df.loc[2, "name"]
```

```
'Gabrielle Aboudi Onguene'
```

Und hier die Zeilen 2-4 sowie die Spalten "name" bis "geburtsjahr". Beachten Sie, dass der durch den zweiten Zeilenindex (4) bezeichnete Wert--anders als etwa bei Zeichenketten oder Listen--noch mit eingeschlossen wird:

```
df.loc[2:4, "name":"geburtsjahr"]
```

|   | name                     | mannschaft | geburtsjahr |
|---|--------------------------|------------|-------------|
| 2 | Gabrielle Aboudi Onguene | Kamerun    | 1989        |
| 3 | Olufolasade Adamolekun   | Jamaika    | 2001        |
| 4 | Yanara Aedo              | Chile      | 1993        |

Mit **loc** kann man auch Bedingungen zur Datenfilterung verwenden:

```
df.loc[df["geburtsjahr"] >= 1998].head()
```

|    | name                   | mannschaft  | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|----|------------------------|-------------|-------------|---------|----------|---------|---------------|-----------|
| 3  | Olufolasade Adamolekun | Jamaika     | 2001        | 173     | ST       | 2019    | 1.73          | 18        |
| 6  | Aitana                 | Spanien     | 1998        | 161     | MF       | 2019    | 1.61          | 21        |
| 7  | Rasheedat Ajibade      | Nigeria     | 1999        | 167     | MF       | 2019    | 1.67          | 20        |
| 30 | Rosario Balmaceda      | Chile       | 1999        | 163     | ST       | 2019    | 1.63          | 20        |
| 40 | Lorena Benítez         | Argentinien | 1998        | 157     | MF       | 2019    | 1.57          | 21        |

Mit mehreren Bedingungen sieht es so aus:

```
df.loc[(df["geburtsjahr"] >= 1998) & (df["mannschaft"] == "Deutschland")].head()
```

|     | name         | mannschaft  | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|-----|--------------|-------------|-------------|---------|----------|---------|---------------|-----------|
| 68  | Klara Bühl   | Deutschland | 2000        | 172     | ST       | 2019    | 1.72          | 19        |
| 181 | Giulia Gwinn | Deutschland | 1999        | 170     | MF       | 2019    | 1.70          | 20        |

|            | <b>name</b>   | <b>mannschaft</b> | <b>geburtsjahr</b> | <b>groesse</b> | <b>position</b> | <b>wm_jahr</b> | <b>groesse_meter</b> | <b>alter2019</b> |
|------------|---------------|-------------------|--------------------|----------------|-----------------|----------------|----------------------|------------------|
| <b>368</b> | Lena Oberdorf | Deutschland       | 2001               | 174            | MF              | 2019           | 1.74                 | 18               |

Wenn wir nur einen Teil der Spalten ausgeben wollen, haben wir zwei Möglichkeiten. Wir können einerseits **loc** nutzen und diesem eine Liste `["name", "geburtsjahr"]` mit den gewünschten Spalten übergeben.

```
df.loc[(df["geburtsjahr"] >= 1998) & (df["mannschaft"] == "Deutschland"), \
       ["name", "geburtsjahr"]].head()
```

|            | <b>name</b>   | <b>geburtsjahr</b> |
|------------|---------------|--------------------|
| <b>68</b>  | Klara Bühl    | 2000               |
| <b>181</b> | Giulia Gwinn  | 1999               |
| <b>368</b> | Lena Oberdorf | 2001               |

Andererseits können wir **loc** nur zur Filterung der Zeilen verwenden und danach aus der resultierenden Datentabelle die benötigten Spalten auswählen:

```
dataframe = df[["name", "geburtsjahr"]].loc[(df["geburtsjahr"] >= 1998) & \
                                              (df["mannschaft"] == "Deutschland"), ].head()
```

|            | <b>name</b>   | <b>geburtsjahr</b> |
|------------|---------------|--------------------|
| <b>68</b>  | Klara Bühl    | 2000               |
| <b>181</b> | Giulia Gwinn  | 1999               |
| <b>368</b> | Lena Oberdorf | 2001               |

## **loc und iloc**

Schauen wir uns noch einmal die Datentabelle aus dem vorherigen Beispiel an:

```
dataframe
```

|            | <b>name</b>   | <b>geburtsjahr</b> |
|------------|---------------|--------------------|
| <b>68</b>  | Klara Bühl    | 2000               |
| <b>181</b> | Giulia Gwinn  | 1999               |
| <b>368</b> | Lena Oberdorf | 2001               |

Sie können sehen, dass wir beim Zeilenindex keine aufeinanderfolgenden Werte haben. Das liegt daran, dass viele Zeilen ausgetiltert und die Indexwerte aus der ursprünglichen Datentabelle übernommen wurden.

Um es noch einmal zu betonen: Wenn wir jetzt mit **loc** auf die erste Zeile in der gefilterten Datentabelle

zugreifen wollen, dann können wir das nicht, wie vielleicht erwartet, über den Indexwert *0* erreichen, denn das führt zu einem *KeyError*:

```
dataframe.loc[0]
```

```
-----
KeyError                                                 Traceback (most recent call last)
c:\users\kurs\appdata\local\programs\python\python39\lib\site-packages\pandas\core\indexes\base.py in get_loc(self, key, method, tolerance)
      3620
-> 3621         return self._engine.get_loc(casted_key)
      3622     except KeyError as err:
c:\users\kurs\appdata\local\programs\python\python39\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()
c:\users\kurs\appdata\local\programs\python\python39\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()
pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTable.get_item()
pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTable.get_item()

KeyError: 0
```

The above exception was the direct cause of the following exception:

```
KeyError                                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_1972\2928089500.py in <cell line: 1>()
----> 1 dataframe.loc[0]

c:\users\kurs\appdata\local\programs\python\python39\lib\site-packages\pandas\core\indexing.py in __getitem__(self, key)
      965
      966         maybe_callable = com.apply_if_callable(key, self.obj)
--> 967         return self._getitem_axis(maybe_callable, axis=axis)
      968
      969     def _is_scalar_access(self, key: tuple):
c:\users\kurs\appdata\local\programs\python\python39\lib\site-packages\pandas\core\indexing.py in _getitem_axis(self, key, axis)
     1200         # fall thru to straight lookup
     1201         self._validate_key(key, axis)
-> 1202         return self._get_label(key, axis=axis)
     1203
     1204     def _get_slice_axis(self, slice_obj: slice, axis: int):
c:\users\kurs\appdata\local\programs\python\python39\lib\site-packages\pandas\core\indexing.py in _get_label(self, label, axis)
     1151     def _get_label(self, label, axis: int):
     1152         # GH#5667 this will fail if the label is not present in the axis.
-> 1153         return self.obj.xs(label, axis=axis)
     1154
     1155     def _handle_lowerdim_multi_index_axis0(self, tup: tuple):
c:\users\kurs\appdata\local\programs\python\python39\lib\site-packages\pandas\core\generic.py in xs(self, key, axis, level, drop_level)
     3862             new_index = index[loc]
     3863         else:
-> 3864             loc = index.get_loc(key)
     3865
     3866             if isinstance(loc, np.ndarray):
```

```
c:\users\kurs\appdata\local\programs\python\python39\lib\site-packages\pandas\core
\indexes\base.py in get_loc(self, key, method, tolerance)
    3621         return self._engine.get_loc(casted_key)
    3622     except KeyError as err:
-> 3623         raise KeyError(key) from err
    3624     except TypeError:
    3625         # If we have a listlike key, _check_indexing_error will ra
ise

KeyError: 0
```

Stattdessen müssen wir den im Datensatz angegebenen Wert 68 verwenden.

```
dataframe.loc[68]
```

```
name          Klara Bühl
geburtsjahr      2000
Name: 68, dtype: object
```

Es gibt allerdings noch ein zweites Attribut, das wir verwenden können, und zwar **iloc**. Damit bekommen wir über die numerische Position einer Zeile (oder Spalte) in der gefilterten Datentabelle Zugriff auf diese:

```
dataframe.iloc[0]
```

```
name          Klara Bühl
geburtsjahr      2000
Name: 68, dtype: object
```

```
dataframe.iloc[0, 1]
```

```
2000
```

## Zeilen auswählen mit **.isin()**

Alternativ können wir auch die Methode **isin()** zur Zeilenauswahl verwenden. Damit können wir jede Zeile überprüfen, ob sie in einer oder mehrerer Spalten einen oder mehr vorgegebene Werte enthält. Als Ergebnis erhalten wir dann eine Serie, in der für jede Zeile entweder ein *True* (wenn der Wert gefunden wurde) oder *False* (wenn er nicht gefunden wurde) steht. Im folgenden Beispiel wollen wir sehen, in welchen Zeilen der Wert "Kamerun" in der Spalte "mannschaft" vorkommt.

```
serie = df["mannschaft"].isin(["Kamerun"])
serie.head()
```

```
2      True
3     False
4     False
5     False
6     False
Name: mannschaft, dtype: bool
```

Diese Serie können wir jetzt verwenden, um die Datentabelle zu filtern. Dabei werden nur die Zeilen ausgewählt, für die in der Serie der Wert *True* steht:

```
serie = df["mannschaft"].isin(["Kamerun"])
df[serie].head()
```

|     | name                     | mannschaft | geburtjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|-----|--------------------------|------------|------------|---------|----------|---------|---------------|-----------|
| 2   | Gabrielle Aboudi Onguene | Kamerun    | 1989       | 153     | ST       | 2019    | 1.53          | 30        |
| 8   | Henriette Akaba          | Kamerun    | 1992       | 160     | ST       | 2019    | 1.60          | 27        |
| 28  | Aurelle Awona            | Kamerun    | 1993       | 172     | AB       | 2019    | 1.72          | 26        |
| 128 | Augustine Ejangue        | Kamerun    | 1989       | 158     | AB       | 2019    | 1.58          | 30        |
| 133 | Gaëlle Enganamouit       | Kamerun    | 1992       | 171     | MF       | 2019    | 1.71          | 27        |

Wenn wir die Auswahl verneinen wollen, wir also alle Zeilen haben möchten, in denen der gewählte Wert nicht in der Spalte vorkommt, können wir den Operator `~` voranstellen:

```
serie = ~df["mannschaft"].isin(["Kamerun"])
df[serie].head()
```

|   | name                   | mannschaft | geburtjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|---|------------------------|------------|------------|---------|----------|---------|---------------|-----------|
| 3 | Olufolasade Adamolekun | Jamaika    | 2001       | 173     | ST       | 2019    | 1.73          | 18        |
| 4 | Yanara Aedo            | Chile      | 1993       | 154     | ST       | 2019    | 1.54          | 26        |
| 5 | Lindsay Agnew          | Kanada     | 1995       | 175     | AB       | 2019    | 1.75          | 24        |
| 6 | Aitana                 | Spanien    | 1998       | 161     | MF       | 2019    | 1.61          | 21        |
| 7 | Rasheedat Ajibade      | Nigeria    | 1999       | 167     | MF       | 2019    | 1.67          | 20        |

## Wertegrenzen

Wir können mit der Methode `clip()` Wertegrenzen einrichten, d.h., dass Werte, die unter oder über einem bestimmten Wert liegen, auf diesen Grenzwert gesetzt werden. Wenn z.B. eine untere Wertegrenze von 15 festgesetzt wird und man hat im Datensatz den Wert 5, dann wird dieser zu 15 geändert. In unserem Beispieldatensatz könnten wir beispielsweise die Größe aller Spielerinnen, die kleiner als 1,60 Meter oder größer als 1,70 Meter sind, auf diese beiden Werte ändern.

Originalwerte:

```
df_beispiel = df[["groesse"]]
df_beispiel.head()
```

|   | groesse |
|---|---------|
| 2 | 153     |
| 3 | 173     |

**groesse**

|          |     |
|----------|-----|
| <b>4</b> | 154 |
| <b>5</b> | 175 |
| <b>6</b> | 161 |

Untere Wertegrenze:

```
df_beispiel = df[["groesse"]].clip(lower=160)
df_beispiel.head()
```

**groesse**

|          |     |
|----------|-----|
| <b>2</b> | 160 |
| <b>3</b> | 173 |
| <b>4</b> | 160 |
| <b>5</b> | 175 |
| <b>6</b> | 161 |

Obere Wertgrenze:

```
# Obere Wertgrenze
df_beispiel = df[["groesse"]].clip(upper=170)
df_beispiel.head()
```

**groesse**

|          |     |
|----------|-----|
| <b>2</b> | 153 |
| <b>3</b> | 170 |
| <b>4</b> | 154 |
| <b>5</b> | 170 |
| <b>6</b> | 161 |

Untere und obere Wertgrenze:

```
df_beispiel = df[["groesse"]].clip(160,170)
df_beispiel.head()
```

**groesse**

|          |     |
|----------|-----|
| <b>2</b> | 160 |
| <b>3</b> | 170 |
| <b>4</b> | 160 |
| <b>5</b> | 170 |
| <b>6</b> | 161 |

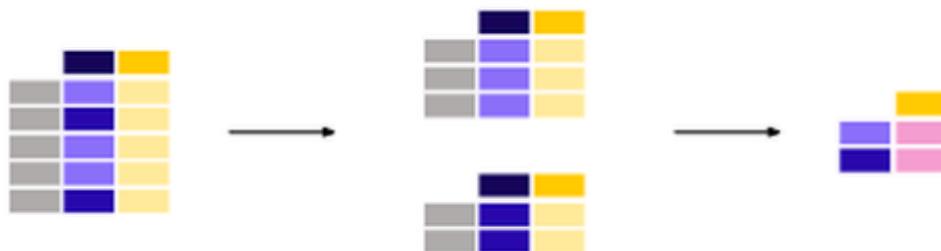
Wir können aber auch auf anderem Weg Wertegrenzen in unserem Datensatz erstellen. Eine Möglichkeit besteht etwa darin, mit dem *loc*-Attribut Zeilen auszuwählen, bei denen in einer Spalte der Grenzwert unter- oder überschritten wird, und dann den jeweiligen Wert anzupassen:

```
df_beispiel.loc[df["groesse"] < 160, "groesse"] = 160  
df_beispiel.loc[df["groesse"] > 170, "groesse"] = 170  
df_beispiel.head()
```

|   | groesse |
|---|---------|
| 2 | 160     |
| 3 | 170     |
| 4 | 160     |
| 5 | 170     |
| 6 | 161     |

## Daten gruppieren

*pandas* bietet die Möglichkeit, mit der Methode **groupby()** Daten zu gruppieren und dann Aggregatfunktionen auf jede dieser Gruppen anzuwenden.



Wir können etwa unseren Beispieldatensatz nach Mannschaft gruppieren und dann für jede Mannschaft Werte wie etwa Durchschnittsgröße oder das höchste Geburtsjahr (= Geburtsjahr der jüngsten Spielerin) ausgeben lassen:

```
df.groupby("mannschaft") ["groesse"].mean().head(3)
```

```
mannschaft  
Argentinien    165.695652  
Australien     168.434783  
Brasilien      167.304348  
Name: groesse, dtype: float64
```

```
df.groupby("mannschaft") ["geburtstag"].max().head(3)
```

```
mannschaft  
Argentinien    2002  
Australien     2003  
Brasilien      1998  
Name: geburtstag, dtype: int16
```

Wenn wir uns das Ergebnis als Datentabelle ausgeben lassen, sehen wir, dass wir keinen numerischen Index mehr haben. Stattdessen werden jetzt die Werte aus der Spalte, nach der gruppiert wurde ("mannschaft"), als Index verwendet.

```
df.groupby("mannschaft") [["geburtsjahr"]].max().head(3)
```

### geburtsjahr

#### mannschaft

|                    |      |
|--------------------|------|
| <b>Argentinien</b> | 2002 |
| <b>Australien</b>  | 2003 |
| <b>Brasilien</b>   | 1998 |

Da wir die Spalte "mannschaft" als normale Datenspalte brauchen, verwenden wir `reset_index()`, um stattdessen wieder einen numerischen Index zu bekommen:

```
dataframe = df.groupby("mannschaft") [["geburtsjahr"]].max().head(3)
dataframe.reset_index()
```

### mannschaft geburtsjahr

|          |             |      |
|----------|-------------|------|
| <b>0</b> | Argentinien | 2002 |
| <b>1</b> | Australien  | 2003 |
| <b>2</b> | Brasilien   | 1998 |

Es ist auch möglich, nach mehreren Spalten zu gruppieren und mehrere Spalten zu aggregieren. Hier gruppieren wir erst nach Mannschaft und dann noch nach Position. Für jede Kombination aus diesen beiden Spalten berechnen wir sowohl die Durchschnittsgröße als auch das durchschnittliche Geburtsjahr.

```
dataframe = df.groupby(["mannschaft", "position"]) [["groesse", "geburtsjahr"]].mean().head(8)
dataframe
```

### groesse geburtsjahr

#### mannschaft position

|                    |           |            |             |
|--------------------|-----------|------------|-------------|
| <b>Argentinien</b> | <b>AB</b> | 163.750000 | 1992.375000 |
|                    | <b>MF</b> | 162.250000 | 1991.625000 |
|                    | <b>ST</b> | 168.750000 | 1991.500000 |
|                    | <b>TW</b> | 176.000000 | 1990.666667 |
| <b>Australien</b>  | <b>AB</b> | 168.000000 | 1994.714286 |
|                    | <b>MF</b> | 163.333333 | 1990.833333 |
|                    | <b>ST</b> | 169.428571 | 1993.714286 |
|                    | <b>TW</b> | 177.333333 | 1993.000000 |

# Daten sortieren

Mit `sort_values()` bietet pandas eine Funktion zum Sortieren von Datentabellen. Hier sortieren wir die Spielerinnen in aufsteigender Reihenfolge nach Größe:

```
df.sort_values("groesse").head(6)
```

|     | name                     | mannschaft | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|-----|--------------------------|------------|-------------|---------|----------|---------|---------------|-----------|
| 175 | Javiera Grez             | Chile      | 2000        | 148     | ST       | 2019    | 1.48          | 19        |
| 475 | Orathai Srimanee         | Thailand   | 1988        | 151     | ST       | 2019    | 1.51          | 31        |
| 419 | María José Rojas         | Chile      | 1987        | 153     | ST       | 2019    | 1.53          | 32        |
| 340 | Yuka Momiki              | Japan      | 1996        | 153     | MF       | 2019    | 1.53          | 23        |
| 2   | Gabrielle Aboudi Onguene | Kamerun    | 1989        | 153     | ST       | 2019    | 1.53          | 30        |
| 395 | Warunee Phetwiset        | Thailand   | 1990        | 153     | AB       | 2019    | 1.53          | 29        |

Falls wir zuerst die größten Spielerinnen sehen möchten, können wir den Datensatz in absteigender Reihenfolge sortieren. `sort_values()` bietet dazu den Parameter *ascending* (aufsteigend), dessen Wert wir auf "False" setzen müssen.

```
df.sort_values("groesse", ascending=False).head(6)
```

|     | name              | mannschaft | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|-----|-------------------|------------|-------------|---------|----------|---------|---------------|-----------|
| 414 | Wendie Renard     | Frankreich | 1990        | 187     | AB       | 2019    | 1.87          | 29        |
| 163 | Emily Gielnik     | Australien | 1992        | 183     | ST       | 2019    | 1.83          | 27        |
| 325 | Jessica McDonald  | USA        | 1988        | 183     | ST       | 2019    | 1.83          | 31        |
| 131 | Christiane Endler | Chile      | 1991        | 182     | TW       | 2019    | 1.82          | 28        |
| 388 | Shimeng Peng      | China      | 1998        | 182     | TW       | 2019    | 1.82          | 21        |
| 33  | Karen Bardsley    | England    | 1984        | 182     | TW       | 2019    | 1.82          | 35        |

Schließlich können wir auch nach mehreren Spalten sortieren. Wir übergeben dazu sowohl die Spaltennamen als auch Sortierrichtungen in jeweils einer Liste an die Funktion. Im folgenden Beispiel wird unser DataFram erst in aufsteigender Reihenfolge nach "groesse" und dann--wenn für eine Größenangabe mehrere Werte vorliegen--in absteigender Reihenfolge nach "geburtsjahr" sortiert.

```
df.sort_values(["groesse", "geburtsjahr"], ascending=[False, True]).head(6)
```

|     | name             | mannschaft | geburtsjahr | groesse | position | wm_jahr | groesse_meter | alter2019 |
|-----|------------------|------------|-------------|---------|----------|---------|---------------|-----------|
| 414 | Wendie Renard    | Frankreich | 1990        | 187     | AB       | 2019    | 1.87          | 29        |
| 325 | Jessica McDonald | USA        | 1988        | 183     | ST       | 2019    | 1.83          | 31        |

|            | <b>name</b>       | <b>mannschaft</b> | <b>geburtsjahr</b> | <b>groesse</b> | <b>position</b> | <b>wm_jahr</b> | <b>groesse_meter</b> | <b>alter2019</b> |
|------------|-------------------|-------------------|--------------------|----------------|-----------------|----------------|----------------------|------------------|
| <b>163</b> | Emily Gielnik     | Australien        | 1992               | 183            | ST              | 2019           | 1.83                 | 27               |
| <b>33</b>  | Karen Bardsley    | England           | 1984               | 182            | TW              | 2019           | 1.82                 | 35               |
| <b>112</b> | Anouk Dekker      | Niederlande       | 1986               | 182            | AB              | 2019           | 1.82                 | 33               |
| <b>131</b> | Christiane Endler | Chile             | 1991               | 182            | TW              | 2019           | 1.82                 | 28               |

## Verkettung

Die hier gezeigten Arbeitsschritte lassen sich in einer Zeile miteinander kombinieren, indem man sie aneinanderhängt. Die Funktionen werden dabei von links nach rechts ausgeführt. Hier ein Beispiel:

```
df.groupby("mannschaft") [["groesse"]].mean().sort_values("groesse",
ascending=False).head(5)
```

| <b>groesse</b>                |
|-------------------------------|
| <b>mannschaft</b>             |
| <b>Schweden</b> 172.043478    |
| <b>Deutschland</b> 170.608696 |
| <b>USA</b> 170.565217         |
| <b>China</b> 170.434783       |
| <b>Niederlande</b> 170.000000 |

## Literatur

- Dokumentation zu *pandas*: <https://pandas.pydata.org/docs/>
- Cheat Sheet: [https://pandas.pydata.org/Pandas\\_Cheat\\_Sheet.pdf](https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf)
- McKinney, Wes. *Datenanalyse mit Python*. Verschiedene Auflagen, auch auf Englisch erhältlich.
  - Sehr gute Übersicht zu Konzept und Implementierung.
  - McKinney begann 2008 mit der Entwicklung von *pandas* und leitet heute noch die Weiterentwicklung.

# Datum und Zeit

## Arbeiten mit Datums- und Zeitangaben

Das Arbeiten mit Datums- und Zeitangaben ist komplizierter, als es auf den ersten Blick aussieht.

Das hat mehrere Gründe:

### Verschiedene Kalender

- Wir verwenden heute normalerweise den gregorianischen Kalender. Dessen Einführung, die 1582 begann, zog sich aber über mehrere Jahrhunderte hin.
- Selbst heute werden--meist zu religiösen Zwecken--noch andere Kalender verwendet, u.a. der julianische Kalender, der von Julius Cäsar eingeführte Vorgänger des gregorianischen Kalenders.

### Kalenderjahr vs. Sonnenjahr und Schaltjahre

- Sowohl der julianische als auch der gregorianische Kalender sind Solarkalender, d.h., die Länge eines Jahres richtet sich danach, wie lange die Erde benötigt, um einmal um die Sonne zu kreisen (ein Sonnenjahr oder tropisches Jahr).
- Der Julianische Kalender berechnete dafür 365,25 Tage, was aber etwa 11 Minuten zu lange war. Dadurch liefen das Kalenderjahr und das tropische Jahr im Laufe der Zeit auseinander. Um das zu korrigieren, wurde das Jahr für den gregorianischen Kalender etwas verkürzt.
- Eine Folge davon war, dass man nicht mehr pauschal alle vier Jahre ein Schaltjahr verwenden konnte, um den Längenunterschied von Kalenderjahr und Sonnenjahr auszugleichen.
- Stattdessen gelten folgende Regeln:
  1. Wenn eine Jahreszahl durch 4 teilbar ist, ist es *ein* Schaltjahr.
  2. Aber wenn eine Jahreszahl durch 100 teilbar ist, ist es *kein* Schaltjahr.
  3. Aber wenn eine Jahreszahl durch 400 teilbar ist, ist es *ein* Schaltjahr.
- Zusätzlich werden in unregelmäßigen Abständen auch Schaltsekunden eingefügt.

### Verschiedene Zeitzonen

- Es gibt 24 Zeitzonen, die sich in den seltensten Fällen nach der auf Längengraden basierenden Einteilung halten und oft eher staatlichen Grenzen oder politischen Überlegungen folgen. Die "Lower 48" der USA sind in vier Zeitzonen eingeteilt, Russland wechselte in den letzten Jahren zwischen 9 und 11 und China hat, trotz seiner mit den USA vergleichbaren Größe, nur eine Zeitzone.
- Manche Gebiete, wie das im östlichen Kanada gelegene Neufundland, sind keiner dieser Zeitzonen zugeordnet. Es liegt eine halbe Stunde vor der nordamerikanischen "Eastern Timezone".

# Winter und Sommerzeit

- Selbst innerhalb eines Landes kann es passieren, dass manche Gebiete Sommerzeit befolgen, andere nicht (z.B. USA und Kanada).
- Manche Länder verwenden Sommerzeit für ein paar Jahre, dann wieder nicht (z.B. Russland).
- In manchen Ländern beginnt und endet die Sommerzeit an anderen Tagen als bei uns (z.B. seit 2007 in den USA).

# Computer und Zeit

Ein wichtiges Konzept ist hier die sog. Unixzeit, die viele Systeme--u.a. auch Pythonpakete--zur Zeitbestimmung verwenden. Sie misst, wie viele Sekunden seit dem Beginn der Unix-Epoche am 1.1.1970 um 0:00 Uhr UTC (*Coordinated Universal Time* oder koordinierte Weltzeit, Nachfolger der GMT [Greenwich Meridian Time]) vergangen sind. Die Unixzeit wird dann als *Unix timestamp* gespeichert.

## Das "Jahr 2038"-Problem

Wer alt genug ist, kann sich noch an das "Jahr 2000"-Problem (*Millenium-Bug*, *Y2K-Bug*) erinnern: Um Speicherplatz zu sparen, speicherten Computersysteme Jahreszahlen oft nur zweistellig, denn "Wenn das ein Problem wird, da gibt es sicher schon andere Plattformen..."

Die Systeme waren aber wesentlich langlebiger als erwartet und man stand dann vor dem Problem, dass Computer nicht mit dem Übergang von 1999 zu 2000 umgehen könnten. Viel Arbeit und Geld haben dann dafür gesorgt, dass es letztendlich kaum zu Ausfällen kam.

Das "Jahr 2038"-Problem liegt nun darin begründet, dass die Unixzeit ursprünglich in einer 32-Bit Zahl mit Vorzeichen gespeichert wurde. Ein Bit fällt für das Vorzeichen weg, deshalb können noch  $2^{31}$ --oder 2.147.483.648--Werte dargestellt werden.

Deshalb konnten alte Systeme nur Zeitangaben bis zum 19.1.2038 um 16:14:08 Uhr handhaben (eben Unixzeit 2147483648), da danach der Zähler auf den negativen Wert -2.147.483.648 umspringen würde (d.h., den 13.12.1901 um 21:45:52 Uhr). Die Lösung in neueren Systemen liegt darin, dass die Unixzeit nun als 64-Bit Zahl gespeichert wird und dadurch ein viel größerer Wertebereich abgebildet werden kann.

## Python und Zeit: `time`

Auch das Python-Modul **time** verwendet die Unixzeit. Um es benutzen zu können, müssen wir es zuerst importieren:

```
import time
```

Wir können uns jetzt die aktuelle Unixzeit anzeigen lassen:

```
unixzeit = time.time()  
unixzeit
```

1606060680.9670367

Die Nachkommastellen können wir ignorieren.

## ***timestamp und struct\_time***

Wie kann man das Ganze für Menschen lesbarer machen? Eine Möglichkeit besteht darin, die jeweilige Unixzeit in ein **struct\_time**-Objekt umzuwandeln. Es handelt sich dabei um ein Tuple mit neun benannten Elementen. Sie können dabei entweder über deren Indexwert oder deren Namen auf die einzelnen Elemente zugreifen:

| Index | Attribut | Mögliche Werte | Kommentar  |
|-------|----------|----------------|--|
| 0     | tm_year  |                |  |
| 1     | tm_mon   | 1 bis 12       |  |
| 2     | tm_mday  | 1 bis 31       |  |
| 3     | tm_hour  | 0 bis 23       |  |
| 4     | tm_min   | 0 bis 59       |  |
| 5     | tm_sec   | 0 bis 61       | Zusätzliche Werte für Schaltsekunden                     |
| 6     | tm_wday  | 0 bis 6        | Montag ist 0, Dienstag ist 1 usw.                        |
| 7     | tm_yday  | 1 bis 366      | Welcher Tag im Jahr                                      |
| 8     | tm_isdst | 0, 1, -1       | 0 = Keine Sommerzeit, 1 = Sommerzeit, -1 = nicht bekannt |

## ***localtime vs. gmtime***

Wir können jetzt mithilfe zweier Funktionen einen *timestamp* in *struct\_time* umwandeln. Warum gibt es zwei Funktionen?

- *localtime()* gibt die lokale Zeit, also die Uhrzeit, die momentan auf der Uhr Ihres Computers angezeigt wird, zurück
- *gmtime()* gibt die UTC zurück

```
lokalzeit_struct_time = time.localtime(unixzeit)
print(lokalzeit_struct_time)
```

```
time.struct_time(tm_year=2020, tm_mon=11, tm_mday=22, tm_hour=16, tm_min=58, tm_sec=0, tm_wday=6, tm_yday=327, tm_isdst=0)
```

```
utc_struct_time = time.gmtime(unixzeit)
print(utc_struct_time)
```

```
time.struct_time(tm_year=2020, tm_mon=11, tm_mday=22, tm_hour=15, tm_min=58, tm_sec=0, tm_wday=6, tm_yday=327, tm_isdst=0)
```

## ***mktime() vs. calendar.timegm()***

Die Konvertierung funktioniert auch in die andere Richtung, d.h. von *struct\_time* zu einem *timestamp*:

- *mktime()* wird verwendet, wenn *struct\_time* eine lokale Zeit angibt
- *timegm()* aus dem Paket *calendar* (dazu mehr im nächsten Video) wird im Fall der UTC verwendet

```
timestamp_aus_lokalzeit = time.mktime(lokalzeit_struct_time)
print(timestamp_aus_lokalzeit)
```

```
1606060680.0
```

```
import calendar
timestamp_aus_utc = calendar.timegm(utc_struct_time)
print(timestamp_aus_lokalzeit)
```

```
1606060680.0
```

## *asctime()*

Mit *asctime()* können wir ein *struct\_time*-Objekt in eine leichter lesbare Zeichenkette, oder String, umwandeln. Wir haben dabei allerdings keine Möglichkeit, die Formatierung zu beeinflussen:

```
string_struct_time = time.asctime(lokalzeit_struct_time)
string_struct_time
```

```
'Sun Nov 22 16:58:00 2020'
```

## *ctime()*

Mit *ctime()* können Sie das Gleiche mit einem *timestamp* machen:

```
string_struct_time = time.ctime(unixzeit)
string_struct_time
```

```
'Sun Nov 22 16:58:00 2020'
```

Was können wir nun machen, wenn wir Datum und Zeit in einem anderen Format ausgeben möchten?

## *strftime()*

Sie kennen möglicherweise schon die Funktion *format()*, mit der man Zeichenketten formatieren kann:

```
name1 = "Karin"
name2 = "Klaus"
zeichenkette = "Hallo {}, hallo {}!".format(name1, name2)
print(zeichenkette)
```

```
Haloo Karin, hallo Klaus!
```

Die Funktion `strftime()` funktioniert in ähnlicher Weise. Sie verwendet ebenfalls Platzhalter, bestehend aus einem Prozentzeichen und einem Zeichen (meist einem Buchstaben), die zusammen für einen bestimmten Teil einer Datums- oder Zeitangabe sowie für eine bestimmte Formatierung stehen:

| Platzhalter | Erklärung  | Bereich   |
|-------------|--|-----------|
| %a          | Lokale Abkürzung für den Namen des Wochentages                           |           |
| %A          | Komplette Name des Wochentages in der lokalen Sprache                    |           |
| %b          | Lokale Abkürzung für den Namen des Monats                                |           |
| %B          | Vollständige Name des Monats in der lokalen Sprache                      |           |
| %c          | Format für angemessene Datums- und Zeitdarstellung der lokalen Plattform |           |
| %d          | Zweistellige Nummer des Tages im aktuellen Monat                         | 01 bis 31 |
| %H          | Zweistellige Stunde im 24-Stunden-Format                                 | 00 bis 23 |
| %I          | Zweistellige Stunde im 12-Stunden-Format                                 | 01 bis 12 |

| Platzhalter | Erklärung  | Bereich     |
|-------------|--|-------------|
| %j          | Dreistellige Nummer des Tages im Jahr  | 001 bis 366 |
| %m          | Zweistellige Nummer des Monats   | 01 bis 12   |
| %M          | Zweistellige Minute  | 00 bis 59   |
| %p          | Lokale Entsprechung für AM bzw. PM   |             |
| %S          | Zweistellige Sekunde   | 00 bis 61   |
| %U          | Zweistellige Wochennummer; Sonntag erster Wochentag; Zeitraum vor erstem Sonntag des Jahres ist 0. Woche | 01 bis 53   |
| %w          | Nummer des aktuellen Tages in der Woche. Sonntag ist 0. Tag  | 0 bis 6     |

| Platzhalter | Erklärung  | Bereich   |
|-------------|--|-----------|
| %W          | Nummer des aktuellen Tages in der Woche. Montag ist 0. Tag                 |           |
| %x          | Datumsformat der lokalen Plattform   |           |
| %X          | Zeitformat der lokalen Plattform   |           |
| %y          | Zweistelliges Jahr ohne Jahrhundertangabe                                  | 00 bis 99 |
| %Y          | Komplette Jahreszahl mit Jahrhundertangabe                                 |           |
| %Z          | Lokale Zeitzone oder leerer String (wenn keine lokale Zeitzone festgelegt) |           |
| %%          | Prozentzeichen % im Resultat-String  |           |

Nach Ernesti und Kaiser, S. 247-248

Sie können diese Platzhalter jetzt in eine Zeichenkette integrieren und `strftime()` ersetzt sie dann durch die entsprechende Zeit- oder Datumsinformation:

```
time.strftime("Heute ist %A, der %d. %B %Y. Die aktuelle Uhrzeit ist %H:%M Uhr.", lokalzeit_struct_time)
```

```
'Heute ist Sunday, der 22. November 2020. Die aktuelle Uhrzeit ist 16:58 Uhr.'
```

## Einschub: *locale*

Es gibt allerdings noch einen Fehler, und zwar werden die Tages- und Monatsnamen--zumindest auf meinem Computer--in ihrer englischen Form angegeben. Mit der Funktion *setlocale()* aus dem Paket *locale* können wir erreichen, dass die länder- und sprachspezifischen Einstellungen--wie sie auf Ihrem Computer eingerichtet sind--auch von Python angewandt werden.

```
import locale  
locale.setlocale(locale.LC_ALL, "")
```

```
'German_Germany.1252'
```

Wenn wir die vorherige Zeile noch einmal ausführen, werden jetzt die korrekten Versionen der Namen ausgegeben:

```
time.strftime("Heute ist %A, der %d. %B %Y. Die aktuelle Uhrzeit ist %H:%M  
Uhr.", lokalzeit_struct_time)
```

```
'Heute ist Sonntag, der 22. November 2020. Die aktuelle Uhrzeit ist 16:58 Uhr.'
```

## *strptime()*

Mit *strptime()* können wir das Gegenteil von *strftime()* erreichen, d.h. aus einer Zeichenkette ein *struct\_time*-Objekt erzeugen. Hier geben wir ein Muster vor, das auf eine zu bearbeitende Zeichenkette angewendet wird:

```
zeit = "Freitag, der 20. November 2020. Die aktuelle Uhrzeit ist 12:41 Uhr"  
time.strptime(zeit, "%A, der %d. %B %Y. Die aktuelle Uhrzeit ist %H:%M Uhr")
```

```
time.struct_time(tm_year=2020, tm_mon=11, tm_mday=20, tm_hour=12, tm_min=41, tm_sec=0, tm_wday=4, tm_yday=325, tm_isdst=-1)
```

## *sleep()*

Die letzte Funktion aus dem *time*-Modul, die hier erwähnt werden soll, ist *sleep()*. Der Name ist Programm, denn damit lässt man ein Programm eine in Sekunden angegebene Dauer "schlafen", d.h., es pausiert während dieser Zeit. Im folgenden Beispiel werden in einer Schleife Zahlenwerte angezeigt. Nach jedem Wert pausiert das Programm eine Sekunde:

```
zahlen = reversed(range(0, 6))  
for i in zahlen:  
    print(i)  
    time.sleep(1)  
print("Ende")
```

1  
0  
Ende

## Python und Zeit: *datetime*

Eine weitere Möglichkeit, in Python mit Zeit- und Datumsangaben zu arbeiten, bietet das Modul *datetime*:

```
import datetime
```

Das Modul stellt dabei sechs verschiedene Klassen zur Verfügung:

- *date*
- *time*
- *datetime*
- *timedelta*
- (*tzinfo* und *timezone* sollen hier nur erwähnt werden)

### *datetime.date*

Ein Objekt der Klasse *date* besteht aus einem Datum mit Jahr, Monat und Tag (year, month, day); zugrunde liegt dabei der gregorianische Kalender, der in die Zeit vor seiner Einführung verlängert wurde. Das Objekt wird folgendermaßen erzeugt:

```
a = datetime.date(1990, 10, 3)
a
```

```
datetime.date(1990, 10, 3)
```

Auch können Sie mit *datetime.date.today()* das aktuelle Datum anzeigen lassen:

```
b = datetime.date.today()
b
```

```
datetime.date(2020, 11, 22)
```

Wir können jetzt verschiedene Methoden auf das Objekt anwenden, z.B.:

| Methode                      | Erklärung   | Beispiel                               | Ergebnis                      |
|------------------------------|---|--|-------------------------------|
| ctime()                      | Erzeugt Zeichenkette                                  | datetime.date.ctime(a)                 | Wed Oct 3 00:00:00<br>1990    |
| isoformat()                  | Erzeugt ISO-8601-konformes Datum                      | datetime.date.isoformat(a)             | 1990-10-03                    |
| replace(year,<br>month, day) | Ersetzt einzelne Elemente im Datum                    | a.replace(day=4)                       | datetime.date(1990,<br>10, 4) |
| strftime()                   | Formatiert Datum (ähnlich<br><i>time.strftime()</i> ) | datetime.date.strftime(a,<br>"%d. %B") | 03. Oktober                   |
| weekday()                    | Wochentag als Zahl, Montag ist 0,<br>Sonntag ist 6    | datetime.date.weekday(a)               | 2                             |

| Methode      | Erklärung                                       | Beispiel                 | Ergebnis |
|--------------|---|--------------------------|----------|
| isoweekday() | Wochentag als Zahl, Montag ist 1, Sonntag ist 7 | datetime.date.weekday(a) | 3        |

## datetime.time

Ein Objekt der Klasse *time* besteht aus einer Zeit mit Stunde, Minute, Sekunde, Mikrosekunde und Informationen zur Zeitzone (hour, minute, second, microsecond, tzinfo). Das Objekt wird folgendermaßen erzeugt:

```
b = datetime.time(9, 30, 25)
b
```

```
datetime.time(9, 30, 25)
```

Es finden sich hier ähnliche Methoden wie in *datetime.date*, z.B.:

| Methode     | Erklärung   | Beispiel                           | Ergebnis                 |
|-------------|---|------------------------------------|--------------------------|
| isoformat() | Erzeugt ISO-8601-konforme Zeit                    | datetime.time.isoformat(b)         | 09:30:25                 |
| replace()   | Ersetzt einzelne Elemente im Datum                | a.replace(hour=4)                  | datetime.time(4, 30, 25) |
| strftime()  | Formatiert Zeit (ähnlich <i>time.strftime()</i> ) | datetime.time.strftime(a, "%H:%M") | 09:30                    |

## datetime.datetime

Ein Objekt der Klasse *datetime* kombiniert *datetime.date* und *datetime.time*, um sowohl Datum als auch Zeit angeben zu können. Es besteht aus Jahr, Monat, Tag, Stunde, Minute, Sekunde und Mikrosekunde:

```
c = datetime.datetime(1990, 10, 3, 9, 45, 20, 123456)
```

```
datetime.datetime(1990, 10, 3, 9, 45, 20, 123456)
```

Es gibt aber noch andere Möglichkeiten, ein *datetime*-Objekt zu erzeugen:

| Methode            | Erklärung  | Beispiel   |
|--------------------|--|--|
| now()              | Aktuelle Lokalzeit   | datetime.datetime.now()  |
| utcnow()           | Aktuelle UTC-Zeit  | datetime.datetime.utcnow()   |
| fromtimestamp()    | Lokalzeit nach <i>timestamp</i>  | datetime.datetime.fromtimestamp(1605975243)                                |
| utcfromtimestamp() | UTC-Zeit nach <i>timestamp</i>   | datetime.datetime.utcfromtimestamp(1605975243)                             |
| combine()          | <i>datetime</i> -Objekt aus <i>date</i> - und <i>time</i> -Objekten              | datetime.datetime.combine(a, b)  |
| strptime()         | extrahiert Zeit/Datum aus Zeichenkette, speichert sie in <i>datetime</i> -Objekt | datetime.datetime.strptime("Mittwoch, der 03.10.1990", "%A, der %d.%m.%Y") |

Und auch hier wieder einige Methoden für die Arbeit mit *datetime.datetime*-Objekten:

| Methoden    | Erklärung  | Beispiel                               | Ergebnis                                  |
|-------------|--|--|---|
| ctime()     | Erzeugt Zeichenkette                               | datetime.datetime.ctime()              | Wed Oct 3 09:45:20 1990                   |
| date()      | Erzeugt <i>datetime.date</i> -Objekt               | datetime.datetime.date(c)              | datetime.date(1990, 10, 3)                |
| isoformat() | Erzeugt ISO-8601-konforme Zeit/Datum               | datetime.datetime.isoformat(c)         | 1990-10-03T09:45:20                       |
| replace()   | Ersetzt einzelne Elemente in Zeit/Datum            | c.replace(day=4)                       | datetime.datetime(1990, 10, 4, 9, 45, 20) |
| strftime()  | Formatiert Datum (ähnlich <i>time.strftime()</i> ) | datetime.datetime.strftime(c, "%d %B") | 03. Oktober                               |
| time()      | Erzeugt <i>datetime.time</i> -Objekt               | datetime.datetime.time(c)              | datetime.time(9, 45, 20)                  |
| weekday()   | Wochentag als Zahl, Montag ist 0, Sonntag ist 6    | datetime.datetime.weekday(c)           | 2   |

## datetime.timedelta

*datetime.timedelta*-Objekte beschreiben Zeitspannen. Diese werden in Tagen, Sekunden und Mikrosekunden gespeichert. Schauen wir uns ein paar Beispiele an. Hier wird berechnet, wie viele Tage seit dem 3.10.1990 vergangen sind:

```
a = datetime.date(1990, 10, 3)
b = datetime.date.today()
c = b - a
c
```

datetime.timedelta(days=11008)

Im zweiten Beispiel nehmen wir ein Datum (3.10.1990) als *datetime*-Objekt und addieren 10 Tage und 2 Stunden (oder 7200 Sekunden) als *timedelta*-Objekt:

```
a = datetime.datetime(1990, 10, 3, 9, 30)
b = datetime.timedelta(10, 7200, 0) # 10 Tage, 7200 Sekunden (= 2 Stunden)
c = a + b
c
```

datetime.datetime(1990, 10, 13, 11, 30)

Zu den möglichen Berechnungsmöglichkeiten siehe <https://docs.python.org/3/library/datetime.html> unter *timedelta* - "Supported operations."

## Literatur:

- Dokumentation zu *datetime*: <https://docs.python.org/3/library/datetime.html>
- Dokumentation zu *time*: <https://docs.python.org/3/library/time.html>
- Ernesti, Johannes und Peter Kaiser. *Python 3: Das umfassende Handbuch*. 5., aktualisierte Auflage. Bonn: Rheinwerk, 2017. Frei zugänglich unter: <http://openbook.rheinwerk-verlag.de/python/index.html>. Kapitel 17: Datum und Zeit.

# Modul calendar

## calendar

Das letzte Modul, das hier für die Arbeit mit Datums- und/oder Zeitangaben vorgestellt werden soll, ist **calendar**. Wie der Name schon andeutet, können Sie das Modul nutzen, um Kalender zu erstellen und um bestimmte Informationen über einen Monat oder ein Jahr zu bekommen.

Zuerst einmal müssen wir das Modul nun importieren. Auch wollen wir sicherstellen, dass die auf dem jeweiligen Computer geltenden "locale"-Einstellungen berücksichtigt werden:

```
import calendar
import locale

locale.setlocale(locale.LC_ALL, "")
```

```
'German_Germany.1252'
```

Uns stehen jetzt mehrere Funktionen zur Verfügung, mit denen wir Einstellungen verändern oder Informationen abfragen können. Hier sei nur eine Auswahl genannt.

Wir können hier beispielsweise einen beliebigen Wochentag als Wochenanfang festlegen (Standardeinstellung ist Montag). Dabei können wir die englischen Namen der Tage verwenden:

```
calendar.setfirstweekday(calendar.WEDNESDAY)
```

Mit **firstweekday()** fragen wir ab, welcher Tag momentan als Wochenbeginn festgelegt ist:

```
calendar.firstweekday()
```

```
2
```

2 steht hier für Mittwoch (dazu gleich mehr).

**isleap()** gibt an, ob es sich bei einem Jahr um ein Schaltjahr handelt oder nicht:

```
calendar.isleap(1900)
```

```
False
```

**leapdays()** gibt die Anzahl der in einer gegebenen Zeitspanne enthaltenen Schalttage aus:

```
calendar.leapdays(1950, 2020)
```

```
17
```

**weekday()** schließlich bestimmt für ein gegebenes Datum den Wochentag:

```
calendar.weekday(1990, 10, 3)
```

## calendar.Calendar

*calender* bietet verschiedene Kalenderklassen an. Zuerst sehen wir uns die Klasse *Calendar* an, die einfache, unformatierte Kalenderobjekte erzeugt:

```
a = calendar.Calendar(firstweekday=0)
a
```

<calendar.Calendar at 0x1d6730b3b80>

`firstweekday=0` bewirkt, dass Montag als Wochenbeginn festgelegt wird. `firstweekday=6` verwendet stattdessen Sonntag als ersten Wochentag, wie es etwa in den USA üblich ist.

Auf dieses Objekt können wir nun verschiedene Methoden anwenden. `itermonthdates()` beispielsweise gibt uns ein iterierbares Objekt zurück, welches die Daten für den gewünschten Monat enthält.

Wenn ein Monat in der Wochenmitte beginnt oder endet, werden auch diejenigen Tage des vorhergehenden oder folgenden Monats eingeschlossen, die noch in der jeweiligen Woche liegen.

```
b = a.itermonthdates(2020, 12)
b
```

<generator object Calendar.itermonthdates at 0x000001D673041BA0>

Um die einzelnen Tage sehen zu können, verwenden wir eine Schleife:

```
for i in b:
    print(i)
```

2020-11-30  
2020-12-01  
2020-12-02  
2020-12-03  
2020-12-04  
2020-12-05  
2020-12-06  
2020-12-07  
2020-12-08  
2020-12-09  
2020-12-10  
2020-12-11  
2020-12-12  
2020-12-13  
2020-12-14  
2020-12-15  
2020-12-16  
2020-12-17  
2020-12-18  
2020-12-19  
2020-12-20  
2020-12-21  
2020-12-22  
2020-12-23  
2020-12-24  
2020-12-25

```
2020-12-26  
2020-12-27  
2020-12-28  
2020-12-29  
2020-12-30  
2020-12-31  
2021-01-01  
2021-01-02  
2021-01-03
```

Eine Variation davon ist **itermonthdays()**. Dabei werden alle Tage außerhalb des ausgewählten Monats nur als "0" angegeben und Monats- sowie Jahresangaben fallen weg:

```
b = a.itermonthdays(2020, 12)  
for i in b:  
    print(i)
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
0  
0  
0
```

Falls man weitere Angaben benötigt--wie etwa Jahreszahlen oder die Position des jeweiligen Tages in der Woche--stehen Variationen (**itermonthdays2**, **itermonthdays3** und **itermonthdays4**) zur Verfügung.

Wenn Sie Daten nach Wochen gruppiert haben möchten, können Sie zwischen den Methoden **monthdatescalendar()**, **monthdayscalendar()** und **monthdays2calendar()** wählen.  
**monthdatescalendar()** etwa gibt für jede Woche eine Liste zurück, die jeweils sieben *datetime.date*-Objekte enthält:

```
b = a.monthdatescalendar(2020, 12)
for i in b:
    print(i)
```

```
[datetime.date(2020, 11, 30), datetime.date(2020, 12, 1), datetime.date(2020, 12, 2), datetime.date(2020, 12, 3), datetime.date(2020, 12, 4), datetime.date(2020, 12, 5), datetime.date(2020, 12, 6)]
[datetime.date(2020, 12, 7), datetime.date(2020, 12, 8), datetime.date(2020, 12, 9), datetime.date(2020, 12, 10), datetime.date(2020, 12, 11), datetime.date(2020, 12, 12), datetime.date(2020, 12, 13)]
[datetime.date(2020, 12, 14), datetime.date(2020, 12, 15), datetime.date(2020, 12, 16), datetime.date(2020, 12, 17), datetime.date(2020, 12, 18), datetime.date(2020, 12, 19), datetime.date(2020, 12, 20)]
[datetime.date(2020, 12, 21), datetime.date(2020, 12, 22), datetime.date(2020, 12, 23), datetime.date(2020, 12, 24), datetime.date(2020, 12, 25), datetime.date(2020, 12, 26), datetime.date(2020, 12, 27)]
[datetime.date(2020, 12, 28), datetime.date(2020, 12, 29), datetime.date(2020, 12, 30), datetime.date(2020, 12, 31), datetime.date(2021, 1, 1), datetime.date(2021, 1, 2), datetime.date(2021, 1, 3)]
```

Mit **monthdayscalendar()** bekommen Sie stattdessen für jede Woche eine Liste mit Tageszahlen:

```
b = a.monthdayscalendar(2020, 12)
for i in b:
    print(i)
```

```
[0, 1, 2, 3, 4, 5, 6]
[7, 8, 9, 10, 11, 12, 13]
[14, 15, 16, 17, 18, 19, 20]
[21, 22, 23, 24, 25, 26, 27]
[28, 29, 30, 31, 0, 0, 0]
```

Schließlich gibt es noch die Möglichkeit, die Daten für ein ganzes Jahr ausgeben zu lassen; dazu stehen Ihnen die Methoden **yeardatescalendar()**, **yeardayscalendar()** und **yeardayscalendar2()** zur Verfügung. Alle drei Methoden akzeptieren neber der Jahreszahl ein zweites Argument (*width*), mit dem Sie angeben können, wie viele Monate jeweils in einer Zeile zusammengefasst werden sollen.

```
b = a.yeardatescalendar(2020, 2)
for i in b:
    print(i)
    print("-----")
```

```
[[[datetime.date(2019, 12, 30), datetime.date(2019, 12, 31), datetime.date(2020, 1, 1), datetime.date(2020, 1, 2), datetime.date(2020, 1, 3), datetime.date(2020, 1, 4), datetime.date(2020, 1, 5)], [datetime.date(2020, 1, 6), datetime.date(2020, 1, 7), datetime.date(2020, 1, 8), datetime.date(2020, 1, 9), datetime.date(2020, 1, 10), datetime.date(2020, 1, 11), datetime.date(2020, 1, 12)], [datetime.date(2020, 1, 13), datetime.date(2020, 1, 14), datetime.date(2020, 1, 15), datetime.date(2020, 1, 16), datetime.date(2020, 1, 17), datetime.date(2020, 1, 18), datetime.date(2020, 1, 19)], [datetime.date(2020, 1, 20), datetime.date(2020, 1, 21), datetime.date(2020, 1, 22), datetime.date(2020, 1, 23), datetime.date(2020, 1, 24), datetime.date(2020, 1, 25), datetime.date(2020, 1, 26)], [datetime.date(2020, 1, 27), datetime.date(2020, 1, 28), datetime.date(2020, 1, 29), datetime.date(2020, 1, 30), datetime.date(2020, 1, 31), datetime.date(2020, 2, 1), datetime.date(2020, 2, 2)], [[datetime.date(2020, 1, 27), datetime.date(2020, 1, 28), datetime.date(2020, 1, 29), datetime.date(2020, 1, 30), datetime.date(2020, 1, 31), datetime.date(2020, 2, 1), datetime.date(2020, 2, 2)]]], [[datetime.date(2020, 2, 1), datetime.date(2020, 2, 2), datetime.date(2020, 2, 3), datetime.date(2020, 2, 4), datetime.date(2020, 2, 5), datetime.date(2020, 2, 6), datetime.date(2020, 2, 7), datetime.date(2020, 2, 8), datetime.date(2020, 2, 9)], [datetime.date(2020, 2, 10), datetime.date(2020, 2, 11), datetime.date(2020, 2, 12), datetime.date(2020, 2, 13), datetime.date(2020, 2, 14), datetime.date(2020, 2, 15), datetime.date(2020, 2, 16), datetime.date(2020, 2, 17), datetime.date(2020, 2, 18), datetime.date(2020, 2, 19), datetime.date(2020, 2, 20), datetime.date(2020, 2, 21), datetime.date(2020, 2, 22), datetime.date(2020, 2, 23), datetime.date(2020, 2, 24), datetime.date(2020, 2, 25), datetime.date(2020, 2, 26), datetime.date(2020, 2, 27), datetime.date(2020, 2, 28), datetime.date(2020, 2, 29), datetime.date(2020, 2, 30), datetime.date(2020, 2, 31)]]]
```

```
te(2020, 2, 15), datetime.date(2020, 2, 16)], [datetime.date(2020, 2, 17), datet
ime.date(2020, 2, 18), datetime.date(2020, 2, 19), datetime.date(2020, 2, 20), d
atetime.date(2020, 2, 21), datetime.date(2020, 2, 22), datetime.date(2020, 2, 2
3)], [datetime.date(2020, 2, 24), datetime.date(2020, 2, 25), datetime.date(202
0, 2, 26), datetime.date(2020, 2, 27), datetime.date(2020, 2, 28), datetime.date
(2020, 2, 29), datetime.date(2020, 3, 1)]]
-----
[[[datetime.date(2020, 2, 24), datetime.date(2020, 2, 25), datetime.date(2020,
2, 26), datetime.date(2020, 2, 27), datetime.date(2020, 2, 28), datetime.date(20
20, 2, 29), datetime.date(2020, 3, 1)], [datetime.date(2020, 3, 2), datetime.dat
e(2020, 3, 3), datetime.date(2020, 3, 4), datetime.date(2020, 3, 5), datetime.da
te(2020, 3, 6), datetime.date(2020, 3, 7), datetime.date(2020, 3, 8)], [datetim
e.date(2020, 3, 9), datetime.date(2020, 3, 10), datetime.date(2020, 3, 11), dat
etime.date(2020, 3, 12), datetime.date(2020, 3, 13), datetime.date(2020, 3, 14),
datetime.date(2020, 3, 15)], [datetime.date(2020, 3, 16), datetime.date(2020, 3,
17), datetime.date(2020, 3, 18), datetime.date(2020, 3, 19), datetime.date(2020,
3, 20), datetime.date(2020, 3, 21), datetime.date(2020, 3, 22)], [datetime.date
(2020, 3, 23), datetime.date(2020, 3, 24), datetime.date(2020, 3, 25), datetim
e.date(2020, 3, 26), datetime.date(2020, 3, 27), datetime.date(2020, 3, 28), datet
ime.date(2020, 3, 29)], [datetime.date(2020, 3, 30), datetime.date(2020, 3, 31),
datetime.date(2020, 4, 1), datetime.date(2020, 4, 2), datetime.date(2020, 4, 3),
datetime.date(2020, 4, 4), datetime.date(2020, 4, 5)], [[datetime.date(2020, 3,
30), datetime.date(2020, 3, 31), datetime.date(2020, 4, 1), datetime.date(2020,
4, 2), datetime.date(2020, 4, 3), datetime.date(2020, 4, 4), datetime.date(2020,
4, 5)], [datetime.date(2020, 4, 6), datetime.date(2020, 4, 7), datetime.date(202
0, 4, 8), datetime.date(2020, 4, 9), datetime.date(2020, 4, 10), datetime.date(2
020, 4, 11), datetime.date(2020, 4, 12)], [datetime.date(2020, 4, 13), datetim
e.date(2020, 4, 14), datetime.date(2020, 4, 15), datetime.date(2020, 4, 16), datet
ime.date(2020, 4, 17), datetime.date(2020, 4, 18), datetime.date(2020, 4, 19)],
[datetime.date(2020, 4, 20), datetime.date(2020, 4, 21), datetime.date(2020, 4,
22), datetime.date(2020, 4, 23), datetime.date(2020, 4, 24), datetime.date(2020,
4, 25), datetime.date(2020, 4, 26)], [datetime.date(2020, 4, 27), datetime.date
(2020, 4, 28), datetime.date(2020, 4, 29), datetime.date(2020, 4, 30), datetim
e.date(2020, 5, 1), datetime.date(2020, 5, 2), datetime.date(2020, 5, 3)]]]
-----
[[[datetime.date(2020, 4, 27), datetime.date(2020, 4, 28), datetime.date(2020,
4, 29), datetime.date(2020, 4, 30), datetime.date(2020, 5, 1), datetime.date(202
0, 5, 2), datetime.date(2020, 5, 3)], [datetime.date(2020, 5, 4), datetime.dat
e(2020, 5, 5), datetime.date(2020, 5, 6), datetime.date(2020, 5, 7), datetime.dat
e(2020, 5, 8), datetime.date(2020, 5, 9), datetime.date(2020, 5, 10)], [datetim
e.date(2020, 5, 11), datetime.date(2020, 5, 12), datetime.date(2020, 5, 13), dat
etime.date(2020, 5, 14), datetime.date(2020, 5, 15), datetime.date(2020, 5, 16),
datetime.date(2020, 5, 17)], [datetime.date(2020, 5, 18), datetime.date(2020, 5,
19), datetime.date(2020, 5, 20), datetime.date(2020, 5, 21), datetime.date(2020,
5, 22), datetime.date(2020, 5, 23), datetime.date(2020, 5, 24)], [datetime.date
(2020, 5, 25), datetime.date(2020, 5, 26), datetime.date(2020, 5, 27), datetim
e.date(2020, 5, 28), datetime.date(2020, 5, 29), datetime.date(2020, 5, 30), datet
ime.date(2020, 5, 31)], [[datetime.date(2020, 6, 1), datetime.date(2020, 6, 2),
datetime.date(2020, 6, 3), datetime.date(2020, 6, 4), datetime.date(2020, 6, 5),
datetime.date(2020, 6, 6), datetime.date(2020, 6, 7)], [datetime.date(2020, 6,
8), datetime.date(2020, 6, 9), datetime.date(2020, 6, 10), datetime.date(2020,
6, 11), datetime.date(2020, 6, 12), datetime.date(2020, 6, 13), datetime.date(20
20, 6, 14)], [datetime.date(2020, 6, 15), datetime.date(2020, 6, 16), datetim
e.date(2020, 6, 17), datetime.date(2020, 6, 18), datetime.date(2020, 6, 19), dateti
me.date(2020, 6, 20), datetime.date(2020, 6, 21)], [datetime.date(2020, 6, 22),
datetime.date(2020, 6, 23), datetime.date(2020, 6, 24), datetime.date(2020, 6, 2
5), datetime.date(2020, 6, 26), datetime.date(2020, 6, 27), datetime.date(2020,
6, 28)], [datetime.date(2020, 6, 29), datetime.date(2020, 6, 30), datetime.date
(2020, 7, 1), datetime.date(2020, 7, 2), datetime.date(2020, 7, 3), datetime.dat
e(2020, 7, 4), datetime.date(2020, 7, 5)]]]
-----
[[[datetime.date(2020, 6, 29), datetime.date(2020, 6, 30), datetime.date(2020,
7, 1), datetime.date(2020, 7, 2), datetime.date(2020, 7, 3), datetime.date(2020,
7, 4), datetime.date(2020, 7, 5)], [datetime.date(2020, 7, 6), datetime.date(202
0, 7, 7), datetime.date(2020, 7, 8), datetime.date(2020, 7, 9), datetime.date(202
0, 7, 10), datetime.date(2020, 7, 11), datetime.date(2020, 7, 12)], [datetim
e.date(2020, 7, 13), datetime.date(2020, 7, 14), datetime.date(2020, 7, 15), dateti
```

```
me.date(2020, 7, 16), datetime.date(2020, 7, 17), datetime.date(2020, 7, 18), da
tetime.date(2020, 7, 19)], [datetime.date(2020, 7, 20), datetime.date(2020, 7, 2
1), datetime.date(2020, 7, 22), datetime.date(2020, 7, 23), datetime.date(2020,
7, 24), datetime.date(2020, 7, 25), datetime.date(2020, 7, 26)], [datetime.date
(2020, 7, 27), datetime.date(2020, 7, 28), datetime.date(2020, 7, 29), datetim
e.date(2020, 7, 30), datetime.date(2020, 7, 31), datetime.date(2020, 8, 1), dateti
me.date(2020, 8, 2)]], [[datetime.date(2020, 7, 27), datetime.date(2020, 7, 28),
datetime.date(2020, 7, 29), datetime.date(2020, 7, 30), datetime.date(2020, 7, 3
1), datetime.date(2020, 8, 1), datetime.date(2020, 8, 2)], [datetime.date(2020,
8, 3), datetime.date(2020, 8, 4), datetime.date(2020, 8, 5), datetime.date(2020,
8, 6), datetime.date(2020, 8, 7), datetime.date(2020, 8, 8), datetime.date(2020,
8, 9)], [datetime.date(2020, 8, 10), datetime.date(2020, 8, 11), datetime.date(2
020, 8, 12), datetime.date(2020, 8, 13), datetime.date(2020, 8, 14), datetime.da
te(2020, 8, 15), datetime.date(2020, 8, 16)], [datetime.date(2020, 8, 17), datet
ime.date(2020, 8, 18), datetime.date(2020, 8, 19), datetime.date(2020, 8, 20), d
atetime.date(2020, 8, 21), datetime.date(2020, 8, 22), datetime.date(2020, 8, 2
3)], [datetime.date(2020, 8, 24), datetime.date(2020, 8, 25), datetime.date(202
0, 8, 26), datetime.date(2020, 8, 27), datetime.date(2020, 8, 28), datetime.date
(2020, 8, 29), datetime.date(2020, 8, 30)], [datetime.date(2020, 8, 31), datetim
e.date(2020, 9, 1), datetime.date(2020, 9, 2), datetime.date(2020, 9, 3), dateti
me.date(2020, 9, 4), datetime.date(2020, 9, 5), datetime.date(2020, 9, 6)]]
-----
```

```
[[[datetime.date(2020, 8, 31), datetime.date(2020, 9, 1), datetime.date(2020, 9,
2), datetime.date(2020, 9, 3), datetime.date(2020, 9, 4), datetime.date(2020, 9,
5), datetime.date(2020, 9, 6)], [datetime.date(2020, 9, 7), datetime.date(2020,
9, 8), datetime.date(2020, 9, 9), datetime.date(2020, 9, 10), datetime.date(202
0, 9, 11), datetime.date(2020, 9, 12), datetime.date(2020, 9, 13)], [datetim
e(2020, 9, 14), datetime.date(2020, 9, 15), datetime.date(2020, 9, 16), datetim
e(2020, 9, 17), datetime.date(2020, 9, 18), datetime.date(2020, 9, 19), dat
etime.date(2020, 9, 20)], [datetime.date(2020, 9, 21), datetime.date(2020, 9, 2
2), datetime.date(2020, 9, 23), datetime.date(2020, 9, 24), datetime.date(2020,
9, 25), datetime.date(2020, 9, 26), datetime.date(2020, 9, 27)], [datetime.date
(2020, 9, 28), datetime.date(2020, 9, 29), datetime.date(2020, 9, 30), datetim
e(2020, 10, 1), datetime.date(2020, 10, 2), datetime.date(2020, 10, 3), dateti
me.date(2020, 10, 4)]], [[datetime.date(2020, 9, 28), datetime.date(2020, 9, 2
9), datetime.date(2020, 9, 30), datetime.date(2020, 10, 1), datetime.date(2020,
10, 2), datetime.date(2020, 10, 3), datetime.date(2020, 10, 4)], [datetime.date
(2020, 10, 5), datetime.date(2020, 10, 6), datetime.date(2020, 10, 7), datetim
e(2020, 10, 8), datetime.date(2020, 10, 9), datetime.date(2020, 10, 10), date
time.date(2020, 10, 11)], [datetime.date(2020, 10, 12), datetime.date(2020, 10,
13), datetime.date(2020, 10, 14), datetime.date(2020, 10, 15), datetime.date(202
0, 10, 16), datetime.date(2020, 10, 17), datetime.date(2020, 10, 18)], [datetim
e(2020, 10, 19), datetime.date(2020, 10, 20), datetime.date(2020, 10, 21),
datetime.date(2020, 10, 22), datetime.date(2020, 10, 23), datetime.date(2020, 1
0, 24), datetime.date(2020, 10, 25)], [datetime.date(2020, 10, 26), datetime.dat
e(2020, 10, 27), datetime.date(2020, 10, 28), datetime.date(2020, 10, 29), datet
ime.date(2020, 10, 30), datetime.date(2020, 10, 31), datetime.date(2020, 11,
1)]]]
-----
```

```
[[[datetime.date(2020, 10, 26), datetime.date(2020, 10, 27), datetime.date(2020,
10, 28), datetime.date(2020, 10, 29), datetime.date(2020, 10, 30), datetime.date
(2020, 10, 31), datetime.date(2020, 11, 1)], [datetime.date(2020, 11, 2), dateti
me.date(2020, 11, 3), datetime.date(2020, 11, 4), datetime.date(2020, 11, 5), da
tetime.date(2020, 11, 6), datetime.date(2020, 11, 7), datetime.date(2020, 11,
8)], [datetime.date(2020, 11, 9), datetime.date(2020, 11, 10), datetime.date(202
0, 11, 11), datetime.date(2020, 11, 12), datetime.date(2020, 11, 13), datetim
e(2020, 11, 14), datetime.date(2020, 11, 15)], [datetime.date(2020, 11, 16), d
atetime.date(2020, 11, 17), datetime.date(2020, 11, 18), datetime.date(2020, 11,
19), datetime.date(2020, 11, 20), datetime.date(2020, 11, 21), datetime.date(202
0, 11, 22)], [datetime.date(2020, 11, 23), datetime.date(2020, 11, 24), datetim
e(2020, 11, 25), datetime.date(2020, 11, 26), datetime.date(2020, 11, 27),
datetime.date(2020, 11, 28), datetime.date(2020, 11, 29)], [datetime.date(2020,
11, 30), datetime.date(2020, 12, 1), datetime.date(2020, 12, 2), datetime.date(2
020, 12, 3), datetime.date(2020, 12, 4), datetime.date(2020, 12, 5), datetim
e(2020, 12, 6)]], [[datetime.date(2020, 11, 30), datetime.date(2020, 12, 1), da
tetime.date(2020, 12, 2), datetime.date(2020, 12, 3), datetime.date(2020, 12,
4), datetime.date(2020, 12, 5), datetime.date(2020, 12, 6)], [datetime.date(202
```

```
0, 12, 7), datetime.date(2020, 12, 8), datetime.date(2020, 12, 9), datetime.date(2020, 12, 10), datetime.date(2020, 12, 11), datetime.date(2020, 12, 12), datetime.date(2020, 12, 13)], [datetime.date(2020, 12, 14), datetime.date(2020, 12, 15), datetime.date(2020, 12, 16), datetime.date(2020, 12, 17), datetime.date(2020, 12, 18), datetime.date(2020, 12, 19), datetime.date(2020, 12, 20)], [datetime.date(2020, 12, 21), datetime.date(2020, 12, 22), datetime.date(2020, 12, 23), datetime.date(2020, 12, 24), datetime.date(2020, 12, 25), datetime.date(2020, 12, 26), datetime.date(2020, 12, 27)], [datetime.date(2020, 12, 28), datetime.date(2020, 12, 29), datetime.date(2020, 12, 30), datetime.date(2020, 12, 31), datetime.date(2021, 1, 1), datetime.date(2021, 1, 2), datetime.date(2021, 1, 3)]]
```

-----

## calendar.TextCalendar

Neben einem unformatierten *Calendar*-Objekt lässt sich auch ein *TextCalendar*-Objekt erstellen, welches Daten (entweder für einen Monat oder ein ganzes Jahr) für eine Anzeige formatiert, wie hier durch die Methode **formatmonth()**:

```
a = calendar.TextCalendar(firstweekday=0)
a.formatmonth(2020, 12, 4, 1)
```

```
'          Dezember 2020\n Mo   Di   Mi   Do   Fr   Sa   So\n4      5     6\n    7     8     9     10    11    12    13\n0\n21     22    23    24    25    26    27\n      28    29    30    31\n'          1     2     3     2
```

Das dritte Argument definiert, wie breit jede Spalte sein soll, während das vierte Argument angibt, wie viele Zeilen pro Woche genutzt werden sollen.

Das Ergebnis ist allerdings noch recht unübersichtlich. So werden z.B. Zeilenumbrüche nicht als solche dargestellt, sondern durch ein **\n** markiert. Das können Sie übrigens auch in formatierten Zeichenketten verwenden, wenn Sie dort einen Zeilenumbruch einfügen möchten.

Um das Ergebnis auch hier angemessen darzustellen, können wir die Methode **prmonth()** nutzen; diese akzeptiert die gleichen Argumente wie **formatmonth()**:

```
a.prmonth(2020, 12, 3, 1)
```

```
Dezember 2020
Mo Di Mi Do Fr Sa So
  1  2  3  4  5  6
  7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

Analog dazu gibt es **formatyear()** und **pryear()**, welche für die Darstellung eines ganzen Jahres genutzt werden können. Die zwei letzten Argumente bestimmen, wie viele Leerzeichen horizontaler Abstand zwischen den einzelnen Monaten eingefügt und wie viele Monate pro Zeile dargestellt werden sollen:

```
a.pryear(2020, 3, 1, 8, 2)
```

2020

| Januar |    |    |    |    |    |    | Februar |    |    |    |    |    |    |
|--------|----|----|----|----|----|----|---------|----|----|----|----|----|----|
| Mo     | Di | Mi | Do | Fr | Sa | So | Mo      | Di | Mi | Do | Fr | Sa | So |
| 1      | 2  | 3  | 4  | 5  |    |    |         |    | 1  | 2  |    |    |    |

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 6  | 7  | 8  | 9  | 10 | 11 | 12 | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 27 | 28 | 29 | 30 | 31 |    |    | 24 | 25 | 26 | 27 | 28 | 29 |    |

| März |    |    |    |    |    |    | April |    |    |    |    |    |    |
|------|----|----|----|----|----|----|-------|----|----|----|----|----|----|
| Mo   | Di | Mi | Do | Fr | Sa | So | Mo    | Di | Mi | Do | Fr | Sa | So |
|      |    |    |    |    | 1  |    |       |    | 1  | 2  | 3  | 4  | 5  |
| 2    | 3  | 4  | 5  | 6  | 7  | 8  | 6     | 7  | 8  | 9  | 10 | 11 | 12 |
| 9    | 10 | 11 | 12 | 13 | 14 | 15 | 13    | 14 | 15 | 16 | 17 | 18 | 19 |
| 16   | 17 | 18 | 19 | 20 | 21 | 22 | 20    | 21 | 22 | 23 | 24 | 25 | 26 |
| 23   | 24 | 25 | 26 | 27 | 28 | 29 | 27    | 28 | 29 | 30 |    |    |    |
| 30   | 31 |    |    |    |    |    |       |    |    |    |    |    |    |

| Mai |    |    |    |    |    |    | Juni |    |    |    |    |    |    |
|-----|----|----|----|----|----|----|------|----|----|----|----|----|----|
| Mo  | Di | Mi | Do | Fr | Sa | So | Mo   | Di | Mi | Do | Fr | Sa | So |
|     |    |    |    | 1  | 2  | 3  | 1    | 2  | 3  | 4  | 5  | 6  | 7  |
| 4   | 5  | 6  | 7  | 8  | 9  | 10 | 8    | 9  | 10 | 11 | 12 | 13 | 14 |
| 11  | 12 | 13 | 14 | 15 | 16 | 17 | 15   | 16 | 17 | 18 | 19 | 20 | 21 |
| 18  | 19 | 20 | 21 | 22 | 23 | 24 | 22   | 23 | 24 | 25 | 26 | 27 | 28 |
| 25  | 26 | 27 | 28 | 29 | 30 | 31 | 29   | 30 |    |    |    |    |    |

| Juli |    |    |    |    |    |    | August |    |    |    |    |    |    |
|------|----|----|----|----|----|----|--------|----|----|----|----|----|----|
| Mo   | Di | Mi | Do | Fr | Sa | So | Mo     | Di | Mi | Do | Fr | Sa | So |
|      |    |    | 1  | 2  | 3  | 4  | 1      | 2  | 3  | 4  | 5  | 6  | 7  |
| 6    | 7  | 8  | 9  | 10 | 11 | 12 | 3      | 4  | 5  | 6  | 7  | 8  | 9  |
| 13   | 14 | 15 | 16 | 17 | 18 | 19 | 10     | 11 | 12 | 13 | 14 | 15 | 16 |
| 20   | 21 | 22 | 23 | 24 | 25 | 26 | 17     | 18 | 19 | 20 | 21 | 22 | 23 |
| 27   | 28 | 29 | 30 | 31 |    |    | 24     | 25 | 26 | 27 | 28 | 29 | 30 |
|      |    |    |    |    |    |    | 31     |    |    |    |    |    |    |

| September |    |    |    |    |    |    | Oktober |    |    |    |    |    |    |
|-----------|----|----|----|----|----|----|---------|----|----|----|----|----|----|
| Mo        | Di | Mi | Do | Fr | Sa | So | Mo      | Di | Mi | Do | Fr | Sa | So |
|           |    | 1  | 2  | 3  | 4  | 5  | 1       | 2  | 3  | 4  | 5  | 6  |    |
| 7         | 8  | 9  | 10 | 11 | 12 | 13 | 5       | 6  | 7  | 8  | 9  | 10 | 11 |
| 14        | 15 | 16 | 17 | 18 | 19 | 20 | 12      | 13 | 14 | 15 | 16 | 17 | 18 |
| 21        | 22 | 23 | 24 | 25 | 26 | 27 | 19      | 20 | 21 | 22 | 23 | 24 | 25 |
| 28        | 29 | 30 |    |    |    |    | 26      | 27 | 28 | 29 | 30 | 31 |    |

| November |    |    |    |    |    |    | Dezember |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----------|----|----|----|----|----|----|
| Mo       | Di | Mi | Do | Fr | Sa | So | Mo       | Di | Mi | Do | Fr | Sa | So |
|          |    |    |    | 1  |    |    | 1        | 2  | 3  | 4  | 5  | 6  |    |
| 2        | 3  | 4  | 5  | 6  | 7  | 8  | 7        | 8  | 9  | 10 | 11 | 12 | 13 |
| 9        | 10 | 11 | 12 | 13 | 14 | 15 | 14       | 15 | 16 | 17 | 18 | 19 | 20 |
| 16       | 17 | 18 | 19 | 20 | 21 | 22 | 21       | 22 | 23 | 24 | 25 | 26 | 27 |
| 23       | 24 | 25 | 26 | 27 | 28 | 29 | 28       | 29 | 30 | 31 |    |    |    |
| 30       |    |    |    |    |    |    |          |    |    |    |    |    |    |

## calendar.HTMLCalendar

Besonders bei der Arbeit mit webbasierten Projekten kann es hilfreich sein, wenn ein Kalender von `calendar` bereits im HTML-Format bereitgestellt wird. Dazu erstellen wir zuerst ein `HTMLCalendar`-Objekt:

```
a = calendar.HTMLCalendar()
```

Mit `formatmonth()` können wir uns jetzt den Monat in einer HTML-Tabelle ausgeben lassen:

```
b = a.formatmonth(2020, 12, withyear=True)
print(b)
```

| Dezember 2020 |    |    |    |    |    |    |
|---------------|----|----|----|----|----|----|
| Mo            | Tu | We | Fr | Sa | Su |    |
|               |    |    |    |    |    |    |
|               | 1  | 2  |    |    |    |    |
| 3             | 4  | 5  |    |    |    | 6  |
| 7             | 8  | 9  |    |    |    | 13 |
| 14            | 15 | 16 |    |    |    | 20 |
| 21            | 22 | 23 |    |    |    | 27 |
| 28            | 29 | 30 |    |    |    |    |
|               |    |    |    |    |    |    |

Die visuelle Gestaltung der Tabelle funktioniert dann, wie bei Webseiten üblich, über CSS (*Cascading Style Sheet*). Dazu werden beispielsweise die einzelnen Zellen mit einer Datumsangabe mit Klassen versehen, so dass man leicht die einzelnen Wochentage unterschiedlich darstellen kann. *HTMLCalendar* bietet auch mehrere Attribute, mit deren Hilfe man die Klassen ändern kann (siehe offizielle Dokumentation für *calendar*).

Hier nun ein Beispiel für **formatyear()**:

```
b = a.formatyear(2020, 2)  
print(b)
```







```

<tr><td class="mon">2</td><td class="tue">3</td><td class="wed">4</td><td class="thu">5</td><td class="fri">6</td><td class="sat">7</td><td class="sun">8</td></tr>
<tr><td class="mon">9</td><td class="tue">10</td><td class="wed">11</td><td class="thu">12</td><td class="fri">13</td><td class="sat">14</td><td class="sun">15</td></tr>
<tr><td class="mon">16</td><td class="tue">17</td><td class="wed">18</td><td class="thu">19</td><td class="fri">20</td><td class="sat">21</td><td class="sun">22</td></tr>
<tr><td class="mon">23</td><td class="tue">24</td><td class="wed">25</td><td class="thu">26</td><td class="fri">27</td><td class="sat">28</td><td class="sun">29</td></tr>
<tr><td class="mon">30</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td></tr>
</table>
</td><td><table border="0" cellpadding="0" cellspacing="0" class="month">
<tr><th colspan="7" class="month">Dezember</th></tr>
<tr><th class="mon">Mo</th><th class="tue">Di</th><th class="wed">Mi</th><th class="thu">Do</th><th class="fri">Fr</th><th class="sat">Sa</th><th class="sun">So</th></tr>
<tr><td class="noday">&nbsp;</td><td class="tue">1</td><td class="wed">2</td><td class="thu">3</td><td class="fri">4</td><td class="sat">5</td><td class="sun">6</td></tr>
<tr><td class="mon">7</td><td class="tue">8</td><td class="wed">9</td><td class="thu">10</td><td class="fri">11</td><td class="sat">12</td><td class="sun">13</td></tr>
<tr><td class="mon">14</td><td class="tue">15</td><td class="wed">16</td><td class="thu">17</td><td class="fri">18</td><td class="sat">19</td><td class="sun">20</td></tr>
<tr><td class="mon">21</td><td class="tue">22</td><td class="wed">23</td><td class="thu">24</td><td class="fri">25</td><td class="sat">26</td><td class="sun">27</td></tr>
<tr><td class="mon">28</td><td class="tue">29</td><td class="wed">30</td><td class="thu">31</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td></tr>
</table>
</td></tr></table>

```

Schließlich gibt es dann noch die Möglichkeit, eine komplette HTML-Seite ausgeben zu lassen.

Dabei kann man ein CSS-Stylesheet für deren Gestaltung sowie die zu verwendende Zeichenkodierung angeben:

```

b = a.formatyearpage(2020, 2, css="mein_stylesheet.css", encoding="utf-8")
print(b)

```

```

b'<?xml version="1.0" encoding="utf-8"?>\n<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">\n<html>
<n><head>\n<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<n><link rel="stylesheet" type="text/css" href="mein_stylesheet.css" />\n<title>Calendar for 2020</title>\n</head>\n<body>\n<table border="0" cellpadding="0" cellspacing="0" class="year">\n<tr><th colspan="2" class="year">2020</th></tr><tr><td border="0" class="month">\n<table border="0" cellpadding="0" cellspacing="0" class="month">\n<tr><th colspan="7" class="month">Januar</th></tr>\n<tr><td class="mon">Mo</td><td class="tue">Di</td><td class="wed">Mi</td><td class="thu">Do</td><td class="fri">Fr</td><td class="sat">Sa</td><td class="sun">So</td></tr>\n<tr><td class="noday">&nbsp;</td><td class="wed">1</td><td class="thu">2</td><td class="fri">3</td><td class="sat">4</td><td class="sun">5</td></tr>\n<tr><td class="mon">6</td><td class="tue">7</td><td class="wed">8</td><td class="thu">9</td><td class="fri">10</td><td class="sat">11</td><td class="sun">12</td></tr>
<tr><td class="mon">13</td><td class="tue">14</td><td class="wed">15</td><td class="thu">16</td><td class="fri">17</td><td class="sat">18</td><td class="sun">19</td></tr>
<tr><td class="mon">20</td><td class="tue">21</td><td class="wed">22</td><td class="thu">23</td><td class="fri">24</td><td class="sat">25</td><td class="sun">26</td></tr>
<tr><td class="mon">27</td><td class="tue">28</td><td class="wed">29</td><td class="thu">30</td><td class="fri">31</td><td class="sat">&nbsp;</td><td class="sun">&nbsp;</td></tr>
</table>
</td></tr>

```





Für die Anzeige hier konvertieren wir noch den Datentyp von "bytes" zu "string":

```
b = a.formatyearpage(2020, 2, css="mein_stylesheet.css").decode("UTF-8")
print(b)
```

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="stylesheet" type="text/css" href="mein_stylesheet.css" />
<title>Calendar for 2020</title>
</head>
<body>
<table border="0" cellpadding="0" cellspacing="0" class="year">
<tr><th colspan="2" class="year">2020</th></tr><tr><td><table border="0" cellpadding="0" cellspacing="0" class="month">
<tr><th colspan="7" class="month">Januar</th></tr>
<tr><th class="mon">Mo</th><th class="tue">Di</th><th class="wed">Mi</th><th class="thu">Do</th><th class="fri">Fr</th><th class="sat">Sa</th><th class="sun">So</th></tr>
<tr><td class="noday">&ampnbsp</td><td class="noday">&ampnbsp</td><td class="wed">1</td><td class="thu">2</td><td class="fri">3</td><td class="sat">4</td><td class="sun">5</td></tr>
<tr><td class="mon">6</td><td class="tue">7</td><td class="wed">8</td><td class="thu">9</td><td class="fri">10</td><td class="sat">11</td><td class="sun">12</td></tr>
<tr><td class="mon">13</td><td class="tue">14</td><td class="wed">15</td><td class="thu">16</td><td class="fri">17</td><td class="sat">18</td><td class="sun">19</td></tr>
<tr><td class="mon">20</td><td class="tue">21</td><td class="wed">22</td><td class="thu">23</td><td class="fri">24</td><td class="sat">25</td><td class="sun">26</td></tr>
<tr><td class="mon">27</td><td class="tue">28</td><td class="wed">29</td><td class="thu">30</td><td class="fri">31</td><td class="sat">&nbsp;</td><td class="sun">&nbsp;</td></tr>
</table></td></tr>
</table></body>
```

```

      ="sun">5</td></tr>
<tr><td class="mon">6</td><td class="tue">7</td><td class="wed">8</td><td class
="thu">9</td><td class="fri">10</td><td class="sat">11</td><td class="sun">12</t
d></tr>
<tr><td class="mon">13</td><td class="tue">14</td><td class="wed">15</td><td cla
ss="thu">16</td><td class="fri">17</td><td class="sat">18</td><td class="sun">19
</td></tr>
<tr><td class="mon">20</td><td class="tue">21</td><td class="wed">22</td><td cla
ss="thu">23</td><td class="fri">24</td><td class="sat">25</td><td class="sun">26
</td></tr>
<tr><td class="mon">27</td><td class="tue">28</td><td class="wed">29</td><td cla
ss="thu">30</td><td class="fri">31</td><td class="noday">&nbsp;</td><td class="n
oday">&nbsp;</td></tr>
</table>
</td><td><table border="0" cellpadding="0" cellspacing="0" class="month">
<tr><th colspan="7" class="month">Februar</th></tr>
<tr><th class="mon">Mo</th><th class="tue">Di</th><th class="wed">Mi</th><th cla
ss="thu">Do</th><th class="fri">Fr</th><th class="sat">Sa</th><th class="sun">So
</th></tr>
<tr><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">
&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class
="sat">1</td><td class="sun">2</td></tr>
<tr><td class="mon">3</td><td class="tue">4</td><td class="wed">5</td><td class
="thu">6</td><td class="fri">7</td><td class="sat">8</td><td class="sun">9</td>
</tr>
<tr><td class="mon">10</td><td class="tue">11</td><td class="wed">12</td><td cla
ss="thu">13</td><td class="fri">14</td><td class="sat">15</td><td class="sun">16
</td></tr>
<tr><td class="mon">17</td><td class="tue">18</td><td class="wed">19</td><td cla
ss="thu">20</td><td class="fri">21</td><td class="sat">22</td><td class="sun">23
</td></tr>
<tr><td class="mon">24</td><td class="tue">25</td><td class="wed">26</td><td cla
ss="thu">27</td><td class="fri">28</td><td class="sat">29</td><td class="noday">
&nbsp;</td></tr>
</table>
</td></tr><tr><td><table border="0" cellpadding="0" cellspacing="0" class="mont
h">
<tr><th colspan="7" class="month">März</th></tr>
<tr><th class="mon">Mo</th><th class="tue">Di</th><th class="wed">Mi</th><th cla
ss="thu">Do</th><th class="fri">Fr</th><th class="sat">Sa</th><th class="sun">So
</th></tr>
<tr><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">
&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class
="noday">&nbsp;</td><td class="sun">1</td></tr>
<tr><td class="mon">2</td><td class="tue">3</td><td class="wed">4</td><td class
="thu">5</td><td class="fri">6</td><td class="sat">7</td><td class="sun">8</td>
</tr>
<tr><td class="mon">9</td><td class="tue">10</td><td class="wed">11</td><td clas
s="thu">12</td><td class="fri">13</td><td class="sat">14</td><td class="sun">15
</td></tr>
<tr><td class="mon">16</td><td class="tue">17</td><td class="wed">18</td><td cla
ss="thu">19</td><td class="fri">20</td><td class="sat">21</td><td class="sun">22
</td></tr>
<tr><td class="mon">23</td><td class="tue">24</td><td class="wed">25</td><td cla
ss="thu">26</td><td class="fri">27</td><td class="sat">28</td><td class="sun">29
</td></tr>
<tr><td class="mon">30</td><td class="tue">31</td><td class="noday">&nbsp;</td><td>
  | | | | | | |
```





```

="thu">8</td><td class="fri">9</td><td class="sat">10</td><td class="sun">11</td>
</tr>
<tr><td class="mon">12</td><td class="tue">13</td><td class="wed">14</td><td class="thu">15</td><td class="fri">16</td><td class="sat">17</td><td class="sun">18</td>
</tr>
<tr><td class="mon">19</td><td class="tue">20</td><td class="wed">21</td><td class="thu">22</td><td class="fri">23</td><td class="sat">24</td><td class="sun">25</td>
</tr>
<tr><td class="mon">26</td><td class="tue">27</td><td class="wed">28</td><td class="thu">29</td><td class="fri">30</td><td class="sat">31</td><td class="sun">nbsp;</td>
</tr>
</table>
</td></tr><tr><td><table border="0" cellpadding="0" cellspacing="0" class="month">
<tr><th colspan="7" class="month">November</th></tr>
<tr><th class="mon">Mo</th><th class="tue">Di</th><th class="wed">Mi</th><th class="thu">Do</th><th class="fri">Fr</th><th class="sat">Sa</th><th class="sun">So</th>
</tr>
<tr><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">1</td></tr>
<tr><td class="mon">2</td><td class="tue">3</td><td class="wed">4</td><td class="thu">5</td><td class="fri">6</td><td class="sat">7</td><td class="sun">8</td>
</tr>
<tr><td class="mon">9</td><td class="tue">10</td><td class="wed">11</td><td class="thu">12</td><td class="fri">13</td><td class="sat">14</td><td class="sun">15</td>
</tr>
<tr><td class="mon">16</td><td class="tue">17</td><td class="wed">18</td><td class="thu">19</td><td class="fri">20</td><td class="sat">21</td><td class="sun">22</td>
</tr>
<tr><td class="mon">23</td><td class="tue">24</td><td class="wed">25</td><td class="thu">26</td><td class="fri">27</td><td class="sat">28</td><td class="sun">29</td>
</tr>
<tr><td class="mon">30</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td></tr>
</table>
</td></tr><table border="0" cellpadding="0" cellspacing="0" class="month">
<tr><th colspan="7" class="month">Dezember</th></tr>
<tr><th class="mon">Mo</th><th class="tue">Di</th><th class="wed">Mi</th><th class="thu">Do</th><th class="fri">Fr</th><th class="sat">Sa</th><th class="sun">So</th>
</tr>
<tr><td class="noday">&nbsp;</td><td class="tue">1</td><td class="wed">2</td><td class="thu">3</td><td class="fri">4</td><td class="sat">5</td><td class="sun">6</td>
</tr>
<tr><td class="mon">7</td><td class="tue">8</td><td class="wed">9</td><td class="thu">10</td><td class="fri">11</td><td class="sat">12</td><td class="sun">13</td>
</tr>
<tr><td class="mon">14</td><td class="tue">15</td><td class="wed">16</td><td class="thu">17</td><td class="fri">18</td><td class="sat">19</td><td class="sun">20</td>
</tr>
<tr><td class="mon">21</td><td class="tue">22</td><td class="wed">23</td><td class="thu">24</td><td class="fri">25</td><td class="sat">26</td><td class="sun">27</td>
</tr>
<tr><td class="mon">28</td><td class="tue">29</td><td class="wed">30</td><td class="thu">31</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td><td class="noday">&nbsp;</td>
</tr>
</table>
</td></tr></table></body>
</html>
```

## Literatur:

- Dokumentation zu *calendar*: <https://docs.python.org/3/library/calendar.html>
- Ernesti, Johannes und Peter Kaiser. *Python 3: Das umfassende Handbuch*. 5., aktualisierte Auflage. Bonn: Rheinwerk, 2017. Frei zugänglich unter: <http://openbook.rheinwerk-verlag.de/python/index.html>. Kapitel 29: Schnittstelle zu Betriebssystem und Laufzeitumgebung.

# Programmausführung

## Programme ausführen

Wenn Sie ein Programm geschrieben haben, möchten Sie es sicher auch ausführen, was Sie u.a. über die Befehlszeile Ihres Computers machen können. Bei einem Windows-PC können Sie in der Suchbox den Begriff "Eingabeaufforderung" eingeben und diese dann aus den Suchergebnissen auswählen. Alternativ können Sie auf Windows auch die integrierte PowerShell verwenden.

In macOS starten Sie das Launchpad und geben im Suchfeld "Terminal" ein. Mit der Eingabeaufforderung oder dem Terminal können Sie nun Ihren Computer über Textbefehle steuern. Nehmen wir folgendes einfache Programm als Beispiel; wie Sie sehen kann es nur den Text "Hallo, Welt!" ausgeben:

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

Hallo, Welt!

Dieses Programm speichern wir mithilfe eines Texteditors in einer Datei mit dem Namen "test.py" ab. Sie haben nun zwei Optionen, um das Programm über die Befehlszeile zu starten. Sie könnten entweder zu dem Verzeichnis navigieren, in dem sich die auszuführende Datei befindet, und dann den Dateinamen eingeben. Beachten Sie, dass Sie dem Computer zuvor noch mitteilen müssen, mit welchem Programm (hier *python*) die Datei ausgeführt werden soll:

```
python test.py
```

Alternativ geben Sie den Dateipfad an:

(Windows) `python C:\Users\Benutzername\Desktop\test.py`

(macOS/Linux/Unix) `python /Users/Benutzername/Desktop/test.py`

Sie sollte auch am Anfang Ihres Programms noch folgende Zeile einfügen: `#!/usr/bin/env python`

```
#!/usr/bin/env python

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

Hallo, Welt!

Während diese Zeile unter Windows ignoriert wird, sorgt sie unter Unix-basierten Betriebssystemen dafür, dass man den Interpreter beim Programmaufruf nicht mehr explizit benennen muss. Es ist dann also möglich, das Programm folgendermaßen aufzurufen:

```
./test.py
```

## Argumente in der Befehlszeile: `sys.argv`

Wenn Sie ein Programm über die Befehlszeile starten, können Sie diesem auch Argumente weitergeben. Dafür müssen Sie im Programm erst das Modul **sys** laden, welches dann innerhalb des Skripts über **sys.argv** die Argumente in einer Liste bereitstellt. Das erste Listenelement ist der Dateiname; die folgenden Elemente sind die eingegebenen Argumente.

```
import sys

# Zeigt Programmnamen mit Pfad
print(f"Das Programm heißt {sys.argv[0]}")

# Zeigt die Argumentenliste
print(f"{sys.argv[1:]}")

# Einzelne Argumente werden über eine Schleife wieder ausgegeben
for a in sys.argv[1:]:
    print(a)
```

```
Das Programm heißt C:\Users\qubwm01\AppData\Local\conda\conda\envs\pythoncourse
\lib\site-packages\ipykernel_launcher.py
['-f', 'C:\\\\Users\\\\qubwm01\\\\AppData\\\\Roaming\\\\jupyter\\\\runtime\\\\kernel-4c00252f-
f1f5-48e4-8aaa-4be48b0e3baa.json']
-f
C:\Users\qubwm01\AppData\Roaming\jupyter\runtime\kernel-4c00252f-f1f5-48e4-8aaa-
4be48b0e3baa.json
```

Auf meinem Windows-PC kann ich dann das Programm wie folgt in der Befehlszeile ausführen; die Wörter nach dem Dateinamen sind die Argumente, die dann ausgegeben werden sollen:

```
python C:\Users\Markus\Desktop>test.py hallo das ist ein test
```

Und hier ist die Ausgabe:

```
Die Datei heißt C:\Users\Markus\Desktop\test.py

['hallo', 'das', 'ist', 'ein', 'test']

hallo

das

ist

ein

test
```

Wenn Sie ein Argument haben, das Leerzeichen enthält, können Sie dieses in Anführungszeichen einschließen, so dass die Leerzeichen nicht als Trennzeichen zwischen einzelnen Argumenten interpretiert werden:

```
python test.py "hallo das ist ein test"
```

Das Ergebnis ist:

```
Das Programm heißt C:\Users\Markus\Desktop\test.py

['hallo das ist ein test']
```

```
hallo das ist ein test
```

## Argumente in der Befehlszeile: argparse

**sys.argv** stellt allerdings nur rudimentäre Funktionen für den Umgang mit Argumenten zur Verfügung. Das Modul **argparse** bietet Ihnen für solche Fälle eine viel größere Funktionalität, wie etwa die Möglichkeit, Optionen zu definieren. Am Beispiel eines kurzen Programms, mit dem man (sehr) einfache statistische Berechnungen durchführen kann, soll der Gebrauch des Moduls skizziert werden.

Dazu importieren wir am Anfang nicht nur **argparse**, sondern auch das Modul **statistics**, mit dem man dann auf eine Reihe von Statistikfunktionen zugreifen kann:

```
import argparse
import statistics
```

Als nächstes erzeugen wir dann eine Instanz der Klasse *ArgumentParser*:

```
parser = argparse.ArgumentParser()
```

Jetzt können wir damit beginnen, mithilfe der Methode *add\_argument* Optionen und Argumente hinzuzufügen:

```
parser.add_argument("-b", "--berechnung", dest="berechnung", default="summe",
\
                    help="Welche Berechnung soll durchgeführt werden?")
parser.add_argument("x", type=float, help="Wert 1")
parser.add_argument("y", type=float, help="Wert 2")
```

In der ersten Zeile beschreiben wir eine Option:

```
parser.add_argument("-b", "--berechnung", dest="berechnung", default="summe",
\
                    choices=["durchschnitt", "median", "summe"], \
                    help="Welche Berechnung soll durchgeführt werden?")
```

- Die ersten zwei Parameter stellen die kurze und lange Version des Optionsnamen dar, über den die Option später aufgerufen wird.
- *dest*: Der Name, unter dem der eingegebene Wert im Programm zur Verfügung stehen wird.
- *default*: Wenn beim Programmaufruf kein Wert für die Option angegeben wird, kommt stattdessen dieser voreingestellte Wert zum Zug
- *choices*: Welche Werte sind für diese Option erlaubt?
- *help*: Python erzeugt automatisch eine Hilfeseite für das Programm, auf welcher der hier angegebene Text angezeigt wird. Bei der Programmausführung können wir uns dann mit der Option *-h* oder *-help* diese Seite anzeigen lassen.

Unser Programm benötigt aber noch einige Werte, die für die Berechnungen verwendet werden können. Diese übergeben wir mit den beiden Argumenten, die wir jetzt definieren:

```
parser.add_argument("x", type=float, help="Wert 1")
parser.add_argument("y", type=float, help="Wert 2")
```

Hier legen wir fest, unter welchem Namen wir später auf den Wert zugreifen können, um welchen Datentyp es sich handelt und welcher Hilfetext für das Argument angezeigt werden soll.

Damit wir nun im Programm auf die per Optionen und Argumente übergebenen Werte zugreifen können, muss der eingegebene Befehl durch einen Parser analysiert werden, um herauszufinden, wie die einzelnen Werte behandelt werden müssen. Das können wir mit der folgenden Zeile machen:

```
args = parser.parse_args()
```

Wir können jetzt über `args.berechnung`, `args.x` und `args.y` auf die einzelnen an das Programm übergebenen Werte zugreifen und damit Berechnungen durchführen:

```
b = args.berechnung

if b == "summe":
    print(sum([args.x, args.y]))
elif b == "durchschnitt":
    print(statistics.mean([args.x, args.y]))
elif b == "median":
    print(statistics.median([args.x, args.y]))
```

Hier nun die vorläufige Version des Programms:

```
import argparse
import statistics

parser = argparse.ArgumentParser()
parser.add_argument("-b", "--berechnung", dest="berechnung", default="summe",
\
    choices=["durchschnitt", "median", "summe"], \
    help="Welche Berechnung soll durchgeführt werden?")
parser.add_argument("x", type=float, help="Wert 1")
parser.add_argument("y", type=float, help="Wert 2")
args = parser.parse_args()

b = args.berechnung

if b == "summe":
    print(sum([args.x, args.y]))
elif b == "durchschnitt":
    print(statistics.mean([args.x, args.y]))
elif b == "median":
    print(statistics.median([args.x, args.y]))
```

Wenn wir das Programm in einer Datei namens `berechnen.py` abspeichern, könnte ein Programmaufruf so aussehen:

```
python test.py -b durchschnitt 5 7
```

oder

```
python test.py --berechnung durchschnitt 5 7
```

Beachten Sie, dass Optionen immer vor Argumenten angegeben werden!

Das Programm ist aber im gegenwärtigen Zustand noch nicht besonders nützlich, da wir ihm nur zwei Zahlen übergeben können. Dies lässt sich aber ändern, indem wir nur ein Argument definieren, welches aber mehrere Werte aufnehmen kann. Dazu verwenden wir den Parameter **nargs**, der folgende Werte haben kann:

Wert	Wie viele Werte können angegeben werden	Erklärung
?	Ein oder kein Wert	
*	Beliebig viele	
+	Beliebig viele, aber mindestens einer	
N	Genau N Werte	<code>nargs=5</code> bedeutet genau 5 Werte sind möglich

Wir ändern daher unsere Argumentendefinition folgendermaßen; wir können Argument **y** jetzt weglassen:

```
parser.add_argument("-x", type=float, help="Wert 1", nargs="+")
```

Wir können jetzt zwischen einem und beliebig vielen Werten an Argument **x** übergeben. Intern werden diese Werte in einer Liste gespeichert.

Da es jetzt kein Argument **y** mehr gibt, können wir auch den letzten Teil des Programms etwas vereinfachen:

```
if b == "summe":  
    print(sum(args.x))  
elif b == "durchschnitt":  
    print(statistics.mean(args.x))  
elif b == "median":  
    print(statistics.median(args.x))
```

Hier wieder das Programm:

```
import argparse  
import statistics  
  
parser = argparse.ArgumentParser()  
parser.add_argument("-b", "--berechnung", dest="berechnung", default="summe",  
\  
    choices=["durchschnitt", "median", "summe"], \  
    help="Welche Berechnung soll durchgeführt werden?")  
parser.add_argument("-x", type=float, help="Wert 1", nargs="+")  
args = parser.parse_args()  
  
b = args.berechnung  
  
if b == "summe":  
    print(sum(args.x))  
elif b == "durchschnitt":  
    print(statistics.mean(args.x))
```

```

elif b == "median":
    print(statistics.median(args.x))

```

Schließlich ist es auch möglich, Optionen ohne einen damit assoziierten Wert zu verwenden; sie funktionieren dann als *flags*, mit denen man bei der Ausführung des Programms bestimmte Funktionen aktivieren oder deaktivieren kann. Im Beispiel haben wir die beiden *flags* **-m** und **-o**. **-m** bewirkt, dass das Ergebnis in einem vollständigen Satz ausgegeben wird, während wir mit **-o** nur die Zahl bekommen:

```

gruppe.add_argument("-m", "--mit", action="store_true", \
                    help="Mit Text")
gruppe.add_argument("-o", "--ohne", action="store_true", \
                    help="Ohne Text")

```

Der Parameter **action** bestimmt, welcher Wert mit **-m** oder **-o** verbunden werden soll. Als Standardeinstellung wird der Wert verwendet, den wir beim Programmaufruf angegeben haben. Hier legen wir aber mit `action="store_true"` fest, dass **-m** oder **-o**--wenn wir sie aufrufen--den Wert *True* haben sollen.

Wir können dann im Programm testen, ob eine der *flags* gesetzt wurde:

```

if m==True:
    print("Das Ergebnis ist {}".format(ergebnis))
else:
    print(ergebnis)

```

Eine Liste der möglichen Werte finden Sie unter  
<https://docs.python.org/3/library/argparse.html#action>.

Nun macht es keinen Sinn, beide Optionen gleichzeitig anzuwenden, da sie sich gegenseitig ausschließen. Wir können nun sicherstellen, dass auch nur eine der Optionen genutzt wird. Dazu fassen wir sie mit *add\_mutually\_exclusive\_group* zu einer Gruppe zusammen, in der nur einer der aufgeführten Optionen auf einmal aufgerufen werden kann:

```

gruppe = parser.add_mutually_exclusive_group()
gruppe.add_argument("-m", "--mit", action="store_true", \
                    help="Mit Text")
gruppe.add_argument("-o", "--ohne", action="store_true", \
                    help="Ohne Text")

```

Und nun die finale Version:

```

import argparse
import statistics

parser = argparse.ArgumentParser()
gruppe = parser.add_mutually_exclusive_group()
gruppe.add_argument("-m", "--mit", action="store_true", \
                    help="Mit Text")
gruppe.add_argument("-o", "--ohne", action="store_true", \
                    help="Ohne Text")
parser.add_argument("-b", "--berechnung", dest="berechnung", default="summe",

```

```
\n    choices=["durchschnitt", "median", "summe"], \n    help="Welche Berechnung soll durchgeführt werden?")\nparser.add_argument("x", type=float, help="Wert 1", nargs="+")\nargs = parser.parse_args()\n\nb = args.berechnung\nx = args.x\nm = args.mit\n\nif b == "summe":\n    ergebnis = sum(x)\nelif b == "durchschnitt":\n    ergebnis = statistics.mean(args.x)\nelif b == "median":\n    ergebnis = statistics.median(args.x)\n\nif m==True:\n    print("Das Ergebnis ist {}".format(ergebnis))\nelse:\n    print(ergebnis)
```

Ein Aufruf könnte so aussehen:

```
python test.py -m -b durchschnitt 1 2 3 68 28384 38 11 283
```

# Datenaustauschformate

## Datenaustausch zwischen Computersystemen

Wenn Daten zwischen unterschiedlichen Computersystemen ausgetauscht werden müssen--wie etwa zwischen Webservern unterschiedlicher Anbieter oder einem Webserver und Ihrem Computer/Tablet/Handy--dann ist es wichtig, dass sowohl Sender als auch Empfänger die Struktur verstehen, in der diese Daten bereitgestellt werden. D.h., sie müssen beispielsweise wissen, welche Teile eines Datenpakets etwa den Namen eines Elements darstellen (z.B. "Nachname") und welche Teile deren Werte (z.B. "Mustermann") repräsentieren.

Im Laufe der Zeit wurden mehrere Formate entwickelt, die standardisierte Datenstrukturierung und damit einen solchen Datenaustausch gewährleisten, unabhängig vom verwendeten Betriebssystem oder der Sprache, in der eine Anwendung geschrieben wurde.

## XML

Ein mögliches Format ist beispielsweise *XML*, die *eXtensible Markup Language*. Falls Sie schon einmal mit *HTML (HyperText Markup Language)* gearbeitet haben--etwa um eine Webseite zu erstellen--wird Ihnen die Syntax sicher bekannt vorkommen:

```
<telefonbuch>
  <person>
    <vorname>Max</vorname>
    <nachname>Mustermann</nachname>
    <wohnort>Tuebingen</wohnort>
    <nummer>07071123456</nummer>
  </person>
</telefonbuch>
```

XML kommt--neben der Kommunikation von Computer zu Computer--noch in vielen anderen Fällen zum Einsatz, etwa für die Speicherung von Metadaten. In den Geisteswissenschaften wird es beispielsweise für die Textkodierung verwendet.

XML-Dateien sind normale Textdateien und können deshalb auch mit jedem Texteditor geöffnet und bearbeitet werden. XML wird auch in vielen Programmiersprachen unterstützt, in Python bspw. durch das Paket *ElementTree*.

## XML

XML kann, im Verhältnis zu den eigentlichen Daten, relativ viele Zusatzinformationen enthalten, die nicht unbedingt benötigt werden. So muss fast jedes XML-Element explizit geschlossen werden, indem man den Namen des Elements--mit einem vorangestellten Schrägstrich--noch einmal wiederholt. Das vergrößert das Datenpaket, das übertragen werden muss.

## JSON

Ein anderer Standard, *JSON*, ist für viele Anwendungen nicht nur einfacher zu handhaben, sondern erzeugt auch kleinere Datenmengen. *JSON*, oder *JavaScript Object Notation*, orientiert sich in seiner Syntax an der Programmiersprache Javascript, speziell an deren Repräsentation von Objekten:

```
{  
    "telefonbuch":  
    {  
        "person":  
        {  
            "vorname": "Max",  
            "nachname": "Mustermann",  
            "wohnort": "Tuebingen",  
            "nummer": "07071123456"  
        }  
    }  
}
```

Wie XML auch wird JSON in regulären Textdateien gespeichert--kann deshalb auch leicht bearbeitet werden--und wird in vielen Programmiersprachen unterstützt; in Python geschieht das durch das Paket *json*. Sie werden JSON oft begegnen, wenn Sie etwa Anwendungen schreiben, die Daten von Onlinediensten über sogenannte *Application Programming Interfaces* (APIs) beziehen und verarbeiten sollen.

Das *json*-Paket bietet folgende Funktionen, um Daten im JSON-Format zu speichern und auch wieder einzulesen:

## ***json.dumps***

*json.dumps* erlaubt es Ihnen, Daten aus einer geeigneten Datenstruktur (in unserem Beispiel einem Dictionary) in eine JSON-formatierte Zeichenkette umzuwandeln. Das kann bspw. nützlich sein, wenn Sie die JSON-Daten drucken möchten oder selbst ein API anbieten wollen:

```
import json  
  
a = {"telefonbuch": {"person": {"vorname": "Max", "nachname": "Mustermann",  
    "wohnort": "Tuebingen", "nummer": "07071123456"}}}  
b = json.dumps(a)  
print(b)  
  
{"telefonbuch": {"person": {"vorname": "Max", "nachname": "Mustermann", "wohnort": "Tuebingen", "nummer": "07071123456"}}}
```

Sie können die JSON-Ausgabe über zusätzliche Parameter verändern. So lässt sich die Anzeige durch Einrückungen und Zeilenumbrüche für Menschen lesbarer gestalten. Hier wird jede Ebene wie im ersten JSON-Beispiel mit `indent=4` um vier Leerzeichen nach rechts gerückt und jedes Element in eine neue Zeile gesetzt:

```
import json  
  
a = {"telefonbuch": {"person": {"vorname": "Max", "nachname": "Mustermann",  
    "wohnort": "Tuebingen", "nummer": "07071123456"}}}
```

```

b = json.dumps(a, indent=4)
print(b)

{
    "telefonbuch": {
        "person": {
            "vorname": "Max",
            "nachname": "Mustermann",
            "wohnort": "Tuebingen",
            "nummer": "07071123456"
        }
    }
}

```

## json.dump

`json.dump` (ohne "s" am Ende) würden Sie verwenden, wenn Sie die JSON-Daten in einer Textdatei abspeichern möchten. Dazu öffnen Sie erst die Datei (bspw. mit `open()`), schreiben dann die Daten in die Datei und schließen sie dann wieder.

```

import json

a = {"telefonbuch": {"person": {"vorname": "Max", "nachname": "Mustermann",
"wohnort": "Tuebingen", "nummer": "07071123456"}}}
f = open("test.json", "w")
b = json.dump(a, f)
f.close()

```

Der folgende Code führt zum gleichen Ergebnis, allerdings müssen Sie hier die Datei nicht explizit schließen:

```

import json

a = {"telefonbuch": {"person": {"vorname": "Max", "nachname": "Mustermann",
"wohnort": "Tuebingen", "nummer": "07071123456"}}}
with open("test.json", "w") as f:
    b = json.dump(a, f)

```

Auch hier können Sie die Formatierung wieder über zusätzliche Parameter--wie etwa `indent`--beeinflussen:

```

import json

a = {"telefonbuch": {"person": {"vorname": "Max", "nachname": "Mustermann",
"wohnort": "Tuebingen", "nummer": "07071123456"}}}
with open("test.json", "w") as f:
    b = json.dump(a, f, indent=4)

```

## json.loads

Zum Einlesen von JSON-Daten bietet das `json`-Paket die Funktionen `json.loads` und `json.load`.

Wenn Sie eine Zeichenkette (oder auch *bytes*- oder *bytearray*-Objekte) mit JSON-Daten haben, können Sie diese mit *json.loads* einlesen. Im Beispiel erzeugen wir zuerst eine Zeichenkette namens *b* und laden diese dann wieder mit *json.loads* unter dem Namen *c*:

```
import json

a = {"telefonbuch": {"person": {"vorname": "Max", "nachname": "Mustermann",
"wohnort": "Tuebingen", "nummer": "07071123456"}}}
b = json.dumps(a, indent=4)

c = json.loads(b)

c
```

{'telefonbuch': {'person': {'nachname': 'Mustermann',
'nummer': '07071123456',
'vorname': 'Max',
'wohnort': 'Tuebingen'}}}

## json.load

Analog zu *json.dump* können Sie mit *json.load* JSON-Daten aus einer Datei laden. Dazu erzeugen wir erst eine Datei namens *test.json* und lesen deren Inhalt dann wieder ein:

```
import json

a = {"telefonbuch": {"person": {"vorname": "Max", "nachname": "Mustermann",
"wohnort": "Tuebingen", "nummer": "07071123456"}}}
with open("test.json", "w") as f:
    json.dump(a, f, indent=4)

with open("test.json", "r") as f:
    c = json.load(f)

print(c)

{'telefonbuch': {'person': {'vorname': 'Max', 'nachname': 'Mustermann', 'wohnort': 'Tuebingen', 'nummer': '07071123456'}}}
```

# Einführung in Reguläre Ausdrücke

## Reguläre Ausdrücke

Bei der Arbeit mit Datensätzen kann es vorkommen, dass Sie bestimmte Informationen (z.B. Rentenversicherungsnummern oder Emailadressen) entfernen müssen. Oder vielleicht möchten Sie eine Liste aller Jahreszahlen aus einem Roman erstellen. Hier können Ihnen reguläre Ausdrücke helfen.

Ein regulärer Ausdruck (*regular expression, regexp, regex*) ist eine Zeichenkette aus Buchstaben, Zahlen und verschiedenen Sonderzeichen. Er definiert ein Muster, das dem einer gesuchten Zeichenkette entspricht. Anstatt also nach einer bestimmten Emailadresse zu suchen--ein Vorgang, den Sie für jede Adresse wiederholen müssten--können Sie nach dem Muster suchen, was wesentlich effektiver und effizienter ist.

## Entwicklung regulärer Ausdrücke

Die Anfänge der regulären Ausdrücke gehen bis in die 1940er Jahre zurück, als die Amerikaner Warren McCulloch und Walter Pitts versuchten, Modelle für die Arbeitsweise des Nervensystems zu entwickeln. 1956 beschrieb dann der Mathematiker Steven Kleene eine Algebra für diese Modelle, die er reguläre Mengen (*regular sets*) nannte. Die Notation für diese neue Algebra waren die regulären Ausdrücke.

Wahrscheinlich 1968 wurden reguläre Ausdrücke zum ersten Mal in einem Computersystem implementiert.

1987 wurde die erste Version der Programmiersprache Perl veröffentlicht, die für die Bearbeitung von Textdateien entwickelt worden war und eine für die damalige Zeit leistungsfähige Implementierung von regulären Ausdrücken enthielt; diese wurde im Laufe der Jahre weiter ausgebaut.

1997 erschien schließlich PCRE (*Perl Compatible Regular Expressions*), eine Programmbibliothek, welche es ermöglicht, die zum damaligen Zeitpunkt aktuelle Funktionalität von Perls regulären Ausdrücken (Perl 5) in andere Programme und Sprachen zu integrieren.

## Verbreitung regulärer Ausdrücke

Heutzutage unterstützen alle gängigen Programmiersprachen reguläre Ausdrücke in irgendeiner Form. Auch viele Anwendungsprogramme, wie etwa Texteditoren, erlauben es Ihnen, beim Suchen/Ersetzen reguläre Ausdrücke zu verwenden.

## Reguläre Ausdrücke testen

Es gibt eine Vielzahl von Webseiten, auf denen Sie reguläre Ausdrücke testen können. Wir verwenden heute <https://regex101.com/>. Es bietet auch gute (englischsprachige) Erklärungen und

zeigt Ihnen eventuelle Fehler an.

## Literele Zeichenmuster

Literele Zeichen sind Zeichen, die sich selbst bezeichnen und sonst keine weitere Bedeutung haben. Wenn wir solche Zeichen in einem regulären Ausdruck verwenden, sprechen wir von **literalen Zeichenmustern** (*literal character patterns*). Z.B. passt das literale Zeichenmuster "abc" auf die Zeichenkette "abc" im Suchtext und nur auf sie. Das Ergebnis unterscheidet sich hierbei nicht von einer Suche ohne reguläre Ausdrücke.

## Metazeichen

Metazeichen haben eine Bedeutung, die sich von ihrer literalen unterscheidet, und üben besondere Funktionen innerhalb von regulären Ausdrücken aus. Hier ein paar Beispiele:

Zeichen	Beispiel	Bedeutung
^	^abc	"abc" am Anfang der untersuchten Zeichenkette
\\$	abc\\$	"abc" am Ende der untersuchten Zeichenkette
\	\w	Escapezeichen (dazu später mehr)
. (Punkt)	a.c	Kann irgendein Zeichen sein (Buchstabe, Zahl, Satzzeichen, ...)
	(a c)	Entweder "a" oder "c"

## Quantoren

Quantoren geben an, wie oft ein Zeichen vorkommen soll:

Zeichen	Beispiel	Bedeutung
?	a?	Null oder ein "a"
*?	a*?	Null oder mehr "a", versucht nur so viele wie unbedingt nötig zu finden: ( <i>lazy</i> )
*	a*	Null oder mehr "a", versucht so viele wie möglich zu finden: gierig ( <i>greedy</i> )
+	a+	Ein oder mehr "a"

Sie können aber auch spezifischere Mengenangaben machen, indem Sie diese in geschweifte Klammern setzen:

Beispiel	Bedeutung
a{2}	Genau zwei a
a{2,}	Zwei oder mehr a
a{2,4}	Zwei bis vier a

## Metazeichen als Literale

Wenn Sie nach einem Metazeichen suchen wollen, müssen Sie Ihrem Programm zeigen, dass es sich dann nicht um ein Metazeichen, sondern ein Literal handelt. Sie können das mit einem vorangestellten *backslash* machen. Wenn Sie etwa nach einem Dollarzeichen suchen, würden Sie folgenden Ausdruck verwenden:

\\$

Wenn Sie nach einem *backslash* suchen, benötigen Sie also tatsächlich zwei:

\\"

## Klassen

Mit Klassen kann man eine limitierte Gruppe von Zeichen definieren, die der reguläre Ausdruck finden soll. Klassen werden in eckige Klammern gesetzt und finden immer ein einzelnes Zeichen, außer Sie fügen einen Quantor hinzu.

Sie können Klassen etwa verwenden, wenn Sie nach einem Namen suchen, bei dessen Schreibweise Sie sich nicht sicher sind. So findet etwa der folgende Ausdruck vier verschiedene Schreibweisen von Maier/Mayer/Meier/Meyer:

M[ae][iy]er

Es gibt auch vordefinierte Klassen für Zeichenkategorien, welche besonders häufig genutzt werden, wie etwa:

Name	Bedeutung	Steht für
[:alpha:]	Buchstaben	[a-zA-Z]
[:upper:]	Großbuchstaben	[A-Z]
[:lower:]	Kleinbuchstaben	[a-z]
[:alnum:]	Buchstaben und Zahlen	[a-zA-Z0-9]
[:digit:]	Zahlen	[0-9]

## Zeichen ausschließen

Wenn Sie ein oder mehrere Zeichen ausschließen möchten, können Sie diese in einer Klasse zusammenfassen und dann mit einem ^ verneinen:

[^abc]

Es gibt für viele dieser Klassen auch Kurzbezeichnungen:

Abkürzung	Steht für
\w	Buchstaben, Zahlen, Unterstrich
\W	Alles außer Buchstaben, Zahlen, Unterstrich
\d	Zahlen

Abkürzung	Steht für
\D	Alles außer Zahlen
\s	<i>Whitespace</i> (Nicht sichtbare Zeichen, wie etwa Leerzeichen, Zeilenumbruch oder Tabulator)
\S	Alles außer <i>Whitespace</i>
\b	Wortgrenze
\B	Keine Wortgrenze

Auch wichtige Kontrollzeichen haben eigene Kurzbezeichnungen:

Abkürzung	Steht für
\t	Tabulator
\r	Wagenrücklauf ( <i>Carriage return</i> )
\n	Zeilenumbruch ( <i>Line feed</i> )

**Wichtig:** In Windows erstellte Textdateien verwenden \r\n, um das Ende einer Zeile zu signalisieren. macOS und Unix verwenden \n.

## Übung 1

Schreiben Sie einen regulären Ausdruck, um eine Rentenversicherungsnummer zu finden. Diese ist folgendermaßen aufgebaut (laut <https://de.wikipedia.org/wiki/Versicherungsnummer>):

Stelle	Beschreibung	Beispiel
1–2	Bereichsnummer des Rentenversicherungsträgers	15
3–4	Geburtstag des Versicherten	07
5–6	Geburtsmonat des Versicherten	06
7–8	Geburtsjahr des Versicherten	49
9	Anfangsbuchstabe des Geburtsnamens des Versicherten	C
10–11	Seriennummer	10
12	Prüfziffer	3

## Mögliche Lösung

\d{8}[a-zA-Z]\d{3}

## Übung 2

Schreiben Sie einen regulären Ausdruck, der Ihre studentische Emailadresse finden könnte.

## Mögliche Lösung

[a-zA-Z\.\.]\*@[a-zA-Z\.\-]\*

# Gruppen

Durch die Verwendung von runden Klammern (einfangende Klammern) können Sie Gruppen bilden, deren Treffer eingefangen, oder abgespeichert, werden und auf die Sie dann später direkt zugreifen können. Wenn Sie die Ergebnisse nicht speichern möchten--weil Sie etwa nur anzeigen wollen, dass Sie mehrere Alternativen benennen möchten--können Sie nicht-einfangende Klammern verwenden:

Beispiel	Ergebnis
(hund katze)	Findet "hund" oder "katze", wird gespeichert
(?:hund katze)	Findet "hund" oder "katze", wird nicht gespeichert

# Rückreferenzierung

Wenn Sie einfangende Klammern verwenden, können Sie mit **Rückreferenzierung** (*backreferences*) später im gleichen regulären Ausdruck wieder auf die gleiche gefundene Zeichenkette zurückgreifen:

Beispiel	Ergebnis
(hund katze)	Findet "hund" oder "katze", wird gespeichert
(hund katze) sieht \1	Findet "hund sieht hund", aber nicht "hund sieht katze"

# Lookaround, lookahead und lookbehind

*Lookaround* beinhaltet zwei Techniken, *lookahead* und *lookbehind*. Wenn Sie etwa testen wollen, ob nach einem Zeichen 1 ein anderes Zeichen 2 folgt, können Sie einen positiven *lookahead* verwenden. Wenn nach Zeichen 1 kein Zeichen 2 kommen darf, verwenden Sie einen negativen *lookahead*:

Beispiel	Bedeutet
a(?=b)	nach dem "a" muss ein "b" kommen
a(?!b)	nach dem "a" darf kein "b" kommen

Der Inhalt der runden Klammer wird dabei nicht eingefangen.

*lookbehind* ist die Umkehrung von *lookahead*:

Beispiel	Bedeutet
(?<=a)b	vor dem "b" muss ein "a" kommen
(?<!a)b	vor dem "b" darf kein "a" kommen

**Wichtig:** Mehrere RegEx-Implementierungen erlauben nur *lookbehind* mit Zeichenketten, welche eine festgesetzte Länge haben.

# Reguläre Ausdrücke in Python

## Reguläre Ausdrücke in Python

Python hat ein separates Modul für die Arbeit mit regulären Ausdrücken: `re`. Wir müssen es deshalb zuerst importieren:

```
import re
```

Jetzt können wir dessen Funktionen nutzen. Hier ist eine Zeichenkette, in der wir mit `search()` suchen können:

*Das hier ist ein Beispielsatz, in dem es einige Zahlen gibt. Diese Zahlen sind 2, 25, 67, 89 und 105.*

```
a = ("Das hier ist ein Beispielsatz, in dem es einige Zahlen gibt. Diese  
Zahlen sind 2, 25, 67, 89 und 105.")
```

## Backslash

Beachten Sie bitte, wie Python mit *Backslashes* umgeht (\). Wie Sie schon wissen, müssen Sie in einem regulären Ausdruck einen doppelten Backslash verwenden, wenn Sie nach einem einfachen *Backslash* suchen wollen. Python möchte auch, dass Sie einen doppelten *Backslash* benutzen, wenn er eigentlich als Metazeichen dient. Sie können einen regulären Ausdruck aber mit einem vorangestellten "r" als *Raw String* schreiben, wo dann ein einfacher Backslash genügt.

Dies kann man auch mit regulären Zeichenketten machen, welche einen *Backslash* enthalten sollen, um diese übersichtlicher zu gestalten.

## search()

Als *Raw String*:

```
x = re.search(r"\b(\w){12}\b", a)  
x
```

```
<re.Match object; span=(17, 29), match='Beispielsatz'>
```

Mit doppeltem *Backslash*:

```
y = re.search("\b(\w){12}\b", a)  
y
```

```
<re.Match object; span=(17, 29), match='Beispielsatz'>
```

## MatchObject

Wenn Python mit `search` einen Treffer findet, gibt es das erste Ergebnis in einem sogenannten `MatchObject` zurück. Sie können dann auf verschiedene Elemente dieses Objekts zugreifen:

Funktion	Beispiel	Bedeutet
group()	x.group()	Gibt den Teil der Zeichenkette zurück, auf den der gesamte reguläre Ausdruck zutrifft
group(ZAHL)	x.group(3)	Gibt das Ergebnis von Gruppe 3 innerhalb des regulären Ausdrucks zurück
start()	x.start()	Gibt die Position des ersten Zeichens an, auf den der reguläre Ausdruck zutrifft
end()	x.end()	Gibt die Position des ersten Zeichens an, das auf den vom regulären Ausdruck gefundenen Teil der Zeichenkette folgt
span()	x.span()	Gibt die Werte von <code>start</code> und <code>end</code> in einem Tuple an

## findall()

Während `search()` nur den ersten Treffer findet, können Sie mit `findall()` alle Treffer ausgeben lassen.

```
x = re.findall(r"\b\d{2}\b", a)
x
```

['25', '67', '89']

## match()

`match()` arbeitet ähnlich wie `search()`, mit dem Unterschied, dass `match()` nur am Anfang der Zeichenkette sucht:

```
x = re.match(r"(\b(\w){12}\b)", a)
x
```

```
x = re.match(r"(\w)*", a)
x
```

<re.Match object; span=(0, 3), match='Das'>

## fullmatch()

`fullmatch()` findet nur ein Ergebnis, wenn der reguläre Ausdruck auf die gesamte Zeichenkette zutrifft:

```
x = re.fullmatch(r"(\w)*", a)
x
```

```
x = re.fullmatch(r"(\w*)", "NurEinWort")
x
```

<re.Match object; span=(0, 10), match='NurEinWort'>

## sub()

Mit **sub()** können Sie Teile von Zeichenketten ersetzen:

```
x = re.sub(r"\b[a-zA-Z]{3}\b", "XXX", a)  
x
```

```
'XXX hier XXX XXX Beispielsatz, in XXX es einige Zahlen gibt. Diese Zahlen sind  
2, 25, 67, 89 XXX 105.'
```

Sie können beim Ersetzen auch Gruppen verwenden; auf diese Weise kann der durch einen regulären Ausdruck gefundene Text auch Teil des neuen Texts sein:

```
x = re.sub(r"(\b[a-zA-Z]{3}\b)", "XXX (Original: \g<1>)", a)  
x
```

```
'XXX (Original: Das) hier XXX (Original: ist) XXX (Original: ein) Beispielsatz,  
in XXX (Original: dem) es einige Zahlen gibt. Diese Zahlen sind 2, 25, 67, 89 XX  
X (Original: und) 105.'
```

Das gleiche Ergebnis können Sie auch mit benannten Gruppen erzielen, bei denen Sie dann anstatt der Position der Gruppe im regulären Ausdruck einen Namen verwenden können, um auf deren Wert zu verweisen:

```
x = re.sub(r"(?P<kurzeswort>\b[a-zA-Z]{3}\b)", "XXX [Original:  
\g<kurzeswort>]", a)  
x
```

```
'XXX [Original: Das] hier XXX [Original: ist] XXX [Original: ein] Beispielsatz,  
in XXX [Original: dem] es einige Zahlen gibt. Diese Zahlen sind 2, 25, 67, 89 XX  
X [Original: und] 105.'
```

## split()

**split()** erlaubt es Ihnen, Zeichenketten aufzubrechen. Sie können dabei einen regulären Ausdruck als Trennzeichen verwenden. Hier wird ein Leerzeichen verwendet, vor dem ein Komma oder Punkt stehen können:

```
x = re.split(r"[,\.]\s?", a)  
x
```

```
['Das',  
'hier',  
'ist',  
'ein',  
'Beispielsatz',  
'in',  
'dem',  
'es',  
'einige',  
'Zahlen',  
'gibt',  
'Diese',  
'Zahlen',  
'sind',
```

```
'2',
'25',
'67',
'89',
'und',
'105. ']
```

## compile()

Mit **compile()** können Sie wiederholt benötigte reguläre Ausdrücke zu einem *RegexObject* kompilieren und damit deren Bearbeitung beschleunigen:

```
x = re.compile(r"(\b[a-zA-Z]{3}\b)")
x.findall(a)
```

```
['Das', 'ist', 'ein', 'dem', 'und']
```

## Weiterführende Lektüre

<https://www.regular-expressions.info> bietet eine kostenlose, umfangreiche und leicht verständliche (englischsprachige) Einführung zu regulären Ausdrücken.

Friedl, Jeffrey E.F. *Reguläre Ausdrücke*. O'Reilly: 2008. UB Druckausgabe Deutsch: 48 A 1671. eBook über die UB: <https://learning.oreilly.com/library/view/-/9783868996753/?ar>



# Datenvisualisierung



## Was ist Datenvisualisierung?

„Datenvisualisierung ist **teils Kunst und teils Wissenschaft**. Die Herausforderung besteht darin, die Kunst richtig zu machen, ohne die Wissenschaft falsch zu machen, und umgekehrt. Eine Datenvisualisierung muss in erster Linie **die Daten präzise darstellen**. Sie darf **nicht irreführen oder verfälschen**. [...] Gleichzeitig soll eine Datenvisualisierung **ästhetisch ansprechend sein**. Optisch gelungene Präsentationen unterstreichen die Aussagekraft der Datenvisualisierung.“ (Wilke, 1)

Wilke, Claus O. *Datenvisualisierung: Grundlagen und Praxis*. Übersetzt von Bilgehan Gür. Heidelberg: dpunkt.verlag, 2020.



## Datenvisualisierung...

... hilft uns, aus Daten gewonnene Erkenntnisse besser zu verstehen und zu vermitteln:

Die Wissenschaft hat zwei Hindernisse vor sich, die ihren Fortschritt behindern: erstens die Unfähigkeit unserer Sinne, Wahrheiten zu entdecken, und zweitens die Unzulänglichkeit der Sprache, um die gewonnenen Wahrheiten auszudrücken und zu vermitteln. Der Zweck wissenschaftlicher Methoden ist es, diese Hindernisse zu beseitigen; die Grafische Methode erreicht dieses doppelte Ziel besser als jede andere.

Étienne-Jules Marey. *La méthode graphique dans les sciences expérimentales et principalement en physiologie et en médecine*. Paris: 1878. S. I. <https://gallica.bnf.fr/ark:/12148/bpt6k6211376f/>



## Datenvisualisierung als Analysewerkzeug

Das Anscombe-Quartett (*Anscombe's Quartet*) besteht aus Vier Mengen mit denselben einfachen statistischen Eigenschaften:

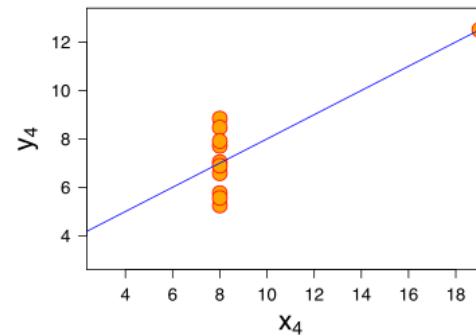
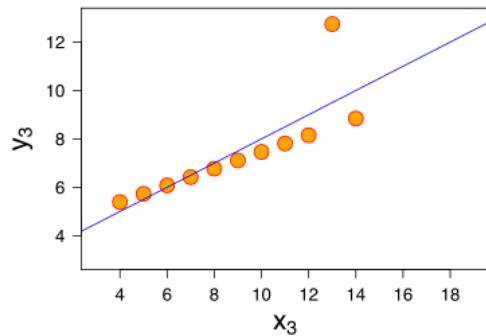
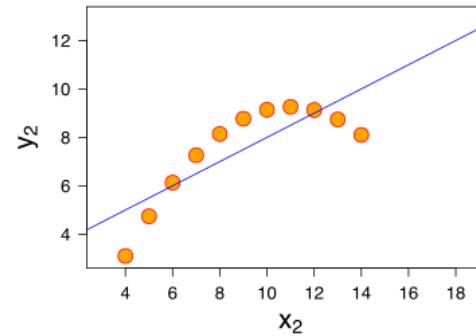
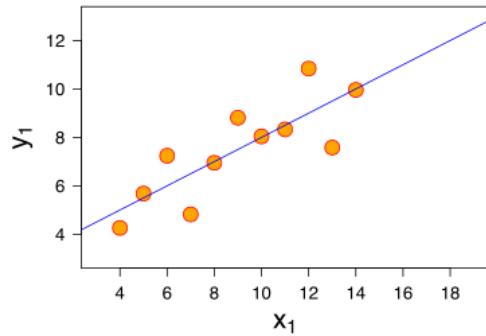
I		II		III		IV	
x	y	x	y	x	y	x	y
4,0	4,26	4,0	3,10	4,0	5,39	8,0	5,25
5,0	5,68	5,0	4,74	5,0	5,73	8,0	5,56
6,0	7,24	6,0	6,13	6,0	6,08	8,0	5,76
7,0	4,82	7,0	7,26	7,0	6,42	8,0	6,58
8,0	6,95	8,0	8,14	8,0	6,77	8,0	6,89
9,0	8,81	9,0	8,77	9,0	7,11	8,0	7,04
10,0	8,04	10,0	9,14	10,0	7,46	8,0	7,71
11,0	8,33	11,0	9,26	11,0	7,81	8,0	7,91
12,0	10,84	12,0	9,13	12,0	8,15	8,0	8,47
13,0	7,58	13,0	8,74	13,0	12,74	8,0	8,84
14,0	9,96	14,0	8,10	14,0	8,84	19,0	12,50

Quelle: <https://de.wikipedia.org/wiki/Anscombe-Quartett>



# Datenvisualisierung als Analysewerkzeug

Große Unterschiede in der visuellen Darstellung:



Quelle: [https://commons.wikimedia.org/wiki/File:Anscombe%27s\\_quartet\\_3.svg](https://commons.wikimedia.org/wiki/File:Anscombe%27s_quartet_3.svg)



# Datenvisualisierung als Kommunikationswerkzeug



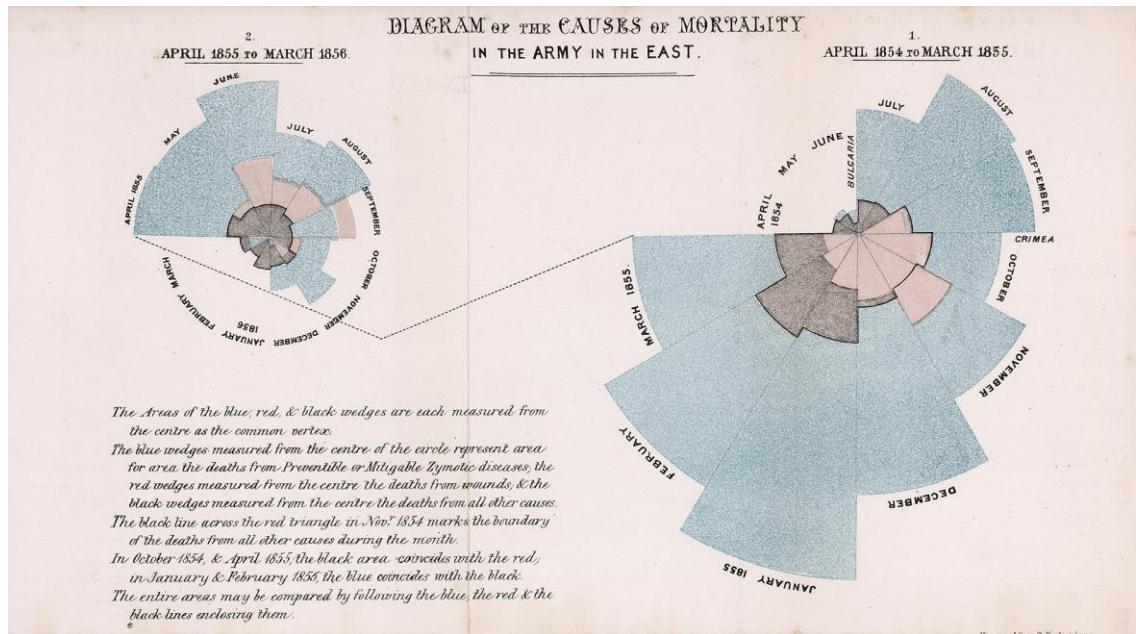
John Snow und Charles Cheffins.  
Cholerafälle während der Londoner  
Epidemie von 1854.

Snow entdeckte, dass Cholera durch  
verseuchtes Wasser verursacht wird  
und konnte den Ausbruch letztendlich  
stoppen. Er gilt heute als der  
Begründer der modernen  
Epidemiologie.

Quelle: [https://en.wikipedia.org/wiki/John\\_Snow#/media/File:Snow-cholera-map-1.jpg](https://en.wikipedia.org/wiki/John_Snow#/media/File:Snow-cholera-map-1.jpg)



# Datenvisualisierung als Kommunikationswerkzeug



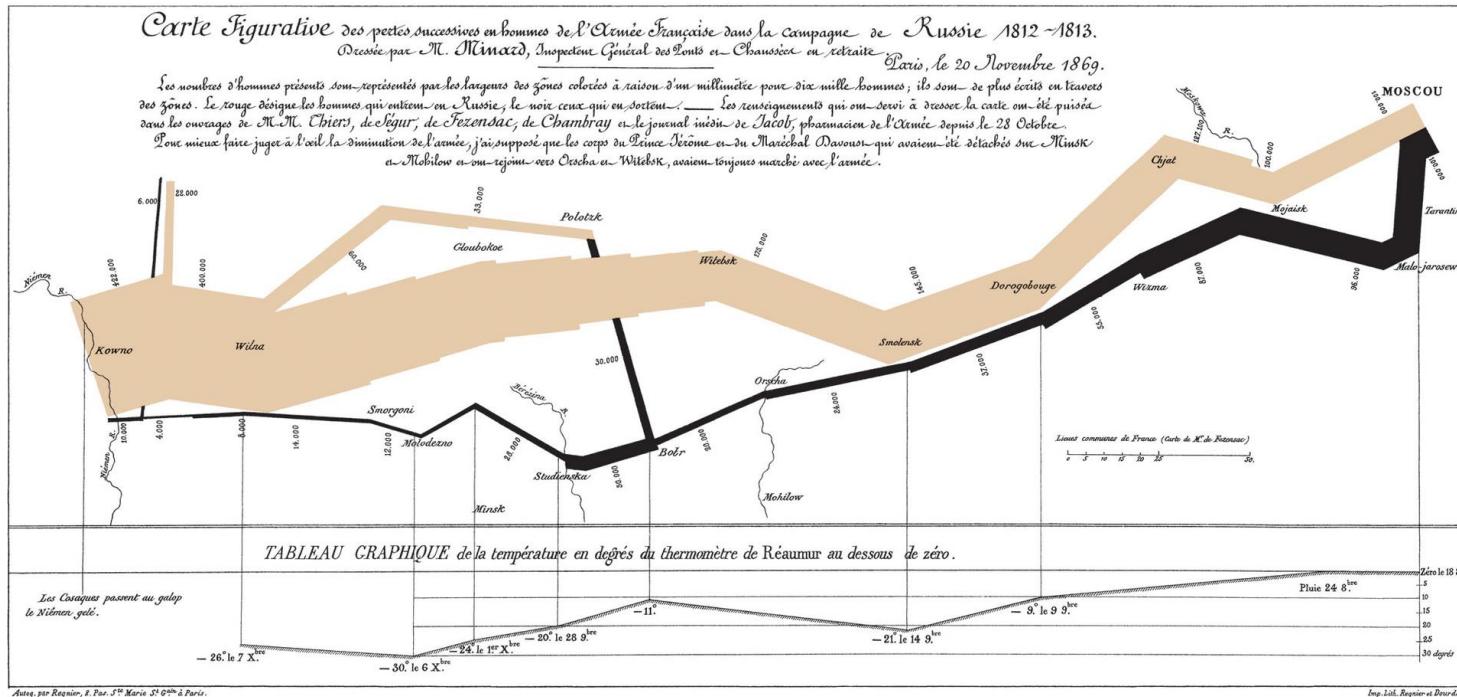
Florence Nightingale, *Diagram of the Causes of Mortality in the Army in the East (1858)*

Nightingale war während des Krimkriegs (1853-1856) Krankenpflegerin. Sie gilt als Pionierin auf dem Gebiet der Datenvisualisierung und Begründerin der modernen Krankenpflege.

Quelle: [https://en.wikipedia.org/wiki/Florence\\_Nightingale#/media/File:Nightingale-mortality.jpg](https://en.wikipedia.org/wiki/Florence_Nightingale#/media/File:Nightingale-mortality.jpg)



# Datenvisualisierung als „Story-Telling“-Werkzeug



Charles Joseph Minard. *Carte figurative des pertes successives en hommes de l'Armée Française dans la campagne de Russie 1812–1813* (1869)

Edward Tufte: *It may well be the best statistical graphic ever drawn.* (40)

Quelle: [https://de.wikipedia.org/wiki/Charles\\_Joseph\\_Minard#/media/Datei:Minard.png](https://de.wikipedia.org/wiki/Charles_Joseph_Minard#/media/Datei:Minard.png)



---

# Was benötigt man für die Datenvisualisierung?



---

**Was benötigt man für die Datenvisualisierung?**

# Daten



---

Die zur Verfügung stehenden Daten bestimmen maßgeblich, welche Darstellungsform man in der Visualisierung wählt.

Ein wichtiger Faktor dabei sind die zur Verfügung stehenden Variabtentypen:

Variabtentyp	Beispiel	Skala	Beschreibung
Quantitativ/ Numerisch kontinuierlich	1,3; 5,7; 83; $15 \times 10^{-2}$	Kontinuierlich	Beliebige numerische Werte
Quantitativ/ Numerisch diskret	1; 2; 3; 4	Diskret	Zahlen in diskreten Einheiten [zwischen den Zahlen können im Datensatz keine Werte vorkommen]

Nach Wilke, S. 9.



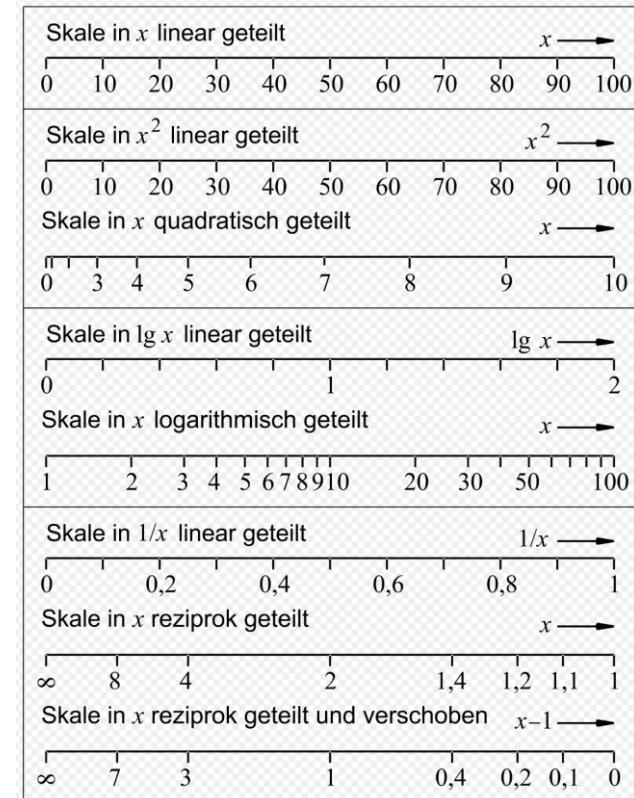
Variabtentyp	Beispiel	Skala	Beschreibung
Qualitativ/ kategorisch ungeordnet	Hund, Katze, Fisch	Diskret	Diskrete und eindeutige Katego- rien ohne feste Reihenfolge. Diese Variablen werden auch als <i>Merkmale</i> bezeichnet.
Qualitativ/ kategorisch geordnet	gut, angemessen, schlecht	Diskret	Diskrete und eindeutige Katego- rien mit fester Reihenfolge. Diese Variablen werden auch als <i>geordnete Merkmale</i> bezeichnet.
Datum/Zeit	5. Jan 2018, 08:30:25	Kontinu- ierlich, diskret	Spezifische Tage und/oder Zeiten.
Text	Das ist ein Satz.	Keine oder diskret	Freiformtext. Kann bei Bedarf als kategorisierbar behandelt werden.

## Daten – Gestaltungselemente - Skalen

Daten werden bei der Visualisierung über Skalen verschiedenen Gestaltungselementen (Punkt, Linie, Fläche, Körper) zugewiesen.

Diese Skalen müssen eindeutig sein und es gibt ein Gestaltungselement pro dargestellten Datenpunkt.

**Wichtig bei Skalen:** Sind sie linear/quadratisch/logarithmisch/...?

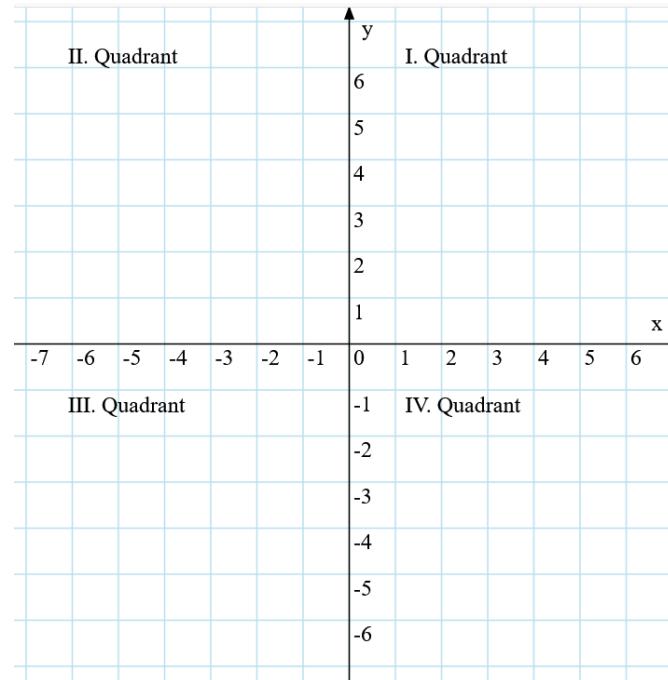


Quelle: <https://de.wikipedia.org/wiki/Skale#/media/Datei:Skalenverl%C3%A4ufe.svg>



# Koordinatensystem

Die meisten Datenvisualisierungen, denen wir im Alltag begegnen, verwenden das kartesische Koordinatensystem mit einer x- und y-Achse (gegebenenfalls auch eine z-Achse für die Darstellung einer dritten Dimension).

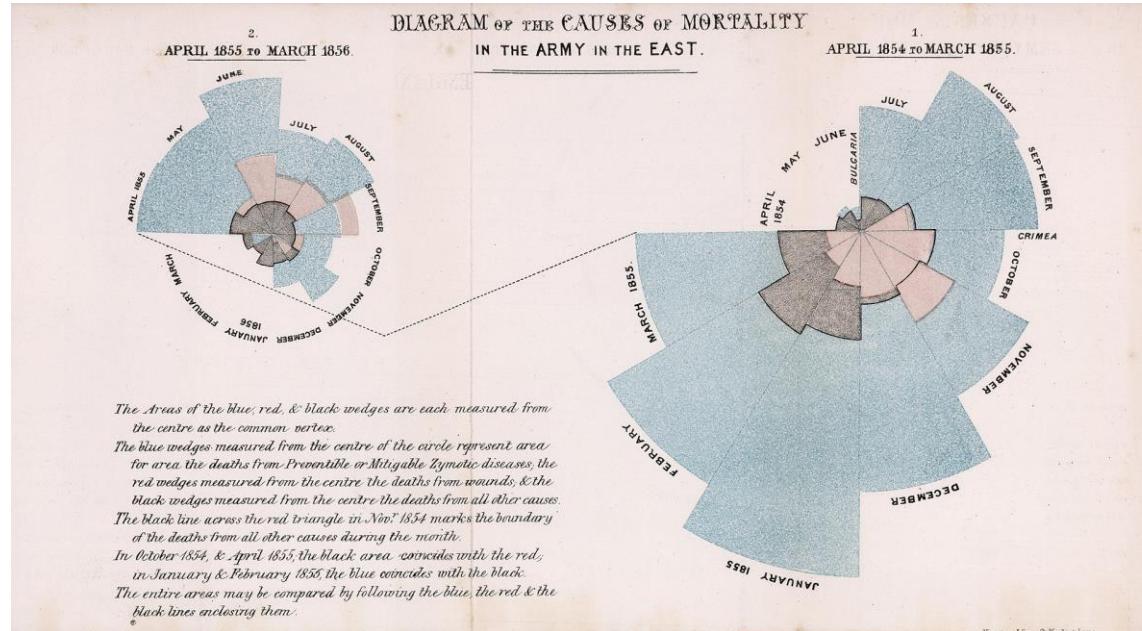


[https://de.wikipedia.org/wiki/Kartesisches\\_Koordinatensystem#/media/Datei:Kartesisches\\_system.svg](https://de.wikipedia.org/wiki/Kartesisches_Koordinatensystem#/media/Datei:Kartesisches_system.svg)



# Koordinatensystem

Eine andere Möglichkeit ist ein Polar- oder Kreiskoordinatensystem, wie wir es schon im Fall von Florence Nightingales Visualisierung gesehen haben:

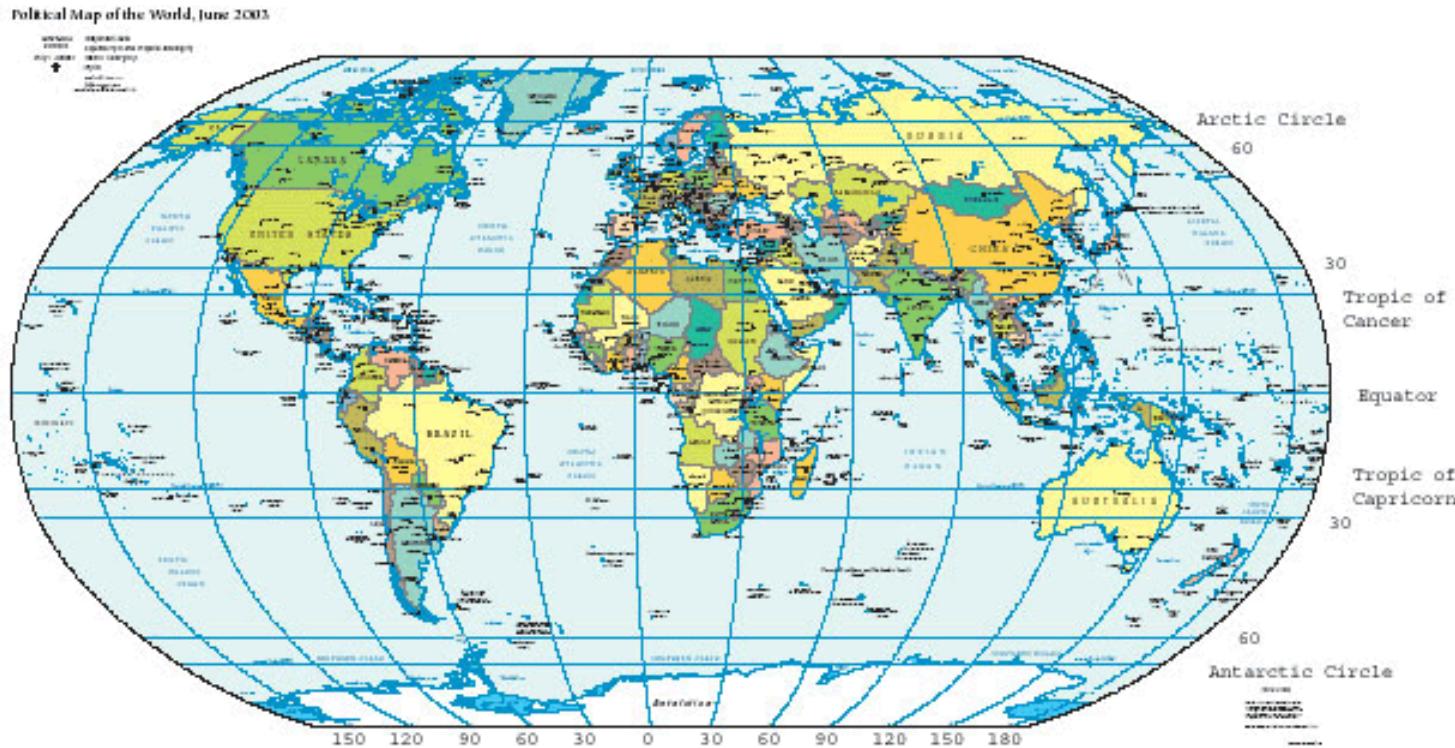


Quelle: [https://en.wikipedia.org/wiki/Florence\\_Nightingale#/media/File:Nightingale-mortality.jpg](https://en.wikipedia.org/wiki/Florence_Nightingale#/media/File:Nightingale-mortality.jpg)



# Koordinatensystem

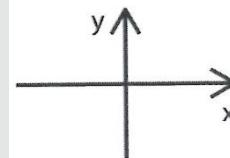
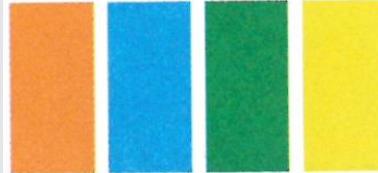
Schließlich gibt es auch geographische Koordinaten, wie wir sie in Karten verwenden:



Quelle: <https://upload.wikimedia.org/wikipedia/commons/a/ab/WorldMapLongLat-eq-circles-tropics-non.png>

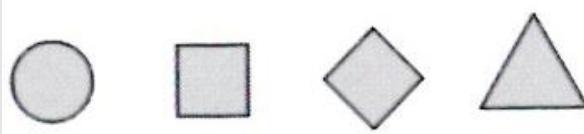
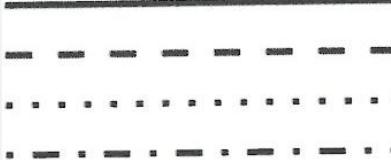


# Gestaltungselemente (Aesthetics)

Element	Beispiel	Kontinuierlich	Diskret
Position		✓	✓
Größe		✓	✓
Farbe		✓	✓
Linienbreite		✓	✓



## Gestaltungselemente (Aesthetics)

Element	Beispiel	Kontinuierlich	Diskret
Form			<input checked="" type="checkbox"/>
Linientyp			<input checked="" type="checkbox"/>

Nach Wilke, S. 8.



## Farben

Farben dienen dazu:

- Datengruppen zu unterscheiden
- Datenwerte darzustellen
- Datenwerte hervorzuheben

Programme und Pakete für Programmiersprachen (z.B. *ColorBrewer*) bieten abgestimmte Farbskalen an, die wir für Visualisierungen verwenden können.

ColorBrewer. <https://colorbrewer2.org/>

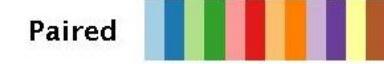
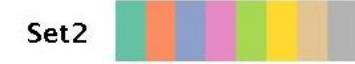
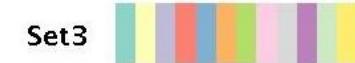


## Farbskalen

### Qualitative Farbskala

- Zur Unterscheidung von Datengruppen
- Keine implizierte Reihen- oder Rangfolge
- Farben sollen sich deutlich unterscheiden, keine soll herausstechen

### Qualitative



<https://de.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/34087/versions/2/screenshot.jpg>



## Farbskalen

### Sequenzielle Farbskala

- Zur Darstellung von Datenwerten
- Entweder verschiedene Helligkeitsstufen eines Farbtöns oder mehrere Farbtöne (z.B. Hellgelb bis Dunkelrot)
- Z.B. bei Karten, in denen Regionen auf der Basis von Datenwerten unterschiedlich eingefärbt werden (*Choroplethen*)



<https://de.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/34087/versions/2/screenshot.jpg>



## Farbskalen

### Divergente Farbskala

- Wenn es einen Mittelwert gibt (z.B. 0 Grad Celsius bei der Darstellung von Temperaturen)
- Zwei Farbtöne
- Zur Mitte hin heller, nach außen dunkler



<https://de.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/34087/versions/2/screenshot.jpg>



# Farbskalen

## Akzentfarbskala

- Wenn man bei mehreren möglichen Kategorien eine oder ein paar wenige herausheben möchte
- Mischung aus gedämpften und herausstechenden Farben

Okabe-Ito-Akzent



Grau mit Akzenten



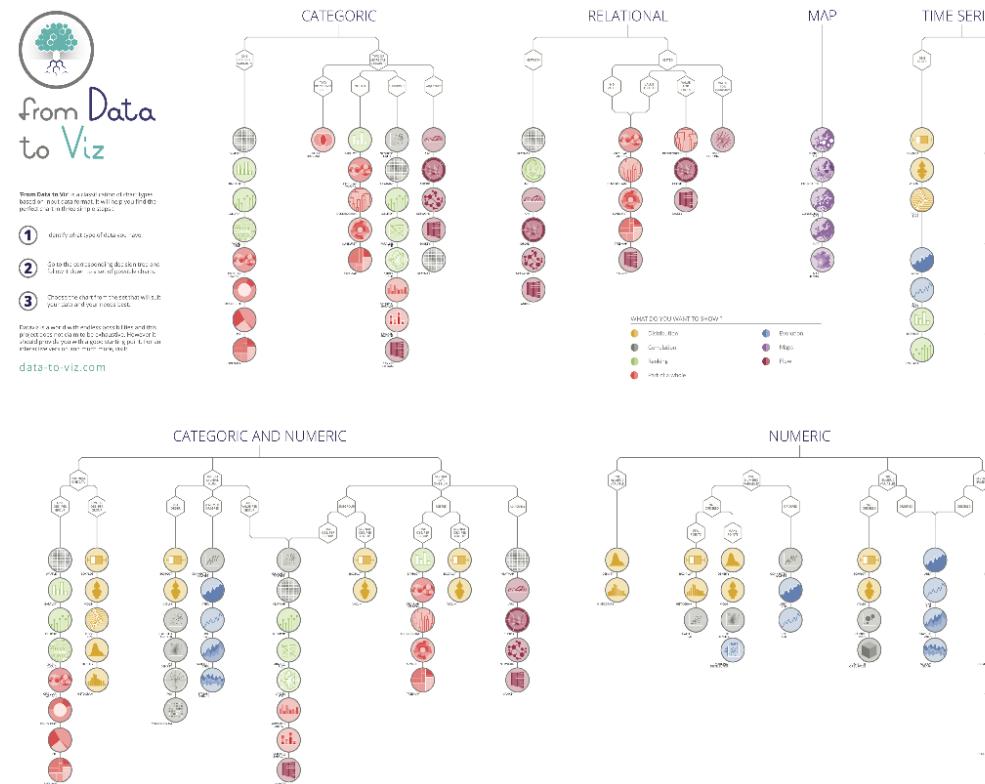
ColorBrewer-Akzent



Wilke, S. 30.

# Welche Art von Diagramm soll ich wählen?

<https://www.data-to-viz.com/>





## Welche Art von Diagramm soll ich wählen?

Diese Seiten bieten auch Leitfäden sowie Codebeispiele in Python bzw. R:

- <https://python-graph-gallery.com>
- <https://r-graph-gallery.com>





## Fallstricke

Datenvisualisierungen können Daten misrepräsentieren. Dies kann aus Unachtsamkeit oder Unwissenheit geschehen; in manchen Fällen sind solche Fehler aber auch beabsichtigt, um das Publikum zu manipulieren.

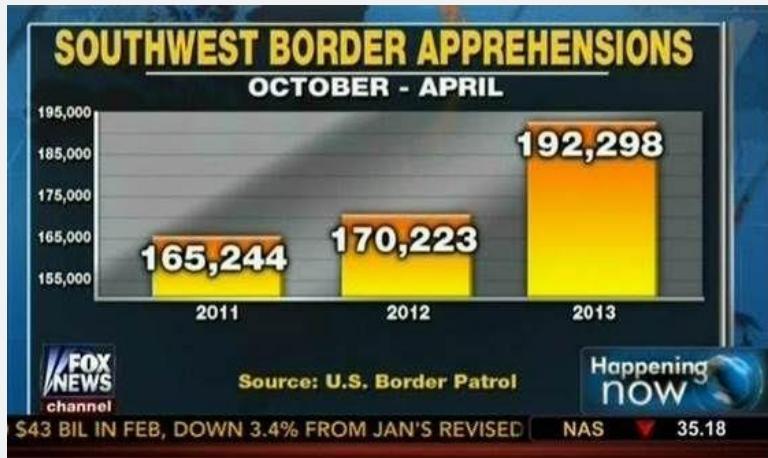
Im Folgenden werden ein paar häufige Probleme bei Datenvisualisierungen besprochen. Wenn nicht anders ausgewiesen stammen die Beispiele von <https://www.data-to-viz.com/>.



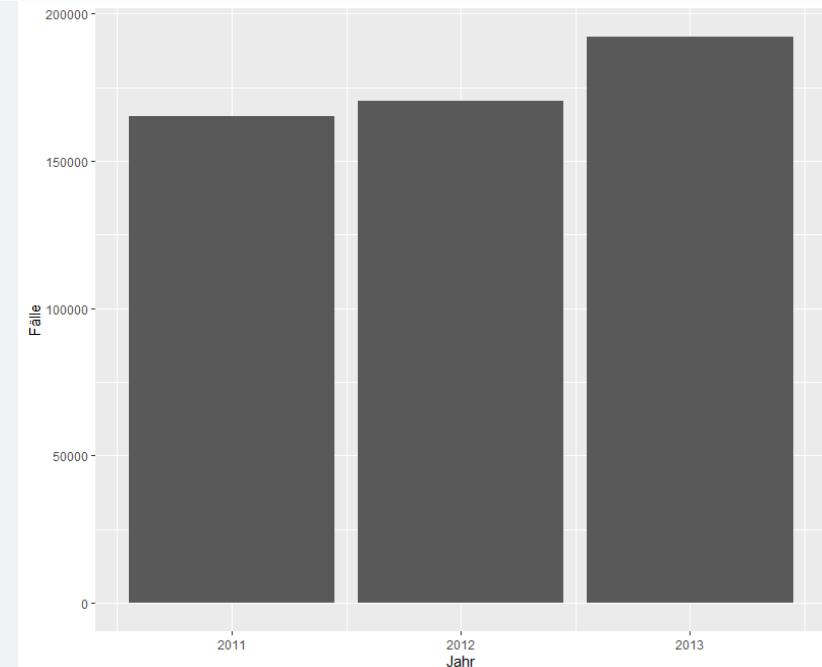
## Y-Achse beginnt nicht bei 0

### Identische Daten

x-Achse beginnt etwa bei 155,000



x-Achse beginnt bei 0



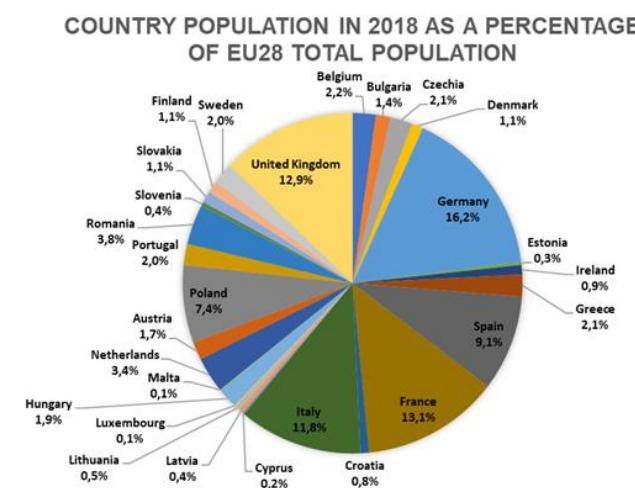
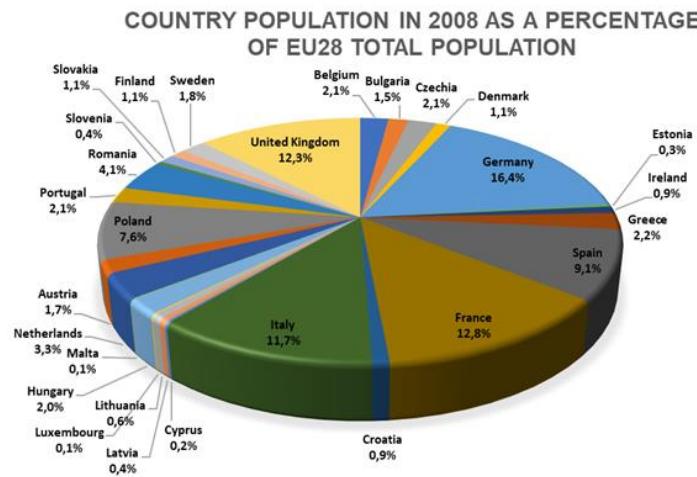
Quelle: Eigenes Beispiel.



## Kreisdiagramme (*pie charts*)

Kreisdiagramme sollten in den meisten Fällen vermieden werden:

- Prozentuale Anteile werden über Winkel dargestellt
- Schwer für Menschen, Winkel zu interpretieren
- Besonders, wenn es viele Werte sind oder das Kreisdiagramm in 3D dargestellt wird



<https://www.martinraffaeiner.blog/using-data-visualizations-bad-guy-pie-charts/>



## Kreisdiagramme (*pie charts*)

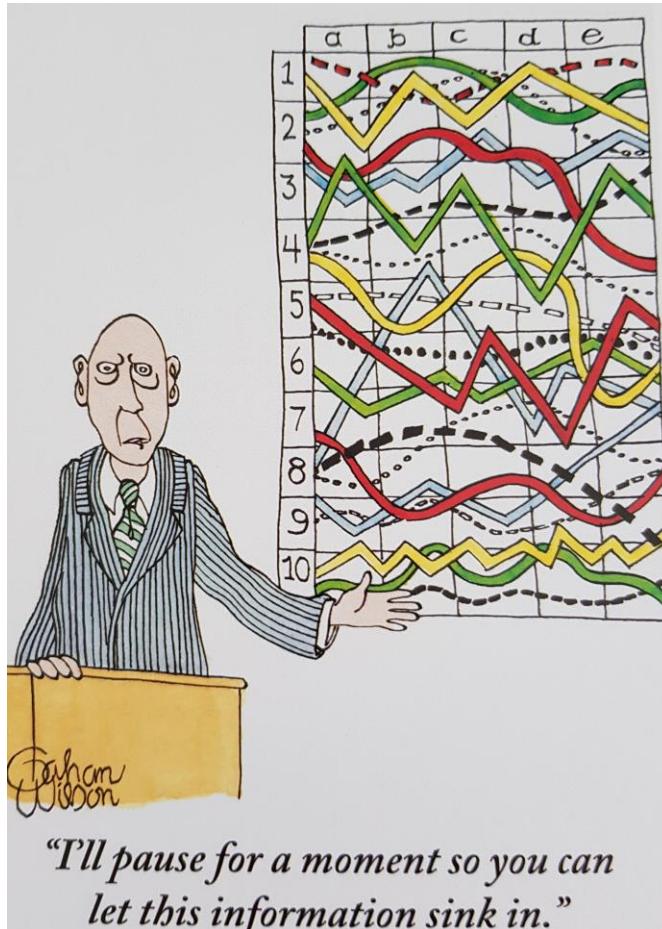
Sind nur geeignet, wenn es wenige und sehr unterschiedliche Werte gibt:



<https://stats.stackexchange.com/questions/423/what-is-your-favorite-data-analysis-cartoon>



## Liniendiagramm mit zu vielen Zeilen



Quelle: Sandra Rendgen. *History of Information Graphics*. Hrsg. Julius Wiedemann. Köln: Taschen, 2019. S. 14.  
Erhältlich in der UB (59 B 343)



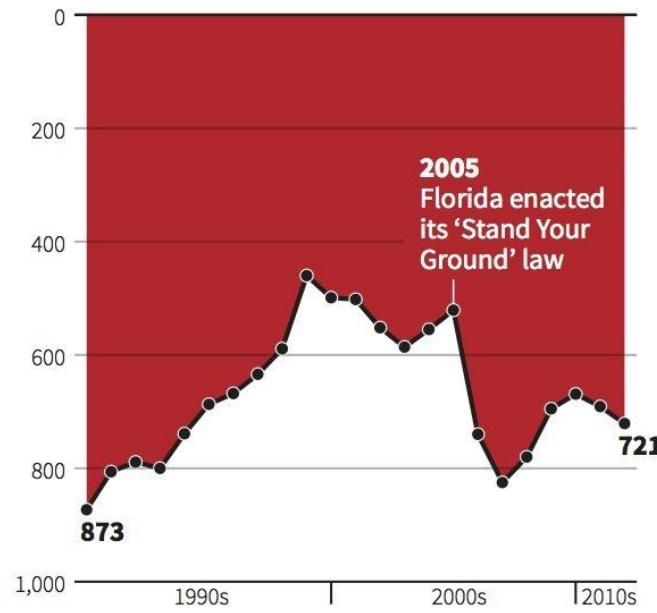
## Konventionen nicht beachten

Es gibt bestimmte etablierte Regeln, die wir beim Interpretieren von Visualisierungen anwenden.

Beispiel: Die y-Achse beginnt normalerweise unten und Werte steigen nach oben hin an. In dieser Grafik wurde die y-Achse umgekehrt. Wenn man dies nicht bemerkt, entsteht der Eindruck, dass die Anzahl an mit Schusswaffen begangenen Morden nach der Einführung des Gesetzes fiel, während sie in Wahrheit anstieg.

### Gun deaths in Florida

Number of murders committed using firearms



Source: Florida Department of Law Enforcement

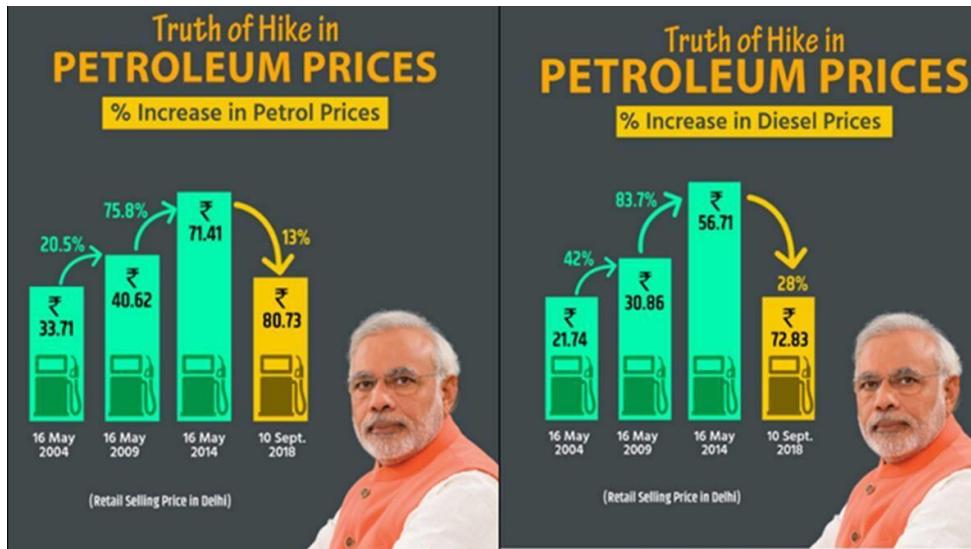
C. Chan 16/02/2014

REUTERS



## Konventionen nicht beachten

Hier ein anderes Beispiel aus Indien: Zwar zeigen die Zahlen in beiden Grafiken an, dass auch 2018 die Benzin- und Dieselpreise stiegen. Allerdings wird durch die nicht skalenegetreue Höhe des jeweils letzten Balkens suggeriert, dass sie fielen.



Quelle: <https://viz.wtf/>



## Fläche vs. Durchmesser

Wenn unterschiedliche große Kreise in Diagrammen verwendet werden, muss die Fläche und nicht der Durchmesser den repräsentierten Wert darstellen, um die Größenverhältnisse nicht zu verzerrten:

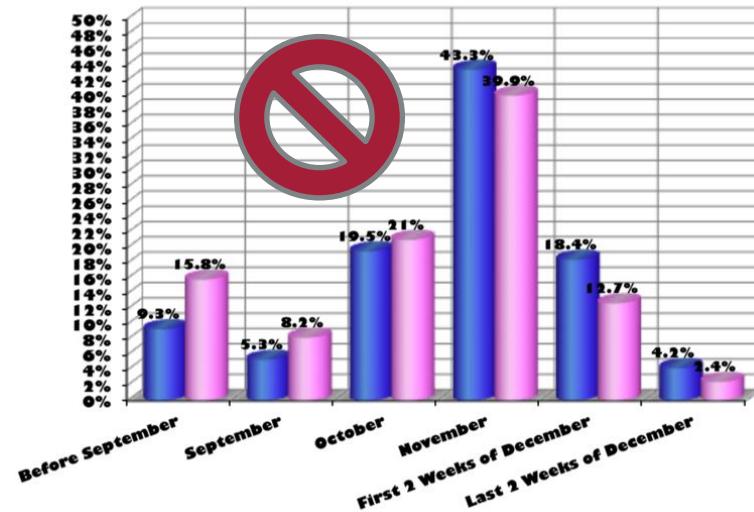


## Zu viele unnötige Elemente

Um Ihre Datenvisualisierung so lesbar und verständlich wie möglich zu machen, sollten Sie alle unnötigen Elemente entfernen:

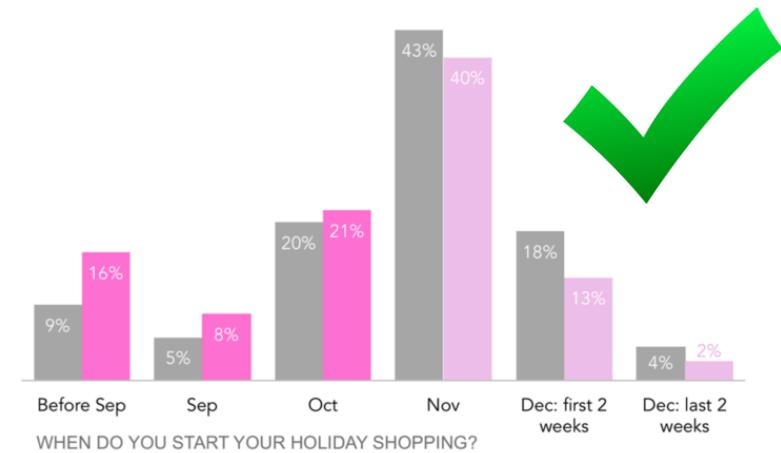
### **Shoppers Begins Shopping for Holidays**

■ Men ■ Women



More women start their holiday shopping early

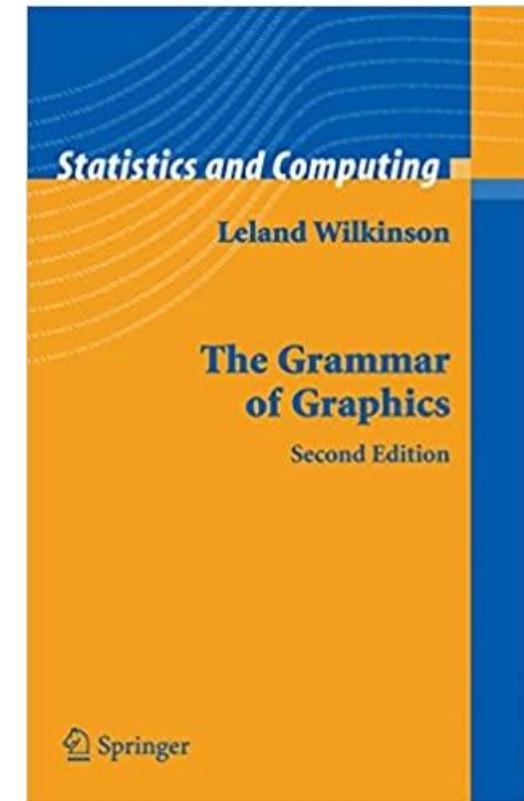
■ Men ■ Women  
% OF TOTAL





## ***Grammar of Graphics (Leland Wilkinson)***

Wilkinson entwickelte eine Grammatik visueller Darstellungen, um Diagramme in ihren Einzelteilen analysieren zu können.



## ***Layered Grammar of Graphics (Hadley Wickham)***

Basierend auf der Grammar of Graphics schuf Hadley Wickham die *Layered Grammar of Graphics*. Diese erlaubt es, Diagramme aus einzelnen Schichten wie etwa Koordinatensysteme, Skalen und Gestaltungselementen zusammenzusetzen.

Die *Layered Grammar of Graphics* wurde dann in dem Grafikpaket *ggplot2* für die Programmiersprache *R* implementiert, welches sich großer Beliebtheit erfreut.

Im folgenden werden wir die Pythonpakete *plotnine* und *folium* für ein paar Visualisierungen verwenden. *plotnine* ist eine Übertragung von *ggplot2* nach Python und lehnt sich stark an dessen Syntax und Funktionen an. *folium* erlaubt es uns, kartenbasierte Visualisierungen zu erzeugen.

---

## Zum Nachlesen

*ColorBrewer.* <https://colorbrewer2.org/>.

*from Data to Viz.* <https://www.data-to-viz.com>.

Marey, Étienne-Jules. *La méthode graphique dans les sciences expérimentales et principalement en physiologie et en médecine*. Paris: 1878. S. I.  
<https://gallica.bnf.fr/ark:/12148/bpt6k6211376f/>

Rendgen, Sandra. *History of Information Graphics*. Hrsg. Julius Wiedemann. Köln: Taschen, 2019. Erhältlich in der UB (59 B 343)

Tufte, Edward. *The Visual Display of Quantitative Information*. Cheshire: Graphics Press, 2001. Mehrere Kopien in der UB und anderen Bibliotheken.

Wilke, Claus O. *Datenvisualisierung: Grundlagen und Praxis*. Übersetzt von Bilgehan Gür. Heidelberg: dpunkt.verlag, 2020. Als eBook verfügbar.

Wilkinson, Leland. *The Grammar of Graphics*. 2. Aufl. New York: Springer, 2005. Als eBook verfügbar.

# Datenvisualisierung in Python

## Datenvisualisierung mit Python

Es gibt mehrere Pakete, die für die Datenvisualisierung mit Python verwendet werden können. Das bekannteste ist *Matplotlib*. In unserem Fall verwenden wir aber *plotnine*, welches wir erst installieren und dann importieren müssen. Wenn Sie schon einmal mit der Programmiersprache *R* gearbeitet und darin mit dem Paket *ggplot2* Visualisierungen erstellt haben, wird Ihnen hier vieles bekannt vorkommen.

*ggplot2* basiert auf einem von Leland Wilkinson begründeten Konzept names *Grammar of Graphics*, nachdem auch komplexe Visualisierungen aus einzelnen Bausteinen zusammengesetzt werden. Dies wurde von Hadley Wickham als *layered grammar of graphics* weiterentwickelt und diente ihm als Basis für die Entwicklung von *ggplot2*. *plotnine* versucht, das Konzept in Python zu übertragen, während die Generierung der Grafik im Hintergrund durch *Matplotlib* erfolgt.

Der für *plotnine* geschriebene Code ähnelt dem für *ggplot2* stark; wenn es Unterschiede gibt, sind sie zumeist durch Syntaxunterschiede zwischen *R* und *Python* bedingt, wie etwa Unterschiede im Gebrauch von Klammern und Anführungszeichen.

## Pakete importieren

```
# Pandas importieren
import pandas as pd

# plotnine importieren
import plotnine
```

## Datensatz laden

Wir laden zuerst einen Datensatz. Dieser enthält weltweite Fall- und Todeszahlen während des ersten Jahres der COVID-19-Pandemie.

```
# CSV Datei einlesen
df = pd.read_csv("covid19_v2.csv")
```

## Visualisierungen erstellen

### Länder mit den höchsten maximalen Tagesfallzahlen

#### A. Datenaufbereitung

Zuerst müssen wir die Daten filtern, da wir nur die 10 Länder mit den höchsten Fallzahlen berücksichtigen wollen. Dazu werden die Tageszahlen nach Ländern gruppiert, wodurch wir die Zahlen für jedes Land separat betrachten können. Dann sucht man für jedes Land den höchsten Wert in der Spalte "cases" und lädt die 10 höchsten dieser Maximalwerte in den DataFrame `max_tag`.

```
# Gruppieren nach Land, jeweils maximale Tagesfallzahl heraussuchen,  
# 10 Länder mit den höchsten Werten auswählen  
max_tag = df.groupby("country") ["cases"].max().nlargest(10)  
# Durch die Gruppierung dient nun die Spalte "country" als Zeilenindex.  
# Das wollen wir zurücksetzen auf numerischen Index:  
max_tag = max_tag.reset_index()  
print(max_tag)
```

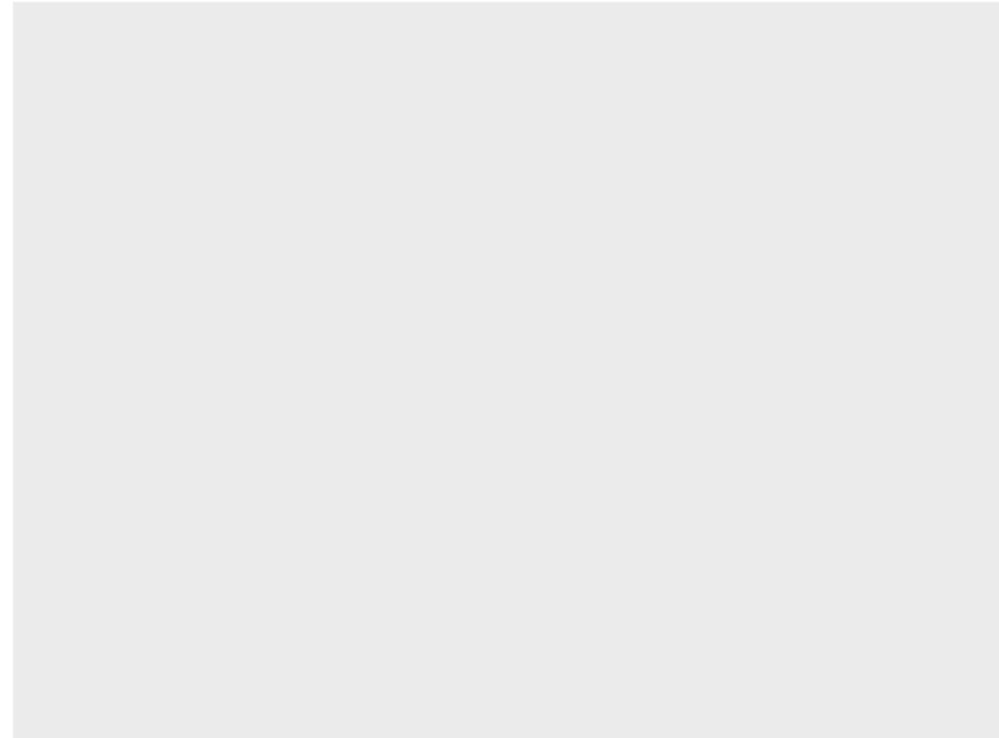
	country	cases
0	United_States_of_America	234633
1	India	97894
2	France	86852
3	Brazil	69074
4	Spain	55019
5	Italy	40902
6	Chile	36179
7	United_Kingdom	33470
8	Turkey	33198
9	Poland	32733

## B. Visualisierung

Für die Visualisierung wählen wir ein Balkendiagramm, mit dem man gut eine kleine Anzahl von Zahlenwerten darstellen kann.

1. Im ersten Schritt erzeugen wir einen *Canvas* (Leinwand)--darauf werden dann alle anderen Elemente der Visualisierung gezeichnet--und übergeben den DataFrame `max_tag` an `plotnine`, damit die dort gespeicherten Daten für die Visualisierung verwendet werden können.

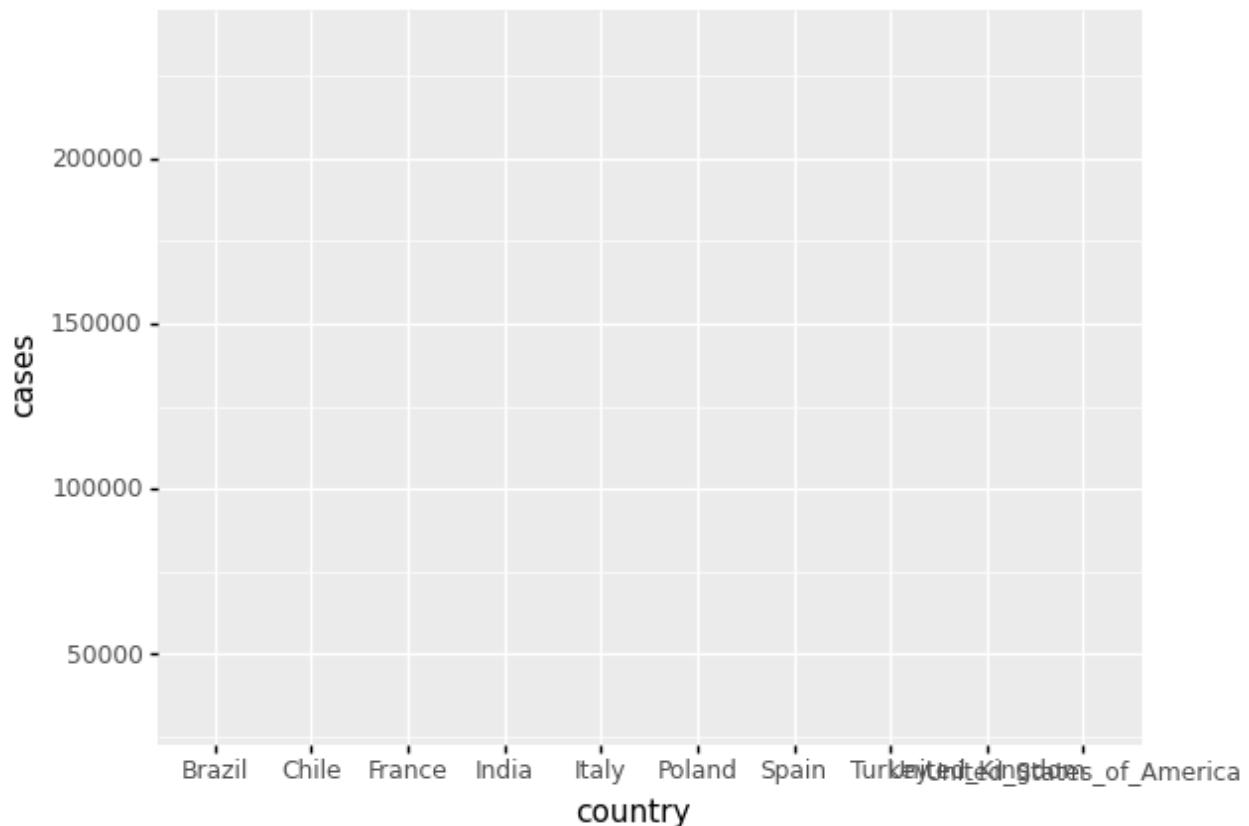
```
plotnine.ggplot(max_tag)
```



```
<ggplot: (115807360308)>
```

1. Jetzt weisen wir den x- und y-Achsen der Grafik die Werte aus dem DataFrame zu: auf der x-Achse werden die Länder dargestellt, auf der y-Achse die Fallzahlen.

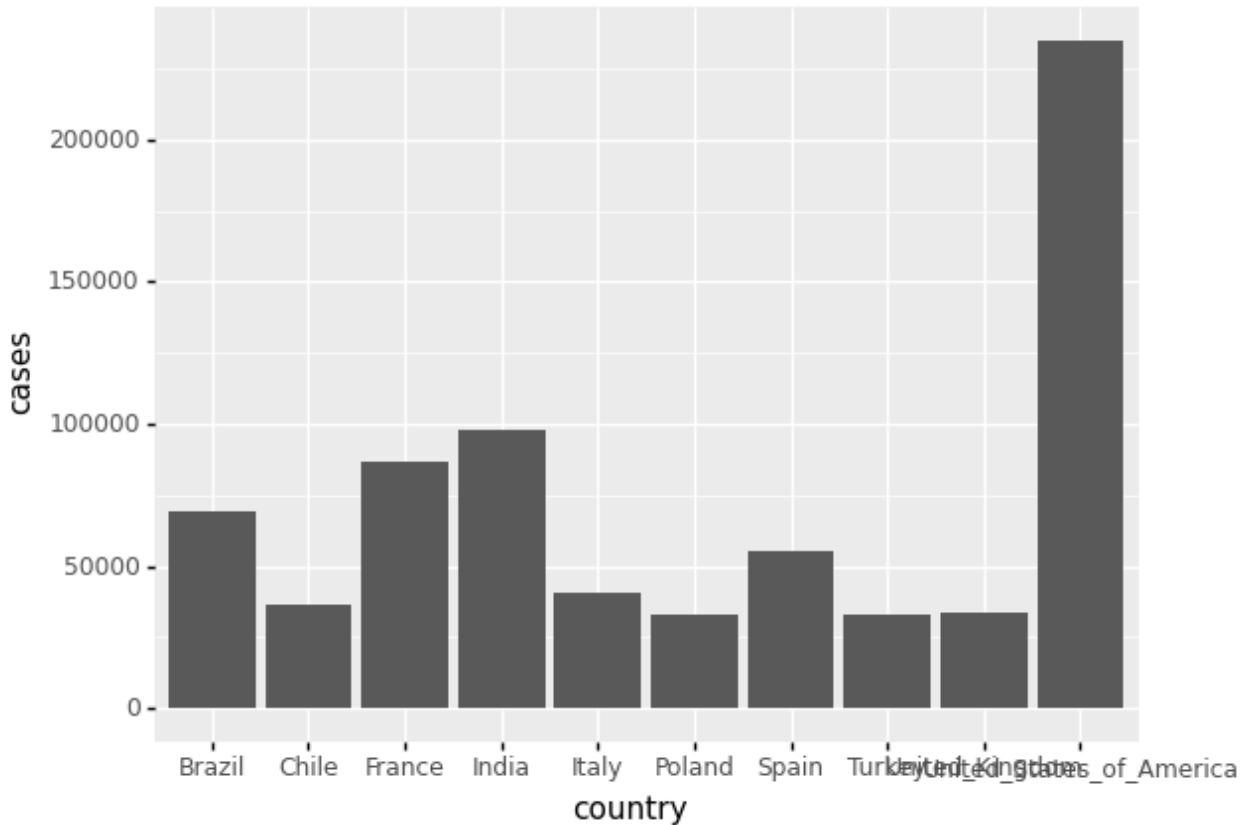
```
plotnine.ggplot(max_tag, plotnine.aes(x="country", y="cases"))
```



```
<ggplot: (115806926038)>
```

1. Hier werden die Balken gezeichnet. Dabei ist es wichtig, den Wert von `stat` auf "identity" zu setzen. In der Standardeinstellung zählt `plotnine`, wie viele Zeilen für jedes Land vorkommen, was wir hier nicht möchten. Mit "identity" können wir sicherstellen, dass die eigentlichen Zahlen aus dem Datensatz genutzt werden. \\ Anmerkung: Da die Erstellung des Diagramms ab hier einen mehrzeiligen Befehl benötigt, muss man entweder am Ende jeder Zeile ein Backslash einfügen oder aber eine Klammer um den gesamten Befehl machen:

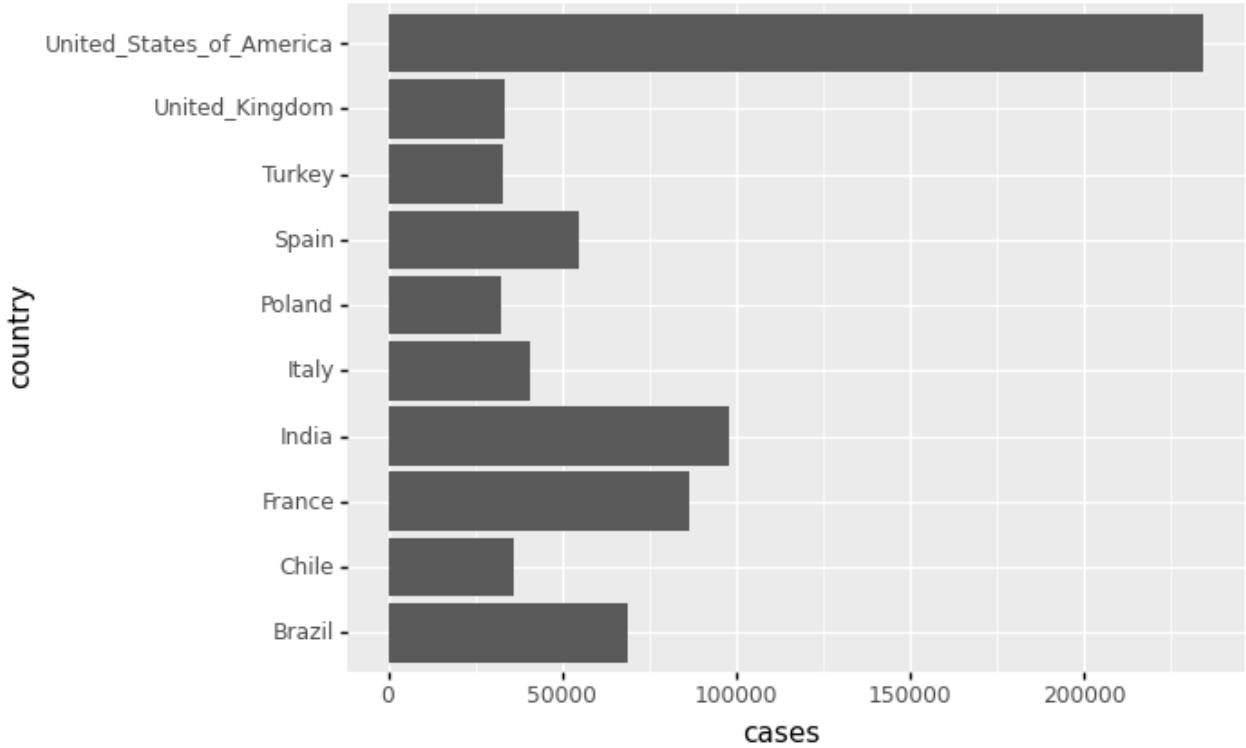
```
(plotnine.ggplot(max_tag, plotnine.aes(x="country", y="cases"))
 + plotnine.geom_bar(stat="identity")
)
```



<ggplot: (115807011380)>

1. Einige der Ländernamen sind zu lang, um sie in der x-Achse lesbar darstellen zu können. Deshalb verwenden wir die Funktion `coord_flip()`, um die x-Werte auf der vertikalen und die y-Werte auf der horizontalen Achse abzubilden.

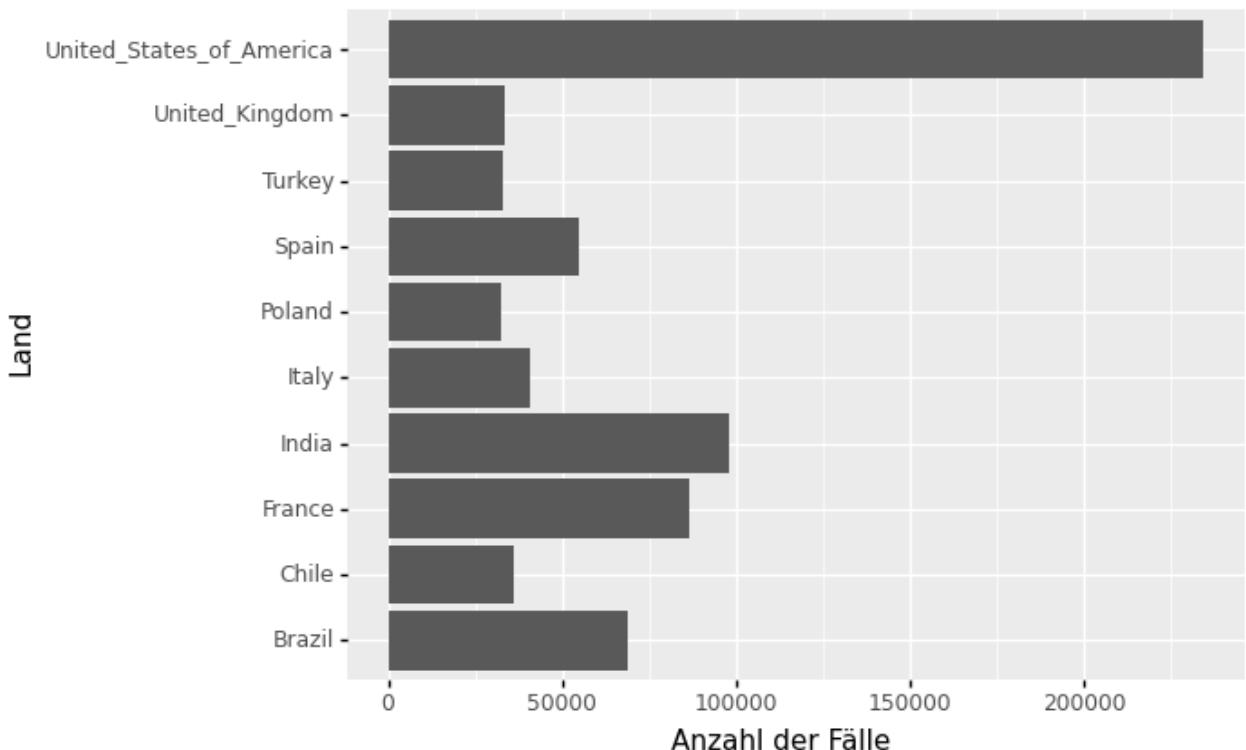
```
(plotnine.ggplot(max_tag, plotnine.aes(x="country", y="cases"))
 + plotnine.geom_bar(stat="identity")
 + plotnine.coord_flip()
)
```



<ggplot: (165893571316)>

- Jetzt ändern wir die Achsenbeschriftungen und ersetzen die aus dem Datensatz entnommenen englischen Begriffe durch ihr deutsches Gegenstück.

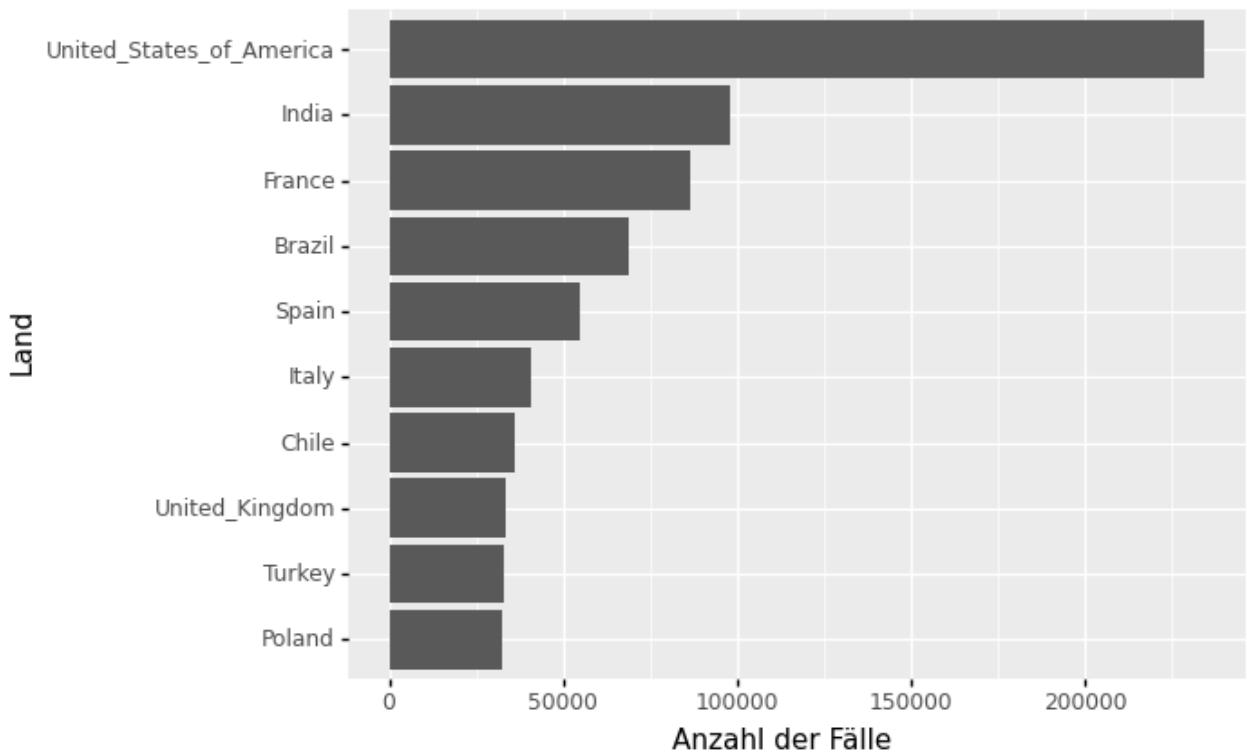
```
(plotnine.ggplot(max_tag, plotnine.aes(x="country", y="cases"))
 + plotnine.geom_bar(stat="identity")
 + plotnine.coord_flip()
 + plotnine.xlab("Land")
 + plotnine.ylab("Anzahl der Fälle")
)
```



```
<ggplot: (165893606310)>
```

1. Im Moment ist es schwer, ähnliche Werte zu vergleichen, wenn sie nicht direkt nebeneinander liegen (wie etwa die Werte für Chile, Großbritannien, die Türkei und Polen). Wir sortieren die Balken deshalb nach Fallzahlen, um solche Vergleiche zu erleichtern.

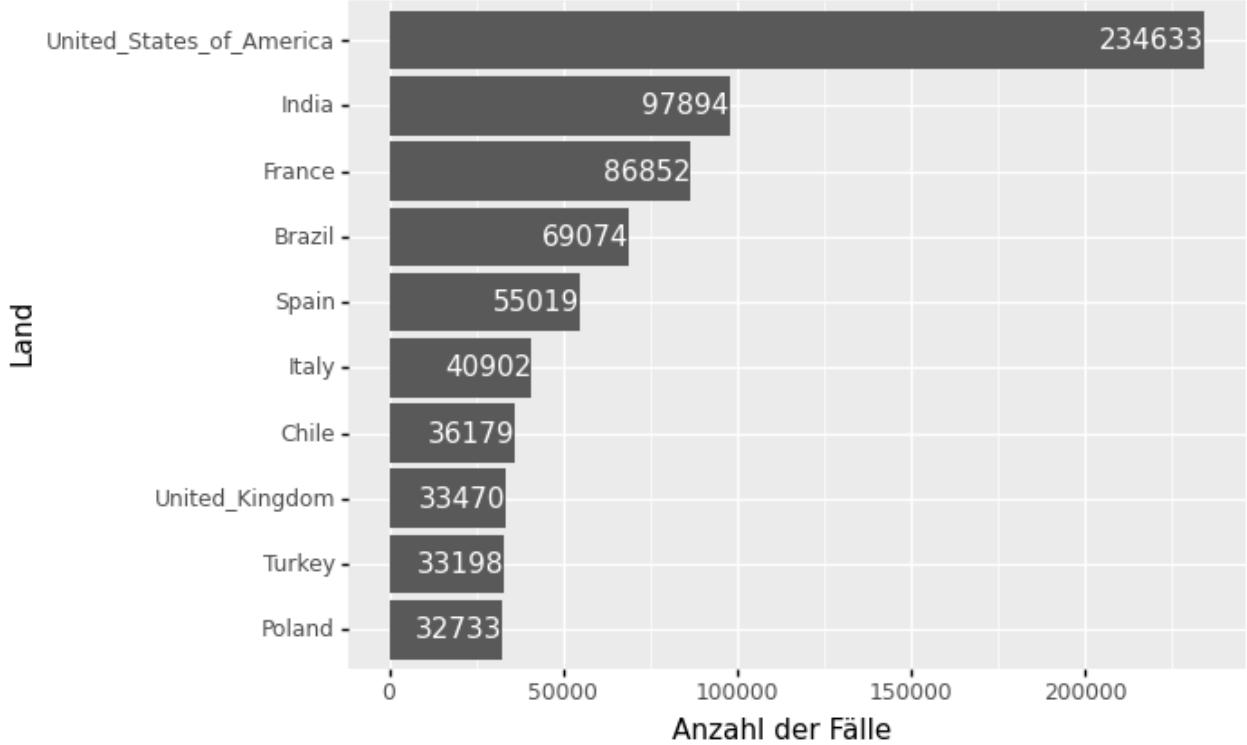
```
(plotnine.ggplot(max_tag, plotnine.aes(x="reorder(country, cases)",  
y="cases"))  
+ plotnine.geom_bar(stat="identity")  
+ plotnine.coord_flip()  
+ plotnine.xlab("Land")  
+ plotnine.ylab("Anzahl der Fälle")  
)
```



```
<ggplot: (165893615770)>
```

1. Nun fügen wir noch Balkenbeschriftungen (*labels*) hinzu, damit wir die genauen Fallzahlen ablesen können.

```
(plotnine.ggplot(max_tag, plotnine.aes(x="reorder(country, cases)", y="cases",  
label="cases"))  
+ plotnine.geom_bar(stat="identity")  
+ plotnine.coord_flip()  
+ plotnine.xlab("Land")  
+ plotnine.ylab("Anzahl der Fälle")  
+ plotnine.geom_text(ha="right", va="center", color="white")  
)
```



<ggplot: (135628350666)>

## Zusammenfassung

```

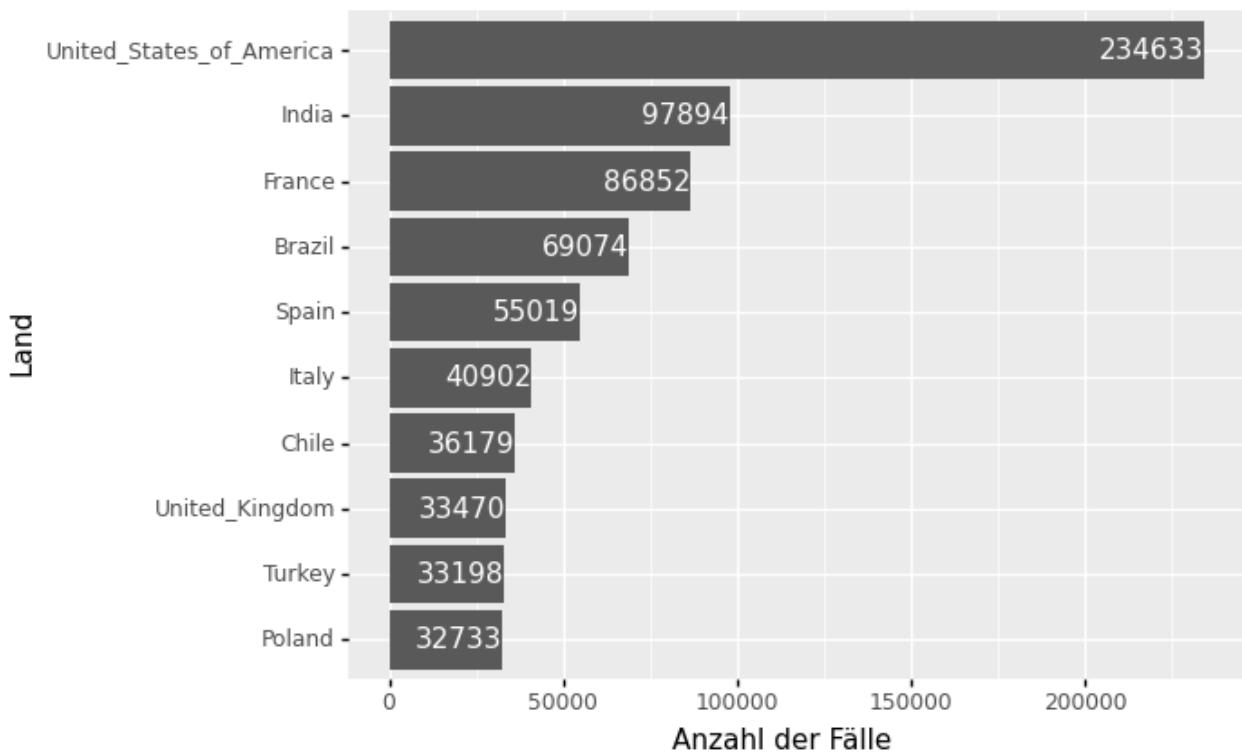
# Wir bilden einen neuen DataFrame aus den geladenen Daten.
# Dieser enthält die Spalten "country" und "cases"
# sowie den höchsten Tageswert für jedes Land.
# Eine Zeile pro Land und wir wählen die 10 Zeilen mit den höchsten Werten
# aus.
max_tag = df.groupby("country")["cases"].max().nlargest(10)
# Durch die Gruppierung dient nun die Spalte "country" als Zeilenindex - das
wollen wir zurücksetzen auf numerischen Index:
max_tag = max_tag.reset_index()

# Hier beginnen wir, die Visualisierung zu definieren.
# Wir legen fest, welcher DataFrame unsere Daten enthält (max_tag) ...
# ... und welche Spalten auf der x- und y-Achse dargestellt werden.
# Auch sortieren wir die Länder nach der Anzahl der Fälle.
(plotnine.ggplot(max_tag, plotnine.aes(x="reorder(country, cases)", y="cases",
label="cases"))

# Hier legen wir fest, dass die Höhe jedes einzelnen Balken durch den Wert
# aus der Spalte "deaths" bestimmt wird und nicht durch die Anzahl der
Werte,
# wie es in der Standardeinstellung von plotnine der Fall ist
+ plotnine.geom_bar(stat="identity")
# Wir vertauschen x- und y-Werte und drehen damit die Grafik um 90 Grad,
# um mehr Platz für die Ländernamen zu haben
+ plotnine.coord_flip()
# Wir ändern die Beschriftung der x-Achse ...
+ plotnine.xlab("Land")
# ... und der y-Achse
+ plotnine.ylab("Anzahl der Fälle")
# Wir integrieren und formatieren die Zahlen am rechten Balkenende und
wählen eine Farbe.

```

```
+ plotnine.geom_text(ha="right", va="center", color="white")
)
```



<ggplot: (115807097929)>

## Fälle pro Tag in ausgewählten Ländern

In der nächsten Visualisierung wollen wir sehen, wie sich die täglichen Fallzahlen in vier ausgewählten Ländern (Deutschland, USA, Großbritannien, Frankreich) über einen längeren Zeitraum entwickelt haben. Wir verwenden hierfür ein Liniendiagramm, das sich für Darstellungen mit einer zeitlichen Komponente gut eignet.

Wie schon im ersten Beispiel müssen wir auch hier zuerst den Datensatz aufbereiten. Wir benötigen nur diejenigen Zeilen, welche Daten für die genannten Länder enthält. Auch müssen wir sicherstellen, dass die Werte in der Spalte "date" als Datumsangaben behandelt werden.

```
# Wir wählen drei Spalten ("date", "deaths" und "country")
# für vier Länder aus (Deutschland, USA, GB, Frankreich)
tf_tag = df[["date", "deaths", "country"]].loc[(df["country"] == "Germany") | \
\                                              (df["country"] == "United_States_of_America") | \
\                                              (df["country"] == "United_Kingdom") | \
\                                              (df["country"] == "France")]

# Wir wandeln die Werte in Spalte "date" in Datumsangaben um
tf_tag["date"] = pd.to_datetime(tf_tag["date"])

print(tf_tag.head(5))
```

	date	deaths	country
20364	2020-12-14	150	France
20365	2020-12-13	194	France

20366	2020-12-12	627	France
20367	2020-12-11	292	France
20368	2020-12-10	296	France

Um unsere Grafik erstellen zu können, müssen wir den DataFrame mit `pd.melt` umstrukturieren und zwar von einem "wide"-Format in ein "tall"-Format umwandeln. Dabei werden Werte, die sich vorher in einer Zeile, aber verschiedenen Spalten befanden, in einer Spalte versammelt, aber auf mehrere Zeilen verteilt:

The diagram illustrates the 'melt' operation. On the left, a wide DataFrame has four columns: Name1, Name2, Name3, and Name4. The first row contains 'Wert1.1', 'Wert2.1', 'Wert3.1', and 'Wert4.1'. The second row contains 'Wert1.2', 'Wert2.2', 'Wert3.2', and 'Wert4.2'. A large black arrow points from this wide DataFrame to a tall DataFrame on the right. The tall DataFrame has three columns: Name1, value\_vars, and value\_name. It contains six rows, each corresponding to one of the four original columns. The first two rows (Wert1.1 and Wert2.1) have 'Name2' in the value\_vars column and 'Wert2.1' and 'Wert2.2' respectively in the value\_name column. The next two rows (Wert1.1 and Wert2.1) have 'Name3' in the value\_vars column and 'Wert3.1' and 'Wert3.2' respectively in the value\_name column. The last two rows (Wert1.1 and Wert2.1) have 'Name4' in the value\_vars column and 'Wert4.1' and 'Wert4.2' respectively in the value\_name column.

Name1	Name2	Name3	Name4
Wert1.1	Wert2.1	Wert3.1	Wert4.1
Wert1.2	Wert2.2	Wert3.2	Wert4.2

Name1	value_vars	value_name
Wert1.1	Name2	Wert2.1
Wert2.1	Name2	Wert2.2
Wert1.1	Name3	Wert3.1
Wert2.1	Name3	Wert3.2
Wert1.1	Name4	Wert4.1
Wert2.1	Name4	Wert4.2

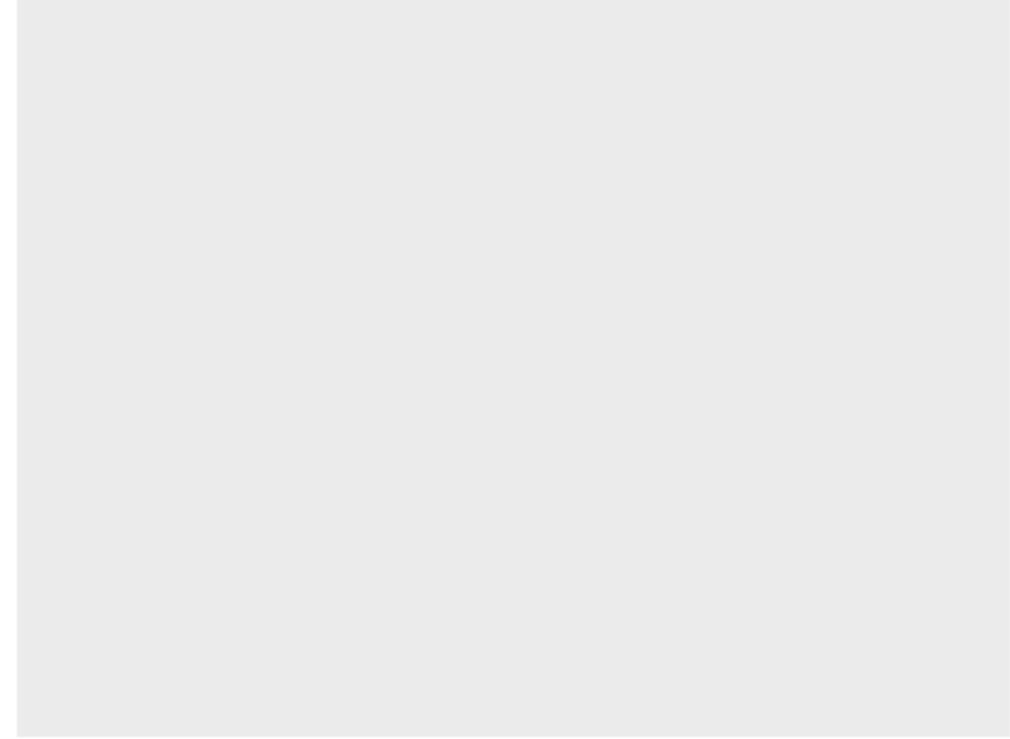
```
tf_tag = tf_tag.melt(id_vars=["date", "country"], value_vars="deaths",
                      value_name="deaths_count")
print(tf_tag.head(5))
```

	date	country	variable	deaths_count
0	2020-12-14	France	deaths	150
1	2020-12-13	France	deaths	194
2	2020-12-12	France	deaths	627
3	2020-12-11	France	deaths	292
4	2020-12-10	France	deaths	296

In diesem Fall wurde eine zusätzliche Spalte mit dem Namen der Variable erzeugt, deren Werte visualisiert werden sollen. Wenn der Datensatz mehr Spalten enthält, kann man diesen Prozess besser beobachten.

1. Auch in diesem Beispiel erzeugen wir zuerst einen leeren *Canvas* und legen den zu verwendenden DataFrame fest.

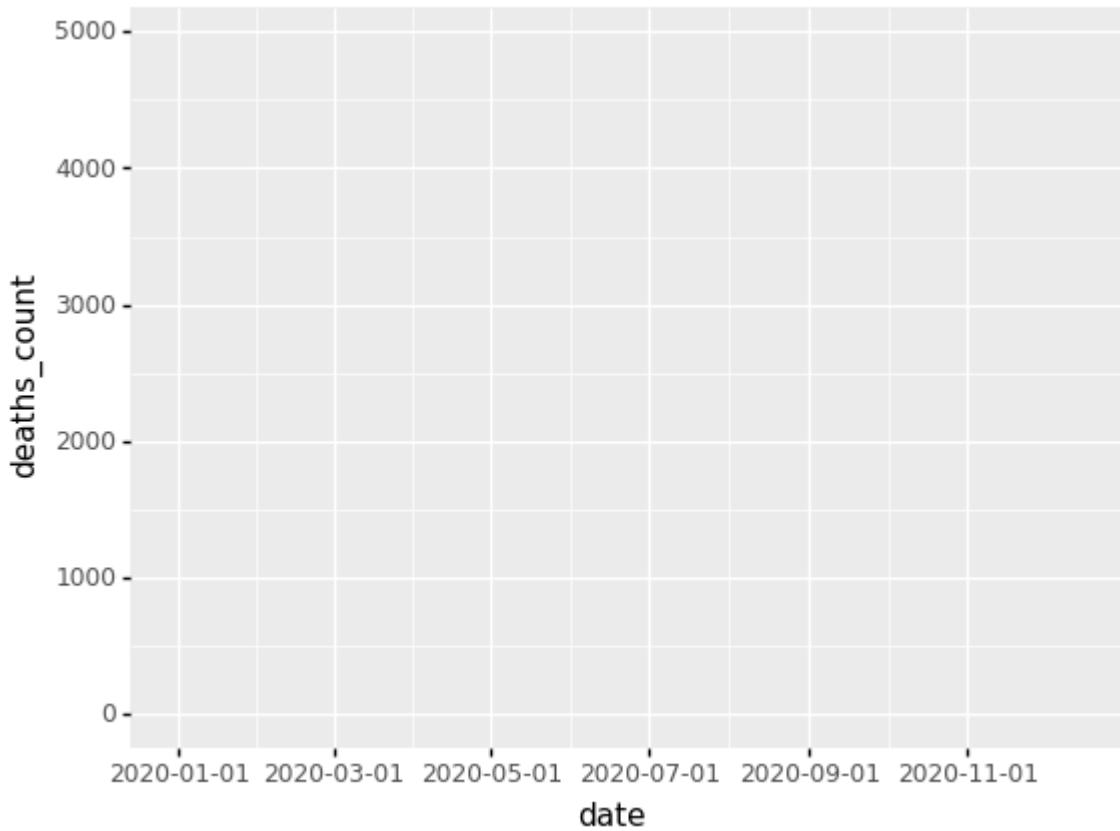
```
plotnine.ggplot(tf_tag)
```



```
<ggplot: (165893769165)>
```

1. Jetzt bestimmen wir, wie *plotnine* die Werte in den verschiedenen Spalten behandeln soll. Die Anzahl der Todesfälle wird auf der y-Achse und die Datumsangaben werden auf der x-Achse dargestellt. Die Fallzahlen werden nach Ländern gruppiert und jedes Land bekommt eine eigene Farbe.

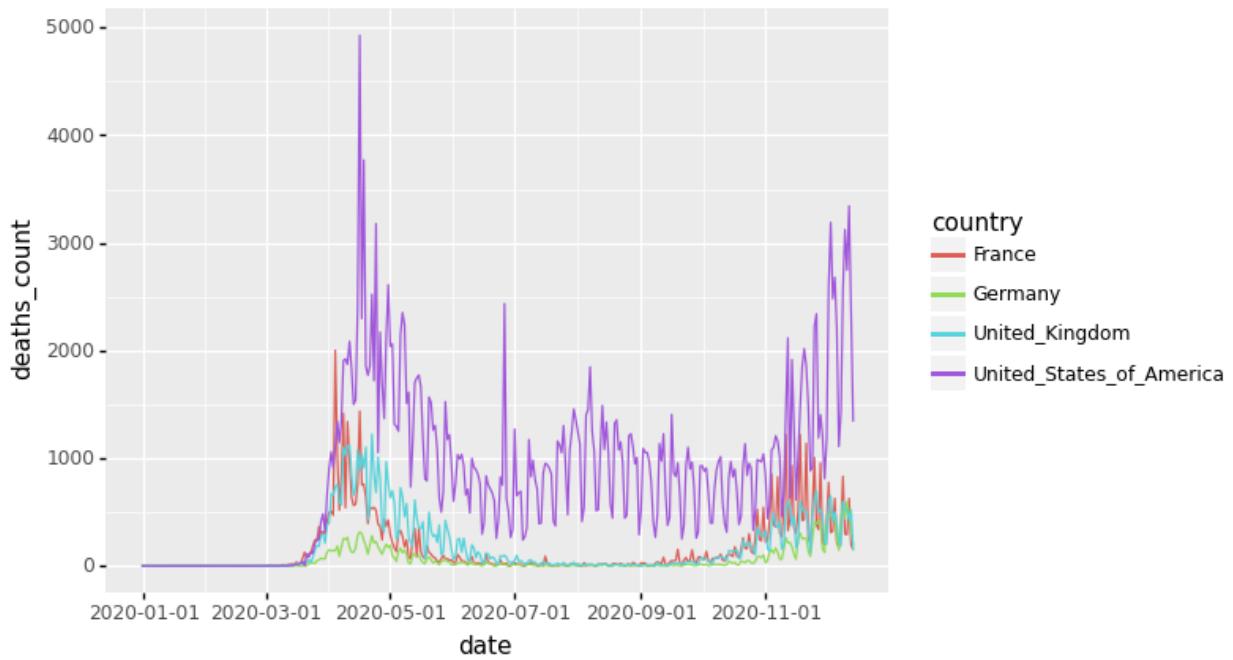
```
plotnine.ggplot(tf_tag, plotnine.aes("date", "deaths_count", group="country",
color="country"))
```



```
<ggplot: (165893778369)>
```

1. Im nächsten Schritt teilen wir *plotnine* mit, dass wir ein Liniendiagramm erstellen möchten.

```
(plotnine.ggplot(tf_tag, plotnine.aes("date", "deaths_count", group="country",
color="country"))
 + plotnine.geom_line())
)
```

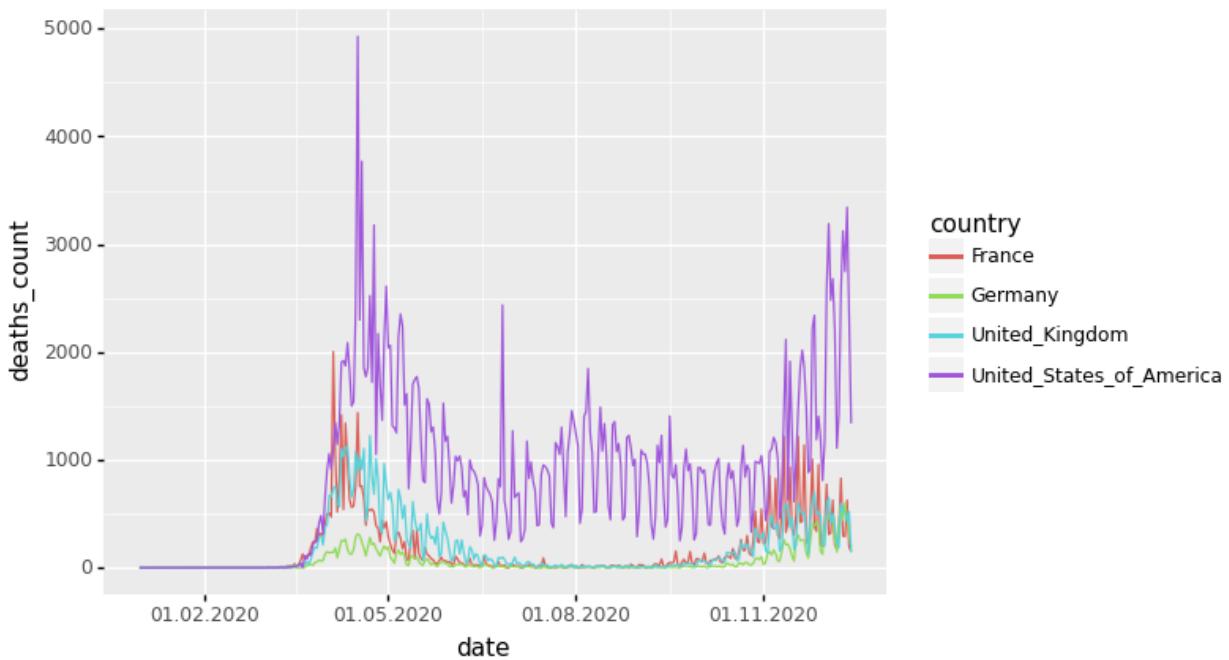


```
<ggplot: (165894306353)>
```

1. Um die Visualisierung leichter lesbar zu machen, vergrößern wir die zeitlichen Abstände

zwischen den Markierungen auf der x-Achse von zwei auf drei Monate und ändern das Datumsformat von JJJJ-MM-TT zu TT.MM.JJJJ (die Codes für die Datumsformatierung kennen Sie bspw. aus der Arbeit mit dem *datetime*-Paket). Sie können die zeitlichen Abstände hier frei wählen (im Beispiel sind es drei Monate), sollten hier aber auch auf die Lesbarkeit achten. Sehr kurze Abstände erleichtern zwar die Zuweisung von Linienpunkten zu einem bestimmten Datum, machen es aber gleichzeitig schwerer, die Labels selbst zu lesen, da sich dann mehr Text in der Zeile befindet.

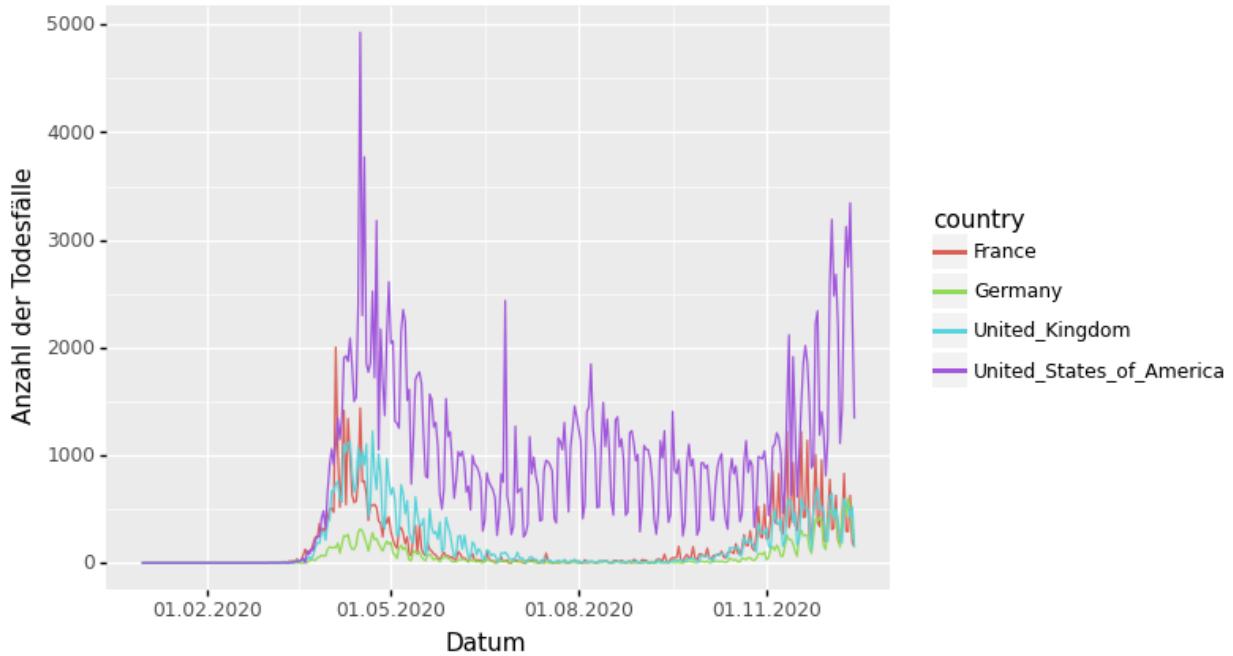
```
(plotnine.ggplot(tf_tag, plotnine.aes("date", "deaths_count", group="country",
color="country"))
+ plotnine.geom_line()
+ plotnine.scales.scale_x_date(breaks="3 months",
date_labels=format ("%d.%m.%Y"))
)
```



<ggplot: (165893634314)>

1. Nun ändern wir die Beschriftungen der x- und y-Achsen:

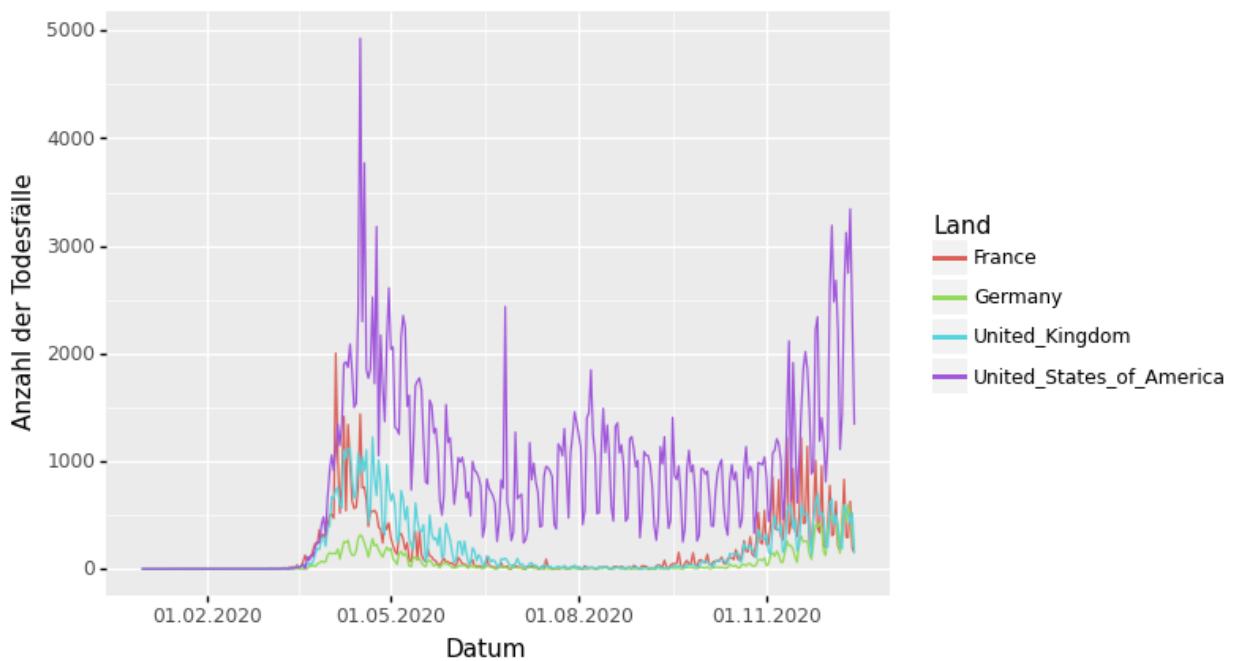
```
(plotnine.ggplot(tf_tag, plotnine.aes("date", "deaths_count", group="country",
color="country"))
+ plotnine.geom_line()
+ plotnine.scales.scale_x_date(breaks="3 months",
date_labels=format ("%d.%m.%Y"))
+ plotnine.xlab("Datum")
+ plotnine.ylab("Anzahl der Todesfälle")
)
```



<ggplot: (165894362362)>

1. Schließlich sollten wir noch die englische Überschrift der Legende auf der rechten Seite anpassen:

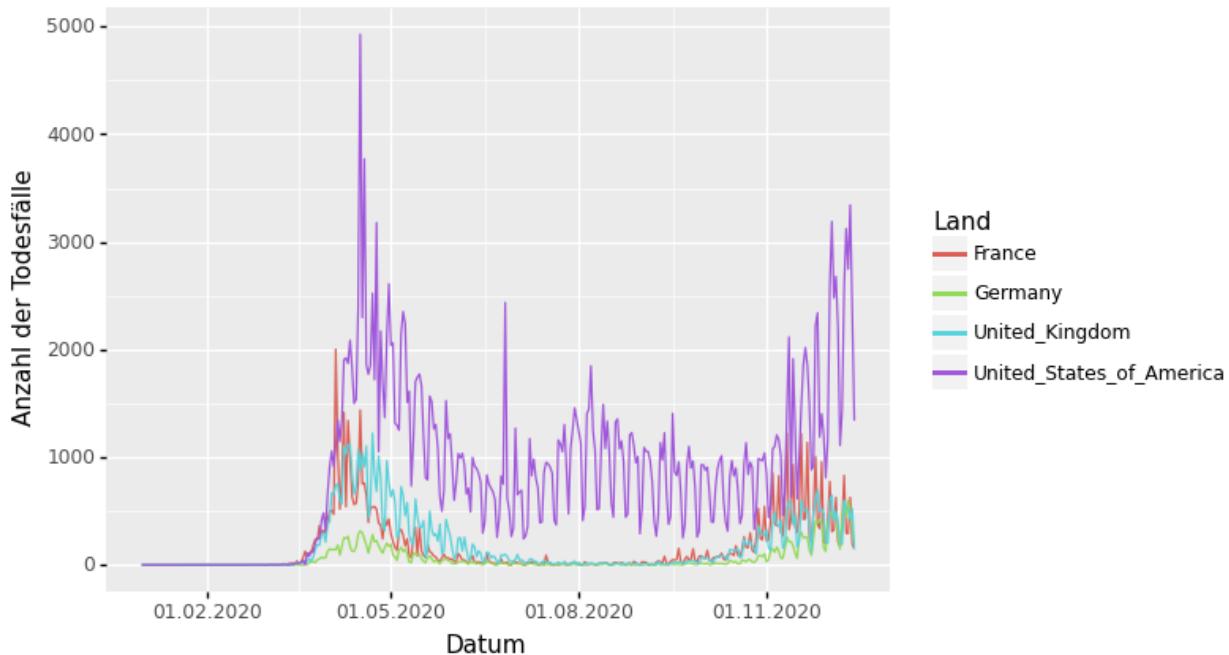
```
(plotnine.ggplot(tf_tag, plotnine.aes("date", "deaths_count", group="country",
color="country"))
 + plotnine.geom_line()
 + plotnine.scales.scale_x_date(breaks="3 months",
date_labels=format ("%d.%m.%Y"))
 + plotnine.xlab("Datum")
 + plotnine.ylab("Anzahl der Todesfälle")
 + plotnine.labs(color="Land")
)
```



<ggplot: (165894487293)>

# Zusammenfassung

```
# Wir wählen drei Spalten ("date", "deaths" und "country")
# für vier Länder aus (Deutschland, USA, GB, Frankreich)
tf_tag = df[["date", "deaths", "country"]].loc[(df["country"] == "Germany") | \
\                                              (df["country"] == "United_States_of_America") | \
\                                              (df["country"] == "United_Kingdom") | \
\                                              (df["country"] == "France")]
# Wir wandeln die Werte in Spalte "date" in Datumsangaben um
tf_tag["date"] = pd.to_datetime(tf_tag["date"])
# Umformung des Datensatzes
tf_tag = tf_tag.melt(id_vars=["date", "country"], value_vars="deaths",
value_name="deaths_count")
# Hier beginnen wir, die Visualisierung zu definieren.
# Wir legen fest, welcher DataFrame unsere Daten enthält (tf_tag) ...
# ... und welche Spalten auf der x- und y-Achse dargestellt werden.
# Wir gruppieren die Datumsangaben und Werte nach Ländern ...
# ... und das Land bestimmt auch die für die Linie verwendete Farbe.
(plotnine.ggplot(tf_tag, plotnine.aes("date", "deaths_count", group="country",
color="country")))
    # Anders als im vorherigen Beispiel verwenden wir keine Balken, sondern
Linien.
    + plotnine.geom_line()
    # Die Markierungen auf der x-Achse sind im Abstand von 3 Monaten plaziert
...
    # ... und werden im Format "tt.mm.jjjj" angegeben.
    # Sie finden eine Liste aller Codes für die Formatierung von Datums- und
Zeitangaben
    # auf: https://docs.python.org/3/library/datetime.html
    + plotnine.scales.scale_x_date(breaks="3 months",
date_labels=format("%d.%m.%Y"))
    # Beschriftung für die x-Achse
    + plotnine.xlab("Datum")
    # Beschriftung für die x-Achse
    + plotnine.ylab("Anzahl der Todesfälle")
    # Beschriftung für die Legende
    + plotnine.labs(color="Land")
)
```



<ggplot: (115807197823)>

## Choroplethenkarte

Das folgende Beispiel basiert auf <https://python-graph-gallery.com/292-choropleth-map-with-folium/>

In unserem letzten Beispiel möchten wir auf einer Weltkarte darstellen, wie viele Todesfälle es pro 100.000 Einwohnern in jedem Land gab. Dazu erstellen wir eine sog. Choroplethenkarte (*choropleth map*), welche besonders gut für die Darstellung "flächenbezogener ordinalskaliger oder gestufter quantitativer Daten" (<https://de.wikipedia.org/wiki/Choroplethenkarte>) geeignet ist.

Wir benötigen dazu das Paket *folium*, das wir zuerst (installieren und) importieren müssen:

```
import folium
```

Dann müssen wir wieder die Daten vorbereiten. Wir gruppieren die Daten nach Ländern und laden die Spalten "country", "code" und "population". In der Spalte "deaths" steht die Summe der in dem jeweiligen Land vorgekommenen Todesfälle. Die Werte in den Spalten "deaths" und "population" werden in Zahlen umgewandelt, damit wir mit ihnen rechnen können. Schließlich benötigen wir noch eine Spalte "deaths100k", in der wir die Todesfälle pro 100.000 Einwohner speichern.

Beachten Sie, dass wir bei der Berechnung

```
daten_karte["deaths"] / (daten_karte["population"] / 100000)
```

die zweite Division in Klammern setzen müssen. Diese muss zuerst ausgeführt werden, bevor wir die Zahl der Todesfälle durch das Ergebnis teilen können.

```
# Quelle für Geodaten: https://geojson-maps.ash.ms/
```

```
# Für die Kartenvizualisierung benötigen wir das Paket "folium", das wir hier importieren.
```

```
import folium
```

```

# Wir bilden einen neuen DataFrame aus den geladenen Daten.
# Dieser enthält die Spalten "country", "code" und "population"
# sowie die Summe der Todesfälle für jedes Land.
daten_karte = df.groupby(["country", "code", "population"])["deaths"].sum()
# Wir erzeugen eine separate Index-Spalte...
daten_karte = daten_karte.reset_index()
# Hier wandeln wir die Werte in den Spalten "deaths" und "population" in
Zahlen um
daten_karte["deaths"] = pd.to_numeric(daten_karte["deaths"])
daten_karte["population"] = pd.to_numeric(daten_karte["population"])
# Wir berechnen die Zahl der Todesfälle pro 100,000 Einwohner.
# Das Ergebnis wird in der Spalte "deaths100k" gespeichert.
daten_karte["deaths100k"] = round(daten_karte["deaths"] /
(deaten_karte["population"] / 100000))

daten_karte.head(5)

```

	<b>country</b>	<b>code</b>	<b>population</b>	<b>deaths</b>	<b>deaths100k</b>
<b>0</b>	Afghanistan	AF	38041757.0	1971	5.0
<b>1</b>	Albania	AL	2862427.0	1003	35.0
<b>2</b>	Algeria	DZ	43053054.0	2596	6.0
<b>3</b>	Andorra	AD	76177.0	79	104.0
<b>4</b>	Angola	AO	31825299.0	371	1.0

1. Zusätzlich zu unserem Datensatz benötigen wir für die Choroplethenkarte noch eine weitere Datei. Diese enthält die geographischen Koordinaten, die *folium* benötigt, um die Umrissse der Länder über die Basiskarte zu zeichnen. Die Daten sind im GeoJSON-Format gespeichert.

Wenn Sie die Karte stark vergrößern, werden Sie feststellen, dass die gezeichneten Umrissse oft nicht genau mit der darunterliegenden Originalkarte übereinstimmen. Der Grund dafür ist, dass *folium* die Umrissse zeichnet, indem es die einzelnen Koordinatenpunkte mit Linien verbindet. Bei unregelmäßigen Grenzen--bspw. Küstenlinien--müssen sehr viele Koordinaten gespeichert werden, um eine nur annähernd originalgetreue Darstellung zu gewährleisten. Dies erhöht nicht nur die Dateigröße, sondern verlangsamt auch die Dateiverarbeitung und Darstellung der Grenzen. Deshalb werden diese Dateien oft in einer niedrigeren Auflösung oder in mehreren unterschiedlichen Auflösungen angeboten. Je höher die Auflösung, desto genauer kann eine geographische Region dargestellt werden.

```
welt_geo = "worldmap.json"
```

1. So wie wir bei *plotnine* zuerst einen *Canvas* angelegt haben, benötigen wir für *folium* eine Grundmappe. Hier können wir mit *location* und einem Paar geographischer Koordinaten den Fokuspunkt sowie mit *zoom\_start* die Vergrößerungsstufe der Karte beim Aufruf festlegen.

```
karte = folium.Map(location=[0, 0], zoom_start=2)
```

1. Nun erzeugen wir eine Instanz der Klasse *Choropleth* und legen im Folgenden die Konfigurationsparameter für unsere Karte fest.

```
folium.Choropleth(  
)
```

1. Wir legen fest, aus welcher Datei *folium* die Geodaten für die gezeichneten Grenzen beziehen soll.

```
folium.Choropleth(  
    geo_data=welt_geo  
)
```

1. Als nächstes bestimmen wir die Datenquelle, d.h., den DataFrame, den wir anfangs erzeugt haben ...

```
folium.Choropleth(  
    geo_data=welt_geo,  
    data=daten_karte  
)
```

1. ... und legen fest, welche Spalten aus dem Datensatz verwendet werden sollen.

```
folium.Choropleth(  
    geo_data=welt_geo,  
    data=daten_karte,  
    columns=["code", "deaths100k"]  
)
```

1. Jetzt müssen wir die Daten aus der Datei "covid19\_v2.csv"--Fallzahlen für jedes Land--mit den Geodaten--Umrisskoordinaten für jedes Land--verbinden. Dazu benötigen wir einen Datenpunkt, der in beiden Datensätzen existiert, damit *folium* weiß, welche Koordinaten zu welchen Fallzahlen passen. In unserem DataFrame befindet sich eine Spalte namens "code", die einen aus zwei Buchstaben bestehenden und im internationalen Standard *ISO 3166-1 alpha-2* definierten Ländercode enthält (eine Liste finden Sie bspw. unter [https://en.wikipedia.org/wiki/ISO\\_3166-1\\_alpha-2](https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2)). Die gleichen Information findet sich in der JSON-Datei im Feld "iso\_a2", welches ein Subelement von "properties" ist.

covid19\_v2.csv

```
1 date,day,month,year,cases,deaths,country,code,population,continent,cum2w100k  
2 2020-12-14,14,12,2020,746,6,Afghanistan,AF,38041757,Asia,9.01377925  
3 2020-12-13,13,12,2020,298,9,Afghanistan,AF,38041757,Asia,7.05277624  
4 2020-12-12,12,12,2020,113,11,Afghanistan,AF,38041757,Asia,6.86876792
```

## worldmap.json

```
1 {"type": "FeatureCollection", "features": [{"type": "Feature", "properties": {"scalerank": 1, "featurecla": "Admin-0 country", "labelrank": 3, "sovereignty": "Afghanistan", "sov_a3": "AFG", "adm0_dif": 0, "level": 2, "type": "Sovereign country", "admin": "Afghanistan", "adm0_a3": "AFG", "geou_dif": 0, "geounit": "Afghanistan", "gu_a3": "AFG", "su_dif": 0, "subunit": "Afghanistan", "su_a3": "AFG", "brk_diff": 0, "name": "Afghanistan", "name_long": "Afghanistan", "brk_a3": "AFG", "brk_name": "Afghanistan", "brk_group": null, "abbrev": "Afg.", "postal": "AF", "formal_en": "Islamic State of Afghanistan", "formal_fr": null, "note_adm0": null, "note_brk": null, "name_sort": "Afghanistan", "name_alt": null, "mapcolor7": 5, "mapcolor8": 6, "mapcolor9": 8, "mapcolor13": 7, "pop_est": 28400000, "gdp_md_est": 22271, "pop_year": -99, "lastcensus": 1979, "gdp_year": -99, "economy": "Least developed region", "income_grp": "5. Low income", "wikipedia": -99, "fips_10": null, "iso_a2": "AF", "iso_a3": "AFG", "iso_n3": "004", "un_a3": "004", "wb_a2": "AF", "wb_a3": "AFG", "oe_id": -99, "adm0_a3_is": "
```

*folium* kann die beiden Datensätze nun über das gemeinsame Feld verbinden. Falls Sie schon einmal in einer Datenbank mehrere Tabellen mit einem SQL (Structured Query Language) *Join* verbunden haben, kennen Sie dieses Prinzip schon.

```
folium.Choropleth(  
    geo_data=welt_geo,  
    data=daten_karte,  
    columns=["code", "deaths100k"],  
    key_on="feature.properties.iso_a2"  
)
```

1. Im nächsten Schritt wird die Farbpalette festgelegt, welche für die Choroplethenkarte verwendet wird. Die Paletten stammen aus dem Projekt *ColorBrewer*, welches auch in anderen Sprachen, wie etwa R, genutzt werden kann. Auf <https://colorbrewer2.org> können Sie verschiedene Paletten ausprobieren.

```
folium.Choropleth(  
    geo_data=welt_geo,  
    data=daten_karte,  
    columns=["code", "deaths100k"],  
    key_on="feature.properties.iso_a2",  
    fill_color="YlOrRd"  
)
```

1. In den nächsten zwei Schritten stellen wir ein, wie transparent die Deck- (*fill\_opacity*) und Linienfarben (*line\_opacity*) sein sollen. 0 bedeutet, dass die gezeichneten Flächen und Linien vollständig durchsichtig sind, während man bei 1 die darunterliegende Karte nicht mehr sehen kann.

```
folium.Choropleth(  
    geo_data=welt.geo..
```

```

        data=daten_karte,
        columns=["code", "deaths100k"],
        key_on="feature.properties.iso_a2",
        fill_color="YlOrRd",
        fill_opacity=1,
        line_opacity=0.2
)

```

1. Jetzt ändern wir die Beschriftung der Legende.

```

folium.Choropleth(
    geo_data=welt_geo,
    data=daten_karte,
    columns=["code", "deaths100k"],
    key_on="feature.properties.iso_a2",
    fill_color="YlOrRd",
    fill_opacity=1,
    line_opacity=0.2,
    legend_name="Todesfälle pro 100.000 Einwohner"
)

```

1. Mit *threshold\_scale* legen wir fest, welche Wertebereiche von den einzelnen Farben abgedeckt werden. Wenn Sie den Parameter weglassen, erfolgt die Abgrenzung automatisch, indem der im Datensatz repräsentierte Wertebereich in gleich große Bereiche aufgeteilt wird.

Alle in der Visualisierung dargestellten Werte müssen zwischen dem niedrigsten und höchsten Wert der *threshold\_scale* liegen. Auch darf die Skala nur Werte in aufsteigender Reihenfolge enthalten und es dürfen nicht mehr Bereiche eingerichtet werden als Farben in der entsprechenden Skala vorhanden sind. Um die Werte der *threshold\_scale* bestimmen zu können, sollten Sie sich die Werteverteilung in Ihrem Datensatz ansehen und mit verschiedenen Kombinationen experimentieren.

```

folium.Choropleth(
    geo_data=welt_geo,
    data=daten_karte,
    columns=["code", "deaths100k"],
    key_on="feature.properties.iso_a2",
    fill_color="YlOrRd",
    fill_opacity=1,
    line_opacity=0.2,
    legend_name="Todesfälle pro 100.000 Einwohner",
    threshold_scale=[0,10,20,40,60,80,100,120,140,160]
)

```

1. Manchmal kann es vorkommen, dass für ein Land keine Werte im Datensatz vorhanden sind. Mit unserer nächsten Codezeile definieren wir eine Farbe, mit welcher ein solches Land markiert werden soll.

```

folium.Choropleth(
    geo_data=welt_geo,

```

```

        data=daten_karte,
        columns=["code", "deaths100k"],
        key_on="feature.properties.iso_a2",
        fill_color="YlOrRd",
        fill_opacity=1,
        line_opacity=0.2,
        legend_name="Todesfälle pro 100.000 Einwohner",
        threshold_scale=[0,10,20,40,60,80,100,120,140,160],
        nan_fill_color="blue"
)

```

1. Schließlich bestimmen wir noch den Deckungsgrad dieser Farbe.

```

folium.Choropleth(
    geo_data=welt_geo,
    data=daten_karte,
    columns=["code", "deaths100k"],
    key_on="feature.properties.iso_a2",
    fill_color="YlOrRd",
    fill_opacity=1,
    line_opacity=0.2,
    legend_name="Todesfälle pro 100.000 Einwohner",
    threshold_scale=[0,10,20,40,60,80,100,120,140,160],
    nan_fill_color="blue",
    nan_fill_opacity=0.5
)

```

1. Jetzt fügen wir das Ganze zu der in Schritt 2 erzeugten Basiskarte hinzu.

```

folium.Choropleth(
    geo_data=welt_geo,
    data=daten_karte,
    columns=["code", "deaths100k"],
    key_on="feature.properties.iso_a2",
    fill_color="YlOrRd",
    fill_opacity=1,
    line_opacity=0.2,
    legend_name="Todesfälle pro 100.000 Einwohner",
    threshold_scale=[0,10,20,40,60,80,100,120,140,160],
    nan_fill_color="blue",
    nan_fill_opacity=0.5
).add_to(karte)

```

1. Wenn wir mit Jupyter Notebook arbeiten, können wir uns die Karte einfach ausgeben lassen:

```

folium.Choropleth(
    geo_data=welt_geo,
    data=daten_karte,
    columns=["code", "deaths100k"],
    key_on="feature.properties.iso_a2",
    fill_color='YlOrRd',
    fill_opacity=1,
    line_opacity=0.2,
    legend_name='Todesfälle pro 100,000 Einwohner',
)

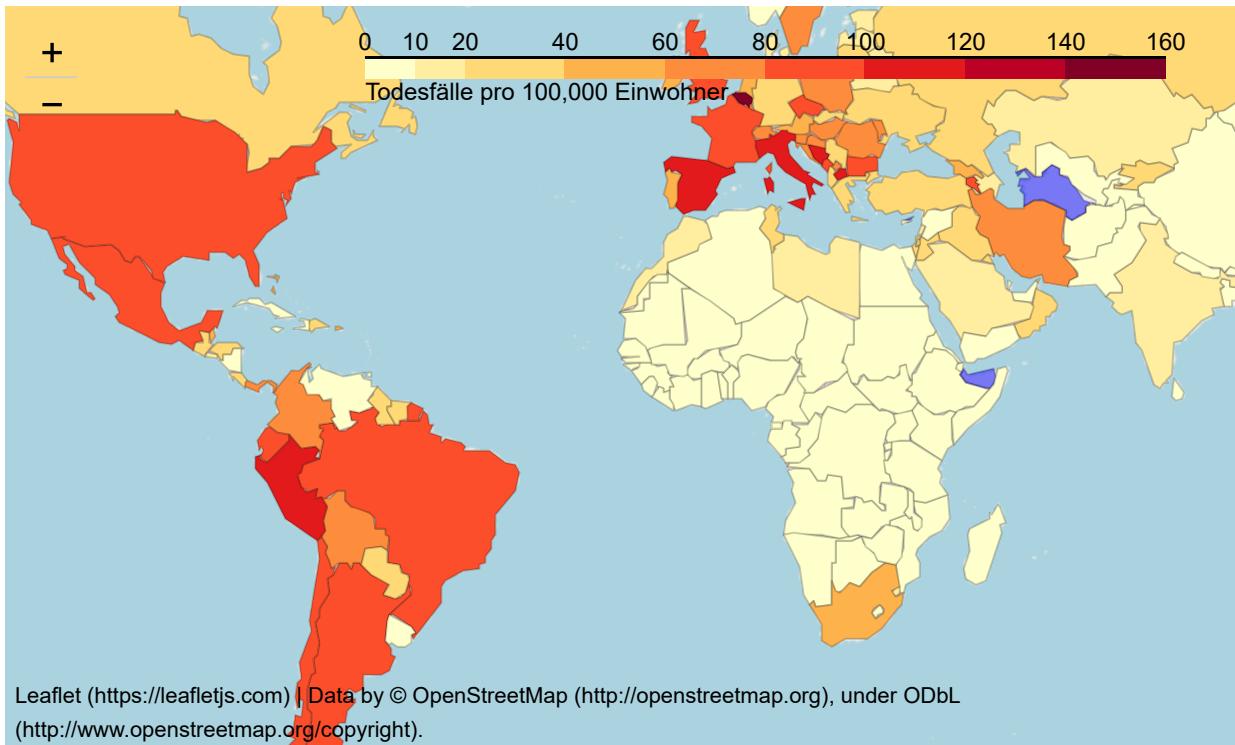
```

```

        threshold_scale=[0,10,20,40,60,80,100,120,140,160],
        nan_fill_color="blue",
        nan_fill_opacity=0.5
    ).add_to(karte)

karte

```



1. In einem anderen Editor müssen wir die Karte abspeichern. Beachten Sie hier, dass dabei keine Bilddatei gespeichert wird, sondern eine HTML-Datei (eine statische Webseite), die Sie dann in einem Browser öffnen können.

```

folium.Choropleth(
    geo_data=welt_geo,
    data=daten_karte,
    columns=["code", "deaths100k"],
    key_on="feature.properties.iso_a2",
    fill_color="YlOrRd",
    fill_opacity=1,
    line_opacity=0.2,
    legend_name="Todesfälle pro 100.000 Einwohner",
    threshold_scale=[0,10,20,40,60,80,100,120,140,160],
    nan_fill_color="blue",
    nan_fill_opacity=0.5
).add_to(karte).save("map.html")

```

## Zusammenfassung

```

# Der Pfad zur JSON-Datei, welche die Koordinaten der Landesgrenzen enthält
welt_geo = "worldmap.json"

# Wir erzeugen eine Grundmappe

```

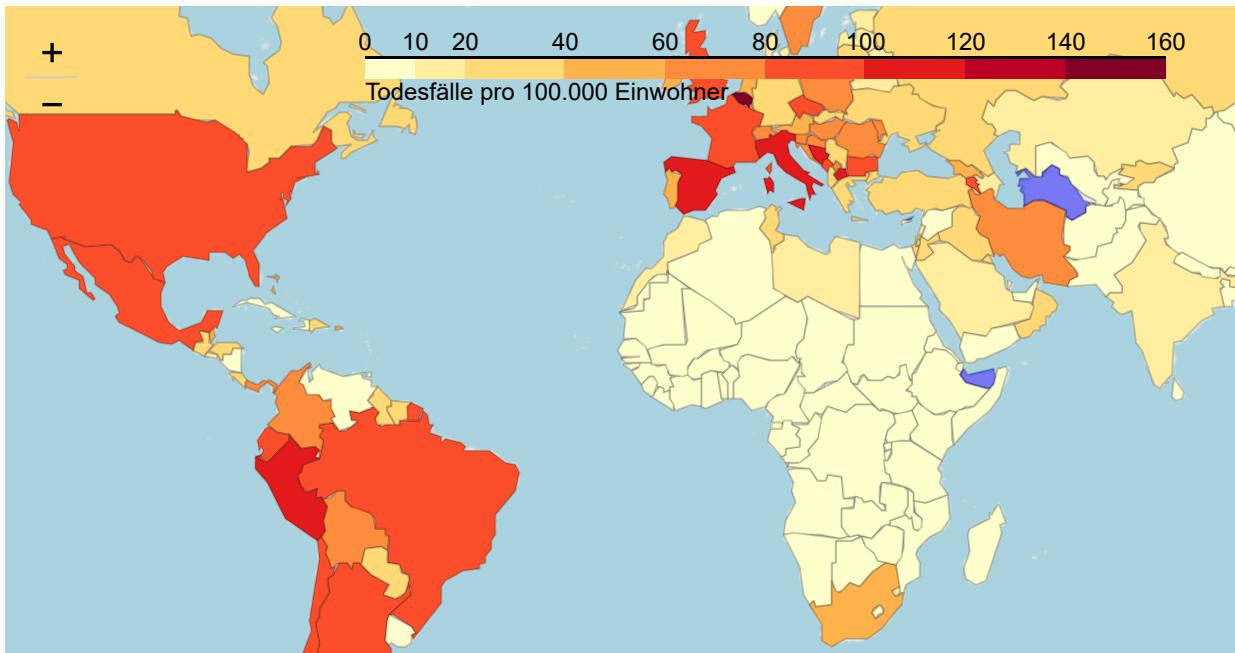
```

# Auch legen wir den Punkt fest, der beim Laden der Karte in der Fenstermitte
liegt...
# ... sowie die Zoomstufe
karte = folium.Map(location=[0, 0], zoom_start=2)

# Jetzt erzeugen wir die eigentliche Karte
folium.Choropleth(
    # Geodaten aus der Datei "worldmap.json"
    geo_data=welt_geo,
    # Sterblichkeitsdaten aus dem DataFrame
    data=daten_karte,
    # Welche Spalten aus dem DataFrame sollen verwendet werden?
    columns=["code", "deaths100k"],
    # Über welches Feld in "worldmap.json" sollen Geodaten und DataFrame
    verbunden werden?
    # Funktioniert analog zu einem "join" in SQL
    key_on="feature.properties.iso_a2",
    # Welche Farbpalette soll für die Repräsentation der Werte verwendet
    # werden?
    fill_color="YlOrRd",
    # Wie stark sollen die Füllfarben decken? Werte gehen von 0 (durchsichtig)
    # bis 1 (kein Hintergrund sichtbar).
    fill_opacity=1,
    # Wie stark sollen die Linienfarben decken? Werte gehen von wieder von 0
    # bis 1.
    line_opacity=0.2,
    # Der Titel der Legende
    legend_name="Todesfälle pro 100.000 Einwohner",
    # Wo liegen die Grenzen der verschiedenen Wertebereiche?
    threshold_scale=[0,10,20,40,60,80,100,120,140,160],
    # Mit welcher Farbe werden Länder markiert, für die es keine Einträge im
    # DataFrame gibt?
    nan_fill_color="blue",
    # Wie stark soll diese Farbe decken?
    nan_fill_opacity=0.5
    # Das Ganze wird jetzt zur Basiskarte hinzugefügt
).add_to(karte)

karte

```



# Ressourcen zu *plotnine* und *folium*

Offizielle *plotnine* Dokumentation:

- <https://plotnine.readthedocs.io/en/stable/>

Kurzes Tutorial von carpentry.org:

- <https://datacarpentry.org/python-ecology-lesson/07-visualization-ggplot-python/index.html>

*ggplot2* Spickzettel (Cheat Sheet):

- <https://rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf>

Offizielle *folium* Dokumentation:

- <https://python-visualization.github.io/folium/>