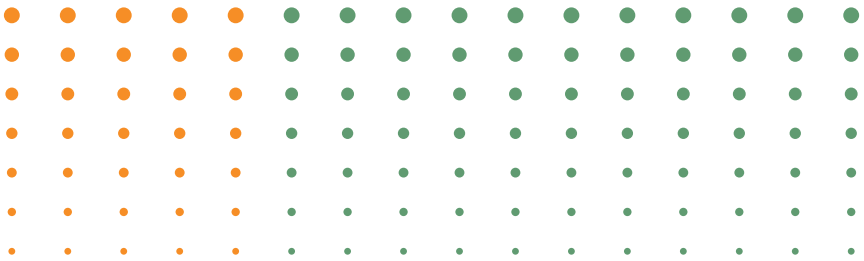
Python-Kurs.eu – Numerisches Python

Mit NumPy, Matplotlib und pandas

Bernd Klein

14. August 2026



Python-Kurs.eu – Numerisches Python

Mit NumPy, Matplotlib und pandas

© 2026 Bernd Klein
Alle Rechte vorbehalten.

Autor: Bernd Klein
Website: https://www.python-kurs.eu
Stand dieser Ausgabe: 14. August 2026

Diese Ausgabe wurde automatisiert aus den autoritativen Jupyter-Notebooks von `python-kurs.eu` erzeugt. Satz und technische Produktion erfolgen reproduzierbar mit Python, Pandoc, LuaLaTeX und - soweit erforderlich - PythonTeX.

Die Online-Fassung kann aktueller sein als diese PDF-Ausgabe. Trotz sorgfältiger Prüfung kann für die Fehlerfreiheit von Texten und Programmen keine Gewähr übernommen werden.

Die im Anhang beworbenen Verlagstitel sind eigenständige Bücher beim Carl Hanser Verlag. Die vorliegende automatisch erzeugte Ausgabe ist **keine Veröffentlichung des Carl Hanser Verlags**.

Das Python-Logo wird zur Kennzeichnung des Bezugs zur Programmiersprache Python verwendet. Python und das Python-Logo sind Marken der Python Software Foundation.

Inhaltsverzeichnis

1 Numerisches Python Ueberblick

1.0.1 Numerisches Python: Überblick

Zusammenspiel von Python, NumPy, SciPy, Matplotlib und pandas

Python ist eine universelle Programmiersprache. Für numerische und wissenschaftliche Aufgaben entsteht ihre besondere Stärke durch ein Ökosystem spezialisierter Bibliotheken:

- **NumPy** stellt das mehrdimensionale `ndarray`, Vektorisierung, Broadcasting und grundlegende numerische Operationen bereit.
- **SciPy** baut auf NumPy auf und ergänzt Algorithmen unter anderem für Optimierung, Integration, Signalverarbeitung, Statistik und lineare Algebra.
- **Matplotlib** dient der Visualisierung numerischer Daten.
- **pandas** ergänzt NumPy um beschriftete Datenstrukturen wie `Series` und `DataFrame` und ist besonders für tabellarische Daten und Zeitreihen geeignet.

Diese Bibliotheken ersetzen Python nicht, sondern ergänzen die Sprache. Python übernimmt Kontrollfluss, Funktionen, Klassen und die allgemeine Programmstruktur; rechenintensive Array-Operationen werden möglichst an NumPy und darauf aufbauende Bibliotheken delegiert.

Empfohlener Weg durch dieses Tutorial

1. NumPy Einführung
2. NumPy-Arrays erzeugen
3. Datentypen mit dtype
4. Numerische Operationen auf Arrays
5. Dimensionen anpassen
6. Matrix-Arithmetik
7. Matplotlib Einführung
8. pandas Einführung

Wer bereits mit Python-Grundlagen vertraut ist, kann direkt mit NumPy beginnen. Für größere Datenanalyse-Aufgaben ist es anschließend sinnvoll, pandas und Matplotlib zu kombinieren.

2 Numpy

2.0.1 NumPy Tutorial

Einführung

NumPy steht für „Numerical Python“ und ist die grundlegende Bibliothek für numerisches Rechnen mit Python. Im Mittelpunkt steht das mehrdimensionale `ndarray`: Seine Elemente besitzen einen definierten Datentyp (`dtype`) und eine Form (`shape`).

NumPy ist besonders stark, wenn Operationen auf ganze Arrays statt in Python-Schleifen elementweise formuliert werden. Solche vektorisierten Operationen und Broadcasting ermöglichen kompakten Code und nutzen optimierte Implementierungen im Hintergrund.

NumPy gehört nicht zur Python-Standardbibliothek. In einer normalen virtuellen Umgebung kann es beispielsweise mit `python -m pip install numpy` installiert werden; Distributionen wie Anaconda bringen NumPy häufig bereits mit.

Die offizielle Dokumentation und Downloads finden Sie unter numpy.org.

Zum Hinton-Diagramm rechts: Die Größe der Quadrate visualisiert den Betrag der Matrixwerte; unterschiedliche Farben kennzeichnen das Vorzeichen.

NumPy entstand historisch aus den älteren Array-Paketen Numeric und Numarray. Diese Vorgänger sind heute nur noch von historischem Interesse.

Vergleich zwischen Kern-Python und NumPy

Wenn wir von Kern-Python sprechen, dann meinen wir das reine Python ohne seine speziellen Module, also in unserem Fall NumPy.

Die Vorteile von Kern-Python:

- Integers und Floats sind als mächtige Klassen implementiert. So können Integer-Zahlen beinahe “unendlich” groß oder klein werden.
- Listen bieten effiziente Methoden zum Einfügen, Anhängen und Löschen von Elementen.
- Dictionaries bieten einen schnellen Lookup.

Vorteile von NumPy-Datenstrukturen gegenüber Python:

- Array-basierte Berechnungen
- Effizient implementierte mehrdimensionale Arrays
- Entworfen für wissenschaftliche Berechnungen

Vier Grundbegriffe: ndarray bezeichnet das Array-Objekt, shape seine Dimensionen, dtype den Elementdatentyp und Broadcasting die Regeln, nach denen NumPy Operationen zwischen kompatibel geformten Arrays ausführt.

Ein einfaches NumPy-Beispiel

Bevor wir NumPy benutzen können, müssen wir es importieren. Es wird importiert wie jedes andere Modul auch:

```
import numpy
```

Die obige import-Anweisung wird man aber nur sehr selten zu sehen bekommen. Üblicherweise wird NumPy in np umbenannt:

```
import numpy as np
```

In unserem ersten einfachen NumPy-Beispiel geht es um Temperaturen. Wir definieren eine Liste mit Temperaturwerten in Celsius:

```
cvalues = [20.1, 20.8, 21.9, 22.5, 22.7,
           21.8, 21.3, 20.9, 20.1]
```

Aus unserer Liste `cvalues` erzeugen wir nun ein eindimensionales NumPy-Array:

```
C = np.array(cvalues)
print(C, type(C))
```

```
[20.1 20.8 21.9 22.5 22.7 21.8 21.3 20.9 20.1] <class 'numpy.ndarray'>
```

Nehmen wir nun an, dass wir die Werte in Grad Fahrenheit benötigen.

Dies kann sehr einfach mit einem NumPy-Array bewerkstelligt werden. Die Lösung unseres Problems besteht in einfachen skalaren Operationen:

```
print(C * 9 / 5 + 32)
```

```
[68.18 69.44 71.42 72.5  72.86 71.24 70.34 69.62 68.18]
```

Das Array C selbst wurde dabei jedoch nicht verändert:

```
print(C)
```

```
[20.1 20.8 21.9 22.5 22.7 21.8 21.3 20.9 20.1]
```

Verglichen zu diesem Vorgehen stellt sich die Python-Lösung, die die Liste mit Hilfe einer Listen-Abstraktion website in eine Liste mit Fahrenheit-Temperaturen wandelt, als umständlich dar!

```
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

```
[68.18, 69.44, 71.42, 72.5, 72.86, 71.24000000000001, 70.34, 69.62, 68.18]
```

Wir haben bisher C als ein Array bezeichnet. Die interne Typbezeichnung lautet jedoch `ndarray` oder noch genauer “C ist eine Instanz der Klasse `numpy.ndarray`”:

```
type(C)
```

```
numpy.ndarray
```

Im Folgenden werden wir die Begriffe “Array” und “ndarray” meistens synonym verwenden.

Grafische Darstellung der Werte

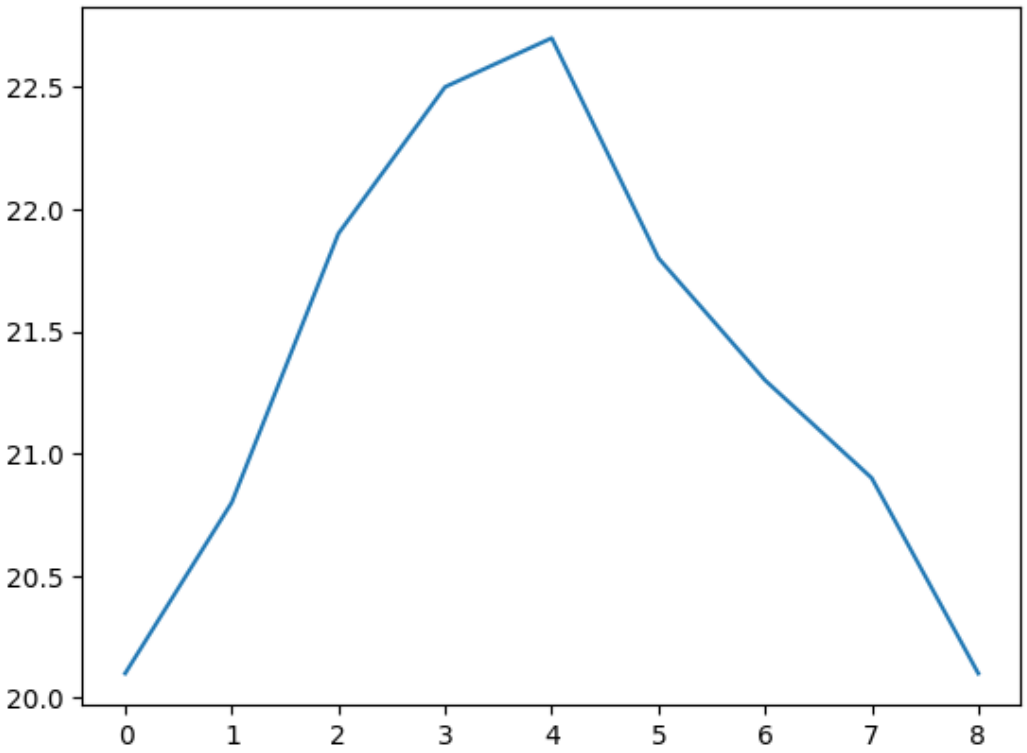
Obwohl wir das Modul Matplotlib erst später im Detail besprechen werden, wollen wir zeigen, wie wir mit diesem Modul die obigen Temperaturwerte ausgeben können. Dazu benutzen wir das Paket `pyplot` aus `matplotlib`. Wenn man mit dem Jupyter-Notebook arbeitet, empfiehlt es sich, die folgende Codezeile zu verwenden, damit der Plot innerhalb des Notebooks erscheint und nicht in einem separat erscheinenden Fenster dargestellt wird:

```
%matplotlib inline
```

Sollen die Plots in einem Notebook jedoch in externen Fenstern auftauchen, schreibt man obige Zeile ohne “inline”, also nur “%matplotlib”.

Der Code zum Erzeugen eines Plots für unsere Werte sieht wie folgt aus:

```
import matplotlib.pyplot as plt

plt.plot(C)
plt.show()
```



Die Funktion `plot` benutzt das Array C als Werte für die Ordinate, also die Y-Achse. Als Werte für die Abszisse wurden die Indizes des Arrays C genommen.

Speicherbedarf

Die wesentlichen Vorteile von NumPy-Arrays sollten ein kleinerer Speicherverbrauch und ein besseres Laufzeitverhalten sein. Wir wollen uns den Speicherverbrauch von NumPy-Arrays in diesem Kapitel unseres Tutorials anschauen und ihn mit dem Speicherverbrauch von Python-Listen vergleichen.

Um den Speicherverbrauch der Liste aus dem vorigen Bild zu berechnen, werden wir die Funktion `getsizeof` aus dem Modul `sys` benutzen:

```
from sys import getsizeof as size

lst = [24, 12, 57]

size_of_list_object = size(lst)   # only green box
size_of_elements = len(lst) * size(lst[0]) # 24, 12, 57

total_list_size = size_of_list_object + size_of_elements
print("Größe ohne Größe der Elemente: ", size_of_list_object)
print("Größe aller Elemente: ", size_of_elements)
print("Gesamtgröße der Liste: ", total_list_size)
```

```
Größe ohne Größe der Elemente:  88
Größe aller Elemente:  84
Gesamtgröße der Liste:  172
```

Der Speicherbedarf einer Python-Liste besteht aus der Größe der allgemeinen Listeninformation, dem Speicherbedarf für die Referenzen auf die Listenelemente und der Größe aller Elemente der Liste. Wenn wir `sys.getsizeof` auf eine Liste anwenden, erhalten wir nur den Speicherbedarf der reinen Liste ohne die Größe der Listenelemente. Im obigen Beispiel sind wir davon ausgegangen, dass alle Integer-Elemente unserer Liste die gleiche Größe haben. Dies stimmt natürlich nicht im allgemeinen Fall, da Integers bei steigender Größe auch einen größeren Speicherbedarf haben.

Wir wollen nun prüfen, wie sich der Speicherverbrauch ändert, wenn wir weitere Integer-Elemente zu der Liste hinzufügen. Außerdem schauen wir uns den Speicherverbrauch einer leeren Liste an:

```
lst = [24, 12, 57, 42]

size_of_list_object = size(lst)
size_of_elements = len(lst) * size(lst[0]) # 24, 12, 57, 42

total_list_size = size_of_list_object + size_of_elements
print("Größe ohne Größe der Elemente: ", size_of_list_object)
print("Größe aller Elemente: ", size_of_elements)
print("Gesamtgröße der Liste: ", total_list_size)

lst = []

print("Speicherbedarf einer leeren Liste: ", size(lst))
```

```
Größe ohne Größe der Elemente:  88
Größe aller Elemente:  112
Gesamtgröße der Liste:  200
Speicherbedarf einer leeren Liste:  56
```

Aus den Ausgaben des vorigen Codes können wir folgern, dass wir für jedes Integer-Element 8 Bytes für die Referenz benötigen. Ein Integer-Objekt selbst benötigt in unserem Fall 28 Bytes. Die Größe der Liste "lst" ohne den Speicherbedarf für die Elemente selbst kann also in unserem Fall wie folgt berechnet werden:

```
64 + 8 * len(lst)
```

Um den kompletten Speicherbedarf einer Integer-Liste auszurechnen, müssen wir noch den Speicherbedarf aller Integer hinzuaddieren.

Nun werden wir den Speicherbedarf eines NumPy-Arrays berechnen. Zu diesem Zweck schauen wir uns zunächst die Implementierung im folgenden Bild an:

Wir erzeugen nun das Array aus dem vorigen Bild und berechnen seinen Speicherbedarf:

```
a = np.array([24, 12, 57])
print(size(a))
```

```
136
```

Den Speicherbedarf für die allgemeine Array-Information können wir berechnen, indem wir ein leeres Array erzeugen:

```
e = np.array([])
print(size(e))
```

```
112
```

Wir können sehen, dass die Differenz zwischen dem leeren Array "e" und dem Array "a", bestehend aus 3 Integern, 24 Bytes beträgt. Dies bedeutet dass sich der Speicherbedarf für ein beliebiges Integer-Array "n" wir folgt ergibt:

```
96 + n * 8 Bytes
```

Im Vergleich dazu berechnet sich der Speicherbedarf einer Integer-Liste, wie wir gesehen haben, als:

```
64 + 8 len(lst) + len(lst) * 28
```

Dies ist eine untere Schranke, da Python-Integers größer als 28 Bytes werden können!

Wenn wir ein NumPy-Array definieren, wählt NumPy automatisch eine feste Integer-Größe, in unserem Fall "int64".

Diese Größe können wir auch bei der Definition eines Arrays festlegen. Damit ändert sich natürlich auch der Gesamtspeicherbedarf des Arrays:

```
a = np.array([24, 12, 57], np.int8)
print(size(a) - 96)

a = np.array([24, 12, 57], np.int16)
print(size(a) - 96)
```

```
a = np.array([24, 12, 57], np.int32)
print(size(a) - 96)

a = np.array([24, 12, 57], np.int64)
print(size(a) - 96)
```

```
19
22
28
40
```

Zeitvergleich zwischen Python-Listen und NumPy-Arrays

Einer der Hauptvorteile von NumPy ist sein Zeitvorteil gegenüber Standardpython. Im Folgenden definieren wir zwei Funktionen. Die erste `pure_python_version` erzeugt zwei Python-Listen mittels `range`, während die zweite zwei NumPy-Arrays mittels der NumPy-Funktion `arange` erzeugt. In beiden Funktionen addieren wir die Elemente komponentenweise:

```
import numpy as np
import time

size_of_vec = 1000

def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = [X[i] + Y[i] for i in range(len(X))]
    return time.time() - t1

def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1
```

Wir rufen diese Funktionen auf und können den Zeitvorteil sehen:

```
t1 = pure_python_version()
t2 = numpy_version()

print(t1, t2)
print("NumPy is in this example " + str(t1/t2) + " faster!")
```

```
9.34600830078125e-05 2.1696090698242188e-05
NumPy is in this example 4.3076923076923075 faster!
```

Die Zeitmessung gestaltet sich einfacher und vor allen Dingen besser, wenn wir dazu das Modul `timeit` verwenden. In dem folgenden Skript werden wir die Timer-Klasse nutzen.

Dem Konstruktor eines Timer-Objektes können zwei Anweisungen übergeben werden: eine, die gemessen werden soll, und eine, die als Setup fungiert. Beide Anweisungen sind auf

'pass' per Default gesetzt. Ansonsten kann noch eine Timer-Funktion übergeben werden.

Ein Timer-Objekt hat eine `timeit`-Methode. Das Argument der `timeit`-Methode ist die Anzahl der Schleifendurchläufe, die der Code wiederholt werden soll.

`timeit(number=1000000)`

`timeit` liefert als Ergebnis die benötigte Zeit für `number`-Durchläufe.

```
import numpy as np
from timeit import Timer

size_of_vec = 1000

def pure_python_version():
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = [X[i] + Y[i] for i in range(len(X))]

def numpy_version():
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y

timer_obj1 = Timer("pure_python_version()",
                   "from __main__ import pure_python_version")
timer_obj2 = Timer("numpy_version()",
                   "from __main__ import numpy_version")

print(timer_obj1.timeit(10))
print(timer_obj2.timeit(10))
```

```
0.0006477549904957414
0.0001723990135360509 2
```

Die `repeat`-Method ist eine vereinfachte Möglichkeit, die Methode `timeit` mehrmals aufzurufen und eine Liste der Ergebnisse zu erhalten:

```
print(timer_obj1.repeat(repeat=3, number=10))
print(timer_obj2.repeat(repeat=3, number=10))
```

```
[0.0006301919929683208, 0.0006212980078998953, 0.0006297839863691479]
[0.0001943639945238828 7, 1.7462007235735655e-05, 2.0657986169680953e-05]
```

3 Numpy Arrays Erzeugen

3.0.1 Arrays in NumPy erzeugen

Wir haben bereits im vorigen Kapitel unseres NumPy-Tutorials gesehen, dass wir NumPy-Arrays aus Listen und Tupel generieren können. Nun wollen wir weitere Funktionen zum Erzeugen von Arrays einführen.

NumPy bietet Funktionen, um Intervalle mit Werten zu erzeugen, deren Abstände gleichmäßig verteilt sind. `arange` benutzt einen gegebenen Abstandwert, um innerhalb von gegebenen Intervallgrenzen entsprechende Werte zu generieren, während `linspace` eine bestimmte Anzahl von Werten innerhalb gegebener Intervallgrenzen berechnet. Den Abstand berechnet `linspace` automatisch.

Erzeugung äquidistanter Intervalle

arange Die Syntax von arange:

```
arange([start,] stop[, step], [, dtype=None])
```

`arange` liefert gleichmäßig verteilte Werte innerhalb eines gegebenen Intervalles zurück. Die Werte werden innerhalb des halb-offenen Intervalles `[start, stop)` generiert. Wird diese Funktion mit Integer-Werten benutzt, ist sie beinahe äquivalent zu der built-in Python-Funktion `range`. `arange` liefert jedoch ein ndarray zurück, während `range` einen Listen-Iterator zurückliefert. Falls der `start`-Parameter nicht übergeben wird, wird `start` auf 0 gesetzt. Das Ende des Intervalls wird durch den Parameter `stop` bestimmt. Üblicherweise wird das Intervall diesen Wert nicht beinhalten, außer in den Fällen, in denen `step` keine Ganzzahl ist und floating-point-Effekte die Länge des Ausgabearrays beeinflussen. Der Abstand zwischen zwei benachbarten Werten des Ausgabearrays kann mittels des optionalen Parameters `step` gesetzt werden. Der Default-Wert für `step` ist 1.

Falls ein Wert für `step` angegeben wird, kann der `start`-Parameter nicht mehr optional sein, d.h. er muss dann auch angegeben werden.

Der Type des Ausgabearrays kann mit dem Parameter `dtype` bestimmt werden. Wird er nicht angegeben, wird der Typ automatisch aus den übergebenen Eingabewerten ermittelt.

```
import numpy as np

a = np.arange(1, 7)
print(a)

# im Vergleich dazu nun range:
x = range(1, 7)
print(x)    # x ist ein Iterator
print(list(x))

# weitere arange-Beispiele:
```

```
x = np.arange(7.3)
print(x)
x = np.arange(0.5, 6.1, 0.8)
print(x)
x = np.arange(0.5, 6.1, 0.8, int)
print(x)
```

```
[1 2 3 4 5 6]
range(1, 7)
[1, 2, 3, 4, 5, 6]
[0. 1. 2. 3. 4. 5. 6. 7.]
[0.5 1.3 2.1 2.9 3.7 4.5 5.3]
[0 1 2 3 4 5 6]
```

linspace Die Syntax von `linspace`:

`linspace(start, stop, num=50, endpoint=True, retstep=False)`

linspace liefert ein ndarray zurück, welches aus 'num' gleichmäßig verteilten Werten aus dem geschlossenen Interval `['start', 'stop']` oder dem halb-offenen Intervall `['start', 'stop')` besteht. Ob ein geschlossenes oder ein halb-offenes Intervall zurückgeliefert wird, hängt vom Wert des Parameters `endpoint` ab. `stop` ist der letzte Wert des Intervalls, falls `endpoint` nicht auf False gesetzt ist. Die Schrittweite ist unterschiedlich, je nachdem, ob `endpoint True` oder `False` ist:

```
import numpy as np

# 50 Werte (Defaut) zwischen 1 und 10:
print(np.linspace(1, 10))
# 7 Werte zwischen 1 und 10:
print(np.linspace(1, 10, 7))
# jetzt ohne Endpunkt:
print(np.linspace(1, 10, 7, endpoint=False))
```

```
[ 1.          1.18367347  1.36734694  1.55102041  1.73469388  1.91836735
  2.10204082  2.28571429  2.46938776  2.65306122  2.83673469  3.02040816
  3.20408163  3.3877551   3.57142857  3.75510204  3.93877551  4.12244898
  4.30612245  4.48979592  4.67346939  4.85714286  5.04081633  5.2244898
  5.40816327  5.59183673  5.7755102   5.95918367  6.14285714  6.32653061
  6.51020408  6.69387755  6.87755102  7.06122449  7.24489796  7.42857143
  7.6122449   7.79591837  7.97959184  8.16326531  8.34693878  8.53061224
  8.71428571  8.89795918  9.08163265  9.26530612  9.44897959  9.63265306
  9.81632653 10.        ]
[ 1.   2.5  4.   5.5  7.   8.5 10. ]
[1.         2.28571429 3.57142857 4.85714286 6.14285714 7.42857143
 8.71428571]
```

Bis jetzt haben wir einen interessanten Parameter nicht besprochen. Falls der Parameter 'retstep' gesetzt ist, wird die Funktion auch den Wert des Abstandes zwischen zwei benachbarten Werten des Ausgabearrays zurückliefern. Die Funktion liefert also ein Tupel ('samples', 'step') zurück:

```
import numpy as np

samples, spacing = np.linspace(1, 10,
```

```
                                retstep=True)
print(spacing)
samples, spacing = np.linspace(1, 10, 5,
                                endpoint=True, retstep=True)
print(samples, spacing)
samples, spacing = np.linspace(1, 10, 5,
                                endpoint=False, retstep=True)
print(samples, spacing)
```

```
0.1836734693877551
[ 1.    3.25  5.5   7.75 10.  ] 2.25
[1.  2.8 4.6 6.4 8.2] 1.8
```

Nulldimensionale Arrays in NumPy In NumPy kann man mehrdimensionale Arrays erzeugen. Skalare sind 0-dimensional. Im folgenden Beispiel erzeugen wir den Skalar `42`. Wenden wir die `ndim`-Methode auf unseren Skalar an, erhalten wir die Dimension des Arrays. Wir können außerdem sehen, dass das Array vom Typ `numpy.ndarray` ist.

```
import numpy as np
x = np.array(42)
print("x: ", x)
print("Der Typ von x: ", type(x))
print("Die Dimension von x:", np.ndim(x))
```

```
x:  42
Der Typ von x:  <class 'numpy.ndarray'>
Die Dimension von x: 0
```

Eindimensionales Array Wir haben bereits in unserem anfänglichen Beispiel ein eindimensionales Array – besser als Vektoren bekannt – gesehen. Was wir bis jetzt noch nicht erwähnt haben, aber was Sie sich sicherlich bereits gedacht haben, ist die Tatsache, dass die NumPy-Arrays Container sind, die nur einen Typ enthalten können, also beispielsweise nur Integers. Den homogenen Datentyp eines Arrays können wir mit dem Attribut `dtype` bestimmen, wie wir im folgenden Beispiel lernen können:

```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])
V = np.array([3.4, 6.9, 99.8, 12.8])
print("F: ", F)
print("V: ", V)
print("Typ von F: ", F.dtype)
print("Typ von V: ", V.dtype)
print("Dimension von F: ", np.ndim(F))
print("Dimension von V: ", np.ndim(V))
```

```
F:  [ 1  1  2  3  5  8 13 21]
V:  [ 3.4  6.9 99.8 12.8]
Typ von F:  int64
Typ von V:  float64
Dimension von F:  1
Dimension von V:  1
```

Zwei- und Mehrdimensionale Arrays Natürlich sind die Arrays in NumPy nicht auf eine Dimension beschränkt. Sie können eine beliebige Dimension haben. Wir erzeugen sie, indem wir verschachtelte Listen (oder Tupel) an die `array`-Methode von NumPy übergeben:

```
A = np.array([ [3.4, 8.7, 9.9],
               [1.1, -7.8, -0.7],
               [4.1, 12.3, 4.8]])
print(A)
print(A.ndim)
```

```
[[ 3.4  8.7  9.9]
 [ 1.1 -7.8 -0.7]
 [ 4.1 12.3  4.8]]
2
```

Dreidimensionale Arrays werden wir uns in einem späteren Kapitel genauer anschauen.

Shape/Gestalt eines Arrays

Die Funktion `shape` liefert die Größe bzw. die Gestalt eines Arrays in Form eines Integer-Tupels zurück. Diese Zahlen bezeichnen die Längen der entsprechenden Array-Dimensionen, d.h. im zweidimensionalen Fall den Zeilen und Spalten. In anderen Worten: Die Gestalt oder Shape eines Arrays ist ein Tupel mit der Anzahl der Elemente pro Achse (Dimension). In unserem Beispiel ist die Shape gleich (6, 3). Das bedeutet, das wir sechs Zeilen und drei Spalten haben.

```
x = np.array([ [67, 63, 87],
               [77, 69, 59],
               [85, 87, 99],
               [79, 72, 71],
               [63, 89, 93],
               [68, 92, 78]])
print(np.shape(x))
```

```
(6, 3)
```

Es gibt auch eine äquivalente Array-Property:

```
print(x.shape)
```

```
(6, 3)
```

Die Shape eines Arrays sagt uns auch etwas über die Reihenfolge, in der die Indizes ausgeführt werden, d.h. zuerst die Zeilen, dann die Spalten und dann gegebenenfalls eine weitere Dimension oder weitere Dimensionen.

`shape` kann auch dazu genutzt werden, die "Shape" eines Arrays zu ändern:

```
x.shape = (3, 6)
print(x)
```

```
[[67 63 87 77 69 59]
 [85 87 99 79 72 71]
```

```
 [63 89 93 68 92 78]]
```

```
x.shape = (2, 9)
print(x)
```

```
[[67 63 87 77 69 59 85 87 99]
 [79 72 71 63 89 93 68 92 78]]
```

Viele haben sicherlich bereits vermutet, dass die neue Shape der Anzahl der Elemente des Arrays entsprechen muss, d.h. die totale Größe des neuen Arrays muss die gleiche wie die alte sein. Eine Ausnahme wird erhoben, wenn dies nicht der Fall ist, wenn man in unserem Fall zum Beispiel

`x.shape = (4, 4)`

eingibt.

Die Shape eines Skalars ist ein leeres Tupel:

```
x = np.array(11)
print(np.shape(x))
```

```
()
```

Im Folgenden sehen wir die Shape eines dreidimensionalen Arrays:

```
B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B.shape)
```

```
(3, 2, 2)
```

Indizierung und Teilbereichsoperator

Der Zugriff oder die Zuweisung an die Elemente eines Arrays funktioniert ähnlich wie bei den sequentiellen Datentypen von Python, d.h. den Listen und Tupeln. Außerdem haben wir verschiedene Möglichkeiten zu indizieren. Dies macht das Indizieren in NumPy sehr mächtig und ähnlich zum Indizieren und dem Teilbereichsoperator der Listen.

Einzelne Elemente zu indizieren funktioniert so, wie es die meisten wahrscheinlich erwarten:

```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])
# Ausgabe des ersten Elements von F
print(F[0])
# Ausgabe letztes Element von F
print(F[-1])
```

```
1
21
```

Mehrdimensionale Arrays indizieren:

```
A = np.array([ [3.4, 8.7, 9.9],
```

```
                [1.1, -7.8, -0.7],
                [4.1, 12.3, 4.8]])
print(A[1][0])

B = np.array([ [[111, 112], [121, 122]],
                [[211, 212], [221, 222]],
                [[311, 312], [321, 322]] ])
print(B[0][1][0])
```

```
1.1
121
```

Wir haben auf das Element in der zweiten Zeile, d.h. die Zeile mit dem Index 1, und der ersten Spalte (Index 0) zugegriffen. Auf dieses Element haben wir in der gleichen Art zugegriffen, wie wir mit einem Element in einer verschachtelten Python-Liste verfahren wären.

Es gibt aber auch eine Alternative: Wir benutzen nur ein Klammernpaar, und alle Indizes werden mit Kommas separiert:

```
print(A[1, 0])
```

```
1.1
```

Man muss sich aber der Tatsache bewusst sein, dass die zweite Art prinzipiell effizienter ist. Im ersten Fall erzeugen wir als Zwischenschritt ein Array `A[1]`, in dem wir dann auf das Element mit dem Index 0 zugreifen. Dies entspricht in etwa dem Folgenden:

```
tmp = A[1]
print(tmp)
print(tmp[0])
```

```
[ 1.1 -7.8 -0.7]
1.1
```

Wir nehmen an, dass Sie bereits vertraut sind mit den

Teilbereichsoperatoren

(slicing) von den Listen und Tupeln.

Das englische Verb “to slice” bedeutet in Deutsch “schneiden” oder auch “in Scheiben schneiden”. Letztere Bedeutung entspricht auch der Arbeitsweise des Teilbereichsoperators in Python und NumPy. Man schneidet sich gewissermaßen eine “Scheibe” aus einem sequentiellen Datentyp oder einem Array heraus.

Die Syntax in NumPy ist analog zu der von Standardpython im Falle von eindimensionalen Arrays. Allerdings können wir “Slicing” auch auf mehrdimensionale Arrays anwenden.

Die allgemeine Syntax für den eindimensionalen Fall lautet wie folgt:

`[start:stop:step]`

Wir demonstrieren die Arbeitsweise des Teilbereichsoperators an einigen Beispielen. Wir beginnen mit dem einfachsten Fall, also dem eindimensionalen Array:

```
S = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
print(S[2:5])
print(S[:4])
print(S[6:])
print(S[:])
```

```
[2 3 4]
[0 1 2 3]
[6 7 8 9]
[0 1 2 3 4 5 6 7 8 9]
```

Die Anwendung des Teilbereichsoperators auf mehrdimensionale Arrays illustrieren wir in den folgenden Beispielen. Die Bereiche für jede Dimension werden durch Kommas getrennt:

```
A = np.array([
[11, 12, 13, 14, 15],
[21, 22, 23, 24, 25],
[31, 32, 33, 34, 35],
[41, 42, 43, 44, 45],
[51, 52, 53, 54, 55]])

print(A[:3, 2:])
```

```
[[13 14 15]
 [23 24 25]
 [33 34 35]]
```

```
print(A[3:, :])
```

```
[[41 42 43 44 45]
 [51 52 53 54 55]]
```

```
print(A[:, 4:])
```

```
[[15]
 [25]
 [35]
 [45]
 [55]]
```

Die folgenden beiden Beispiele benutzten auch noch den dritten Parameter `step`. Die `reshape`-Funktion benutzen wir, um ein eindimensionales Array in ein zweidimensionales zu wandeln. Wir werden `reshape` im folgenden Unterkapitel erklären:

```
X = np.arange(28).reshape(4, 7)
print(X)
```

```
[[ 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]]
```

```
print(X[::2, ::3])
```

```
[[ 0  3  6]
 [14 17 20]]
```

```
print(X[::, ::3])
```

```
[[ 0  3  6]
 [ 7 10 13]
 [14 17 20]
 [21 24 27]]
```

Falls die Zahl der Objekte in dem Auswahltupel kleiner als die Dimension N ist, dann wird ":" für die weiteren, nicht angegebenen Dimensionen angenommen:

```
A = np.array(
    [ [ [45, 12, 4], [45, 13, 5], [46, 12, 6] ],
      [ [46, 14, 4], [45, 14, 5], [46, 11, 5] ],
      [ [47, 13, 2], [48, 15, 5], [52, 15, 1] ] ])

A[1:3, 0:2]  # equivalent to A[1:3, 0:2, :]
```

```
array([[[46, 14,  4],
        [45, 14,  5]],

       [[47, 13,  2],
        [48, 15,  5]]])
```

Achtung: Während der Teilbereichsoperator bei Listen und Tupel neue Objekte erzeugt, generiert er bei NumPy nur eine Sicht (englisch: "view") auf das Originalarray. Dadurch erhalten wir eine andere Möglichkeit, das Array anzusprechen, oder besser einen Teil des Arrays. Daraus folgt, dass wenn wir etwas in einer Sicht verändern, wir auch das Originalarray verändern:

```
A = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
S = A[2:6]
S[0] = 22
S[1] = 23
print(A)
```

```
[ 0  1 22 23  4  5  6  7  8  9]
```

Wenn wir das analog bei Listen tun, sehen wir, dass wir eine Kopie erhalten. Genaugenommen müssten wir sagen, eine flache Kopie. Den Unterschied zwischen flacher und tiefer Kopie haben wir in unserem Kapitel "Flaches und tiefes Kopieren" ausführlich erklärt.

```
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
lst2 = lst[2:6]
lst2[0] = 22
lst2[1] = 23
print(lst)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Will man prüfen, ob zwei Arrays auf den gleichen Speicherbereich zugreifen, so kann man die Funktion `np.may_share_memory` benutzen:

`np.may_share_memory(A, B)`

Um zu entscheiden, ob sich zwei Arrays `A` und `B` Speicher teilen, werden die Speichergrenzen von `A` und `B` berechnet. Die Funktion liefert `True` zurück, falls sie überlappen, und ansonsten `False`.

```
A = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
B = A[2:5]

np.may_share_memory(A, B)
```

```
True
```

Die Funktion kann allerdings falsch-positive Ergebnisse liefern, d.h. falls sie `True` zurückliefert, könnten die Arrays dennoch verschieden sein.

Im folgenden Beispiel haben `B1` und `B2` keine gemeinsamen Daten, aber die Speicherorte sind verzahnt, da beide ja "nur" eine View auf A darstellen:

```
A = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
B1 = A[::2]
B2 = A[1::2]
print(np.may_share_memory(B1, B2))
```

```
True
```

Das obige Beispiel ist ein falsch-positiv-Beispiel für `may_share_memory` in dem Sinn, dass jemand wegen der Ausgabe `True` denken könnte, dass die Arrays gemeinsamen Speicher hätten.

Dreidimensionale Arrays

Dreidimensionale Arrays sind vom Zugriff her etwas schwerer vorstellbar. Betrachten wir dazu das folgende Beispielarray:

```
import numpy as np
X = np.array(
    [[[3, 1, 2],
      [4, 2, 2]],

     [[-1, 0, 1],
      [1, -1, -2]],

     [[3, 2, 2],
      [4, 4, 3]],

     [[2, 2, 1],
      [3, 1, 3]]])

print(X.shape)
```

```
(4, 2, 3)
```

Wir sehen, dass dieses Array eine Shape `(4, 2, 3)` hat. Wir benutzen nun die Slicing-Funktionalität, um uns die Schnitte durch die Dimensionen zu veranschaulichen:

```
print("Dimension 0 with size ", X.shape[0])
for i in range(X.shape[0]):
    print(f"\nAusgabe von X[{i:1},:,:]:")
    print(X[i,:,:])

print("\nDimension 1 with size ", X.shape[1])
for i in range(X.shape[1]):
    print(f"\nAusgabe von X[:,{i:1},:]:")
    print(X[:,i,:])

print("\nDimension 2 with size ", X.shape[2])
for i in range(X.shape[2]):
    print(f"\nAusgabe von X[:,:,{i:1}]:")
    print(X[:,:,i])
```

```
Dimension 0 with size  4

Ausgabe von X[0,:,:]:
[[3 1 2]
 [4 2 2]]

Ausgabe von X[1,:,:]:
[[-1  0  1]
 [ 1 -1 -2]]

Ausgabe von X[2,:,:]:
[[3 2 2]
 [4 4 3]]

Ausgabe von X[3,:,:]:
[[2 2 1]
 [3 1 3]]

Dimension 1 with size  2

Ausgabe von X[:,0,:]:
[[ 3  1  2]
 [-1  0  1]
 [ 3  2  2]
 [ 2  2  1]]

Ausgabe von X[:,1,:]:
[[ 4  2  2]
 [ 1 -1 -2]
 [ 4  4  3]
 [ 3  1  3]]

Dimension 2 with size  3

Ausgabe von X[:,:,0]:
[[ 3  4]
 [-1  1]
 [ 3  4]
 [ 2  3]]
```

```
Ausgabe von X[:,:,1]:
[[ 1  2]
 [ 0 -1]
 [ 2  4]
 [ 2  1]]

Ausgabe von X[:,:,2]:
[[ 2  2]
 [ 1 -2]
 [ 2  3]
 [ 1  3]]
```

Arrays mit Nullen und Einsen

Arrays können auf zwei Arten mit Nullen und Einsen initialisiert werden. Die Methode `ones(t)` hat als Parameter ein Tupel `t` mit der Shape des Arrays und erzeugt entsprechend ein Array mit Einsen. Defaultmäßig wird es mit Float-Einsen gefüllt. Wenn man Integer-Einsen benötigt, kann man den optionalen Parameter `dtype` auf `int` setzen:

```python
import numpy as np

E = np.ones((2, 3))
print(E)

F = np.ones((3, 4), dtype=int)
print(F)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

Was wir über die Methode `ones` gesagt haben, gilt analog auch für die Methode `zeros`, wie wir im folgenden Beispiel erkennen können:

```python
Z = np.zeros((2, 4))
print(Z)

Z = np.zeros((2, 4), dtype=int)
print(Z)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[0 0 0 0]
 [0 0 0 0]]
```

Es gibt noch eine andere interessante Möglichkeit, ein Array mit Einsen oder Nullen zu erzeugen, wenn es die gleiche Shape wie ein anderes existierendes Array 'a' haben soll. Für diesen Zweck stellt NumPy die Methoden `ones_like` und `zeros_like` zur Verfügung:

```
x = np.array([2, 5, 18, 14, 4])
E = np.ones_like(x)
print(E)

Z = np.zeros_like(x)
print(Z)
```

```
[1 1 1 1 1]
[0 0 0 0 0]
```

Arrays kopieren

numpy.copy(A) und A.copy() Zum Kopieren eines NumPy-Arrays `A` gibt es generell zwei Möglichkeiten:

- `numpy.copy(A)`
- `A.copy()`

Beide sind nahezu gleich und liefern jeweils eine Kopie des Arrays `A` zurück. Sie unterscheiden sich aber beim Defaultwert des optionalen Parameters. Bei `numpy.copy(obj)` steht der Defaultwert auf `order='K'`, und bei `obj.copy()` steht er auf `order='C'`.

Parameter	Bedeutung
obj	array-ähnliche Eingabedaten
order	Die möglichen Werte sind {'C', 'F', 'A', 'K'}. Dieser Parameter kontrolliert das Speicher-Layout der Kopie. 'C' bedeutet C-Reihenfolge oder C-zusammenhängend, 'F' Fortran-zusammenhängend, 'A' verhält sich wie 'F', falls das Objekt 'obj' in Fortran-Reihenfolge ist, ansonsten verhält sich 'A' wie 'C'. 'K' bedeutet, dass das Layout von 'obj' so nahe wie möglich realisiert werden soll.

Zusammenhängend gespeicherte Arrays Um den Parameter `order` zu verstehen, gehen wir noch kurz auf den Begriff "zusammenhängend" (englisch "contiguous") ein. Die Speicherstruktur eines Arrays wird als zusammenhängend bezeichnet, wenn die Speicherung eines Arrays entweder C-zusammenhängend (`C_CONTIGUOUS`) oder Fortran-zusammenhängend (`F_CONTIGUOUS`) erfolgt. Betrachten wir dazu folgendes Array:

Wenn dieses Array zeilenweise abgespeichert ist, spricht man von C-zusammenhängend. Wir veranschaulichen dies im folgenden Bild:

Wird ein Array entsprechend spaltenweise abgespeichert, so bezeichnet man dies als F-zusammenhängend:

Wir demonstrieren nun, wie man die Speicherart mittels des Parameters `order` bestimmen kann:

```
import numpy as np
```

```
F = np.array([[11, 12, 13, 14],
              [21, 22, 23, 24],
              [31, 32, 33, 34]], order='F')

C = F.copy()
C2 = np.copy(F)

print("F array: \n", F)
print("C array: \n", C)

print("Ist F 'C continuous?': ", F.flags['C_CONTIGUOUS'])
print("Ist C 'C continuous?': ", C.flags['C_CONTIGUOUS'])
print("Ist C 'C continuous?': ", C2.flags['C_CONTIGUOUS'])
```

```
F array:
 [[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
C array:
 [[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
Ist F 'C continuous?':  False
Ist C 'C continuous?':  True
Ist C 'C continuous?':  False
```

Transponiert man ein Array, werden die Daten nicht umgespeichert, sondern lediglich das `strides`-Attribut entsprechend angepasst, d.h. die Speicherung wird anders interpretiert.

```
T = C.transpose()
print(C.flags['C_CONTIGUOUS'])
print(T.flags['C_CONTIGUOUS'])
print(T.strides, C.strides)
```

```
True
False
(8, 32) (32, 8)
```

```
T = F.transpose()
print(F.flags['C_CONTIGUOUS'])
print(T.flags['C_CONTIGUOUS'])
print(T.strides, F.strides)
```

```
False
True
(24, 8) (8, 24)
```

Wir sehen, dass `F.strides` das Tupel `(8, 24)` zurückliefert. Dies besagt, dass man 24 Bytes – was drei Zahlen à 8 Bytes entspricht – überspringen muss, um von einem Arrayelement zum benachbarten Spaltenelement zu kommen. In anderen Worten: Wegen der Fortran-zusammenhängenden Speicherung liegen im Speicher zwischen den Werten von `F[1, 1]` und `F[1, 2]` die Werte von `F[2, 1]` und `F[3, 1]`. Die erste Komponente `F.strides` besagt, dass man entsprechend 8 Bytes überspringen muss, um zum benachbarten Element der nächsten Zeile zu kommen. Also zwischen den Werten von `F[1, 1]` und `F[2, 1]` liegen keine weiteren Werte.

Im Folgenden können wir noch sehen, wie sich die verschiedenen möglichen Werte für `order` auswirken.

```
order_values = ['C', 'F', 'A', 'K']
for order in order_values:
    R1 = F.copy(order=order)
    R2 = C.copy(order=order)
    print(f"R1: order='{order}': ",
          R1.flags['C_CONTIGUOUS'],
          R1.flags['F_CONTIGUOUS'])
    print(f"R2: order='{order}': ",
          R2.flags['C_CONTIGUOUS'],
          R2.flags['F_CONTIGUOUS'])
```

```
R1: order='C':  True False
R2: order='C':  True False
R1: order='F':  False True
R2: order='F':  False True
R1: order='A':  False True
R2: order='A':  True False
R1: order='K':  False True
R2: order='K':  True False
```

Identitäts-Array

In der linearen Algebra versteht man unter der Identitätsmatrix oder Einheitsmatrix eine quadratische Matrix, deren Hauptdiagonalelemente eins und deren Außerdiagonalelemente null sind.

NumPy bietet zwei Möglichkeiten, solche Arrays zu erzeugen:

- identy
- eye

Die identity-Funktion Wir können Identitäts-Arrays mit der Funktion `identity` generieren:

```
identity(n, dtype=None)
```

Parameter	Bedeutung
n	Eine Integer-Zahl, welche die Anzahl der Zeilen und Spalten der Ausgabe definiert, d.h. 'n' x 'n'
dtype	Ein optionales Argument, welches den Datentyp des Ergebnisses definiert. Der Default ist 'float'

Die Ausgabe von `identity` ist ein `n` x `n`-Array, in dem die Elemente auf der Hauptdiagonalen auf 1 gesetzt sind und alle anderen Elemente auf 0.

```
import numpy as np
```

```
print(np.identity(4))
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

```
print(np.identity(4, dtype=int))
```

```
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

Die eye-Funktion Die Funktion `eye` bietet eine andere Möglichkeit, Identitätsarrays, aber auch allgemeine Diagonalarrays, mit Einsen zu erzeugen. Die Ausgabe von `eye` ist ein zweidimensionales Array, in dem die Elemente auf der Hauptdiagonalen auf 1 gesetzt sind und alle anderen Elemente auf 0.

`eye(N, M=None, k=0, dtype=float)`

Parameter	Bedeutung
N	Eine Integer-Zahl, welche die Anzahl der Zeilen des Ausgebearrays bestimmt.
M	Eine Integer-Zahl, welche die Spalten des Ausgabearrays bestimmt. Falls dieser Parameter nicht gesetzt ist oder None ist, wird er per Default auf 'N' gesetzt.
k	Mit 'k' wird die Position der Diagonalen gesetzt. Der Default ist 0. 0 bezeichnet die Hauptdiagonale. Ein positiver Wert bezeichnet eine obere Diagonale und ein negativer Wert eine untere Diagonale.
dtype	Ein optionales Argument, welches den Datentyp des Ergebnisses definiert. Der Default ist 'float'.

`eye` liefert ein ndarray der Shape `(N, M)` zurück. Alle Elemente dieses Arrays sind 0 außer denen auf der `k`-ten Diagonale, die auf 1 gesetzt sind.

```
import numpy as np

print(np.eye(5, 8, k=1, dtype=int))
```

```
[[0 1 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0]
 [0 0 0 1 0 0 0 0]
```

```
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]]
```

Die Arbeitsweise des Parameters `d` von `eye` illustrieren wir in folgendem Diagramm:

4 Numpy Dtype

4.0.1 Datentyp-Objekt: dtype

dtype

Das Datentypobjekt 'dtype' ist eine Instanz der numpy.dtype-Klasse. Es kann mit numpy.dtype konstruiert werden.

Bis jetzt haben wir in unseren Beispielen von NumPy-Arrays nur grundlegende numerische Typen wie 'int' und 'float' benutzt. Diese NumPy-Arrays enthielten nur homogene Datentypen. dtype-Objekte werden aus einer Kombination von grundlegenden Datentypen erzeugt. Mit Hilfe von dtype sind wir in der Lage "Strukturierte Arrays" zu erzeugen, auch bekannt als "record arrays". Strukturierte Arrays statten uns mit der Möglichkeit aus verschiedene Datentypen in verschiedenen Spalten zu haben. Es gibt somit Ähnlichkeit zu Excel- oder CSV-Dokumenten. Dies ermöglicht es uns somit Daten wie in der folgenden Tabelle mit dtype zu erzeugen:

Land	Bevölkerungsdichte	Fläche	Einwohner
Niederlande	393	41526	16,928,800
Belgien	337	30510	11,007,020
Vereinigtes Königreich	256	243610	62,262,000
Deutschland	233	357021	81,799,600
Liechtenstein	205	160	32,842
Italien	192	301230	59,715,625
Schweiz	177	41290	7,301,994
Luxemburg	173	2586	512,000
Frankreich	111	547030	63,601,002
Österreich	97	83858	8,169,929
Griechenland	81	131940	11,606,813
Irland	65	70280	4,581,269
Schweden	20	449964	9,515,744
Finnland	16	338424	5,410,233
Norwegen	13	385252	5,033,675

Bevor wir jedoch mit komplexen Daten wie den obigen starten, wollen wir dtype an einem sehr einfachen Beispiel einführen. Wir definieren einen `Pixel`-Datentyp, der einem `numpy.uint8`-Datentyp entspricht.

Die Elemente der Liste `lst` werden in Pixel-Typen gewandelt, um das zweidimensionale Array `A` zu erzeugen. Wir können sehen, dass dabei auch Float-Werte automatisch in Pixel-Datentypen, also `numpy.uint8`, gewandelt werden.

```
import numpy as np
```

```
Pixel = np.dtype(np.uint8)
print(Pixel)

lst = [ [115, 230.9, 229.2, 234],
        [117, 229, 232.1, 235],
        [116, 140, 141, 142] ]

A = np.array(lst, dtype=Pixel)

print(A)
```

```
uint8
[[115 230 229 234]
 [117 229 232 235]
 [116 140 141 142]]
```

Im vorigen Beispiel haben wir lediglich einen neuen Namen für einen Basisdatentyp eingeführt. Damit kann man beispielsweise die Lesbarkeit und Verständlichkeit eines Programmes erhöhen. Dies hat noch nichts mit "Strukturierten Arrays" zu tun, die wir am Anfang dieses Kapitels

unseres Tutorials

erwähnt hatten.

Strukturierte Arrays

ndarrays sind homogene Datenobjekte, d.h. alle Elemente eines Arrays haben den gleichen Datentyp. Der Datentyp `dtype` hingegen erlaubt es uns, spaltenweise Typen zu deklarieren.

Nun gehen wir den ersten Schritt in Richtung Implementierung der Tabelle europäischer Länder mit den Informationen über Fläche, Bevölkerung und Bevölkerungsdichte.

Wir erzeugen ein strukturiertes Array mit einer Spalte `density`. Den Datentyp definieren wir als `np.dtype([('density', np.int)])`. Diesen Datentyp weisen wir der Variablen `Density` zu. Wir haben die Variable groß geschrieben, damit man sieht, dass es einen Unterschied gibt zu dem `density` in der Typdefinition selbst. Den Datentyp `Density` benutzen wir dann in der Definition des NumPy-Arrays, in dem wir die ersten drei Werte benutzen:

```
import numpy as np

Density = np.dtype([('density', np.int32)])

x = np.array([(393,), (337,), (256,)],
             dtype=Density)

print(x)

print("\nDie interne Darstellung:")
print(repr(x))
```

```
[(393,) (337,) (256,)]
```

```
Die interne Darstellung:
array([(393,), (337,), (256,)], dtype=[('density', '<i4')])
```

Wir können auf die `density`-Spalte zugreifen, indem wir als Schlüssel `density` eingeben. Es ähnelt einem

Dictionary-Zugriff

in Python:

```
print(x['density'])
```

```
[393 337 256]
```

Man wird sich vielleicht wundern, dass wir `np.int32` in unserer Definition benutzt haben, aber dass die interne Repräsentierung `<i4` zeigt.

In einer `dtype`-Definition können wir den Typ direkt verwenden, also beispielsweise `np.int32`, oder wir können einen String benutzen, z.B. `i4`.

In unserem Beispiel hätten wir unseren `dtype` auch wie folgt definieren können:

```
Density = np.dtype([('density',  'i4')])
x = np.array([(393,), (337,), (256,)],
             dtype=Density)


print(x)
```

```
[(393,) (337,) (256,)]
```

Das `i` steht für Integer, und die 4 bedeutet "4 Bytes". Was bedeutet aber das Kleiner-als-Zeichen vor der 4? Wir hätten ebensogut '<i4' schreiben können. Wir können einem Typ ein '<'- oder ein '>'-Zeichen voranstellen. '<' bedeutet, dass bei der Speicherorganisation Little-Endian verwendet wird, und '>' bedeutet entsprechend, dass Big-Endian verwendet wird. Ohne Präfix wird die natürliche Byte-Reihenfolge des Systems verwendet. Wir demonstrieren dies im folgenden Beispiel, indem wir eine Gleitkommazahl mit doppelter Genauigkeit (double precision floating-point) in verschiedenen Byte-Reihenfolgen definieren:

```
# little-endian ordering
dt = np.dtype('<d')
print(dt.name, dt.byteorder, dt.itemsize)

# big-endian ordering
dt = np.dtype('>d')
print(dt.name, dt.byteorder, dt.itemsize)

# native byte ordering
dt = np.dtype('d')
print(dt.name, dt.byteorder, dt.itemsize)
```

```
float64 = 8
float64 > 8
float64 = 8
```

Das Gleichheitszeichen '=' steht für die natürliche Byte-Reihenfolge ('native byte ordering'),

definiert durch das Betriebssystem. In unserem Fall bedeutet dies Little-Endian, weil wir uns auf einem Linux-Rechner befinden.

Eine andere Sache in unserem Density-Array könnte verwirrend sein. Wir definierten das Array mit einer Liste, die 1-Tupels enthält. Vielleicht fragen Sie sich nun, ob es möglich ist Tupels und Listen austauschbar zu verwenden? Dies ist nicht möglich. Die Tupels werden verwendet um die Records zu definieren - in unserm Fall bestehen diese nur aus der Bevölkerungsdichte ('density') -, und die Liste ist der 'Container' für die Records. Die Tupel definieren die atomaren Elemente der Struktur und die Listen die Dimensionen.

Nun werden wir die Ländernamen, die Flächen und die Populationen zu unserem Typ hinzufügen:

```
dt = np.dtype([('country', 'S20'),
               ('density', 'i4'),
               ('area', 'i4'),
               ('population', 'i4')])
population_table = np.array([('Netherlands', 393, 41526, 16928800),
('Belgium', 337, 30510, 11007020),
('United Kingdom', 256, 243610, 62262000),
('Germany', 233, 357021, 81799600),
('Liechtenstein', 205, 160, 32842),
('Italy', 192, 301230, 59715625),
('Switzerland', 177, 41290, 7301994),
('Luxembourg', 173, 2586, 512000),
('France', 111, 547030, 63601002),
('Austria', 97, 83858, 8169929),
('Greece', 81, 131940, 11606813),
('Ireland', 65, 70280, 4581269),
('Sweden', 20, 449964, 9515744),
('Finland', 16, 338424, 5410233),
('Norway', 13, 385252, 5033675)],
             dtype=dt)
print(x[:4])
```

```
[(393,) (337,) (256,)]
```

Wir können auf jedes Element individuell zugreifen:

```
print(population_table['density'])
print(population_table['country'])
print(population_table['area'][2:5])
```

```
[393 337 256 233 205 192 177 173 111  97  81  65  20  16  13]
[b'Netherlands' b'Belgium' b'United Kingdom' b'Germany' b'Liechtenstein'
 b'Italy' b'Switzerland' b'Luxembourg' b'France' b'Austria' b'Greece'
 b'Ireland' b'Sweden' b'Finland' b'Norway']
[243610 357021    160]
```

Die Städtenamen sind Instanzen der NumPy-Klasse `numpy.bytes_`. Man kann sie mit der Funktion `str` wieder in Unicode-Strings wandeln. Weiter unten werden wir auch sehen, wie man direkt mit Unicode-Strings in dtype-Arrays arbeitet.

```
s = population_table['country'][0]
print(s, type(s))
```

```
s = str(s)
print(s, type(s))
```

```
b'Netherlands' <class 'numpy.bytes_'>
b'Netherlands' <class 'str'>
```

Ein- und Ausgabe von strukturierten Arrays

In den meisten Applikationen ist es notwendig, die Daten aus einem Programm in einer Datei zu speichern. Wir werden nun unser zuvor erzeugtes Array in einer Datei mit dem Kommando `savetxt` speichern. Eine detaillierte Einführung in diese Thematik findet man im Kapitel

Lesen und Schreiben von Daten-Dateien

```
np.savetxt("population_table.csv",
           population_table,
           fmt="%s;%d;%d;%d",
           delimiter=";")
```

Sehr wahrscheinlich wird man zu einem späteren Zeitpunkt die Daten der eben gespeicherten Datei wieder einlesen wollen. Dies können wir mit dem Kommando `genfromtxt` bewerkstelligen.

```
dt = np.dtype([('country', np.unicode, 20),
               ('density', 'i4'),
               ('area', 'i4'),
               ('population', 'i4')])

x = np.genfromtxt("population_table.csv",
               dtype=dt,
               delimiter=";")

print(x)
```

```
[("b'Netherlands'", 393,  41526, 16928800)
 ("b'Belgium'", 337,  30510, 11007020)
 ("b'United Kingdom'", 256, 243610, 62262000)
 ("b'Germany'", 233, 357021, 81799600)
 ("b'Liechtenstein'", 205,    160,    32842)
 ("b'Italy'", 192, 301230, 59715625)
 ("b'Switzerland'", 177,  41290,  7301994)
 ("b'Luxembourg'", 173,   2586,   512000)
 ("b'France'", 111, 547030, 63601002)
 ("b'Austria'",  97,  83858,  8169929)
 ("b'Greece'",  81, 131940, 11606813)
 ("b'Ireland'",  65,  70280,  4581269)
 ("b'Sweden'",  20, 449964,  9515744)
 ("b'Finland'",  16, 338424,  5410233)
 ("b'Norway'",  13, 385252,  5033675)]
```

Statt `genfromtxt` kann man auch `loadtxt` verwenden:

```
dt = np.dtype([('country',
```

```
                np.unicode, 25),
               ('density', 'i4'),
               ('area', 'i4'),
               ('population', 'i4')])

x = np.loadtxt("population_table.csv",
               dtype=dt,
               delimiter=";")

print(x)
```

```
[("b'Netherlands'", 393,  41526, 16928800)
 ("b'Belgium'", 337,  30510, 11007020)
 ("b'United Kingdom'", 256, 243610, 62262000)
 ("b'Germany'", 233, 357021, 81799600)
 ("b'Liechtenstein'", 205,    160,    32842)
 ("b'Italy'", 192, 301230, 59715625)
 ("b'Switzerland'", 177,  41290,  7301994)
 ("b'Luxembourg'", 173,   2586,   512000)
 ("b'France'", 111, 547030, 63601002)
 ("b'Austria'",  97,  83858,  8169929)
 ("b'Greece'",  81, 131940, 11606813)
 ("b'Ireland'",  65,  70280,  4581269)
 ("b'Sweden'",  20, 449964,  9515744)
 ("b'Finland'",  16, 338424,  5410233)
 ("b'Norway'",  13, 385252,  5033675)]
```

Unicode-Strings in Arrays

Wir hatten ja bereits darauf hingewiesen, dass die Strings in unserem vorigen Beispielarray ein kleines `b` als Präfix hatten. Dies kam dadurch, dass wir in unserer `dtype`-Definition `('country', 'S20')` geschrieben und dadurch unsere Ländernamen als Binärstrings definiert hatten.

Um Unicode-Strings zu erhalten, müssen wir die Definition in `('country', np.unicode, 20)` umändern. Wir ändern die Definition für `population_table` wie folgt:

```
dt = np.dtype([('country', np.unicode, 25),
               ('density', 'i4'),
               ('area', 'i4'),
               ('population', 'i4')])
population_table = np.array([
    ('Netherlands', 393, 41526, 16928800),
    ('Belgium', 337, 30510, 11007020),
    ('United Kingdom', 256, 243610, 62262000),
    ('Germany', 233, 357021, 81799600),
    ('Liechtenstein', 205, 160, 32842),
    ('Italy', 192, 301230, 59715625),
    ('Switzerland', 177, 41290, 7301994),
    ('Luxembourg', 173, 2586, 512000),
    ('France', 111, 547030, 63601002),
    ('Austria', 97, 83858, 8169929),
    ('Greece', 81, 131940, 11606813),
    ('Ireland', 65, 70280, 4581269),
    ('Sweden', 20, 449964, 9515744),
```

```
    ('Finland', 16, 338424, 5410233),
    ('Norway', 13, 385252, 5033675)],
    dtype=dt)
print(population_table[:4])
```

```
[('Netherlands', 393,  41526, 16928800) ('Belgium', 337,  30510, 11007020)
 ('United Kingdom', 256, 243610, 62262000)
 ('Germany', 233, 357021, 81799600)]
```

Umbenennen von Spaltennamen

Nun wollen wir die Spaltennamen in deutsche Bezeichnungen umbenennen. Auf die Spaltennamen kann man mit der Property `names` von dtype zugreifen:

```
print(population_table.dtype.names)
```

```
('country', 'density', 'area', 'population')
```

Das Umbenennen gestaltet sich denkbar einfach. Man weist dieser Property einfach ein neues Tupel mit den neuen Namen zu:

```
population_table.dtype.names = ('Land',
                                'Bevölkerungsdichte',
                                'Fläche',
                                'Bevölkerung')

print(population_table['Land'])
```

```
['Netherlands' 'Belgium' 'United Kingdom' 'Germany' 'Liechtenstein'
 'Italy' 'Switzerland' 'Luxembourg' 'France' 'Austria' 'Greece' 'Ireland'
 'Sweden' 'Finland' 'Norway']
```

Spaltenwerte austauschen

Nun wollen wir auch die Ländernamen von Englisch nach Deutsch übersetzen. Dazu erzeugen wir eine Liste `lands`. Wir können diese mit `np.array` in ein Array wandeln und dann die bisherige Spalte `population_table['Land']` komplett austauschen:

```
lands = ['Niederlande', 'Belgien', 'Vereinigtes Königreich',
         'Deutschland', 'Liechtenstein', 'Italien', 'Schweiz',
         'Luxemburg', 'Frankreich', 'Österreich', 'Griechenland',
         'Irland', 'Schweden', 'Finnland', 'Norwegen']

population_table['Land'] = np.array(lands, dtype='<U25')

print(population_table)
```

```
[('Niederlande', 393,  41526, 16928800) ('Belgien', 337,  30510, 11007020)
 ('Vereinigtes Königreich', 256, 243610, 62262000)
 ('Deutschland', 233, 357021, 81799600)
 ('Liechtenstein', 205,    160,    32842)
 ('Italien', 192, 301230, 59715625) ('Schweiz', 177,  41290,  7301994)
```

```
('Luxemburg', 173,   2586,   512000)
('Frankreich', 111, 547030, 63601002)
('Österreich',  97,  83858,  8169929)
('Griechenland',  81, 131940, 11606813) ('Irland',  65,  70280,  4581269)
('Schweden',  20, 449964,  9515744) ('Finnland',  16, 338424,  5410233)
('Norwegen',  13, 385252,  5033675)]
```

Komplexeres Beispiel

In den bisherigen Beispielen haben wir unsere Arrays direkt erzeugt. Normalerweise müssen wir uns jedoch die Daten für unsere strukturierten Arrays aus Datenbanken oder Dateien beschaffen.

Wir werden nun die Liste benutzen, die wir im Kapitel über Dateimanagement erzeugt und gespeichert hatten. Die Liste hatten wir mit Hilfe von pickle.dumpy in der Datei cities_and_times.pkl gespeichert.

Die erste Aufgabe besteht also darin, diese Datei wieder zu ent"pickeln":

```
import pickle
fh = open("cities_and_times.pkl", "br")
cities_and_times = pickle.load(fh)
for i in range(5):
    print(cities_and_times[i])
```

```
('Amsterdam', 'Sun', (8, 52))
('Anchorage', 'Sat', (23, 52))
('Ankara', 'Sun', (10, 52))
('Athens', 'Sun', (9, 52))
('Atlanta', 'Sun', (2, 52))
```

Nun wandeln wir unsere Daten in ein strukturiertes Array:

```
time_type = np.dtype([('city', 'U30'),
                      ('day', 'U3'),
                      ('time', [('h', int), ('min', int)])])

times = np.array( cities_and_times , dtype=time_type)
print(times[:4])
```

```
[('Amsterdam', 'Sun', ( 8, 52)) ('Anchorage', 'Sat', (23, 52))
 ('Ankara', 'Sun', (10, 52)) ('Athens', 'Sun', ( 9, 52))]
```

```
lst = []
for row in times:
    t = row[2]
    t = f"{t[0]:02d}:{t[1]:02d}"
    lst.append((row[0], row[1], t))

time_type = np.dtype([('city', 'U30'),
                      ('day', 'U3'),
                      ('time', 'U5')])
times2 = np.array( lst , dtype=time_type)
```

```
print(times2[:10])
```

```
[('Amsterdam', 'Sun', '08:52') ('Anchorage', 'Sat', '23:52')
 ('Ankara', 'Sun', '10:52') ('Athens', 'Sun', '09:52')
 ('Atlanta', 'Sun', '02:52') ('Auckland', 'Sun', '20:52')
 ('Barcelona', 'Sun', '08:52') ('Beirut', 'Sun', '09:52')
 ('Berlin', 'Sun', '08:52') ('Boston', 'Sun', '02:52')]
```

Nun wollen wir diese Daten in einer csv-Datei speichern. Leider können wir die Funktion `np.savetxt` nicht nutzen, da diese Funktion nicht mit Unicode-Strings zurechtkommt. Wir benutzen deshalb die normale `write`-Methode eines File-Streams:

```
with open("cities_and_times.csv", "w") as fh:
    for city_data in times2:
        fh.write(",".join(city_data) + "\n")
```

```
# prog4book
import numpy as np

mytype = [('produktNr', np.int32), ('preise', np.float64)]

produkte = np.array([(34765, 603.76),
                     (45765, 439.93),
                     (99661, 344.19),
                     (12129, 129.39)], dtype=mytype)

print(produkte[1])
print(produkte["produktNr"])
print(produkte[2]["preise"])
print(produkte)
print(produkte.shape)
```

```
(45765, 439.93)
[34765 45765 99661 12129]
344.19
[(34765, 603.76) (45765, 439.93) (99661, 344.19) (12129, 129.39)]
(4,)
```

```
# prog4book
verkaufszahlen = np.array([3, 5, 2, 1])

erlöse = produkte["preise"] * verkaufszahlen

print("Erlöse pro Item: ", erlöse)
print("Gesamterlös: ", erlöse.sum())
```

```
Erlöse pro Item:  [1811.28 2199.65  688.38  129.39]
Gesamterlös:  4828.700000000001
```

```
# prog4book
time_type = np.dtype( [('h', int),
                       ('min', int),
                       ('sec', int)])
```

```
times = np.array([(11, 38, 5),
                  (14, 56, 0),
                  ( 3,  9, 1)], dtype=time_type)
print(times)
print(times['h'])
print(times['min'])
print(times['sec'])
```

```
[(11, 38, 5) (14, 56, 0) ( 3,  9, 1)]
[11 14  3]
[38 56  9]
[5 0 1]
```

```
# prog4book
np.column_stack((times['h'],
                 times['min'],
                 times['sec']))
```

```
array([[11, 38,  5],
       [14, 56,  0],
       [ 3,  9,  1]])
```

```
# prog4book
time_temp_type = np.dtype( np.dtype([('time', [('h', int), ('min', int),
↪  ('sec', int)]),
                                     ('temperature', float)] ))

time_temp = np.array( [((11, 42, 17), 20.8),
                       ((13, 19,  3), 23.2),
                       ((14, 50, 29), 24.6)], dtype=time_temp_type)
print(time_temp)
print(time_temp['time'])
print(time_temp['time']['h'])
print(time_temp['temperature'])
```

```
[((11, 42, 17), 20.8) ((13, 19,  3), 23.2) ((14, 50, 29), 24.6)]
[(11, 42, 17) (13, 19,  3) (14, 50, 29)]
[11 13 14]
[20.8 23.2 24.6]
```

```
# prog4book
with open("time_temp.csv", "w") as fh:
    for row in time_temp:
        zeit = [f"{el:02d}" for el in row[0]]
        zeit = ":".join(zeit)
        fh.write(zeit + " " + str(row[1]) + "\n")

!cat time_temp.csv
```

```
11:42:17 20.8
13:19:03 23.2
14:50:29 24.6
```

5 Numpy Numerische Operationen Auf Arrays

```
# invisible
#from IPython.display import HTML, display

import numpy as np
np.core.arrayprint._line_width = 65
```

5.0.1 Numerische Operationen auf NumPy-Arrays

In unserem Python-Tutorial haben wir viele Operatoren gesehen. Darüber haben wir dort gelernt, wie man Operatoren mittels "magischer Funktionen" überladen kann. Wir wissen nun, dass wir beispielsweise das "+"-Operatorzeichen nutzen können, um numerische Werte zu addieren oder Strings und Listen zu konkatenieren:

42 + 5

"Python ist eine der besten" + "oder die beste Programmiersprache!"

In dieser Einführung werden wir nun lernen, dass auch in NumPy die Operatorzeichen entsprechend überladen sind, sodass wir sie in "natürlicher" Weise nutzen können.

So können wir beispielsweise Skalare zu Arrays addieren, d.h. der Skalar wird zu jeder Komponente addiert. Das Gleiche ist möglich für die Subtraktion, die Division, die Multiplikation und sogar für Funktionen wie Sinus, Kosinus und so weiter.

Selbstverständlich können wir auch all diese Operatoren auf zwei Arrays anwenden.

Operatoren und Skalare

Beginnen wir mit der skalaren Addition:

```
# prog4book
import numpy as np
kalorientabelle = np.array([30, 52, 88, 36, 86,
                            160, 375, 115, 43, 71])

jouletabelle = kalorientabelle * 4.1868

print(jouletabelle)
```

```
[ 125.604   217.7136  368.4384  150.7248  360.0648  669.888  1570.05
  481.482   180.0324  297.2628]
```

```
# prog4book
```

```
jouletabelle = np.round((kalorientabelle * 4.1868), 0)
print(jouletabelle)
```

```
[ 126.  218.  368.  151.  360.  670. 1570.  481.  180.  297.]
```

```
# prog4book
jouletabelle = jouletabelle.astype(np.int16)
print(jouletabelle)
```

```
[ 126  218  368  151  360  670 1570  481  180  297]
```

```
# prog4web
import numpy as np
lst = [2, 3, 7.9, 3.3, 6.9, 0.11, 10.3, 12.9]
v = np.array(lst)
print(v + 2)
```

```
[ 4.    5.    9.9   5.3   8.9   2.11 12.3  14.9 ]
```

Multiplikation, Subtraktion, Division und Exponentiation sind ebenso leicht zu bewerkstelligen wie die vorige Addition:

```
# prog4book

verzehr_in_gramm = np.array([240, 95, 135, 120, 200,
                             160, 290, 450, 500, 0])

verzehr_in_kg = verzehr_in_gramm / 1000
print(verzehr_in_kg)

# Diät, 10 % weniger essen:
reduzierter_verzehr_in_kg = verzehr_in_kg * 0.9
print(reduzierter_verzehr_in_kg)
```

```
[0.24  0.095 0.135 0.12  0.2   0.16  0.29  0.45  0.5   0.   ]
[0.216  0.0855 0.1215 0.108  0.18   0.144  0.261  0.405  0.45   0.    ]
```

```
# prog4web
print(v * 2.2)
```

```
[ 4.4    6.6   17.38   7.26  15.18   0.242 22.66  28.38 ]
```

```
# prog4web
print(v - 1.38)
```

```
[ 0.62  1.62  6.52  1.92  5.52 -1.27  8.92 11.52]
```

```
# prog4web
print(v ** 2)
print(v ** 1.5)
```

```
[4.0000e+00 9.0000e+00 6.2410e+01 1.0890e+01 4.7610e+01 1.2100e-02
```

```
 1.0609e+02 1.6641e+02]
[2.82842712e+00 5.19615242e+00 2.22044815e+01 5.99474770e+00
 1.81248172e+01 3.64828727e-02 3.30564215e+01 4.63323753e+01]
```

Wir hatten dieses Beispiel mit einer Liste lst begonnen. Wie kann man zu einer numerischen Liste einen Skalar addieren, so wie wir es mit dem Array v getan hatten?

Zu diesem Zweck kann man natürlich eine for-Schleife nutzen. Im folgenden addieren wir 2 zu den Werten dieser Liste:

```
# prog4web
lst = [2,3, 7.9, 3.3, 6.9, 0.11, 10.3, 12.9]
res = []
for val in lst:
    res.append(val + 2)

print(res)
```

```
[4, 5, 9.9, 5.3, 8.9, 2.11, 12.3, 14.9]
```

Obwohl diese Lösung funktioniert, ist sie nicht elegant und pythonisch. Zu diesem Zweck nutzt man besser eine Listenabstraktion (englisch: list comprehension) statt der umständlichen Lösung mit for-Schleife. Wir empfehlen unser Kapitel über Listenabstraktion unseres Python-Tutorials, falls jemand mit der Thematik nicht oder nicht mehr vertraut sein sollte.

```
# prog4web
res = [ val + 2 for val in lst]
print(res)
```

```
[4, 5, 9.9, 5.3, 8.9, 2.11, 12.3, 14.9]
```

Obwohl wir bereits einen Zeitvergleich zwischen NumPy und reinem Python durchgeführt hatten, wollen wir dennoch auch diese beiden Ansätze vergleichen:

```
# prog4web
v = np.random.randint(0, 100, 1000)

%timeit v + 1
```

```
939 ns ± 18.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
# prog4web
lst = list(v)

%timeit [ val + 2 for val in lst]
```

```
123 µs ± 1.35 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Arithmetische Operationen auf zwei Arrays

Falls wir ein weiteres Array statt einem Skalar benutzen, werden die Elemente von beiden Arrays komponentenweise miteinander verknüpft.

Das Array `verzehr_in_gramm` enthält den Verzehr einer fiktiven Person, die genauestens über ihren Esskonsum buchführt. Die Kalorienaufnahme pro Lebensmittel können wir dann mittels einer komponentenweisen Multiplikation mit dem Array bestimmen:

```
# prog4book
import numpy as np
kalorientabelle = np.array([30, 52, 88, 36, 86,
                            160, 375, 115, 43, 71])

verzehr_in_gramm = np.array([240, 95, 135, 120, 200,
                             160, 290, 450, 500, 0])

kalorien_aufgenommen = kalorientabelle * verzehr_in_gramm / 100
print(kalorien_aufgenommen)
```

```
[  72.    49.4  118.8   43.2  172.   256.  1087.5  517.5  215.     0. ]
```

```
print(kalorien_aufgenommen.sum())
```

```
2531.4
```

```
import numpy as np

A = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
B = np.array([[5, 4, 2], [1, 0, 2], [3, 8, 2]])

print("Addition zweier Arrays: ")
print(A + B)

print("\nMultiplikation zweier Arrays: ")
print(A * B)
```

```
Addition zweier Arrays:
[[16 16 15]
 [22 22 25]
 [34 40 35]]

Multiplikation zweier Arrays:
[[ 55  48  26]
 [ 21   0  46]
 [ 93 256  66]]
```

`A * B` im vorigen Beispiel sollte keinesfalls mit der Matrizenmultiplikation verwechselt werden. Wie bereits gesagt werden in unserem Beispiel die Arrays nur komponentenweise multipliziert!

Matrizenmultiplikation und Skalarprodukt

Definition der dot-Funktion In englischen Texten wird häufig bei dieser NumPy-Funktionalität vom "dot product" gesprochen. Mathematisch versteht man unter dem "dot product" das Skalarprodukt oder "innere Produkt" von zwei Vektoren. Seltener wird auch die Bezeichnung "Punktprodukt", also die wörtliche Übersetzung von "dot product", für das Skalarprodukt verwendet. Da das Skalar- oder Punktprodukt in der Mathematik üblicherweise nur für

den eindimensionalen Fall bei Vektoren definiert ist und die `dot`-Funktion von NumPy aber auf beliebige Funktionen angewendet werden kann, sprechen wir von `dot`-Funktion, um Verwechslungen zur Mathematik vorzubeugen.

Die Syntax der dot-Funktion sieht wie folgt aus:

`dot(a, b, out=None)`

Die Funktion `dot` liefert das `dot`-Produkt von `a` und `b` zurück.

Falls sowohl `a` als auch `b` Skalare sind oder beide eindimensionale Arrays, wird ein Skalar zurückgegeben, ansonsten ein Array.

Für eindimensionale Arrays entspricht es dem Skalarprodukt, auch inneres Produkt genannt, von Vektoren, jedoch ohne komplexe Konjugation.

Für zweidimensionale Arrays entspricht das `dot`-Produkt der Matrizenmultiplikation.

Für N Dimensionen wird das Summenprodukt über die letzte Achse von `a` und die vorletzte Achse von `b` gebildet.

$$dot(a, b)[i, j, k, m] = sum(a[i, j, :] * b[k, :, m])$$

Diesen Fall werden wir im Folgenden noch an Beispielen verdeutlichen.

Die Funktion erhebt einen `ValueError`, falls die Shape der letzten Dimension von `a` nicht die gleiche Größe wie die Shape der zweitletzten Dimension von `b` hat, d.h. es muss gelten `a.shape[-1] == b.shape[-2]`.

Beispiele zur dot-Funktion Wir beginnen mit den Fällen, in denen beide Argumente Skalare oder eindimensionale Arrays sind:

```
print(np.dot(3, 4))

x = np.array([3])
y = np.array([4])
print(x.ndim)
print(np.dot(x, y))

x = np.array([3, -2])
y = np.array([-4, 1])
print(np.dot(x, y))
```

```
12
1
12
-14
```

Im zweidimensionalen Fall realisiert die dot-Funktion die Matrizenmultiplikation. Betrachten wir dazu folgendes Beispiel

```
import numpy as np

A = np.array([[11, 12, 13, 14],
              [21, 22, 23, 24],
              [31, 32, 33, 34]])
```

```
B = np.array([[5, 4, 2],
              [1, 0, 2],
              [3, 8, 2],
              [24, 12, 57]])
print(np.dot(A, B))
```

```
[[ 442  316  870]
 [ 772  556 1500]
 [1102  796 2130]]
```

Seit Python 3.5 gibt es für das `dot`-Produkt auch einen Infix-Operator, und zwar `@`:

```
print(A @ B)
```

```
[[ 442  316  870]
 [ 772  556 1500]
 [1102  796 2130]]
```

Damit die Matrizenmultiplikation (also dot) für zwei Matrizen A und B im zweidimensionalen Fall funktionieren kann, muss gelten:

`A.shape[-1] == B.shape[-2]`

```
# es muss gelten:
print(A.shape[-1] == B.shape[-2])
```

```
True
```

Aus dem vorigen Beispiel können wir lernen, dass die Anzahl der Spalten des ersten zweidimensionalen Arrays gleich der Anzahl der Zeilen des zweiten zweidimensionalen Arrays sein muss.

Das dot-Produkt im 3-dimensionalen Fall Es wird ziemlich verzwickt, wenn wir 3-dimensionale Arrays als Argumente von dot benutzen.

Im ersten Beispiel benutzen wir zwei symmetrische 3-dimensionale Arrays:

```
import numpy as np
X = np.array( [[[3, 1, 2],
                [4, 2, 2],
                [2, 4, 1]],

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

               [[2, 2, 1],
                [3, 1, 3],
                [3, 2, 3]]])

Y = np.array( [[[2, 3, 1],
                [2, 2, 4],
                [3, 4, 4]],

               [[1, 4, 1],
```

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

               [[1, 2, 3],
                [4, 1, 1],
                [3, 1, 4]]])


R = np.dot(X, Y)

print("Die Größen:")
print(X.shape)
print(Y.shape)
print(R.shape)

print("\nDas Ergebnis der Matrizenmultiplikation:")
print(R)
```

```
Die Größen:
(3, 3, 3)
(3, 3, 3)
(3, 3, 3, 3)

Das Ergebnis der Matrizenmultiplikation:
[[[[14 19 15]
   [15 15  9]
   [13  9 18]]

  [[18 24 20]
   [20 20 12]
   [18 12 22]]

  [[15 18 22]
   [22 13 12]
   [21  9 14]]]


 [[[16 21 19]
   [19 16 11]
   [17 10 19]]

  [[25 32 32]
   [32 23 18]
   [29 15 28]]

  [[13 18 12]
   [12 18  8]
   [11 10 17]]]


 [[[11 14 14]
   [14 11  8]
   [13  7 12]]

  [[17 23 19]
   [19 16 11]
   [16 10 22]]
```

```
  [[19 25 23]
   [23 17 13]
   [20 11 23]]]]
```

Nun betrachten wir das Produkt von zwei nicht-symmetrischen dreidimensionalen Arrays:

```
import numpy as np
X = np.array( [ [[11, 12, 13], [14, 15, 16], [17, 18, 19]],
                [[21, 22, 23], [24, 25, 26], [27, 28, 29]],
                [[31, 32, 33], [34, 34, 35], [36, 37, 39]]])

Y = np.array( [[[0, 0, 0],
                [0, 1, 0],
                [0, 0, 1]]])

R = np.dot(X, Y)

print("Die Gestalten und die Dimensionen:")
print("X.shape: ", X.shape, "   X.ndim: ", X.ndim)
print("Y.shape: ", Y.shape, "   Y.ndim: ", Y.ndim)
print("R.shape: ",     R.shape, "R.ndim: ", R.ndim)
print("\nDas Ergebnis-Array R:\n", R)
```

```
Die Gestalten und die Dimensionen:
X.shape:  (3, 3, 3)    X.ndim:  3
Y.shape:  (1, 3, 3)    Y.ndim:  3
R.shape:  (3, 3, 1, 3) R.ndim:  4

Das Ergebnis-Array R:
 [[[[ 0 12 13]]

  [[ 0 15 16]]

  [[ 0 18 19]]]


 [[[ 0 22 23]]

  [[ 0 25 26]]

  [[ 0 28 29]]]


 [[[ 0 32 33]]

  [[ 0 34 35]]

  [[ 0 37 39]]]]
```

Die Funktion `squeeze` erlaubt uns, Dimensionen zu schrumpfen, indem eindimensionale Einträge aus der `shape` eines Arrays entfernt werden. Wir zeigen dies an einem einfachen Beispiel:

```
x = np.array([3, 5, 7]).reshape(1, 3, 1)
print(x)
print("x.squeeze()")
```

```
print(x.squeeze())
print("x.squeeze(axis=0)")
print(x.squeeze(axis=0))
print("x.squeeze(axis=2)")
print(x.squeeze(axis=2))
```

```
[[[3]
  [5]
  [7]]]
x.squeeze()
[3 5 7]
x.squeeze(axis=0)
[[3]
 [5]
 [7]]
x.squeeze(axis=2)
[[3 5 7]]
```

Nun wenden wir `squeeze` auf das Array `R` an, um die Shape von (3, 3, 1, 3) auf (3, 3, 3) zu schrumpfen:

```
print(R.shape)
R = np.squeeze(R, axis=2)
print(R.shape, R)
```

```
(3, 3, 1, 3)
(3, 3, 3) [[[ 0 12 13]
  [ 0 15 16]
  [ 0 18 19]]

 [[ 0 22 23]
  [ 0 25 26]
  [ 0 28 29]]

 [[ 0 32 33]
  [ 0 34 35]
  [ 0 37 39]]]
```

Um zu zeigen, wie das dot-Produkt im dreidimensionalen Fall funktioniert, werden wir jedoch nun zwei nicht-symmetrische dreidimensionale Arrays im folgenden Beispiel benutzen:

```
import numpy as np
X = np.array(
    [[[3, 1, 2],
      [4, 2, 2]],

     [[-1, 0, 1],
      [1, -1, -2]],

     [[3, 2, 2],
      [4, 4, 3]],

     [[2, 2, 1],
      [3, 1, 3]]])

Y = np.array(
```

```
    [[[2, 3, 1, 2, 1],
      [2, 2, 2, 0, 0],
      [3, 4, 0, 1, -1]],

     [[1, 4, 3, 2, 2],
      [4, 1, 1, 4, -3],
      [4, 1, 0, 3, 0]]])


R = np.dot(X, Y)

print("X.shape: ", X.shape, "   X.ndim: ", X.ndim)
print("Y.shape: ", Y.shape, "   Y.ndim: ", Y.ndim)
print("R.shape: ",     R.shape, "R.ndim: ", R.ndim)

print("\nDas Ergebnis-Array R:\n")
print(R)
```

```
X.shape:  (4, 2, 3)    X.ndim:  3
Y.shape:  (2, 3, 5)    Y.ndim:  3
R.shape:  (4, 2, 2, 5) R.ndim:  4

Das Ergebnis-Array R:

[[[[ 14  19   5   8   1]
   [ 15  15  10  16   3]]

  [[ 18  24   8  10   2]
   [ 20  20  14  22   2]]]


 [[[  1   1  -1  -1  -2]
   [  3  -3  -3   1  -2]]

  [[ -6  -7  -1   0   3]
   [-11   1   2  -8   5]]]


 [[[ 16  21   7   8   1]
   [ 19  16  11  20   0]]

  [[ 25  32  12  11   1]
   [ 32  23  16  33  -4]]]


 [[[ 11  14   6   5   1]
   [ 14  11   8  15  -2]]

  [[ 17  23   5   9   0]
   [ 19  16  10  19   3]]]]
```

Schauen wir uns nun die folgenden Summen-Produkte an:

```
i = 0
for j in range(X.shape[1]):
    for k in range(Y.shape[0]):
        for m in range(Y.shape[2]):
```

```
            fmt = "    sum(X[{}, {}, :] * Y[{}, :, {}] :  {}"
            arguments = (i, j, k, m, sum(X[i, j, :] * Y[k, :, m]))
            print(fmt.format(*arguments))
```

```
    sum(X[0, 0, :] * Y[0, :, 0] :  14
    sum(X[0, 0, :] * Y[0, :, 1] :  19
    sum(X[0, 0, :] * Y[0, :, 2] :  5
    sum(X[0, 0, :] * Y[0, :, 3] :  8
    sum(X[0, 0, :] * Y[0, :, 4] :  1
    sum(X[0, 0, :] * Y[1, :, 0] :  15
    sum(X[0, 0, :] * Y[1, :, 1] :  15
    sum(X[0, 0, :] * Y[1, :, 2] :  10
    sum(X[0, 0, :] * Y[1, :, 3] :  16
    sum(X[0, 0, :] * Y[1, :, 4] :  3
    sum(X[0, 1, :] * Y[0, :, 0] :  18
    sum(X[0, 1, :] * Y[0, :, 1] :  24
    sum(X[0, 1, :] * Y[0, :, 2] :  8
    sum(X[0, 1, :] * Y[0, :, 3] :  10
    sum(X[0, 1, :] * Y[0, :, 4] :  2
    sum(X[0, 1, :] * Y[1, :, 0] :  20
    sum(X[0, 1, :] * Y[1, :, 1] :  20
    sum(X[0, 1, :] * Y[1, :, 2] :  14
    sum(X[0, 1, :] * Y[1, :, 3] :  22
    sum(X[0, 1, :] * Y[1, :, 4] :  2
```

Hoffentlich ist Ihnen aufgefallen, dass die Werte, die wir erzeugt haben, den Elementen von `R[0]` entsprechen:

```
print(R[0])
```

```
[[[14 19  5  8  1]
  [15 15 10 16  3]]

 [[18 24  8 10  2]
  [20 20 14 22  2]]]
```

Dies bedeutet, dass wir das Array R auch über die Summen-Produkte hätten erzeugen können. Um dies zu "beweisen", werden wir im folgenden Beispiel ein Array R2 unter Benutzung der Summen-Produkte erzeugen. Anschließend prüfen wir, ob R2 gleich R ist.

```
def sum_prod(X, Y):
    """ sum product for 3-dimensional arrays """
    res_shape = X.shape[:-1] + Y.shape[:-2] + (Y.shape[-1],)
    R = np.zeros(res_shape, dtype=type(X))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            for k in range(Y.shape[0]):
                for m in range(Y.shape[2]):
                    R[i, j, k, m] = sum(X[i, j, :] * Y[k, :, m])
    return R

print( np.array_equal(np.dot(X, Y), sum_prod(X, Y)) )
```

```
True
```

Es gilt also, was wir eingangs behauptet haben:

```
dot(X, Y)[i,j,k,m] = sum(X[i,j,:] * Y[k,:,m])
```

Vergleichsoperatoren

Wir kennen bereits Vergleichsoperatoren in Python, die wir auf Integer, Floats oder Strings angewendet haben. Sie liefern `True` oder `False` zurück. Vergleichen wir zwei Arrays miteinander, erhalten wir keinen "einfachen" Booleschen Wert zurück. Die Vergleiche werden elementweise durchgeführt. Dies bewirkt, dass wir ein Boolesches Array als Rückgabewert erhalten:

```
import numpy as np

A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])
B = np.array([ [11, 102, 13], [201, 22, 203], [31, 32, 303] ])

A == B
```

```
array([[ True, False,  True],
       [False,  True, False],
       [ True,  True, False]])
```

Man kann aber auch Arrays vollständig auf Gleichheit überprüfen. Dazu benutzen wir die Funktion `array_equal`, die `True` zurückliefert, falls zwei Arrays die gleiche Shape haben und alle Elemente gleich sind. Ansonsten wird `False` zurückgeliefert.

```
print(np.array_equal(A, B))
print(np.array_equal(A, A))
```

```
False
True
```

Logische Operatoren

Wir können Arrays auch komponentenweise auf ein logisches 'oder' und ein logisches 'und' vergleichen. Dazu gibt es die Funktionen `logical_or` und `logical_and`.

```
a = np.array([ [True, True], [False, False]])
b = np.array([ [True, False], [True, False]])

print(np.logical_or(a, b))
print(np.logical_and(a, b))
```

```
[[ True  True]
 [ True False]]
[[ True False]
 [False False]]
```

Broadcasting

Bisher gingen wir davon aus, dass Arrays die gleiche Shape haben müssen, damit wir numerische Operatoren auf diese anwenden können. Unter dem Namen "Broadcasting"

stellt NumPy einen mächtigen Mechanismus zur Verfügung, der es erlaubt, arithmetische Operatoren auch auf Arrays mit unterschiedlicher Shape anzuwenden. Um die entsprechende Operation durchzuführen, wird nun das "kleinere" Array entweder in eine "passende Form" transformiert oder mehrmals auf das "größere" angewendet. In anderen Worten: Unter bestimmten Bedingungen erfolgt ein "Broadcast" des kleineren Arrays, bis es die gleiche Shape wie das größere hat.

Mit Hilfe des Broadcasting können wir Schleifen in unseren Python-Programmen vermeiden. Die Schleifenbildung erfolgt dann implizit in der NumPy-Implementierung, d.h. in C. Dadurch vermeiden wir außerdem unnötige Kopien von unseren Daten.

Prinzipiell gibt es drei verschiedene Formen von Broadcasting:

- in horizontaler Richtung
- in vertikaler Richtung
- in horizontaler und vertikaler Richtung

Broadcasting bei zwei Arrays in NumPy erfolgt nach folgenden Regeln:

Regel 1

Falls die beiden Arrays in ihrer Anzahl von Dimensionen sich unterscheiden, wird die Shape des Arrays, das weniger Dimensionen hat, mit Einsen von der linken Seite aus aufgefüllt.

Regel 2

Falls die Shapes von zwei Arrays an einer Shape-Position nicht übereinstimmen, wird die Shape desjenigen Arrays angepasst, die eine 1 enthält. Der Wert wird dann auf den Wert des anderen Arrays erhöht.

Regel 3

Falls in irgendeiner Dimension die Größen unterschiedlich sind und keine von beiden gleich 1 ist, wird ein Fehler erhoben.

Im Folgenden werden wir klarer sehen, was die Anwendung dieser Regeln bewirken.

Einen besonders einfachen Fall von Broadcasting haben wir schon kennengelernt. Der einfachste Fall ist die Skalarenmultiplikation:

```
import numpy as np

v = np.array([3, 5, 1])
x = 4
print(v * x)
```

```
[12 20  4]
```

Statt einem Skalar für x hätte man auch den Vektor `np.array([4, 4, 4])` nehmen können und hätte das gleiche Ergebnis erhalten:

```
import numpy as np

v = np.array([3, 5, 1])
x = np.array([4, 4, 4])
print(v * x)
```

```
[12 20  4]
```

Normalerweise denkt man aber nicht an diesen Fall, wenn man von Broadcasting spricht.

Zeilenweises Broadcasting Betrachten wir die beiden Arrays `A` und `B` und ihre Shapes:

```
import numpy as np

A = np.array([[11, 12, 13],
              [21, 22, 23],
              [31, 32, 33] ])
B = np.array([1, 2, 3])

print(A.shape)
print(B.shape)
```

```
(3, 3)
(3,)
```

```
print("Multiplikation mit Broadcasting: ")
print(A * B)
print("... und nun die Addition mit Broadcasting: ")
print(A + B)
```

```
Multiplikation mit Broadcasting:
[[11 24 39]
 [21 44 69]
 [31 64 99]]
... und nun die Addition mit Broadcasting:
[[12 14 16]
 [22 24 26]
 [32 34 36]]
```

Das folgende Diagramm illustriert die Arbeitsweise von Broadcasting:

B wird benutzt, als wäre es wie folgt aufgebaut:

```
B = np.array([1, 2, 3])
print("Die Shape von B: ", B.shape)
# Anwendung der Regel 1:
B = B[np.newaxis, :]
print("Shape nach Anwendung der ersten Regel: ", B.shape)
B = np.tile(B, (3, 1))
print("Shape nach Anwendung der zweiten Regel: ", B.shape)
print()
print(B)
```

```
Die Shape von B:  (3,)
Shape nach Anwendung der ersten Regel:  (1, 3)
Shape nach Anwendung der zweiten Regel:  (3, 3)

[[1 2 3]
 [1 2 3]
 [1 2 3]]
```

“Zeilenweise” bedeutet also, dass wir ein eindimensionales Array als die zu broadcastende Zeile betrachten.

Broadcasting funktioniert auch bei höheren Dimensionen, solange sich die beiden ersten Regeln erfolgreich anwenden lassen. Wir demonstrieren dies im folgenden Beispiel:

```
X = np.array(
    [[[2, 3, 1, 2, 1],
      [2, 2, 2, 0, 0],
      [3, 4, 0, 1, -1]],

     [[1, 4, 3, 2, 2],
      [4, 1, 1, 4, -3],
      [4, 1, 0, 3, 0]]])

print(Y.shape)

X = np.array([1, 2, 3, 4, 5])

print(X + Y)
```

```
(2, 3, 5)
[[[3 5 4 6 6]
  [3 4 5 4 5]
  [4 6 3 5 4]]

 [[2 6 6 6 7]
  [5 3 4 8 2]
  [5 3 3 7 5]]]
```

Auch in diesem Falle wollen wir uns anschauen, was man machen müsste, um das Broadcasting nach den Regeln zu simulieren:

```
print(Y.shape)
X = X[np.newaxis, np.newaxis, :]
print("Shape nach Anwendung der ersten Regel: ", X.shape)
X = np.tile(X, (2, 3, 1))
print("Shape nach Anwendung der zweiten Regel: ", X.shape)
print()

print(X + Y)
```

```
(2, 3, 5)
Shape nach Anwendung der ersten Regel:  (1, 1, 5)
Shape nach Anwendung der zweiten Regel:  (2, 3, 5)

[[[3 5 4 6 6]
  [3 4 5 4 5]
  [4 6 3 5 4]]

 [[2 6 6 6 7]
  [5 3 4 8 2]
  [5 3 3 7 5]]]
```

Spaltenweises Broadcasting: In diesem Fall haben wir wieder ein eindimensionales Array, betrachten es nun aber als Spaltenvektor des Broadcast-Arrays.

Für dieses Beispiel müssen wir wissen, wie man einen Zeilenvektor in einen Spaltenvektor wandelt. Hierzu gibt es zwei Möglichkeiten. Mittels `reshape`:

```
B = np.array([1, 2, 3])
print(B.reshape((B.shape[0], 1)))
```

```
[[1]
 [2]
 [3]]
```

Alternativ können wir auch `newaxis` verwenden:

```
B = np.array([1, 2, 3])
print(B[:, np.newaxis])
```

```
[[1]
 [2]
 [3]]
```

Nun können wir die Multiplikation mittels Broadcasting durchführen:

```
import numpy as np

A = np.array([[11, 12, 13],
              [21, 22, 23],
              [31, 32, 33] ])

A * B[:, np.newaxis]
```

```
array([[11, 12, 13],
       [42, 44, 46],
       [93, 96, 99]])
```

B wird benutzt als wäre es wie folgt aufgebaut:

```
B = np.array([1, 2, 3])
B = B.reshape((B.shape[0], 1))
print("Shape nach Anwendung der ersten Regel: ", B.shape)
B = np.tile(B, (1, 3))
print("Shape nach Anwendung der zweiten Regel: ", B.shape)
print()

print(B)
```

```
Shape nach Anwendung der ersten Regel:  (3, 1)
Shape nach Anwendung der zweiten Regel:  (3, 3)

[[1 1 1]
 [2 2 2]
 [3 3 3]]
```

Broadcasting von zwei eindimensionalen Arrays Nun betrachten wir den Fall, dass wir zwei eindimensionale Arrays verknüpfen möchten. Den ersten wollen wir als Spaltenvektor betrachten und den zweiten als Zeilenvektor. Im Prinzip kombinieren wir nun die Verfahren der beiden vorigen Fälle, also spaltenweises und zeilenweises Broadcasting:

```
A = np.array([10, 20, 30])
B = np.array([1, 2, 3])

# wir richten A auf:
A = A.reshape(A.shape[0], 1)
print(A)
# nun können wir das Broadcasting durchführen:

print(A * B)
```

```
[[10]
 [20]
 [30]]
[[10 20 30]
 [20 40 60]
 [30 60 90]]
```

Distanzmatrix

In der Mathematik, der Informatik und insbesondere in der Graph-Theorie, versteht man unter der Distanzmatrix ein zweidimensionales Array, das die "Entfernungen" zwischen den Elementen einer Menge paarweise beinhaltet. Die Größe dieses zweidimensionalen Arrays ist n x n, falls die Menge aus n Elementen besteht.

Ein praktisches Beispiel einer Distanzmatrix ist eine Matrix mit den Entfernungen zwischen geografischen Lokationen, in unserem Beispiel europäische Städte:

```
cities = ["Barcelona", "Berlin", "Brüssel", "Bukarest",
          "Budapest", "Kopenhagen", "Dublin", "Hamburg",
          "Istanbul", "Kiew", "London", "Madrid",
          "Mailand", "Moskau", "München", "Paris", "Prag",
          "Rome", "Sankt Petersburg", "Stockholm", "Wien",
          "Warschau"]

dist2barcelona = [0,  1498, 1063, 1968,
                  1498, 1758, 1469, 1472, 2230,
                  2391, 1138, 505, 725, 3007, 1055,
                  833, 1354, 857, 2813,
                  2277, 1347, 1862]

dists =  np.array(dist2barcelona[:12])
print(dists)
print(np.abs(dists - dists[:, np.newaxis]))
```

```
[   0 1498 1063 1968 1498 1758 1469 1472 2230 2391 1138  505]
[[   0 1498 1063 1968 1498 1758 1469 1472 2230 2391 1138  505]
 [1498    0  435  470    0  260   29   26  732  893  360  993]
 [1063  435    0  905  435  695  406  409 1167 1328   75  558]
 [1968  470  905    0  470  210  499  496  262  423  830 1463]
```

```
[1498    0  435  470    0  260   29   26  732  893  360  993]
[1758  260  695  210  260    0  289  286  472  633  620 1253]
[1469   29  406  499   29  289    0    3  761  922  331  964]
[1472   26  409  496   26  286    3    0  758  919  334  967]
[2230  732 1167  262  732  472  761  758    0  161 1092 1725]
[2391  893 1328  423  893  633  922  919  161    0 1253 1886]
[1138  360   75  830  360  620  331  334 1092 1253    0  633]
[ 505  993  558 1463  993 1253  964  967 1725 1886  633    0]]
```

Ufuncs

Bisher haben wir diverse Operatoren wie Addition, Subtraktion, Multiplikation und so weiter für Arrays kennengelernt. Allen sind bestimmte Features gemeinsam, wie Broadcasting und erzwungene Typumwandlung (englisch: type coercion). Allen Operationen ist auch die elementweise Anwendung gemeinsam.

Zu den bereits kennengelernten Infix-Operatoren wie $+$, $-$, $*$ und so weiter gibt es auch Funktionen wie `add`, `subtract`, `multiply`, `divide`, `remainder`, `power` und weitere. Aber es gibt auch Funktionen, für die es keine Infix-Variante gibt, da es sich um Funktionen handelt, die nur ein Argument erwarten, also unäre Operatoren: `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctanh`, `cos`, `cosh`, `tan`, `tanh`, `log10`, `sin`, `sinh`, `sqrt`, `absolute`, `fabs`, `floor`, `ceil`, `fmod`, `exp`, `log`, `conjugate`, `maximum`, `minimum`.

Auch für die Infix-Vergleichsoperatoren $<$, $<=$, $==$, $>$, $>=$, $!=$ und so weiter gibt es entsprechende funktionale Varianten: `greater`, `greater_equal`, `equal`, `less`, `less_equal`, `not_equal`, `logical_or`, `logical_xor`, `logical_not`, `logical_and`, `bitwise_or`, `bitwise_xor`, `bitwise_not`, `bitwise_and`, `rshift`, `lshift`

Diese Funktionen bezeichnet man als Ufuncs, auch als universelle Funktionen bekannt.

Anwendung von Ufuncs Die funktionalen und Infix-Varianten sind äquivalent.

```
import numpy as np
from numpy import add

x = np.array([3, 5, -1, 0])
y = np.array([1, -2, 0, 3])

print(x + y)
# äquivalente Möglicheit
print(add(x, y))
print("Typ der 'add'-Funktion:", type(add))
```

```
[ 4  3 -1  3]
[ 4  3 -1  3]
Typ der 'add'-Funktion: <class 'numpy.ufunc'>
```

Die Ufuncs lassen sich auf beliebige Python-Sequenzen anwenden, sofern sie aus numerischen Datentypen bestehen und bezüglich ihrer Längen und Shapes übereinstimmen:

```
import numpy as np

x = [2, 5, 6.7]
y = [4, 6, 7.8]
```

```
print(np.add(x, y))

print(np.add(tuple(x), y))
print(np.add(x, range(3)))
```

```
[ 6.  11.  14.5]
[ 6.  11.  14.5]
[2.  6.  8.7]
```

Die unären Ufunc-Funktionen können sowohl auf NumPy-Arrays als auch auf andere Python-Sequenzen angewendet werden. Der Rückgabewert ist aber in allen Fällen ein `numpy.ndarray`-Wert:

```
import numpy as np

x = [2, 5, 6.7]
v = np.array(x)

res = np.sin(x)
print(res, type(res))
res = np.sin(v)
print(res, type(res))
res = np.sin(range(1, 100, 7))
print(res, type(res))
```

```
[ 0.90929743 -0.95892427  0.40484992] <class 'numpy.ndarray'>
[ 0.90929743 -0.95892427  0.40484992] <class 'numpy.ndarray'>
[ 0.84147098  0.98935825  0.65028784 -0.00885131 -0.66363388 -0.99177885
 -0.83177474 -0.26237485  0.43616476  0.92002604  0.95105465  0.51397846
 -0.17607562 -0.77946607 -0.99920683] <class 'numpy.ndarray'>
```

Sie können auch auf Instanzen der `int`- und `float`-Klassen aufgerufen werden. Rückgabewerte sind in diesen Fällen `numpy.int64`- bzw. `numpy.float64`-Typen:

```
res = np.sin(2)
print(res, type(res))

res = np.ceil(2.3)
print(res, type(res))
```

```
0.9092974268256817 <class 'numpy.float64'>
3.0 <class 'numpy.float64'>
```

Parameter für Rückgabewerte bei Ufuncs Eine weitere Besonderheit der Ufuncs besteht darin, dass ihnen auch ein Parameter für das Rückgabeobjekt übergeben werden kann. Betrachten wir dazu das folgende Beispiel:

```
import numpy as np

x = np.array([25.6, 29.3, 30.9])
x = x * 1.8

# oder äquivalent dazu in Ufunc-Schreibweise
```

```
x = np.array([25.6, 29.3, 30.9])
x = np.multiply(x, 1.8)
print(x)
```

```
[46.08 52.74 55.62]
```

In den obigen Fällen wurde zuerst ein neues `ndarray`-Array mit dem Ergebnis erzeugt und dann wurde dieses Array der Variablen `x` zugewiesen. Das alte Array wird danach nicht mehr benötigt und muss gelöscht werden.

Indem wir die Variable `x` als Rückgabeparameter spezifizieren, wird die Operation in-place durchgeführt, also effizienter:

```
import numpy as np

x = np.array([25.6, 29.3, 30.9])
np.multiply(x, 1.8, x)
print(x)
```

```
[46.08 52.74 55.62]
```

Die in-place-Variante mit Rückgabeparameter ist im Allgemeinen deutlich schneller, aber nicht immer, wie wir im Folgenden sehen können:

```
import timeit

setup = 'import numpy as np; A = np.random.random((100, 100))'
for i in range(5):
    time1 = timeit.timeit("A = np.multiply(A, 1.0234)",
                          setup=setup,
                          number=10)
    time2 = timeit.timeit("np.multiply(A, 1.0234, A)",
                          setup=setup,
                          number=10)
    print(time1, time2, time1/time2)
```

```
0.00010651699994923547 5.7821998780127615e-05 1.8421535435721301
0.0003842940022877883 5.538400000659749e-05 6.938718804023006
0.00034500000037951395 9.413500083610415e-05 3.6649492464570526
9.594500079401769e-05 0.00010985400149365887 0.8733864901548984
0.0004484130004129838 6.163700163597241e-05 7.275061870486613
```

accumulate Auf die Ufunc-Funtionen kann die `accumulate`-Methode angewendet werden.

Die `accumulate`-Funktion bewirkt die Anwendung des Operators auf alle Elemente des Arrays und der darauffolgenden Akkumulation der Ergebnisse.

Wendet man `accumulate` auf die `add`-Funktion an, erhält man eine zu `np.cumsum` äquivalente Funktionalität, wie wir im folgenden Beispiel demonstrieren.

```
import numpy as np

x = np.arange(1, 6)
print(x)
```

```
print(np.add.accumulate(x))
print(np.cumsum(x))
```

```
[1 2 3 4 5]
[ 1  3  6 10 15]
[ 1  3  6 10 15]
```

Den mehrdimensionalen Fall und die Benutzung des optionalen Parameters `axis` verdeutlichen wir mit folgendem selbsterklärenden Beispiel:

```
import numpy as np

x = np.arange(24).reshape((6, 4))
print(x)
print("Akkumulation über Zeilen, d.h. axis = 0:")
print(np.add.accumulate(x))
print("Akkumulation über Zeilen, d.h. axis = 0:")
print(np.add.accumulate(x, axis=0))
print("Akkumulation über Spalten, d.h. axis = 1:")
print(np.add.accumulate(x, axis=1))
print("Akkumulation über Zeilen und Spalten:")
print(np.add.accumulate(np.add.accumulate(x, axis=0), axis=1))
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
Akkumulation über Zeilen, d.h. axis = 0:
[[ 0  1  2  3]
 [ 4  6  8 10]
 [12 15 18 21]
 [24 28 32 36]
 [40 45 50 55]
 [60 66 72 78]]
Akkumulation über Zeilen, d.h. axis = 0:
[[ 0  1  2  3]
 [ 4  6  8 10]
 [12 15 18 21]
 [24 28 32 36]
 [40 45 50 55]
 [60 66 72 78]]
Akkumulation über Spalten, d.h. axis = 1:
[[ 0  1  3  6]
 [ 4  9 15 22]
 [ 8 17 27 38]
 [12 25 39 54]
 [16 33 51 70]
 [20 41 63 86]]
Akkumulation über Zeilen und Spalten:
[[  0   1   3   6]
 [  4  10  18  28]
 [ 12  27  45  66]
 [ 24  52  84 120]
 [ 40  85 135 190]
 [ 60 126 198 276]]
```

reduce Die Arbeitsweise von `reduce` im eindimensionalen Fall, kann wie folgt beschreiben werden:

Nehmen wir an, `op` ist eine Funktion wie `numpy.add` oder `numpy.multiply`. Wendet man `op.reduce` auf ein Objekt `x` an, welches eine numerische Python-Sequenz oder ein eindimensionales Array sein kann, so liefert sie einen einzelnen Wert zurück. Die Berechnung geschieht wie folgt: Falls `x` leer ist, wird `0.0` zurückgeliefert. Besitzt `x` nur ein Element, wird dieses als Ergebnis von `op.reduce(x)` zurückgeliefert. Bei mehr als einem Element in `x` wird `op` zuerst auf die ersten beiden Elemente von `x` angewendet. Gibt es noch weitere Elemente, wird `op` auf dieses Ergebnis und auf das folgende Element von x angewendet. Dies wird solange fortgesetzt, bis es keine weiteren Elemente mehr gibt, und das letzte Ergebnis wird dann als Gesamtergebnis zurückgeliefert:

```
import numpy as np

x = np.arange(1, 5)

print(np.add.reduce(x)) # äquivalent zu ```sum```
print(np.multiply.reduce(x)) # äquivalent zu ```factorial```
```

```
10
24
```

Im mehrdimensionalen Fall bewirkt die Anwendung von `reduce` die Reduktion um eine Dimension. Eine Reduktion über alle Dimensionen erhält man, indem man `axis` auf `None` setzt:

```
import numpy as np

x = np.arange(12).reshape((3, 4))
print(x)
print("Reduce über Zeilen, d.h. axis = 0:")
print(np.add.reduce(x))
print("Reduce über Spalten, d.h. axis = 1:")
print(np.add.reduce(x, axis=1))
print("Reduce über Zeilen und Spalten:")
print(np.add.reduce(np.add.reduce(x)))
print("Letzteres geht einfacher mit axis = None:")
print(np.add.reduce(x, axis=None))
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Reduce über Zeilen, d.h. axis = 0:
[12 15 18 21]
Reduce über Spalten, d.h. axis = 1:
[ 6 22 38]
Reduce über Zeilen und Spalten:
66
Letzteres geht einfacher mit axis = None:
66
```

outer `op.outer(A, B)` bewirkt, dass `op` auf alle Paare `(a, b)` mit `a` in `A` und `b` in `B` angewendet wird.

In der Mathematik ist dieses spezielles Produkt zweier Vektoren auch als das “dyadische Produkt” oder “tensorielle Produkt” bekannt. Das Ergebnis ist eine Matrix.

```
import numpy as np

np.multiply.outer([1, 2, 3, 4], [11, 22, 33])
```

```
array([[ 11,  22,  33],
       [ 22,  44,  66],
       [ 33,  66,  99],
       [ 44,  88, 132]])
```

at `at(a, indices, b=None)` führt eine ungepufferte in-place-Operation auf den durch `indices` spezifizierten Indizes von `a` durch. Für die Addition ist dies beispielsweise äquivalent mit `a[indices] += b`, außer dass die Elemente, die mehrfach indiziert sind, akkumuliert werden.

```
import numpy as np

a = np.array([3, 5, 12, 5])
b = np.array([3, 11])
np.add.at(a, [0, 2], b)
print(a)

# äquivalent mit
a = np.array([3, 5, 12, 5])
a[[0, 2]] += b
print(a)

# nun mit mehrfachen Indizes:
a = np.array([3, 5, 12, 5])
np.add.at(a, [0, 0], b)
print(a)

a[[0, 0]] += b
print(a)
```

```
[ 6  5 23  5]
[ 6  5 23  5]
[17  5 12  5]
[28  5 12  5]
```

```
# prog4book
B = np.array([1, 2, 3])
print(B.shape)
B = B[np.newaxis, :]
print(B.shape)
B = np.concatenate((B, B, B)).transpose()
print(B.shape)
B = B[:, np.newaxis]
print(B.shape)
print(B)
```

```
(3,)
```

```
(1, 3)
(3, 3)
(3, 1, 3)
[[[1 1 1]]

 [[2 2 2]]

 [[3 3 3]]]
```

```
# prog4book
A = np.random.randint(-10, 10, (3, 4))
print(A)

print("Zeilenminima: ")
print(np.min(A, axis=1))

print("Spaltenminima: ")
print(np.min(A, axis=0))

print("Absolutes Minimum: ")
print(np.min(A))
```

```
[[ 1  6 -9  9]
 [ 1  0  1  9]
 [-6  8 -4 -7]]
Zeilenminima:
[-9  0 -7]
Spaltenminima:
[-6  0 -9 -7]
Absolutes Minimum:
-9
```

```
# prog4book

# prog4book
A = np.random.randint(-10, 10, (3, 4))
print(A)

print("Zeilenminima: ")
print(np.min(A, axis=1))

print("Spaltenminima: ")
print(np.min(A, axis=0))

print("Absolutes Minimum: ")
print(np.min(A))
```

```
[[ 6 -1  5 -2]
 [ 0  2  9  6]
 [-8 -9  7 -4]]
Zeilenminima:
[-2  0 -9]
Spaltenminima:
[-8 -9  5 -4]
Absolutes Minimum:
-9
```

6 Numpy Dimensionen Anpassen

6.0.1 Dimensionsänderungen

Bisher lernten wir, wie man Arrays erzeugt und wie wir numerische Operationen auf NumPy Arrays anwenden können. Wenn wir mit NumPy programmieren, kommen wir früher oder später zu dem Punkt, wo wir Funktionen benötigen, um die Gestalt (shape) und die Dimension von Arrays zu manipulieren. Die dazu nötigen Funktionalitäten lernen wir in diesem Kapitel kennen. Wir werden auch lernen, wie man Arrays zusammenhängt bzw. konkateniert. Weiterhin werden wir die Möglichkeiten demonstrieren, wie man weitere Dimensionen an existierende Arrays anhängen kann und wie man mehrere Arrays horizontal und vertikal "stapeln" (stack) kann. Dieses Kapitel beenden wir, indem wir zeigen, wie man neue Arrays durch Wiederholungen aus existierenden Arrays erzeugen kann.

Das Bild zeigt einen Tesserakt, den man auch als vierdimensionalen Hyperwürfel bezeichnet. Ein Tesserakt kann man als die Übertragung des Konzeptes eines dreimensionalen Würfels in den vierdimensionalen Raum sehen. Ein Tesserakt verhält sich zum Würfel wie ein Würfel zum Quadrat.

Reduktion und Reshape von Arrays

Es gibt mehrere Methoden, um ein multidimensionales Array zu reduzieren:

- `flatten`
- `ravel`
- `reshape`

flatten `flatten` ist eine ndarray-Methode mit einem optionalen Parameter `order`, der die Werte `C`, `F` und `A` annehmen kann. Der Default-Wert von `order` ist `C`. `C` steht dafür, dass im C-Stil in der Zeilen-Haupt-Ordnung linearisiert bzw. flach gemacht wird, d.h. der am weitesten rechts liegende Index "ändert sich am schnellsten". In anderen Worten: Der Zeilenindex variiert in der Zeilen-Haupt-Ordnung am langsamsten und am Spaltenindex am schnellsten, sodass `a[0, 1]` auf `a[0, 0]` folgt. `F` steht für "Fortran Spalten-Haupt-Ordnung". `A` steht für den Erhalt der "C/Fortran-Anordnung".

```
import numpy as np

A = np.array([[[ 0,  1],
               [ 2,  3],
               [ 4,  5],
               [ 6,  7]],
              [[ 8,  9],
               [10, 11],
               [12, 13],
               [14, 15]],
              [[16, 17],
```

```
                [18, 19],
                [20, 21],
                [22, 23]]])

Flattened_X = A.flatten()
print(Flattened_X)

print(A.flatten(order="C"))
print(A.flatten(order="F"))
print(A.flatten(order="A"))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[ 0  8 16  2 10 18  4 12 20  6 14 22  1  9 17  3 11 19  5 13 21  7 15 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

ravel Die Reihenfolge der Elemente, die durch ravel() zurückgeliefert wird, ist standardmäßig im "C-Stil".

`ravel(X, order='C')`

`ravel` erzeugt ein linearisiertes, also eindimensionales, Array . Eine Kopie wird nur bei Notwendigkeit erstellt.

Der optionale Parameter `order` kann die Werte `C`, `F`, `A` oder `K` annehmen.

`C`: C-Stil Reihenfolge, wobei sich der letzte Achsenindex am schnellsten ändert, zurück zum ersten Achsenindex, der sich am langsamsten ändert. `C` ist der Default-Wert.

`F`: Fortran-Stil Indexreihenfolge, wobei sich der erste Index am schnellsten ändert und der letzte Index am langsamsten.

`A`: Fortran-Stil Indexreihenfolge, wenn das Array 'a' im Speicher als Fortran vorliegt, andernfalls wird die C-Stil Reihenfolge verwendet.

`K`: Die Elemente werden so gelesen, wie sie im Speicher vorkommen, außer für Datenumkehrung, wenn die Schrittweiten negativ sind.

```
print(A.ravel())

print(A.ravel(order="A"))

print(A.ravel(order="F"))

print(A.ravel(order="A"))

print(A.ravel(order="K"))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[ 0  8 16  2 10 18  4 12 20  6 14 22  1  9 17  3 11 19  5 13 21  7 15 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

Unterschiede zwischen ravel und flatten

- `ravel` liefert in der Regel keine Kopie, sondern eine auf die Dimension angepasste View auf das Originalarray zurück.
- `flatten` liefert immer eine Kopie zurück.
- `ravel` ist schneller als `flatten`, weil es keine Kopie erzeugen muss.

Wir zeigen dies in Beispielen:

```
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6]])

B = A.flatten()
B[4] = 42

print("B: \n", B)
print("A: \n", A)
print(np.may_share_memory(A, B))

print("\n... und jetzt das Ganze mit ravel:")
B = A.ravel()
B[4] = 42

print("B: \n", B)
print("A: \n", A)
print(np.may_share_memory(A, B))
```

```
B:
 [ 1  2  3  4 42  6]
A:
 [[1 2 3]
 [4 5 6]]
False

... und jetzt das Ganze mit ravel:
B:
 [ 1  2  3  4 42  6]
A:
 [[ 1  2  3]
 [ 4 42  6]]
True
```

reshape Die `reshape`-Methode wandelt ein Array in eine neue Gestalt, englisch „Shape'', ohne die darin enthaltenen Daten zu ändern, d.h. die eigentlichen Daten müssen nicht kopiert werden.

```
reshape(a, newshape, order='C')
```

Parameter	Bedeutung
a	array-ähnlich, Array, das geändert werden soll.
newshape	Integer-Wert oder Integer-Tupel
order	'C', 'F', 'A', wie in flatten() oder ravel()

Mittels `reshape` können wir ein Array auch linearisieren:

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])

B = A.reshape((6,))
print(B)
```

```
[1 2 3 4 5 6]
```

Damit kann `reshape` die Aufgaben von `ravel` und `flatten` übernehmen. `reshape` kann aber noch mehr. Wir können damit ein Array A in eine beliebige Gestalt `x` überführen, solange das Produkt der Shape-Komponenten von A gleich dem Produkt der Shape-Komponenten von `x` ist, also

`np.prod(A.shape) == np.prod(x)`

```
X = np.array(range(24))
Y1 = X.reshape((3, 4, 2))
print(Y1)
new_shape = (2, 3, 4)
Y2 = Y1.reshape(new_shape)
print(Y2)
```

```
[[[ 0  1]
  [ 2  3]
  [ 4  5]
  [ 6  7]]

 [[ 8  9]
  [10 11]
  [12 13]
  [14 15]]

 [[16 17]
  [18 19]
  [20 21]
  [22 23]]]
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Es gilt:

```
np.prod(Y1.shape) == np.prod(new_shape)
```

```
True
```

Konkatenation von Arrays

Im folgenden Beispiel konkatenieren wir drei eindimensionale Arrays zu einem. Die Elemente des zweiten Arrays werden an das erste Array horizontal angefügt. Anschließend werden

die Elemente des dritten Arrays ebenfalls horizontal angefügt:

```
x = np.array([11, 22])
y = np.array([18, 7, 6])
z = np.array([1, 3, 5])
c = np.concatenate((x, y, z))
print(c)
```

```
[11 22 18  7  6  1  3  5]
```

Wenn wir multidimensionale Arrays zusammenführen, müssen wir auf die Achsen achten. Die Arrays müssen die gleiche Shape haben, um mit `concatenate` zusammen gefügt werden zu können. Bei multidimensionalen Arrays können wir diese entsprechend anordnen. Der Default-Wert ist `axis = 0`:

```
x = np.array(range(12))
x = x.reshape((3, 4))
y = np.array(range(100, 112))
y = y.reshape((3, 4))
z = np.concatenate((x, y))
print(z)
```

```
[[  0   1   2   3]
 [  4   5   6   7]
 [  8   9  10  11]
 [100 101 102 103]
 [104 105 106 107]
 [108 109 110 111]]
```

Wir führen die Zusammenführung nun mit `axis = 1` durch:

```
z = np.concatenate((x, y), axis=1)
print(z)
```

```
[[  0   1   2   3 100 101 102 103]
 [  4   5   6   7 104 105 106 107]
 [  8   9  10  11 108 109 110 111]]
```

Weitere Dimensionen hinzufügen

Weitere Dimensionen können zu einem Array mit Hilfe von Slicing und `np.newaxis` hinzugefügt werden. Wir demonstrieren die Technik im folgenden Beispiel:

```
x = np.array([2,5,18,14,4])
y = x[:, np.newaxis]
print(y)
```

```
[[ 2]
 [ 5]
 [18]
 [14]
 [ 4]]
```

Das gleiche Resultat lässt sich auch mit `reshape` bewerkstelligen:

```
x = np.array([2,5,18,14,4])
y = x.reshape( (x.shape[0], 1) )
print(y)
```

```
[[ 2]
 [ 5]
 [18]
 [14]
 [ 4]]
```

Vektoren stapeln

```
A = np.array([[3, 4, 5]])
B = np.array([[1,9,0]])

C = np.dstack((A, B))

print("Elemente von A:")
for i in range(C.shape[1]):
    print(C[0, i, 0], end=", ")
print("\nElemente von B:")
for i in range(C.shape[1]):
    print(C[0, i, 1], end=", ")
```

```
Elemente von A:
3, 4, 5,
Elemente von B:
1, 9, 0,
```

Wir können sehen, dass wir den i-ten Eingabevektor von C erhalten, indem wir den Ausdruck `C[:, :, i]` ausführen:

```
print("Erster Eingabevektor, d.h. A:", C[:, :, 0])
print("Zweiter Eingabevektor, d.h. B:", C[:, :, 1])
```

```
Erster Eingabevektor, d.h. A: [[3 4 5]]
Zweiter Eingabevektor, d.h. B: [[1 9 0]]
```

Als Nächstes betrachten wir die Methoden `row_stack` und `column_stack`:

```
A = np.array([3, 4, 5])
B = np.array([1,9,0])

print(np.row_stack((A, B)))

print(np.column_stack((A, B)))
print(np.shape(A))
```

```
[[3 4 5]
 [1 9 0]]
[[3 1]
 [4 9]
```

```
 [5 0]]
(3,)
```

```
A = np.array([[1, 2],
              [3, 4]])
X = np.column_stack((A, A, A))
print(np.row_stack((X, X, X)))
```

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

"Fliesen" mit "tile"

Manchmal ist es nötig, eine Matrix zu erstellen (mit einer anderen Shape oder Dimension), die den Inhalt einer existierenden Matrix mehrfach enthält.

Beispiel:

Man möchte das eindimensionale Array `array([ 3.4])` in das Array `array([ 3.4,  3.4, 3.4,  3.4,  3.4])` wandeln.

Weiteres Beispiel:

Man möchte ein zweidimensionales Array, wie `np.array([ [1, 2], [3, 4]])`, als Baustein benutzen, um ein Array mit der Shape (6, 8) zu erstellen:

Die Konstruktionsidee ist im folgenden Diagramm dargestellt:

```
import numpy as np
x = np.array([ [1, 2], [3, 4]])
print(np.tile(x, (3, 4)))
```

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

```
import numpy as np

x = np.array([ 3.4])

y = np.tile(x, (5,))

print(y)
```

```
[3.4 3.4 3.4 3.4 3.4]
```

Im vorigen `tile`-Beispiel hätten wir ebenso `y = np.tile(x, 5)` schreiben können.

Wenn wir `reps` in Tupel- oder List-Form schreiben, oder `reps = 5` als Ersatz für `reps = (5,)` in Erwägung ziehen, so ist Folgendes wahr:

Wenn `reps` die Länge `n` hat, so wird die Dimension des resultierenden Arrays maximal `n` und `A.ndim` sein.

Wenn `A.ndim < n` ist, so wird `A` durch Voranstellen neuer Achsen auf die n-Dimensionen befördert. Beispielsweise wird ein Array mit der Shape `(5,)` auf `(1, 5)` bei einer 2-D-Replikation befördert oder auf die Shape `(1, 1, 5)` bei einer 3-D-Replikation. Sollte das nicht dem gewünschten Verhalten entsprechen, so ist `A` vor dem Aufruf der tile-Funktion auf die gewünschte Shape anzupassen.

Wenn `A.ndim > d` ist, so wird `reps` auf `A.ndim` angepasst, indem 1's vorangestellt werden. Beispielsweise wird ein Array `A` mit der Shape `(2, 3, 4, 5)` und `reps = (2, 2)` als `(1, 1, 2, 2)` behandelt.

Weitere Beispiele:

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
print(np.tile(x, 2))
```

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

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
print(np.tile(x, (2, 1)))
```

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

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
print(np.tile(x, (2, 2)))
```

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

7 Python Numpy Wahrscheinlichkeit

7.0.1 Python, NumPy and Wahrscheinlichkeiten

Einführung

"Every American should have above average income, and my Administration is going to see they get it."

Wir beginnen mit diesem Zitat von Bill Clinton unser Kapitel über Statistik, weil es ein besonders hervorstechendes Beispiel eines weit vorherrschenden Problems ist. Ein Problem, was der Mathematiker John Allen Paulos in einem seiner Bücher als "Innumeracy" bezeichnet hatte. (Paulos, John Allen. 1988. "Innumeracy, Mathematical Illiteracy and its consequences". New York. Hill and Wang) Ein Ausdruck für den es noch keine deutsche Übersetzung gibt, aber soviel bedeutet wie "numerischer Analphabetismus". Darunter versteht er die weit-verbreitete Unfähigkeit einfachste statistische und wahrscheinlichkeitstheoretische Zusammenhänge zu begreifen. Während Analphabetismus in allen Gesellschaften und Gesellschaftskreisen geächtet wird, gilt "innumeracy" als tolerierbar. Nehmen wir an, es gäbe ein Land, in dem fast alle Leute eine grüne Hautfarbe hätten, außer einer blaufarbigen Minderheit. Nun können wir davon ausgehen, dass es unter beiden Gruppen Verbrecher geben wird. Stellen wir uns aber vor, dass innerhalb kurzer Zeit Zeitungen mit Artikeln der Form "Blauer mordet Mutter von zwei Kindern" oder "Polizei ermittelt erneut gegen Blauen im Fall XYZZ!" veröffentlicht werden. Über die gleichzeitig stattfindenden Straftaten von Menschen grüner Hautfarbe wird jedoch nicht berichtet. Numerischer Analphabetismus bedeutet nun, dass Leute auf die Strasse gehen und gegen die durch "Blaue" ausgelöste Gewalt demonstrieren, Politiker ein mögliches Problem bei Blauen untersuchen wollen oder gar zu erkenen glauben und so weiter.

Statistik und Wahrscheinlichkeitsrechnung begegnet uns nahezu überall in unserem Leben. Wir müssen damit umgehen, häufig wenn wir zwischen verschiedenen Alternativen zu wählen haben. Können wir morgen wandern gehen, oder wird es regnen? Die Wettervorhersage sagt uns, dass die Niederschlagswahrscheinlichkeit bei 30% liegt. Was jetzt? Können wir es riskieren? Anderes Szenario: Sie spielen jede Woche Lotto und träumen von einer weit entfernten Insel. Wie hoch ist die Wahrscheinlichkeit, den Jackpot zu gewinnen, sodass Sie niemals mehr arbeiten müssen und im "Paradies" leben können? Nicht sehr wahrscheinlich, aber nun stellen Sie sich vor, dass Sie den Jackpot geknackt haben. Wie groß ist die Wahrscheinlichkeit, dass Sie auf der Insel Ihrer Träume, weit weg von zu Hause, Ihren Nachbarn treffen? Wie hoch sind die Chancen, dass so etwas passiert?

Die Ungewissheit umgibt uns, und nur einige Menschen verstehen die Grundlagen der Wahrscheinlichkeitstheorie.

Die Programmiersprache Python und die Module NumPy und SciPy helfen uns nicht, die oben genannten alltäglichen Probleme zu verstehen. Jedoch bieten Python und NumPy starke Funktionalitäten, um Probleme der Statistik und Wahrscheinlichkeitstheorie zu berechnen.

Zufallszahlen mit Python

Die Module random und secrets Das Modul `secrets` wurde erst mit Python 3.6 neu eingeführt. Mit diesem Modul kann man kryptografisch starke Pseudozufallszahlen erzeugen, die sich als Passwörter, Tokens oder Ähnliches eignen. Dieses Modul wurde mit diesem Ziel entwickelt. Als Generator enthält es eine CSPRNG (Cryptographically Strong Pseudo Random Number Generator). Das `random`-Modul von Python wurde nicht in Hinblick auf kryptografische Anwendungen entwickelt. Der Fokus dieses Moduls lag auf Modellbildungen und Simulationen.

Es gibt sogar eine ausdrückliche Warnung in der Dokumentation des `random`-Modules:

Warnung: Beachten Sie bitte, dass der Pseudo-Zufalls-Generator im random-Modul NICHT für sicherheitsrelevante Zwecke benutzt werden sollte. Benutzen Sie secrets ab Python 3.6+ und os.urandom() bei Python 3.6+ und früher. Im Original: "Note that the pseudo-random generators in the random module should NOT be used for security purposes. Use secrets on Python 3.6+ and os.urandom() on Python 3.5 and earlier."

Man kann allerdings die `SystemRandom`-Klasse des `random`-Moduls verwenden, die die sichere `os.urandom`-Systemfunktion benutzt.

`random.SystemRandom` und `secrets.SystemRandom` sind identisch.

Mit der Funktion `random.random` können wir eine Zufallszahl im halb-offenen Intervall `[0, 1)` erzeugen:

```
import random
random_number = random.random()
print(random_number)
```

```
0.38478750165423237
```

Nun wollen wir eine kryptografisch starke Zufallszahl erzeugen:

```
from secrets import SystemRandom
# from random import SystemRandom   # äquivalent
crypto = SystemRandom()
print(crypto.random())
```

```
0.9204267821655968
```

Erzeugen einer Liste von Zufallszahlen Häufig benötigen wir mehr als eine Zufallszahl. Die folgende Funktion `random_list` kann eine Liste von Zufallszahlen mit vorgegebener Anzahl erzeugen. Mit dem Parameter `secure` können wir steuern, ob wir sichere, also mit `SystemRandom` erzeugte Zahlen wollen:

```
import random

def random_list(n, secure=True):
    random_floats = []
    if secure:
        crypto = random.SystemRandom()
        random_float = crypto.random
    else:
```

```
        random_float = random.random
    for _ in range(n):
        random_floats.append(random_float())
    return random_floats

print(random_list(3, secure=False))
```

```
[0.7909000388784871, 0.10284379569236879, 0.050261473684067526]
```

Im Folgenden können wir sehen, dass der Preis für die Sicherheit eine deutliche Erhöhung der Rechenzeit bedeutet:

```
%%timeit
random_list(100, secure=True)
```

```
357 µs ± 5.46 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
%%timeit
random_list(100, secure=False)
```

```
7.18 µs ± 485 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Am einfachsten, was den Programmieraufwand betrifft, und am schnellsten, was die Laufzeit betrifft, lassen sich Zufallszahlen mit dem `random`-Untermodul von `numpy` erzeugen:

```
import numpy as np

np.random.random(10)
```

```
array([0.25935606, 0.07716387, 0.84629528, 0.83033122, 0.72322316,
       0.06717942, 0.21947731, 0.37639418, 0.49988897, 0.55997848])
```

```
%%timeit
np.random.random(100)
```

```
1.6 µs ± 8.23 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Allerdings werden auch hier keine sicheren Zufallszahlen erzeugt!

Zufällige Integer-Zahlen mit Python

Jeder ist mit der Generierung von Zufallszahlen ohne Computer vertraut. Wenn Sie einen Würfel werfen, erhalten Sie eine Zufallszahl zwischen 1 und 6. Im Zusammenhang mit der Wahrscheinlichkeitstheorie nennen wir "Werfen eines Würfels" ein Experiment, dessen Menge der möglichen Ergebnisse {1, 2, 3, 4, 5, 6} ist. Diese Menge wird auch als "Stichprobenraum" bezeichnet.

Integer-Zahlen könnten wir relativ einfach aus dem erzeugen, was wir schon kennengelernt haben. Unsere Funktion erzeugt Zahlen im halboffenen Intervall zwischen `start` und `stop`, also `[start, stop)`:

```
def randint(start, stop, secure=True):
```

```
    if secure:
        crypto = random.SystemRandom()
        rnd = crypto.random
    else:
        rnd = random.random
    result = int(rnd() * (stop-start)) + start
    return result

[ randint(3, 7) for i in range(20)]
```

```
[5, 4, 4, 6, 4, 6, 6, 6, 3, 3, 4, 3, 6, 5, 6, 6, 5, 4, 5, 3]
```

Es empfiehlt sich jedoch, die Funktion `randint` zu nutzen, die die Module `random`, `secrets` und `numpy.random` bietet .

```
import random

# randint
outcome = random.randint(1, 6)
print(outcome)
```

```
1
```

```
import secrets

c = secrets.SystemRandom()
print(c.randint(1, 6))
```

```
3
```

Würfeln wir sehr oft, so sollte jede Augenzahl ungefähr in einem Sechstel der Fälle auftreten. Wir testen dies im Folgenden. Das Zählen erledigen wir mit der `Counter`-Klasse aus dem `collections`-Modul:

```
from collections import Counter
import random

c = Counter()

anzahl_wuerfe = 100000
for wurf in range(anzahl_wuerfe):
    augenzahl = random.randint(1, 6)
    c[augenzahl] += 1

print(c)
for augenzahl in c:
    print(f"Augenzahl: {augenzahl}, \
          relative Häufigkeit: {c[augenzahl]/anzahl_wuerfe}")
```

```
Counter({2: 16862, 3: 16687, 4: 16675, 1: 16639, 6: 16604, 5: 16533})
Augenzahl: 1,           relative Häufigkeit: 0.16639
Augenzahl: 5,           relative Häufigkeit: 0.16533
Augenzahl: 2,           relative Häufigkeit: 0.16862
Augenzahl: 6,           relative Häufigkeit: 0.16604
Augenzahl: 3,           relative Häufigkeit: 0.16687
```

```
Augenzahl: 4,           relative Häufigkeit: 0.16675
```

Obiges Beispiel lässt sich mit der random-Funktion aus dem NumPy-Modul einfacher realisieren.

```
import numpy as np
from collections import Counter

anzahl_wuerfe = 100000
outcome = np.random.randint(1, 7, size=anzahl_wuerfe)

c = Counter(outcome)
for augenzahl in c:
    print(f"Augenzahl: {augenzahl}, \
          relative Häufigkeit: {c[augenzahl]/anzahl_wuerfe}")
```

```
Augenzahl: 2,           relative Häufigkeit: 0.16764
Augenzahl: 6,           relative Häufigkeit: 0.16496
Augenzahl: 5,           relative Häufigkeit: 0.16557
Augenzahl: 3,           relative Häufigkeit: 0.16739
Augenzahl: 1,           relative Häufigkeit: 0.16742
Augenzahl: 4,           relative Häufigkeit: 0.16702
```

Sicherlich haben Sie bemerkt, dass wir in `random.randint` beim zweiten Parameter ein 7 statt einer 6 verwendet haben. Diese Funktion nutzt ein "halb-offenes" Intervall im Gegensatz zu Pythons `random`-Modul, das ein geschlossenes Intervall erwartet.

Die formale Definition:

`numpy.random.randint(low, high=None, size=None)`

Diese Funktion liefert zufällige ganze Zahlen zwischen "low" (inklusiv) und "high" (exklusiv). Mit anderen Worten: `randint` liefert zufällige Integers aus der diskret uniformen Distribution im "halb-offenen" Intervall [`low`, `high`).

Wenn für `high` der Wert `None` oder nichts übergeben wird, liegen die Ergebnisse in der Range von [0, `low`).

Der Parameter `size` definiert die Shape des Ergebnisses. Wenn für `size` None oder nichts übergeben wird, generiert die Funktion einen Integer-Wert. Andernfalls ist das Ergebnis ein Array. `size` sollte ein Tupel sein. Wenn als `size` ein Integer n übergeben wird, entspricht dies dem Tupel (n,).

Folgende Beispiele verdeutlichen das Verhalten der Parameter:

```
import numpy as np

print("Eine Integer-Zahl: ",
      np.random.randint(1, 7))
print("\nEin eindimensionales Array mit einem Element:\n",
      np.random.randint(1, 7, size=1))
print("\nEin eindimensionales Array mit zehn Elementen:\n",
      np.random.randint(1, 7, size=10))
print("\nWie eben, aber alternative size-Definition:\n",
      np.random.randint(1, 7, size=(10,)))
print("\nZweidimensionales Array:\n",
      np.random.randint(1, 7, size=(5, 4)))
```

```
Eine Integer-Zahl:  4

Ein eindimensionales Array mit einem Element:
 [3]

Ein eindimensionales Array mit zehn Elementen:
 [5 3 5 4 3 1 2 5 5 1]

Wie eben, aber alternative size-Definition:
 [2 3 1 6 1 2 5 4 4 4]

Zweidimensionales Array:
 [[3 5 2 5]
 [1 3 3 5]
 [1 2 4 3]
 [3 2 4 1]
 [6 3 2 3]]
```

Wir können das Würfeln mit dem Code `np.random.randint(1, 7, size=1)` aus dem NumPy-Modul simulieren oder anhand des Codes `random.randint(1, 6)` aus dem Standard-random-Modul. Wir nehmen an, dass unser Würfel fair ist, d.h. die Wahrscheinlichkeit für jede Seite bei 1/6 liegt.

Wie können wir einen gezinkten Würfen simulieren? Die `randint`-Methoden beider Module sind dafür nicht geeignet. Wir werden im Folgenden ein paar Funktionen schreiben, um das Problem zu lösen.

Zunächst schauen wir uns weitere nützliche Funktionen des `random`-Moduls an.

Stichproben/Auswahlen

`choice` ist eine weitere extrem nützliche Funktion des `random`-Moduls. Sie kann genutzt werden, um aus einer nicht-leeren Sequenz ein zufälliges Element zu wählen.

Sequenzen können beispielsweise Listen, Strings und Tupels sein, aber auch Iteratoren. Das bedeutet, wir sind in der Lage, aus einem String ein zufälliges Zeichen zu holen oder ein zufälliges Element aus einer Liste oder einem Tupel:

```
from random import choice

professions = ["scientist", "philosopher", "engineer", "priest"]
print(choice("abcdefghij"))
print(choice(professions))
print(choice(("apples", "bananas", "cherries")))
print(choice(range(10)))
```

```
i
scientist
apples
9
```

`choice` liefert zufällig ein Objekt aus einer nicht-leeren Sequenz, wobei die Chancen für die Elemente, ausgewählt zu werden, gleichmäßig verteilt sind. So liegt die Chance für die Rückgabe von `scientist` beim Aufruf von `choice(profession)` bei 1/4. Das hat allerdings wenig mit der Realität zu tun. Um die Realität nachzubilden benötigen wir auch hier – wie beim gezinkten Würfel – eine gewichtete Auswahl.

Wir definieren nun eine Funktion `weighted_choice`, die wie `random.choice` ein zufälliges Element einer Sequenz zurückliefert, jedoch sind die Elemente der Sequenz gewichtet.

Zufallsintervalle

Bevor wir mit dem Design der gewichteten Auswahl beginnen, definieren wir eine Funktion `find_interval(x, partition)`, die wir für unsere `weighted_choice`-Funktion brauchen. `find_interval` erwartet zwei Argumente:

- Einen numerischen Wert x
- Eine Liste oder einen Tupel mit numerischen Werten p0, p1, p2, … pn

Die Funktion liefert i zurück, wenn pi $<$ x $<$ pi+1. -1 wird zurückgeliefert, wenn x kleiner als p0 ist oder x größer oder gleich pn ist.

```
def find_interval(x, partition):
    """ find_interval -> i
        partition ist eine Sequenz von numerischen Werten
        x ist eine Int- oder Float-Zahl
        Bei dem Rückgabewert "i" handelt es sich um den Index
        für den gilt partition[i] < x < partition[i+1],
        falls solch ein Index existiert. -1 ansonsten
    """

    for i in range(0, len(partition)):
        if x < partition[i]:
            return i-1
    return -1


I = [0, 3, 5, 7.8, 9, 12, 13.8, 16]
for x in [-1.3, 0, 0.1, 3.2, 5, 6.2, 7.9, 10.8, 13.9, 15, 16, 16.5]:
    print(find_interval(x, I), end=", ")
```

```
-1, 0, 0, 1, 2, 2, 3, 4, 6, 6, -1, -1,
```

Gewichtete Zufallsauswahl

Jetzt können wir die `weighted_choice`-Funktion definieren. Nehmen wir an, dass wir drei Gewichtungen haben, d.h. 1/5, 1/2 und 3/10. Wir bilden die kumulative Summe der Gewichtungen mit `np.cumsum(weights)`.

```
import numpy as np

weights = [0.2, 0.5, 0.3]
cum_weights = [0] + list(np.cumsum(weights))
print(cum_weights)
```

```
[0, 0.2, 0.7, 1.0]
```

Wenn wir eine Zufallszahl x zwischen 0 und 1 mit `random.random()` generieren, so ist die Wahrscheinlichkeit, dass x im Intervall `[0, cum_weights[0])` liegt, 1/5. Die Wahrscheinlichkeit, dass x im Intervall `[cum_weights[0], cum_weights[1])` liegt, ist 1/2. Letztlich

ist die Wahrscheinlichkeit, dass x im Intervall `[cum_weights[1], cum_weights[2])` , bei 3/10.

Jetzt sind Sie in der Lage, die grundlegende Idee zu verstehen, auf der `weighted_choice` basiert:

```
import numpy as np

def weighted_choice(sequence, weights):
    """
    weighted_choice wählt ein Zufallselement aus
    'sequence' aus unter Berücksichtigung der
    List bzw. Tupel von Gewichten
    """
    x = np.random.random()
    cum_weights = [0] + list(np.cumsum(weights))
    index = find_interval(x, cum_weights)
    return sequence[index]
```

Beispiel:

Wir können die Funktion `weighted_choice` nun für die folgende Aufgabe verwenden: Stellen wir uns vor, dass wir einen gezinkten Würfel haben, sodass für die Wahrscheinlichkeiten des Auftretens der Augenzahlen 6 und 1 gilt: P(6)=3/12 und P(1)=1/12. Die Wahrscheinlichkeit für alle anderen möglichen Ergebnisse sind gleich, d.h. P(2) = P(3) = P(4) = P(5) = p. Wir können p wie folgt ausrechnen mit

`1 - P(1) - P(6) = 4 x p`

Das entspricht

`p = 1/6`

Wie können wir diesen Würfel mit unserer `weighted_choice`-Funktion simulieren?

Wir rufen `weighted_choice` mit den Augenzahlen des Würfels und der Liste der korrespondierenden Gewichte auf. Jeder Aufruf entspricht einem Wurf des gezinkten Würfels.

Wir sehen, dass nach 10.000 Würfen die geschätzte Wahrscheinlichkeit der Gewichtung entspricht

```
from collections import Counter

faces_of_dice = [1, 2, 3, 4, 5, 6]
weights = [1/12, 1/6, 1/6, 1/6, 1/6, 3/12]

outcomes = []
n = 10000
for _ in range(n):
    outcomes.append(weighted_choice(faces_of_dice, weights))

c = Counter(outcomes)
for key in c:
    c[key] = c[key] / n

print(sorted(c.values()))
```

```
[0.0832, 0.1654, 0.1682, 0.169, 0.1697, 0.2445]
```

Die Werte der Aufteilungs-Liste definieren die Intervalle, in denen wir den Wert x erwarten. Wenn der Wert x kleiner als p0 oder grösser/gleich pn, liefern wir -1 zurück.

Wir könnten unsere erste Unterteilung als ein Intervall von

- ∞

bis p0 definieren und 0 zurückliefern. Die letzte Unterteilung wäre dann ein Intervall von pn bis
∞.

Um zwischen beiden Fällen unterscheiden zu können, erweitern wir die Funktion find_interval um den Parameter “endpoints”. “True” entspricht unserem Eingangs erwähnten Vorgehen. “False” entspricht unserem soeben beschriebenen Fall.

In anderen Worten - Wenn “endpoints” False ist, passiert folgendes: - i wird zurückgegeben wenn x kleiner als pi ist - len(partition) wird zurückgegeben, wenn x größer oder gleich plen(partition)-1 ist.

Wir demonstrieren dies im folgenden Diagramm:

Die neue Funktion sieht folgendermaßen aus:

```
def find_interval(x,
                  partition,
                  endpoints=True):
    """
    find_interval -> i
    Falls endpoints gleich True ist, ist "i" der Index für den gilt
    partition[i] < x < partition[i+1] gilt, falls solch ein Index
    existiert. -1 otherwise

    Falls endpoints gleich False ist, ist "i" der kleinste
    Index für den x < partition[i] gilt. Falls kein solcher
    Index existiert, len(partition) auf i gesetzt.
    """
    for i in range(0, len(partition)):
        if x < partition[i]:
            return i-1 if endpoints else i
    return -1 if endpoints else len(partition)

I = [0, 3, 5, 7.8, 9, 12, 13.8, 16]
print("Endpunkte sind inbegriffen:")
for x in [-1.3, 0, 0.1, 3.2, 5, 6.2, 7.9, 13.9, 15, 16, 16.5]:
    print(find_interval(x, I), end=", ")
print("\nEndpunkte sind nicht inbegriffen:")
for x in [-1.3, 0, 0.1, 3.2, 5, 6.2, 7.9, 13.9, 15, 16, 16.5]:
    print(find_interval(x, I, endpoints=False), end=", ")
```

```
Endpunkte sind inbegriffen:
-1, 0, 0, 1, 2, 2, 3, 6, 6, -1, -1,
Endpunkte sind nicht inbegriffen:
0, 1, 1, 2, 3, 3, 4, 7, 7, 8, 8,
```

Stichproben mit Python

Eine Sample oder Stichprobe kann als ein repräsentativer Teil einer größeren Gruppe angesehen werden, die wir "Population" nennen.

Das Modul `numpy.random` beinhaltet die Funktion `random_sample`, die zufällige Float-Werte im halb-offenen Intervall `[0.0, 1.0)` liefert. Die Ergebnisse sind gleichmäßig über das angegebene Intervall verteilt. Die Funktion erwartet lediglich einen Parameter `size`, der die Shape der Ausgabe definiert. Wenn wir die `size` zum Beispiel mit (3, 4) angeben, erhalten wir ein Array mit der Shape (3, 4), das mit Zufallswerten gefüllt ist:

```
import numpy as np

x = np.random.random_sample((3, 4))
print(x)
```

```
[[0.45711549 0.09158949 0.4548514  0.04671449]
 [0.96838841 0.1316467  0.09991307 0.71975092]
 [0.82073319 0.70767492 0.99399057 0.00218589]]
```

Wenn random_sample mit einem Integer-Wert aufgerufen wird, erhalten wir ein eindimensionales Array. Ein Integer-Wert bewirkt den gleichen Effekt, wie ein einfaches Tupel als Argument:

```
x = np.random.random_sample(7)
print(x)

y = np.random.random_sample((7,))
print(y)
```

```
[0.85343344 0.89420157 0.86387465 0.62418901 0.01643744 0.53475791
 0.75412648]
[0.64027827 0.70709433 0.825084   0.18850273 0.0087134  0.58973563
 0.07864032]
```

Es können ebenfalls Arrays aus einem beliebigen Intervall `[a, b)` generiert werden, wobei `a` kleiner als `b` sein muss. Das kann wie folgt aussehen:

```
(b - a) * random_sample() + a
```

Beispiel:

```
a = -3.4
b = 2

A = (b - a) * np.random.random_sample((3, 4)) + a

print(A)
```

```
[[-0.61985952  0.00678915  0.23543105  0.61122334]
 [-3.03073703 -1.49931968 -3.00836952 -2.79980084]
 [-0.41013124 -1.47345358 -1.23384511  1.05400615]]
```

Das Standard-Modul `random` von Python hat eine allgemeinere Funktion `sample`, die Stichproben einer Population produziert. Die Population ist eine Sequenz, also beispielsweise

eine Liste, Menge oder ein String.

Die Syntax von `sample`:

`sample(population, k)`

Die Funktion erstellt eine Liste, die k Elemente aus der `population` beinhaltet. Die Ergebnisliste beinhaltet keine Duplikate, wenn in der Population keine Duplikate vorkommen.

Wenn eine Sample aus einer Range von Integer-Werten ausgewählt werden soll, dann können – oder besser sollten – Sie `range` als Argument für die Population verwenden.

Im folgenden Beispiel ziehen wir sechs Zahlen aus der Range zwischen 1 und 49 (inklusive). Das entspricht einer Lotto-Ziehung in Deutschland:

```
import random

print(random.sample(range(1, 50), 6))
```

```
[13, 28, 26, 19, 36, 31]
```

```
def weighted_sample(population, weights, k):
    """
    weighted_sample zieht eine Zufallsstichprobe der
    Länge k aus der Sequenz 'population' entsprechend
    der Liste der Gewichte.
    """
    sample = set()
    population = list(population)
    weights = list(weights)
    while len(sample) < k:
        choice = weighted_choice(population, weights)
        sample.add(choice)
        index = population.index(choice)
        weights.pop(index)
        population.remove(choice)
        weights = [ x / sum(weights) for x in weights]
    return list(sample)
```

```
def weighted_sample_alternative(population, weights, k):
    """
    weighted_sample zieht eine Zufallsstichprobe der
    Länge k aus der Sequenz 'population' entsprechend
    der Liste der Gewichte.
    """
    sample = set()
    population = list(population)
    weights = list(weights)
    while len(sample) < k:
        choice = weighted_choice(population, weights)
        if choice not in sample:
            sample.add(choice)
    return list(sample)
```

Beispiel:

Nehmen wir an, wir haben acht Zuckerstücke in den Farben rot, grün, blau, gelb, schwarz, weiß, pink und orange. Unser Freund Peter hat für die Farben die "gewichteten" Vorlieben 1/24, 1/6, 1/6, 1/12, 1/12, 1/24, 1/8, 7/24. Peter darf sich 3 Zuckerstücke aussuchen:

```
balls = ["red", "green", "blue", "yellow", "black",
         "white", "pink", "orange"]
weights = [ 1/24, 1/6, 1/6, 1/12, 1/12, 1/24, 1/8, 7/24]
for i in range(10):
    print(weighted_sample(balls, weights, 3))
```

```
['pink', 'blue', 'orange']
['pink', 'blue', 'orange']
['black', 'pink', 'orange']
['white', 'blue', 'orange']
['red', 'yellow', 'white']
['red', 'pink', 'yellow']
['pink', 'blue', 'green']
['white', 'green', 'orange']
['black', 'blue', 'white']
['black', 'green', 'orange']
```

Im Folgenden vergleichen wir die beiden Varianten, gewichtete Stichproben zu berechnen:

```
n = 10000
orange_counter = 0
orange_counter_alternative = 0
for i in range(n):
    if "orange" in weighted_sample(balls, weights, 3):
        orange_counter += 1
    if "orange" in weighted_sample_alternative(balls, weights, 3):
        orange_counter_alternative += 1

print(orange_counter / n)
print(orange_counter_alternative / n)
```

```
0.7135
0.7154
```

Kartesische Auswahl

Die Funktion `cartesian_choice`, die wir im Folgenden definieren werden, ist nach dem Kartesischen Produkt aus der Mengenlehre benannt.

Kartesisches Produkt Das kartesische Produkt, auch Mengenprodukt genannt, ist ein Konstruktionsprinzip, um aus Mengen eine neue Menge zu konstruieren.

Für zwei Mengen `A` und `B` ist das kartesische Produkt `A x B` die Menge aller geordneten Paare `(a, b)`, für die `a  A` und `b  B` gilt:

`A x B = { (a, b) | a &isin; A and  b &isin; B }`

Ganz allgemein besteht das kartesische Produkt mehrerer Mengen aus der Menge aller Tupel von Elementen der Mengen, wobei die Reihenfolge der Mengen und damit der

entsprechenden Elemente fest vorgegeben ist. Die Ergebnismenge des kartesischen Produkts wird auch Produktmenge, Kreuzmenge oder Verbindungsmenge genannt.

Wenn wir n Mengen haben A1, A2, ... An, können wir das kartesische Produkt entsprechend wie folgt bilden:

A1 x A2 x ... x An = { (a1, a2, ... an) | a1 A1, a2 A2, ... an An]

Das kartesische Produkt aus n Mengen wird manchmal auch n-faches kartesisches Produkt genannt.

Kartesische Auswahl: cartesian_choice Wir schreiben nun eine Funktion `cartesian_choice`, die eine beliebige Anzahl von iterierbaren Argumenten erwartet und eine Liste zurückliefert, die eine zufällige Auswahl von jedem Iterator in der entsprechenden Reihenfolge beinhaltet.

Aus mathematischer Sicht können wir das Ergebnis der Funktion `cartesian_choice` als Element des kartesischen Produkts der übergebenen iterierbaren Argumente ansehen.

```
import random

def cartesian_choice(*iterables):
    """
    cartesian_choice erzeugt eine Liste deren i-tes Element
    einer zufälligen Auswahl aus dem i-ten Eingabeargument entspricht.
    Die Ergebnisliste kann als kartesisches Produkt über die iterierbaren
    Eingabeobjekte gesehen werden.
    """
    res = []
    for population in iterables:
        res.append(random.choice(population))
    return res


cartesian_choice(["The", "A"],
                 ["red", "green", "blue", "yellow", "grey"],
                 ["car", "house", "fish", "light"],
                 ["smells", "dreams", "blinks"])
```

```
['A', 'red', 'house', 'blinks']
```

Wir definieren nun eine gewichtete Version der vorher definierten Funktion:

```
import random

def weighted_cartesian_choice(*iterables):
    """
    weighted_cartesian_choice liefert eine Liste mit
    gewichteten Auswahlen aus den iterierbaren Eingabeobjekten
    unter Wahrung der Reihenfolge.
    """
    res = []
    for population, weight in iterables:
        lst = weighted_choice(population, weight)
        res.append(lst)
    return res
```

```
determiners = (["The", "A", "Each", "Every", "No"],
               [0.3, 0.3, 0.1, 0.1, 0.2])
colours = (["red", "green", "blue", "yellow", "grey"],
           [0.1, 0.3, 0.3, 0.2, 0.2])
nouns = (["door", "elephant", "fish", "light",
          "programming language", "Python"],
         [0.2, 0.2, 0.1, 0.1, 0.3, 0.1])
nouns2 = (["of happiness", "of chocolate", "of wisdom",
           "of challenges", "of air"],
          [0.5, 0.2, 0.1, 0.1, 0.1])
verb_phrases = (["smells", "dreams", "thinks",
                 "is made", "consists"],
          [0.1, 0.3, 0.3, 0.2, 0.1])

print("It may or may not be true:")
for i in range(10):
    res = weighted_cartesian_choice(determiners,
                                    colours,
                                    nouns,
                                    verb_phrases,
                                    nouns2)
    print(" ".join(res) + ".")
```

```
It may or may not be true:
No green Python thinks of happiness.
Each yellow Python smells of happiness.
The red elephant thinks of happiness.
The blue programming language smells of challenges.
No green door smells of chocolate.
A green fish dreams of wisdom.
The green programming language consists of chocolate.
No red programming language is made of happiness.
The yellow door thinks of air.
The grey programming language is made of challenges.
```

In der folgenden Version prüfen wir, ob alle “Wahrscheinlichkeiten” korrekt sind:

```
import random

def weighted_cartesian_choice(*iterables):
    """
    weighted_cartesian_choice liefert eine Liste mit
    gewichteten Auswahlen aus den iterierbaren Eingabeobjekten
    unter Wahrung der Reihenfolge.
    """
    res = []
    for population, weight in iterables:
        lst = weighted_choice(population, weight)
        res.append(lst)
    return res

determiners = (["The", "A", "Each", "Every", "No"],
               [0.3, 0.3, 0.1, 0.1, 0.2])
colours = (["red", "green", "blue", "yellow", "grey"],
           [0.1, 0.3, 0.3, 0.2, 0.2])
nouns = (["water", "elephant", "fish", "light", "programming language"],
         [0.3, 0.2, 0.1, 0.1, 0.3])
```

```
nouns2 = (["of happiness", "of chocolate", "of wisdom",
           "of challenges", "of air"],
          [0.5, 0.2, 0.1, 0.1, 0.1])
verb_phrases = (["smells", "dreams", "thinks", "is made of"],
          [0.4, 0.3, 0.2, 0.1])

print("It may or may not be true:")
sentences = []
for i in range(10000):
    res = weighted_cartesian_choice(determiners,
                                    colours,
                                    nouns,
                                    verb_phrases,
                                    nouns2)
    sentences.append(" ".join(res) + ".")

words = ["smells", "dreams", "thinks", "is made of"]
from collections import Counter
c = Counter()
for sentence in sentences:
    for word in words:
        if word in sentence:
            c[word] += 1

wsum = sum(c.values())
for key in c:
    print(key, c[key] / wsum)
```

```
It may or may not be true:
dreams 0.3057
thinks 0.1981
smells 0.3928
is made of 0.1034
```

Echte Zufallszahlen

Zur Erzeugung echter Zufallszahlen kann man physikalische Phänomene nutzen. Man kann sie aus radioaktiven Zerfallsprozessen oder dem Rauschen elektronischer Bauelemente gewinnen. Aber es geht auch einfacher. Beispielsweise durch Werfen einer oder mehrerer Münzen oder unter Benutzung von Würfeln. Diese Verfahren sind jedoch einerseits zeitaufwendig oder technisch kompliziert zu bewerkstelligen.

Wie wir bereits geschrieben haben, liefert das random-Modul von Python keine “echten” oder “wahren” Zufallszahlen. Die meisten Programme liefern nur Zufallszahlen, die pseudozufällig sind. Die Zahlen werden in einer vorhersehbaren Art generiert, weil der verwendete Algorithmus deterministisch ist. Pseudozufallszahlen sind für viele Situation gut genug, aber sie sind nicht “wirklich” zufällig, wenn man beispielsweise Würfel- oder Lotterie-Ziehungen simulieren will.

Die Webseite RANDOM.ORG erhebt den Anspruch “wahre” Zufallszahlen zu generieren. Sie nutzen die Zufälligkeit aus atmosphärischen Geräuschen. Diese Zahlen sind für viele Anwendungen besser als die pseudozufälligen Zahlen aus einem Algorithmus in Computer-Programmen.

Seed/Startwert

Unter einem "Seed Key" oder einfach nur Seed (deutsch wörtlich "Saatschlüssel") versteht man einen Wert, mit dem ein Zufallszahlengenerator initialisiert wird. Deshalb bezeichnet man ihn häufig auch als Startwert einer Zufallsfolge. Der Zufallszahlengenerator erzeugt mit der Seed als Startwert eine Folge von Zufallszahlen bzw. Pseudozufallszahlen. Startet man einen Algorithmus mit dem gleichen Seed-Wert, erhält man auch exakt die gleichen Pseudozufallszahlen.

Verwendet man Zufallszahlen in der Kryptographie, um damit verschlüsselte Daten zu erzeugen, so möchte man normalerweise nicht, dass die Seed erraten werden kann. In diesem Fall sollte die Seed ein Zufallswert sein. Diesen kann man beispielsweise in einem Programm durch das Hin- und Herbewegen einer Maus erzeugen.

Wenn wir `random.random()` aufrufen, so erwarten wir einen Zufallswert zwischen 0 und 1. `random.random()` berechnet einen neuen Zufallswert anhand des vorangegangenen Zufallswertes. Wie sieht es aber aus, wenn wir diese Funktion zum ersten Mal in unserem Programm verwenden? Genau, dann gibt es noch keinen vorangegangenen Zufallswert. Wenn ein Zufallgenerator zum ersten Mal aufgerufen wird, so muss der erste "Zufalls"-Wert irgendwie erzeugt werden.

Setzt man den Startwert, also die Seed, eines Pseudo-Zufallszahlen-Generators, so stellt man einen ersten Zufallswert zur Verfügung. Aus diesem wird dann deterministisch eine Sequenz von weiteren Zufallswerten generiert. Startet man wieder eine Zufallsfolge mit dem gleichen Startwert, so erhält man wieder exakt dieselbe Folge von Werten. Kennt man also den Startwert, kann man auch die aus diesem erzeugten verschlüsselten Werte berechnen.

Geht es also um Sicherheit, braucht man einen Weg, einen echten Zufallswert als Startwert zu verwenden.

Zufällige Startwerte werden in vielen Programmiersprachen anhand des Systemstatus generiert, der meistens der Systemzeit entspricht.

Für Python trifft dies ebenfalls zu. `help(random.seed)` sagt, dass wenn die Funktion mit `None` oder keinem Argument aufgerufen wird, der Seedwert aus der aktuellen Systemzeit oder einer anderen systemspezifischen Zufallsquelle generiert wird.

```
# prog4web
import random

help(random.seed)
```

```
Help on method seed in module random:

seed(a=None, version=2) method of random.Random instance
    Initialize internal state from hashable object.

    None or no argument seeds from current time or from an operating
    system specific randomness source if available.

    If *a* is an int, all bits are used.

    For version 2 (the default), all of the bits are used if *a* is a str,
    bytes, or bytearray.  For version 1 (provided for reproducing random
    sequences from older versions of Python), the algorithm for str and
    bytes generates a narrower range of seeds.
```

Die `seed`-Funktion liefert eine deterministische Sequenz von Zufallszahlen. Die Sequenz kann beliebig wiederholt werden, um beispielsweise bestimmte Situationen zu debuggen.

```
import random

random.seed(42)

for _ in range(10):
    print(random.randint(1, 10), end=", ")

print("\nNochmals die selben Zufallszahlen:")
random.seed(42)
for _ in range(10):
    print(random.randint(1, 10), end=", ")
```

```
2, 1, 5, 4, 4, 3, 2, 9, 2, 10,
Nochmals die selben Zufallszahlen:
2, 1, 5, 4, 4, 3, 2, 9, 2, 10,
```

Gauss'sche Normalverteilung

Bei `random.gauss` und `random.normalvariate` handelt es sich um zwei verschiedene Implementierungen von Funktionen, die jeweils normalverteilte Zufallszahlen zurückliefern.

Wir möchten nun 1000 Zufallszahlen zwischen 130 und 230 generieren die eine Gausssche Verteilung aufweisen mit dem Hauptwert `mu = 550` und eine Standardabweichung von `sigma = 30`.

Wenn wir uns die help-Information anschauen, sehen wir, dass der wesentliche Unterschied darin besteht, dass `gauss` etwas schneller ist und nicht Thread-sicher, während `random.normalvariate` Thread-sicher ist.

```
import random

help(random.normalvariate)
help(random.gauss)
```

```
Help on method normalvariate in module random:

normalvariate(mu, sigma) method of random.Random instance
    Normal distribution.

    mu is the mean, and sigma is the standard deviation.

Help on method gauss in module random:

gauss(mu, sigma) method of random.Random instance
    Gaussian distribution.

    mu is the mean, and sigma is the standard deviation.  This is
    slightly faster than the normalvariate() function.

    Not thread-safe without a lock around calls.
```

Wir berechnen uns nun mittels der Funktion `gauss` 1000 Zufallzahlen mit einem Mittelwert

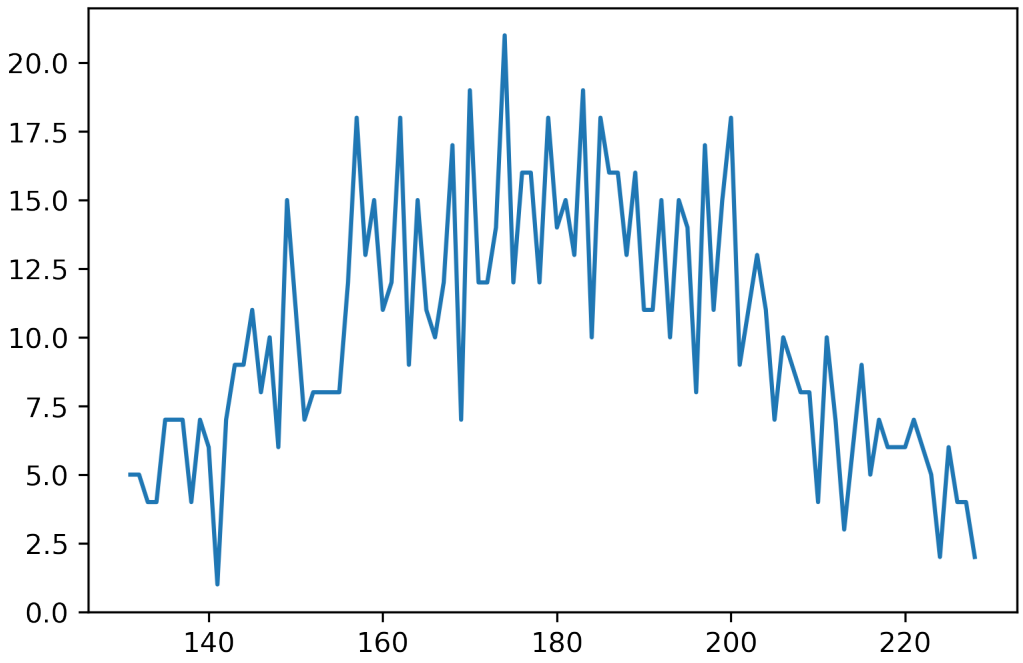
von 180 und einer Standardabweichung von 30:

```
from random import gauss

n = 1000

values = []
frequencies = {}

while len(values) < n:
    value = int(gauss(180, 30))
    if 130 < value < 230:
        frequencies[value] = frequencies.get(value, 0) + 1
        values.append(value)
values[:7]
```

```
[173, 183, 186, 214, 199, 183, 157]
```

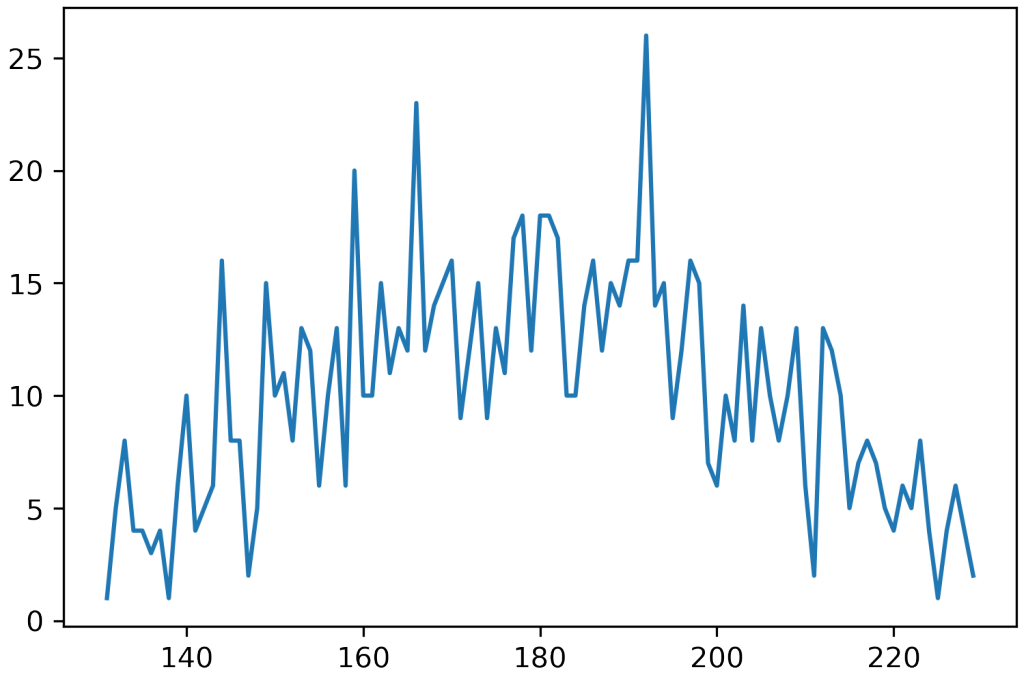
Das folgende Programm zeichnet die Zufallswerte, die wir soeben generiert haben. Wir haben das Modul `matplotlib` bis jetzt noch nicht behandelt. Das ist aber für das Verständnis des folgenden Codes nicht unbedingt notwendig:

```
import matplotlib.pyplot as plt

freq = list(frequencies.items())
freq.sort()


plt.plot(*list(zip(*freq)))
plt.show()
```



Wir können dies natürlich auch mit der Funktion `normalvariate` durchführen:

```
from random import normalvariate

n = 1000

values = []
frequencies = {}

while len(values) < n:
    value = int(normalvariate(180, 30))
    if 130 < value < 230:
        frequencies[value] = frequencies.get(value, 0) + 1
        values.append(value)

freq = list(frequencies.items())
freq.sort()


plt.plot(*list(zip(*freq)))
plt.show()
```



Übung mit Binärsender

Es ist vielleicht keine schlechte Idee, folgende Funktion als Übung selbst zu schreiben. Die Funktion soll mit einem Parameter p aufgerufen werden, der einen Wahrscheinlichkeitswert zwischen 0 und 1 beinhaltet. Die Funktion liefert eine 1 mit der Wahrscheinlichkeit von p, d.h. in p Prozent der Fälle werden 1en zurückgegeben und 0en in (1-p) Prozent der Fälle:

```
import random

def random_ones_and_zeros(p):
```

```
    """ p: probability 0 <= p <= 1
        returns a 1 with the probability p
    """
    x = random.random()
    if x < p:
        return 1
    else:
        return 0
```

Wir testen unsere kleine Funktion:

```
n = 1000000
sum(random_ones_and_zeros(0.8) for i in range(n)) / n
```

```
0.800606
```

Eine weitere gute Idee ist es, die Aufgabe mit einem Generator zu implementieren.

Sollten Sie nicht mit der Arbeitsweise eines Python-Generators vertraut sein, empfehlen wir unser Kapitel zu Generatoren und Iteratoren aus dem Python-Tutorial.

```
import random

def random_ones_and_zeros(p):
    while True:
        x = random.random()
        yield 1 if x < p else 0

def firstn(generator, n):
    for i in range(n):
        yield next(generator)
```

```
n = 1000000

firstn_values = firstn(random_ones_and_zeros(0.8), n)
sum(x for x in firstn_values) / n
```

```
0.799768
```

Unser 0en-und-1en-Generator kann wie ein Sender betrachtet werden, der 0en und 1en mit einer Wahrscheinlichkeit von p, respektive (1-p), abgibt.

Wir schreiben nun einen anderen Generator, der diesen Bitstrom empfängt. Die Aufgabe dieses neuen Generators ist es, den Bitstrom zu lesen und einen anderen Bitstrom aus 0en und 1en zu erzeugen mit einer Wahrscheinlichkeit von 0.5, ohne die Wahrscheinlichkeit p zu kennen. Es sollte für jeden beliebigen Wert p funktionieren.

1

```
def ebitter(bitstream):
    while True:
        bit1 = next(bitstream)
        bit2 = next(bitstream)
        if bit1 + bit2 == 1:
```

```
            bit3 = next(bitstream)
            if bit2 + bit3 == 1:
                yield 1
            else:
                yield 0
```

```
def ebitter2(bitstream):
    bit1 = next(bitstream)
    bit2 = next(bitstream)
    bit3 = next(bitstream)
    while True:
        if bit1 + bit2 == 1:
            if bit2 + bit3 == 1:
                yield 1
            else:
                yield 0
        bit1, bit2, bit3 = bit2, bit3, next(bitstream)
```

```
n = 1000000
sum(x for x in firstn(ebitter(random_ones_and_zeros(0.8)), n)) / n
```

```
0.499749
```

```
n = 1000000
sum(x for x in firstn(ebitter2(random_ones_and_zeros(0.8)), n)) / n
```

```
0.500011
```

Grundlagen der Theorie:

Unser erster Generator erzeugt einen Bitstrom B0, B1, B2,…

Wir prüfen nun ein beliebiges Paar aufeinanderfolgender Bits Bi, Bi+1,…

Ein solches Paar kann die Werte 01, 10, 00 oder 11 haben. Die Wahrscheinlichkeit p(01) = (p-1) x p und die Wahrscheinlichkeit p(10) = p x (p-1) ergibt eine kombinierte Wahrscheinlichkeit, dass die aufeinanderfolgenden Bits entweder 01 oder 10 sind, von 2 x (p-1) x p.

Betrachten wir ein anderes Bit Bi+2. Wie ist die Wahrscheinlichkeit dieser beiden

Bi + Bi+1 = 1

und

Bi+1 + Bi+2 = 1 ?

Die möglichen Ausgaben passen auf die Bedingungen, und die zugehörigen Wahrscheinlichkeiten sind in der folgenden Tabelle aufgeführt:

Wahrscheinlichkeit	Bi	Bi+1	Bi+2
p2 x (1-p)	0	1	0
p x (1 - p)2	1	0	1

Wir bezeichnen das Ergebnis sum(Bi, Bi+1)=1 als X1 und entsprechend dazu sum(Bi+1, Bi+2)=1 als X2.

Die verknüpfte Wahrscheinlichkeit P(X1, X2) = p2 x (1-p) + p x (1 - p)2 kann zu p x (1-p) umgestellt werden.

Die bedingte Wahrscheinlichkeit von X2 gibt X1:

P(X2 | X1) = P(X1, X2) / P(X2)

P(X2 | X1) = p x (1-p) / 2 x p x (1-p) = 1 / 2

Synthetische Verkaufszahlen

In diesem Unterkapitel geht es um die synthetische Erzeugung von Daten. Dazu erzeugen wir eine Datei mit Verkaufszahlen einer fiktiven Kette von Geschäften in diversen europäischen Städten.

Wir beginnen mit einem Array `sales` mit Verkaufszahlen für das Jahr 1997:

```
import numpy as np

sales = np.array([1245.89, 2220.00, 1635.77, 1936.25, 1002.03,
                  2099.13,  723.99, 990.37, 541.44, 1765.00,
                  1802.84, 1999.00])
```

Das Ziel ist, eine kommaseparierte Liste zu erzeugen, wie wir sie aus Excel kennen. Die Datei soll fiktive Verkaufszahlen für unsere nicht-existierenden Ladenlokale für die Jahre von 1997 bis 2018 enthalten.

Wir fügen für jedes Jahr den Verkaufszahlen noch zufällige Werte hinzu. Dafür konstruieren wir ein Array mit Wachstumsraten. Die Wachstumsraten können variieren zwischen einem minimalen Prozent-Wert (`min_percent`) und einem maximalen Prozent-Wert (`max_percent`):

```
min_percent = 0.98  # corresponds to 0.98 %
max_percent = 1.06   # 6 %

sample = np.random.random_sample(12)
print(sample)
growthrates = (max_percent - min_percent) * sample + min_percent
print(growthrates)
```

```
[0.76709447 0.31841101 0.64330855 0.13591889 0.09436673 0.2097398
 0.02373195 0.25975183 0.87423196 0.32085012 0.06566312 0.04253395]
[1.04136756 1.00547288 1.03146468 0.99087351 0.98754934 0.99677918
 0.98189856 1.00078015 1.04993856 1.00566801 0.98525305 0.98340272]
```

Für die neuen Verkaufszahlen nach einem Jahr multiplizieren wir das `sales`-Array mit dem `growthrates`-Array:

```
sales * growthrates
```

```
array([1297.42942665, 2232.14979487, 1687.2389867 , 1918.57883625,
        989.55406327, 2092.3690888 ,  710.88473563,  991.14263363,
        568.47873238, 1775.00403702, 1776.25360838, 1965.82202987])
```

Um eine nachhaltige Verkaufsentwicklung zu erhalten, verändern wir die Wachstums-Raten alle 5 Jahre.

Das ist unser komplettes Programm, welches die Daten in der Datei `sales_figures.csv` ablegt:

```
import numpy as np
fh = open("sales_figures.csv", "w")

fh.write("Year, Frankfurt, Munich, Berlin, Zurich, Hamburg, \
         London, Paris, Luxembourg, Wien, Amsterdam, \
         Rotterdam, The Hague\n")
sales = np.array([1245.89, 2220.00, 1635.77, 1936.25, 1002.03,
                  2099.13,  723.99, 990.37, 541.44, 1765.00,
                  1802.84, 1999.00])

for year in range(1997, 2018):
    line = str(year) + ", " + ", ".join(map(str, sales))
    fh.write(line + "\n")
    if year % 4 == 0:
         min_percent = 0.98  # corresponds to -1.5 %
         max_percent = 1.06   # 6 %
         sample = np.random.random_sample(12)
         growthrates = (max_percent - min_percent) * sample + min_percent
    sales = np.around(sales * growthrates, 2)
fh.close()
```

Das Ergebnis ist in der Datei `sales_figures.csv` zu finden.

Wir werden diese Daten in einem folgenden Kapitel (Lesen und Schreiben in NumPy) noch benutzen.

```
# prog4book"
from random import randint

outcomes = [ randint(1, 6) for _ in range(10000)]

even_pips = [ x for x in outcomes if x % 2 == 0]
greater_two = [ x for x in outcomes if x > 2]

combined = [ x for x in outcomes if x % 2 == 0 and x > 2]

print(len(even_pips) / len(outcomes))
print(len(greater_two) / len(outcomes))
print(len(combined) / len(outcomes))
```

```
0.5057
0.6716
0.3397
```

Lösung 2. Aufgabe Wir schreiben zuerst die Funktion “process_datafile”, um die Daten aus der Datei zu verarbeiten:

```
# prog4book"
def process_datafile(filename):
```

```
    """ process_datafile -> (universities,
                             enrollments,
                             total_number_of_students)
        universities: list of University names
        enrollments: corresponding list with enrollments
        total_number_of_students: over all universities
    """

    universities = []
    enrollments = []
    with open(filename) as fh:
        total_number_of_students = 0
        fh.readline() # get rid of descriptive first line
        for line in fh:
             line = line.strip()
             *praefix, under, post, total = line.rsplit()
             university = praefix[1:]
             total = int(total.replace(",", ""))
             enrollments.append(total)
             universities.append(" ".join(university))
             total_number_of_students += total
    return (universities, enrollments, total_number_of_students)
```

Lassen wir die Funktion laufen und prüfen das Ergebnis:

```
# prog4book"
universities, enrollments, total_students = \
                      process_datafile("universities_uk.txt")

for i in range(14):
    print(universities[i], end=": ")
    print(enrollments[i])
print("Number of students enrolled in the UK: ", total_students)
```

```
Open University in England: 123490
University of Manchester: 37925
University of Nottingham: 33270
Sheffield Hallam University: 33100
University of Birmingham: 32335
Manchester Metropolitan University: 32160
University of Leeds: 30975
Cardiff University: 30180
University of South Wales: 29195
University College London: 28430
King's College London: 27645
University of Edinburgh: 27625
Northumbria University: 27565
University of Glasgow: 27390
Number of students enrolled in the UK:  2299380
```

Wir wollen einen virtuellen Studenten in eine zufällige Universität einschreiben. Um eine gewichtete Liste zu erhalten, die für die `weighted_choice` geeignet ist, müssen wir die Werte der Liste `enrollments` normalisieren:

```
# prog4book"
normalized_enrollments = [ students / total_students \
```

```
                            for students in enrollments]

# enrolling a virtual student:
print(weighted_choice(universities, normalized_enrollments))
```

```
University of Plymouth
```

Die Aufgabe war 100,000 fiktive Studenten zu "immatrikulieren". Dies kann mit einer Schleife einfach durchgeführt werden:

```
# prog4book"
from collections import Counter
from pprint import pprint # schöne Print-Ausgabe

outcomes = []
n = 100000
for i in range(n):
    outcomes.append(weighted_choice(universities,
                                    normalized_enrollments))

c = Counter(outcomes)

pprint(c.most_common(20), indent=2, width=70)
```

```
[ ('Open University in England', 5385),
  ('University of Manchester', 1635),
  ('University of Birmingham', 1491),
  ('Sheffield Hallam University', 1445),
  ('University of Nottingham', 1428),
  ('Manchester Metropolitan University', 1381),
  ('University of Leeds', 1349),
  ('Cardiff University', 1314),
  ('University of South Wales', 1276),
  ('University of Central Lancashire', 1262),
  ('University of Plymouth', 1257),
  ('Nottingham Trent University', 1254),
  ('University of Glasgow', 1245),
  ('University College London', 1185),
  ('Northumbria University', 1164),
  ('University of Edinburgh', 1163),
  ('University of the West of England', 1141),
  ('Ulster University', 1136),
  ("King's College London", 1134),
  ('University of Sheffield', 1129)]
```

Lösung 3. Aufgabe Die Sammlung der Amazonen ist als Liste implementiert, während für die Menge aus Pysseusses Favoritinnen auswählen. Die Gewichtung liegt zu Beginn bei 1/11 für alle, d.h. 1/len(amazons).

Jeder Schleifen-Durchlauf entspricht einem neuen Tag. Jedes Mal, wenn wir einen neuen Durchlauf starten, ziehen wir `n` Samples aus den Pythoniern, um das Verhältnis zu berechnen, wie oft die Sample gleich den Favoritinnen des Königs, geteilt durch die Häufigkeit, wie oft die Sample nicht der Idee einer Schwiegertochter entspricht. Dies entspricht der Wahrscheinlichkeit `prob`. Wir stoppen das erste Mal, wenn die Wahrscheinlichkeit bei 0.9 oder größer liegt.

Die beiden Funktionen `weighted_same` und `weighted_same_alternative` lassen sich für die Ziehung verwenden.

```
import time

amazons = ["Airla", "Barbara", "Eos",
           "Glykeria", "Hanna", "Helen",
           "Agathangelos", "Iokaste",
           "Medousa", "Sofronia",
           "Andromeda"]

weights = [ 1/len(amazons) for _ in range(len(amazons)) ]

Pytheusses_favorites = {"Iokaste", "Medousa",
                        "Sofronia", "Andromeda"}
n = 1000
counter = 0

prob = 1 / 330
days = 0
factor1 = 1 / 13
factor2 = 1 / 12

start = time.perf_counter()
while prob < 0.9:
    for i in range(n):
        the_chosen_ones = weighted_sample_alternative(amazons,
                                                      weights,
                                                      4)
        if set(the_chosen_ones) == Pytheusses_favorites:
            counter += 1
    prob = counter / n
    counter = 0
    weights[:7] = [ p - p*factor1 for p in weights[:7] ]
    weights[7:] = [ p + p*factor2 for p in weights[7:] ]
    weights = [ x / sum(weights) for x in weights]
    days += 1
print(time.perf_counter() - start)

print("Number of days, he has to wait: ", days)
```

```
2.0128858579555526
Number of days, he has to wait:  32
```

Die Werte für die Anzahl der Tage weichen ab, wenn n nicht groß genug ist.

Der folgende Code ist die Lösung ohne Rundungsfehler. Wir verwenden `Fraction` aus dem Modul `fractions`.

```
import time
from fractions import Fraction


amazons = ["Airla", "Barbara", "Eos",
           "Glykeria", "Hanna", "Helen",
           "Agathangelos", "Iokaste",
           "Medousa", "Sofronia",
```

```
           "Andromeda"]

weights = [ Fraction(1, 11) for _ in range(len(amazons)) ]

Pytheusses_favorites = {"Iokaste", "Medousa",
                        "Sofronia", "Andromeda"}
n = 1000
counter = 0

prob = Fraction(1, 330)
days = 0
factor1 = Fraction(1, 13)
factor2 = Fraction(1, 12)

start = time.perf_counter()
while prob < 0.9:
    #print(prob)
    for i in range(n):
        the_chosen_ones = weighted_sample_alternative(amazons, weights, 4)
        if set(the_chosen_ones) == Pytheusses_favorites:
            counter += 1
    prob = Fraction(counter, n)
    counter = 0
    weights[:7] = [ p - p*factor1 for p in weights[:7] ]
    weights[7:] = [ p + p*factor2 for p in weights[7:] ]
    weights = [ x / sum(weights) for x in weights]
    days += 1
print(time.perf_counter() - start)

print("Number of days, he has to wait: ", days)
```

```
20.071284992969595
Number of days, he has to wait:  32
```

Wir können sehen, dass die Lösung mit `fractions` schön aber langsam ist. Dabei spielt die Präzision in unserem Fall keine Rolle.

Jedoch haben wir die Leistung von Python nicht genutzt. Das machen wir in der nächsten Implementierung:

```
import time
import numpy as np

amazons = ["Airla", "Barbara", "Eos",
           "Glykeria", "Hanna", "Helen",
           "Agathangelos", "Iokaste",
           "Medousa", "Sofronia",
           "Andromeda"]

weights = np.full(11, 1/len(amazons))


Pytheusses_favorites = {"Iokaste", "Medousa",
                        "Sofronia", "Andromeda"}


n = 1000
```

```
counter = 0

prob = 1 / 330
days = 0
factor1 = 1 / 13
factor2 = 1 / 12

start = time.perf_counter()
while prob < 0.9:
    for i in range(n):
        the_chosen_ones = weighted_sample_alternative(amazons,
                                                      weights,
                                                      4)
        if set(the_chosen_ones) == Pytheusses_favorites:
            counter += 1
    prob = counter / n
    counter = 0
    weights[:7] = weights[:7] - weights[:7] * factor1
    weights[7:] = weights[7:] + weights[7:] * factor2
    weights = weights / np.sum(weights)
    #print(weights)
    days += 1
print(time.perf_counter() - start)

print("Number of days, he has to wait: ", days)
```

```
2.0671991429990157
Number of days, he has to wait:  32
```

Fußnoten:

1 Ich bedanke mich bei Dr. Hanno Baehr der mich auf das Problem der “Zufalls-Extrahierung” aufmerksam gemacht hat bei der Teilnahme eine Python-Training-Kurses in Nürnberg im Januar 2014. Hanno hat ein paar Bits des theoretischen Rahmens entworfen. Während einer Nacht-Session in der Bar “Zeit & Raum” (engl. “Time & Space”) habe ich ein entsprechendes Python-Programm implementiert um die theoretische Lösung empirisch zu unterstützen.

8 Synthetische Testdatenerzeugung Mit Python

8.0.1 Synthetische Testdaten mit Python

Definition synthetischer Daten

Es wird kaum einen Ingenieur oder Wissenschaftler geben, der nicht die Notwendigkeit kennt, synthetische Daten zu erzeugen. Aber einige fragen sich sicherlich, was denn mit "synthetischen Testdaten" gemeint ist. Es gibt viele Situationen, in denen ein Wissenschaftler oder ein Ingenieur Testdaten braucht, es aber nahezu unmöglich ist, an "richtige" Daten heranzukommen, z.B. an eine Stichprobe aus einer Population, die aus Messungen gewonnen wurde. Die Aufgabe bei der Erstellung von synthetischen Daten besteht darin, Daten zu produzieren, die den "richtigen Daten" sehr nahe kommen. Python ist eine ideale Sprache, um auf einfache Weise solche Daten zu produzieren, weil sie starke numerische und linguistische Funktionen bereitstellt.

Synthetische Daten sind ebenfalls notwendig, um spezielle Bedürfnisse oder bestimmte Bedingungen zu erfüllen, die nicht oder nicht häufig genug in "echten Daten" vorkommen.

Im vorigen Kapitel "Python, NumPy und Wahrscheinlichkeiten" haben wir einige Funktionen beschrieben, die wir im folgenden benötigen:

- find_interval
- weighted_choice
- cartesian_choice
- weighted_cartesian_choice
- weighted_sample

Sie sollten mit der Funktionsweise dieser Funktionen vertraut sein.

Wir haben die Funktionen im Modul mit dem Namen bk_random abgelegt.

Definition des Umfangs für die synthetische Datenerzeugung

Wir möchten Lösungen vorstellen für die folgende Aufgabe:

Wir haben n endliche Mengen die Daten verschiedener Typen enthalten:

D1, D2, … Dn

Die Mengen Di sind die Daten-Mengen aus denen wir unsere synthetischen Daten herleiten wollen.

In der aktuellen Implementierung sind die Mengen Tupels oder Listen, um es praktisch zu halten.

Der Prozess der synthetischen Datenerzeugung kann in den zwei Funktionen "synthesi-

zer" und "synthesize" definiert werden. Den Begriff "Synthesizer" wird normalerweise für ein computergestütztes Gerät verwendet, welches Sound produziert. Unser "Synthesizer" prouziert Strings oder alternativ Tupels mit Daten, wie wir später sehen werden.

Die Funktion "synthesizer" erstellt die Funktion "synthesize":

synthesize = synthesizer((D1, D2, … Dn))

Die Funktion "synthesize", - die in unserer Implementierung auch ein Generator sein kann, - erwartet kein Argument und das Ergebnis des Funktions-Aufrufs von "synthesize()" ist

- eine Liste oder ein Tupel t = (d1, d2, … dn), wobei di zufällig aus Di gezogen wurde
- oder ein String der die Elemente str(d1), str(d2), … str(dn) enthält, bei denen di ebenfalls zufällig aus Di gezogen wurde.

Lassen Sie uns mit einem einfachen Beispiel starten. Wir haben eine Liste mit Vornamen und eine Liste mit Nachnamen. Wir möchten weitere Mitarbeiter anheuern für ein Firma. Natürlich ist es einfacher einen Spezialisten in unserer synthetischen Umgebung anzuheuern als jemanden im richtigen Leben. Alles was benötigt ist die Funktion "cartesian_choice" aus dem bk_random-Modul und die Verkettung der zufällig gezogenen Vornamen und Nachnamen.

```
import bk_random

firstnames = ["John", "Eve", "Jane", "Paul",
              "Frank", "Laura", "Robert",
              "Kathrin", "Roger", "Simone",
              "Bernard", "Sarah", "Yvonne"]
surnames = ["Singer", "Miles", "Moore",
            "Looper", "Rampman", "Chopman",
            "Smiley", "Bychan", "Smith",
            "Baker", "Miller", "Cook"]

number_of_specialists = 15

employees = set()
while len(employees) < number_of_specialists:
    employee = bk_random.cartesian_choice(firstnames, surnames)
    employees.add(" ".join(employee))

print(employees)
```

```
{'Jane Smiley', 'Simone Rampman', 'Sarah Moore', 'Laura Miles', 'Roger Rampman',
↪  'Eve Cook', 'Roger Looper', 'Eve Moore', 'Paul Rampman', 'Robert Smith',
↪  'Simone Chopman', 'Sarah Smiley', 'Bernard Chopman', 'Simone Looper', 'John
↪  Chopman'}
```

Das war einfach genug, aber wir möchten dies nun mehr strukturieren, indem wir den bereits erwähnten Synthesizer verwenden. Im folgenden Code fehlt noch die Implementierung für den Fall, dass der Parameter "weights" nicht None ist:

```
import bk_random

firstnames = ["John", "Eve", "Jane", "Paul",
              "Frank", "Laura", "Robert",
              "Kathrin", "Roger", "Simone",
```

```
              "Bernard", "Sarah", "Yvonne"]
surnames = ["Singer", "Miles", "Moore",
            "Looper", "Rampman", "Chopman",
            "Smiley", "Bychan", "Smith",
            "Baker", "Miller", "Cook"]

def synthesizer( data, weights=None, format_func=None, repeats=True):
    """
    data is a tuple or list of lists or tuples containing the
    data
    weights is a list or tuple of lists or tuples with the
    corresponding weights of the data lists or tuples
    format_func is a reference to a function which defines
    how a random result of the creator function will be formated.
    If None, "creator" will return the list "res".
    If repeats is set to True, the results of helper will not be unique
    """
    if not repeats:
        memory = set()
    def synthesize():
        while True:
            res = bk_random.cartesian_choice(*data)
            if not repeats:
                sres = str(res)
                while sres in memory:
                    res = bk_random.cartesian_choice(*data)
                    sres = str(res)
                memory.add(sres)
            if format_func:
                yield format_func(res)
            else:
                yield res
    return synthesize

recruit_employee = synthesizer( (firstnames, surnames),
                                format_func=lambda x: " ".join(x),
                                repeats=False)

employee = recruit_employee()
for _ in range(15):
    print(next(employee))
```

```
Yvonne Rampman
Yvonne Singer
Frank Bychan
Yvonne Chopman
Frank Cook
Robert Bychan
Eve Bychan
Jane Baker
Jane Miller
Kathrin Singer
Roger Baker
Roger Miles
Laura Miles
```

```
Frank Baker
Paul Smiley
```

Im vorigen Beispiel hat jeder Name, d.h. Vor- und Nachname, die gleiche Wahrscheinlichkeit gezogen zu werden. Das ist nicht wirklich realisitsch, weil wir in Ländern wie US oder England Namen wie “Smith” oder “Miller” öfter erwarten als Namen wie “Rampman” oder “Bychan”. Wir erweitern unsere synthesizer-Funktion um zusätzlichen Code für den “gewichteten” Fall, d.h. “weights” ist nicht None. Wenn Gewichtungen (weights) gegeben sind, müssen wir die Funktion weighted_cartesian_choice aus dem Modul bk_random verwenden. Wenn “weights” auf None gesetzt ist, müssen wir die Funktion cartesian_choice verwenden. Für lagern diese Entscheidung in eine weitere Unterfunktion des Synthesizers aus diese sauberer zu halten.

Wir wollen an dieser Stelle nicht mit Wahrscheinlichkeiten zwischen 0 und 1 umher werfen um die Gewichtungen zu definieren. Also nehmen wir den Umweg über Integer-Werte, die wir nachher normalisieren.

```
from bk_random import cartesian_choice, weighted_cartesian_choice

weighted_firstnames = [ ("John", 80), ("Eve", 70), ("Jane", 2),
                        ("Paul", 8), ("Frank", 20), ("Laura", 6),
                        ("Robert", 17), ("Zoe", 3), ("Roger", 8),
                        ("Simone", 9), ("Bernard", 8), ("Sarah", 7),
                        ("Yvonne", 11), ("Bill", 12), ("Bernd", 10)]

weighted_surnames = [('Singer', 2), ('Miles', 2), ('Moore', 5),
                     ('Looper', 1), ('Rampman', 1), ('Chopman', 1),
                     ('Smiley', 1), ('Bychan', 1), ('Smith', 150),
                     ('Baker', 144), ('Miller', 87), ('Cook', 5),
                     ('Joyce', 1), ('Bush', 5), ('Shorter', 6),
                     ('Klein', 1)]


firstnames, weights = zip(*weighted_firstnames)
wsum = sum(weights)
weights_firstnames = [ x / wsum for x in weights]

surnames, weights = zip(*weighted_surnames)
wsum = sum(weights)
weights_surnames = [ x / wsum for x in weights]

weights = (weights_firstnames, weights_surnames)


def synthesizer( data, weights=None, format_func=None, repeats=True):
    """
    "data" is a tuple or list of lists or tuples containing the
    data.

    "weights" is a list or tuple of lists or tuples with the
    corresponding weights of the data lists or tuples.

    "format_func" is a reference to a function which defines
    how a random result of the creator function will be formated.
    If None,the generator "synthesize" will yield the list "res".
```

```
    If "repeats" is set to True, the output values yielded by
    "synthesize" will not be unique.
    """

    if not repeats:
        memory = set()

    def choice(data, weights):
        if weights:
            return weighted_cartesian_choice(*zip(data, weights))
        else:
            return cartesian_choice(*data)
    def synthesize():
        while True:
            res = choice(data, weights)
            if not repeats:
                sres = str(res)
                while sres in memory:
                    res = choice(data, weights)
                    sres = str(res)
                memory.add(sres)
            if format_func:
                yield format_func(res)
            else:
                yield res
    return synthesize

recruit_employee = synthesizer( (firstnames, surnames),
                                weights = weights,
                                format_func=lambda x: " ".join(x),
                                repeats=False)

employee = recruit_employee()
for _ in range(8):
    print(next(employee))
```

```
John Miller
Eve Baker
Yvonne Smith
John Baker
Roger Baker
Robert Smith
John Smith
Zoe Baker
```

Wein Beispiel

Stellen Sie sich vor, dass Sie ein Dutzend Weine beschreiben müssen. Sehr wahrscheinlich für die meisten eine schöne Vorstellung, allerdings nicht für mich. Der Hauptgrund: “Ich bein kein Weintrinker!”

Wir können eine kleines Python-Programm schreiben, die unsere synthesize-Funktion benutzt um automatisch “anspruchsvolle Kritiken” zu schreiben, wie z.B.:

This wine is light-bodied with a conveniently juicy bouquet leading to a lingering flamboyant finish!

Finden Sie ein paar Adverben, wie “nahtlos”, “bestimmend”, und ein paar Adjektive wie “fruchtig” und “veredelt” um das Aroma zu beschreiben.

Wenn Sie ihre Liste definiert haben, können Sie die synthesize-Funktion verwenden.

Sollten Sie es nicht selbst machen wollen, so ist hier unsere Lösung:

```
import bk_random

body = ['light-bodied', 'medium-bodied', 'full-bodied']

adverbs = ['appropriately', 'assertively', 'authoritatively',
           'compellingly', 'completely', 'continually',
           'conveniently', 'credibly', 'distinctively',
           'dramatically', 'dynamically', 'efficiently',
           'energistically', 'enthusiastically', 'fungibly',
           'globally', 'holisticly', 'interactively',
           'intrinsically', 'monotonectally', 'objectively',
           'phosfluorescently', 'proactively', 'professionally',
           'progressively', 'quickly', 'rapidiously',
           'seamlessly', 'synergistically', 'uniquely']

noun = ['aroma', 'bouquet', 'flavour']

aromas = ['angular', 'bright', 'lingering', 'butterscotch',
          'buttery', 'chocolate', 'complex', 'earth', 'flabby',
          'flamboyant', 'fleshy', 'flowers', 'food friendly',
          'fruits', 'grass', 'herbs', 'jammy', 'juicy', 'mocha',
          'oaked', 'refined', 'structured', 'tight', 'toast',
          'toasty', 'tobacco', 'unctuous', 'unoaked', 'vanilla',
          'velvetly']

example = """This wine is light-bodied with a completely buttery
bouquet leading to a lingering fruity  finish!"""

def describe(data):
    body, adv, adj, noun, adj2 = data
    format_str = "This wine is %s with a %s %s %s\nleading to"
    format_str += " a lingering %s finish!"
    return format_str % (body, adv, adj, noun, adj2)

t = bk_random.cartesian_choice(body, adverbs, aromas, noun, aromas)

data = (body, adverbs, aromas, noun, aromas)
synthesize = synthesizer( data, weights=None, format_func=describe,
↪  repeats=True)
criticism = synthesize()

for i in range(1, 13):
    print("{0:d}. wine:".format(i))
    print(next(criticism))
    print()
```

```
1. wine:
This wine is full-bodied with a continually structured bouquet
leading to a lingering toast finish!
```

```
2. wine:
This wine is light-bodied with a interactively mocha flavour
leading to a lingering angular finish!

3. wine:
This wine is full-bodied with a monotonectally lingering bouquet
leading to a lingering tight finish!

4. wine:
This wine is medium-bodied with a interactively angular flavour
leading to a lingering tobacco finish!

5. wine:
This wine is light-bodied with a dramatically flowers aroma
leading to a lingering tight finish!

6. wine:
This wine is light-bodied with a progressively juicy flavour
leading to a lingering toasty finish!

7. wine:
This wine is light-bodied with a credibly bright bouquet
leading to a lingering unoaked finish!

8. wine:
This wine is light-bodied with a dramatically flabby bouquet
leading to a lingering tobacco finish!

9. wine:
This wine is light-bodied with a progressively bright bouquet
leading to a lingering jammy finish!

10. wine:
This wine is medium-bodied with a rapidiously butterscotch flavour
leading to a lingering tobacco finish!

11. wine:
This wine is light-bodied with a professionally velvetly flavour
leading to a lingering jammy finish!

12. wine:
This wine is light-bodied with a proactively bright bouquet
leading to a lingering unctuous finish!
```

Übung: Internationale Katastrophen-Operation

Es wäre großartig, wenn die in dieser Übung beschriebenen Probleme wirklich künstlich bleiben. Komplett unrealistisch, aber ein schöner Tagtraum. Jedoch, die Aufgabe dieser Übung ist es, synthetische Test-Daten bereitszustellen für eine internationale Katastrophen-Operation. Die Länder, die an dieser Operation beteiligt sind, sind z.B. Frankreich, Schweiz, Deutschland, Kanada, die Niederlande, die vereinigten Staaten, Österreich, Belgien und Luxemburg.

Wir möchten eine Datei erstellen mit zufälligen Einträgen verschiedener Berater. Jede Zeile sollte folgende Informationen beinhalten:

eindeutige Identifikation, Vorname, Nachname, Land, Fachbereich

Beispiel:

Aus praktischen Gründen reduzieren wir in der folgenden Beispiel-Implementierung die Länder auf Frankreich, Schweiz und Deutschland:

```
from bk_random import cartesian_choice, weighted_cartesian_choice

countries = ["France", "Switzerland", "Germany"]

w_firstnames = { "France" : [ ("Marie", 10), ("Thomas", 10),
                              ("Camille", 10), ("Nicolas", 9),
                              ("Léa", 10), ("Julien", 9),
                              ("Manon", 9), ("Quentin", 9),
                              ("Chloé", 8), ("Maxime", 9),
                              ("Laura", 7), ("Alexandre", 6),
                              ("Clementine", 2), ("Grégory", 2),
                              ("Sandra", 1), ("Philippe", 1)],
                 "Switzerland": [ ("Sarah", 10), ("Hans", 10),
                              ("Laura", 9), ("Peter", 8),
                              ("Mélissa", 9), ("Walter", 7),
                              ("Océane", 7), ("Daniel", 7),
                              ("Noémie", 6), ("Reto", 7),
                              ("Laura", 7), ("Bruno", 6),
                              ("Eva", 2), ("Urli", 4),
                              ("Sandra", 1), ("Marcel", 1)],
                 "Germany": [ ("Ursula", 10), ("Peter", 10),
                              ("Monika", 9), ("Michael", 8),
                              ("Brigitte", 9), ("Thomas", 7),
                              ("Stefanie", 7), ("Andreas", 7),
                              ("Maria", 6), ("Wolfgang", 7),
                              ("Gabriele", 7), ("Manfred", 6),
                              ("Nicole", 2), ("Matthias", 4),
                              ("Christine", 1), ("Dirk", 1)] }

w_surnames = { "France" : [ ("Matin", 10), ("Bernard", 10),
                            ("Camille", 10), ("Nicolas", 9),
                            ("Dubois", 10), ("Petit", 9),
                              ("Durand", 8), ("Leroy", 8),
                              ("Fournier", 7), ("Lambert", 6),
                              ("Mercier", 5), ("Rousseau", 4),
                              ("Mathieu", 2), ("Fontaine", 2),
                              ("Muller", 1), ("Robin", 1)],
                 "Switzerland": [ ("Müller", 10), ("Meier", 10),
                              ("Schmid", 9), ("Keller", 8),
                              ("Weber", 9), ("Huber", 7),
                              ("Schneider", 7), ("Meyer", 7),
                              ("Steiner", 6), ("Fischer", 7),
                              ("Gerber", 7), ("Brunner", 6),
                              ("Baumann", 2), ("Frei", 4),
                              ("Zimmermann", 1), ("Moser", 1)],
                 "Germany": [ ("Müller", 10), ("Schmidt", 10),
                              ("Schneider", 9), ("Fischer", 8),
                              ("Weber", 9), ("Meyer", 7),
                              ("Wagner", 7), ("Becker", 7),
                              ("Schulz", 6), ("Hoffmann", 7),
                              ("Schäfer", 7), ("Koch", 6),
                              ("Bauer", 2), ("Richter", 4),
```

```
                              ("Klein", 2), ("Schröder", 1)] }

# separate names and weights
synthesize = {}
identifier = 1
for country in w_firstnames:
    firstnames, weights = zip(*w_firstnames[country])
    wsum = sum(weights)
    weights_firstnames = [ x / wsum for x in weights]
    w_firstnames[country] = [firstnames, weights_firstnames]

    surnames, weights = zip(*w_surnames[country])
    wsum = sum(weights)
    weights_surnames = [ x / wsum for x in weights]
    w_surnames[country] = [surnames, weights_firstnames]

    synthesize[country] = synthesizer( (firstnames, surnames),
                                       (weights_firstnames,
                                        weights_surnames),
                                format_func=lambda x: " ".join(x),
                                repeats=False)
nation_prob = [("Germany", 0.3),
               ("France", 0.3),
               ("Switzerland", 0.4)]
profession_prob = [("Medical Aid", 0.3),
                   ("Social Worker", 0.5),
                   ("Security Aid", 0.2)]

helpers = []
for _ in range(50):
    country = weighted_cartesian_choice(zip(*nation_prob))
    profession = weighted_cartesian_choice(zip(*profession_prob))
    country, profession = country[0], profession[0]
    s = synthesize[country]()
    uid = "{id:05d}".format(id=identifier)
    helpers.append((uid, country, next(s), profession ))
    identifier += 1

print(helpers)
```

```
[('00001', 'Switzerland', 'Laura Fischer', 'Security Aid'), ('00002', 'Germany',
↪  'Michael Meyer', 'Social Worker'), ('00003', 'France', 'Alexandre Petit',
↪  'Social Worker'), ('00004', 'Switzerland', 'Hans Schmid', 'Medical Aid'),
↪  ('00005', 'Germany', 'Maria Schmidt', 'Social Worker'), ('00006', 'Germany',
↪  'Peter Müller', 'Social Worker'), ('00007', 'France', 'Alexandre Nicolas',
↪  'Social Worker'), ('00008', 'Switzerland', 'Walter Schmid', 'Social Worker'),
↪  ('00009', 'Switzerland', 'Laura Weber', 'Security Aid'), ('00010', 'Germany',
↪  'Gabriele Bauer', 'Social Worker'), ('00011', 'Switzerland', 'Bruno Fischer',
↪  'Medical Aid'), ('00012', 'Germany', 'Stefanie Meyer', 'Social Worker'),
↪  ('00013', 'Switzerland', 'Hans Gerber', 'Social Worker'), ('00014',
↪  'Germany', 'Brigitte Weber', 'Medical Aid'), ('00015', 'Germany', 'Wolfgang
↪  Müller', 'Social Worker'), ('00016', 'Switzerland', 'Noémie Keller', 'Social
↪  Worker'), ('00017', 'France', 'Alexandre Mercier', 'Social Worker'),
↪  ('00018', 'Switzerland', 'Walter Frei', 'Social Worker'), ('00019',
↪  'Germany', 'Brigitte Richter', 'Social Worker'), ('00020', 'Switzerland',
↪  'Marcel Keller', 'Medical Aid'), ('00021', 'Switzerland', 'Hans Fischer',
↪  'Security Aid'), ('00022', 'Switzerland', 'Urli Schneider', 'Medical Aid'),
↪  ('00023', 'Switzerland', 'Daniel Meier', 'Social Worker'), ('00024',
↪  'Germany', 'Wolfgang Schneider', 'Social Worker'), ('00025', 'Switzerland',
↪  'Bruno Steiner', 'Medical Aid'), ('00026', 'France', 'Marie Matin', 'Social
↪  Worker'), ('00027', 'France', 'Léa Mercier', 'Medical Aid'), ('00028',
↪  'Switzerland', 'Mélissa Brunner', 'Medical Aid'), ('00029', 'Germany',
↪  'Monika Schulz', 'Security Aid'), ('00030', 'France', 'Maxime Durand',
↪  'Security Aid'), ('00031', 'France', 'Manon Durand', 'Medical Aid'),
↪  ('00032', 'France', 'Alexandre Dubois', 'Social Worker'), ('00033',
↪  'Germany', 'Ursula Koch', 'Security Aid'), ('00034', 'Germany', 'Ursula
↪  Schulz', 'Security Aid'), ('00035', 'Germany', 'Peter Koch', 'Social
↪  Worker'), ('00036', 'Switzerland', 'Hans Meyer', 'Social Worker'), ('00037',
↪  'Switzerland', 'Hans Huber', 'Medical Aid'), ('00038', 'France', 'Léa
↪  Dubois', 'Security Aid'), ('00039', 'France', 'Nicolas Leroy', 'Social
↪  Worker'), ('00040', 'Switzerland', 'Reto Müller', 'Security Aid'), ('00041',
↪  'Switzerland', 'Noémie Weber', 'Social Worker'), ('00042', 'Germany',
↪  'Stefanie Richter', 'Social Worker'), ('00043', 'Germany', 'Michael
↪  Schneider', 'Security Aid'), ('00044', 'France', 'Laura Nicolas', 'Security
↪  Aid'), ('00045', 'France', 'Chloé Mathieu', 'Social Worker'), ('00046',
↪  'Switzerland', 'Daniel Huber', 'Social Worker'), ('00047', 'Switzerland',
↪  'Sarah Fischer', 'Medical Aid'), ('00048', 'Germany', 'Matthias Richter',
↪  'Medical Aid'), ('00049', 'Switzerland', 'Walter Schneider', 'Social
↪  Worker'), ('00050', 'Switzerland', 'Laura Huber', 'Security Aid')]
```

9 Python Numpy Maskierung

9.0.1 Boolesche Maskierung und Indizierung

In diesem Kapitel geht es um die Boolesche Maskierung (englisch: Boolean Masking) und Booleschen Masken. Wir zeigen, wie man damit die Werte von NumPy-Arrays verändern kann.
Maskierung ist hilfreich, um Daten mit bestimmten Eigenschaften zu extrahieren, zu verändern, zu zählen und so weiter. Außerdem können damit auch sehr leicht einfache Binarisierungen vorgenommen werden, also alle Werte, die über einer bestimmten Schwelle liegen, auf einen Wert setzen und alle darunter liegenden Werte auf einen anderen. Die Benutzung von Maskierungen gestaltet sich meistens nicht nur sehr einfach, sondern dabei handelt es sich meistens auch um die effizienteste Art, diese Operationen durchzuführen.

Im ersten Beispiel werden alle Komponenten des Arrays `A` mit der Zahl verglichen. Das Ergebnis der Maskierung besteht in einem neuen Array mit der gleichen Shape, in dem ein `True` steht, falls an der entsprechenden Position in `A` eine 4 stand, ansonsten wird der Wert auf `False` gesetzt.

```
import numpy as np
A = np.array([4, 7, 3, 4, 2, 8])
print(A == 4)
```

```
[ True False False  True False False]
```

Analog kann man die einzelnen Komponenten auch mittels der Vergleichsoperatoren “<”, “<=”, “>” und “>=” bearbeiten. Die Arbeitsweise ist analog zu dem vorigen Fall:

```
print(A < 5)
```

```
[ True False  True  True  True False]
```

Dies lässt sich auch auf Arrays mit höherer Dimension anwenden:

```
B = np.array([[42, 56, 89, 65],
              [99, 88, 42, 12],
              [55, 42, 17, 18]])

print(B >= 42)
```

```
[[ True  True  True  True]
 [ True  True  True False]
 [ True  True False False]]
```

Damit lassen sich auch Arrays binarisieren. Betrachten wir das folgende Array `A` als ein Grauwertbild, so können wir dieses mit der Schwelle 15 binarisieren:

```
import numpy as np

A = np.array([
[12, 13, 14, 12, 16, 14, 11, 10,  9],
[11, 14, 12, 15, 15, 16, 10, 12, 11],
[10, 12, 12, 15, 14, 16, 10, 12, 12],
[ 9, 11, 16, 15, 14, 16, 15, 12, 10],
[12, 11, 16, 14, 10, 12, 16, 12, 13],
[10, 15, 16, 14, 14, 14, 16, 15, 12],
[13, 17, 14, 10, 14, 11, 14, 15, 10],
[10, 16, 12, 14, 11, 12, 14, 18, 11],
[10, 19, 12, 14, 11, 12, 14, 18, 10],
[14, 22, 17, 19, 16, 17, 18, 17, 13],
[10, 16, 12, 14, 11, 12, 14, 18, 11],
[10, 16, 12, 14, 11, 12, 14, 18, 11],
[10, 19, 12, 14, 11, 12, 14, 18, 10],
[14, 22, 12, 14, 11, 12, 14, 17, 13],
[10, 16, 12, 14, 11, 12, 14, 18, 11]])

B = A < 15
B.astype(np.int)
```

```
array([[1, 1, 1, 1, 0, 1, 1, 1, 1],
       [1, 1, 1, 0, 0, 0, 1, 1, 1],
       [1, 1, 1, 0, 1, 0, 1, 1, 1],
       [1, 1, 0, 0, 1, 0, 0, 1, 1],
       [1, 1, 0, 1, 1, 1, 0, 1, 1],
       [1, 0, 0, 1, 1, 1, 0, 0, 1],
       [1, 0, 1, 1, 1, 1, 1, 0, 1],
       [1, 0, 1, 1, 1, 1, 1, 0, 1],
       [1, 0, 1, 1, 1, 1, 1, 0, 1],
       [1, 0, 0, 0, 0, 0, 0, 0, 1],
       [1, 0, 1, 1, 1, 1, 1, 0, 1],
       [1, 0, 1, 1, 1, 1, 1, 0, 1],
       [1, 0, 1, 1, 1, 1, 1, 0, 1],
       [1, 0, 1, 1, 1, 1, 1, 0, 1],
       [1, 0, 1, 1, 1, 1, 1, 0, 1]])
```

Alle Werte des Originalarrays `A` wurden durch 0 bzw. 1 ersetzt. Im Bild kann man übrigens auch ein großes A erkennen.

Fancy-Indizierung

Das Prinzip der "Fancy-Indizierung" ist recht einfach: Statt eines einzelnen Indexes benutzt man ein Array mit Indizes. Dadurch kann man mehrere Elemente auf einen Schlag ansprechen.

```
A = np.array([34, 8, 99, 12, 1, 102, 44])

# umständlich:
B = np.array([A[1], A[3], A[5]])
print(B)

# mit fancy Indizierung:
```

```
B2 = A[[1, 3, 5]]
print(B2)
```

```
[  8  12 102]
[  8  12 102]
```

In unserem nächsten Beispiel benutzen wir die boolesche Maske eines Arrays, um die entsprechenden Elemente eines anderen Arrays auszuwählen, d.h. wir indizieren das Array `C` mit einer Booleschen Maske, die wir mittels Maskierung des Arrays `A` erzeugen. Das Ergebnis ist dann eine Kopie und keine Sicht (View).

Das neue Array R beinhaltet all die Elemente aus C, bei denen im Array A der Test A <= 5 `True` liefert.

```
C = np.array([123, 188, 190, 99, 77, 88, 100])
A = np.array([4, 7, 2,8, 6, 9, 5])
print(A <= 5)
R = C[A <= 5]
print(R)
```

```
[ True False  True False False False  True]
[123 190 100]
```

Indizierung mit einem Integer-Array

Indizieren lässt sich auch beispielsweise mit einem Integer-Array oder mit einer Integer-Liste. Letzteres tun wir im nächsten Beispiel:

```
lst = [0, 2, 3, 1, 4, 1]
C[lst]
```

```
array([123, 190,  99, 188,  77, 188])
```

Wir wir sehen, können Indizes mehrfach und in beliebiger Reihenfolge auftreten!

Übung Extrahieren Sie aus dem Array `np.array([3, 4, 6, 10, 24, 89, 45, 43, 46, 99, 100])`, anhand von boolescher Indizierung, die Werte, die:

- nicht durch 3 teilbar sind
- durch 5 teilbar sind
- die durch 3 und durch 5 teilbar sind
- die durch 3 teilbar sind und setzen Sie diese auf 42

Lösung

```
import numpy as np
A = np.array([3, 4, 6, 10, 24, 89, 45, 43, 46, 99, 100])

div3 = A[A % 3 != 0]
print("Elemente von A, die nicht durch 3 teilbar sind:")
print(div3)
```

```
div5 = A[A % 5 == 0]
print("Elemente von A, die durch 5 teilbar sind:")
print(div5)

print("Elemente von A, die durch 3 und 5 teilbar sind:")
print(A[(A % 3 == 0) & (A % 5 == 0)])

A[ A % 3 == 0] = 42
print("Alle durch 3 teilbaren Werte von A wurden auf 42 gesetzt:")
print(A)
```

```
Elemente von A, die nicht durch 3 teilbar sind:
[  4  10  89  43  46 100]
Elemente von A, die durch 5 teilbar sind:
[ 10  45 100]
Elemente von A, die durch 3 und 5 teilbar sind:
[45]
Alle durch 3 teilbaren Werte von A wurden auf 42 gesetzt:
[ 42   4  42  10  42  89  42  43  46  42 100]
```

nonzero und where

Die Methode `nonzero` liefert die Indizes der Elemente aus einem Array zurück, die nicht 0 (non-zero) sind. Die Indizes werden als Tupel von eindimensionalen Arrays zurückgeliefert, eins für jede Dimension. Die entsprechenden non-zero-Werte eines Arrays A kann man dann durch Boolesches Indizieren erhalten:

`A[numpy.nonzero(A)]`

```
import numpy as np

A = np.array([[0, 2, 3, 0, 1],
              [1, 0, 0, 7, 0],
              [5, 0, 0, 1, 0]])

print(A.nonzero())
print(A[A.nonzero()])
```

```
(array([0, 0, 0, 1, 1, 2, 2]), array([1, 2, 4, 0, 3, 0, 3]))
[2 3 1 1 7 5 1]
```

Möchte man die Elemente als Pärchen von Zeilen und Spalten haben, so kann man `transpose` benutzen:

`transpose(nonzero(A))`

Es wird ein zweidimensionales Array erzeugt. Jede Zeile entspricht den Indizes eines non-zero-Elements in der Form `[Zeile, Spalte]`

```
np.transpose(A.nonzero())
```

```
array([[0, 1],
       [0, 2],
       [0, 4],
       [1, 0],
```

```
       [1, 3],
       [2, 0],
       [2, 3]])
```

Die Funktion `nonzero` kann dazu verwendet werden, um die Indizes aus einem Array zu holen, bei denen die Bedingung True ist. Im folgenden Skript erstellen wir das boolesche Array B >= 42:

```
B = np.array([[42, 56, 89, 65],
              [99, 88, 42, 12],
              [55, 42, 17, 18]])

print(B >= 42)
```

```
[[ True  True  True  True]
 [ True  True  True False]
 [ True  True False False]]
```

`np.nonzero(B >= 42)` produziert die Indizes aus B, auf die die Bedingung zutrifft.

```
B = np.array([[42, 56, 89, 65],
              [99, 88, 42, 12],
              [55, 42, 17, 18]])

np.nonzero(B >= 42)
```

```
(array([0, 0, 0, 0, 1, 1, 1, 2, 2]), array([0, 1, 2, 3, 0, 1, 2, 0, 1]))
```

Übung Berechnen Sie die Primzahlen zwischen 0 und 100 mit Hilfe eines booleschen Arrays.

Lösung

```
import numpy as np

is_prime = np.ones((100,), dtype=bool)

# Cross out 0 and 1 which are not primes:
is_prime[:2] = 0

# cross out its higher multiples (sieve of Eratosthenes):
nmax = int(np.sqrt(len(is_prime)))
for i in range(2, nmax):
    is_prime[2*i::i] = False

print(np.nonzero(is_prime))
```

```
(array([ 2,  3,  5,  7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
       61, 67, 71, 73, 79, 83, 89, 97]),)
```

Flatnonzero und count_nonzero Ähnliche Funktionen:

- `flatnonzero`:

Liefert die Indizes zurück, die non-zero sind, jedoch aus der flachen (eindimensionalen) Version der übergebenen Arrays.

- `count_nonzero`:

  Zählt die non-zero-Elemente in dem übergebenen Array.

10 Matrix Arithmetik

10.0.1 Matrix-Arithmetik unter NumPy und Python

Im vorigen Kapitel unserer Einführung in NumPy zeigten wir, wie man Arrays erzeugen und ändern kann. In diesem Kapitel wollen wir zeigen, wie wir in Python mittels NumPy ohne Aufwand und effizient Matrizen-Arithmetik betreiben können, also

- Matrizenaddition
- Matrizensubtraktion
- Matrizenmultiplikation
- Skalarprodukt
- Kreuzprodukt
- und weitere arithmetische Funktionen auf Matrizen

Die arithmetischen Standardoperationen

- \+
- \-
- *
- /
- **
- %

werden bei `ndarray`-Objekten elementweise angewendet. Die Array-Formen müssen dabei entweder übereinstimmen oder nach den Broadcasting-Regeln kompatibel sein. Für die eigentliche Matrixmultiplikation verwenden wir dagegen den Operator `@`.

```
import numpy as np

x = np.array([1, 5, 2])
y = np.array([7, 4, 1])
x + y
```

```
array([8, 9, 3])
```

```
x * y
```

```
array([ 7, 20,  2])
```

```
x - y
```

```
array([-6,  1,  1])
```

```
x / y
```

```
array([0.14285714, 1.25      , 2.        ])
```

```
x % y
```

```
array([1, 1, 0])
```

Vektor-Addition und Vektor-Subtraktion

Vielen dürfte die Vektoraddition aus dem Physikunterricht der Schule bekannt sein. Man benutzt die Vektoraddition, um die Gesamtkraft zu berechnen, wenn verschiedene Einzelkräfte mit unterschiedlichen Ausrichtungen auf einen als punktförmig angenommenen Körper einwirken. Wenn man wissen will, welche Kraft insgesamt auf den Körper ausgeübt wird, muss man eine sogenannte Vektoraddition durchführen. Grafisch wird eine Vektoraddition realisiert, indem man durch Parallelverschiebung an die Spitze des ersten Vektors, also die Stelle, an der sich der Pfeil befindet, den Anfang des zweiten Vektors ansetzt.

Rechnerisch kann man mit der Vektoraddition die Gesamtverschiebung der Vektoren ermitteln, indem man die die jeweiligen Komponenten miteinander addiert.

```
x = np.array([3, 2])
y = np.array([5, 1])
z = x + y
z
```

```
array([8, 3])
```

Einen Vektor y von einem Vektor x zu subtrahieren ist das Gleiche wie wenn man das Negative von y zu dem Vektor x addiert. Es gilt also: x - y = x + (-y) Geometrisch kann man die Subtraktion eines Vektors wie folgt durchführen: Um y von x zu subtrahieren plazieren wir die Endpunkte von x und y auf den gleichen Punkt. Dann zeichnen wir einen Pfeil von der Spitze von y zu der Spitze von x. Dieser “Pfeil” ist ein Repräsentant des Vektors x - y, siehe Bild auf der rechten Seite.

Mathematisch gesehen, wird die Vektorsubtraktion durch eine Komponentenweise Subtraktion der einzelnen Komponenten durchgeführt.

Skalarprodukt / Dotprodukt

Das Skalarprodukt wird häufig auch als Inneres Produkt oder Punktprodukt bezeichnet. Mathematisch stellt das Skalarprodukt eine algebraische Operation dar, die zwei Koordinationvektoren gleicher Größe als Argument nimmt und eine einfache Zahl zurückliefert. Das Ergebnis wird berechnet, indem die Komponenten mit gleichem Index multipliziert werden und die so erhaltenen Produkte anschließend addiert werden. Der Name Punktprodukt (oder englisch “dot product”) stammt übrigens davon, dass der “.” häufig als Operatorzeichen für diese Operation genutzt wird. Der Name Skalarprodukt stellt mehr den Aspekt in den Vordergrund, dass das Ergebnis der Operation ein Skalar ist.

Definition des Skalarproduktes: Aus der Definition des Skalarproduktes können wir ersehen, dass es benutzt werden kann, um den Winkel zwischen zwei Vektoren zu ermittlen.

Berechnung des Skalarproduktes: Im folgenden zeigen wir, wie man das Skalarprodukt mit Python und NumPy berechnet:

```
x = np.array([1, 2, 3])
y = np.array([-7, 8, 9])
np.dot(x, y)
```

```
np.int64(36)
```

```
dot = np.dot(x ,y)
x_modulus = np.sqrt((x*x).sum())
y_modulus = np.sqrt((y*y).sum())
cos_angle = dot / x_modulus / y_modulus # cosine of angle between x and y
angle = np.arccos(cos_angle)
angle
```

```
np.float64(0.808233789010825)
```

```
angle * 360 / 2 / np.pi # angle in degrees
```

```
np.float64(46.308384970187326)
```

Matrixprodukt

Das Matrixprodukt zweier Matrizen kann berechnet werden, wenn die Anzahl der Spalten der ersten Matrix gleich der Anzahl der Zeilen der zweiten Matrix ist.

Das Produkt einer (l \times m)-Matrix (A) und einer (m \times n)-Matrix (B) ist eine (l \times n)-Matrix (C).

Die folgende Grafik verdeutlicht dieses Verfahren:

Für gewöhnliche NumPy-Arrays (`ndarray`) ist `@` die klarste Schreibweise für Matrixmultiplikation. `np.matmul(a, b)` ist die entsprechende Funktion. Der Operator `*` bleibt dagegen die elementweise Multiplikation.

```
x = np.array([[2, 3],
              [3, 5]])
y = np.array([[1, 2],
              [5, -1]])

x @ y
```

```
array([[17,  1],
       [28,  1]])
```

Eine einfache praktische Anwendung zum Matrizenprodukt

In dem folgenden praktischen Beispiel kommen wir auf die Schokoladenseite des Lebens zu sprechen. Nehmen wir an, wir haben vier Personen, die wir Lukas, Mia, Leon und Hannah taufen. Jeder von ihnen hat Pralinen der Marken A, B und C gekauft. Lukas kaufte 100

g der Marke A, 175 g der Marke B und 210 von C. Mia wählte 90 g von A, 160 g von B und 150 g von C. Leon kaufte 200 g von A, 50 von B und 100 g von C. Hannah mag allem Anschein nach nicht die Sorte B, weil sie keine Pralinen dieser Sorte gekauft hatte. Dafür scheint sie ein echter Fan der Sorte C zu sein, weil sie davon gleich 310 g gekauft hatte. Außerdem kaufte sie noch 120 g der Make A. Nun wollen Sie natürlich wissen - oder auch nicht - wieviel die einzelnen Sorten kosten: A kostet 2.98 pro 100 g, B kostet 3.90 und C nur 1.99 Euro. Wenn wir nun berechnen wollen wieviel jeder von ihnen zu zahlen hatte, können wir NumPy und die Matrizenmultiplikation nutzen:

```
menge_pro_person = np.array([[100, 175, 210],
                             [90, 160, 150],
                             [200, 50, 100],
                             [120, 0, 310]])
preis_per_100_g = np.array([2.98, 3.90, 1.99])
preis_in_Cent = np.dot(menge_pro_person, preis_per_100_g)
preis_in_euro = preis_in_Cent / 100
preis_in_euro
```

```
array([13.984, 11.907,  9.9  ,  9.745])
```

Kreuzprodukt / Vektorprodukt

So nun müssen wir wieder den Konsum der leckeren Pralinen einstellen und uns einem kalorienärmeren mathematischeren Thema zuwenden, dem Kreuzprodukt oder Vektorprodukt. Das Kreuz- oder Vektorprodukt ist eine binäre Operation im dreidimensionalen Raum. Das Kreuzprodukt zweier Vektoren a und b wird mit a $\times$ b bezeichnet. Das Ergebnis ist ein Vektor, der senkrecht zu den Vektoren steht, die multipliziert werden. Senkrecht im Sinne eines Rechtssystems, d.h. die beiden Vektoren a und b sowie a x b verhalten sich wie Daumen, Zeigefinger und Mittelfinger der rechten Hand (sogenannte Drei-Finger-Regel).

Das Kreuzprodukt ist definiert als: wobei n ein Einheitsvektor ist, der senkrecht auf der von a und b aufgespannten Ebene steht. Es gibt zwei Vektoren, die diese Eigenschaft erfüllen. Der richtige ist der, der mit der Drei- Finger-Regel (siehe oben) ermittelt werden kann. Falls einer der Vektoren, die multipliziert werden Null ist oder wenn die beiden Vektoren parallel sind, dann ist ihr Kreuzprodukt gleich Null. Die Länge des Ergebnisvektors entspricht der Fläche des von den beiden multiplizierten Vektoren aufgespannten Parallelogrammes. Falls die beiden Vektoren senkrecht zueinander stehen, erhalten wir ein Rechteck.

```
x = np.array([0, 0, 1])
y = np.array([0, 1, 0])

np.cross(x, y)
```

```
array([-1,  0,  0])
```

```
np.cross(y, x)
```

```
array([1, 0, 0])
```

11 Linear Kombinationen

11.0.1 Linear Combination

Definitions

A linear combination in mathematics is an expression constructed from a set of terms by multiplying each term by a constant and adding the results. Example of a linear combination: a · x + b · y is a linear combination of x and y with a and b constants. Generally; p = 1 · x1 + 2 · x2 … n · xn p is the scalar product of the values x1, x2 … xn and
1, 2 … n are called scalars. In most applications x1, x2 … xn are vectors and the lambdas are integers or real numbers. (For those, who prefer it mor formally: x1, x2 … xn V and V is a vector space, and
1, 2 … n K with K being a field)

Linear Combinations in Python

The vector y = (3.21, 1.77, 3.65) can be easily written as a linear combination of the unit vectors (0,0,1), (0,1,0) and (1,0,0): (3.21, 1.77, 3.65) = 3.21 · (1,0,0) + 1.77 (0,1,0) + 3.65 · (0,0,1) We can do the calculation with Python, using the module numpy:

```
import numpy as np
x = np.array([[0, 0, 1],
              [0, 1, 0],
              [1, 0, 0]])
y = ([3.65, 1.55, 3.42])
scalars = np.linalg.solve(x, y)
scalars
```

```
array([3.42, 1.55, 3.65])
```

The previous example was very easy, because we could work out the result in our head. What about writing our vector y = (3.21, 1.77, 3.65) as a linear combination of the vectors (0,1,1), (1,1,0) and (1,0,1)? It looks like this in Python:

```
import numpy as np
x = np.array([[0, 1, 1],
              [1, 1, 0],
              [1, 0, 1]])
y = ([3.65, 1.55, 3.42])
scalars = np.linalg.solve(x, y)
scalars
```

```
array([0.66, 0.89, 2.76])
```

Another Example

Any integer between -40 and 40 can be written as a linear combination of 1, 3, 9, 27 with scalars being elements of the set {-1, 0, 1}. For example: $7 = 1 \cdot 1 + (-1) \cdot 3 + 1 \cdot 9 + 0 \cdot 27$ We can calculate these scalars with Python. First we need a generator generating all the possible scalar combinations. If you have problems in understanding the concept of a generator, we recommend the chapter "Iterators and Generators" of our tutorial.

```
def factors_set():
    for i in [-1, 0, 1]:
        for j in [-1,0,1]:
            for k in [-1,0,1]:
                for l in [-1,0,1]:
                    yield (i, j, k, l)
```

We will use the memoize() technique (see chapter "Memoization and Decorators" of our tutorial) to memorize previous results:

```
def memoize(f):
    results = {}
    def helper(n):
        if n not in results:
            results[n] = f(n)
        return results[n]
    return helper
```

Finally, in our function linear_combination() we check every scalar tuple, if it can create the value n:

```
@memoize
def linear_combination(n):
    """ returns the tuple (i,j,k,l) satisfying
        n = i*1 + j*3 + k*9 + l*27      """
    weighs = (1,3,9,27)

    for factors in factors_set():
       sum = 0
       for i in range(len(factors)):
          sum += factors[i] * weighs[i]
       if sum == n:
          return factors
```

Putting it all together results in the following script:

```
def factors_set():
    for i in [-1, 0, 1]:
        for j in [-1, 0, 1]:
            for k in [-1, 0, 1]:
                for l in [-1, 0, 1]:
                    yield (i, j, k, l)


def memoize(f):
    results = {}
```

```
    def helper(n):
        if n not in results:
            results[n] = f(n)
        return results[n]
    return helper

@memoize
def linear_combination(n):
    """ returns the tuple (i,j,k,l) satisfying
        n = i*1 + j*3 + k*9 + l*27      """
    weighs = (1, 3, 9, 27)

    for factors in factors_set():
       sum = 0
       for i in range(len(factors)):
          sum += factors[i] * weighs[i]
       if sum == n:
          return factors

# calculate the linear combinations of the first 10 positive integers:
for i in range(1,11):
    print(linear_combination(i))
```

```
(1, 0, 0, 0)
(-1, 1, 0, 0)
(0, 1, 0, 0)
(1, 1, 0, 0)
(-1, -1, 1, 0)
(0, -1, 1, 0)
(1, -1, 1, 0)
(-1, 0, 1, 0)
(0, 0, 1, 0)
(1, 0, 1, 0)
```

12 Numpy Dateien Lesen Schreiben

12.0.1 Lesen und Schreiben von Daten-Dateien

Es gibt in NumPy viele Wege um Daten aus Dateien zu lesen und ebenso zu schreiben. In diesem Kapitel werden die verschiedenen Möglichkeiten diskutieren und die dazugehörigen Funktionen betrachten.

Text-Dateien speichern mit savetxt

Die ersten zwei Funktionen, die wir uns anschauen möchten, sind `savetxt` und `loadtxt`.

Im folgenden einfachen Beispiel definieren wir ein Array `x` und speichern es als Text-Datei mit `savetxt`:

```
import numpy as np

x = np.array([[1, 2],
              [3, 4],
              [5, 6]], np.int32)

np.savetxt("test.txt", x)
```

Die Datei “test.txt” ist eine Text-Datei und der Inhalt sieht wie folgt aus, wenn wir ihn uns in einem Linux-Terminal anschauen:\footnote{Unter Windows kann man das Kommanda ‘type test.txt’ benutzen.}

```
$ more test.txt
1.00000000000000000e+00 2.00000000000000000e+00
3.00000000000000000e+00 4.00000000000000000e+00
5.00000000000000000e+00 6.00000000000000000e+00
```

Da das Array aus Integern besteht, hätte man hier eher eine Datei erwartet, in der ganze Zahlen und nicht Float-Zahlen stehen. Man kann aber das Ausgabeformat selbst bestimmen. Im Folgenden speichern wir das Array in der Datei `test2.txt` mit drei Nachkommastellen und in der Datei `test3.txt` als Integers mit vorangestellten Leerzeichen, wenn die Anzahl der Stellen kleiner als 4 ist. Dafür übergeben wir einen Format-String an den Parameter `fmt`. Im vorigen Beispiel haben wir gesehen, dass der Default-Delimiter ein Leerzeichen ist. Beim Schreiben mit `np.savetxt` darf `delimiter` auch aus mehreren Zeichen bestehen, beispielsweise aus dem Smiley ” :-)“. Beim Einlesen ist zu beachten: `np.loadtxt` akzeptiert in aktuellen NumPy-Versionen nur ein einzelnes Trennzeichen. Für mehrzeichenige Trenner verwenden wir deshalb `np.genfromtxt`.

```
np.savetxt("test2.txt", x, fmt="%2.3f", delimiter=",")
np.savetxt("test3.txt", x, fmt="%04d", delimiter=" :-) ")
```

Die neu erstellten Dateien sehen folgendermaßen aus:

```
$ more test2.txt
1.000,2.000
3.000,4.000
5.000,6.000
$ more test3.txt
0001 :-) 0002
0003 :-) 0004
0005 :-) 0006
```

Text-Dateien laden mit loadtxt

loadtxt ohne Parameter Jetzt werden wir die Datei "test.txt" einlesen, die wir im vorigen Unterkapitel erstellt haben:

```
y = np.loadtxt("test.txt")
print(y)
```

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]
```

Spezielle Trenner

```
y = np.loadtxt("test2.txt", delimiter=",")
print(y)
```

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]
```

Die Datei mit dem mehrzeichenigen Smiley-Trenner kann in aktuellen NumPy-Versionen nicht mit `loadtxt` eingelesen werden, weil `loadtxt` nur ein einzelnes Trennzeichen akzeptiert. `genfromtxt` unterstützt dagegen auch mehrzeichenige Separatoren:

```
# np.loadtxt akzeptiert nur ein einzelnes Trennzeichen.
# Für den mehrzeichenigen Separator verwenden wir np.genfromtxt.
y = np.genfromtxt("test3.txt", delimiter=" :-) ")
print(y)
```

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]
```

Selektives Einlesen von Spalten Häufig ist es so, dass man aus einer solchen Datei nur bestimmte Spalten einlesen will. Zu diesem Zweck übergibt man dem Parameter `usecols` ein Tupel mit den gewünschten Spaltenindizes. Dabei beginnt die Nummerierung wie üblich mit dem Index 0. Um die Wirkungsweise des Parameters `usecols` besser demonstrieren zu könnten, erzeugen und speichern wir zuerst ein Array mit 6 Spalten:

```
Z = np.random.randint(-10, 10, size=(4,10))
print(Z)
np.savetxt("test3.txt", Z, fmt="%1d", delimiter=" ")
```

```
[[  8   5   1   0   1  -4  -8  -4  -2   9]
 [  0   8   1 -10  -5  -8   8  -1  -1   1]
 [ -6 -10   7 -10  -2  -7  -8  -3  -2   8]
 [  1   2   9   1  -7  -1   2  -3   7   6]]
```

```
y = np.loadtxt("test3.txt",
               delimiter=" ",
               usecols=(0, 1, 6))
print(y)
```

```
[[  8.   5.  -8.]
 [  0.   8.   8.]
 [ -6. -10.  -8.]
 [  1.   2.   2.]]
```

Datenkonvertierung beim Einlesen

```
def fahrenheit2celsius(t):
    return (float(t) - 32) * 5 / 9

converters_dict = {0:fahrenheit2celsius,
                   1:fahrenheit2celsius,
                   2:fahrenheit2celsius,
                   3:fahrenheit2celsius}
# Alternativ könnte converters_dict auch so definiert werden:
#converters_dict = dict(zip(range(4), [fahrenheit2celsius]*4)
y = np.loadtxt("temperatures.txt",
               delimiter=" ",
               skiprows=1,
               converters=converters_dict)

print(y)
```

```
[[23.5        20.94444444 24.5        25.27777778]
 [26.77777778 23.22222222 28.         29.        ]
 [30.16666667 26.33333333 31.77777778 32.33333333]]
```

In unserem nächsten Beispiel lesen wir die Datei “times_and_temperatures.txt” aus dem Kapitel Generatoren des Python-Tutorials. Jede Zeile enthält eine Zeitangabe im Format “hh::mm::ss” und eine zufällige Temperatir zwischen 10.0 und 25.0 C°. Wir müssen den Zeit-String in einen Float-Wert konvertieren. Die Zeit wird in Minuten und Sekunden angegeben. Wir definieren zuerst eine Funktion, die “hh::mm::ss” in Minuten wandelt:

```
def time2float_minutes(time):
    if type(time) == bytes:
        time = time.decode()
    t = time.split(":")
    minutes = float(t[0])*60 + float(t[1]) + float(t[2]) * 0.05 / 3
    return minutes
```

```
for t in ["06:00:10", "06:27:45", "12:59:59"]:
    print(time2float_minutes(t))
```

```
360.1666666666667
387.75
779.9833333333333
```

Sie werden festgestellt haben, dass wir den Typ der Zeit gegen Binär geprüft haben. Der Grund liegt in der Benutzung unserer Funktion `time2float_minutes` in `loadtxt` im nächsten Beispiel. Der Schlüsselwort-Parameter `converters` beinhaltet ein Dictionary, welches eine Funktion für jede Spalte hält (der Schlüssel der Spalte entspricht dem Schlüssel des Dictionaries), um die String-Daten der Spalte in Float-Werte zu konvertieren. Die String-Daten sind ein Byte-String. Deshalb müssen wir diesen in unserer Funktion in einen Unicode-String transferieren:

```
y = np.loadtxt("times_and_temperatures.txt",
               converters={ 0: time2float_minutes})
print(y)
```

```
[[ 360.    20.1]
 [ 361.5   16.1]
 [ 363.    16.9]
 ...
 [1375.5   22.5]
 [1377.    11.1]
 [1378.5   15.2]]
```

tofile

`tofile` ist eine Funktion, die es ermöglicht, den Inhalt eines Arrays sowohl im Binär-Format als auch im Text-Format in eine Datei zu schreiben.

`A.tofile(fid, sep=' ', format='%s')`

Die Daten aus dem ndarray A sind nun in "C"-Reihenfolge geschrieben, ohne Rücksicht der Reihenfolge aus A.

Die Datei, die mit dieser Methode geschrieben wurde, kann mit der Funktion `fromfile()` wieder geladen werden.

Parameter	Bedeutung
fid	Kann entweder ein offenes Datei-Objekt sein, oder ein String mit einem Dateinamen.
sep	Der String "sep" definiert den Separator, der für die Text-Ausgabe zwischen den Elementen verwendet wird. Übergibt man diesem Parameter einen leeren String, wird eine Binär-Datei geschrieben, genau wie file.write(a.tostring()).

Parameter	Bedeutung
format	Format-String für die Text-Ausgabe. Jeder Eintrag im Array wird so formatiert, indem dieser in den nächsten Python-Typ konvertiert wird und dann "format" angewendet wird.

Anmerkung:

Informationen zur Byte-Reihenfolge und Präzision gehen verloren. Deshalb ist es keine gute Idee, die Funktion zu benutzen, um Daten zu archivieren oder zwischen Maschinen mit verschiedener Byte-Reihenfolge zu transportieren. Einige dieser Probleme können überwunden werden. Bei der Ausgabe der Daten als Textdatei geht es auf Kosten der Geschwindigkeit und Dateigröße.

```
dt = np.dtype([('time', [('min', int), ('sec', int)]),
               ('temp', float)])
x = np.zeros((1,), dtype=dt)
x['time']['min'] = 10
x['temp'] = 98.25
print(x)

fh = open("test6.txt", "bw")
x.tofile(fh)
```

```
[((10, 0), 98.25)]
```

fromfile

`fromfile` liest Daten, die mit `tofile` geschrieben wurden. Es ist möglich, Binär-Daten zu lesen, wenn der Typ der Daten bekannt ist. Es ist ebenfalls möglich, einfach formatierte Textdateien zu parsen. Die Daten aus der Datei werden als Array zurückgegeben.

Die allgemeine Syntax sieht wie folgt aus:

```
numpy.fromfile(file, dtype=float, count=-1, sep='')
```

```
import numpy as np

data = np.arange(50, dtype=np.int32)
data.tofile("test4.txt")

with open("test4.txt", "rb") as fh:
    # 32 int32 values * 4 bytes = 128 bytes
    fh.seek(128)
    x = np.fromfile(fh, dtype=np.int32)

print(x)
```

```
[32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49]
```

Vorsicht:

Es können Probleme auftreten, wenn `tofile` und `fromfile` für Datenspeicherung (Storage) benutzt werden, denn Binär-Dateien sind nicht plattformunabhängig. Über `tofile` werden keine Informationen zur Byte-Reihenfolge oder Daten-Typen abgespeichert. Daten können im .npy-Format plattformunabhängig gespeichert werden, wenn stattdessen `save` und `load` verwendet werden.

Best Practice, um Daten zu laden und zu speichern

Der empfohlene Weg, um Daten mit NumPy in Python zu speichern und zu laden, besteht in der Benutzung von `load` und `save`. Im folgenden Beispiel benutzen wir eine temporäre Datei:

```
import numpy as np

print(x)

from tempfile import TemporaryFile

outfile = TemporaryFile()

x = np.arange(10)
np.save(outfile, x)

outfile.seek(0) # Only needed here to simulate closing & reopening file
np.load(outfile)
```

```
[32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49]
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Und noch ein anderer Weg: genfromtxt

Es gibt noch einen weiteren Weg um tabellarische Daten aus einer Datei zu lesen um Arrays zu konstruieren. Wie der Name schon verrät, sollte die Datei eine Text-Datei sein. Die Datei kann ebenfalls eine Archiv-Datei sein. `genfromtxt` kann die Archiv-Formate gzip und bzip2 verarbeiten. Der Typ des Archivs wird durch die Datei-Erweiterung angegeben d.h. '.gz' für gzip und '.bz2' für bzip2.

`genfromtxt` ist langsamer als `loadtxt`, jedoch kann es mit fehlenden Daten umgehen. Es verarbeitet die Datei in zwei Phasen. Zuerst werden die Zeilen in Strings konvertiert. Anschließend werden die Strings in den geforderten Daten-Typ konvertiert. Auf der anderen Seite arbeitet `loadtxt` mit nur einem Schritt, wodurch es schneller ist.

13 Matplotlib

13.0.1 Einführung

Matplotlib ist eine freie Python-Bibliothek zum Erzeugen von Diagrammen und wissenschaftlichen Visualisierungen. Sie arbeitet sehr gut mit NumPy-Arrays zusammen und kann ebenso Daten aus anderen array-ähnlichen Strukturen visualisieren.

Die Bibliothek unterscheidet zwischen einer *Figure* – der gesamten Abbildung – und einem oder mehreren *Axes*-Objekten, auf denen die eigentlichen Daten gezeichnet werden. Für kurze interaktive Beispiele ist die zustandsbehaftete `pyplot`-Schnittstelle bequem. In wiederverwendbaren Funktionen und größeren Programmen ist die explizite objektorientierte Schnittstelle mit `fig, ax = plt.subplots()` meist leichter zu kontrollieren.

Matplotlib kann Linien-, Punkt-, Balken-, Histogramm-, Kontur- und viele weitere Diagrammtypen erzeugen und Abbildungen in gängigen Raster- und Vektorformaten speichern.

Hinweis zu älteren Beispielen: Die frühere pylab-Schnittstelle mit from pylab import * sollte in neuem Code nicht verwendet werden. Üblich ist import matplotlib.pyplot as plt.

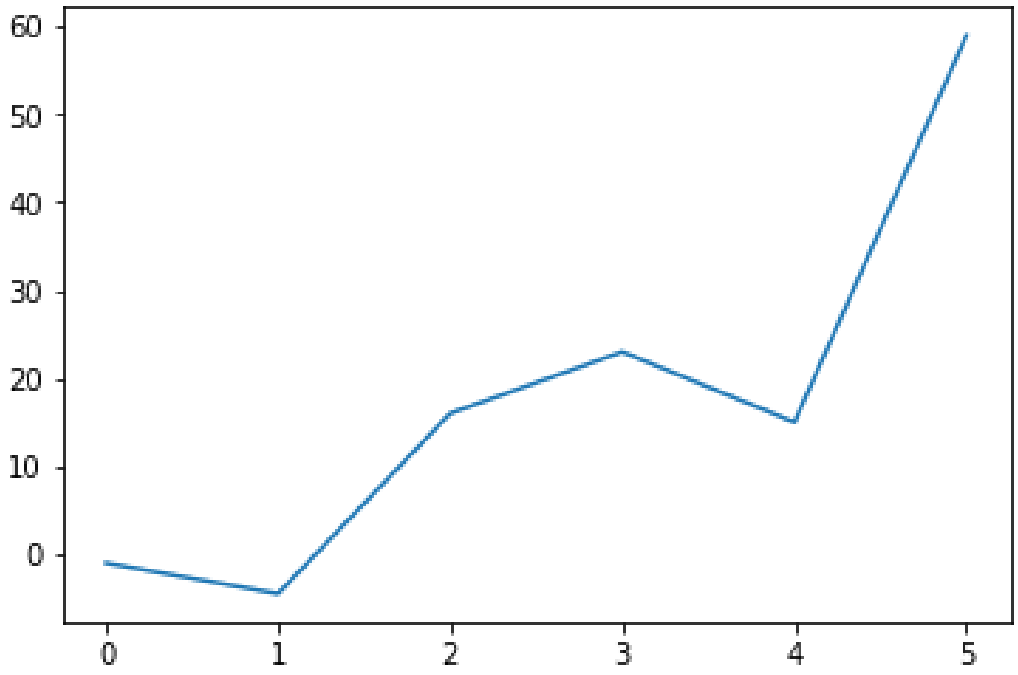
Ein erstes Beispiel

Wir werden mit einem einfachen Graphen beginnen. So einfach, dass es nicht mehr einfacher geht. Ein Graph in Matplotlib ist eine zwei- oder dreidimensionale Zeichnung, die mit Hilfe von Punkten, Kurven, Balken oder anderem einen Zusammenhang herstellt. Es gibt zwei Achsen: die horizontale x-Achse für die unabhängigen Werte und die vertikale y-Achse für die abhängigen Werte.

Wir beginnen im Folgenden mit `pyplot`, weil sich damit die Grundideen besonders kompakt zeigen lassen. Für komplexere Beispiele werden Sie später häufig explizit mit `Figure`- und `Axes`-Objekten arbeiten.

Es ist allgemein üblich, `matplotlib.pyplot` in `plt` umzubenennen. In unserem ersten Beispiel werden wir die `plot`-Funktion von `pyplot` benutzen. Wir übergeben an die `plot`-Funktion eine Liste von Werten. `plot` betrachtet und benutzt die Werte dieser Liste als y-Werte. Die Indizes dieser Liste werden automatisch als x-Werte genommen.

```
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23, 15, 59])
plt.show()
```



Dasselbe objektorientiert:

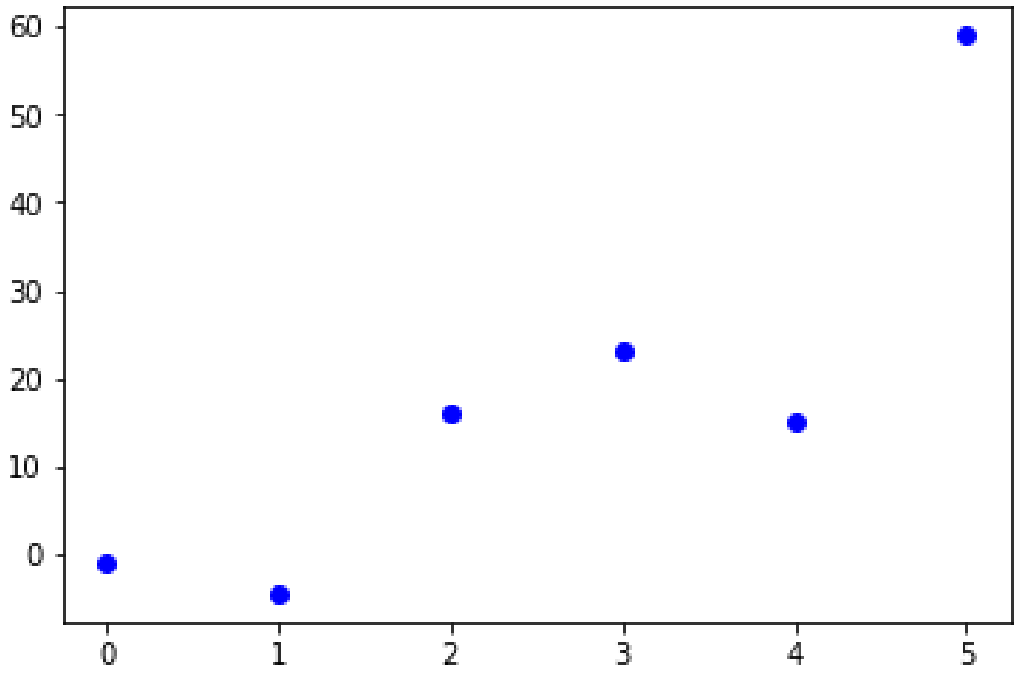
```
fig, ax = plt.subplots()
ax.plot([-1, -4.5, 16, 23, 15, 59])
ax.set_xlabel("Index")
ax.set_ylabel("Wert")
plt.show()
```

Hier wird das `Axes`-Objekt `ax` ausdrücklich angesprochen. Das ist besonders hilfreich, sobald eine Figure mehrere Unterdiagramme enthält.

Wir sehen einen zusammenhängenden Graphen, obwohl wir nur diskrete Werte für die Ordinate, allgemein auch als Y-Achse bezeichnet, zur Verfügung gestellt hatten. Als Werte für die Abszisse, also die X-Achse, wurden die Indizes genommen.

Indem wir einen Formatstring beim Funktionsaufruf mitübergeben, können wir einen Graphen mit diskreten Werten erzeugen, in unserem Fall mit blauen Vollkreisen. Der Formatstring definiert, wie die diskreten Punkte dazustellen sind:

```
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23, 15, 59], "ob")
plt.show()
```



Der Format-Parameter von pyplot.plot

In unserem vorigen Beispiel hatten wir `ob` als Formatparameter genutzt. Er besteht aus zwei Zeichen. Das erste definiert den Linienstil oder die Darstellung der diskreten Werte, die Markierungen (englisch 'markers'). Mit dem zweiten Zeichen wählt man die Farbe für den Graphen aus. Die Reihenfolge der beiden Zeichen hätte aber auch umgekehrt sein können, d.h. wir hätten auch `bo` schreiben können. Falls kein Formatparameter angegeben wird, wie es in unserem ersten Beispiel der Fall war, wird `b-` als Defaultwert benutzt, d.h. eine durchgehende blaue Linie wird ausgegeben.

Die folgenden Zeichen werden in einem Formatstring akzeptiert, um den Linienstil oder die Marker zu steuern:

```
============================================
Zeichen       Beschreibung
============================================
'-'  (Bindestrich) durchgezogene Linie
'--'  (zwei Bindestriche) gestrichelte Linie
'-.' Strichpunkt-Linie
':' punktierte Linie
'.' Punkt-Marker
',' Pixel-Marker
'o' Kreis-Marker
'v' Dreiecks-Marker, Spitze nach unten
'^' Dreiecks-Marker, Spitze nach oben
'<' Dreiecks-Marker, Spitze nach links
'>' Dreiecks-Marker, Spitze nach rechs
'1' tri-runter-Marker
'2' tri-hoch-Marker
'3' tri-links Marker
```

```
'4' tri-rechts Marker
's' quadratischer Marker
'p' fünfeckiger Marker
'*' Stern-Marker
'h' Sechseck-Marker1
'H' Sechseck-Marker2
'+' Plus-Marker
'x' x-Marker
'D' rautenförmiger Marker
'd' dünner rautenförmiger Marker
'|' Marker in Form einer vertikalen Linie
'_' Marker in Form einer horizontalen Liniw
================================================
```

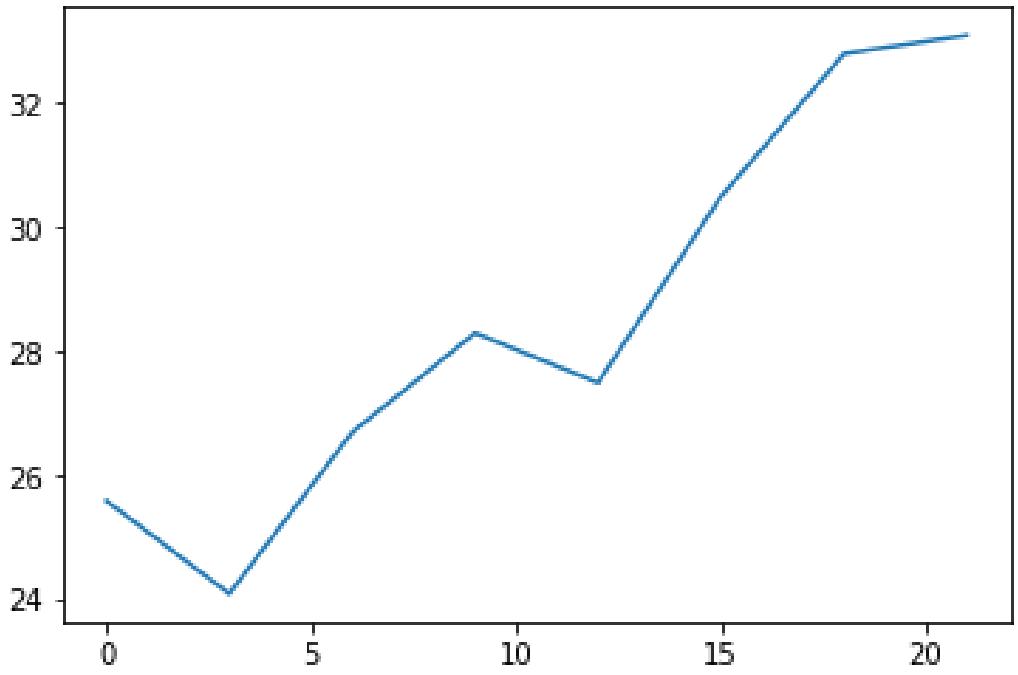
Die folgenden Farbabkürzungen sind möglich:

```
==================
Zeichen    Farbe
==================
'b' blau
'g' grün
'r' rot
'c' cyan
'm' magenta
'y' gelb
'k' schwarz
'w' weiß
==================
```

Wie einige sicherlich schon vermutet haben, kann man auch X-Werte an die Plot-Funktion übergeben. Im folgenden Beispiel übergeben wir eine Liste mit den Vielfachen von 3 zwischen 0 und 21 als X-Werte an plot:

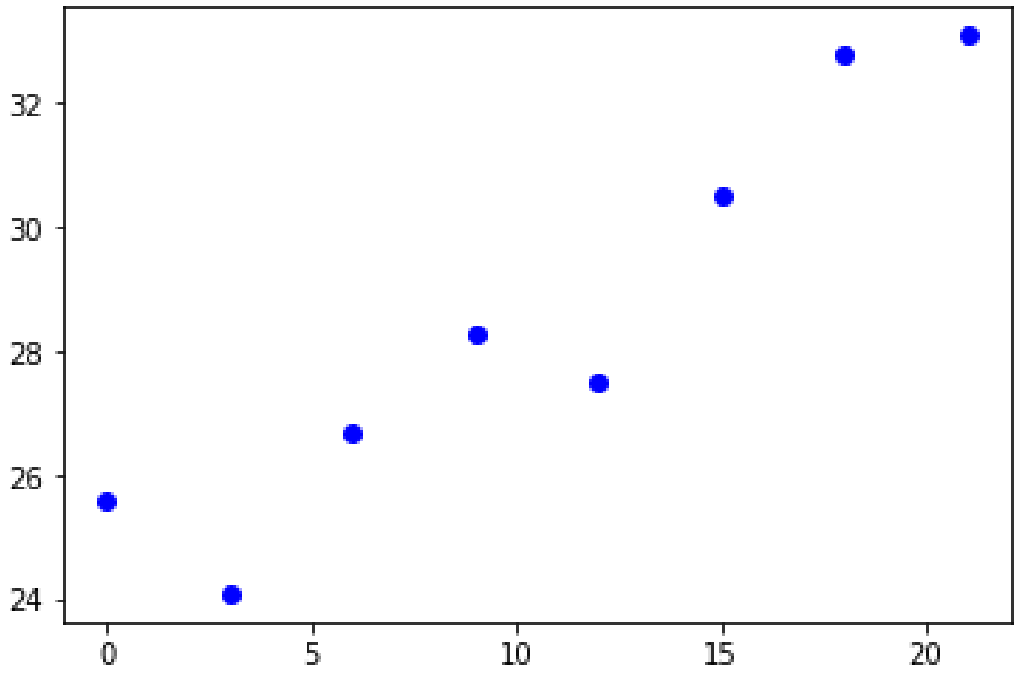
```
import matplotlib.pyplot as plt

# die X-Werte:
days = list(range(0, 22, 3))
print(days)
# die Y-Werte:
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]

plt.plot(days, celsius_values)
plt.show()
```

```
[0, 3, 6, 9, 12, 15, 18, 21]
```



... und das ganze wieder mit diskreten Werten:

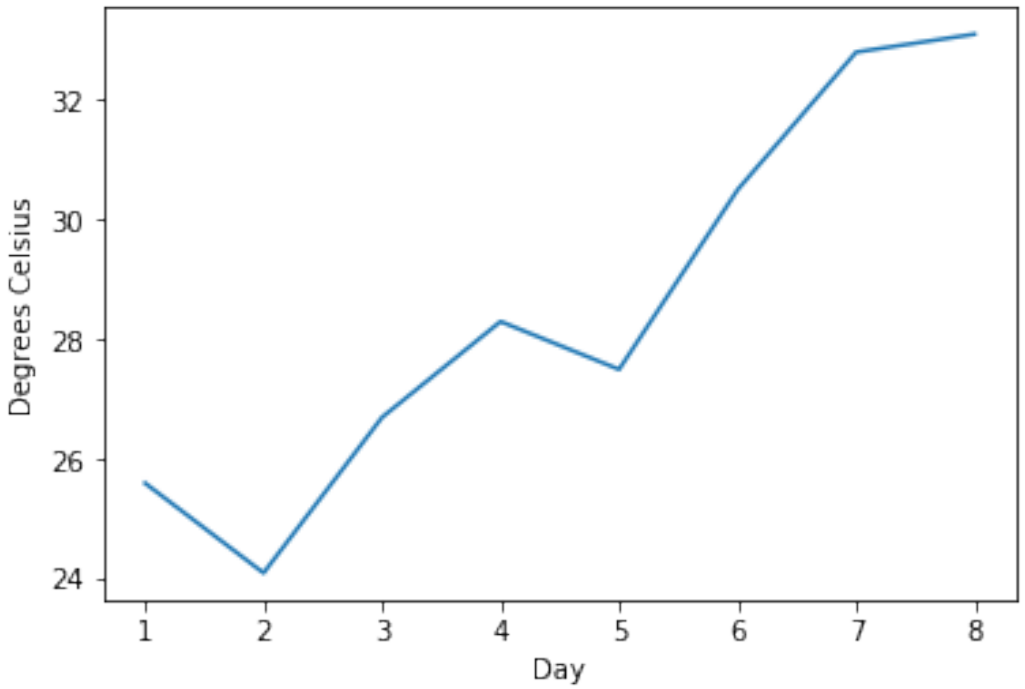
```
plt.plot(days, celsius_values, 'bo')
plt.show()
```



Bezeichnungen für die Achsen

Wir können das Aussehen unseres Graphen verbessern, indem wir die Achsen mit Bezeichnungen versehen. Dazu benutzen wir die `ylabel`- und `xlabel`-Funktionen von `pyplot`.

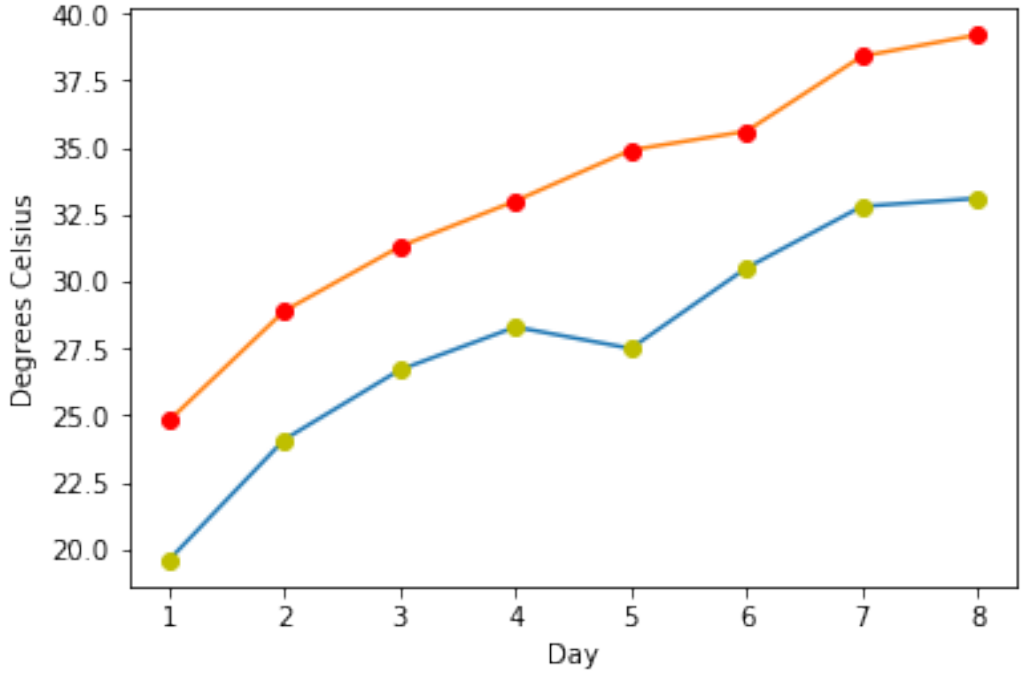
```
import matplotlib.pyplot as plt

days = list(range(1,9))
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]

plt.plot(days, celsius_values)
plt.xlabel('Day')
plt.ylabel('Degrees Celsius')
plt.show()
```



Wir können eine beliebige Anzahl von (x, y, fmt)-Gruppen in einer Plot-Funktion spezifizieren. Im folgenden Beispiel benutzen wir zwei verschiedene Listen mit Y-Werten:

```
import matplotlib.pyplot as plt

days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]

plt.xlabel('Day')
plt.ylabel('Degrees Celsius')

plt.plot(days, celsius_min,
         days, celsius_min, "oy",
         days, celsius_max,
         days, celsius_max, "or")
```

```
plt.show()
```



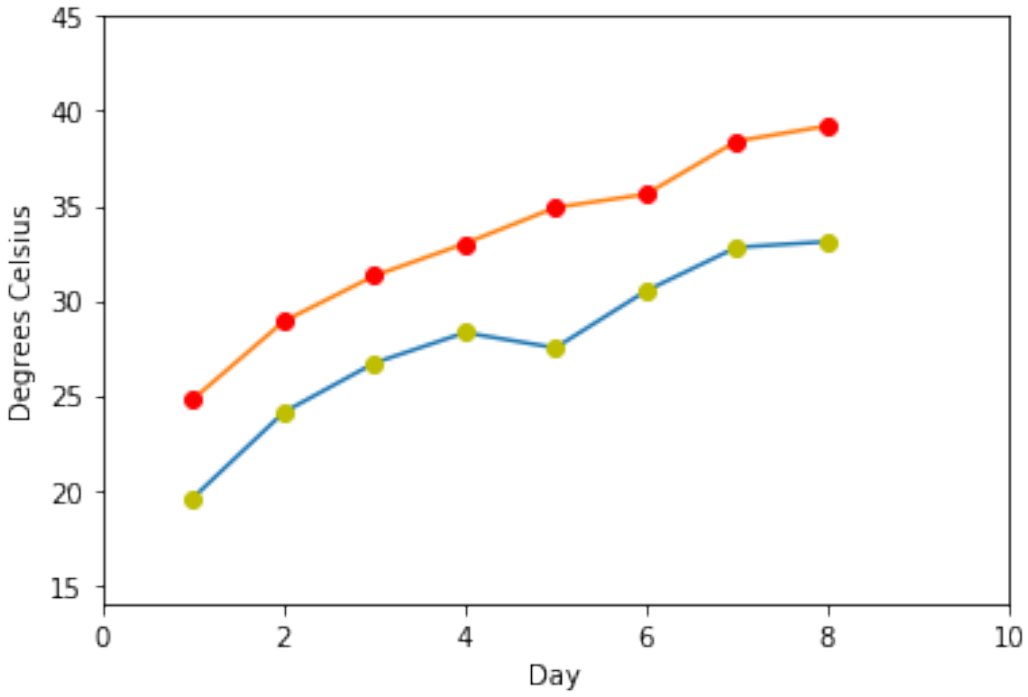
Abfragen und Ändern des Wertebereichs der Achsen

Mit der Funktion `axis` lässt sich der Wertebereich einer Achse abfragen und ändern. Ruft man `axis` ohne Argumente auf, liefert sie den aktuellen Wertebereich einer Achse zurück:

```
import matplotlib.pyplot as plt

days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]

plt.xlabel('Day')
plt.ylabel('Degrees Celsius')

plt.plot(days, celsius_min,
         days, celsius_min, "oy",
         days, celsius_max,
         days, celsius_max, "or")

print("The current limits for the axes are:")
print(plt.axis())
print("We set the axes to the following values:")
xmin, xmax, ymin, ymax = 0, 10, 14, 45
print(xmin, xmax, ymin, ymax)
plt.axis([xmin, xmax, ymin, ymax])
plt.show()
```

```
The current limits for the axes are:
```

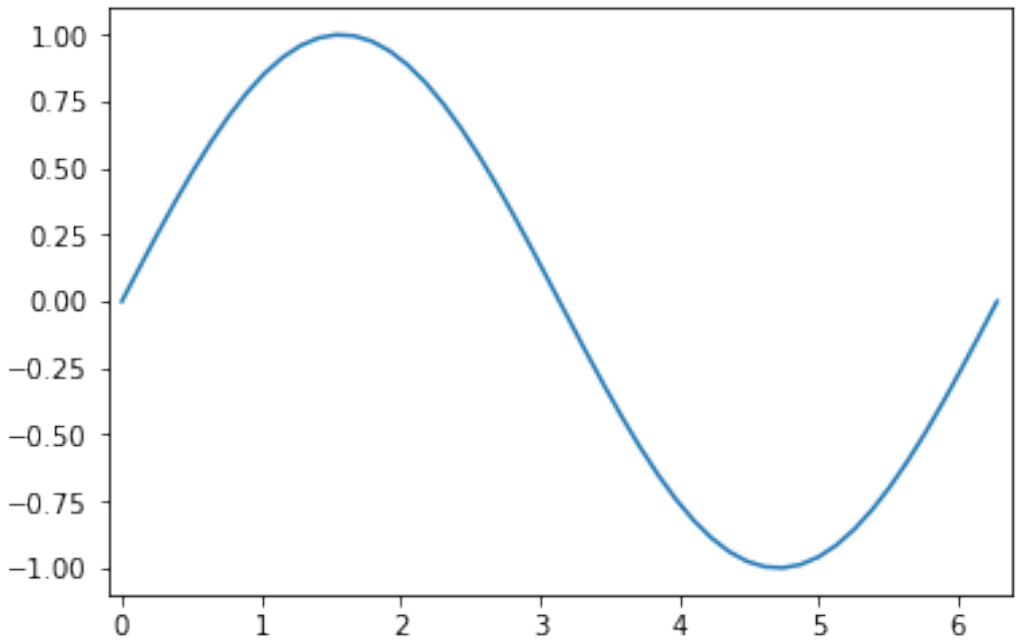
```
(0.6499999999999999, 8.35, 18.62, 40.18)
We set the axes to the following values:
0 10 14 45
```



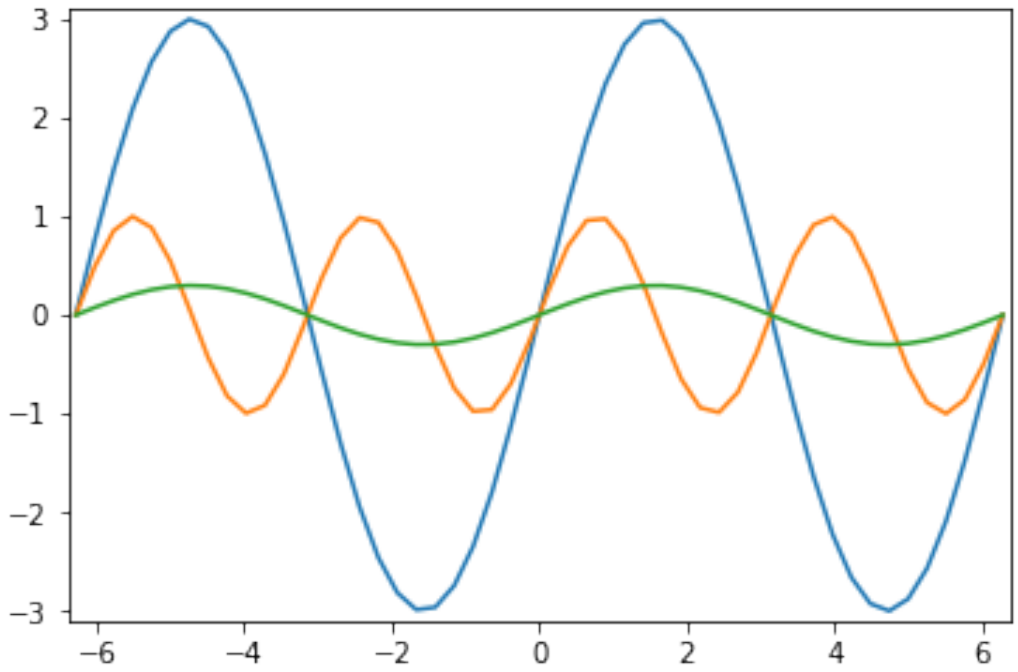
linspace zur Definition von X-Werten

Im folgenden Beispiel werden wir die NumPy-Funktion `linspace` verwenden. `linspace` wird dazu benutzt, gleichmäßig verteilte Werte innerhalb eines spezifizierten Intervalls zu erzeugen. Wir haben `linspace` ausführlich in unserem NumPy-Kapitel behandelt.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F = np.sin(X)
plt.plot(X,F)
startx, endx = -0.1, 2*np.pi + 0.1
starty, endy = -1.1, 1.1
plt.axis([startx, endx, starty, endy])
plt.show()
```

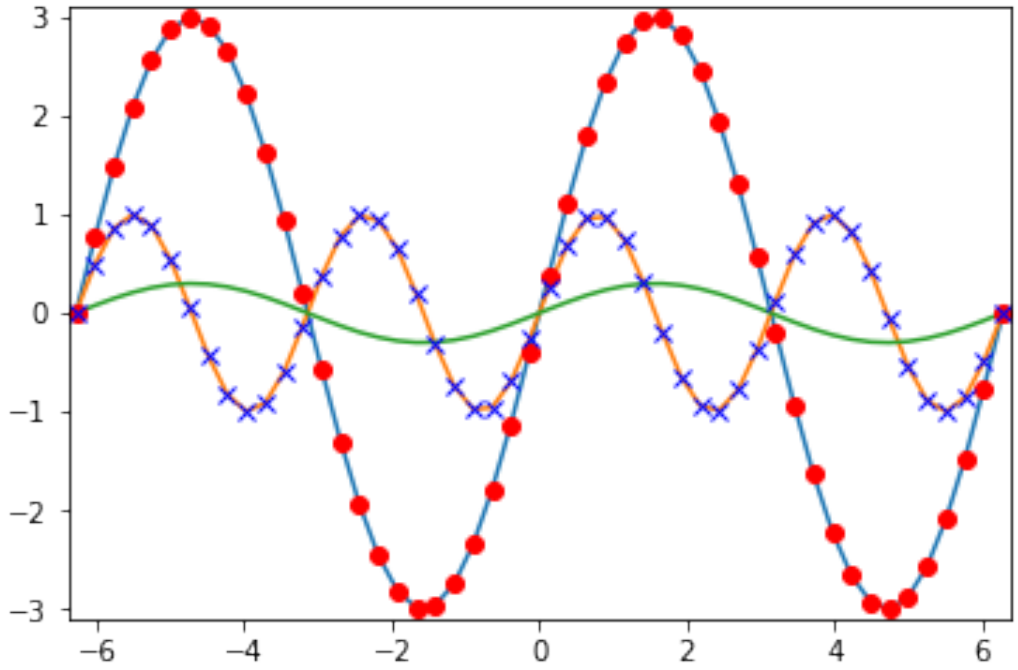



```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)

startx, endx = -2 * np.pi - 0.1, 2*np.pi + 0.1
starty, endy = -3.1, 3.1
plt.axis([startx, endx, starty, endy])

plt.plot(X,F1)
plt.plot(X,F2)
plt.plot(X,F3)
plt.show()
```



Das nächste Beispiel ist im Prinzip nichts Neues. Wir fügen lediglich zwei weitere Plots mit diskreten Punkten hinzu:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)
startx, endx = -2 * np.pi - 0.1, 2*np.pi + 0.1
starty, endy = -3.1, 3.1

plt.axis([startx, endx, starty, endy])
plt.plot(X,F1)
plt.plot(X,F2)
plt.plot(X,F3)
plt.plot(X, F1, 'ro')
plt.plot(X, F2, 'bx')
plt.show()
```



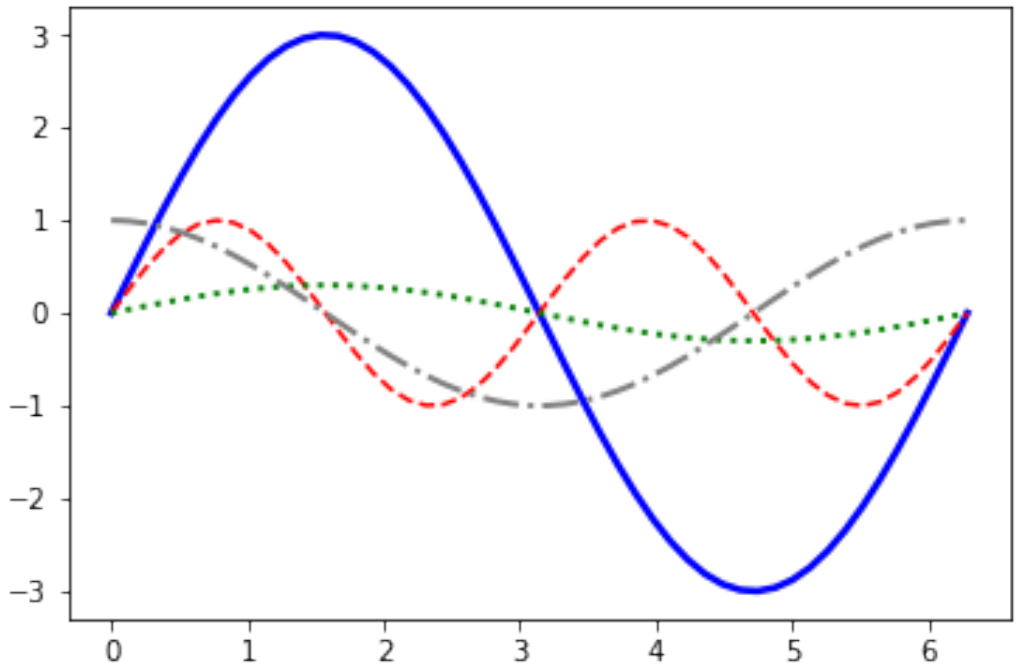
Linienstil ändern

Der Linienstil eines Plots kann durch die Parameter `linestyle` oder `ls` der `plot`-Funktion beeinflusst werden. Sie können auf einen der folgenden Werte gesetzt werden:

'-', '–', '-.', ':', 'None', ' ', ''

Wir können mit `linewidth`, wie der Name impliziert, die Linienstärke oder Liniendicke setzen:

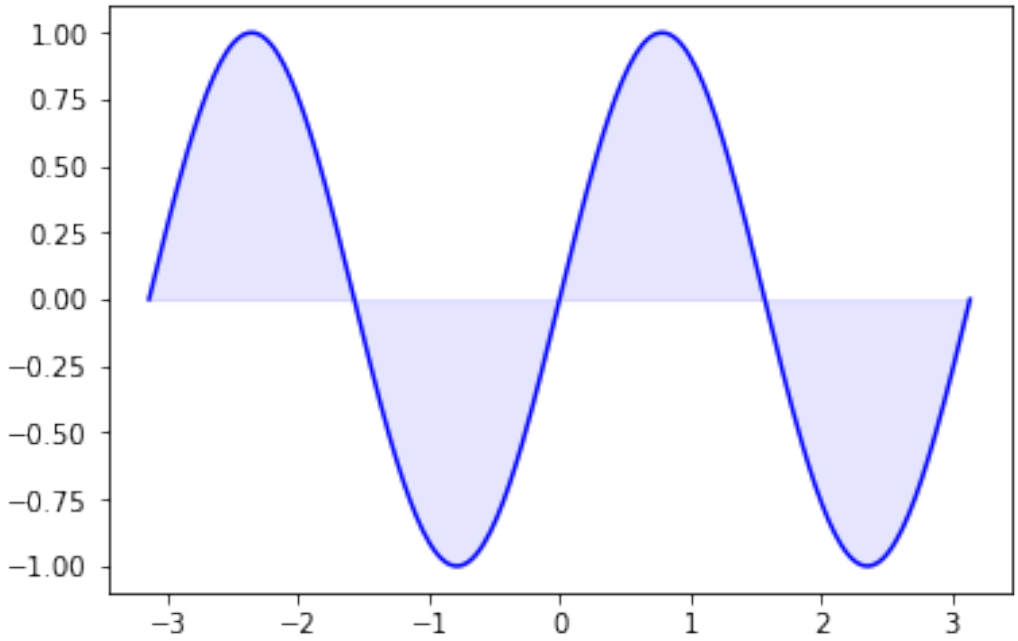
```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)
F4 = np.cos(X)

plt.plot(X, F1, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, F2, color="red", linewidth=1.5, linestyle="--")
plt.plot(X, F3, color="green", linewidth=2, linestyle=":")
plt.plot(X, F4, color="grey", linewidth=2, linestyle="-.")
plt.show()
```



Flächen einfärben

Mit der pyplot-Funktion `fill_between` ist es möglich, Flächen zwischen Kurven oder Achsen zu schraffieren oder einzufärben. Im folgenden Beispiel füllen wir die Flächen zwischen der X-Achse und dem Graph der Funktion `sin(2*X)`:

```
import numpy as np
import matplotlib.pyplot as plt

n = 256
X = np.linspace(-np.pi,np.pi,n,endpoint=True)
Y = np.sin(2*X)

plt.plot (X, Y, color='blue', alpha=1.00)
plt.fill_between(X, 0, Y, color='blue', alpha=.1)
plt.show()
```



Die allgemeine Syntax von `fill_between`:

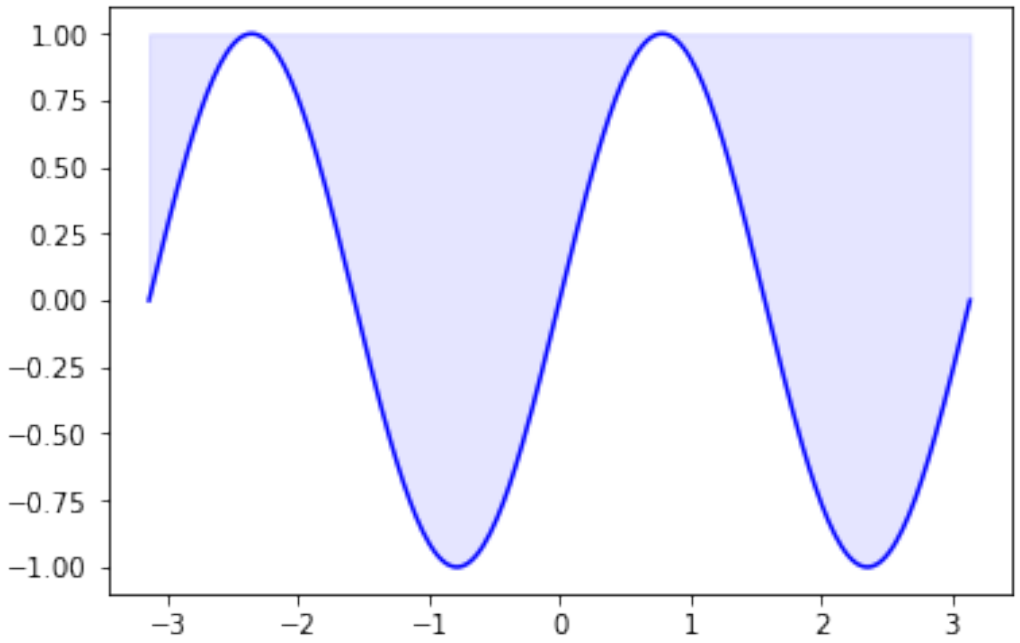
```
fill_between(x, y1, y2=0, where=None, interpolate=False, **kwargs)
```

Die Parameter von `fill_between`:

Parameter	Bedeutung
x	Ein Array mit N Elementen mit X-Werten
y1	Ein Array mit N Elementen (oder ein Skalar) von Y-Daten
y2	Ein Array mit N Elementen (oder ein Skalar) von Y-Daten
where	Wenn auf None gesetzt, wird per Default alles gefüllt. Wenn es nicht auf `None` gesetzt wird, so wird ein numpy boolean-Array erwartet mit N Elementen. Es werden nur dann die Regionen eingefärbt, bei denen where==True ist.
interpolate	Wenn True, so wird zwischen zwei Linien interpoliert, um den genauen Schnittpunkt zu finden. Andernfalls werden die Start- und Endwerte nur als explizite Werte auf der Region erscheinen.
kwargs	Schlüsselwort-Argumente, die an PolyCollection übergeben werden.

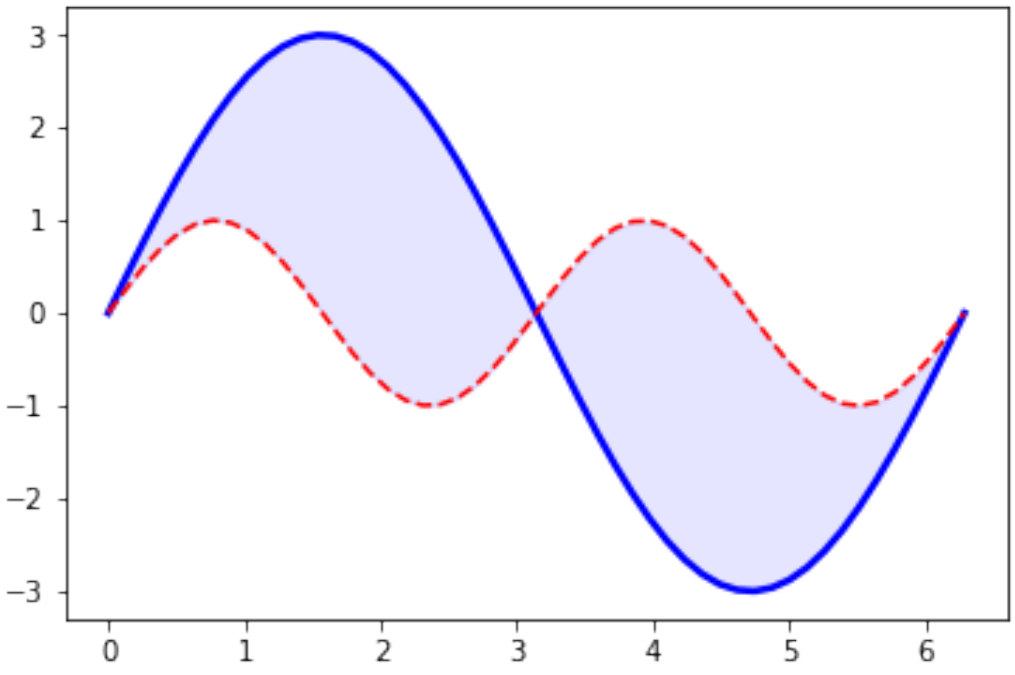
```
import numpy as np
import matplotlib.pyplot as plt
```

```
n = 256
X = np.linspace(-np.pi,np.pi,n,endpoint=True)
Y = np.sin(2*X)

plt.plot (X, Y, color='blue', alpha=1.00)
plt.fill_between(X, Y, 1, color='blue', alpha=.1)
plt.show()
```



Im nächsten Beispiel füllen wir die Flächen zwischen den Funktionen F1 und F2:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)


plt.plot(X, F1, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, F2, color="red", linewidth=1.5, linestyle="--")
plt.fill_between(X, F1, F2, color='blue', alpha=.1)
plt.show()
```


3
2
1
0
−1
−2
−3
0
1
2
3
4
5
6

14 Matplotlib Spines Und Ticks

14.0.1 Matplotlib Tutorial: Achsen und Skalenteilung

Achsenverschiebungen und Achsenbezeichnungen

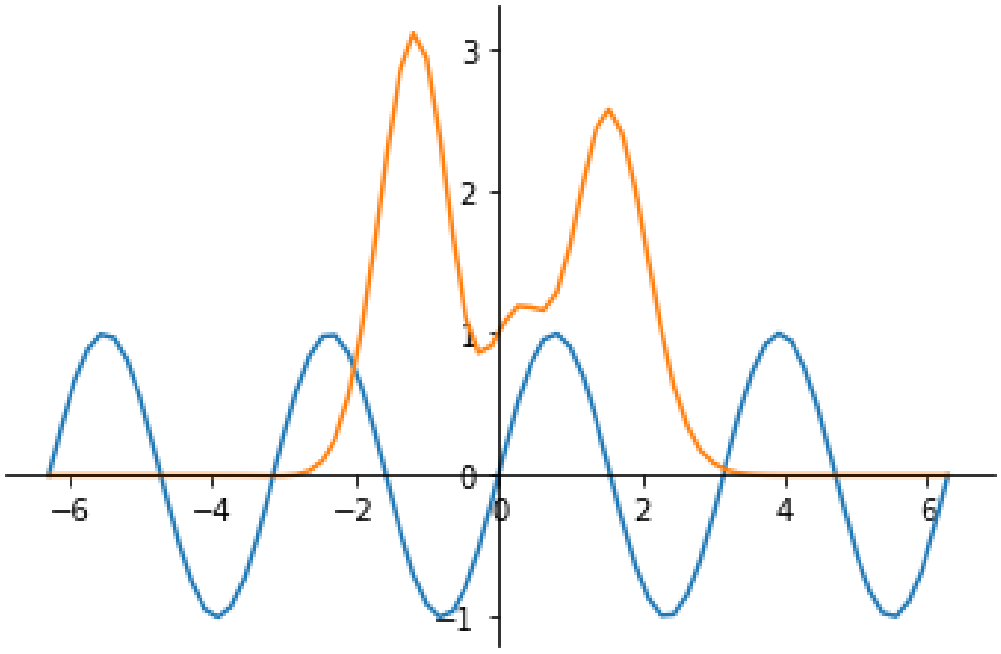
In Matplotlib werden die Achsen als "spines" bezeichnet. Das englische Wort "spine" bezeichnet normalerweise das Rückgrat oder die Wirbelsäule des menschlichen Skeletts. Es kann aber auch für die Stacheln eines Kaktus stehen. Im Bild haben wir ein Bild mit Kakteenstacheln soweit verändert, dass sie Ähnlichkeit mit einem Brustkorb haben. "spines" bezeichnen in Matplotlib die Linien, welche die Achsen-Markierungen verbinden unter Berücksichtigung des Datenbereichs. Im Deutschen sprechen wir von Achsen.

Wir demonstrieren im Folgenden, wie wir die Achsen ("spines") an beliebige Stellen verschieben können.

Bevor wir damit beginnen können, führen wir die gca-Funktion ein. Diese Funktion liefert eine Referenz auf die aktuelle Plotinstanz bzw. Figur zurück.

Wir können zum Beispiel `plt.gca(projection='polar')` aufrufen, um die aktuellen Polar-Achsen zu erhalten.

```
import numpy as np
import matplotlib.pyplot as plt

X  = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(2* X)
F2 = (2*X**5 + 4*X**4 - 4.8*X**3 + 1.2*X**2 + X + 1)*np.exp(-X**2)

# aktuellen Plot zuweisen:
ax = plt.gca()

# Obere und rechte Achse unsichtbar werden lassen:
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')

# Untere Achse auf die y=0 Position bewegen:
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))

# Linke Achse auf die Position x == 0 bewegen:
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

plt.plot(X, F1)
plt.plot(X, F2)

plt.show()
```

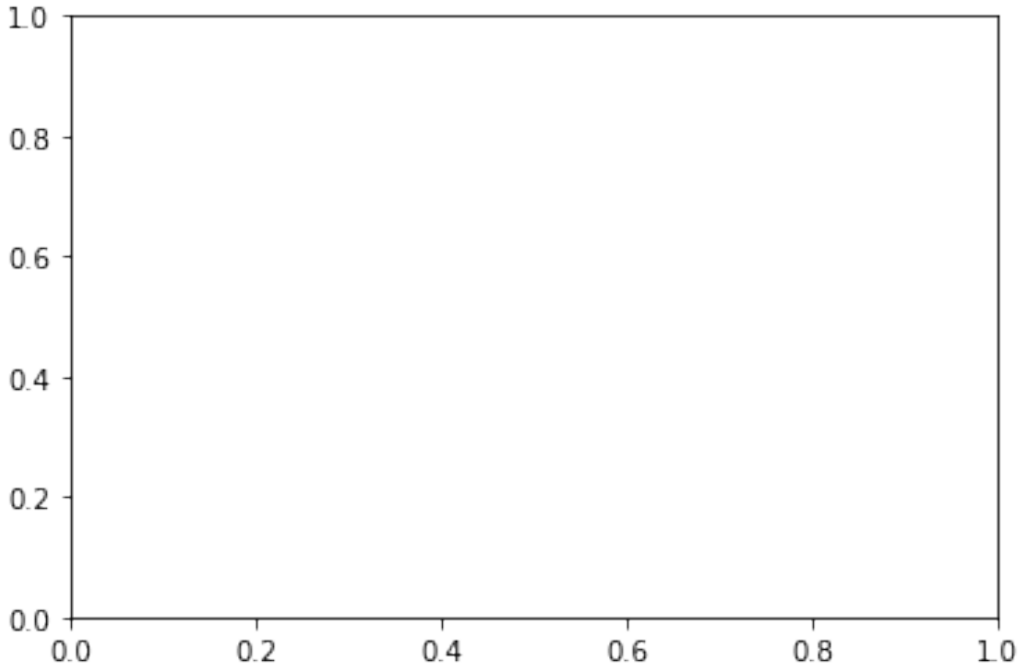


Matplotlib hat bis jetzt – in allen vorangegangenen Beispielen – automatisch den Abstand der Punkte auf der Achse ermittelt. In unserem vorigen Beispiel haben wir gesehen dass die X-Achse mit `-8, -6. -4, -2, 0, 2, 4, 6, 8` nummeriert war, während die Y-Achse mit `-2.0, -1.5, -1.0, -0.5, 0, 0.5, 1.0, 1.5, 2.0` beschriftet ist.

xticks ist eine Methode, die für das Abfragen und Verändern von Tick-Positionen und Beschriftungen benutzt werden kann. Das Gleiche gilt für die Methode yticks.

```
ax = plt.gca()

locs, labels = plt.xticks()
print(locs, labels)

locs, labels = plt.yticks()
print(locs, labels)
```

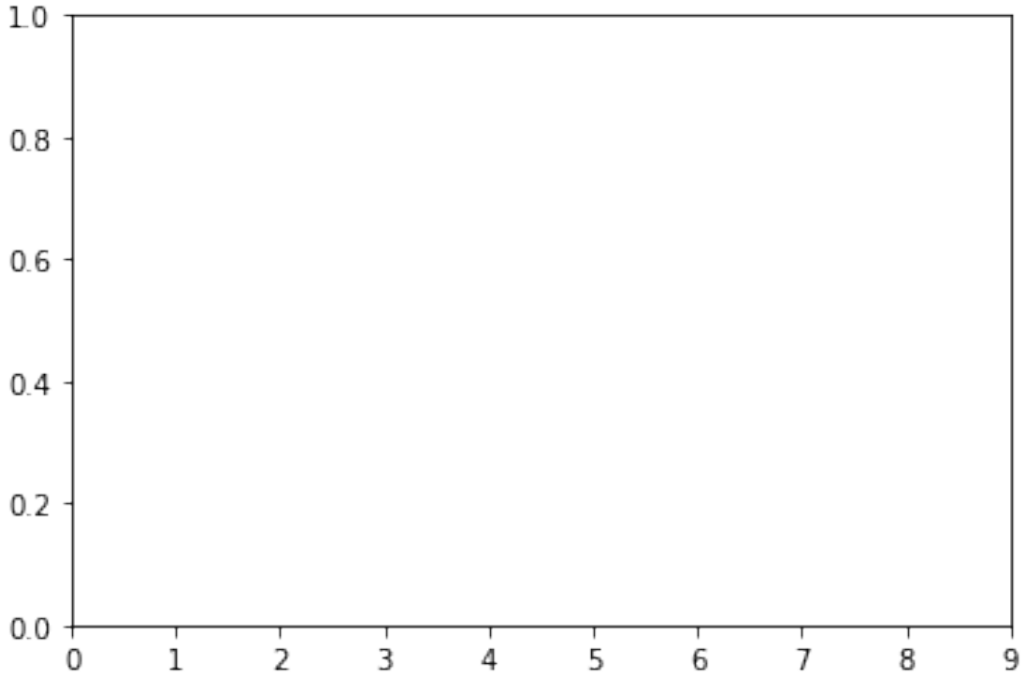
```
[0.  0.2 0.4 0.6 0.8 1. ] <a list of 6 Text xticklabel objects>
[0.  0.2 0.4 0.6 0.8 1. ] <a list of 6 Text yticklabel objects>
```



Wie bereits erwähnt, kann xticks ebenso dazu verwendet werden, um die Position der Ticks auf der X-Achse zu verändern:

```
plt.xticks( np.arange(10) )

locs, labels = plt.xticks()
print(locs, labels)
```

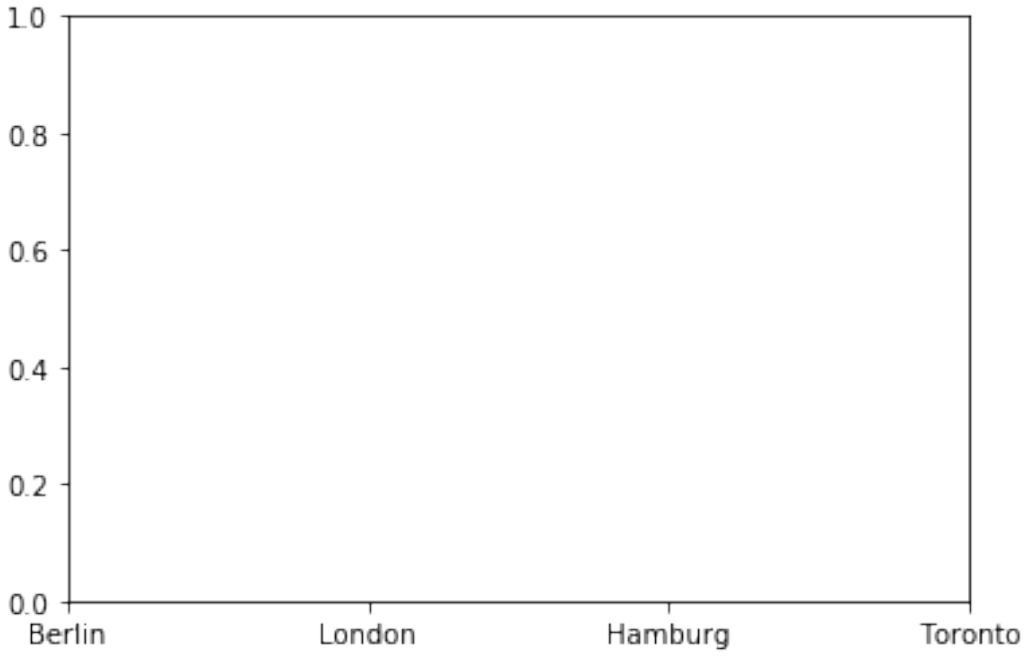
```
[0 1 2 3 4 5 6 7 8 9] <a list of 10 Text xticklabel objects>
```



Jetzt verändern wir sowohl die Position als auch die Beschriftung der Ticks auf der X-Achse:

```
plt.xticks( np.arange(4),
           ('Berlin', 'London', 'Hamburg', 'Toronto') )
```

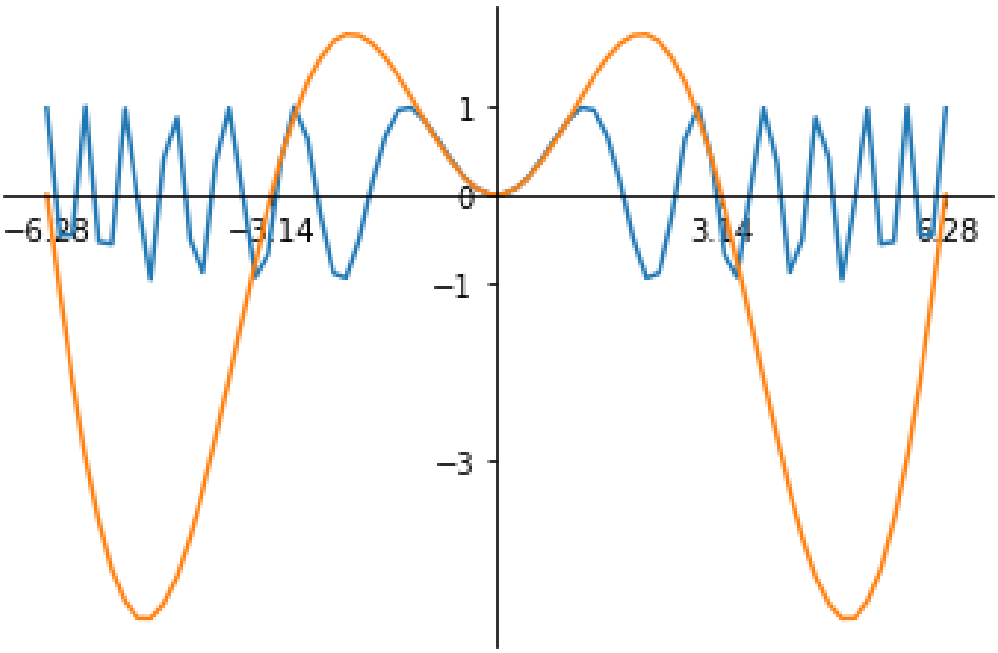
```
([<matplotlib.axis.XTick at 0x7f179fbfc4e0>,
  <matplotlib.axis.XTick at 0x7f179facb320>,
  <matplotlib.axis.XTick at 0x7f179facb208>,
  <matplotlib.axis.XTick at 0x7f179fae84e0>],
 <a list of 4 Text xticklabel objects>)
```



Gehen wir nochmal zurück zum vorigen Beispiel der Trigonometrischen Funktionen. Die meisten werden eine Beschriftung der X-Achse in Teilen bzw. Vielfachen von pi bevorzugen:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(X**2)
F2 = X * np.sin(X)

# aktuelle Achsen werden zurückgeliefert,
# falls notwendig, werden sie erzeugt:
ax = plt.gca()

# der obere rechte Spine wird unsichtbar gemacht:
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')

# der untere Spine wird in Position y=0  gebracht:
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
```

```
# der linke Spine wird nach rechts zu x == 0 bewegt:
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

plt.xticks( [-6.28, -3.14, 3.14, 6.28])
plt.yticks([-3, -1, 0, +1, 3])
plt.plot(X, F1)
plt.plot(X, F2)

plt.show()
```

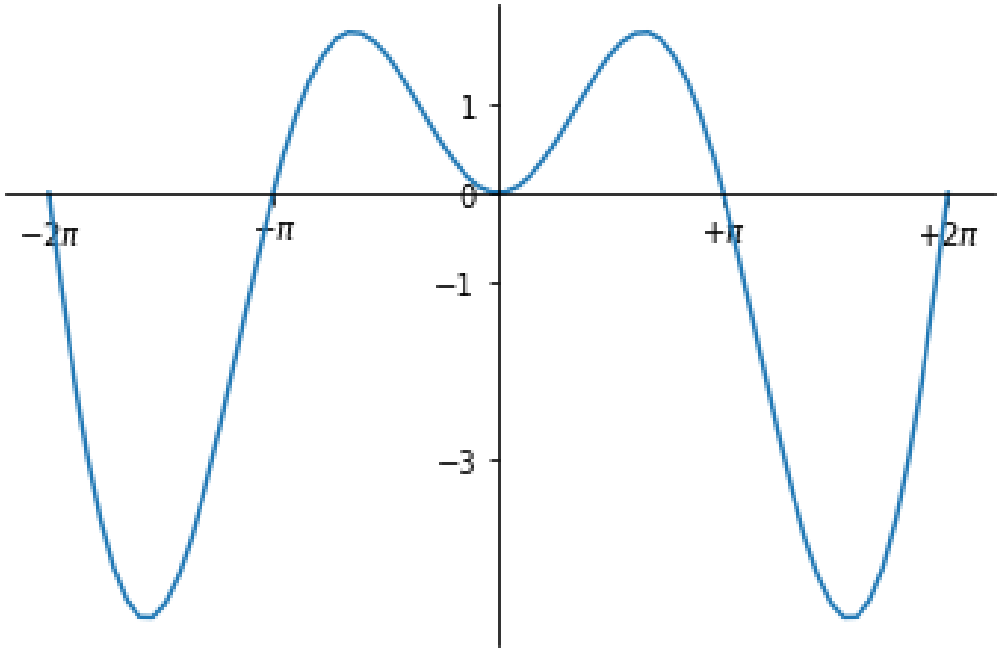


Verändern der Achsenbeschriftungen

Wir möchten nun die Beschriftungen der X-Achse umbenennen und durch eigene Markierungen ersetzen. Wir benutzen dafür ebenfalls die Methode `xticks` wie auch schon im vorigen Beispiel. Diesmal jedoch, rufen wir `xticks` mit zwei Parametern auf: Der erste ist die gleiche Liste, die wir auch schon vorher benutzt haben, d.h. Positionen auf der X-Achse, an denen die Ticks gesetzt werden sollen. Der zweite Parameter ist eine Liste mit gleicher Anzahl von Elementen mit den entsprechenden LaTeX-Tick-Markierungen, d.h. der Text, der anstelle der Werte stehen soll. Die LaTeX-Schreibweise muss ein raw-String sein, um den Escape-Mechanismus von Python abzustellen, da in LaTeX das Backslash häufig genutzt wird.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)

F1 = X * np.sin(X)
```

```
ax = plt.gca()
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

# Labelung der X-Achse:
plt.xticks( [-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$', r'$+\pi$', r'$+2\pi$'])
plt.yticks([-3, -1, 0, +1, 3])


plt.plot(X, F1)

plt.show()
```



Justierung der Tick-Beschriftungen

Wir wollen die Lesbarkeit der Tick-Beschriftungen erhöhen. Dazu vergrößern wir die Schrift und zeichnen diese auf einen halbtransparenten Hintergrund.

```
print(ax.get_xticklabels())
```

```
<a list of 4 Text xticklabel objects>
```

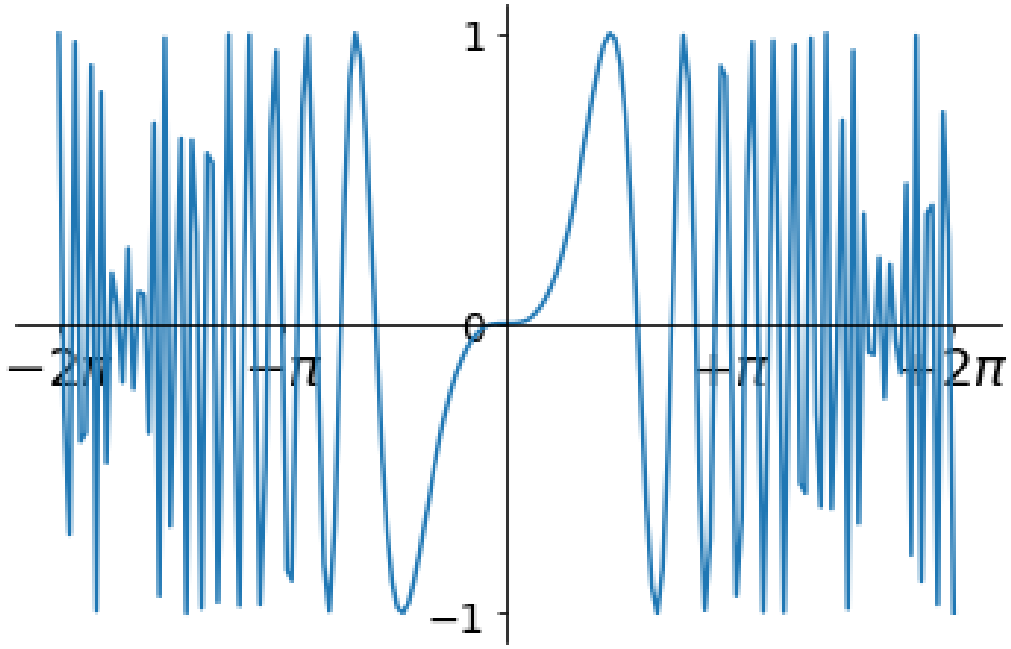
```
for xtick in ax.get_xticklabels():
    print(xtick)
```

```
Text(-6.28, 0, '$-2\\pi$')
Text(-3.14, 0, '$-\\pi$')
Text(3.14, 0, '$+\\pi$')
Text(6.28, 0, '$+2\\pi$')
```

Jetzt vergrößern wir die Schrift und stellen die Transparenz ein:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 170, endpoint=True)
F1 = np.sin(X**3 / 2)

ax = plt.gca()
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

plt.xticks( [-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$', r'$+\pi$', r'$+2\pi$'])
plt.yticks([-3, -1, 0, +1, 3])


for xtick in ax.get_xticklabels():
    xtick.set_fontsize(18)
    xtick.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.7 ))

for ytick in ax.get_yticklabels():
    ytick.set_fontsize(14)
    ytick.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.7 ))

plt.plot(X, F1, label="$sin(x)$")
```

```
[<matplotlib.lines.Line2D at 0x7f179fa81518>]
```


1
0
−1
−2π
−π
π
2π

15 Matplotlib Legenden And Annotationen

15.0.1 Matplotlib Tutorial: Legenden und Kommentare hinzufügen

Legende hinzufügen

Wenn wir uns die Linien-Graphen der vorigen Beispiele anschauen, dann merken wir, dass wir uns immer den Code anschauen müssen, um zu verstehen, welche Art von Funktion dargestellt wird. Diese Information sollte der Einfachheit halber direkt im Diagramm erscheinen. Dafür werden Legenden verwendet. Der Begriff stammt aus dem Lateinischen und steht für "das, was zu lesen ist". Also was gelesen werden muss, um den Graphen zu verstehen.

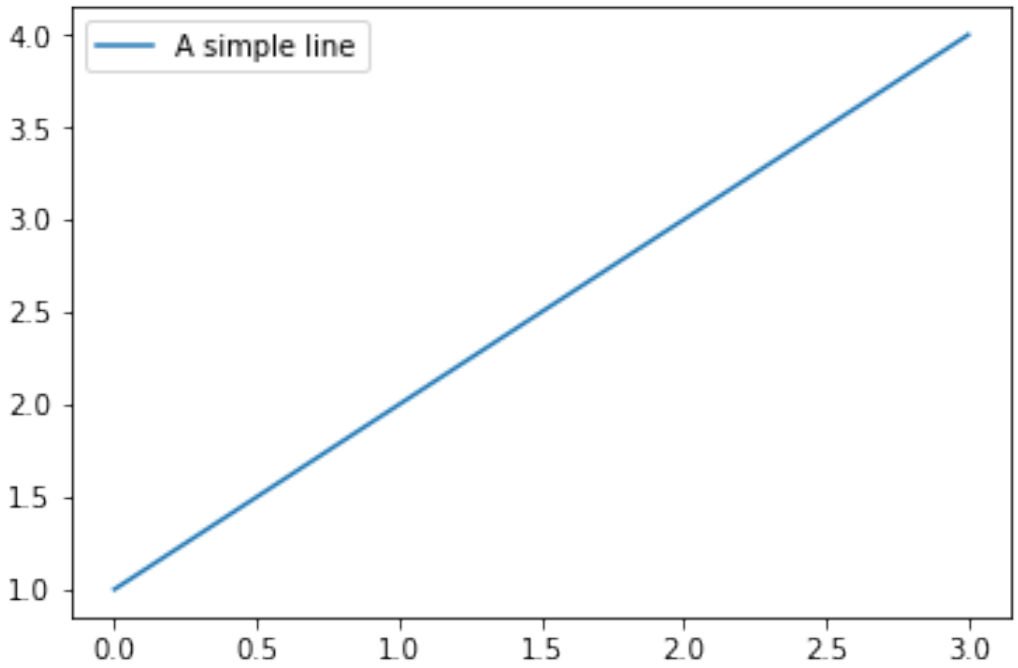
Bevor Legenden in mathematischen Graphen Verwendung fanden, wurden sie auf Karten verwendet. Legenden – wie sie auf Karten zu finden waren – haben die Bildsprache oder Symbolik auf der Karte erläutert. Auf Graphen erläutern sie die Funktion oder die Werte, die hinter den verschiedenen Linien des Graphen liegen.

Im Folgenden demonstrieren wir ein einfaches Beispiel, wie eine Legende auf dem Graphen platziert werden kann. Eine Legende enthält einen oder mehrere Einträge. Jeder Eintrag besteht aus einem Schlüssel und einer Beschriftung.

Die pyplot-Funktion `legend(*args, **kwargs)` platziert eine Legende im Plot.

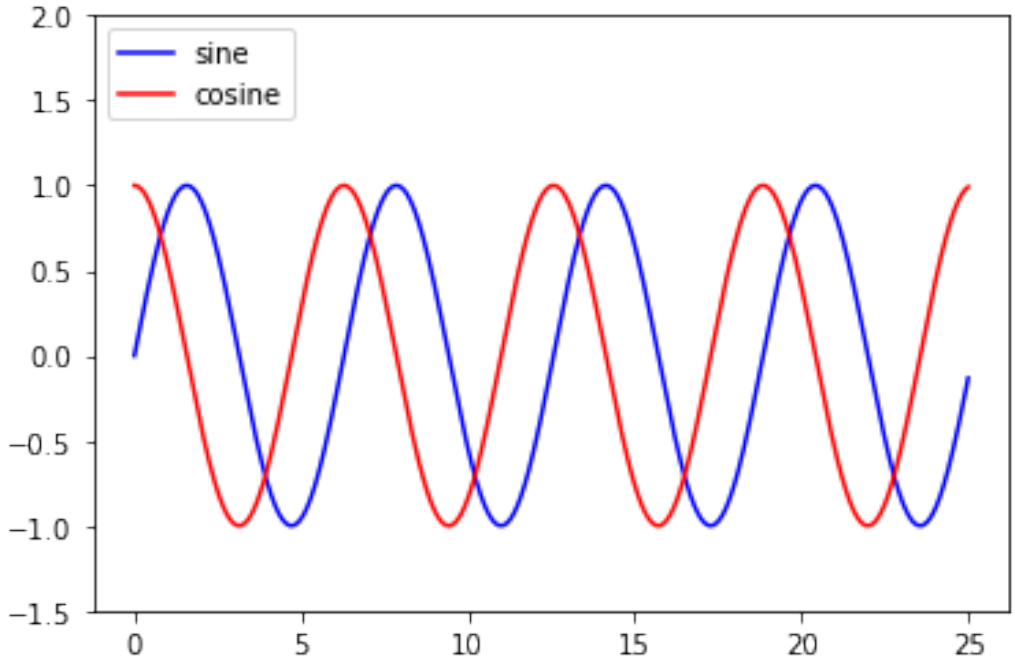
Alles, was wir tun müssen, um eine Legende für Linien zu erstellen, die bereits im Plot existieren, ist der einfache Aufruf der Funktion `legend` mit einem iterierbaren Array aus Strings. Eins für jedes Element der Legende. Zum Beispiel:

```
import numpy as np
import matplotlib.pyplot as plt

ax = plt.gca()

ax.plot([1, 2, 3, 4])
ax.legend(['A simple line'])
plt.show()
```



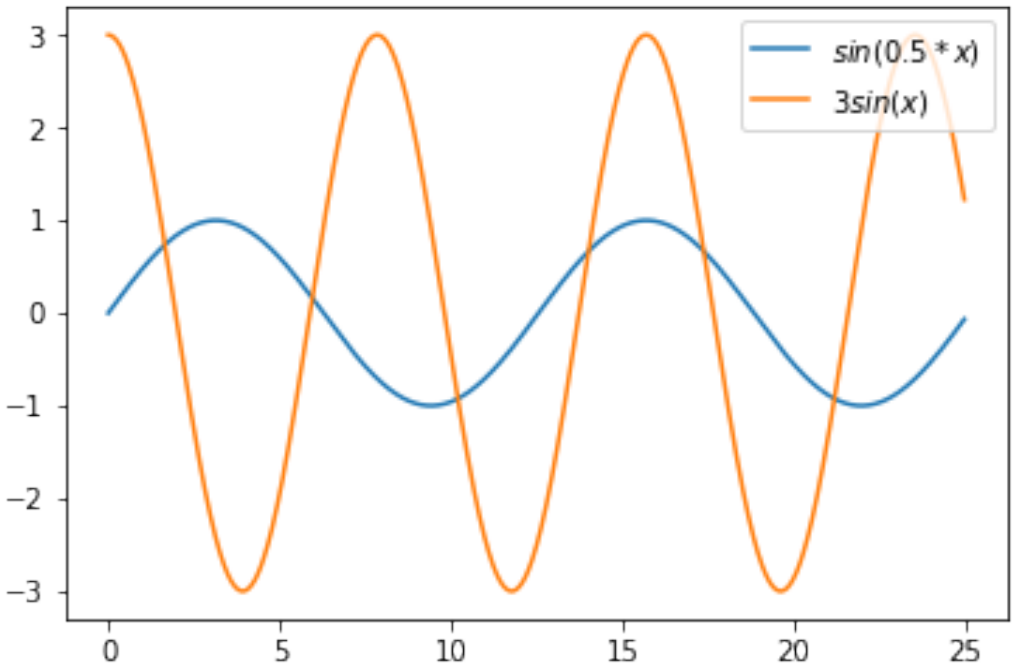
Wenn wir eine Beschriftung hinzufügen, um die Funktion zu zeichnen (plotten), wird der Wert automatisch als Beschriftung im legend-Kommando verwendet. Das einzige Argument, welches die legend-Funktion braucht, ist das Positions-Argument `loc`:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 25, 1000)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, '-b', label='sine')
plt.plot(x, y2, '-r', label='cosine')
plt.legend(loc='upper left')
plt.ylim(-1.5, 2.0)
plt.show()
```



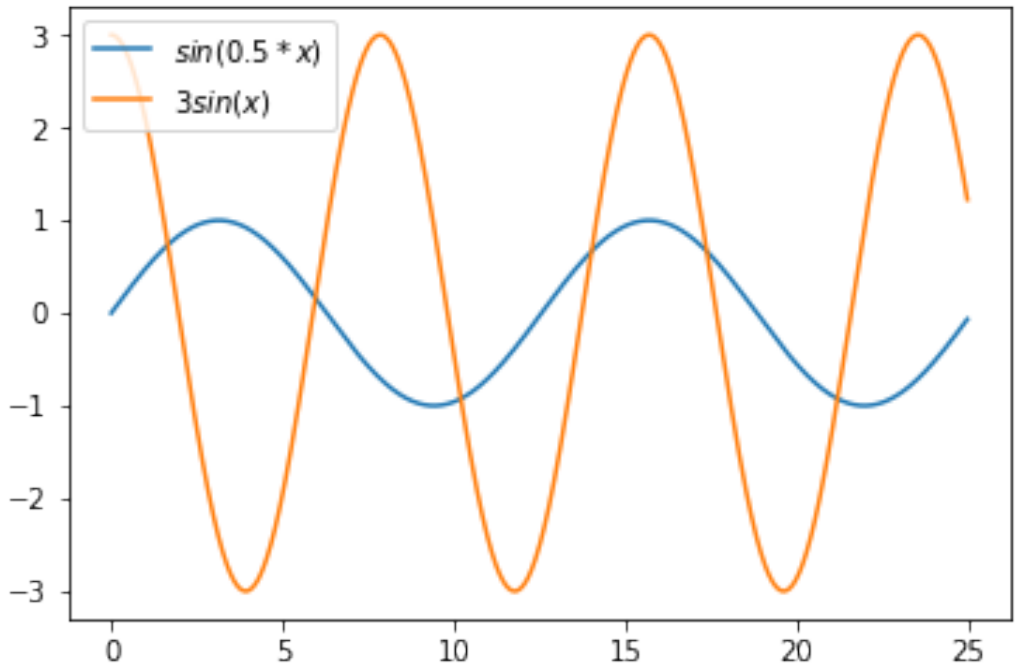
Es folgt nun ein Beispiel mit der Legende in der rechten oberen Ecke:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0, 25, 1000)

F1 = np.sin(0.5 * X)
F2 = 3 * np.cos(0.8*X)

plt.plot(X, F1, label="$sin(0.5 * x)$")
plt.plot(X, F2, label="$3 sin(x)$")
plt.legend(loc='upper right')
plt.show()
```



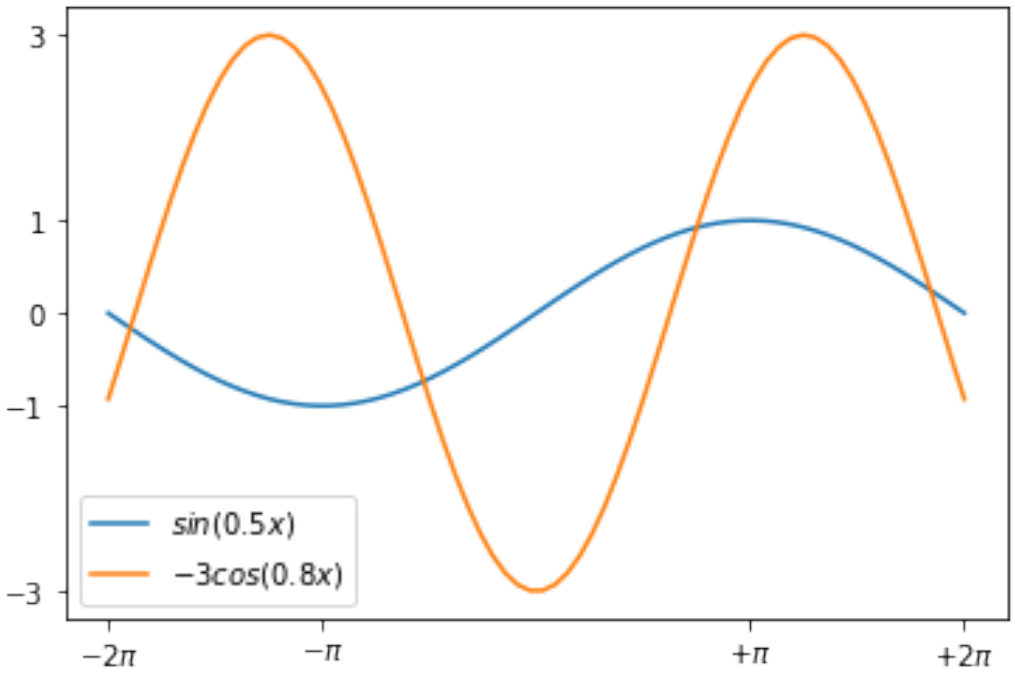
In den meisten Fällen weiß man jedoch nicht, wie das Ergebnis aussehen wird, bevor es nicht ausgegeben wurde. Möglicherweise könnte die Legende einen wichtigen Teil des Plots überdecken. Wenn Sie nicht wissen, wie die Daten aussehen werden, können Sie `best` als Argument für `loc` verwenden. Matplotlib wird automatisch versuchen, die bestmögliche Position für die Legende zu finden:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0, 25, 1000)

F1 = np.sin(0.5 * X)
F2 = 3 * np.cos(0.8*X)

plt.plot(X, F1, label="$sin(0.5 * x)$")
plt.plot(X, F2, label="$3 sin(x)$")
plt.legend(loc='best')
plt.show()
```

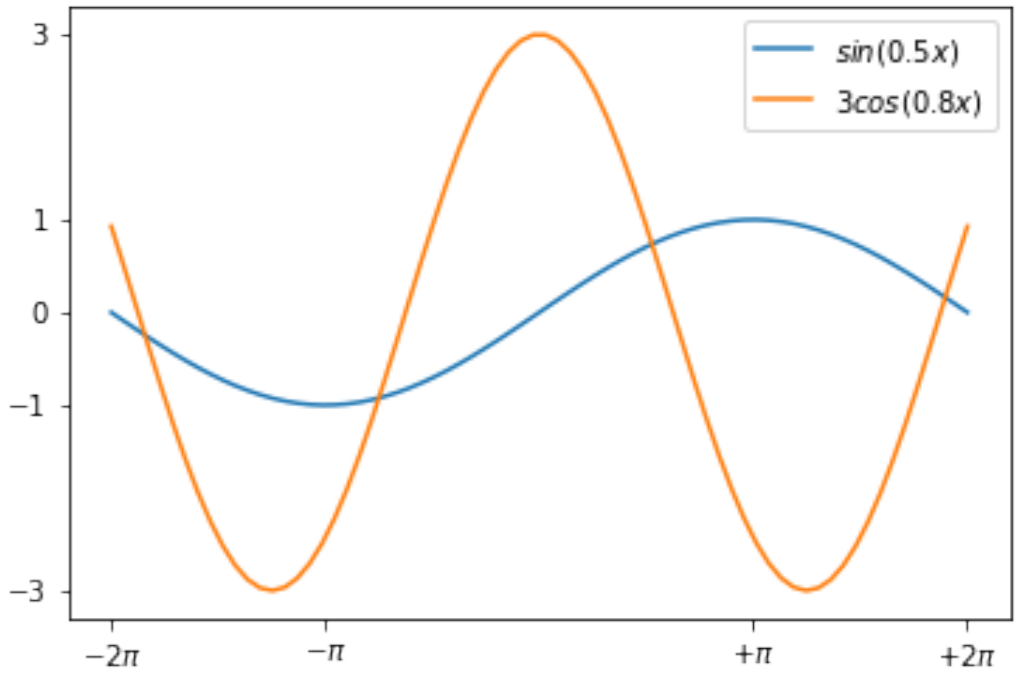


In den folgenden beiden Beispielen kann man sehen, dass `loc='best'` sehr gut funktioniert:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(0.5*X)
F2 = -3 * np.cos(0.8*X)

plt.xticks( [-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$', r'$+\pi$', r'$+2\pi$'])
plt.yticks([-3, -1, 0, +1, 3])
plt.plot(X, F1, label="$sin(0.5x)$")
plt.plot(X, F2, label="$-3 cos(0.8x)$")
plt.legend(loc='best')

plt.show()
```




```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(0.5*X)
F2 = 3 * np.cos(0.8*X)

plt.xticks( [-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$', r'$+\pi$', r'$+2\pi$'])
plt.yticks([-3, -1, 0, +1, 3])
plt.plot(X, F1, label="$sin(0.5x)$")
plt.plot(X, F2, label="$3 cos(0.8x)$")
plt.legend(loc='best')

plt.show()
```



Kommentare

Natürlich hat die Sinus-Funktion “langweilige” und “interessante” Werte. Nehmen wir an, dass uns speziell der Wert von

$3 * sin(3 * pi/4)$

interessiert.

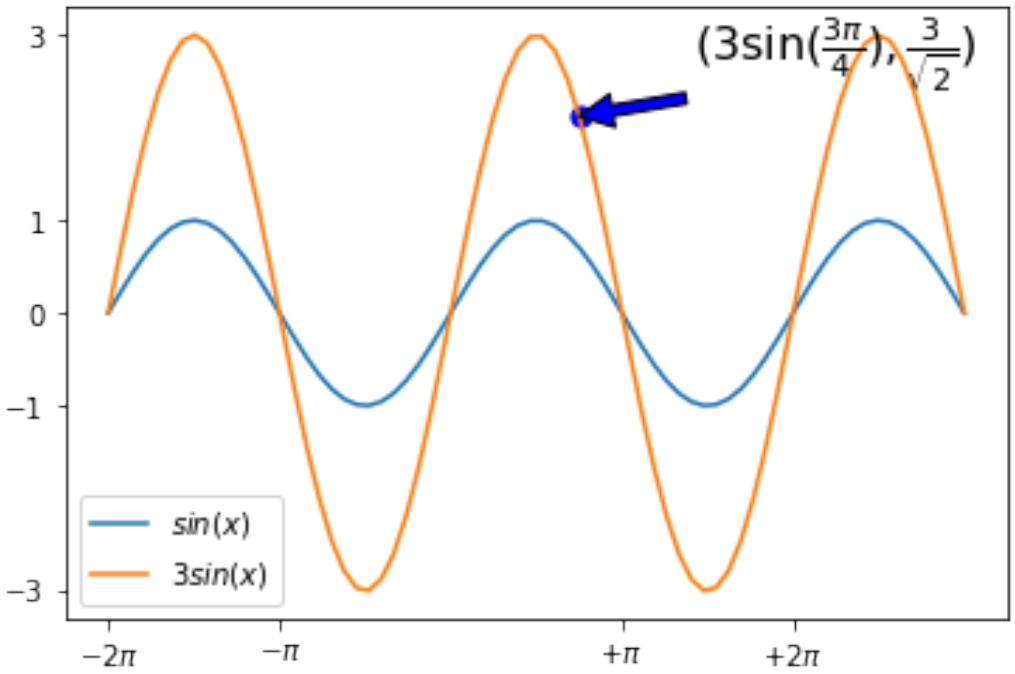
```
import numpy as np

print(3 * np.sin(3 * np.pi/4))
```

```
2.121320343559643
```

Der Zahlenwert sieht nicht sehr speziell aus. Wenn wir aber eine symbolische Berechnung durchführen, resultiert daraus $\frac{3}{\sqrt{2}}$. Jetzt möchten wir genau diesen Punkt auf dem Graph markieren. Dies erreichen wir mit der Funktion `annotate`.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 3 * np.pi, 70, endpoint=True)
F1 = np.sin(X)
F2 = 3 * np.sin(X)
ax = plt.gca()

plt.xticks( [-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$', r'$+\pi$', r'$+2\pi$'])
plt.yticks([-3, -1, 0, +1, 3])

x = 3 * np.pi / 4
```

```
# mit scatter kann man einen einzelnen Punkt erzeugen:
plt.scatter([x,],[3 * np.sin(x),], 50, color ='blue')

# Die in annotate verwendete Notation kommt von LaTeX:
plt.annotate(r'$(3\sin(\frac{3\pi}{4}),\frac{3}{\sqrt{2}})$',
         xy=(x, 3 * np.sin(x)),
         xycoords='data',
         xytext=(+40, +20),
         textcoords='offset points',
         fontsize=16,
         arrowprops=dict(facecolor='blue'))

plt.plot(X, F1, label="$sin(x)$")
plt.plot(X, F2, label="$3 sin(x)$")
plt.legend(loc='lower left')

plt.show()
```



Dafür mussten wir der `annotate`-Funktion einige Informationen als Parameter bereitstellen.

Parameter	Bedeutung
xy	Koordinaten der Pfeilspitze
xytext	Koordinaten der Text-Position

Die `xy`- und `xytext`-Positionen sind in unserem Beispiel in den Daten-Koordinaten enthalten. Es gibt weitere Koordinaten-Systeme, aus denen wir wählen können. Das Koordinaten-System von `xy` und `xytext` kann über String-Werte mit `xycoords` und `textcoords` spezifiziert werden. Der Default-Wert ist “data”:

String-Wert	Koordinaten-System
figure points	Punkte von der linken unteren Ecke der Abbildung
figure pixels	Pixel von der linken unteren Ecke der Abbildung
figure fraction	0,0 ist links unten und 1,1 ist rechts oben in der Abbildung
axes points	Punkte von der linken unteren Ecke der Achsen
axes pixels	Pixel von der linken unteren Ecke der Achsen
axes fraction	proportionaler Anteil, 0,0 ist links unten und 1,1 ist rechts oben
data	Verwende das Achsen-Daten-Koordinaten-System

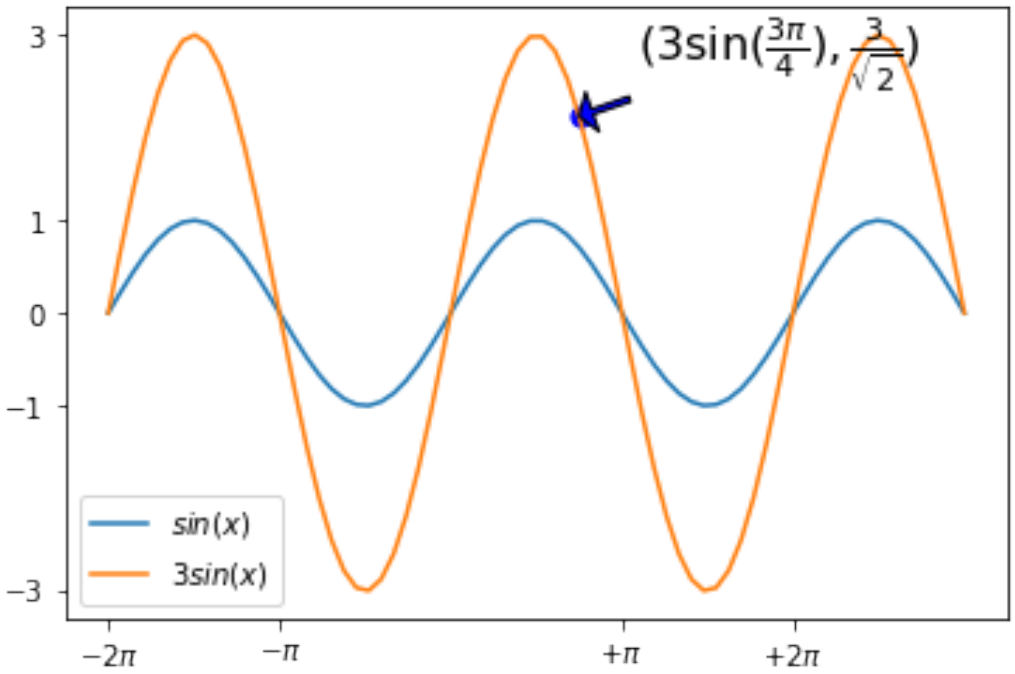
Zusätzlich können die Eigenschaften des Pfeils spezifiziert werden. Dafür müssen wir ein Dictionary mit Pfeil-Eigenschaften als Parameter `arrowprops` bereitstellen:

arrowprops Schlüssel	Beschreibung
width	Die Breite des Pfeils in Punkten
headlength	Der Teil, der vom Pfeilkopf eingenommen wird.
headwidth	Die Breite der Pfeilspitzen-Basis in Punkten
shrink	Verschiebe die Spitze und Basis um ein paar Prozent vom Kommentar-Punkt und -Text
**kwargs	ein Schlüssel für matplotlib.patches.Polygon, z.B. facecolor

Im folgenden Beispiel verändern wir das Erscheinungsbild des Pfeils aus unserem vorigen Beispiel:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 3 * np.pi, 70, endpoint=True)
F1 = np.sin(X)
F2 = 3 * np.sin(X)
ax = plt.gca()

plt.xticks( [-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$', r'$+\pi$', r'$+2\pi$'])
plt.yticks([-3, -1, 0, +1, 3])

x = 3 * np.pi / 4

plt.scatter([x,],[3 * np.sin(x),], 50, color ='blue')
```

```
plt.annotate(r'$(3\sin(\frac{3\pi}{4}),\frac{3}{\sqrt{2}})$',
         xy=(x, 3 * np.sin(x)),
         xycoords='data',
         xytext=(+20, +20),
         textcoords='offset points',
         fontsize=16,
         arrowprops=dict(facecolor='blue',
                         headwidth=15,
                         headlength=5,
                         width=2))

plt.plot(X, F1, label="$sin(x)$")
plt.plot(X, F2, label="$3 sin(x)$")
plt.legend(loc='lower left')

plt.show()
```



16 Matplotlib Unterdiagramme

16.0.1 Matplotlib-Tutorial: Mehrfache Plots und Doppelachsen

WEBOFF

In den bisherigen Kapiteln des Matplotlib-Tutorials haben wir in zahlreichen Beispiele gezeigt, wie wir Diagramme und Graphen erzeugen können. Ein häufig gestellte Frage ist, wie man mehrere Plots in einem Diagramm unterbringen kann.

Im einfachsten Fall heißt das, dass wir eine Kurve haben, und wir eine weitere Kurve darüber legen. Der interessantere Fall ist jedoch, wenn zwei Plots in einem Fenster gewünscht werden. In einem Fenster bedeutet, dass es zwei Unterdiagramme geben soll, d.h. dass diese nicht übereinander gezeichnet werden. Die Idee ist, mehr als einen Graphen in einem Fenster zu haben, und jeder Graph erscheint in seinem eigenen Unterdiagramm.

Wir stellen zwei verschiedene Wege vor, wie dies erreicht werden kann:

- `subplot`
- `gridspec`

Wir sind der Meinung, dass `gridspec` die beste Option ist, weil es einfacher in der Anwendung ist, wenn das Layout komplexer wird.

Mehrere Abbildungen und Achsen

Die Funktion `subplot` hat folgende Parameter:

`subplot(nrows, ncols, plot_number)`

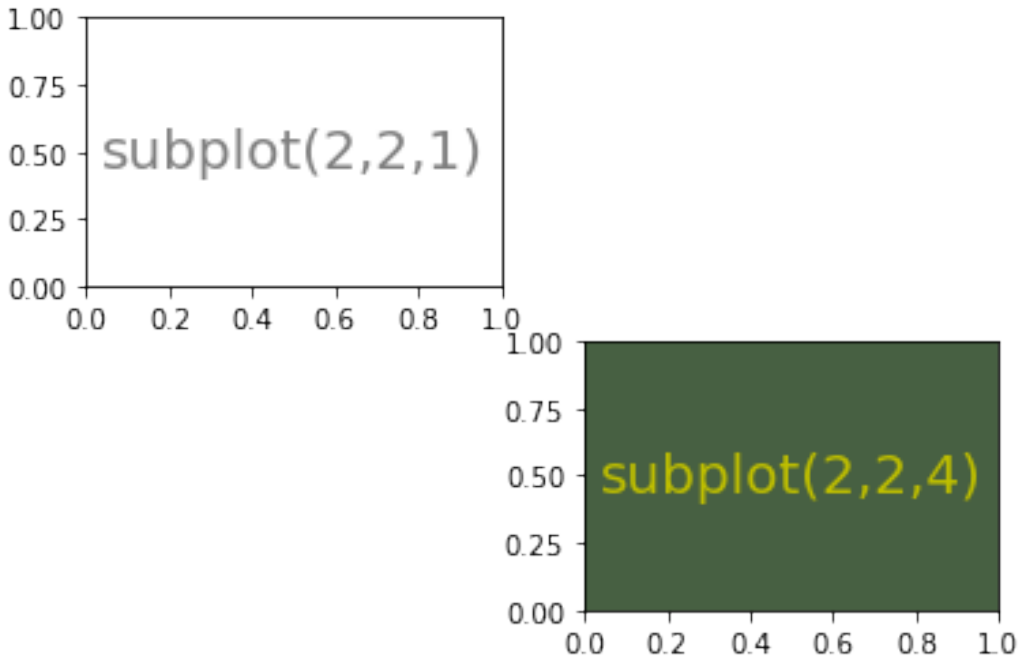
Wenn ein Unterdiagramm auf eine Abbildung angewendet wird, so wird die Abbildung theoretisch aufgeteilt in `nrows * ncols` Unterachsen. Der Parameter `plot_number` bezeichnet den Subplot, den der Funktionsaufruf erstellen muss. `plot_number` kann einen Wert zwischen 1 und dem Maximum von `nrows * ncols` annehmen.

Wenn der Wert der drei Parameter kleiner als 10 ist, kann die Funktion mit einem Integer Wert aufgerufen werden, wobei die Hunderter `nrows`, die Zehner `ncols` und die Einheiten `plot_number` repräsentieren. Das bedeutet: Statt `subplot(2, 3, 4)` kann `subplot(234)` geschrieben werden.

Im folgenden Beispiel "aktivieren" wir zwei Unterplots in einem "theoretischen" 2x2-Gitter:

```
import matplotlib.pyplot as plt
python_course_green = "#476042"
plt.figure(figsize=(6, 4))
plt.subplot(221) # äquivalent zu: plt.subplot(2, 2, 1)
plt.text(0.5, # x-Koordinate, 0 ganz links, 1 ganz rechts
         0.5, # y-Koordinate, 0 ganz oben, 1 ganz unten
         'subplot(2,2,1)', # der Text der ausgegeben wird
```

```
        horizontalalignment='center', # abgekürzt 'ha'
        verticalalignment='center', # abgekürzt 'va'
        fontsize=20, # kann auch  'font' genannt werden
        alpha=.5 # float (0.0 tranparent bis 1.0 undurchsichtig)
        )
plt.subplot(224, facecolor=python_course_green)
plt.text(0.5, 0.5,
        'subplot(2,2,4)',
        ha='center', va='center',
        fontsize=20,
        color="y")
plt.show()
```



Für unsere Absichten benötigen wir keine Ticks auf den Achsen. Wir können sie loswerden, indem wir ein leeres Tupel setzen und folgenden Code ergänzen:

```
plt.xticks(())
plt.yticks(())
```

Das gesamte Programm sieht dann wie folgt aus:

```
import matplotlib.pyplot as plt

python_course_green = "#476042"
# figsize werden wir erst später erklären:
plt.figure(figsize=(6, 4)) # Größe des Plots
plt.subplot(221) # öquivalent zu: plt.subplot(2, 2, 1)
plt.xticks(())
plt.yticks(())
plt.text(0.5,
        0.5,
```

```
         'subplot(2,2,1)',
         horizontalalignment='center',
         verticalalignment='center',
         fontsize=20,
         alpha=.5
         )
plt.subplot(224, facecolor=python_course_green)
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5,
         'subplot(2,2,4)',
         ha='center', va='center',
         fontsize=20,
         color="y")
plt.show()
```

Der vorige Ansatz ist durchaus akzeptabel. Jedoch ist es ein besserer Stil, Instanzen der Figure-Klasse im Sinne der objektorientierten Programmierung zu verwenden. Wir demonstrieren dies, indem wir das vorige Beispiel umschreiben. In diesem Fall müssen wir die `add_subplot`-Methode auf das Figure-Objekt anwenden.

Wir empfehlen dazu die Kapitel zu OOP in unserem Python-Tutorial zu lesen, wenn Sie nicht mit objektorientierter Programmierung vertraut sind:

Allgemeine Einführung zur objektorientierten Programmierung (OOP)

Klassen- und Instanz-Attribute

Properties vs. Getters und Setters

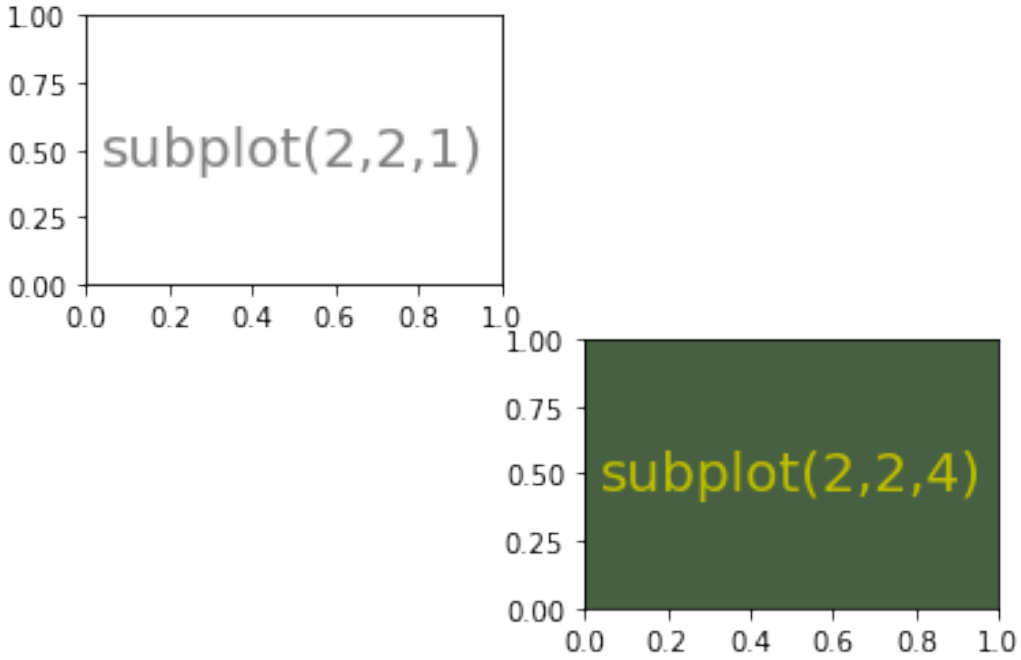
Vererbung

Mehrfachvererbung

Magische Methoden und Opereator-Überladung

Die überarbeitete Version des Codes sieht wie folgt aus:

```
import matplotlib.pyplot as plt

python_course_green = "#476042"
fig = plt.figure(figsize=(6, 4))
sub1 = fig.add_subplot(221) # alternativ: plt.subplot(2, 2, 1)

sub1.text(0.5, # x-Koordinate: 0 ganz links, 1 ganz rechts
          0.5, # y-Koordinate: 0 ganz oben, 1 ganz unten
          'subplot(2,2,1)', # der Text der ausgegeben wird
          horizontalalignment='center', # Abkürzung 'ha'
          verticalalignment='center', # sAbkürzung 'va'
          fontsize=20, #  'font' ist äquivalent
          alpha=.5 # Floatzahl von 0.0 transparent bis 1.0 opak
          )
sub2 = fig.add_subplot(224, facecolor=python_course_green)
sub2.text(0.5, 0.5,
          'subplot(2,2,4)',
          ha='center', va='center',
          fontsize=20,
          color="y")
plt.show()
```



Auch in diesem Fall wollen wir die Ticks wieder loswerden. Dieses Mal können wir nicht `plt.xticks(())` und `plt.yticks(())` benutzen. Wir müssen die Methoden `set_xticks(())` und `set_yticks(())` stattdessen benutzen.

```
import matplotlib.pyplot as plt

python_course_green = "#476042"
fig = plt.figure(figsize=(6, 4))
```

```
sub1 = fig.add_subplot(221) # alternativ: plt.subplot(2, 2, 1)
sub1.set_xticks([])
sub1.set_yticks([])
sub1.text(0.5, # x-Koordinate: 0 ganz links, 1 ganz rechts
          0.5, # y-Koordinate: 0 ganz oben, 1 ganz unten
          'subplot(2,2,1)', # der Text der ausgegeben wird
          horizontalalignment='center', # Abkürzung 'ha'
          verticalalignment='center', # sAbkürzung 'va'
          fontsize=20, #  'font' ist äquivalent
          alpha=.5 # Floatzahl von 0.0 transparent bis 1.0 opak
          )
sub2 = fig.add_subplot(224, facecolor=python_course_green)
sub2.set_xticks([])
sub2.set_yticks([])
sub2.text(0.5, 0.5,
          'subplot(2,2,4)',
          ha='center', va='center',
          fontsize=20,
          color="y")
plt.show()
```

Wenn alle Unterplots des 2x2-Gitters aktiviert sind, sieht es wie folgt aus:

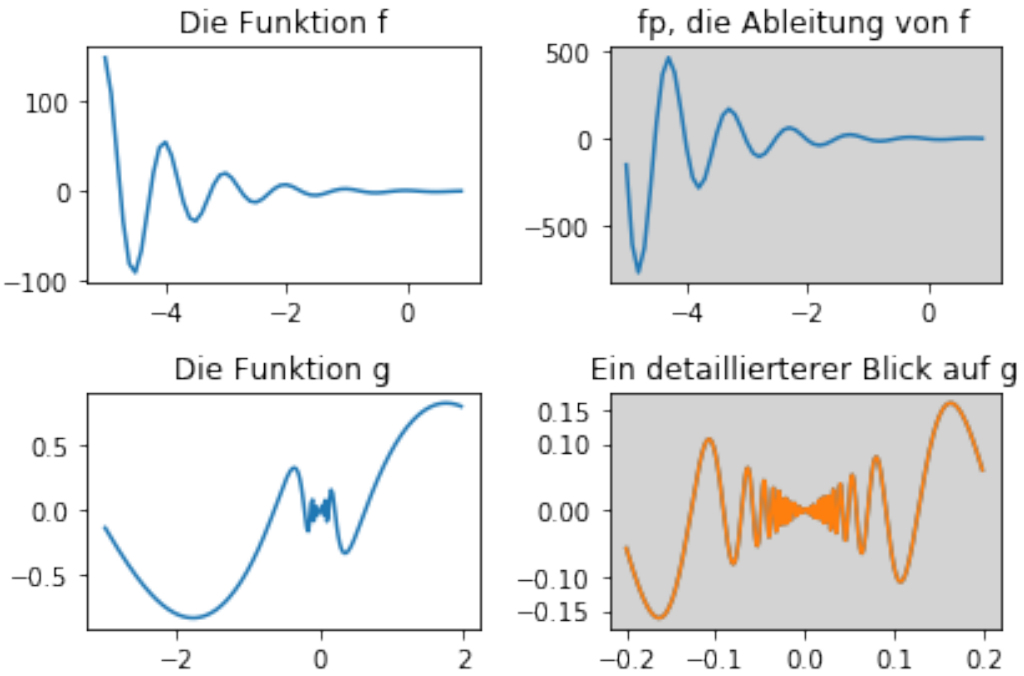
```
import matplotlib.pyplot as plt

python_course_green = "#476042"
fig = plt.figure(figsize=(6, 4))
sub1 = plt.subplot(2, 2, 1)
sub1.set_xticks(())
sub1.set_yticks(())
sub1.text(0.5, 0.5, 'subplot(2,2,1)', ha='center', va='center',
        size=20, alpha=.5)
```

```
sub2 = plt.subplot(2, 2, 2)
sub2.set_xticks(())
sub2.set_yticks(())
sub2.text(0.5, 0.5, 'subplot(2,2,2)', ha='center', va='center',
        size=20, alpha=.5)

sub3 = plt.subplot(2, 2, 3)
sub3.set_xticks(())
sub3.set_yticks(())
sub3.text(0.5, 0.5, 'subplot(2,2,3)', ha='center', va='center',
        size=20, alpha=.5)

sub4 = plt.subplot(2, 2, 4, facecolor=python_course_green)
sub4.set_xticks(())
sub4.set_yticks(())
sub4.text(0.5, 0.5, 'subplot(2,2,4)', ha='center', va='center',
        size=20, alpha=.5, color="y")

fig.tight_layout()
plt.show()
```

Das vorige Beispiel zeigt lediglich, wie man das Unterplot-Design erstellen kann. Normalerweise möchte man die subplot-Funktion benutzen, um mehrere Graphen darzustellen. Wir demonstrieren nun, wie man das vorige Unterplot-Design mit einigen Graphen füllt:

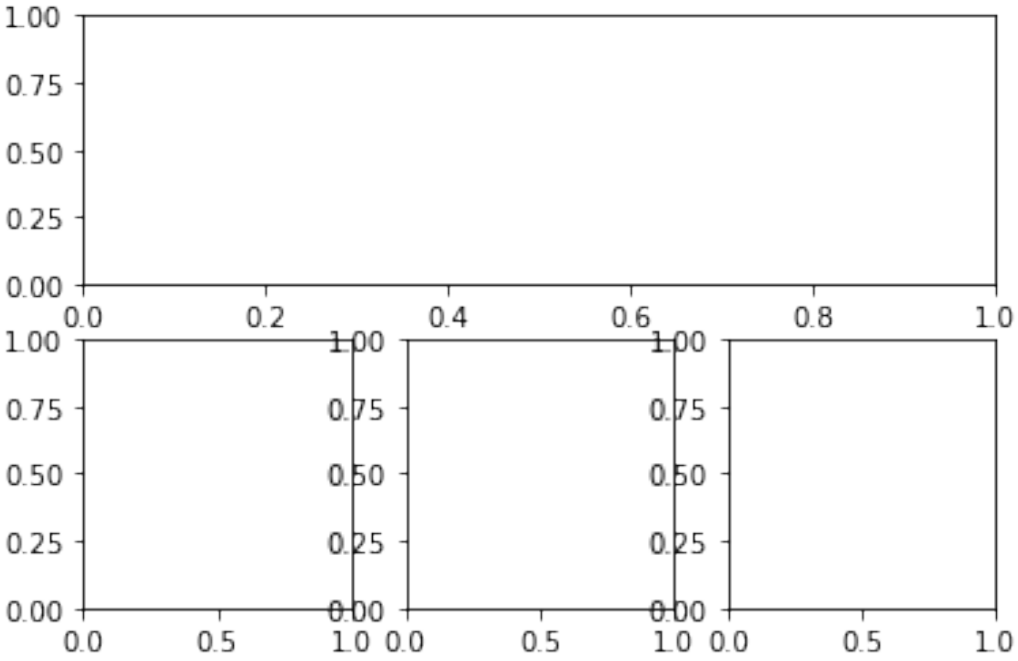
```
import numpy as np
from numpy import e, pi, sin, exp, cos
import matplotlib.pyplot as plt

def f(t):
    return exp(-t) * cos(2*pi*t)
```

```
def fp(t):
    return -2*pi * exp(-t) * sin(2*pi*t) - e**(-t)*cos(2*pi*t)

def g(t):
    return sin(t) * cos(1/(t+0.1))

def g(t):
    return sin(t) * cos(1/(t))


python_course_green = "#476042"
fig = plt.figure(figsize=(6, 4))

t = np.arange(-5.0, 1.0, 0.1)

sub1 = fig.add_subplot(221) # statt plt.subplot(2, 2, 1)
sub1.set_title('Die Funktion f')
sub1.plot(t, f(t))


sub2 = fig.add_subplot(222, facecolor="lightgrey")
sub2.set_title('fp, die Ableitung von f')
sub2.plot(t, fp(t))


t = np.arange(-3.0, 2.0, 0.02)
sub3 = fig.add_subplot(223)
sub3.set_title('Die Funktion g')
sub3.plot(t, g(t))

t = np.arange(-0.2, 0.2, 0.001)
sub4 = fig.add_subplot(224, facecolor="lightgrey")
sub4.set_title('Ein detaillierterer Blick auf g')
sub4.set_xticks([-0.2, -0.1, 0, 0.1, 0.2])
sub4.set_yticks([-0.15, -0.1, 0, 0.1, 0.15])
sub4.plot(t, g(t))

plt.plot(t, g(t))

plt.tight_layout()
plt.show()
```



Weiteres Beispiel:

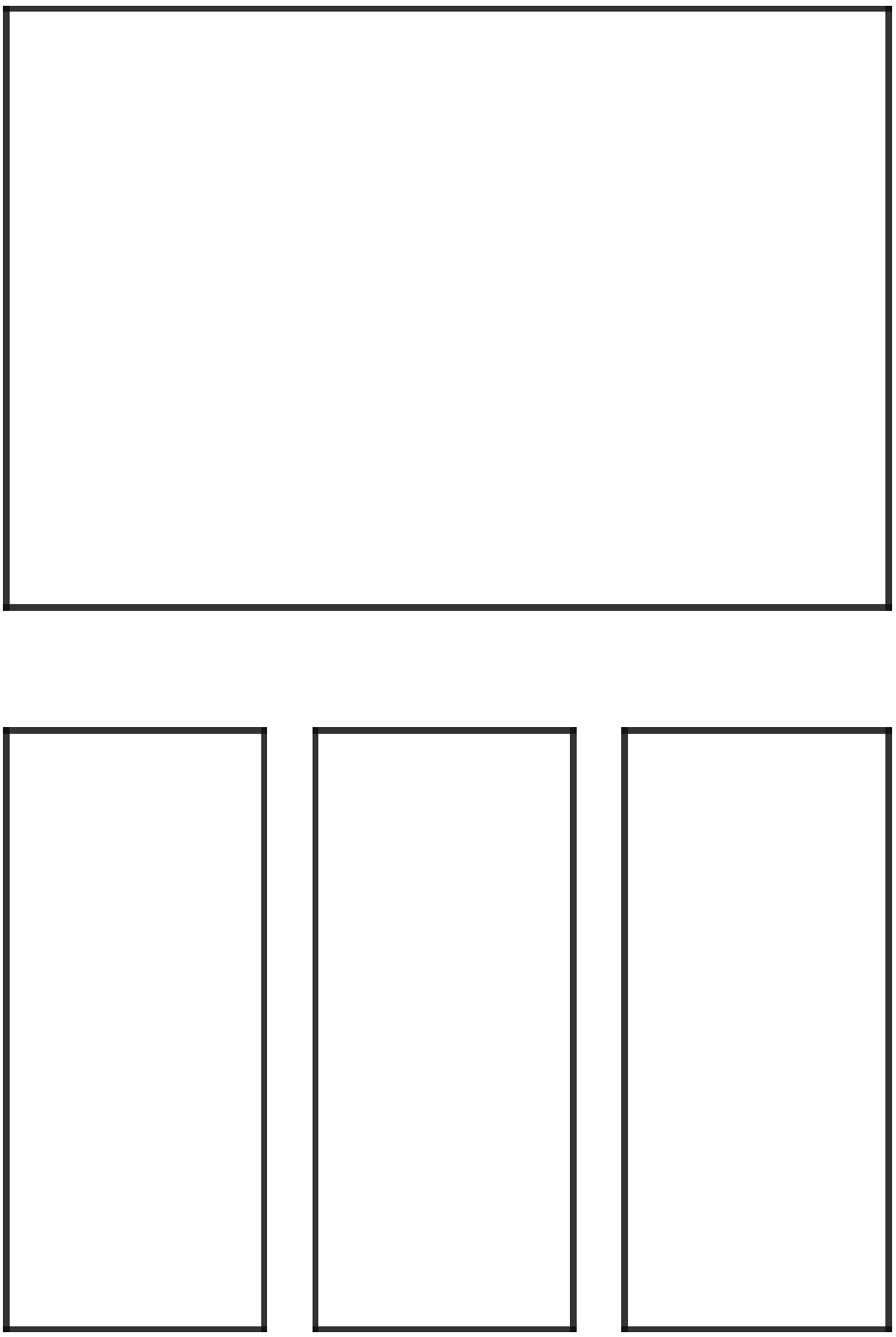
```
import  matplotlib.pyplot as plt

X = [ (2,1,1), (2,3,4), (2,3,5), (2,3,6) ]
for nrows, ncols, plot_number in X:
    plt.subplot(nrows, ncols, plot_number)
```



Das folgende Beispiel zeigt nichts Besonderes. Wir entfernen die `xticks` und experimentieren etwas mit der Größe der Abbildung und der Unterplots. Dafür führen wir nun den

Schlüsselwort-Parameter `figsize` für "figure" ein. `figsize` erwartet ein Tupel mit einem Wert für Breite und Höhe in Inches. Außerdem nutzen wir die Funktion `subplot_adjust` mit deren Schlüsselwort-Parametern `bottom`, `left`, `top` und `right`:

```
import  matplotlib.pyplot as plt

fig =plt.figure(figsize=(2, 3))
fig.subplots_adjust(bottom=0.025,
                    left=0.025,
                    top = 0.975,
                    right=0.975)

X = [ (2,1,1), (2,3,4), (2,3,5), (2,3,6) ]
for nrows, ncols, plot_number in X:
    sub = fig.add_subplot(nrows, ncols, plot_number)
    sub.set_xticks([])
    sub.set_yticks([])
```

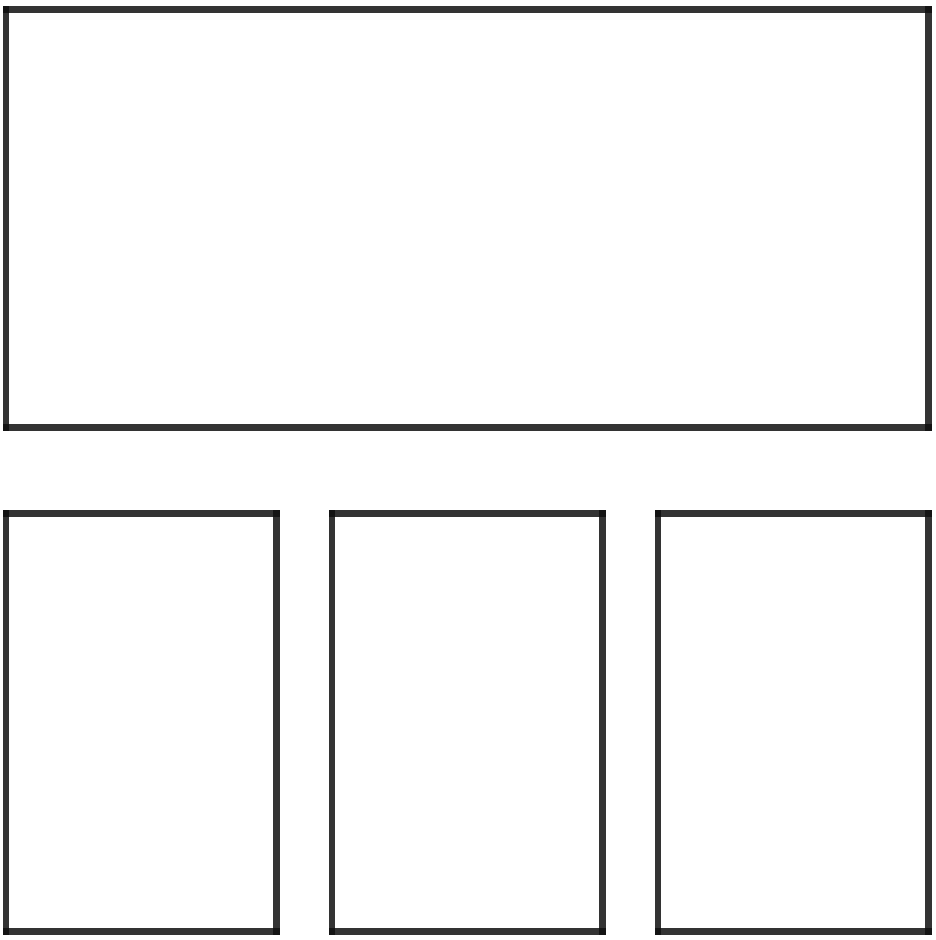


Alternative Lösung:

Um die ersten drei Elemente des 2x3-Gitters zu verbinden, können wir auch eine Tupel-Schreibweise verwenden. In unserem Fall (1,3) in (2,3,(1,3)), um festzulegen, dass die ersten drei Elemente des theoretischen 2x3-Gitters verbunden werden sollen:

```
import  matplotlib.pyplot as plt

fig =plt.figure(figsize=(2.2, 2.2))
```

```
fig.subplots_adjust(bottom=0.025,
                    left=0.025,
                    top = 0.975,
                    right=0.975)

X = [ (2,3,(1,3)), (2,3,4), (2,3,5), (2,3,6) ]
for nrows, ncols, plot_number in X:
    sub = fig.add_subplot(nrows, ncols, plot_number)
    sub.set_xticks([])
    sub.set_yticks([])
```

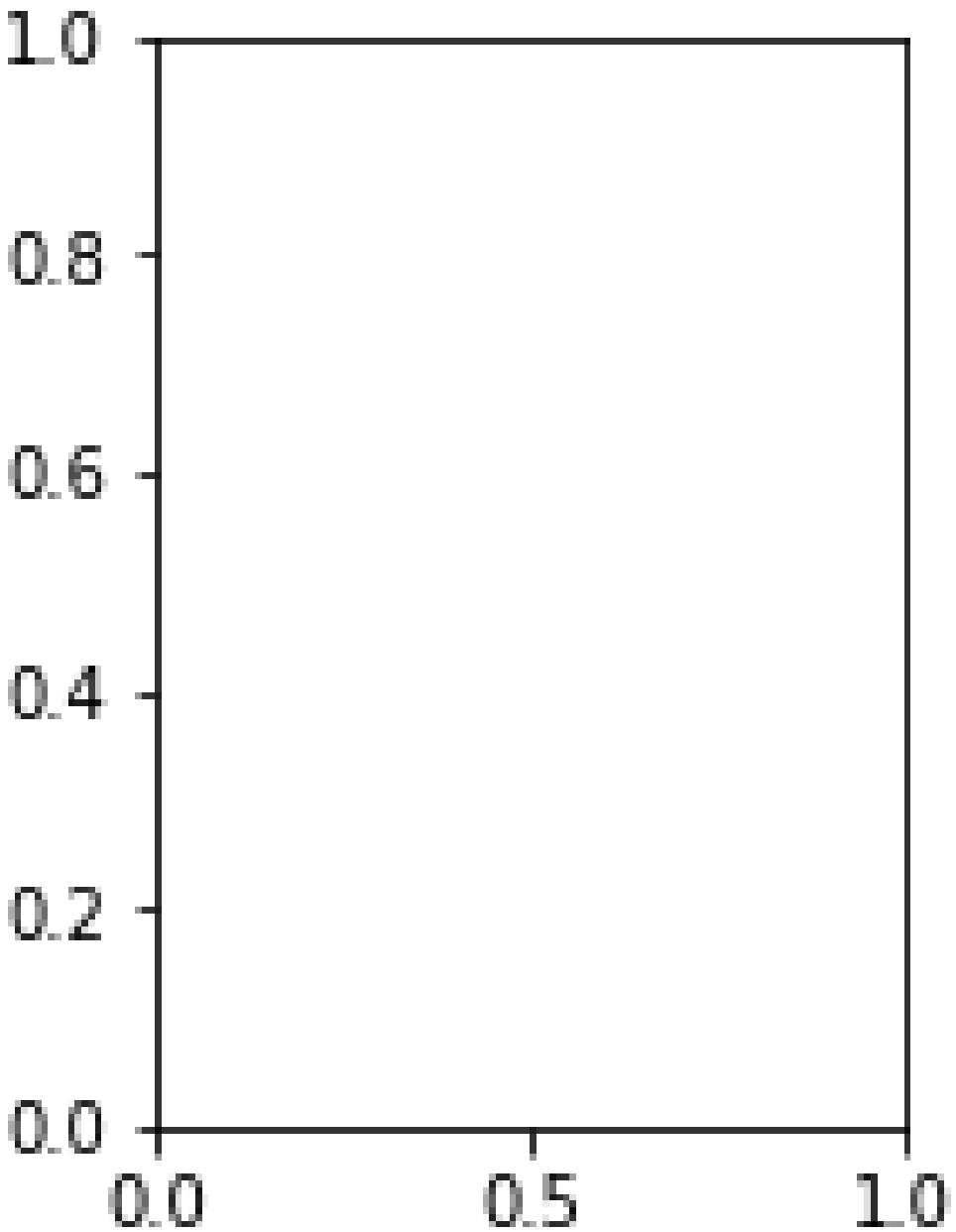


Unterdiagramm mit gridspec

`matplotlib.gridspec` beinhaltet die Klasse `GridSpec`. Diese kann als Alternative zu subplot verwendet werden, um die Geometrie der Unterplots zu spezifizieren. Die Grund-idee hinter GridSpec ist ein "Gitter", englisch "grid". Daher kommt auch der Name "gridspec". Ein Gitter wird erstellt, indem die Anzahl der benötigten Zeilen und Spalten angegeben wird. Anschließend definiert man innnerhalb des Gesamtgitters Unterbereiche oder Unterdiagramme (englisch "subplots").

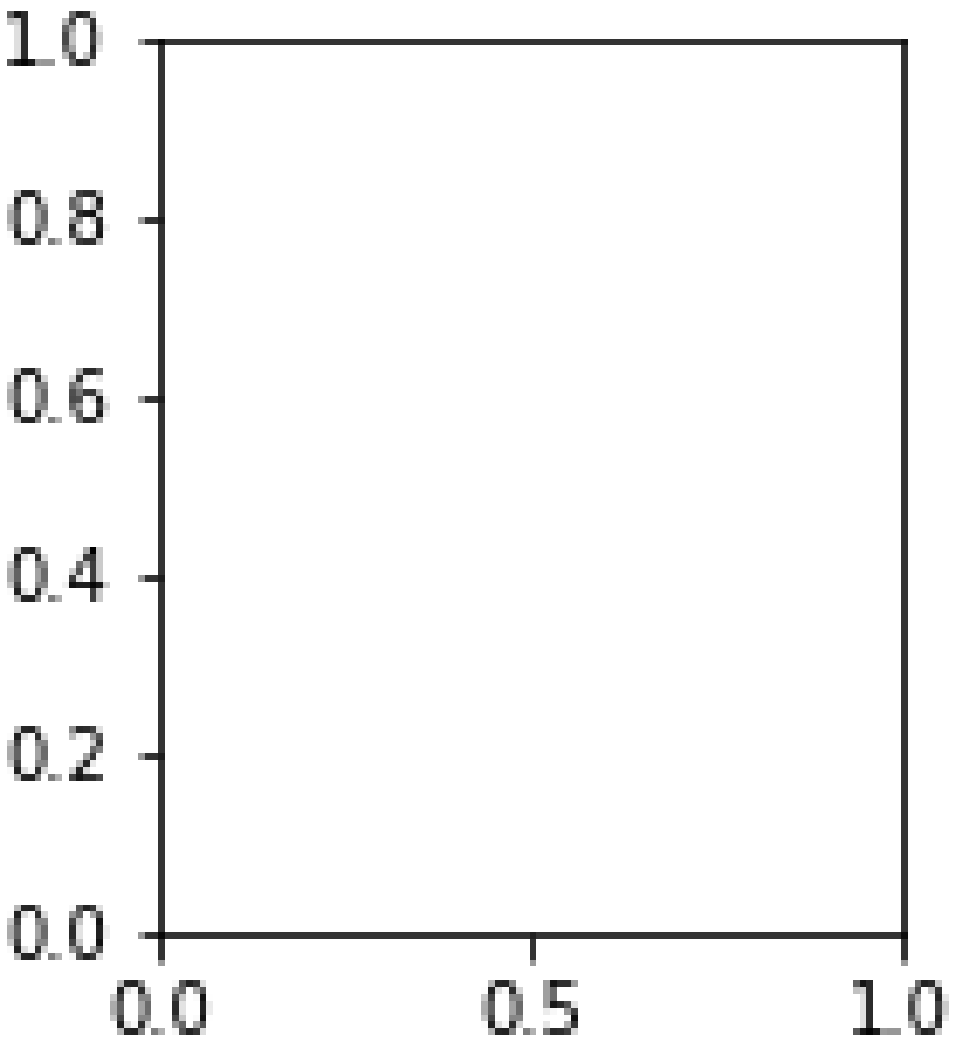
Das folgende Beispiel zeigt den einfachsten Fall, d.h. ein 1x1-Gitter:

```
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

fig = plt.figure(figsize=(2, 3))
gs = GridSpec(1, 1)
ax = fig.add_subplot(gs[0,0])

plt.show()
```



Wir könnten noch einige der Parameter aus `GridSpec` benutzen, z.B. können wir definieren, dass der Graph erst 20% von unten und 15% von links der verfügbaren Abbildungsfläche beginnen soll:

```
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

fig = plt.figure(figsize=(2, 3))
gs = GridSpec(1, 1,
              bottom=0.2,
              left=0.15,
              top=0.8)
ax = fig.add_subplot(gs[0,0])

plt.show()
```



Das nächste Beispiel zeigt ein komplexeres Beispiel mit einem aufwendigeren Gitter-Design:

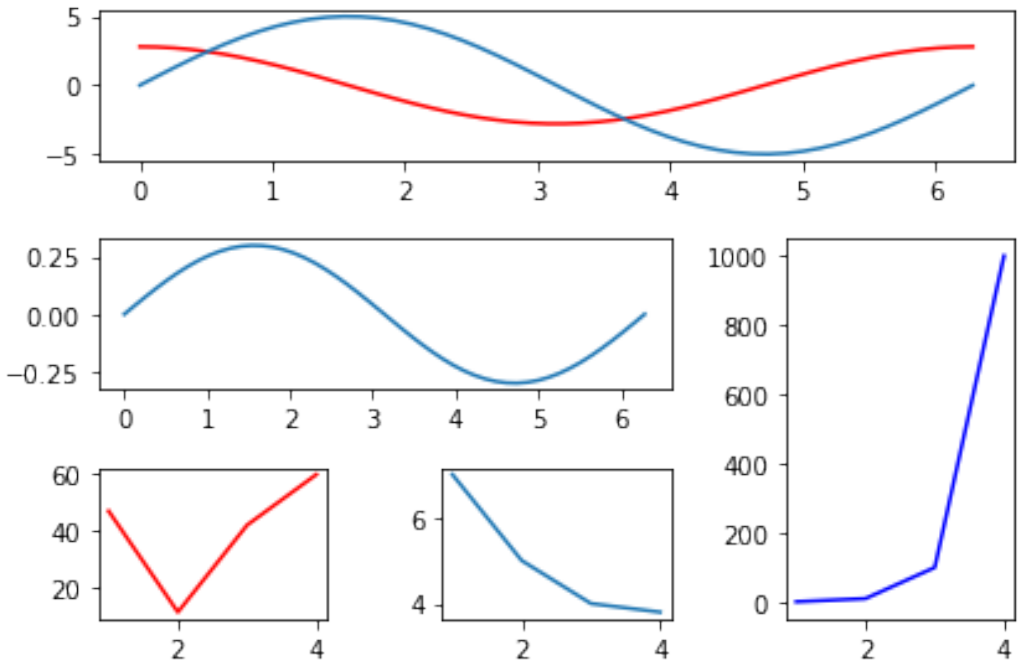
```
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as pl

pl.figure(figsize=(5,3))
```

```
G = gridspec.GridSpec(3, 3)

axes_1 = pl.subplot(G[0, :])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'Axes 1', ha='center',
        va='center', size=24, alpha=.5)

axes_2 = pl.subplot(G[1, :-1])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'Axes 2', ha='center',
        va='center', size=24, alpha=.5)

axes_3 = pl.subplot(G[1:, -1])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'Axes 3', ha='center',
        va='center', size=24, alpha=.5)

axes_4 = pl.subplot(G[-1, 0])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'Axes 4', ha='center',
        va='center', size=24, alpha=.5)

axes_5 = pl.subplot(G[-1, -2])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'Axes 5', ha='center',
        va='center', size=24, alpha=.5)

pl.tight_layout()
pl.show()
```

Die Gitter-Spezifikation aus dem vorigen Beispiel verwenden wir nun, um sie mit Funkti-

onsgraphen zu bestücken:

```
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(6, 4))
G = gridspec.GridSpec(3, 3)

X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F1 = 2.8 * np.cos(X)
F2 = 5 * np.sin(X)
F3 = 0.3 * np.sin(X)

axes_1 = plt.subplot(G[0, :])
axes_1.plot(X, F1, 'r-', X, F2)

axes_2 = plt.subplot(G[1, :-1])
axes_2.plot(X, F3)

axes_3 = plt.subplot(G[1:, -1])
axes_3.plot([1,2,3,4], [1,10,100,1000], 'b-')

axes_4 = plt.subplot(G[-1, 0])
axes_4.plot([1,2,3,4], [47, 11, 42, 60], 'r-')

axes_5 = plt.subplot(G[-1, -2])
axes_5.plot([1,2,3,4], [7, 5, 4, 3.8])

plt.tight_layout()
plt.show()
```

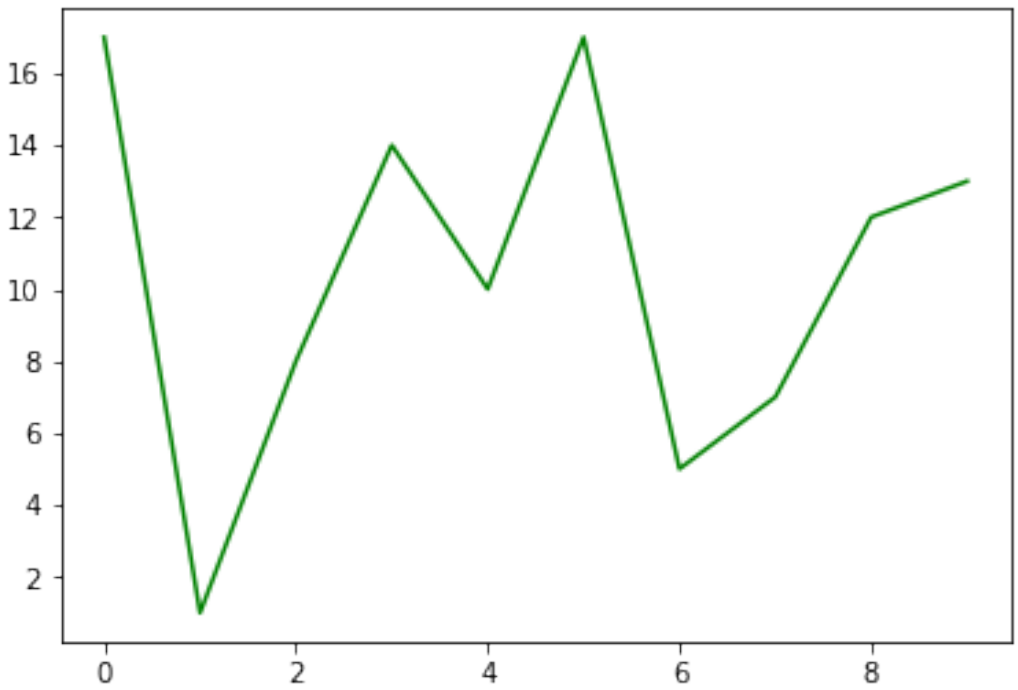


Arbeiten mit Objekten

Matplotlib ist komplett objektorientiert designt und programmiert – wie Python selbst. Die bis jetzt gezeigten Beispiel waren sehr einfach. Um die Beispiele so einfach wie möglich zu halten, arbeiteten wir z.B. nicht immer mit figure-Objekten. Trotzdem werden die Objekte automatisch angelegt. Der Vorteil der Verwendung von Objekten kommt dann zum Vorschein, wenn mehr als eine Abbildung verwendet wird, oder wenn eine Abbildung mehrere Unterdiagramme (subplots) enthält.

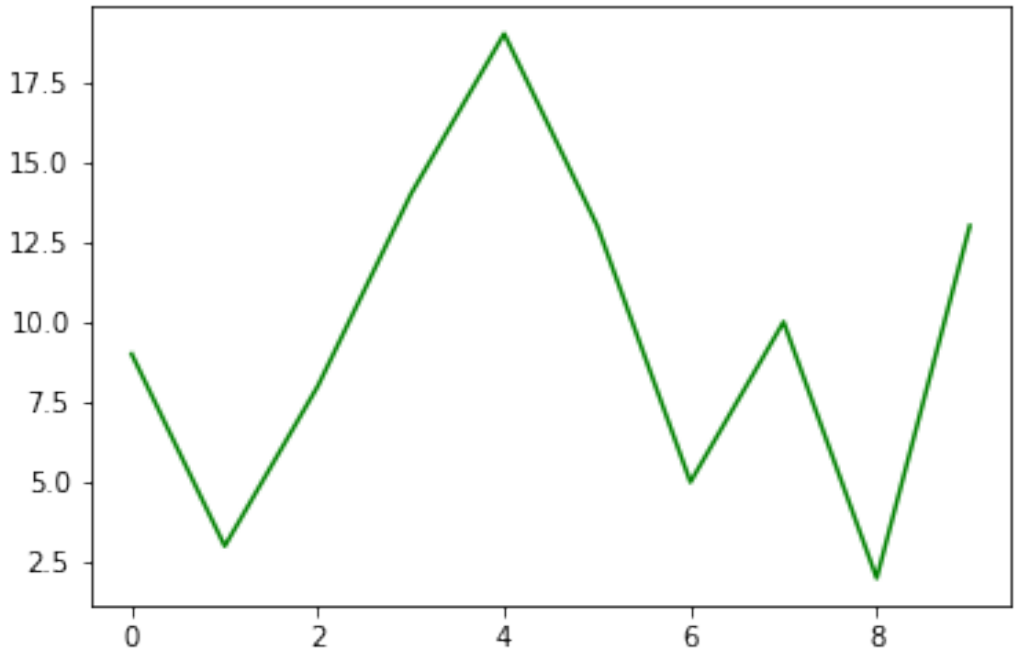
Im folgenden Beispiel erstellen wir einen Plot auf eine streng objektorientierte Art und Weise. Wir beginnen mit der Erstellung eines neuen figure-Objektes. Wir speichern dazu eine Referenz in der Variable `fig`. Diese benutzen wir, um mit `add_axes` aus der figure-Klasse neue axis-Instanzen zu erstellen:

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()

X = np.arange(0,10)
Y = np.random.randint(1,20, size=10)

left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
axes = fig.add_axes([left, bottom, width, height])

axes.plot(X, Y, 'g')

plt.show(fig)
```



Ohne explizite Instanzen zu verwenden, sieht der Code folgendermaßen aus:

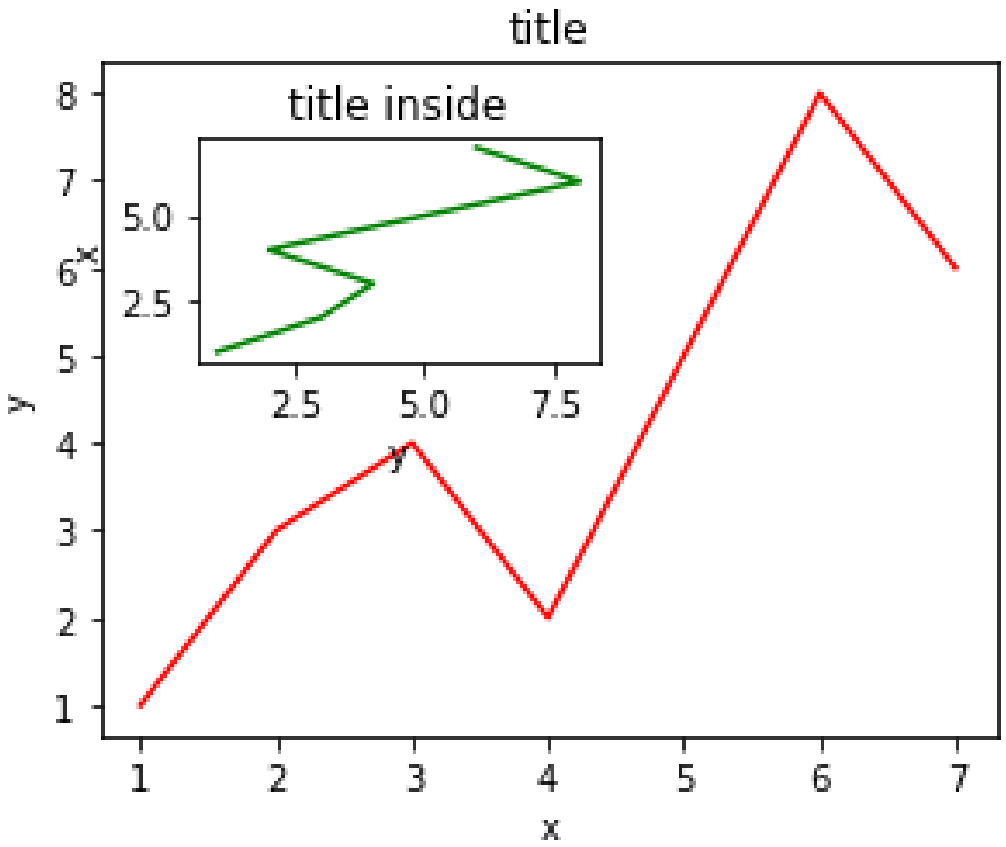
```
import numpy as np
import matplotlib.pyplot as plt

X = np.arange(0,10)
Y = np.random.randint(1,20, size=10)

plt.plot(X, Y, 'g')

plt.show()
```



Ein Plot innerhalb eines anderen Plots

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(4, 3))

X = [1, 2, 3, 4, 5, 6, 7]
Y = [1, 3, 4, 2, 5, 8, 6]

axes1 = fig.add_axes([0.1, 0.1, 0.9, 0.9]) # main axes
axes2 = fig.add_axes([0.2, 0.6, 0.4, 0.3]) # inset axes

# main figure
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('title')

axes1.plot(X, Y, 'r')
```

```
# insert
axes2.set_xlabel('y')
axes2.set_ylabel('x')
axes2.set_title('title inside')
axes2.plot(Y, X, 'g')

plt.show()
```

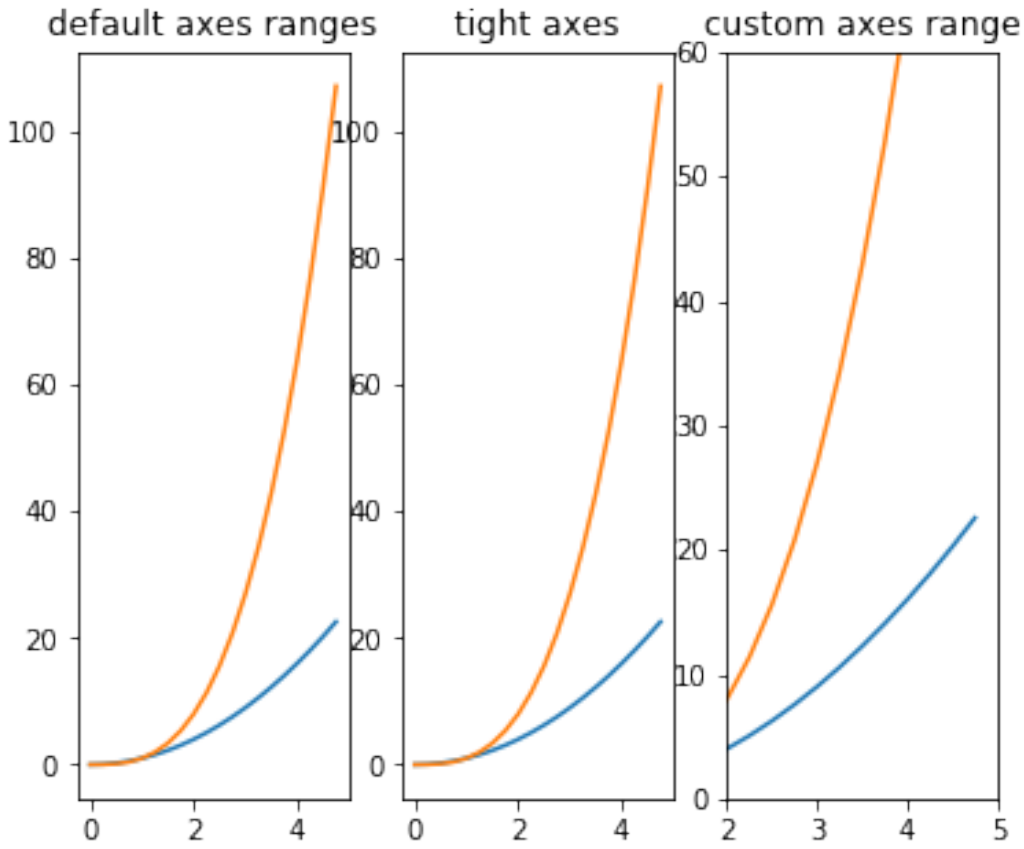


Setzen des Plotbereichs

Es gibt die Möglichkeit, den Bereich der Achsen zu konfigurieren. Dafür verwendet man die Methoden `set_ylim` und `set_xlim` des Achsen-Objektes. Mit `axis('tight')` erstellen wir automatisch “eng anliegende” Plot-Bereiche:

```
import numpy as np
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 3, figsize=(6, 5))

x = np.arange(0, 5, 0.25)

axes[0].plot(x, x**2, x, x**3)
axes[0].set_title("default axes ranges")

axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
```

```
axes[1].set_title("tight axes")

axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range");
```

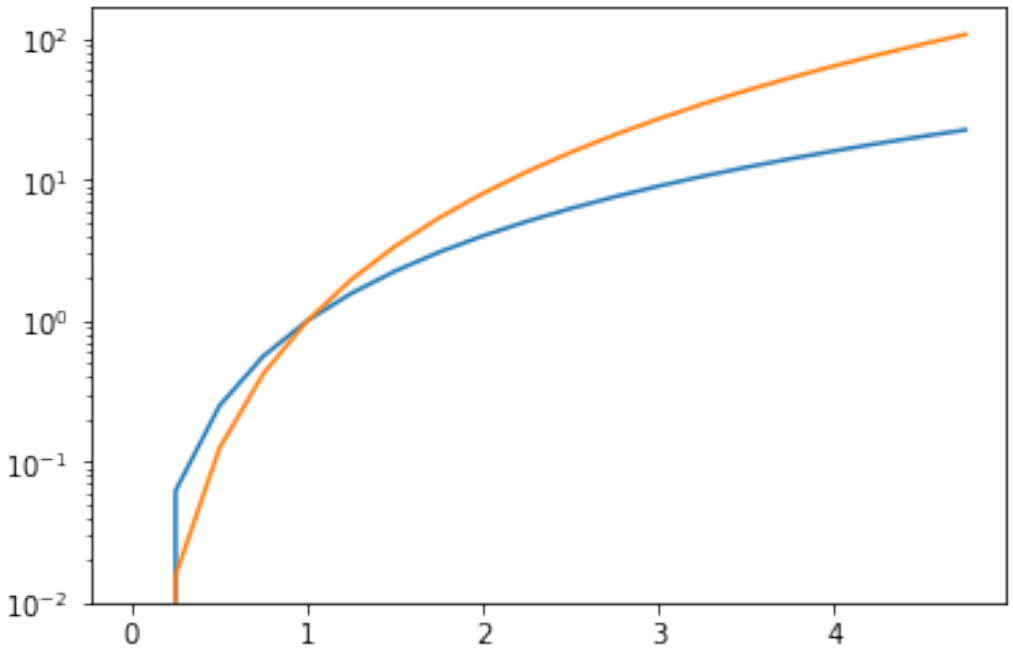


Logarithmische Darstellung

Es gibt auch die Möglichkeit, eine logarithmische Darstellung für eine oder beide Achsen einzustellen. Die Funktionalität ist tatsächlich nur eine einzige Applikation eines allgemeineren Transformations-Systems in Matpltolib. Jede der Achsen-Darstellung wird durch eine separate `set_xscale`- und `set_yscale`-Methode gesetzt. Diese nehmen einen Parameter entgegen (in diesem Fall mit dem Wert “log”):

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

x = np.arange(0, 5, 0.25)

ax.plot(x, x**2, x, x**3)
```

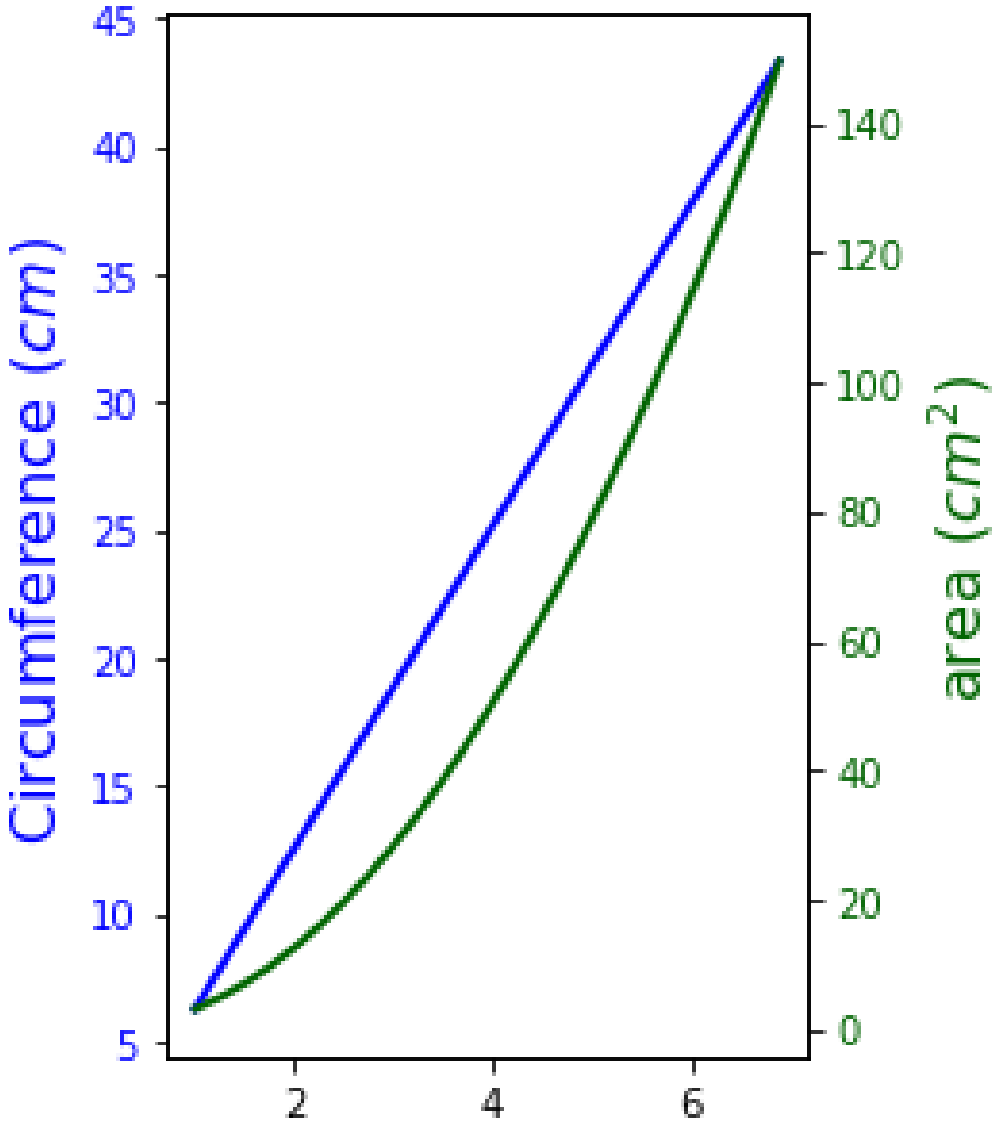
```
ax.set_yscale("log")

plt.show()
```



Sekundäre Y-Achse

Bisher haben wir schon öfters zwei verschiedene Graphen in einem Plot dargestellt. Allerdings war bisher immer der Wertebereich, also die Einheit auf der y-Achse, die gleiche gewesen. Im Folgenden haben wir zwei Wertebereiche, einmal "cm" und einmal "qcm". Wir erzeugen die zweite y-Achse, die man üblicherweise als Sekundärachse bezeichnet, mit der Methode `twinx`:

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots(figsize=(3, 5))

x = np.arange(1,7,0.1)
ax1.plot(x, 2 * np.pi * x, lw=2, color="blue")
ax1.set_ylabel(r"Circumference $(cm)$", fontsize=16, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, np.pi * x ** 2, lw=2, color="darkgreen")
ax2.set_ylabel(r"area $(cm^2)$", fontsize=16, color="darkgreen")
for label in ax2.get_yticklabels():
    label.set_color("darkgreen")
```

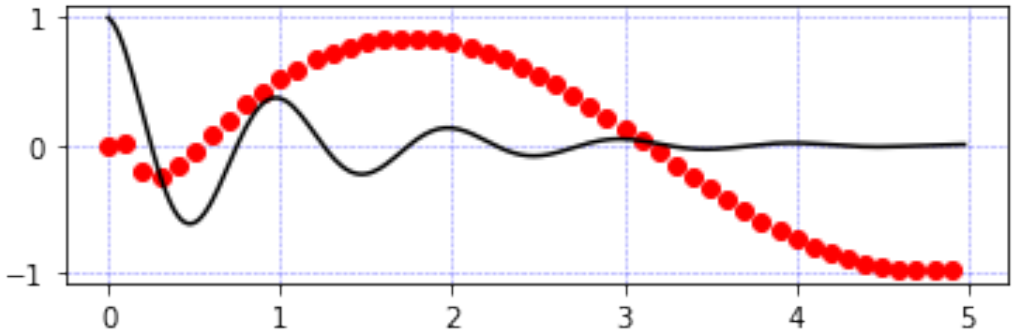


Die folgenden Themen stehen nicht im direkten Zusammenhang mit Unterplots. Trotzdem wollen wir sie hier erwähnen, um die Einführung der Basis-Möglichkeiten von Matplotlib abzurunden. Das erste zeigt, wie Gitter-Linien definiert werden. Das zweite, sehr wichtige Thema, befasst sich mit der Speicherung von Plots in Bild-Dateien.

Gitter-Linien

```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)
def g(t):
    return np.sin(t) * np.cos(1/(t+0.1))

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
```

```
plt.subplot(212)
plt.plot(t1, g(t1), 'ro', t2, f(t2), 'k')
plt.grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
plt.show()
```



Abbildungen speichern

Die savefig-Methode kann verwendet werden, um Abbildungen als Dateien zu speichern:

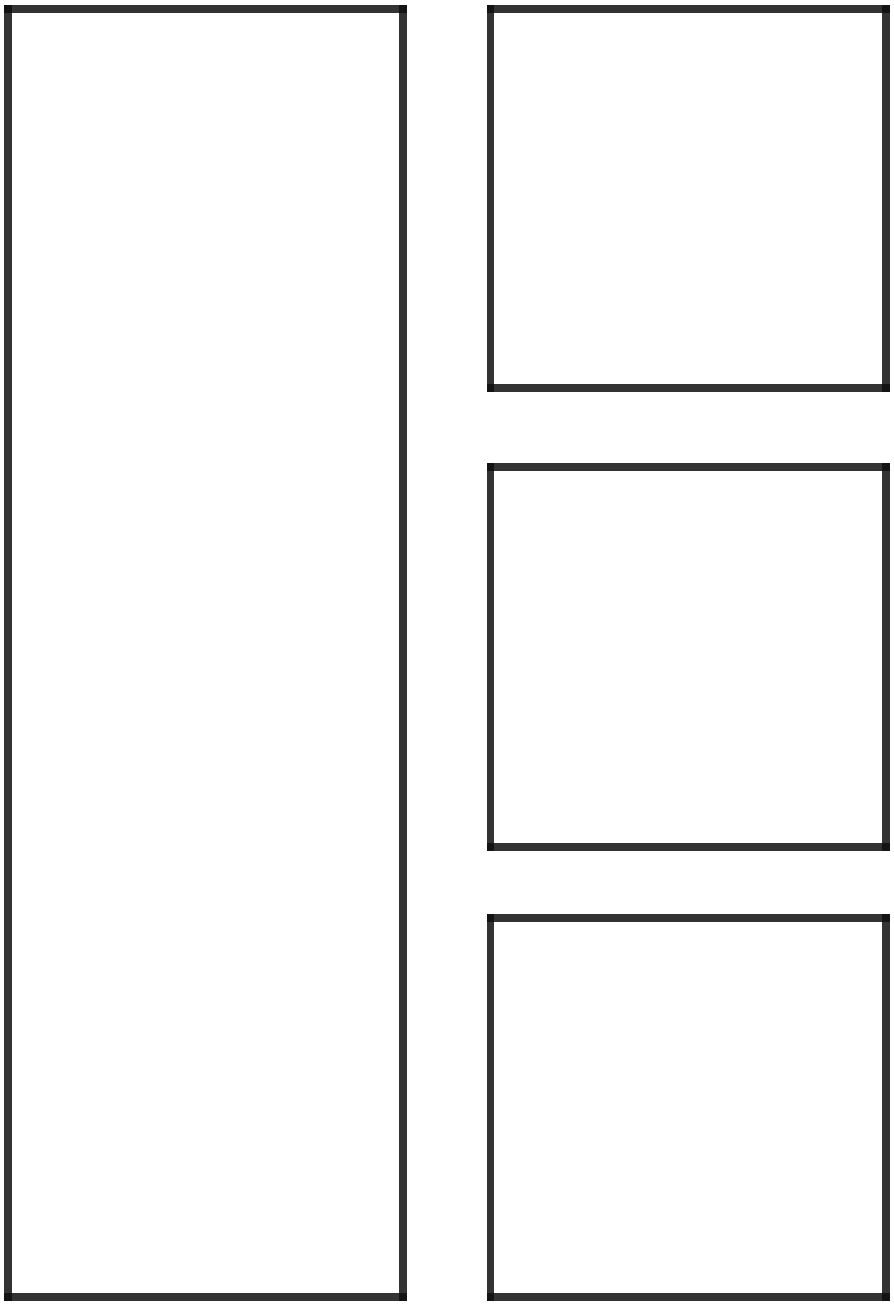
`fig.savefig('filename.png')`

Optional kann die DPI und das Ausgabeformat festgelegt werden:

`fig.savefig('filename.png', dpi=200)`

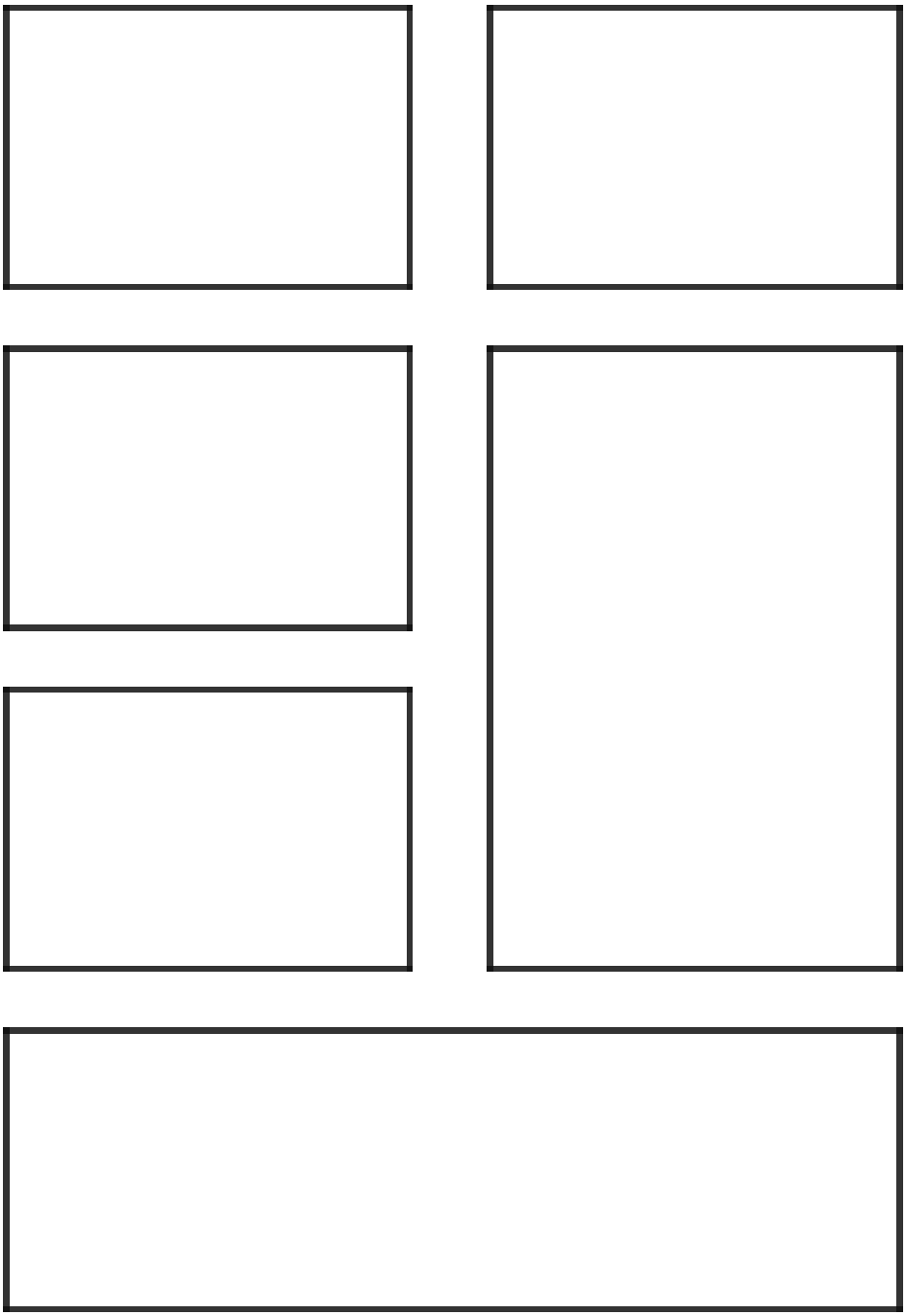
Die gängigen Bildformate sind als Ausgabe von savefig möglich: PNG, JPG, EPS, SVG, PGF und PDF.

```
# prog4book

import  matplotlib.pyplot as plt
fig =plt.figure(figsize=(2, 3))
X = [ (1,2,1), (3,2,2), (3,2,4), (3,2,6) ]
for nrows, ncols, plot_number in X:
    plt.subplot(nrows, ncols, plot_number)
    plt.xticks([])
    plt.yticks([])
```



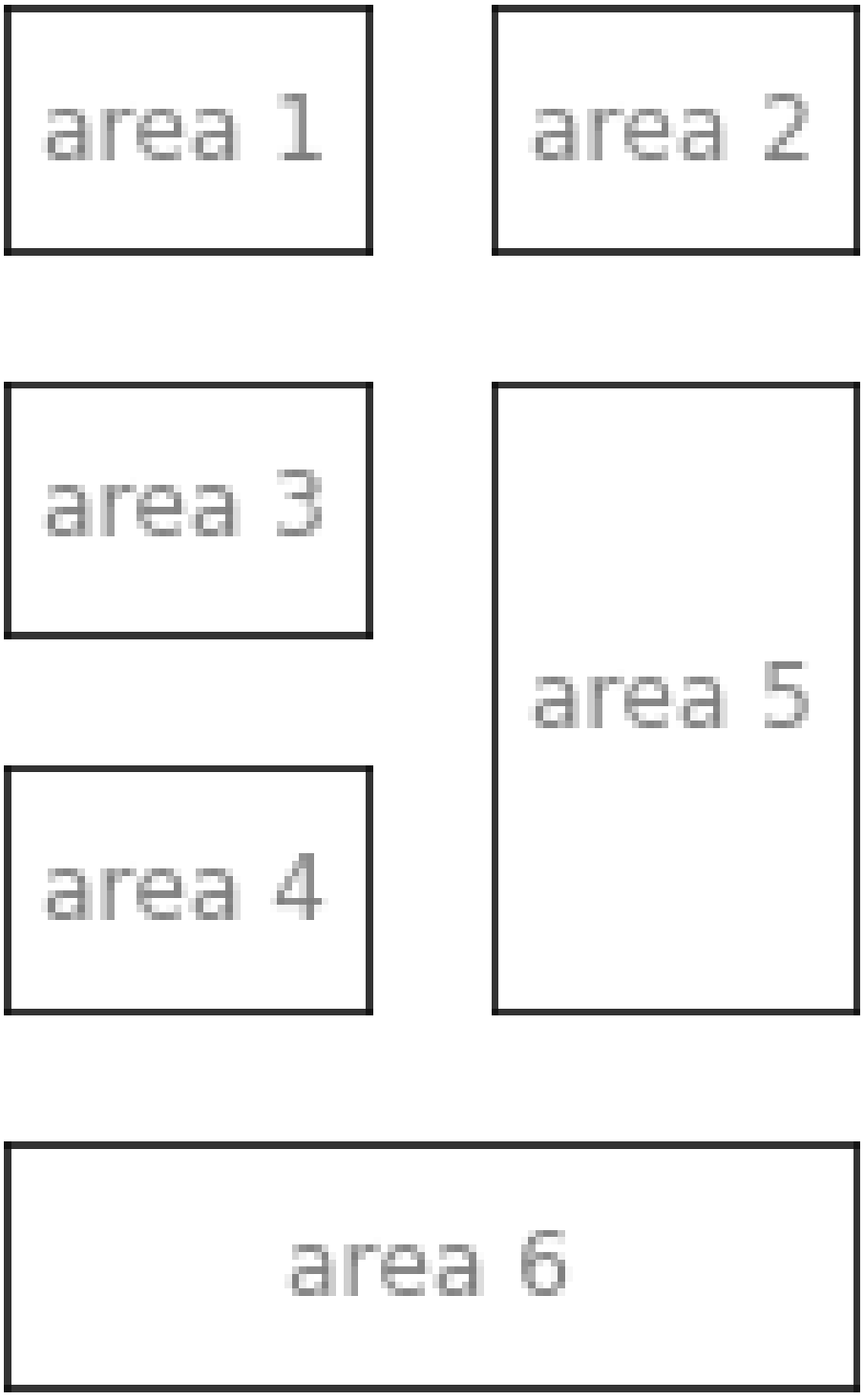
```
# prog4book
import  matplotlib.pyplot as plt
fig =plt.figure(figsize=(2, 3))
X = [  (4,2,1),(4,2,2), (4,2,3), (4,2,5), (4,2,(4,6)), (4,1,4)]
plt.subplots_adjust(bottom=0, left=0, top = 0.975, right=1)
for nrows, ncols, plot_number in X:
    plt.subplot(nrows, ncols, plot_number)
    plt.xticks([])
    plt.yticks([])
```

```
plt.show()
```



```
# prog4book

import matplotlib.gridspec as gridspec
import matplotlib.pyplot as pl

pl.figure(figsize=(2, 3))
G = gridspec.GridSpec(4, 2)
```

```
fontsize = 12
alpha = 0.5
axes_1 = pl.subplot(G[0, 0])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'area 1', ha='center',
        va='center', size=fontsize, alpha=alpha)

axes_2 = pl.subplot(G[0, 1])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'area 2', ha='center',
        va='center', size=fontsize, alpha=alpha)

axes_3 = pl.subplot(G[1, 0])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'area 3', ha='center',
        va='center', size=fontsize, alpha=alpha)

axes_4 = pl.subplot(G[2, 0])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'area 4', ha='center',
        va='center', size=fontsize, alpha=alpha)

axes_5 = pl.subplot(G[1:3, 1])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'area 5', ha='center',
        va='center', size=fontsize, alpha=alpha)

axes_6 = pl.subplot(G[3, :])
pl.xticks(())
pl.yticks(())
pl.text(0.5, 0.5, 'area 6', ha='center',
        va='center', size=fontsize, alpha=alpha)

pl.tight_layout()
pl.show()
```


area 1
area 2
area 3
area 5
area 4
area 6

17 Matplotlib Histogramme

17.0.1 Matplotlib-Tutorial: Histogramme

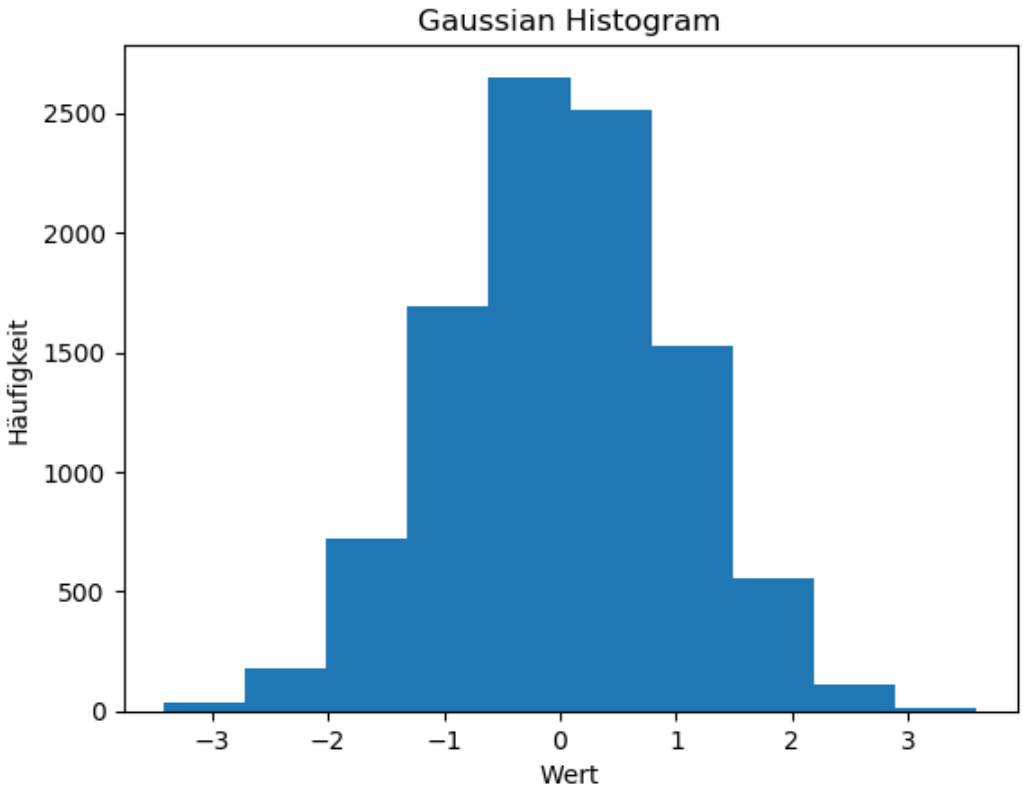
Dieses Kapitel des Tutorials befasst sich mit Balken-, Säulendiagramme und Histogrammen. Es ist schwierig eine Zeitung oder ein Magazin zu finden, in der keine Histogramme zu finden sind die etwas über die Anzahl der Raucher einer bestimmten Altergruppe oder die Anzahl der Geburten pro Jahr aussagen. Es ist eine gute Möglichkeit Fakten ohne viele Worte wiederzugeben. Auf der anderen Seite jedoch können sie auch zur Manipulation von Statistiken werden.

Was ist ein Histogramm? Eine formale Definition könnte sein: Es ist eine grafische Repräsentation einer Verteilung von numerischen Daten. Rechtecke mit der gleichen Breite haben Höhen, die den zugehörigen Frquenzen entsprechen.

Wenn wir ein Histogramm konstruieren, beginnen wir mit der Verteilung des Bereichs der möglichen x-Werte in gewöhnlich gleich große und benachbarte Klassen bzw. Intervalle, die man in Englisch Bins nennt. Die Daten werden nun entsprechend ihrer Größe in diese Bins verteilt. Für jede Klasse bzw. Intervall werden dann Rechtecke gezeichnet, deren Höhe bzw. Flächeninhalt der relativen oder absoluten Häufigkeit der Klasse entspricht.

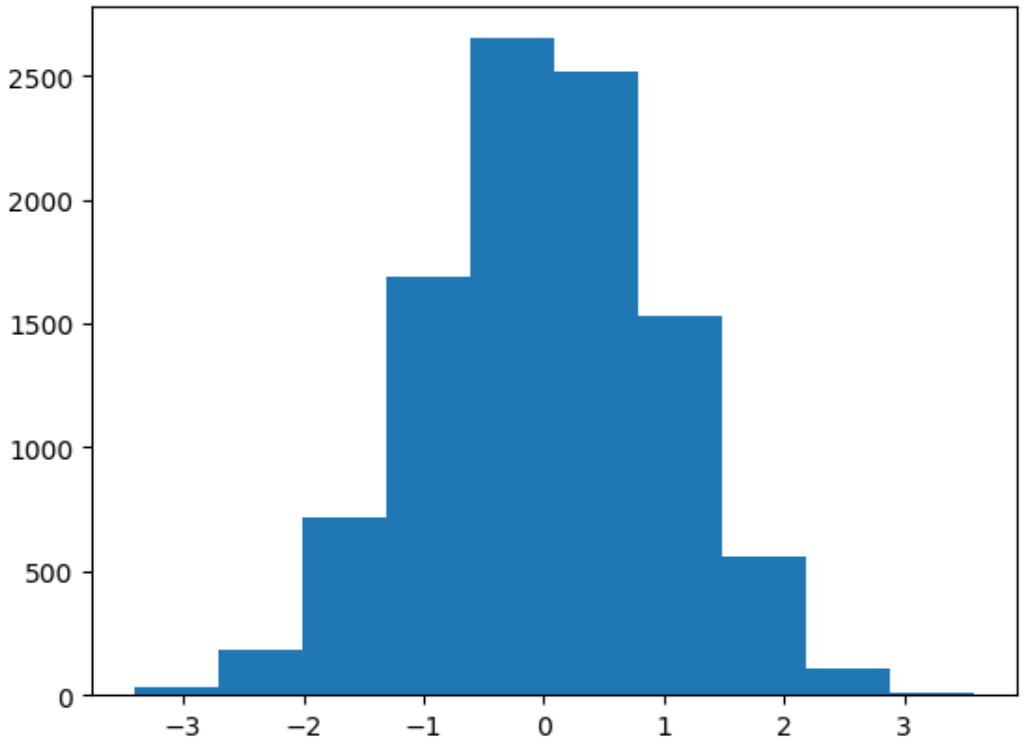
Wir schreiben nun ein Python-Programm, indem wir Zufallszahlen erzeugen und aus diesen ein Histogramm erzeugen:

```
import matplotlib.pyplot as plt
import numpy as np

gaussian_numbers = np.random.normal(size=10000)
plt.hist(gaussian_numbers)
plt.title("Gaussian Histogram")
plt.xlabel("Wert")
plt.ylabel("Häufigkeit")
plt.show()
```



Wir haben gesehen, dass die Funktion `hist` (eigentlich `matplotlib.pyplot.hist`) die Histogrammwerte berechnet und den Graphen zeichnet. Außerdem gibt die Funktion auch ein Tupel mit drei Objekten `(n, bins, patches)` zurück:

```
n, bins, patches = plt.hist(gaussian_numbers)
```



Schauen wir uns die Rückgabewerte genauer an. Bei der Histogrammbildung werden die Zufallswerte von unserem Array `gaussian_numbers` in gleichgroße Intervalle aufgeteilt, d.h. die "bins". Die von `hist` berechneten Intervallgrenzen erhalten wir in der zweiten Komponente des Rückgabetupels. In unserem Beispiel werden sie mit der Variable `bins` bezeichnet:

```
print("Die ersten drei 'bins': ", bins[0:3])
```

```
Die ersten drei 'bins':  [-3.41215556 -2.71239973 -2.0126439 ]
```

`n[i]` enthält die Anzahl der Werte von `gaussian_numbers`, die innerhalb des Intervalls mit den Grenzen `bins[i]` und `bins[i+1]` liegen:

```
print("n: ", n, sum(n))
```

```
n:  [  32.  182.  719. 1692. 2651. 2515. 1528.  558.  110.   13.] 10000.0
```

`n` ist also ein Array mit den Häufigkeiten. Der letzte Rückgabewert von `hist` ist eine Liste `patches`, was den Rechtecken mit ihren Eigenschaften entspricht:

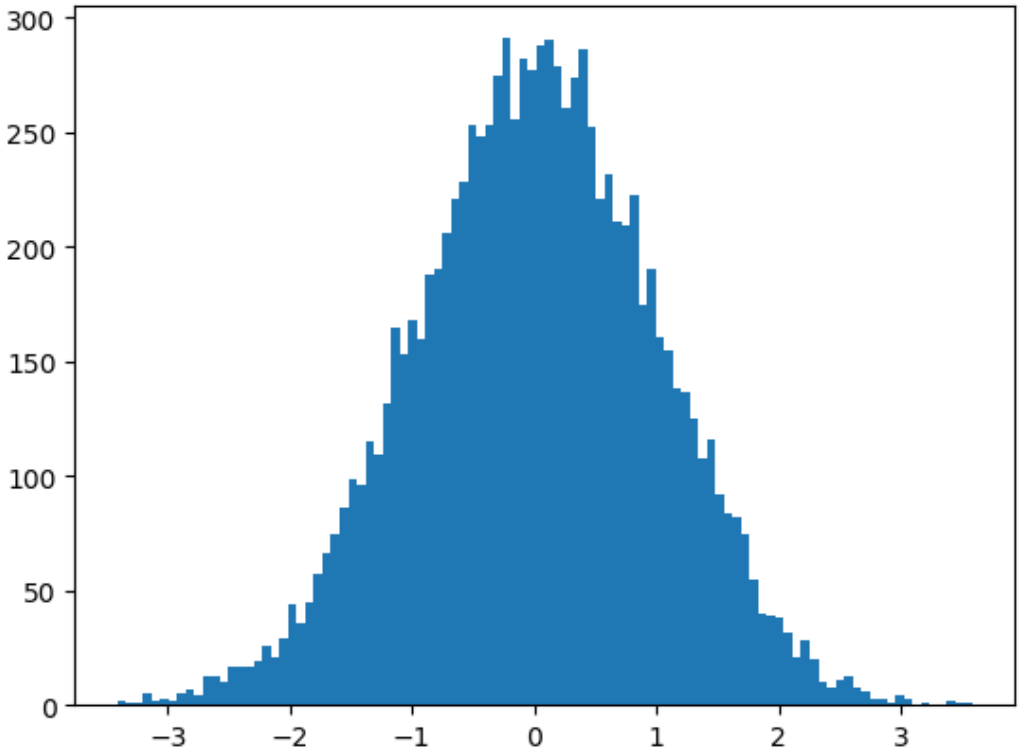
```
print("patches: ", patches)
for i in range(10):
    print(patches[i])
```

```
patches:  <a list of 10 Patch objects>
Rectangle(xy=(-3.41216, 0), width=0.699756, height=32, angle=0)
Rectangle(xy=(-2.7124, 0), width=0.699756, height=182, angle=0)
Rectangle(xy=(-2.01264, 0), width=0.699756, height=719, angle=0)
Rectangle(xy=(-1.31289, 0), width=0.699756, height=1692, angle=0)
```

```
Rectangle(xy=(-0.613132, 0), width=0.699756, height=2651, angle=0)
Rectangle(xy=(0.0866236, 0), width=0.699756, height=2515, angle=0)
Rectangle(xy=(0.786379, 0), width=0.699756, height=1528, angle=0)
Rectangle(xy=(1.48614, 0), width=0.699756, height=558, angle=0)
Rectangle(xy=(2.18589, 0), width=0.699756, height=110, angle=0)
Rectangle(xy=(2.88565, 0), width=0.699756, height=13, angle=0)
```

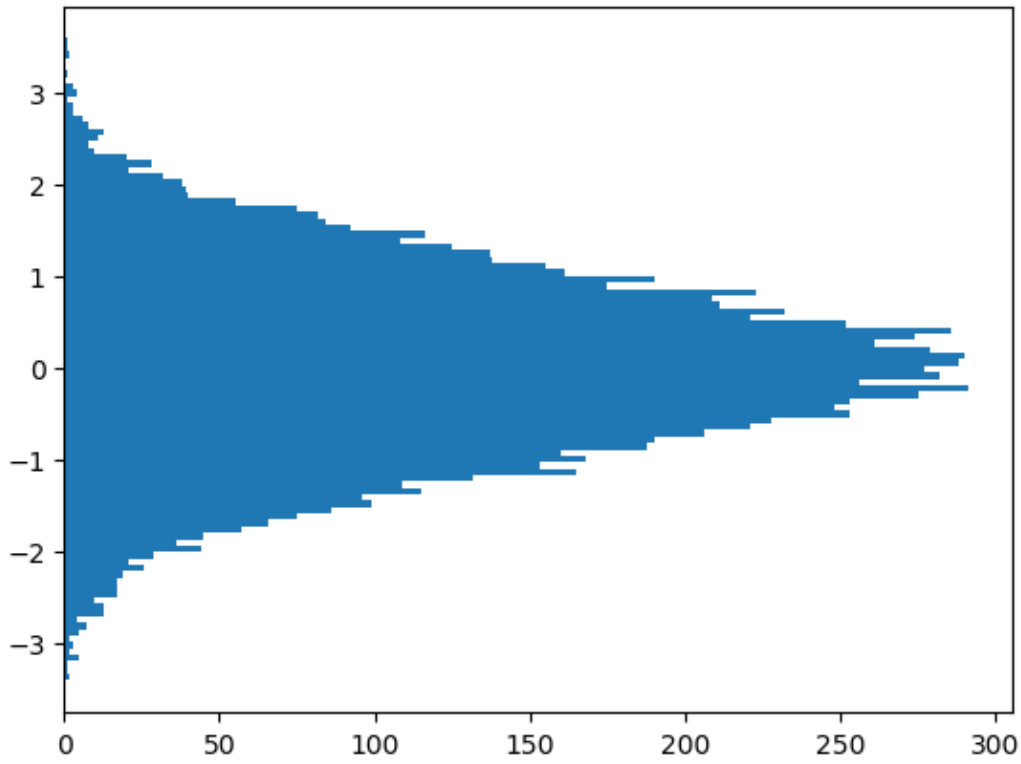
10 Bins sind nicht viel, wenn wir von 10.000 Zufallszahlen sprechen. Deshalb erhöhen wir im folgenden Programm die Anzahl der Klassen. Dazu setzen wir den Schlüsselwort-Parameter `bins` auf 100:

```
plt.hist(gaussian_numbers, bins=100)
plt.show()
```



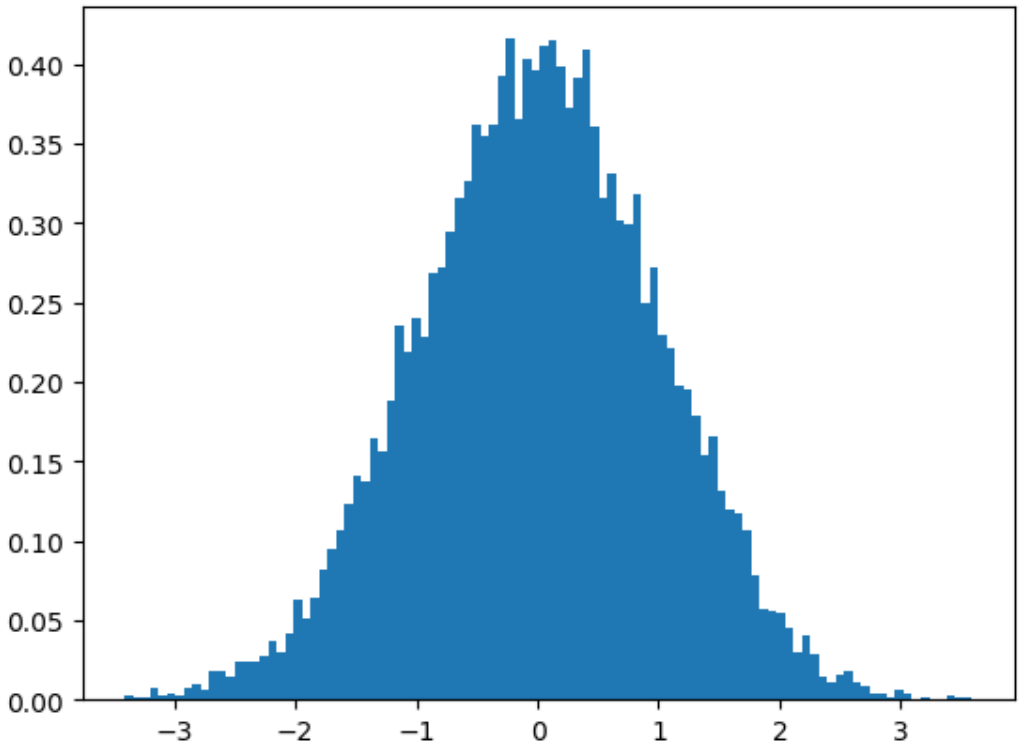
Indem wir den Parameter `orientation` auf `vertical` setzen, können wir das Histogramm auch seitwärts ausgeben:

```
plt.hist(gaussian_numbers, bins=100, orientation="horizontal")
plt.show()
```



Ein weiterer wichtiger Schlüsselwort-Parameter von `hist` ist `density`, der den veralteten Parameter `normed` ablöst. Falls er auf `True` gesetzt wird, wird die erste Komponente – also die Häufigkeiten – des Rückgabetupels normalisiert, um eine Wahrscheinlichkeitsdichte zu bilden, d.h. die Fläche (oder das Integral) unter dem Histogramm bildet die Summe 1

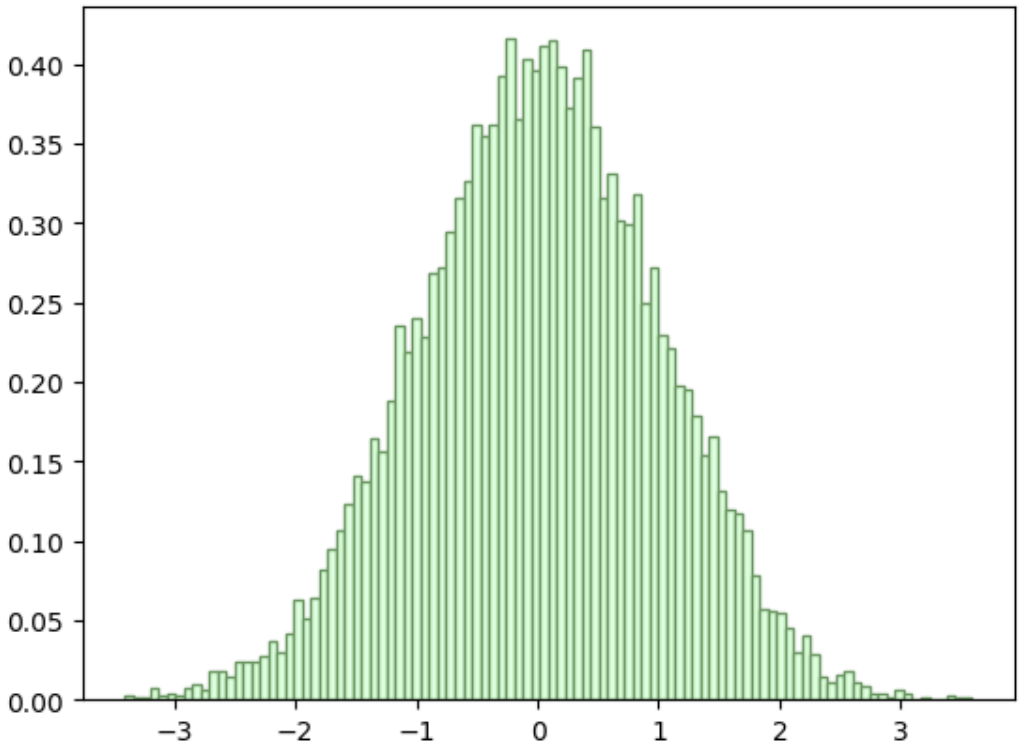
```
n, bins, patches = plt.hist(gaussian_numbers,
                            bins=100,
                            density=True)
plt.show()
print("Fläche unter dem Integral: ", np.sum(n * np.diff(bins)))
```




```
Fläche unter dem Integral:  1.0
```

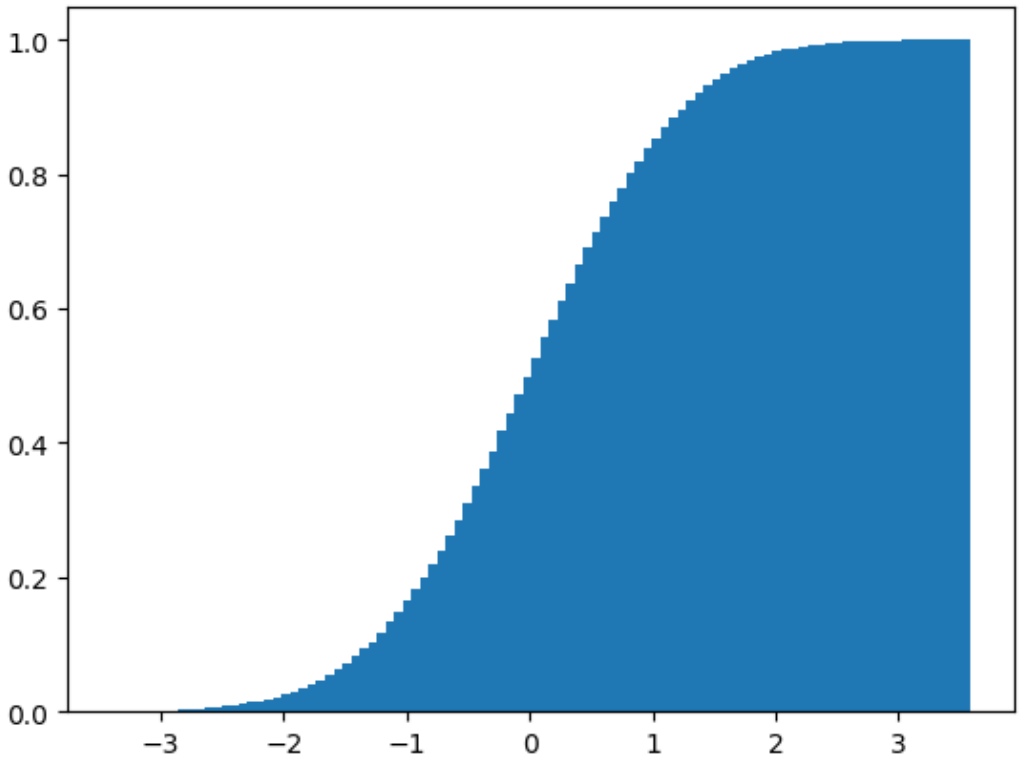
Mit den Parametern `edgecolor` und `color` können wir die Linienfarbe und die Farbe der Flächen festlegen:

```
n, bins, patches = plt.hist(gaussian_numbers,
                            bins=100,
                            density=True,
                            edgecolor="#6A9662",
                            color="#DDFFDD")
plt.show()
```



Okay, Sie möchten eine Darstellung von kumulierten Werten sehen? Wir können eine kumulative Verteilungs-Funktion ausgeben, indem wir den Paremter `cumulative` setzen.

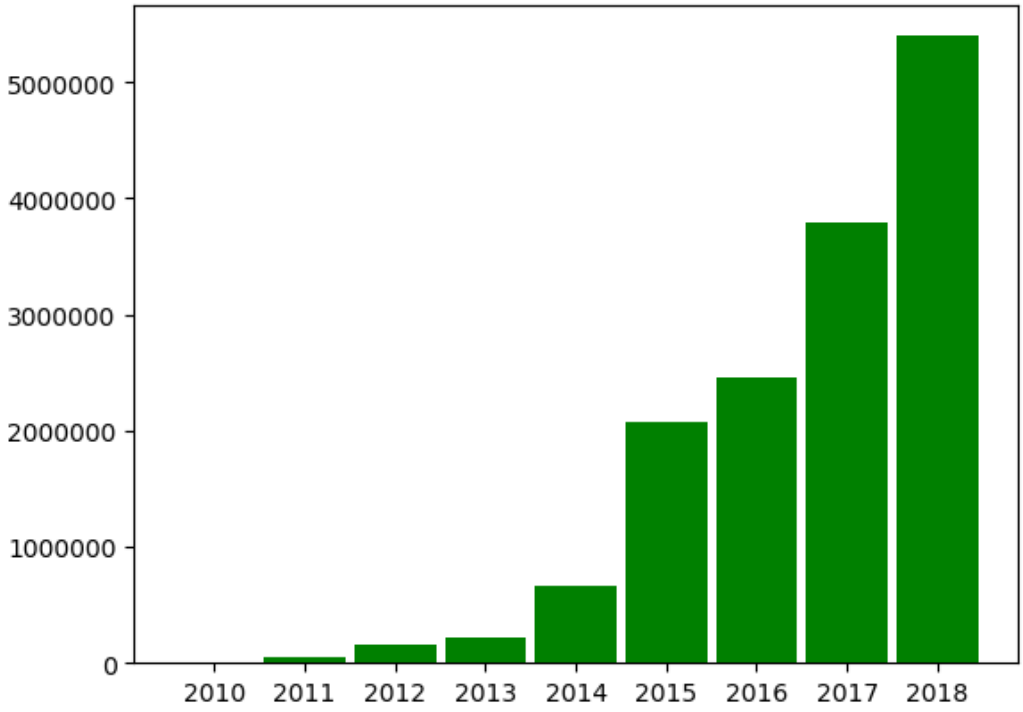
```
plt.hist(gaussian_numbers,
         bins=100,
         density=True,
         stacked=True,
         cumulative=True)

plt.show()
```



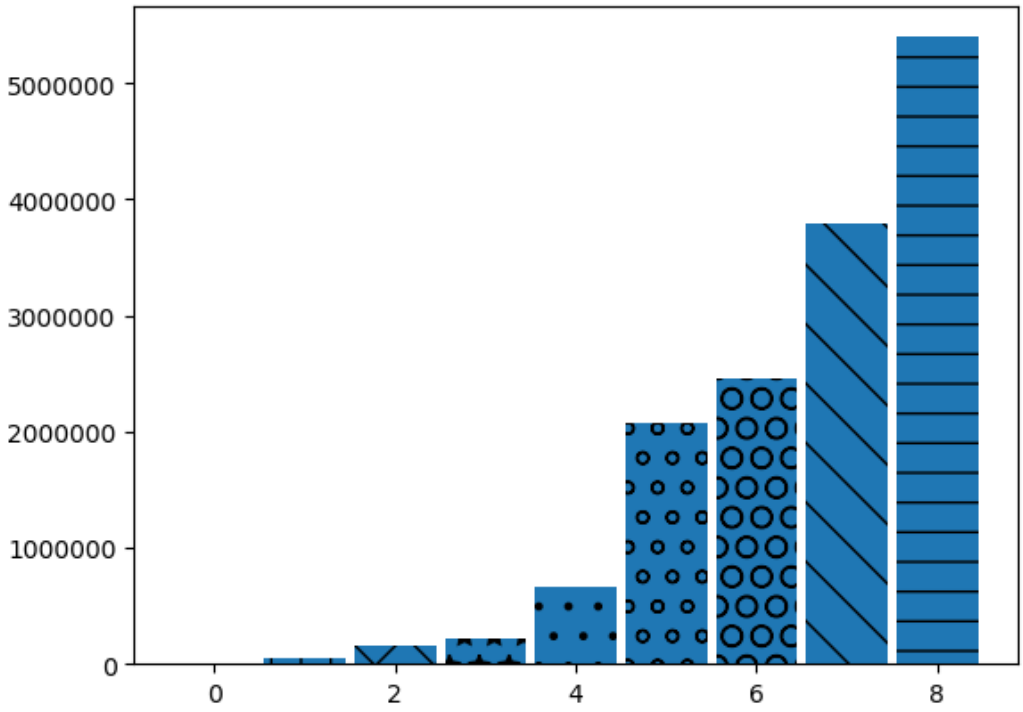
Säulendiagramm

Nun kommen wir zu einem der am häufigsten benutzten Diagrammtypen, der auch unter Nichtwissenschaftlern bestens bekannt ist. Das Säulendiagramm ist so benannt, weil es sich aus auf der x-Achse senkrecht stehenden Rechtecken zusammensetzt, die sich wie Säulen aufrichten. Ist die Breite der Säulen sehr schmal, werden sie auch als Stabdiagramm bezeichnet. Die Breite der Rechtecke hat keine mathematische Bedeutung.

```
import matplotlib.pyplot as plt
import numpy as np

#years = ('2010', '2011', '2012', '2013', '2014', '2015')
years = [str(year) for year in range(2010, 2019)]
visitors = (1241, 50927, 162242, 222093, 665004,
            2071987, 2460407, 3799215, 5399000)

index = np.arange(len(years))
bar_width = 0.9
plt.bar(index, visitors, bar_width,  color="green")
plt.xticks(index, years) # labels get centered
plt.show()
```



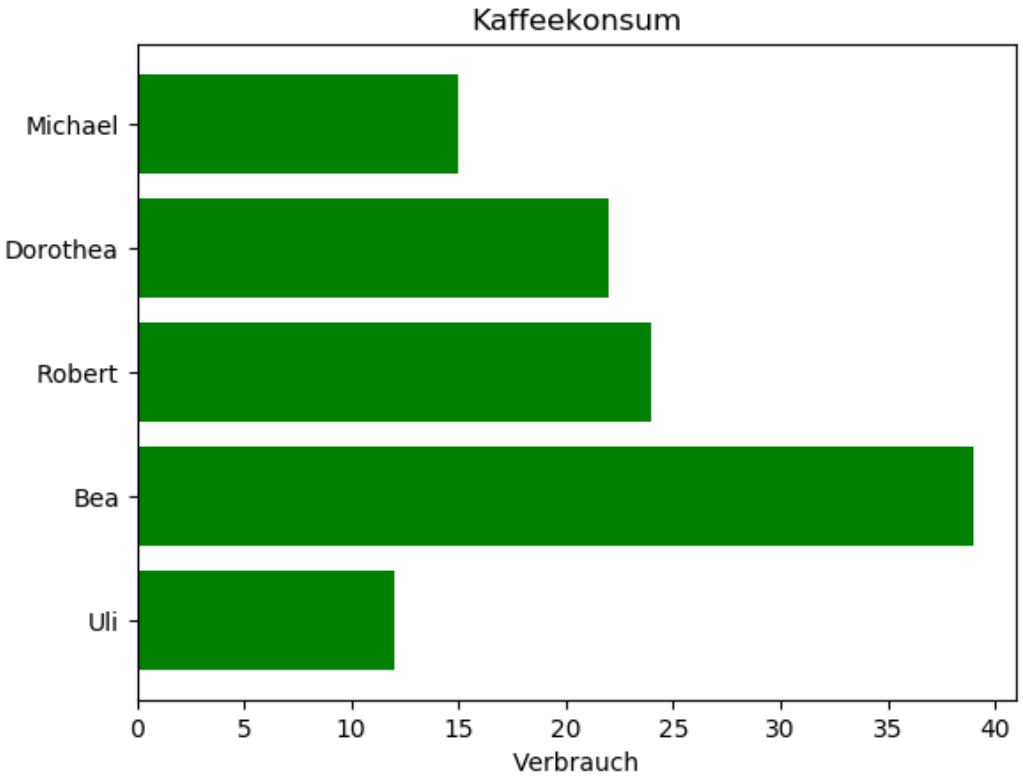

```
# prog4book
bars = plt.bar(index, visitors, bar_width)

patterns = ('-', '+', 'x', '*', '.', 'o', 'O', '\\', '-')
for bar, pattern in zip(bars, patterns):
    bar.set_hatch(pattern)

plt.show()
```



Balkendiagramme

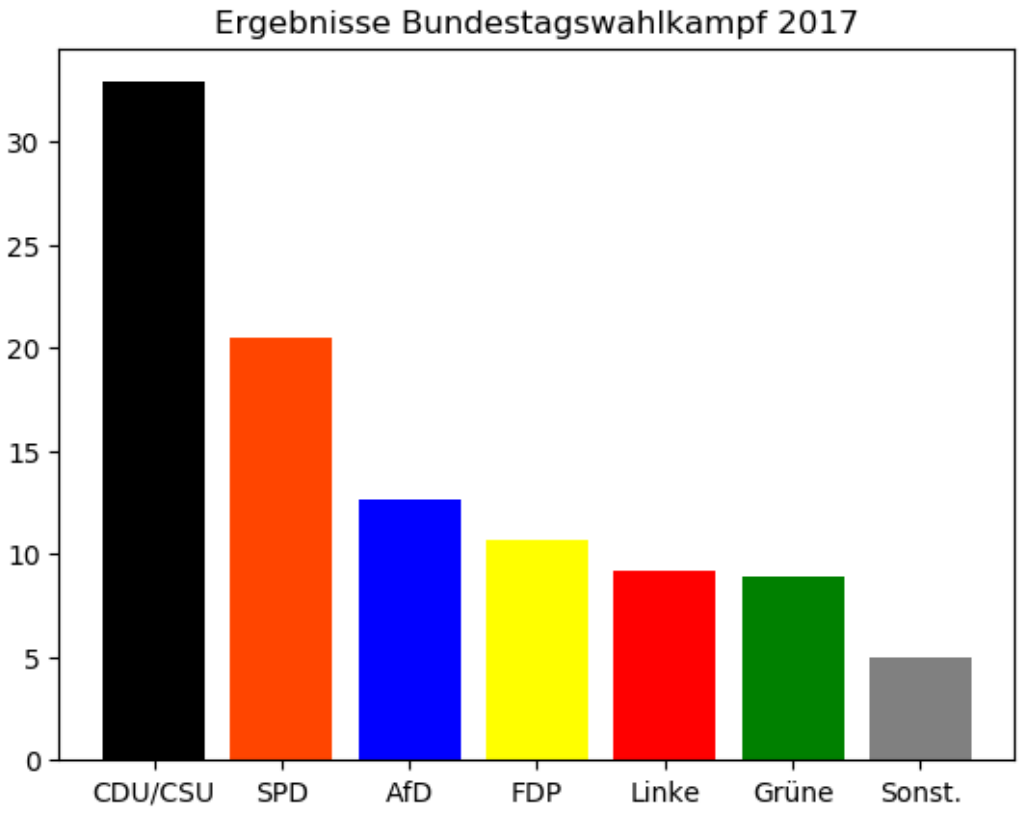
Häufig werden Säulendiagramme auch fälschlicherweise als Balkendiagramme bezeichnet, da sie diesen sehr ähnlich sind. Bei dem Balkendiagramm sind die Rechtecke allerdings waagerecht orientiert.

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt

# original default Parameter werden wiederhergestellt:
plt.rcdefaults()
fig, ax = plt.subplots()

personen = ('Michael', 'Dorothea', 'Robert', 'Bea', 'Uli')
y_pos = np.arange(len(personen))
verbrauch = (15, 22, 24, 39, 12)

ax.barh(y_pos, verbrauch, align='center',
        color='green', ecolor='black')
ax.set_yticks(y_pos)
ax.set_yticklabels(personen)
ax.invert_yaxis()  # labels von oben nach unten
ax.set_xlabel('Verbrauch')
ax.set_title('Kaffeekonsum')

plt.show()
```




```
parteien = ["CDU/CSU", "SPD", "AfD", "FDP", "Linke", "Grüne", "Sonst."]
```

```
colors = ['black', 'orangered', 'blue',  'yellow',  'red', 'green', 'grey']
anteile = [32.9, 20.5, 12.6, 10.7, 9.2, 8.9, 5]

bars = plt.bar(parteien, anteile, color=colors)
plt.title("Ergebnisse Bundestagswahlkampf 2017")
plt.show()
```



18 Matplotlib Konturdiagramme

18.0.1 Matplotlib-Tutorial: Konturflächen und -linien

WEBOFF

Eine Konturlinie oder Isolinie einer Funktion aus zwei Variablen ist eine Kurve entlang des konstanten Wertes der Funktion. Es ist ein Querschnitt des dreidimensionalen Graphen der Funktion f(x,y) parallel zur x,y-Ebene.

Konturlinien werden beispielsweise in der Geographie oder Meteorologie benutzt. In der Kartographie verbindet eine Konturlinie die Punkte gleicher Höhe über einem bestimmten Level, wie z.B. der mittlere Meeresspiegel

Allgemeiner können wir also sagen, dass eine Konturlinie einer Funktion mit zwei Variablen eine Kurve ist, die Punkte mit gleichen Werten verbindet.

Erstellen eines Maschengitters

Ein Maschengitter (Meshgrid) ist ein rechteckiges Gitter (Datengitter), was aus zwei eindimensionalen Arrays erzeugt wird, d.h. den x-Werten und den y-Werten. Im weiteren Verlauf dieses Kapitels werden wir nochmals auf die Funktion `meshgrid` und ihre Alternativen zurückkommen.

WEBOFF

```
import numpy as np

xlist = np.linspace(-3.0, 3.0, 3)
ylist = np.linspace(-3.0, 3.0, 4)
X, Y = np.meshgrid(xlist, ylist)
print(xlist)
print(ylist)
print(X)
print(Y)
```

```
[-3.  0.  3.]
[-3. -1.  1.  3.]
[[-3.  0.  3.]
 [-3.  0.  3.]
 [-3.  0.  3.]
 [-3.  0.  3.]]
[[-3. -3. -3.]
 [-1. -1. -1.]
 [ 1.  1.  1.]
 [ 3.  3.  3.]]
```

Berechnung der Werte

Nun berechnen wir die Funktionswerte zu den Wertepaaren des Maschengitters:

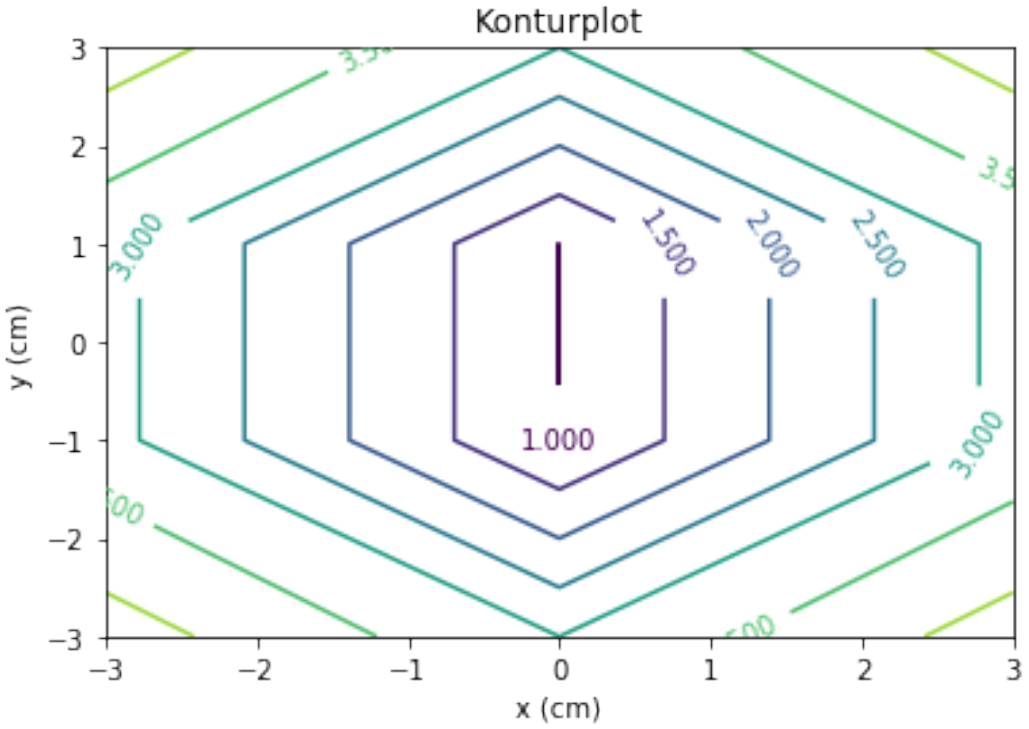
WEBOFF

```
import numpy as np

xlist = np.linspace(-3.0, 3.0, 3)
ylist = np.linspace(-3.0, 3.0, 4)
X, Y = np.meshgrid(xlist, ylist)

Z = np.sqrt(X**2 + Y**2)
print(Z)
```

```
[[4.24264069 3.         4.24264069]
 [3.16227766 1.         3.16227766]
 [3.16227766 1.         3.16227766]
 [4.24264069 3.         4.24264069]]
```

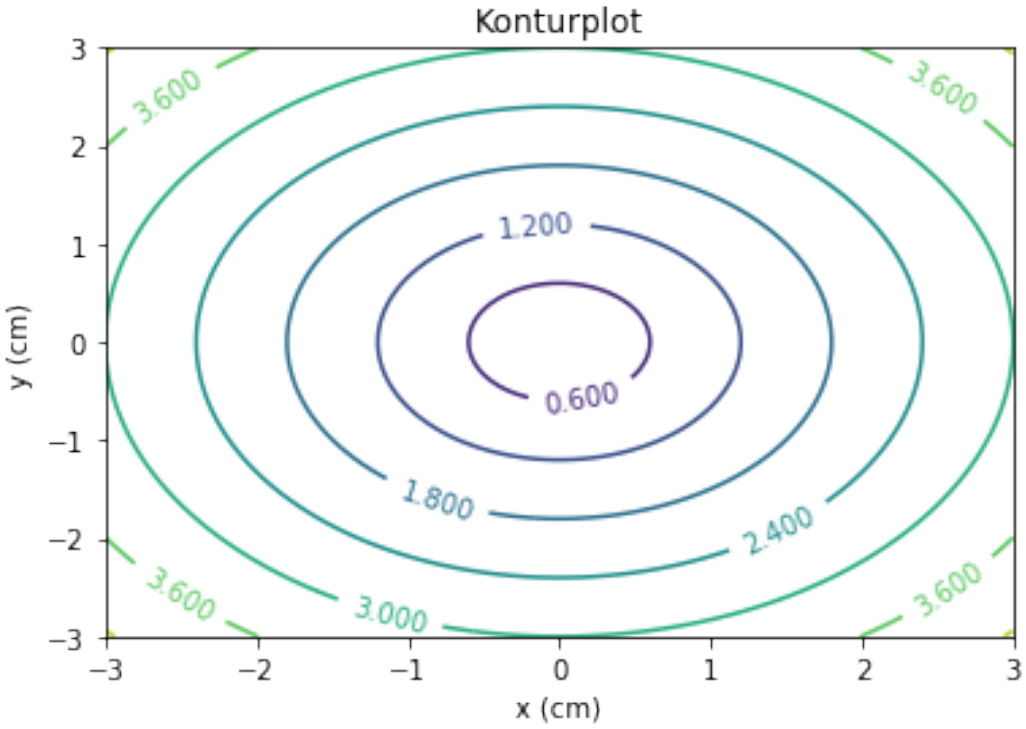
Aus den Daten erzeugen wir nun den Konturplot:

```
import matplotlib.pyplot as plt

plt.figure()
cp = plt.contour(X, Y, Z)
plt.clabel(cp, inline=True,
          fontsize=10)
plt.title('Konturplot')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.show()
```



Unser Konturplot sieht sehr kantig aus, weil unser Maschengitter nur aus 12 Punkten besteht. Im Folgenden verfeinern wir unser Maschengitter:

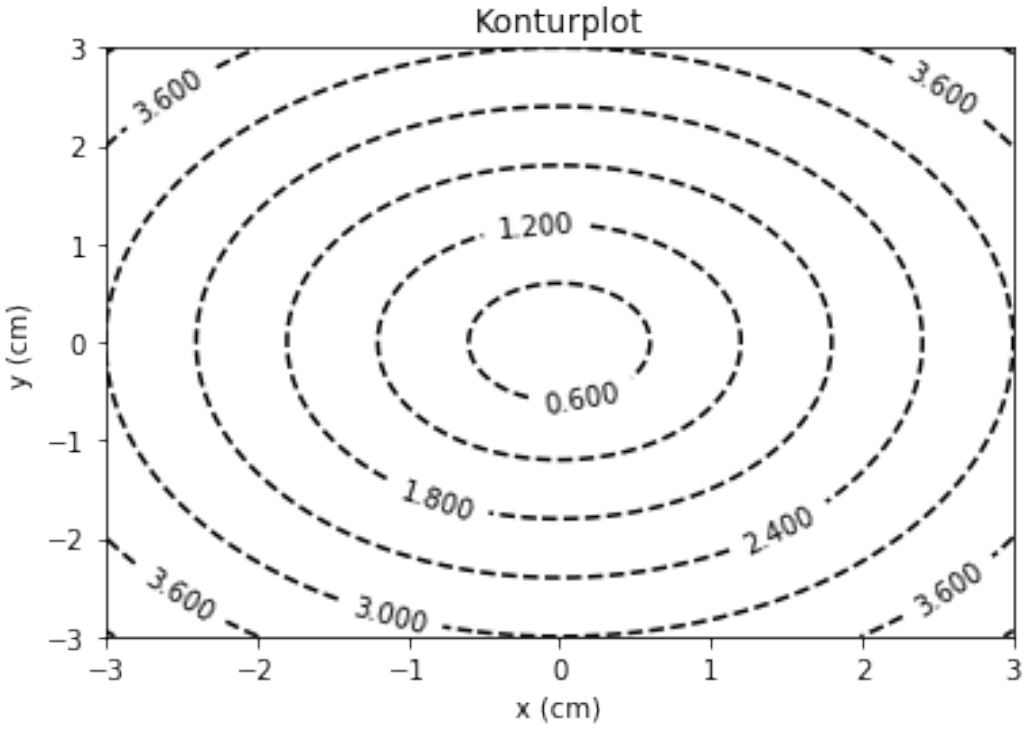
```
import numpy as np
import matplotlib.pyplot as plt

xlist = np.linspace(-3.0, 3.0, 100)
ylist = np.linspace(-3.0, 3.0, 100)
X, Y = np.meshgrid(xlist, ylist)

Z = np.sqrt(X**2 + Y**2)

plt.figure()
cp = plt.contour(X, Y, Z)
plt.clabel(cp, inline=True,
           fontsize=10)
plt.title('Konturplot')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.show()
```



Linienstil und Farben anpassen

Bisher hatten wir den Linienstil automatisch von Matplotlib bestimmen lassen, ebenso wie die Einfärbung. Mit den Parametern `linestyles` und `colors` können wir diese individuell einstellen.

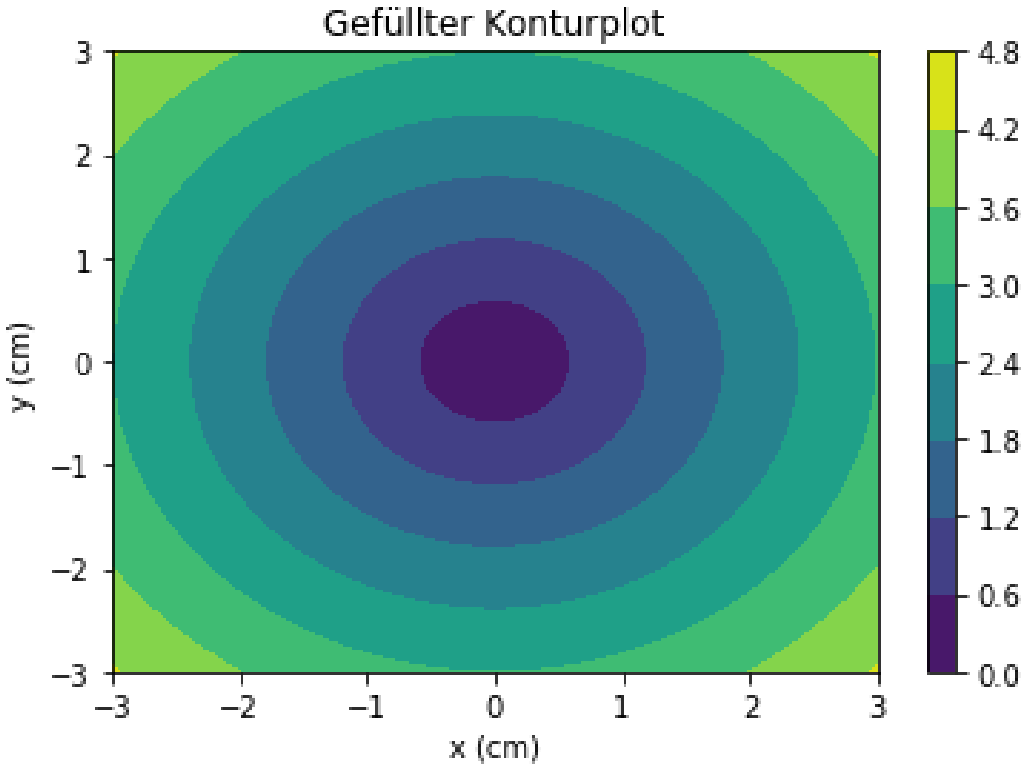
```
import matplotlib.pyplot as plt

plt.figure()
cp = plt.contour(X, Y, Z, colors='black', linestyles='dashed')
plt.clabel(cp, inline=True,
           fontsize=10)
plt.title('Konturplot')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.show()
```



Gefüllte Konturen

Wir können auch den Zwischenraum zwischen den Konturlinien einfärben:

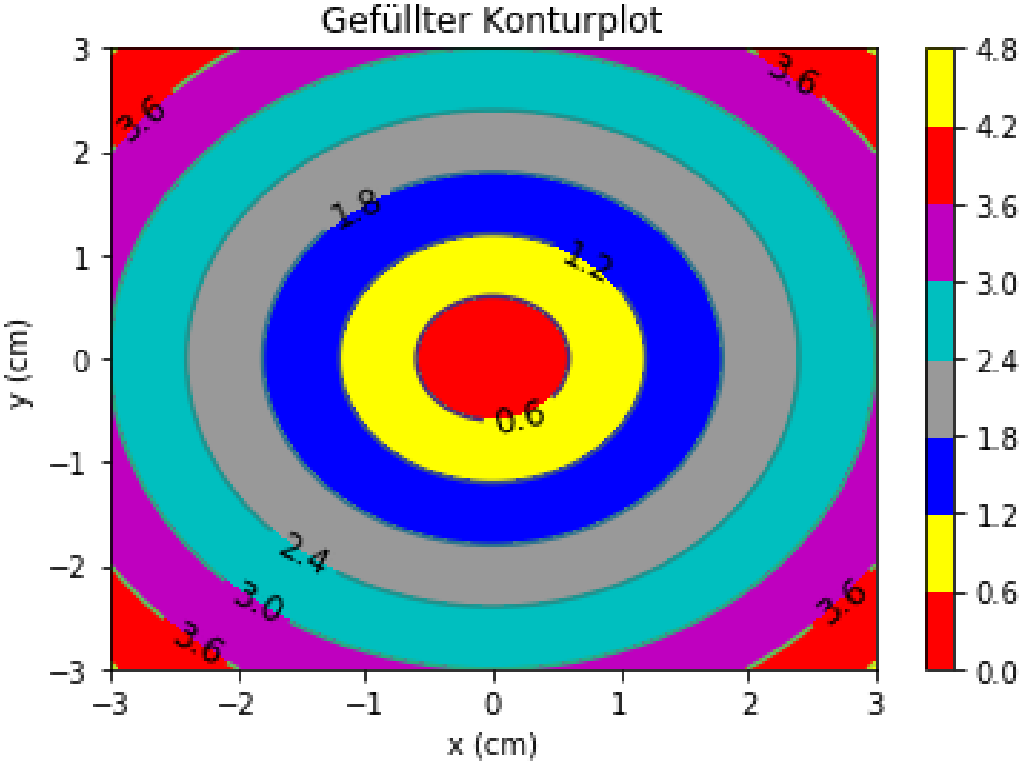
```
import numpy as np
import matplotlib.pyplot as plt

xlist = np.linspace(-3.0, 3.0, 100)
ylist = np.linspace(-3.0, 3.0, 100)
X, Y = np.meshgrid(xlist, ylist)
Z = np.sqrt(X**2 + Y**2)

plt.figure()

cp = plt.contourf(X, Y, Z)
plt.colorbar(cp)

plt.title('Gefüllter Konturplot')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.show()
```



Individuelle Farben

Die Farben für die Flächen können wir natürlich auch selbst bestimmen, wie wir im folgenden Beispiel sehen:

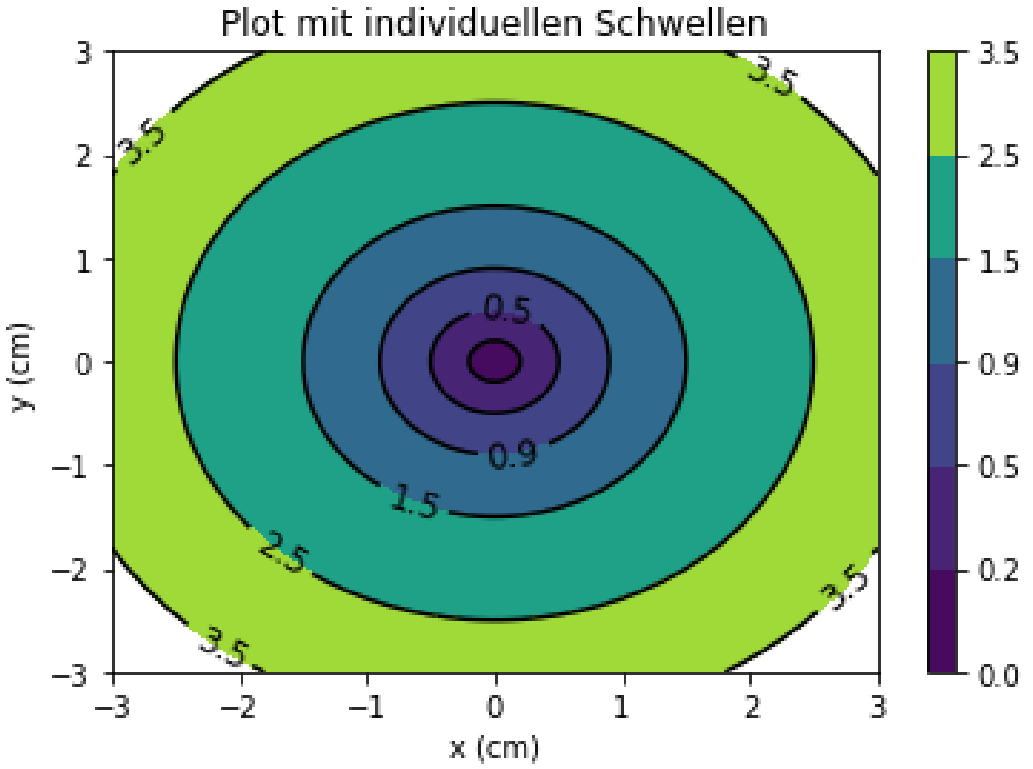
```
import numpy as np
import matplotlib.pyplot as plt

xlist = np.linspace(-3.0, 3.0, 100)
ylist = np.linspace(-3.0, 3.0, 100)
X, Y = np.meshgrid(xlist, ylist)
Z = np.sqrt(X**2 + Y**2)

plt.figure()

contour = plt.contour(X, Y, Z)
plt.clabel(contour, colors = 'k', fmt = '%2.1f', fontsize=12)
c = ('#ff0000', '#ffff00', '#0000FF', '0.6', 'c', 'm')
contour_filled = plt.contourf(X, Y, Z, colors=c)
plt.colorbar(contour_filled)

plt.title('Gefüllter Konturplot')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.show()
```



Schwellen

Die Schwellen für die Konturlinien und die Flächen werden automatisch durch `contour` und `contourf` gesetzt. Diese können auch manuell definiert werden, indem als viertes Argument eine Liste mit Levels übergeben wird. Konturlinien werden für jeden Wert in der Liste gezeichnet, wenn wir `contour` benutzen. Wenn `contourf` benutzt wird, so werden die Zwischenräume zwischen den Werten der Liste gefüllt.

```
import numpy as np
import matplotlib.pyplot as plt

xlist = np.linspace(-3.0, 3.0, 100)
ylist = np.linspace(-3.0, 3.0, 100)
X, Y = np.meshgrid(xlist, ylist)

Z = np.sqrt(X ** 2 + Y ** 2 )
plt.figure()

levels = [0.0, 0.2, 0.5, 0.9, 1.5, 2.5, 3.5]
contour = plt.contour(X, Y, Z, levels, colors='k')
plt.clabel(contour, colors = 'k', fmt = '%2.1f', fontsize=12)
contour_filled = plt.contourf(X, Y, Z, levels)
plt.colorbar(contour_filled)

plt.title('Plot mit individuellen Schwellen')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.show()
```



Andere Grids

Wir hatten bereits die NumPy-Funktion `meshgrid` kennengelernt. NumPy enthält aber noch zwei weitere wichtige Funktionen zur Erzeugung von gitterähnlichen Strukturen:

- `ogrid`
- `mgrid`

Meshgrid genauer Die Aufgabe von `meshgrid` besteht darin, wie wir gesehen hatten, aus zwei eindimensionalen Koordinatenvektoren eine zweidimensionale Koordinatenmatrix zu erzeugen. Im allgemeinen Fall kann man aus `n` eindimensionalen Array-ähnlichen Strukturen ein `n`-dimensionales Array zur vektorisierten Auswertung von `n`-dimensionalen Vektorfeldern über einem `n`-Gitter erzeugen.

Man könnte eine Gitterstruktur (englisch "grid") auch ohne `meshgrid` erzeugen. Im folgenden Beispiel erzeugen wir ein Gitter G mit den Werten 0, 1, 2 als x- und als y-Werte:

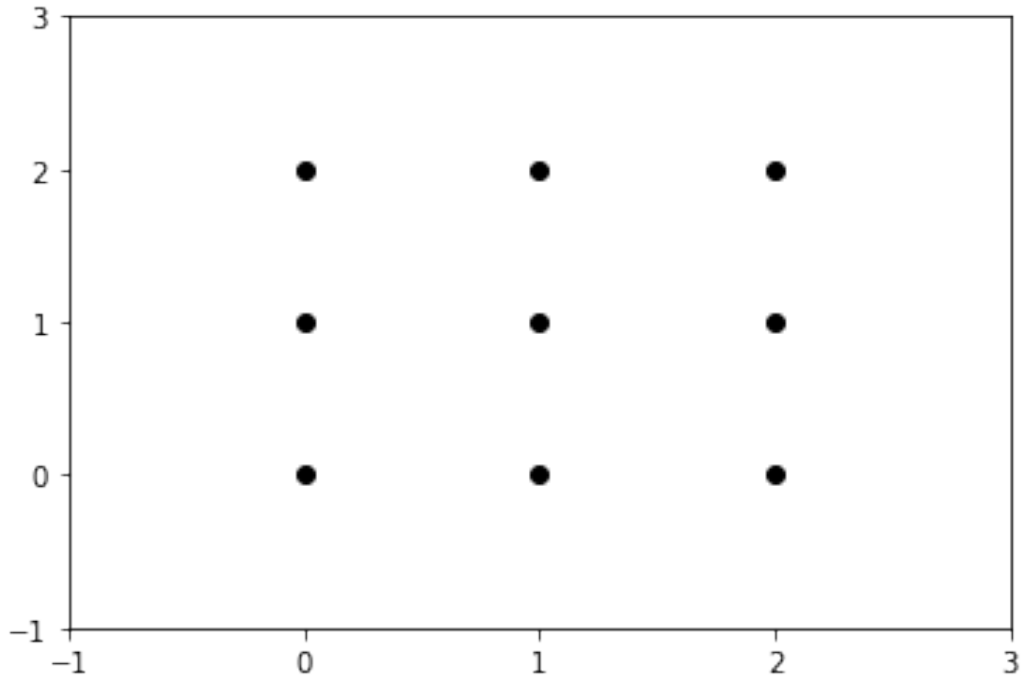
```
n = 3
X, Y = np.zeros((n, n), np.int8), np.zeros((n, n), np.int8)
for row in range(0, n):
    for col in range(0, n):
        X[row, col] = col
        Y[row, col] = row

print(X)
print(Y)
```

```
[[0 1 2]
```

```
 [0 1 2]
 [0 1 2]]
[[0 0 0]
 [1 1 1]
 [2 2 2]]
```

Das $(3, 3)$-Gitter entspricht den Paarungen der entsprechenden Komponenten aus den Arrays X und Y, also $X[i, j]$ gepaart mit $Y[i, j]$ mit $0 <= i <= 2$ und $0 <= j <= 2$. Mit einem Plot können wir dieses Gitter sichtbar machen:

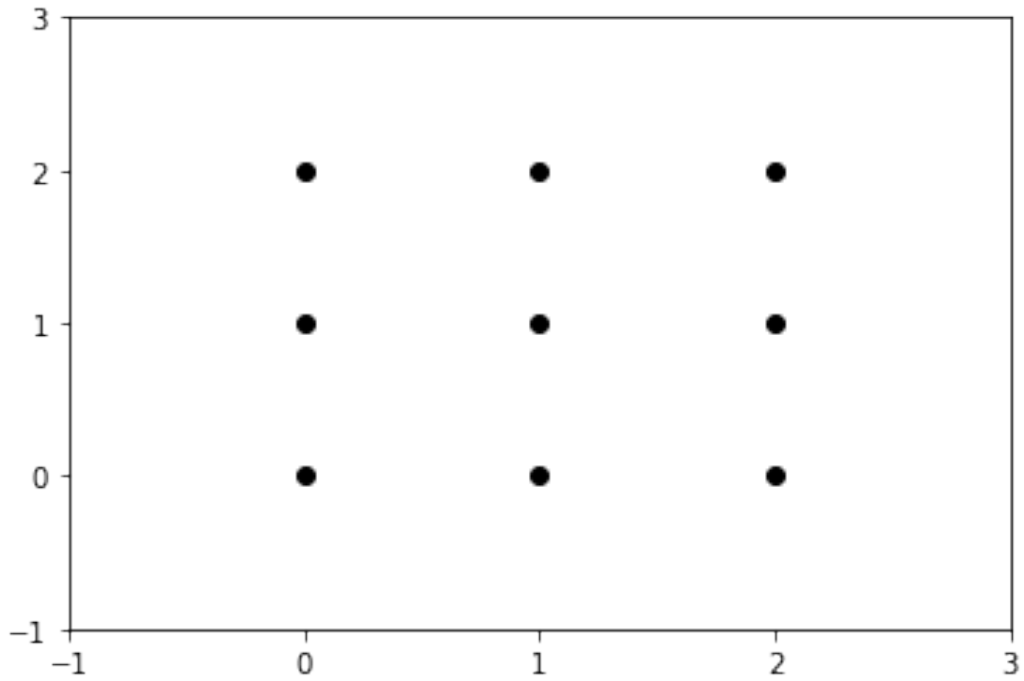
```
import matplotlib.pyplot as plt
import numpy as np

plt.plot(X, Y, marker='o', color='k', linestyle='none')
plt.xticks(range(-1, n+1))
plt.yticks(range(-1, n+1))
plt.show()
```



Dies war der umständliche direkte Weg, und mit `meshgrid` geht es deutlich schneller und leichter:

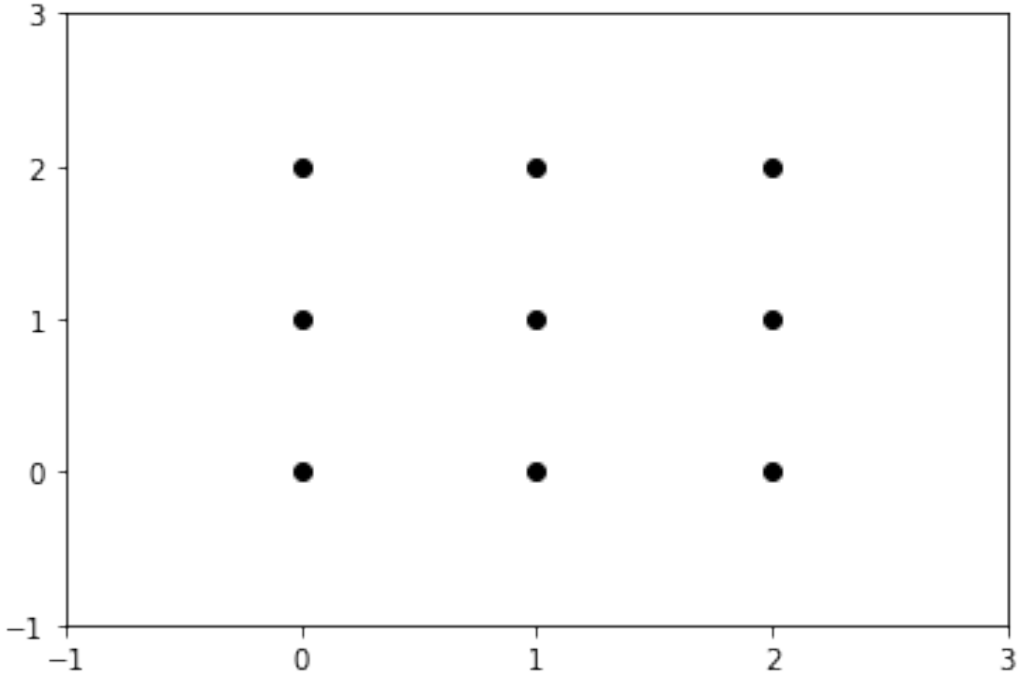
```
import matplotlib.pyplot as plt
import numpy as np

n = 3
x_values = np.arange(0, n)
y_values = np.arange(0, n)

X, Y = np.meshgrid(x_values, y_values)

plt.plot(X, Y, marker='o', color='k', linestyle='none')
plt.xticks(range(-1, n+1))
```

```
plt.yticks(range(-1, n+1))
plt.show()
```



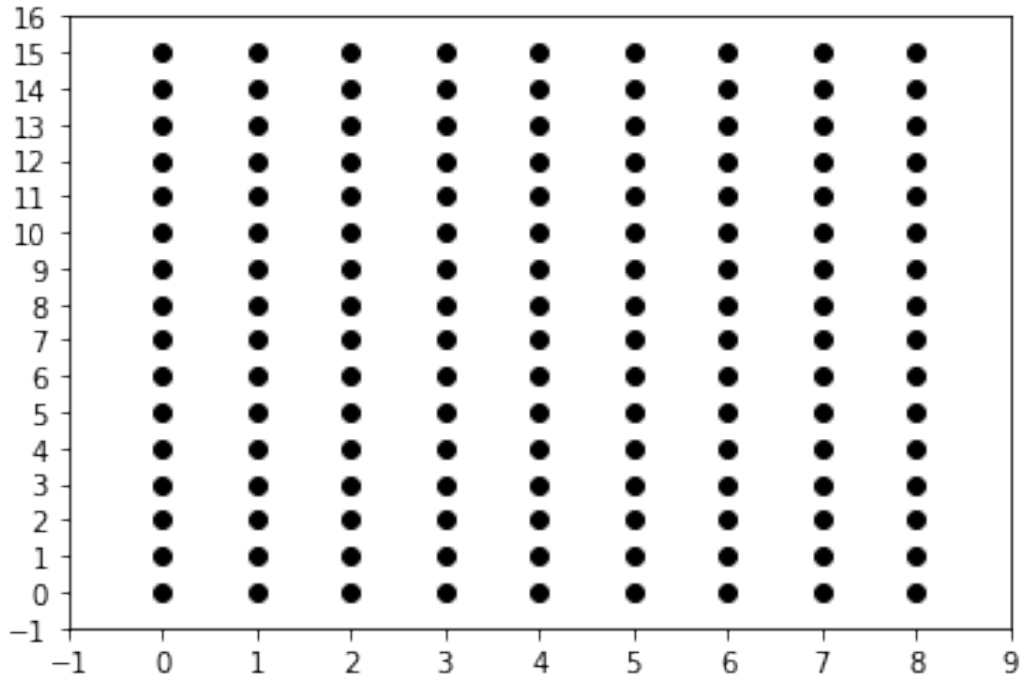
Im gewissen Sinne ist `meshgrid` überflüssig, da man das gleiche Resultat auch mittels Broadcasting erreichen kann:

```
import matplotlib.pyplot as plt
import numpy as np

n = 3
x_values = np.arange(0, n)
y_values = np.arange(0, n)

# meshgrid mit broadcasting:
X = np.ones((n, 1)) * x_values
Y = y_values.reshape((n, 1)) * np.ones((1, n))

plt.plot(X, Y, marker='o', color='k', linestyle='none')
plt.xticks(range(-1, n+1))
plt.yticks(range(-1, n+1))
plt.show()
```



Wir hatten mit `meshgrid` eine quadratische Gitterstruktur erzeugt. Selbstverständlich können wir auch beliebige rechteckige Strukturen erzeugen:

```
import matplotlib.pyplot as plt
import numpy as np

n, m = 9, 16
x_values = np.arange(0, n)
y_values = np.arange(0, m)

X, Y = np.meshgrid(x_values, y_values)

plt.plot(X, Y, marker='o', color='k', linestyle='none')
plt.xticks(range(-1, n+1))
plt.yticks(range(-1, m+1))
plt.show()
```



mgrid `mgrid` benötigt keine Array-ähnlichen Eingabevektoren, sondern wird mit Indices indiziert. Deshalb verwenden wir hier auch eckige Klammern, da es sich nicht um einen Funktionsaufruf handelt. `mgrid` und `meshgrid` liefern prinzipiell das selbe Ergebnis, allerdings sind die Achsen vertauscht.

```
import numpy as np

n = 3
X_mgrid, Y_mgrid = np.mgrid[0:n, 0:n]

n = 3
X_meshgrid, Y_meshgrid = np.meshgrid(np.arange(0, n),
                                     np.arange(0, n))

print(X_mgrid == Y_meshgrid)
print(Y_mgrid == X_meshgrid)
```

```
[[ True  True  True]
 [ True  True  True]
 [ True  True  True]]
[[ True  True  True]
 [ True  True  True]
 [ True  True  True]]
```

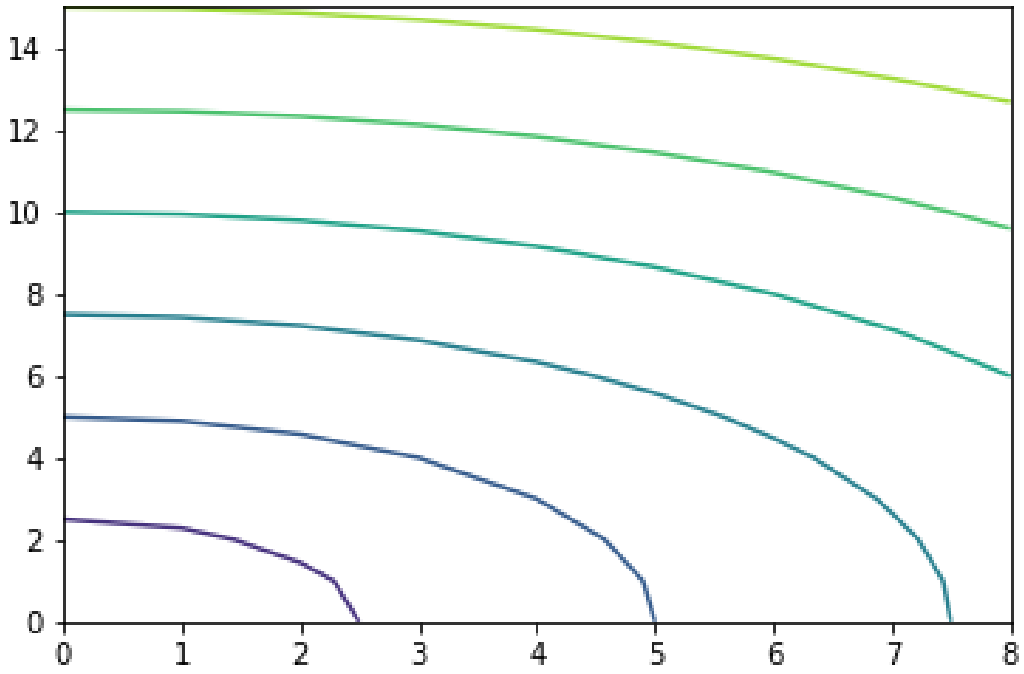
ogrid Wir haben sowohl bei `meshgrid` als auch bei `mgrid` gesehen, dass sich die Werte der beiden erzeugten Matrizen jeweils zeilen- bzw. spaltenweise wiederholen. `ogrid` liefert nun jeweils nur einen Zeilen- und einen Spaltenvektor zurück. Dadurch erhalten wir eine speicherschonende Repräsentierung der Werte. Mittels Broadcasting können dann andere Funktionen, die diese Matrizen benötigen, diese implizit erzeugen.

```
import numpy as np
```

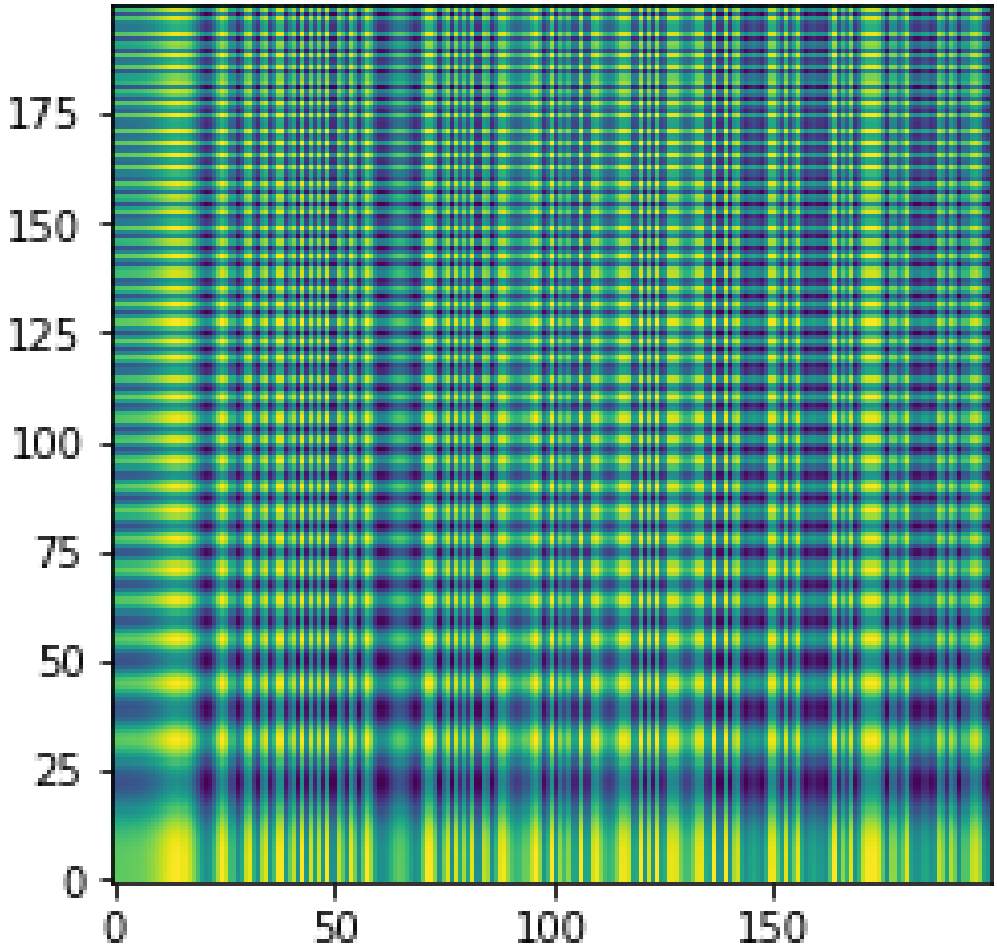
```
n = 5
X_ogrid, Y_ogrid = np.ogrid[0:n, 0:n]
print(X_ogrid)
print(Y_ogrid)
```

```
[[0]
 [1]
 [2]
 [3]
 [4]]
[[0 1 2 3 4]]
```

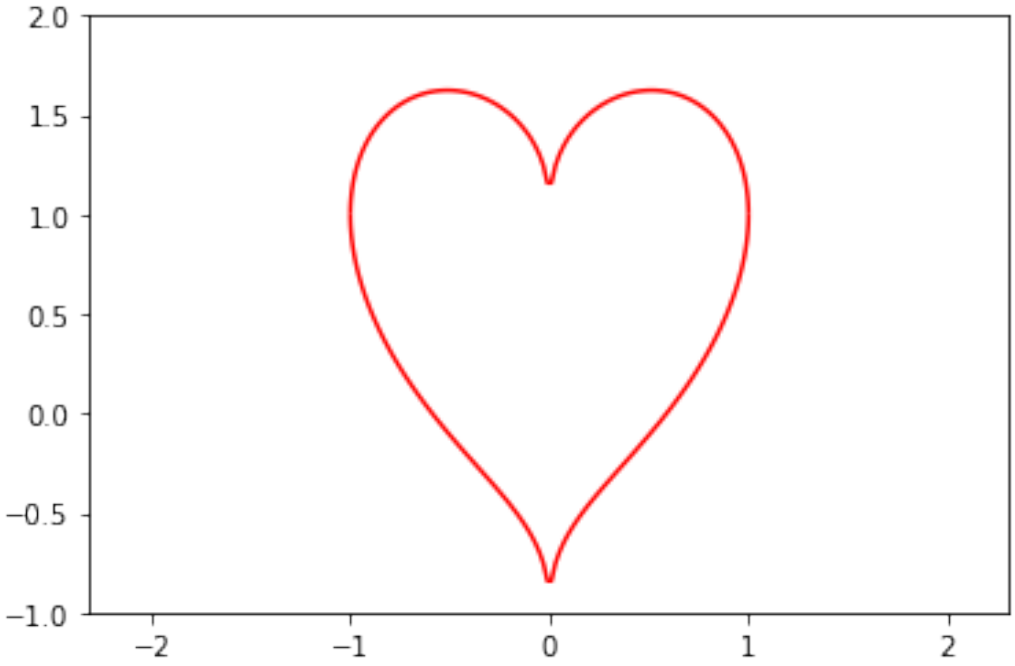
```
Z = np.sqrt(X**2 + Y**2)

plt.figure()
cp = plt.contour(X, Y, Z)
```



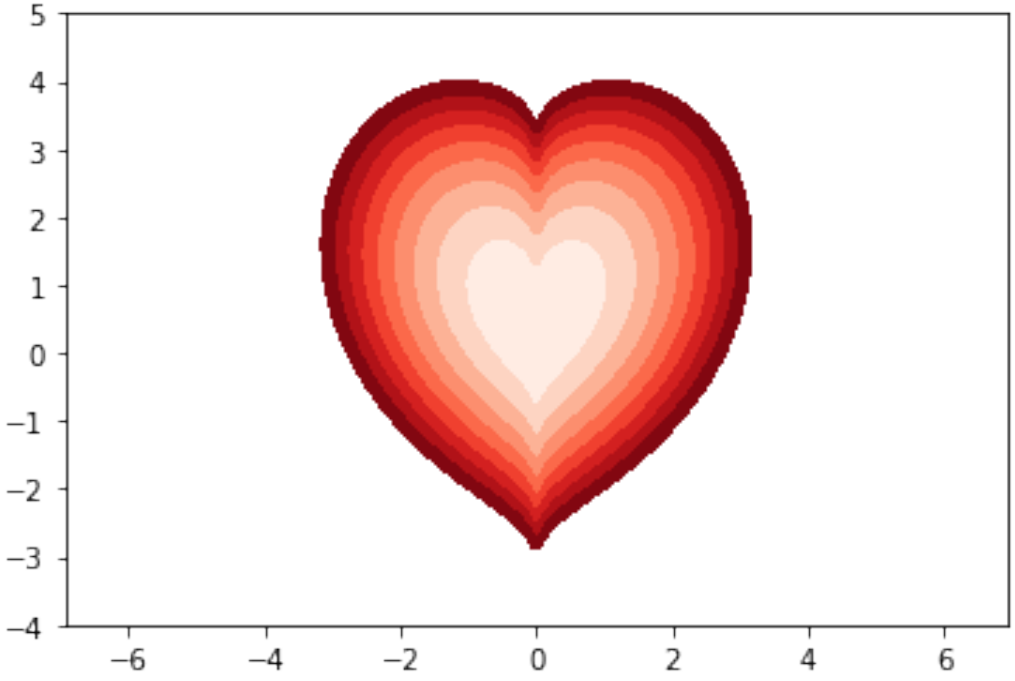

```
# progr4book

import numpy as np
import matplotlib.pyplot as plt

def sin2d(x, y):
    return np.sin(x**3) + np.cos(y**2)

X, Y = np.meshgrid(np.linspace(0, 5*np.pi, 200),
                   np.linspace(0, 5*np.pi, 200))
Z = sin2d(X, Y)

plt.imshow(Z, origin='lower')
plt.show()
```




```
# prog4book
import matplotlib.pyplot as plt
import numpy as np

y, x = np.ogrid[-1:2:100j, -1:1:100j]


plt.contour(x.ravel(),
            y.ravel(),
            x ** 2 + (y - ((x ** 2) ** (1.0 / 5))) ** 2,
            [1],
            colors='red')
plt.axis('equal')
plt.show()
```




```
# prog4book
import matplotlib.pyplot as plt
import numpy as np

y, x = np.ogrid[-4:5:100j, -4:4:100j]

plt.contourf(x.ravel(),
             y.ravel(),
             x ** 2 + (y - ((x ** 2) ** (1.0/5))) ** 2,
             levels=np.linspace(0, 10, 10),
            cmap='Reds')
plt.axis('equal')
plt.show()
```


5
4
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2
4
6

19 Pandas

```
# invisible
import numpy as np
import pandas as pd
np.set_printoptions(linewidth=60)


pd.set_option('display.max_colwidth', 65)
pd.set_option('display.max_columns', 5)
```

19.0.1 Pandas Einführung

pandas ist eine Python-Bibliothek für Datenanalyse und Datenmanipulation. Sie stellt beschriftete Datenstrukturen bereit und eignet sich besonders für tabellarische Daten, Zeitreihen und heterogene Datensätze. Der Name geht auf den statistischen Begriff „panel data“ zurück.

Die beiden zentralen Datenstrukturen sind `Series` (eindimensional) und `DataFrame` (zweidimensional). pandas baut eng auf NumPy auf, ergänzt dessen Arrays aber um Labels, Indizes, komfortable Auswahloperationen, fehlende Werte, Gruppierungen, Zusammenführungen und Datei-I/O.

NumPy ist eine Kernabhängigkeit von pandas; Bibliotheken wie Matplotlib ergänzen pandas beispielsweise bei der Visualisierung.

Daten-Strukturen in Pandas

Wir beginnen mit beiden wichtigsten Daten-Strukturen von Pandas:

- Series und
- DataFrame

Series

Eine `Series` ist eine eindimensionale, beschriftete Datenstruktur. Sie besitzt einen Index und einen `dtype`. Abhängig vom Datentyp kann eine Series Zahlen, Strings, Datumswerte, boolesche Werte oder auch Python-Objekte enthalten; innerhalb einer Series beschreibt der `dtype` jedoch den gemeinsamen Speichertyp bzw. Extension-Datentyp der Werte.

Eine Series kann als eine Datenstruktur mit zwei Arrays angesehen werden: Ein Array fungiert als Index, d.h. als Bezeichner (Label), und ein Array beinhaltet die aktuellen Daten (Werte).

Wir definieren im folgenden Beispiel ein einfaches Series-Objekt, indem wir dieses Objekt

mit einer Liste instanziieren. Wir werden später sehen, dass wir auch andere Daten-Objekte verwenden können, z.B. Numpy-Arrays und Dictionaries.

```
import pandas as pd
S = pd.Series([11, 28, 72, 3, 5, 8])
print(S)
```

```
0    11
1    28
2    72
3     3
4     5
5     8
dtype: int64
```

Wir haben in unserem Beispiel keinen Index definiert. Trotzdem sehen wir zwei Spalten in der Ausgabe: Die rechte Spalte zeigt unsere Daten, die linke Spalte stellt den Index dar. Pandas erstellt einen Default-Index, der bei 0 beginnt und bis 5 läuft.

Wir können direkt auf die Indizes und die Werte der Series S zugreifen:

```
print(S.index)
print(S.values)
```

```
RangeIndex(start=0, stop=6, step=1)
[11 28 72  3  5  8]
```

Wenn wir dies mit der Erstellung eines Arrays in NumPy vergleichen, stellen wir viele Gemeinsamkeiten fest:

```
import numpy as np
X = np.array([11, 28, 72, 3, 5, 8])
print(X)
print(S.values)
# both are the same type:
print(type(S.values), type(X))
```

```
[11 28 72  3  5  8]
[11 28 72  3  5  8]
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

Bis hierhin unterscheiden sich die Series noch nicht wirklich von den ndarrays aus Numpy. Das ändert sich aber, sobald wir Series-Objekte mit individuellen Indizes definieren:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
print(S)
```

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

Eine großer Vorteil gegenüber Numpy-Arrays ist hier ganz offensichtlich: Wir können beliebige Indizes verwenden.

Wenn wir zwei Series-Objekte mit denselben Indizes addieren, so erhalten wir ein neues Series-Objekt mit diesem Index, und die Werte entsprechen den Summen der entsprechenden Werte aus den beiden Series-Objekten.

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("Summe aus S: ", sum(S))
```

```
apples      37
oranges     46
cherries    83
pears       42
dtype: int64
Summe aus S:  115
```

Die Indizes müssen nicht identisch sein für die Addition von Series-Typen. Der resultierende Index ist eine "Vereinigung" beider Indizes. Wenn ein Index nicht in beiden Series-Objekten vorkommt, so wird der entsprechende Wert auf NaN gesetzt:

```
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

```
cherries       83.0
oranges        46.0
peaches         NaN
pears          42.0
raspberries     NaN
dtype: float64
```

Prinzipiell können die Indizes auch komplett verschieden sein, wie im folgenden Beispiel. Der zweite Index besteht aus der türkischen Übersetzung der englischen Fruchtnamen:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

fruits_tr = ['elma', 'portakal', 'kiraz', 'armut']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits_tr)
print(S + S2)
```

```
apples     NaN
armut      NaN
cherries   NaN
elma       NaN
kiraz      NaN
```

```
oranges    NaN
pears      NaN
portakal   NaN
dtype: float64
```

Indizierung Es ist möglich, auf einzelne Werte eines Series-Objektes zuzugreifen:

```
print(S['apples'])
```

```
20
```

Man kann auch auf mehrere Indizes gleichzeitig zugreifen, wenn man die Indices als Liste übergibt:

```
print(S[['apples', 'oranges', 'cherries']])
```

```
apples      20
oranges     33
cherries    52
dtype: int64
```

Filterung mit einem Booleschen Array:

```
S[S>30]
```

```
oranges     33
cherries    52
dtype: int64
```

Wie bei Numpy sind auch Operationen mit Skalaren oder die Anwendung von mathematischen Funktionen auf ein Series-Objekt möglich:

```
import numpy as np
print((S + 3) * 4)
print("======================")
print(np.sin(S))
```

```
apples       92
oranges     144
cherries    220
pears        52
dtype: int64
======================
apples      0.912945
oranges     0.999912
cherries    0.986628
pears      -0.544021
dtype: float64
```

pandas.Series.apply Ab pandas 3.0 lautet die Signatur:

```
Series.apply(func, args=(), *, by_row=compat", **kwargs)
```

`apply()` ruft eine Funktion für Werte einer Series auf. Für einfache elementweise Abbildungen ist auch `Series.map()` häufig eine gute Wahl; für Aggregationen stehen unter anderem `agg()` bzw. `aggregate()` zur Verfügung.

Parameter	Bedeutung
`func`	Aufzurufende Python-Funktion oder NumPy-Ufunc.
`args`	Zusätzliche Positionsargumente, die an `func` weitergereicht werden.
`by_row`	Steuert bei aktuellen pandas-Versionen die Art der Anwendung; der Standardwert ist `compat"`.
`**kwargs`	Zusätzliche Schlüsselwortargumente für `func`.

Bei pandas 2.x kann die lokal angezeigte Signatur noch den inzwischen entfernten Parameter `convert_dtype` enthalten.

Beispiel:

```
S.apply(np.log)
```

```
apples      2.995732
oranges     3.496508
cherries    3.951244
pears       2.302585
dtype: float64
```

Wir können auch Python-Lambda-Funktionen benutzen. Wir werden nun die Anzahl der Früchte prüfen: Wenn weniger als 50 von einer Sorte vorhanden sind, so soll der Bestand um 10 erhöht werden. Ansonsten lassen wir den Betrag unverändert:

```
S.apply(lambda x: x if x > 50 else x+10 )
```

```
apples      30
oranges     43
cherries    52
pears       20
dtype: int64
```

Zusammenhang zu Dictionaries Ein Series-Objekt kann angesehen werden wie ein geordnetes Python-Dictionary mit einer festen Länge.

Wir können bei der Erstellung eines Series-Objektes ein Dictionary übergeben. Wir erhalten ein Series-Objekt mit den Schlüsseln des Dictionarys als Indizes.

```
cities = {"London":    8615246,
          "Berlin":    3562166,
          "Madrid":    3165235,
          "Rome":      2874038,
          "Paris":     2273305,
          "Vienna":    1805681,
```

```
          "Bucharest": 1803425,
          "Hamburg":   1760433,
          "Budapest":  1754000,
          "Warsaw":    1740119,
          "Barcelona": 1602386,
          "Munich":    1493900,
          "Milan":     1350680}
city_series = pd.Series(cities)
print(city_series)
```

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

NaN - Fehlende Daten

Ein Problem bei Aufgaben in der Datenanalyse besteht in fehlenden Daten.

Schauen wir uns noch einmal das vorherige Beispiel an. Dabei erkennen wir, dass die Indizes der Series mit den Keys des Dictionaries übereinstimmt, aus dem das Series-Objekt cities_series erzeugt wurde. Nehmen wir nun an, dass wir einen Index haben wollen, der sich nicht mit den Keys des Dictionaries überschneidet. Dafür können wir eine Liste oder ein Tupel dem Keyword-Argument 'index' mitgeben, um die Indizes zu definieren. Im nächsten Beispiel übergeben wir eine Liste (oder ein Tupel) als Indizes, welches nicht mit den Keys übereinstimmt. Das bedeutet, dass einige Städte des Dictionaries fehlen und für Stuttgart und Zürich keine Daten vorhanden sind.

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities, index=my_cities)
print(my_city_series)
```

```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

Abgesehen von den NaN-Werten werden bei den anderen Bevölkerungswerten die Werte in float-Werte gewandelt. Im folgenden Beispiel gibt es keine fehlenden Daten, und damit werden die Werte in Integer-Werte gewandelt:

```
my_cities = ["London", "Paris", "Berlin", "Hamburg"]
my_city_series = pd.Series(cities, index=my_cities)
my_city_series
```

```
London     8615246
Paris      2273305
Berlin     3562166
Hamburg    1760433
dtype: int64
```

Fehlende Werte mit isna() und notna() Wir sehen, dass für nicht vorhandene Städte ein fehlender Wert entsteht. Je nach Datentyp verwendet pandas unterschiedliche Missing-Value-Repräsentationen, darunter `NaN`, `NaT` und `pd.NA`.

Mit `isna()` und `notna()` können wir fehlende Werte prüfen. Die älteren Namen `isnull()` und `notnull()` existieren weiterhin als Aliase:

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities, index=my_cities)
print(my_city_series.isna())
```

```
London       False
Paris        False
Zurich        True
Berlin       False
Stuttgart     True
Hamburg      False
dtype: bool
```

```
print(my_city_series.notna())
```

```
London        True
Paris         True
Zurich       False
Berlin        True
Stuttgart    False
Hamburg       True
dtype: bool
```

Zusammenhang zwischen NaN und None Wir erhalten ebenfalls `NaN`, wenn ein Wert in dem Dictionary `None` ist:

```
d = {"a":23, "b":45, "c":None, "d":0}
S = pd.Series(d)
print(S)
```

```
a    23.0
b    45.0
c     NaN
d     0.0
dtype: float64
```

```
pd.isna(S)
```

```
a    False
b    False
c     True
d    False
dtype: bool
```

```
pd.notna(S)
```

```
a     True
b     True
c    False
d     True
dtype: bool
```

Fehlende Daten filtern Es ist möglich, die fehlenden Daten mit der Methode `dropna` aus einem Series-Objekt herauszufiltern. Die Methode liefert ein neues Series-Objekt zurück, welches keine `NaN`-Werte enthält:

```
print("Vorher:\n")
print(my_city_series)
print("\nNachher:\n")
print(my_city_series.dropna())
```

```
Vorher:

London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64

Nachher:

London     8615246.0
Paris      2273305.0
Berlin     3562166.0
Hamburg    1760433.0
dtype: float64
```

Fehlende Daten auffüllen In vielen Fällen will man die fehlenden Daten gar nicht filtern. Stattdessen möchten man diese mit passenden Werten auffüllen. Eine gute Methode ist `fillna`:

```
print(my_city_series.fillna(0))
```

```
London       8615246.0
Paris        2273305.0
Zurich             0.0
```

```
Berlin       3562166.0
Stuttgart          0.0
Hamburg      1760433.0
dtype: float64
```

Okay, das sind nicht wirklich passende Werte für die Bevölkerung von Zürich und Stuttgart. Wenn wir der Methode fillna() ein Dictionary mitgeben, können wir so die passenden Daten bereitstellen, z.B. die Bevölkerungswerte für Zürich und Stuttgart. Wir setzen den Parameter `inplace` auf `True`, damit die Änderungen auch in dem Objekt geändert werden. Bei `True` wird ein neues Objekt mit den Einsetzungen erzeugt und zurückgeliefert, und das alte bleibt dabei unverändert:

```
missing_cities = {"Stuttgart":597939, "Zurich":378884}
my_city_series.fillna(missing_cities, inplace=True)
my_city_series
```

```
London       8615246.0
Paris        2273305.0
Zurich        378884.0
Berlin       3562166.0
Stuttgart     597939.0
Hamburg      1760433.0
dtype: float64
```

Dabei haben wir aber immer noch das Problem mit Integer-Werten. Die Werte, die Integer sein sollten, wie die Anzahl der Menschen, werden nach wie vor in Float-Werte gewandelt. Mit der Methode `astype` können wir die Daten in Integer wandeln:

```
my_city_series = my_city_series.astype(int)
print(my_city_series)
```

```
London       8615246
Paris        2273305
Zurich        378884
Berlin       3562166
Stuttgart     597939
Hamburg      1760433
dtype: int64
```

20 Pandas Dataframe

```
# invisible
import numpy as np
import pandas as pd
np.core.arrayprint._line_width = 60


pd.set_option('display.max_colwidth', 65)
pd.set_option('display.max_columns', 5)
```

Im vorigen Kapitel haben wir gesehen, dass der Datentyp `Series` logisch gesehen einer Spalte mit Index einer Excel-Tabelle eintspricht. In diesem Kapitel geht es nun um den Datentyp `DataFrame`, den man sich nun wie eine komplette Excel-Tabelle vorstellen kann. Man kann also sagen, dass dieser Datentyp auf Tabellen basiert. Ein `DataFrame` besteht aus einer geordneten Sequenz von Spalten. Jede Spalte besteht aus einem eindeutigen Daten-Typ – wie eine `Series`, – aber verschiedene Spalten können verschiedene Typen haben. So könnte beispielsweise eine Spalte Verkaufszahlen als Float-Zahlen enthalten, während eine andere die zugehörigen Jahreszahlen als Integer-Zahlen enthalten könnte.

Zusammenhang zu Series

Ein DataFrame hat einen Zeilen- und einen Spalten-Index. Es ist wie ein Dictionary aus Series mit einem normalen Index. Jede Series wird über einen Index, d.h. Namen der Spalte angesprochen. Wir demonstrieren diesen Zusammenhang im folgenden Beispiel, in dem drei Series-Objekte definiert und zu einem `DataFrame` zusammengebaut werden:

20.0.1 DataFrame

Die grundlegende Idee von DataFrame basiert auf Tabellen. Wir können die Daten-Struktur eines DataFrame als tabellarisch und tabellenähnlich sehen. Ein Dataframe beinhaltet eine geordnete Sammlung von Spalten. Jede Spalte besteht aus einem eindeutigen Daten-Typen, aber verschiedene Spalten haben verschiedene Typen, z.B. könnte die erste Spalte vom Typ Integer sein, während die zweite Spalte vom Typ Boolean ist, usw.

Ein DataFrame hat einen Zeilen- und ein Spalten-Index. Es ist wie ein Dictionary aus Series mit einem normalen Index. Wir demonstrieren dies im folgenden Beispiel, in dem drei Series-Objekte definiert werden:

```
import pandas as pd

years = range(2014, 2018)

shop1 = pd.Series([2409.14, 2941.01, 3496.83, 3119.55], index=years)
shop2 = pd.Series([1203.45, 3441.62, 3007.83, 3619.53], index=years)
```

```
shop3 = pd.Series([3412.12, 3491.16, 3457.19, 1963.10], index=years)
```

Was passiert, wenn diese "shop"-Series-Objekte konkateniert werden? Pandas stellt eine concat()-Funktion für diesen Zweck zur Verfügung:

```
pd.concat([shop1, shop2, shop3])
```

```
2014    2409.14
2015    2941.01
2016    3496.83
2017    3119.55
2014    1203.45
2015    3441.62
2016    3007.83
2017    3619.53
2014    3412.12
2015    3491.16
2016    3457.19
2017    1963.10
dtype: float64
```

Das Ergebnis ist wohl nicht das, was wir erwartet haben. Der Grund dafür ist, dass concat() für den axis-Parameter 0 verwendet. Probieren wir es mit "axis=1":

```
shops_df = pd.concat([shop1, shop2, shop3], axis=1)
print(shops_df)
```

```
            0        1        2
2014  2409.14  1203.45  3412.12
2015  2941.01  3441.62  3491.16
2016  3496.83  3007.83  3457.19
2017  3119.55  3619.53  1963.10
```

Die Frage ist, von welchem Datentyp das Ergebnis ist:

```
print(type(shops_df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

Das bedeutet, dass wir Series-Objekte durch Konkatenierung in DataFrame-Objekte wandeln können!

Die Spaltennamen lauten:

```
shops_df.columns
```

```
RangeIndex(start=0, stop=3, step=1)
```

```
shops_df.columns.values
```

```
array([0, 1, 2])
```

Wir geben den Spalten noch Namen, um das DataFrame etwas leichter lesbar zu machen:

```
cities = ["Zürich", "Winterthur", "Freiburg"]
shops_df.columns = cities
print(shops_df)
```

```
      Zürich  Winterthur  Freiburg
2014  2409.14     1203.45   3412.12
2015  2941.01     3441.62   3491.16
2016  3496.83     3007.83   3457.19
2017  3119.55     3619.53   1963.10
```

Andererseits wäre eine Umbenennung in unserem Fall überhaupt nicht notwendig gewesen, wenn die Series bereits entsprechend benamt gewesen wären. Wir zeigen dies in folgendem Fall:

```
shop1.name = "Zürich"
shop2.name = "Winterthur"
shop3.name = "Freiburg"
shops_df2 = pd.concat([shop1, shop2, shop3], axis=1)
print(shops_df2)
```

```
      Zürich  Winterthur  Freiburg
2014  2409.14     1203.45   3412.12
2015  2941.01     3441.62   3491.16
2016  3496.83     3007.83   3457.19
2017  3119.55     3619.53   1963.10
```

Auf die Spalten eines Dataframes können wir einfach durch Indizierung zugreifen:

```
print(shops_df["Zürich"])
```

```
2014    2409.14
2015    2941.01
2016    3496.83
2017    3119.55
Name: Zürich, dtype: float64
```

Jede einzelne der Spalten entsprach ursprünglich einer Series und entspricht auch immer noch einer Series. Dies können wir sehen, wenn wir uns den Typ anschauen:

```
print(type(shops_df["Zürich"]))
```

```
<class 'pandas.core.series.Series'>
```

Pandas bietet noch eine syntaktisch deutlich einfachere Methode, auf die Spalten zuzugreifen. Die Spaltennamen wurden dazu als Properties implementiert, und dies bedeutet, dass man einfach den Spaltennamen mittels Punkt an das Dataframe anhängen kann, um die entsprechende Spalte anzusprechen.

```
print(shops_df.Zürich)
```

```
2014    2409.14
2015    2941.01
2016    3496.83
```

```
2017    3119.55
Name: Zürich, dtype: float64
```

DataFrames aus Dictionaries

Wie schon erwähnt, hat ein DataFrame einen Spalten- und einen Zeilen-Index. Man kann es sich also wie ein Dictionary aus Series-Objekten mit Keys vorstellen.

```
cities = {"name": ["London", "Berlin", "Madrid", "Rome",
                   "Paris", "Vienna", "Bucharest", "Hamburg",
                   "Budapest", "Warsaw", "Barcelona",
                   "Munich", "Milan"],
          "population": [8615246, 3562166, 3165235, 2874038,
                         2273305, 1805681, 1803425, 1760433,
                         1754000, 1740119, 1602386, 1493900,
                         1350680],
          "country": ["England", "Germany", "Spain", "Italy",
                      "France", "Austria", "Romania",
                      "Germany", "Hungary", "Poland", "Spain",
                      "Germany", "Italy"]}

city_frame = pd.DataFrame(cities)
print(city_frame)
```

```
         name  population  country
0      London     8615246  England
1      Berlin     3562166  Germany
2      Madrid     3165235    Spain
3        Rome     2874038    Italy
4       Paris     2273305   France
5      Vienna     1805681  Austria
6   Bucharest     1803425  Romania
7     Hamburg     1760433  Germany
8    Budapest     1754000  Hungary
9      Warsaw     1740119   Poland
10  Barcelona     1602386    Spain
11     Munich     1493900  Germany
12      Milan     1350680    Italy
```

Index eines DataFrames ändern

Der Index 0,1,2,... wird automatisch dem DataFrame zugewiesen. Wir können ebenfalls einen angepassten beliebigen Index verwenden:

```
ordinals = ["first", "second", "third", "fourth",
            "fifth", "sixth", "seventh", "eigth",
            "ninth", "tenth", "eleventh", "twelvth",
            "thirteenth"]
city_frame = pd.DataFrame(cities, index=ordinals)
print(city_frame)
```

```
               name  population  country
first        London     8615246  England
```

```
second          Berlin     3562166  Germany
third           Madrid     3165235    Spain
fourth            Rome     2874038    Italy
fifth            Paris     2273305   France
sixth           Vienna     1805681  Austria
seventh      Bucharest     1803425  Romania
eigth          Hamburg     1760433  Germany
ninth         Budapest     1754000  Hungary
tenth           Warsaw     1740119   Poland
eleventh     Barcelona     1602386    Spain
twelvth         Munich     1493900  Germany
thirteenth       Milan     1350680    Italy
```

Umsortierung der Spalten Die Sortierung kann zum Zeitpunkt der Erstellung des DataFrame definiert und angepasst werden. Damit kann sichergestellt werden, dass wir eine definierte Sortierung der Spalten haben, wenn das DataFrame aus einem Dictionary erzeugt wird. Dictionaries waren bis Python 3.6 nicht geordnet, wie wir es im Kapitel zu Dictionaries gezeigt haben. Dadurch konnte man bis Python 3.6 nicht wissen, in welcher Reichenfolge die Indizes iteriert worden. Mit dem Parameter `columns` können wir jedoch eine Reihenfolge festlegen:

```
city_frame = pd.DataFrame(cities,
                          columns = ["name",
                                     "country",
                                     "population"])
print(city_frame)
```

```
         name  country  population
0      London  England     8615246
1      Berlin  Germany     3562166
2      Madrid    Spain     3165235
3        Rome    Italy     2874038
4       Paris   France     2273305
5      Vienna  Austria     1805681
6   Bucharest  Romania     1803425
7     Hamburg  Germany     1760433
8    Budapest  Hungary     1754000
9      Warsaw   Poland     1740119
10  Barcelona    Spain     1602386
11     Munich  Germany     1493900
12      Milan    Italy     1350680
```

Im Folgenden ändern wir sowohl die Spaltenreihenfolge und die Indexreihenfolge mit der Funktion `reindex`:

```
city_frame.reindex(index=[0, 2, 4, 6,  8, 10, 12, 1, 3, 5, 7, 9, 11],
                   columns=['country', 'name', 'population'])
```

```
    country       name  population
0   England     London     8615246
2     Spain     Madrid     3165235
4    France      Paris     2273305
6   Romania  Bucharest     1803425
8   Hungary   Budapest     1754000
10    Spain  Barcelona     1602386
```

```
12    Italy      Milan     1350680
1   Germany     Berlin     3562166
3     Italy       Rome     2874038
5   Austria     Vienna     1805681
7   Germany    Hamburg     1760433
9    Poland     Warsaw     1740119
11  Germany     Munich     1493900
```

Jetzt wollen wir die Spalten umbenennen. Dafür verwenden wir die DataFrame-Methode `rename`. Die Methode unterstützt folgende Konventionen:

- (index=index_mapper, columns=columns_mapper, …)
- (mapper, axis={'index', 'columns'}, …)

Wir benennen nun im folgenden Beispiel die Spalten unseres DataFrame in rumänische Bezeichnungen um. Den Parameter inplace setzen wir auf True, damit das DataFrame-Objekt direkt geändert und kein neues erzeugt wird. Der Default-Wert für den Parameter `inplace` ist `False`.

```
city_frame.rename(columns={"name":"Nume",
                           "country":"țară",
                           "population":"populație"},
                  inplace=True)
print(city_frame)
```

```
         Nume     țară  populație
0      London  England    8615246
1      Berlin  Germany    3562166
2      Madrid    Spain    3165235
3        Rome    Italy    2874038
4       Paris   France    2273305
5      Vienna  Austria    1805681
6   Bucharest  Romania    1803425
7     Hamburg  Germany    1760433
8    Budapest  Hungary    1754000
9      Warsaw   Poland    1740119
10  Barcelona    Spain    1602386
11     Munich  Germany    1493900
12      Milan    Italy    1350680
```

Bestehende Spalte als Index im DataFrame Wir wollen im nächsten Beispiel einen "nützlicheren" Index für unser Beispiel erzeugen. Dafür verwenden wir die Landesnamen als Index, d.h. die Werte aus der Liste mit dem Key `country` aus unserem `cities`-Dictionary. Wir zeigen zuerst, wie man dies direkt bei der Erzeugung des DataFrames bewerkstelligen kann:

```
city_frame = pd.DataFrame(cities,
                          columns=['name', 'population'],
                          index=cities['country'])

print(city_frame)
```

```
             name  population
England    London     8615246
Germany    Berlin     3562166
```

```
Spain       Madrid     3165235
Italy         Rome     2874038
France       Paris     2273305
Austria     Vienna     1805681
Romania  Bucharest     1803425
Germany    Hamburg     1760433
Hungary   Budapest     1754000
Poland      Warsaw     1740119
Spain    Barcelona     1602386
Germany     Munich     1493900
Italy        Milan     1350680
```

Man kann auch in einem bestehenden DataFrame den Index neu setzen. Dazu nutzen wir die Methode `set_index`, um eine Spalte in einen Index zu wandeln. Dabei ist zu beachten, dass `set_index` ein neues DataFrame zurückliefert, bei dem die gewählte Spalte als Index verwendet wird:

```
city_frame = pd.DataFrame(cities)
city_frame2 = city_frame.set_index("country")
print(city_frame2)
```

```
              name  population
country
England     London     8615246
Germany     Berlin     3562166
Spain       Madrid     3165235
Italy         Rome     2874038
France       Paris     2273305
Austria     Vienna     1805681
Romania  Bucharest     1803425
Germany    Hamburg     1760433
Hungary   Budapest     1754000
Poland      Warsaw     1740119
Spain    Barcelona     1602386
Germany     Munich     1493900
Italy        Milan     1350680
```

Im vorherigen Beispiel haben wir gesehen, dass die Methode `set_index` ein neues DataFrame-Objekt liefert und nicht das originale Objekt verändert. Möchte man kein neues DataFrame erzeugen, sondern das bestehende direkt mit einem neuen Index versehen, so kann man den Parameter “inplace” auf True setzen. Dadurch wird dann das originale Objekt direkt verändert:

```
city_frame = pd.DataFrame(cities)
city_frame.set_index("country", inplace=True)
print(city_frame)
```

```
              name  population
country
England     London     8615246
Germany     Berlin     3562166
Spain       Madrid     3165235
Italy         Rome     2874038
France       Paris     2273305
Austria     Vienna     1805681
Romania  Bucharest     1803425
```

```
Germany    Hamburg     1760433
Hungary    Budapest    1754000
Poland      Warsaw     1740119
Spain     Barcelona    1602386
Germany     Munich     1493900
Italy        Milan     1350680
```

Selektion von Zeilen

Bis jetzt haben wir die DataFrame-Objekte über die Spalten indiziert, d.h. wir haben nur auf Spalten zugegriffen. Nun möchten wir demonstrieren, wie wir auch selektiv auf die Zeilen zugreifen können. Dazu verwenden wir die Locators `loc` und `iloc`.

Im ersten Beispiel erzeugen wir ein Dataframe, das nur aus den Zeilen besteht, in denen wir den Index "Germany" haben:

```
city_frame = pd.DataFrame(cities,
                          columns=("name", "population"),
                          index=cities["country"])
print(city_frame.loc["Germany"])
```

```
           name  population
Germany   Berlin     3562166
Germany  Hamburg     1760433
Germany   Munich     1493900
```

Will man mehrere Index-Werte angeben, übergibt man diese als Liste an `loc`:

```
print(city_frame.loc[["Germany", "France"]])
```

```
           name  population
Germany   Berlin     3562166
Germany  Hamburg     1760433
Germany   Munich     1493900
France     Paris     2273305
```

Nun wählen wir alle Zeilen aus, in denen in einer Spalte eine Bedingung erfüllt ist, also in unserem Beispiel die Bevölkerungsanzahl größer als zwei Millionen ist:

```
print(city_frame.loc[city_frame.population > 2000000])
```

```
          name  population
England  London     8615246
Germany  Berlin     3562166
Spain    Madrid     3165235
Italy      Rome     2874038
France    Paris     2273305
```

Summen und kumulative Summen

Mit der Methode `sum` kann man die Summe von allen Spalten eines DataTypes berechnen, was aber in unserem Fall wenig Sinn macht:

```
print(city_frame.sum())
```

Nun berechnen wir die Summe der Bevölkerungszahlen:

```
city_frame["population"].sum()
```

```
33800614
```

Mit `cumsum` berechnen wir die kumulative Summe:

```
x = city_frame["population"].cumsum()
print(x)
```

```
England     8615246
Germany    12177412
Spain      15342647
Italy      18216685
France     20489990
Austria    22295671
Romania    24099096
Germany    25859529
Hungary    27613529
Poland     29353648
Spain      30956034
Germany    32449934
Italy      33800614
Name: population, dtype: int64
```

Spaltenwerte ersetzen

Das eben berechnete 'x' ist ein Series-Objekt mit der kumulativen Summe. Diese Series können wir der `population`-Spalte zuweisen und ersetzen damit die alten Werte. Im Folgenden nutzen wir die Methode `head`, die nur die ersten fünf Zeilen ausgibt, da dies zur Veranschaulichung des Prinzips genügt:

```
city_frame["population"] = x
print(city_frame.head())
```

```
           name  population
England  London     8615246
Germany  Berlin    12177412
Spain    Madrid    15342647
Italy      Rome    18216685
France    Paris    20489990
```

Anstelle die Werte in der `population`-Spalte komplett durch die kumulativen Summen zu ersetzen, wollen wir die neuen Werte als neue zusätzliche Spalte `cum_population` dem ursprünglchen DataFrame anfügen.

```
city_frame = pd.DataFrame(cities,
                          columns=["country",
                                   "population",
                                   "cum_population"],
                          index=cities["name"])
```

```
print(city_frame.head())
```

```
        country  population cum_population
London  England     8615246            NaN
Berlin  Germany     3562166            NaN
Madrid    Spain     3165235            NaN
Rome       Italy     2874038            NaN
Paris     France     2273305            NaN
```

Die neue Spalte `cum_population` enthält nur NaN-Werte, weil noch keine Daten zur Verfügung gestellt wurden.

Nun weisen wir die kumulativen Summen dieser neuen Spalte zu:

```
city_frame["cum_population"] = city_frame["population"].cumsum()
print(city_frame.head())
```

```
        country  population  cum_population
London  England     8615246         8615246
Berlin  Germany     3562166        12177412
Madrid    Spain     3165235        15342647
Rome       Italy     2874038        18216685
Paris     France     2273305        20489990
```

Bei der Erstellung eines DataFrame-Objektes aus einem Dictionary können auch Spalten angegeben werden, die nicht im Dictionary enthalten sind. In diesem Fall werden die Werte ebenfalls auf NaN gesetzt:

```
city_frame = pd.DataFrame(cities,
                          columns=["country",
                                   "area",
                                   "population"],
                          index=cities["name"])
print(city_frame.head())
```

```
        country area  population
London  England  NaN     8615246
Berlin  Germany  NaN     3562166
Madrid    Spain  NaN     3165235
Rome       Italy  NaN     2874038
Paris     France  NaN     2273305
```

In einem weiteren Schritt kann man dann die Werte für die Fläche in Form einer Liste bzw. eines Arrays an die Spalte `area` zuweisen.

```
# Flächen in qkm:
area = [1572, 891.85, 605.77, 1285,
        105.4, 414.6, 228, 755,
        525.2, 517, 101.9, 310.4,
        181.8]

city_frame["area"] = area
print(city_frame.head())
```

```
        country     area  population
London  England  1572.00     8615246
Berlin  Germany   891.85     3562166
Madrid    Spain   605.77     3165235
Rome      Italy  1285.00     2874038
Paris    France   105.40     2273305
```

Sortierung

DataFrames lassen sich anhand von bestimmten Kritierien sortieren. Im folgenden Beispiel sortieren wir den Inhalt des DataFrame-Objektes anhand der area-Werte in absteigender Größe:

```
city_frame = city_frame.sort_values(by="area", ascending=False)
print(city_frame)
```

```
           country     area  population
London     England  1572.00     8615246
Rome         Italy  1285.00     2874038
Berlin     Germany   891.85     3562166
Hamburg    Germany   755.00     1760433
Madrid       Spain   605.77     3165235
Budapest   Hungary   525.20     1754000
Warsaw      Poland   517.00     1740119
Vienna     Austria   414.60     1805681
Munich     Germany   310.40     1493900
Bucharest  Romania   228.00     1803425
Milan        Italy   181.80     1350680
Paris       France   105.40     2273305
Barcelona    Spain   101.90     1602386
```

Nehmen wir an, dass wir lediglich die Flächenwerte von London, Hamburg und Milan hätten. Die areas-Werte befinden sich in einem Series-Objekt mit den korrekten Indizes. Die Zuweisung funktioniert ebenfalls:

```
city_frame = pd.DataFrame(cities,
                          columns=["country",
                                   "area",
                                   "population"],
                          index=cities["name"])

some_areas = pd.Series([1572, 755, 181.8],
                       index=['London', 'Hamburg', 'Milan'])

city_frame['area'] = some_areas
print(city_frame)
```

```
           country    area  population
London     England  1572.0     8615246
Berlin     Germany     NaN     3562166
Madrid       Spain     NaN     3165235
Rome         Italy     NaN     2874038
Paris       France     NaN     2273305
Vienna     Austria     NaN     1805681
Bucharest  Romania     NaN     1803425
```

```
Hamburg    Germany   755.0     1760433
Budapest   Hungary     NaN     1754000
Warsaw      Poland     NaN     1740119
Barcelona    Spain    NaN     1602386
Munich     Germany     NaN     1493900
Milan        Italy   181.8     1350680
```

Spalten einfügen

In vorherigen Beispielen haben wir Spalten bei der Erstellung des DataFrame hinzugefügt. Es ist jedoch oft notwendig, Spalten direkt in ein bereits bestehendes DataFrame einzufügen.

```
city_frame = pd.DataFrame(cities,
                          columns = ["country",
                                     "population"],
                          index = cities["name"])


city_frame.insert(loc = 1,
                  column = 'area',
                  value = area)
print(city_frame)
```

```
          country     area  population
London     England  1572.00     8615246
Berlin     Germany   891.85     3562166
Madrid       Spain   605.77     3165235
Rome         Italy  1285.00     2874038
Paris       France   105.40     2273305
Vienna     Austria   414.60     1805681
Bucharest  Romania   228.00     1803425
Hamburg    Germany   755.00     1760433
Budapest   Hungary   525.20     1754000
Warsaw      Poland   517.00     1740119
Barcelona    Spain   101.90     1602386
Munich     Germany   310.40     1493900
Milan        Italy   181.80     1350680
```

DataFrame und verschachtelte Dictionaries

Verschachtelte Dictionaries können ebenfalls an ein DataFrame übergeben werden. Die Indizes des äußeren Dictionarys entsprechen den Spalten, und die inneren Schlüssel der Dictionaries entsprechen den Indizes der einzelnen Zeilen:

```
growth = {"Switzerland": {"2010": 3.0,
                          "2011": 1.8,
                          "2012": 1.1,
                          "2013": 1.9},
          "Germany": {"2010": 4.1,
                      "2011": 3.6,
                      "2012": 0.4,
                      "2013": 0.1},
          "France": {"2010": 2.0,
                     "2011": 2.1,
```

```
                    "2012": 0.3,
                    "2013": 0.3},
          "Greece": {"2010": -5.4,
                     "2011": -8.9,
                     "2012": -6.6,
                     "2013": -3.3},
          "Italy": {"2010": 1.7,
                    "2011": 0.6,
                    "2012": -2.3,
                    "2013": -1.9}
          }
```

```
growth_frame = pd.DataFrame(growth)
print(growth_frame)
```

```
      Switzerland  Germany  France  Greece  Italy
2010          3.0      4.1     2.0    -5.4    1.7
2011          1.8      3.6     2.1    -8.9    0.6
2012          1.1      0.4     0.3    -6.6   -2.3
2013          1.9      0.1     0.3    -3.3   -1.9
```

Sie möchten vielleicht die Jahre als Spalten und die Länder als Zeilen? Eine Vertauschung von Index und Spalten ist mittels `transpose` ganz einfach zu realisieren:

```
print(growth_frame.transpose())
```

```
             2010  2011  2012  2013
Switzerland   3.0   1.8   1.1   1.9
Germany       4.1   3.6   0.4   0.1
France        2.0   2.1   0.3   0.3
Greece       -5.4  -8.9  -6.6  -3.3
Italy         1.7   0.6  -2.3  -1.9
```

Statt `transpose()` kann man auch einfach die Property-Schreibweise `T` verwenden:

```
print(growth_frame.T)
```

```
             2010  2011  2012  2013
Switzerland   3.0   1.8   1.1   1.9
Germany       4.1   3.6   0.4   0.1
France        2.0   2.1   0.3   0.3
Greece       -5.4  -8.9  -6.6  -3.3
Italy         1.7   0.6  -2.3  -1.9
```

```
growth_frame = growth_frame.T
growth_frame2 = growth_frame.reindex(["Switzerland",
                                      "Italy",
                                      "Germany",
                                      "Greece"])
print(growth_frame2)
```

```
             2010  2011  2012  2013
Switzerland   3.0   1.8   1.1   1.9
Italy         1.7   0.6  -2.3  -1.9
```

```
Germany       4.1   3.6   0.4   0.1
Greece       -5.4  -8.9  -6.6  -3.3
```

```
import pandas as pd

cities = ["Vienna", "Vienna", "Vienna",
          "Hamburg", "Hamburg", "Hamburg",
          "Berlin", "Berlin", "Berlin",
          "Zürich", "Zürich", "Zürich"]

data = ["Austria", 414.60,    1805681,
        "Germany", 755.00,    1760433,
        "Germany", 891.85,    3562166,
        "Switzerland", 87.88, 378884]

index = [cities, ["country", "area", "population",
                  "country", "area", "population",
                  "country", "area", "population",
                  "country", "area", "population"]]

city_series = pd.Series(data, index=index)
print(city_series)
```

```
Vienna   country           Austria
         area                414.6
         population        1805681
Hamburg  country           Germany
         area                  755
         population        1760433
Berlin   country           Germany
         area                891.85
         population        3562166
Zürich   country       Switzerland
         area                87.88
         population         378884
dtype: object
```

```
city_series = city_series.sort_index()

city_series = city_series.swaplevel()
city_series.sort_index(inplace=True)
print(city_series)
```

```
area        Berlin          891.85
            Hamburg            755
            Vienna           414.6
            Zürich           87.88
country     Berlin          Germany
            Hamburg         Germany
            Vienna          Austria
            Zürich      Switzerland
population  Berlin          3562166
            Hamburg         1760433
            Vienna          1805681
            Zürich           378884
dtype: object
```

```
import pandas as pd
persons = { "Name" : ["Henry", "Sarah", "Elke",
                      "Lulu", "Vera", "Toni",
                      "Maria", "Chris"],
            "Größe" : [179, 165, 172, 154, 150,
                       189, 176, 175],
            "Gewicht" : [65, 58, 58, 45, 43, 99, 68, 60]
          }

pdf = pd.DataFrame(persons,
                   columns = ["Gewicht", "Größe"],
                   index=persons["Name"])

bmi = (pdf.Gewicht / ((pdf.Größe/100) ** 2))
bmi_okay = pdf.loc[(20 < bmi) & (bmi < 25)]
print(bmi_okay)
```

```
       Gewicht  Größe
Henry       65    179
Sarah       58    165
Maria       68    176
```

```
pdf.loc[pdf.index.str.contains("i")]
```

```
       Gewicht  Größe
Toni        99    189
Maria       68    176
Chris       60    175
```

```
pdf.insert(loc=len(pdf.columns),
           column="BMI",
           value=(pdf.Gewicht / ((pdf.Größe/100) ** 2)))
pdf
```

```
       Gewicht  Größe        BMI
Henry       65    179  20.286508
Sarah       58    165  21.303949
Elke        58    172  19.605192
Lulu        45    154  18.974532
Vera        43    150  19.111111
Toni        99    189  27.714790
Maria       68    176  21.952479
Chris       60    175  19.591837
```

```
pdf.sort_values(by="BMI", ascending=False)
```

```
       Gewicht  Größe        BMI
Toni        99    189  27.714790
Maria       68    176  21.952479
Sarah       58    165  21.303949
Henry       65    179  20.286508
Elke        58    172  19.605192
Chris       60    175  19.591837
Vera        43    150  19.111111
```

```
Lulu        45    154  18.974532
```

```
bmi_okay = (18.5 < pdf['BMI']) & (pdf['BMI'] < 23.5)
name_contains_a = pdf.index.str.contains('a')
print(pdf.loc[bmi_okay & name_contains_a])
```

```
       Gewicht  Größe        BMI
Sarah       58    165  21.303949
Vera        43    150  19.111111
Maria       68    176  21.952479
```

```
import numpy as np
import pandas as pd

names = ['Jonas', 'Leon', 'Finn', 'Guido',
         'Lara', "Hannah", "Mila", "Lina"]
index = ["Januar", "Februar", "März",
         "April", "Mai", "Juni",
         "Juli", "August", "September",
         "Oktober", "November", "Dezember"]
df = pd.DataFrame(np.random.randint(120,
                                    200,
                                    size=(len(index),
                                          len(names))),
                  columns = names,
                  index = index)
print(df)
```

```
           Jonas  Leon  ...   Mila  Lina
Januar       175   129  ...    173   190
Februar      124   124  ...    171   163
März         196   168  ...    176   192
April        172   183  ...    152   160
Mai          143   167  ...    197   154
Juni         170   159  ...    177   143
Juli         192   134  ...    127   182
August       155   138  ...    167   166
September    172   170  ...    182   136
Oktober      157   196  ...    122   163
November     188   183  ...    134   144
Dezember     193   138  ...    171   180

[12 rows x 8 columns]
```

```
new_df = df.transpose()
print(new_df)
```

```
        Januar  Februar     ...      November  Dezember
Jonas      175       124     ...           188       193
Leon       129       124     ...           183       138
Finn       188       136     ...           181       180
Guido      187       180     ...           195       130
Lara       132       171     ...           194       166
Hannah     156       157     ...           174       133
```

```
Mila       173      171    ...          134       171
Lina       190      163    ...          144       180

[8 rows x 12 columns]
```

21 Pandas Groupby

21.0.1 Pandas: groupby

In diesem Kapitel unseres Pandas- und Python-Kurses geht es um eine äußerst wichtige Funktionalität, nämlich `groupby`. Eigentlich ist dieses Thema nicht wirklich kompliziert, aber nicht immer auf den ersten Blick ersichtlich und es wird manchmal als schwierig empfunden. Völlig zu Unrecht, wie wir sehen werden. Es ist auch sehr wichtig, sich mit `groupby` vertraut zu machen, weil man damit wichtige Probleme lösen kann, die ohne `groupby` nicht oder nur sehr schwer lösbar wären. Die Pandas `groupby`-Operation geschieht in mehreren Schriten

- Aufsplitten des Objekts,
- Anwendung einer Funktion und
- Kombination der Ergebnisse.

Wir können ein DataFrame-Objekt nach verschiedenen Kriterien und zeilen- und spaltenweise, d.h. mit Hilfe des Parameters `axis`, in Gruppen aufteilen.

Durch die Anwendung einer Funktion können wir die Daten

- filtern,
- umwandeln
- aggregieren.

`groupby` kann auf Pandas Series-Objekte und DataFrame-Objekte angewendet werden! Wie das funktioniert, werden wir in diesem Tutorial anhand vieler kleiner praktischer Beispiele verstehen lernen.

groupby mit Series-Objekten

Wir erzeugen mit dem folgenden Python-Programm ein Series-Objekt mit einem Index der Größe `nvalues`. Der Index wird nicht eindeutig sein, da die Zeichenketten für den Index aus der Liste `fruits` entnommen werden, die weniger Elemente hat als 'nvalues":

```
import pandas as pd
import numpy as np
import random

nvalues = 30
# wir erstellen Zufallswerte für unser Series-Objekt:
values = np.random.randint(1, 20, (nvalues,))
fruits = ["bananas", "oranges", "apples",
          "clementines", "cherries", "pears"]
fruits_index = np.random.choice(fruits, (nvalues,))

s = pd.Series(values, index=fruits_index)
```

```
print(s[:10])
```

```
cherries       5
pears          4
cherries       3
oranges        5
clementines    1
cherries       9
oranges        5
oranges        8
pears          5
pears          9
dtype: int64
```

```
grouped = s.groupby(s.index)
grouped
```

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x7fa1c6700640>
```

Wir können sehen, dass wir ein `SeriesGroupBy`-Objekt erhalten, wenn wir `groupby` auf den Index unseres Serienobjekts `s` anwenden. Das Ergebnis dieser Operation `grouped` ist iterierbar. In jedem Schritt erhalten wir ein Tupel-Objekt zurück, das aus einem Index-Label und einem Series-Objekt besteht. Das Series-Objekt `s` ist auf dieses Label reduziert.

```
grouped = s.groupby(s.index)

for fruit, s_obj in grouped:
    print(f"===== {fruit} =====")
    print(s_obj)
```

```
===== apples =====
apples    18
apples     5
apples     3
dtype: int64
===== bananas =====
bananas    16
bananas     4
bananas     3
bananas     3
bananas     1
dtype: int64
===== cherries =====
cherries     5
cherries     3
cherries     9
cherries    12
cherries    17
cherries     2
cherries    10
dtype: int64
===== clementines =====
clementines     1
clementines     4
clementines    19
dtype: int64
```

```
===== oranges =====
oranges     5
oranges     5
oranges     8
oranges    18
oranges    18
oranges    18
dtype: int64
===== pears =====
pears     4
pears     5
pears     9
pears     2
pears    14
pears    13
dtype: int64
```

Mit folgendem Python-Code hätten wir das gleiche Ergebnis - abgesehen von der Reihenfolge - auch ohne die Verwendung von `groupby` erzielen können.

```
for fruit in set(s.index):
    print(f"===== {fruit} =====")
    print(s[fruit])
```

```
===== oranges =====
oranges     5
oranges     5
oranges     8
oranges    18
oranges    18
oranges    18
dtype: int64
===== bananas =====
bananas    16
bananas     4
bananas     3
bananas     3
bananas     1
dtype: int64
===== apples =====
apples    18
apples     5
apples     3
dtype: int64
===== pears =====
pears     4
pears     5
pears     9
pears     2
pears    14
pears    13
dtype: int64
===== cherries =====
cherries     5
cherries     3
cherries     9
cherries    12
cherries    17
```

```
cherries     2
cherries    10
dtype: int64
===== clementines =====
clementines     1
clementines     4
clementines    19
dtype: int64
```

groupby mit DataFrames

Wir werden mit einem sehr einfachen DataFrame beginnen. Der DataFrame hat zwei Spalten, von denen die eine den Namen `Name` enthält und die andere `Coffee` die Anzahl der Tassen Kaffee, die die Person getrunken hat, in ganzen Zahlen.

```
import pandas as pd
beverages = pd.DataFrame({'Name': ['Robert', 'Melinda', 'Brenda',
                                   'Samantha', 'Melinda', 'Robert',
                                   'Melinda', 'Brenda', 'Samantha'],
                          'Coffee': [3, 0, 2, 2, 0, 2, 0, 1, 3],
                          'Tea':    [0, 4, 2, 0, 3, 0, 3, 2, 0]})

beverages
```

```
       Name  Coffee  Tea
0    Robert       3    0
1   Melinda       0    4
2    Brenda       2    2
3  Samantha       2    0
4   Melinda       0    3
5    Robert       2    0
6   Melinda       0    3
7    Brenda       1    2
8  Samantha       3    0
```

Es ist einfach, und wir haben bereits in den vorherigen Kapiteln unseres Tutorials gesehen, wie man die Gesamtzahl der Kaffeetassen berechnet. Die Aufgabe besteht darin, eine Spalte eines DatFrame zu summieren, z.B. die Spalte `Coffee`:

```
beverages['Coffee'].sum()
```

```
13
```

Berechnen wir nun die Gesamtzahl der Kaffees und Tees:

```
beverages[['Coffee', 'Tea']].sum()
```

```
Coffee    13
Tea       14
dtype: int64
```

`groupby` war für die bisherigen Aufgaben nicht erforderlich. Werfen wir noch einmal einen Blick auf unseren DataFrame. Wir können sehen, dass einige der Namen mehrfach vorkommen. Es wäre also sehr interessant sein, zu sehen, wie viele Tassen Kaffee und Tee

jede Person insgesamt getrunken hat. Das heißt, wir wenden 'groupby' auf die Spalte 'Name' an. Dadurch teilen wir den DatFrame auf. Dann wenden wir 'sum' auf die Ergebnisse von 'groupby' an:

```
res = beverages.groupby(['Name']).sum()
print(res)
```

```
          Coffee  Tea
Name
Brenda         3    4
Melinda        0   10
Robert         5    0
Samantha       5    0
```

Wie wir sehen können, sind die Namen jetzt der Index des resultierenden DataFrame:

```
print(res.index)
```

```
Index(['Brenda', 'Melinda', 'Robert', 'Samantha'], dtype='object', name='Name')
```

Wir können auch die durchschnittliche Anzahl der Kaffee- und Teetassen berechnen, die die Personen getrunken haben:

```
beverages.groupby(['Name']).mean()
```

```
          Coffee       Tea
Name
Brenda       1.5  2.000000
Melinda      0.0  3.333333
Robert       2.5  0.000000
Samantha     2.5  0.000000
```

Weiteres groupby-Beispiel Der folgende Python-Code wird verwendet, um die Daten zu erzeugen, die wir in unserem nächsten `groupby`-Beispiel verwenden werden. Es ist nicht notwendig, den folgenden Python-Code für den nachfolgenden Inhalt zu verstehen. Das Modul `faker` muss installiert sein. Im Falle einer Anaconda-Installation kann dies durch die Ausführung eines der folgenden Befehle in einer Shell geschehen:

```
conda install -c conda-forge faker
conda install -c conda-forge/label/gcc7 faker
conda install -c conda-forge/label/cf201901 faker
conda install -c conda-forge/label/cf202003 faker
```

```
from faker import Faker
import numpy as np
from itertools import chain

fake = Faker('de_DE')

number_of_names = 10
names = []
for _ in range(number_of_names):
```

```
    names.append(fake.first_name())


data = {}
workweek = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
weekend = ("Saturday", "Sunday")

for day in chain(workweek, weekend):
    data[day] = np.random.randint(0, 10, (number_of_names,))

data_df = pd.DataFrame(data, index=names)
data_df
```

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Victor	1	2	4	4	0	1	1
Daria	5	2	2	6	4	7	9
Ignaz	2	3	1	2	2	5	7
Marco	6	4	5	9	1	5	7
Bertha	8	6	3	4	4	4	5
Nadeschda	1	4	4	5	1	4	7
Alberto	5	5	0	3	6	2	3
Sylke	4	1	5	1	3	6	0
Marion	0	0	1	3	1	1	7
Ayse	0	2	7	6	3	0	8

```
print(names)
```

```
['Victor', 'Daria', 'Ignaz', 'Marco', 'Bertha', 'Nadeschda', 'Alberto', 'Sylke',
↪  'Marion', 'Ayse']
```

```
names = ('Ortwin', 'Mara', 'Siegrun', 'Sylvester', 'Metin', 'Adeline', 'Utz',
↪  'Susan', 'Gisbert', 'Senol')
data = {'Monday': np.array([0, 9, 2, 3, 7, 3, 9, 2, 4, 9]),
        'Tuesday': np.array([2, 6, 3, 3, 5, 5, 7, 7, 1, 0]),
        'Wednesday': np.array([6, 1, 1, 9, 4, 0, 8, 6, 8, 8]),
        'Thursday': np.array([1, 8, 6, 9, 9, 4, 1, 7, 3, 2]),
        'Friday': np.array([3, 5, 6, 6, 5, 2, 2, 4, 6, 5]),
        'Saturday': np.array([8, 4, 8, 2, 3, 9, 3, 4, 9, 7]),
        'Sunday': np.array([0, 8, 7, 8, 9, 7, 2, 0, 5, 2])}

data_df = pd.DataFrame(data, index=names)
data_df
```

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Ortwin	0	2	6	1	3	8	0
Mara	9	6	1	8	5	4	8
Siegrun	2	3	1	6	6	8	7
Sylvester	3	3	9	9	6	2	8
Metin	7	5	4	9	5	3	9
Adeline	3	5	0	4	2	9	7
Utz	9	7	8	1	2	3	2
Susan	2	7	6	7	4	4	0
Gisbert	4	1	8	3	6	9	5
Senol	9	0	8	2	5	7	2

Mit diesem DataFrame wollen wir demonstrieren, wie man Spalten mittels einer Funktion

kombinieren kann. Es geht darum festzustellen wieviele Stunden jede Person während der Arbeits- und während des Wochenendes gearbeitet hatte:

```
def is_weekend(day):
    if day in {'Saturday', 'Sunday'}:
        return "Weekend"
    else:
        return "Workday"
```

```
for res_func, df in data_df.groupby(by=is_weekend, axis=1):
    print(df)
```

```
          Saturday  Sunday
Ortwin           8       0
Mara             4       8
Siegrun          8       7
Sylvester        2       8
Metin            3       9
Adeline          9       7
Utz              3       2
Susan            4       0
Gisbert          9       5
Senol            7       2
          Monday  Tuesday  Wednesday  Thursday  Friday
Ortwin         0        2          6         1       3
Mara           9        6          1         8       5
Siegrun        2        3          1         6       6
Sylvester      3        3          9         9       6
Metin          7        5          4         9       5
Adeline        3        5          0         4       2
Utz            9        7          8         1       2
Susan          2        7          6         7       4
Gisbert        4        1          8         3       6
Senol          9        0          8         2       5
```

```
data_df.groupby(by=is_weekend, axis=1).sum()
```

```
          Weekend  Workday
Ortwin          8       12
Mara           12       29
Siegrun        15       18
Sylvester      10       30
Metin          12       30
Adeline        16       14
Utz             5       27
Susan           4       26
Gisbert        14       22
Senol           9       24
```

Aufgaben / Übungen

Aufgabe 1 Berechnen Sie die Durchschnittspreise für die Produkte des folgenden DataFrame:

```
import pandas as pd

d = {"products": ["Oppilume", "Dreaker", "Lotadilo",
                  "Crosteron", "Wazzasoft", "Oppilume",
                  "Dreaker", "Lotadilo", "Wazzasoft"],
     "colours": ["blue", "blue", "blue",
                 "green", "blue", "green",
                 "green", "green", "red"],
     "customer_price": [2345.89, 2390.50, 1820.00,
                        3100.00, 1784.50, 2545.89,
                        2590.50, 2220.00, 2084.50],
     "non_customer_price": [2445.89, 2495.50, 1980.00,
                            3400.00, 1921.00, 2645.89,
                            2655.50, 2140.00, 2190.00]}

product_prices = pd.DataFrame(d)
product_prices
```

	products	colours	customer_price	non_customer_price
0	Oppilume	blue	2345.89	2445.89
1	Dreaker	blue	2390.50	2495.50
2	Lotadilo	blue	1820.00	1980.00
3	Crosteron	green	3100.00	3400.00
4	Wazzasoft	blue	1784.50	1921.00
5	Oppilume	green	2545.89	2645.89
6	Dreaker	green	2590.50	2655.50
7	Lotadilo	green	2220.00	2140.00
8	Wazzasoft	red	2084.50	2190.00

Aufgabe 2 Berechnen Sie die Summe der Preise nach den Farben.

Aufgabe 3 Lesen Sie die Datei project_times.txt ein. Die Zeilen dieser Datei enthalten kommagetrennt das Datum, den Namen des Programmierers, den Namen des Projekts und die Zeit, die die Personen für das Projekt aufgewendet haben.

Berechnen Sie die Zeit, die für alle Projekte pro Tag aufgewendet wurde

Aufgabe 4 Erstellen Sie einen DateFrame mit der Gesamtzeit, die alle Programmiererinnen und Programmierer pro Tag für ein Projekt aufwenden

Aufgabe 5 Berechnen Sie die Gesamtzeit, die für die Projekte über den ganzen Monat hinweg aufgewendet wurde.

Aufgabe 6 Berechnen Sie die monatlichen Zeiten der einzelnen Programmierer und Programmiererinnen unabhängig von den Projekten

Aufgabe 7 Ordnen Sie den DataFrame mit einem MultiIndex, der aus dem Datum und den Projektnamen besteht, neu an. Für jede Person soll es eine Spalte mit den aufgewendeten Zeiten geben. Also so:

```
                   time
programmer         Antonie  Elise  Fatima  Hella  Mariola
date       project
2020-01-01 BIRDY   NaN      NaN    NaN     1.50   1.75
           NSTAT   NaN      NaN    0.25    NaN    1.25
           XTOR    NaN      NaN    NaN     1.00   3.50
2020-01-02 BIRDY   NaN      NaN    NaN     1.75   2.00
           NSTAT   0.5      NaN    NaN     NaN    1.75
```

Zusatzaufgabe: ersetzte NaN durch 0.

Aufgabe 8: Die Datei donations.txt enthält die folgenden Daten, also "Vorname" (firstname), "Nachname" (surname), "Stadt" (city), "Job", "Einkommen" (income), "Spenden" (donations):

```
firstname,surname,city,job,income,donations
Janett,Schwital,Karlsruhe,Politician,244400,2512
Daniele,Segebahn,Freiburg,Student,16800,336
Kirstin,Klapp,Hamburg,Engineer,116900,1479
Oswald,Segebahn,Köln,Musician,57700,1142
```

Gruppiere die Daten anhand des Jobs der Personen.

Lösungen

Solution to Exercise 1

```
x = product_prices.groupby("products").mean()
x
```

```
           customer_price  non_customer_price
products
Crosteron         3100.00             3400.00
Dreaker           2490.50             2575.50
Lotadilo          2020.00             2060.00
Oppilume          2445.89             2545.89
Wazzasoft         1934.50             2055.50
```

Lösung für Aufgabe 2

```
x = product_prices.groupby("colours").sum()
x
```

```
          customer_price  non_customer_price
colours
blue            8340.89             8842.39
green          10456.39            10841.39
red             2084.50             2190.00
```

Lösung für Aufgabe 3

```
import pandas as pd

df = pd.read_csv("data1/project_times.txt", index_col=0)
df
```

```
           programmer project  time
date
2020-01-01      Hella    XTOR  1.00
2020-01-01      Hella   BIRDY  1.50
2020-01-01     Fatima   NSTAT  0.25
2020-01-01    Mariola   NSTAT  0.50
2020-01-01    Mariola   BIRDY  1.75
...               ...     ...   ...
2030-01-30    Antonie    XTOR  0.50
2030-01-31      Hella   BIRDY  1.25
2030-01-31      Hella   BIRDY  1.75
2030-01-31    Mariola   BIRDY  1.00
2030-01-31      Hella   BIRDY  1.00

[17492 rows x 3 columns]
```

```
times_per_day = df.groupby(df.index).sum()
print(times_per_day[:10])
```

```
             time
date
2020-01-01   9.25
2020-01-02   6.00
2020-01-03   2.50
2020-01-06   5.75
2020-01-07  15.00
2020-01-08  13.25
2020-01-09  10.25
2020-01-10  17.00
2020-01-13   4.75
2020-01-14  10.00
```

Lösung für Aufgabe 4

```
times_per_day_project = df.groupby([df.index, 'project']).sum()
print(times_per_day_project[:10])
```

```
                    time
date       project
2020-01-01 BIRDY    3.25
           NSTAT    1.50
           XTOR     4.50
2020-01-02 BIRDY    3.75
           NSTAT    2.25
2020-01-03 BIRDY    1.00
           NSTAT    0.25
           XTOR     1.25
2020-01-06 BIRDY    2.75
```

```
NSTAT    0.75
```

Lösung für Aufgabe 5

```
df.groupby(['project']).sum()
```

```
           time
project
BIRDY    9605.75
NSTAT    8707.75
XTOR     6427.50
```

Lösung für Aufgabe 6

```
df.groupby(['programmer']).sum()
```

```
               time
programmer
Antonie      1511.25
Elise          80.00
Fatima        593.00
Hella       10642.00
Mariola     11914.75
```

Lösung für Aufgabe 7

```
x = df.groupby([df.index, 'project', 'programmer']).sum()

x = x.unstack()
x
```

```
                    time
programmer         Antonie Elise Fatima Hella Mariola
date       project
2020-01-01 BIRDY       NaN   NaN    NaN  1.50    1.75
           NSTAT       NaN   NaN   0.25   NaN    1.25
           XTOR        NaN   NaN    NaN  1.00    3.50
2020-01-02 BIRDY       NaN   NaN    NaN  1.75    2.00
           NSTAT       0.5   NaN    NaN   NaN    1.75
...                    ...   ...    ...   ...     ...
2030-01-29 XTOR        NaN   NaN    NaN  1.00    5.50
2030-01-30 BIRDY       NaN   NaN    NaN  0.75    4.75
           NSTAT       NaN   NaN    NaN  3.75     NaN
           XTOR        0.5   NaN    NaN  0.75     NaN
2030-01-31 BIRDY       NaN   NaN    NaN  4.00    1.00

[7037 rows x 5 columns]
```

```
x = x.fillna(0)
print(x[:10])
```

```
                    time
programmer         Antonie Elise Fatima Hella Mariola
```

```
date       project
2020-01-01 BIRDY      0.00   0.0   0.00  1.50    1.75
           NSTAT      0.00   0.0   0.25  0.00    1.25
           XTOR       0.00   0.0   0.00  1.00    3.50
2020-01-02 BIRDY      0.00   0.0   0.00  1.75    2.00
           NSTAT      0.50   0.0   0.00  0.00    1.75
2020-01-03 BIRDY      0.00   0.0   1.00  0.00    0.00
           NSTAT      0.25   0.0   0.00  0.00    0.00
           XTOR       0.00   0.0   0.00  0.50    0.75
2020-01-06 BIRDY      0.00   0.0   0.00  2.50    0.25
           NSTAT      0.00   0.0   0.00  0.00    0.75
```

Lösung für Aufgabe 8:

```
import pandas as pd

data = pd.read_csv('data1/donations.txt')
data_sum = data.groupby(['job']).sum()
data_sum.sort_values(by='donations')
```

```
             income  donations
job
Student       372900       7458
Musician     1448700      24376
Engineer     2067200      25564
Politician   4118300      30758
Manager     12862600      87475
```

```
data_sum['relative'] = data_sum.donations * 100 / data_sum.income

data_sum.sort_values(by='relative')
```

```
             income  donations  relative
job
Manager     12862600      87475  0.680072
Politician   4118300      30758  0.746862
Engineer     2067200      25564  1.236649
Musician     1448700      24376  1.682612
Student       372900       7458  2.000000
```

22 Pandas Groupby Beispiele

22.0.1 Pandas: Groupby Beispiele

In diesem Teil unseres Python- und Pandas-Kurses möchten wir ein ausführliches Beispiel für die Verwendung von groupby geben. Wir werden eine Datendatei [donations.txt](https://python-course.eu/data/donations.txt) verwenden.

Die ersten Zeilen dieser Datendatei sehen wie folgt aus:

```
firstname,surname,city,job,income,donations
Janett,Schwital,Karlsruhe,Politician,244400,2512
Daniele,Segebahn,Freiburg,Student,16800,336
Kirstin,Klapp,Hamburg,Engineer,116900,1479
Oswald,Segebahn,Köln,Musician,57700,1142
Heinz-Joachim,Wagner,Stuttgart,Engineer,109300,1592
```

Die Daten dieser Datei wurden mit Hilfe des Python-Faker-Moduls künstlich erzeugt, eine mögliche Übereinstimmung mit real existierenden Personen ist also rein zufällig und nicht beabsichtigt. Jede Zeile enthält den Vor- und Nachnamen einer fiktiven Person sowie den Wohnort, die berufliche Position (einer der fünf Berufe Politiker, Student, Ingenieur, Musiker und Manager), das frühe Einkommen und die Summe der Spenden pro Jahr.

Zunächst lesen wir die Datendatei ein, die trotz der Endung 'txt' natürlich eine csv-Datei ist:

```
import pandas as pd

fname = 'data1/donations.txt'
data = pd.read_csv(fname, usecols=[2, 3, 4, 5])
data[:10]
```

```
        city         job  income  donations
0  Karlsruhe  Politician  244400       2512
1   Freiburg     Student   16800        336
2    Hamburg    Engineer  116900       1479
3       Köln    Musician   57700       1142
4  Stuttgart    Engineer  109300       1592
5    Hamburg     Student   12500        250
6   Freiburg    Engineer  128700       1984
7  Stuttgart  Politician  161300        822
8   Freiburg    Engineer  129000       2159
9  Stuttgart    Musician  108800       1516
```

Schauen wir uns zunächst an, wie viel jede Berufsgruppe insgesamt verdient und gespendet hat. Dazu können wir die Pandas-Funktion `groupby` zusammen mit `sum` verwenden:

```
data_sum = data.groupby(['job']).sum()
```

```
data_sum
```

```
              income  donations
job
Engineer     2067200      25564
Manager     12862600      87475
Musician     1448700      24376
Politician   4118300      30758
Student       372900       7458
```

Durch die Verwendung von “sort_values” für die Spalte “donations” können wir diesen DataFrame sortieren:

```
data_sum.sort_values(by='donations')
```

```
              income  donations
job
Student       372900       7458
Musician     1448700      24376
Engineer     2067200      25564
Politician   4118300      30758
Manager     12862600      87475
```

Schauen wir uns die folgenden drei Personen aus unserem Datei an:

```
Janett,Schwital,Karlsruhe,Politician,244400,2512
Daniele,Segebahn,Freiburg,Student,16800,336
Kirstin,Klapp,Hamburg,Engineer,116900,1479
```

Wer ist der großzügigste von ihnen? Die Politikerin Janett Schwital? 2512 Euro sind doch fast achtmal so viel wie das, was die Studentin Daniele Segebahn gespendet hat, und fast zweimal so viel wie die Ingenieurin Kirstin Klapp gespendet hat.

Die meisten Menschen würden denken, dass wir die Spenden in Relation zu ihren Einkommen sehen müssen. Dann erhalten wir folgendes:

Der Politikerin hat etwa 1 % ihres Einkommens gespendet, die Ingenieurin etwa 1,3 % und die Studentin 2 %. Wir können also sagen, dass Daniele im Verhältnis zu ihrem Einkommen am großzügigsten ist!

Machen wir das Gleiche mit dem vorherigen DataFrame:

```
data_sum['relative'] = data_sum.donations * 100 / data_sum.income
data_sum.sort_values(by='relative')
```

```
              income  donations  relative
job
Manager     12862600      87475  0.680072
Politician   4118300      30758  0.746862
Engineer     2067200      25564  1.236649
Musician     1448700      24376  1.682612
Student       372900       7458  2.000000
```

Wir sehen also, dass die Manager und Managerinnen und die Politikerinnen und Politiker sehr knauserig sind und die Studierenden extrem großzügig sind. Können wir dies verallgemeinern,

indem wir sagen: Je weniger Leute haben, desto großzügiger sind sie. Jedoch darf man nicht vergessen, dass der ganze Datensatz gefälscht ist!

Wir wollen nun die Anzahl der Personen in jeder Berufsgruppe für jede Stadt zählen. Die Verwendung von “count” und “groupby” in der folgenden Weise ist hilfreich, aber nicht schön. Die Zahlen in den Spalten “Einkommen” und “Spenden” zeigen die gewünschten Zahlen, aber die Spaltennamen sind irreführend und wir brauchen die Ergebnisse nicht doppelt:

```
x = data.groupby(['city', 'job']).count()
x
```

		income	donations
city	job		
Berlin	Engineer	3	3
	Manager	3	3
	Musician	2	2
	Politician	1	1
	Student	2	2
Freiburg	Engineer	3	3
	Manager	5	5
	Musician	3	3
	Politician	3	3
	Student	5	5
Hamburg	Engineer	4	4
	Musician	4	4
	Politician	1	1
	Student	2	2
Karlsruhe	Engineer	1	1
	Manager	3	3
	Politician	7	7
	Student	2	2
Konstanz	Engineer	2	2
	Manager	5	5
	Musician	3	3
	Politician	4	4
	Student	3	3
Köln	Engineer	1	1
	Manager	3	3
	Musician	2	2
	Politician	1	1
	Student	3	3
Stuttgart	Engineer	4	4
	Manager	4	4
	Musician	4	4
	Politician	3	3
	Student	4	4

Wir könnten das Folgende tun, um ein gutes Ergebnis zu erzielen, aber der ganze Weg ist ungeschickt:

```
people_job_city = data.groupby(['city', 'job']).count()
people_job_city.drop(columns=['income'], inplace=True)
people_job_city.rename(columns={'donations': 'number of people'})
```

		number of people

```
city       job
Berlin     Engineer                3
           Manager                 3
           Musician                2
           Politician              1
           Student                 2
Freiburg   Engineer                3
           Manager                 5
           Musician                3
           Politician              3
           Student                 5
Hamburg    Engineer                4
           Musician                4
           Politician              1
           Student                 2
Karlsruhe  Engineer                1
           Manager                 3
           Politician              7
           Student                 2
Konstanz   Engineer                2
           Manager                 5
           Musician                3
           Politician              4
           Student                 3
Köln       Engineer                1
           Manager                 3
           Musician                2
           Politician              1
           Student                 3
Stuttgart  Engineer                4
           Manager                 4
           Musician                4
           Politician              3
           Student                 4
```

Der bessere Weg besteht darin, `groupby` von Pandas zu benutzen. In der folgenden Lösung gruppiert die `groupby`-Methode den DataFrame nach den angegebenen Spalten (in diesem Fall "city" und "job") und erstellt für jede Gruppe einen neuen DataFrame. Die `size`-Methode gibt die Anzahl jeder Gruppe zurück, was zu einem neuen DataFrame mit den Anzahlen für jede Kombination von Stadt und Stelle führt. Beachten Sie, dass das Ergebnis ein Series-Objekt ist:

```
result = data[['city', 'job']].groupby(['city', 'job']).size()
result
```

```
city       job
Berlin     Engineer      3
           Manager       3
           Musician      2
           Politician    1
           Student       2
Freiburg   Engineer      3
           Manager       5
           Musician      3
           Politician    3
           Student       5
Hamburg    Engineer      4
```

```
           Musician      4
           Politician    1
           Student       2
Karlsruhe  Engineer      1
           Manager       3
           Politician    7
           Student       2
Konstanz   Engineer      2
           Manager       5
           Musician      3
           Politician    4
           Student       3
Köln       Engineer      1
           Manager       3
           Musician      2
           Politician    1
           Student       3
Stuttgart  Engineer      4
           Manager       4
           Musician      4
           Politician    3
           Student       4
dtype: int64
```

Interessanter als das vorherige Ergebnis ist die Anzahl der Spenden und insbesondere die Spenden im Verhältnis zum Einkommen für jeden Job und jede Stadt:

```
city_job_data = data.groupby(['city', 'job']).sum()
city_job_data[:12]  # the first 12 lines
```

```
                    income  donations
city     job
Berlin   Engineer     334300       3732
         Manager     1916000      16290
         Musician     135500       2644
         Politician   246700       1103
         Student       31200        624
Freiburg Engineer     358400       5431
         Manager     2423700      17811
         Musician     273800       4640
         Politician   489300       4352
         Student       87500       1750
Hamburg  Engineer     448400       5390
         Musician     358800       5359
```

Auch hier ziehen wir es vor, die Spenden im Verhältnis zu den Einnahmen zu sehen:

```
city_job_data['rel_donations'] = (city_job_data['donations'] * 100 /
↪  city_job_data['income']).round(2)
city_job_data.drop(['income', 'donations'], axis=1, inplace=True)
city_job_data[:12]
```

```
                    rel_donations
city     job
Berlin   Engineer             1.12
         Manager              0.85
         Musician             1.95
```

```
          Politician           0.45
          Student              2.00
Freiburg  Engineer             1.52
          Manager              0.73
          Musician             1.69
          Politician           0.89
          Student              2.00
Hamburg   Engineer             1.20
          Musician             1.49
```

Vielleicht wollen Sie jetzt wissen, in welcher Stadt die großzügigsten Menschen leben?

```
cities_donations = data.groupby(['city']).sum()
cities_donations['relative'] = cities_donations['donations'] * 100 /
↪  cities_donations['income']
cities_donations.sort_values(by='relative')
```

```
            income  donations  relative
city
Karlsruhe  3457600      22265  0.643944
Konstanz   4165500      29386  0.705462
Köln       2369500      21158  0.892931
Stuttgart  3522900      31891  0.905249
Berlin     2663700      24393  0.915756
Freiburg   3632700      33984  0.935503
Hamburg    1057800      12554  1.186803
```

23 Pandas Data Files

23.0.1 Dateien lesen und schreiben

All die starken Daten-Strukturen wie Series und DataFrames würden fast nichts nützen, wenn das Pandas-Modul keine Funktionalitäten unterstützen würde, um Daten einzulesen und rauszuschreiben. Dabei geht es nicht um die einfache Möglichkeit mit Dateien umzugehen. Damit der Nutzen für Data-Scientists sichtbar wird, müssen die wichtigsten Daten-Formate unterstützt werden, wie z.B.:

- Trenner-Separierte Dateien, z.B. csv
- Microsoft Excel Dateien
- HTML
- XML
- JSON

Trennerseparierte Werte

Die meisten Menschen verwenden den Namen "CSV-Datei" als Synonym für eine trennerseparierte-Datei. Sie beachten nicht die Tatsache, das CSV ein Akronym ist für "comma separated values" (also in Deutsch "kommaseparierte-Liste"), was in den meisten Situationen nicht der Fall ist. Pandas verwendet "csv" ebenfalls in Zusammenhängen, in denen "dsv" die passendere Bezeichnung wäre.

Trennerseparierte Werte (Delimiter-separated values - DSV) sind definiert und abgelegt in zweidimensionalen Arrays, bei denen die Werte mit zweckmäßig definierten Trennzeichen in jeder Zeile getrennt sind. Diese Arte und Weise wird oft in Kombination mit Tabellenprogrammen eingesetzt, die Daten als DSV ein- und auslesen können. Auch wird die Implementierung in allgemeinen Datenaustauschformaten verwendet.

Bei der Datei dollar_euro.txt handelt es sich um eine DSV-Datei, die Tabulatoren (\t) als Trennzeichen benutzt.

CSV- und DSV-Dateien lesen

Pandas bietet zwei Wege, um CSV/DSV Dateien zu lesen. Das bedeutet konkret:

- DataFrame.from_csv
- read_csv

Es gibt zwischen beiden Methoden keinen großen Unterschied, d.h. es gibt in manchen Fällen verschiedene Default-Werte, und `read_csv` hat mehr Parameter. Wir konzentrieren uns auf `read_csv`, weil `DataFrame.from_csv` nur wegen Auf- und Abwärtskompatibilität innerhalb von Pandas gehalten wird.

```
import pandas as pd
```

```
exchange_rates = pd.read_csv("data1/dollar_euro.txt",
                             sep="\t")
print(exchange_rates)
```

```
    Year   Average  Min USD/EUR  Max USD/EUR  Working days
0   2016  0.901696     0.864379     0.959785           247
1   2015  0.901896     0.830358     0.947688           256
2   2014  0.753941     0.716692     0.823655           255
3   2013  0.753234     0.723903     0.783208           255
4   2012  0.778848     0.743273     0.827198           256
5   2011  0.719219     0.671953     0.775855           257
6   2010  0.755883     0.686672     0.837381           258
7   2009  0.718968     0.661376     0.796495           256
8   2008  0.683499     0.625391     0.802568           256
9   2007  0.730754     0.672314     0.775615           255
10  2006  0.797153     0.750131     0.845594           255
11  2005  0.805097     0.740357     0.857118           257
12  2004  0.804828     0.733514     0.847314           259
13  2003  0.885766     0.791766     0.963670           255
14  2002  1.060945     0.953562     1.165773           255
15  2001  1.117587     1.047669     1.192748           255
16  2000  1.085899     0.962649     1.211827           255
17  1999  0.939475     0.848176     0.998502           261
```

Wie wir gesehen haben, benutzt `read_csv` automatisch die erste Zeile als Überschriften bzw. Spaltennamen für die Spalten. Wir können den Spalten auch beliebige andere Namen geben. Dazu muss die erste Zeile übersprungen werden, was wir dadurch erreichen, dass wir den Parameter `header` auf `0` setzen, und eine Liste mit Spalten-Namen an den Parameter `names` zuweisen:

```
import pandas as pd

exchange_rates = pd.read_csv("data1/dollar_euro.txt",
                             sep="\t",
                             header=0,
                             names=["year", "min", "max", "days"])
print(exchange_rates.head())
```

```
          year       min       max  days
2016  0.901696  0.864379  0.959785   247
2015  0.901896  0.830358  0.947688   256
2014  0.753941  0.716692  0.823655   255
2013  0.753234  0.723903  0.783208   255
2012  0.778848  0.743273  0.827198   256
```

Schreiben von CSV-Dateien

CSV-Dateien können wir mit der Methode `to_csv` schreiben. Wir werden dies an einem Beispiel demonstrieren. Zuerst erzeugen wir jedoch Daten, die wir dann rausschreiben werden. Im Verzeichnis `data1` liegen die beiden Dateien countries_male_population.csv und countries_female_population.csv, die entsprechend die Zahlen der männlichen und weiblichen Bevölkerungs von Ländern enthalten.

```
column_names = ["Country"] + list(range(2003, 2013))
```

```
male_pop = pd.read_csv("data1/countries_male_population.csv",
                       header=None,
                       index_col=0,
                       names=column_names)

female_pop = pd.read_csv("data1/countries_female_population.csv",
                         header=None,
                         index_col=0,
                         names=column_names)


population = male_pop + female_pop
population.head()
```

```
                  2003      2004    ...         2011      2012
Country                             ...
Australia     19872646  20091504    ...     22620554  22683573
Austria        8067289   8140122    ...      8404252   8443018
Belgium       10355844  10396421    ...     10366843  11035958
Canada        31361611  31372587    ...     33927935  34492645
Czech Republic 10203269 10211455    ...     10532770  10505445

[5 rows x 10 columns]
```

In der Datei `countries_total_population.csv` im Verzeichnis `data1` speichern wir die eben erzeugte DataFrame `population`:

```
population.to_csv("data1/countries_total_population.csv")
```

Wir möchten nun ein DataFrame bzw. eine Datei erzeugen, die alle Informationen enthalten soll, also sowohl die weibliche und die männliche Bevölkerung als auch die Gesamtbevölkerung. Dazu konkatenieren wir die drei DataFrames:

```
pop_complete = pd.concat([population,
                          male_pop,
                          female_pop],
                          keys=["total", "male", "female"])
```

Um das Ergebnis der vorigen Konkatenation besser zu verstehen, geben wir im Folgenden nur die interessanten Indizes aus:

```
pop_complete.iloc[[0, 1, 2, 29, 30, 31, 32, 59, 60, 61, 62]]
```

```
                        2003       2004    ...         2011       2012
       Country                              ...
total  Australia      19872646   20091504    ...     22620554   22683573
       Austria         8067289    8140122    ...      8404252    8443018
       Belgium        10355844   10396421    ...     10366843   11035958
       United States 288774226  290810719    ...    309989078  312232049
male   Australia      9873447    9990513     ...     11260747   11280804
       Austria        3909120    3949825     ...      4095337    4118035
       Belgium        5066885    5087176     ...      5370234    5413801
       United States 141957038  143037260    ...    152449134  153596908
female Australia      9999199    10100991    ...     11359807   11402769
```

```
       Austria          4158169    4190297    ...        4308915    4324983
       Belgium          5288959    5309245    ...        4996609    5622157

[11 rows x 10 columns]
```

Wir wollen nun den hierarchischen Index umdrehen, sodass man für jedes Land direkt alle Bevölkerungsinformationen im Blick hat:

```
df = pop_complete.swaplevel()
df.sort_index(inplace=True)
df.head(12)
```

```
                      2003      2004    ...         2011      2012
Country                                 ...
Australia female   9999199  10100991    ...     11359807  11402769
          male     9873447   9990513    ...     11260747  11280804
          total   19872646  20091504    ...     22620554  22683573
Austria   female   4158169   4190297    ...      4308915   4324983
          male     3909120   3949825    ...      4095337   4118035
          total    8067289   8140122    ...      8404252   8443018
Belgium   female   5288959   5309245    ...      4996609   5622157
          male     5066885   5087176    ...      5370234   5413801
          total   10355844  10396421    ...     10366843  11035958
Canada    female  15829173  15834015    ...     17101813  17379104
          male    15532438  15538572    ...     16826122  17113541
          total   31361611  31372587    ...     33927935  34492645

[12 rows x 10 columns]
```

```
df.to_csv("data1/countries_total_population.csv")
```

Lesen und Schreiben von Excel-Dateien

Es ist auch möglich, Microsoft-Excel-Dateien zu lesen und zu schreiben. Um diese Funktionalitäten bereitzustellen, benutzt Pandas die Module `xlrd` und `openpyxl`. Diese Module werden automatisch von Pandas installiert, sodass man sie nicht extra installieren muss.

Wir werden ein einfaches Excel-Dokument benutzen, um Lesemöglichkeiten von Pandas zu demonstrieren. Das Dokument sales.xls enthält zwei Blätter (englisch “sheet”), das eine mit dem Namen ‘week1’ und das andere ‘week2’. Eine Excel-Datei lässt sich mit der Funktion “read_excel” einlesen. Wir zeigen dies mit dem folgenden Python-Programm:

```
excel_file = pd.ExcelFile("data1/sales.xls")

sheet = pd.read_excel(excel_file)
sheet
```

```
     Weekday          Sales
0     Monday  123432.980000
1    Tuesday  122198.650200
2  Wednesday  134418.515220
3   Thursday  131730.144916
4     Friday  128173.431003
```

Von den beiden Blättern der Datei "sales.xls" haben wir nur eine mit `read_excel` eingelesen. Eine Excel-Datei, die aus zahlreichen Blättern bestehen kann, kann mit allen Blättern wie folgt eingelesen werden:

```
document = {}
excel_file = pd.ExcelFile("data1/sales.xls")
for sheet_name in excel_file.sheet_names:
    document[sheet_name] = excel_file.parse(sheet_name)

for sheet_name in document:
    print("\n" + sheet_name + ":\n", document[sheet_name])
```

```
week1:
      Weekday          Sales
0      Monday  123432.980000
1     Tuesday  122198.650200
2   Wednesday  134418.515220
3    Thursday  131730.144916
4      Friday  128173.431003

week2:
      Weekday          Sales
0      Monday  223277.980000
1     Tuesday  234441.879000
2   Wednesday  246163.972950
3    Thursday  241240.693491
4      Friday  230143.621590
```

```
#prog4book
pop = pd.read_csv("data1/countries_population.csv",
                  header=None,
                  names=["Country", "Population"],
                  index_col=0,
                  quotechar="'",
                  sep=" ",
                  thousands=",")
print(pop.head(5))
```

```
               Population
Country
China          1355692576
India          1236344631
European Union   511434812
United States    318892103
Indonesia        253609643
```

```
#prog4book
lands = pd.read_csv('data1/bundeslaender.txt', sep=" ")
print(lands.columns.values)
```

```
['land' 'area' 'male' 'female']
```

```
#prog4book
# swap the columns of our DataFrame:
```

```
lands = lands.reindex(columns=['land', 'area', 'female', 'male'])
lands[:2]
```

```
               land      area  female  male
0  Baden-Württemberg  35751.65    5465  5271
1             Bayern  70551.57    6366  6103
```

```
#prog4book
lands.insert(loc=len(lands.columns),
             column='population',
             value=lands['female'] + lands['male'])
lands[:3]
```

```
               land      area  female  male  population
0  Baden-Württemberg  35751.65    5465  5271       10736
1             Bayern  70551.57    6366  6103       12469
2             Berlin    891.85    1736  1660        3396
```

```
#prog4book
lands.insert(loc=len(lands.columns),
             column='density',
             value=(lands['population'] * 1000 / lands['area']).round(0))

lands[:4]
```

```
               land      area   ...     population  density
0  Baden-Württemberg  35751.65   ...          10736    300.0
1             Bayern  70551.57   ...          12469    177.0
2             Berlin    891.85   ...           3396   3808.0
3        Brandenburg  29478.61   ...           2560     87.0

[4 rows x 6 columns]
```

```
#prog4book
print(lands.loc[(lands.area>30000) & (lands.population>10000)])
```

```
                 land      area   ...     population  density
0    Baden-Württemberg  35751.65   ...          10736    300.0
1               Bayern  70551.57   ...          12469    177.0
9  Nordrhein-Westfalen  34085.29   ...          18058    530.0

[3 rows x 6 columns]
```

```
#prog4book
pop = pd.read_csv("data1/person_data.txt",
                  header=None,
                  names=["Vorname", "Nachname", "Größe",
                         "Gewicht", "Geschlecht"],
                  index_col=0,
                  quotechar="'",
                  sep=" ",
                  thousands=",")

pop.insert(loc=len(pop.columns),
```

```
                    column='BMI',
                    value=pop["Gewicht"]*10000 / (pop["Größe"]**2))

pop.head(10)
```

```
          Nachname  Größe  Gewicht Geschlecht        BMI
Vorname
Randy       Carter    184     73.0       male  21.561909
Stephanie    Smith    149     52.0     female  23.422368
Cynthia     Watson    174     63.0     female  20.808561
Jessie      Morgan    175     67.0       male  21.877551
Katherine   Carter    183     81.0     female  24.187046
David         Reed    187     60.0       male  17.158054
Stephen      Jones    192     96.0       male  26.041667
Jerry        Allen    204     91.0       male  21.866590
Billy       Wright    180     66.0       male  20.370370
Earl         Green    184     52.0       male  15.359168
```

24 Umgang Mit Nan In Pandas Und Python

```
# invisible
import pandas as pd

pd.set_option('display.max_colwidth', 65)
pd.set_option('display.max_columns', 65)
```

24.0.1 Umgang mit NaN

NaN wurde offiziell eingeführt vom IEEE-Standard für Floating-Point Arithmetic (IEEE 754). Es ist ein technischer Standard für Fließkommaberechnungen, der 1985 durch das "Institute of Electrical and Electronics Engineers" (IEEE) eingeführt wurde – Jahre bevor Python entstand, und noch mehr Jahre, bevor Pandas kreiert wurde. Der Standard wurde eingeführt, um Probleme zu lösen, die man in vielen Fließkommaßmplementierungen gefunden hatte, welche es schwierig gemacht haben, diese einfach und übergreifend zu verwenden.

Der Standard fügte NaN zu den arithmetischen Formaten – Mengen aus binären und dezimalen Fließkommadaten – hinzu.

'nan' in Python

Python ohne Pandas kennt auch NaN-Werte. Wir können solche mit `float()` erstellen:

```
n1 = float("nan")
n2 = float("Nan")
n3 = float("NaN")
n4 = float("NAN")
print(n1, n2, n3, n4)
print(type(n1))
```

```
nan nan nan nan
<class 'float'>
```

`nan` ist auch Teil des `math`-Moduls seit Python 3.5:

```
import math
n1 = math.nan
print(n1)
print(math.isnan(n1))
```

```
nan
```

```
True
```

Achtung: Führen Sie keine Vergleiche durch zwischen "NaN"-Werten und regulären Zahlen-Werten durch. Darüber hinaus gibt es keine Möglichkeit, NaN-Werte zu vergleichen und zu sortieren:

```
print(n1 == n2)
print(n1 == 0)
print(n1 == 100)
print(n2 < 0)
```

```
False
False
False
False
```

NaN in Pandas

In diesem Abschnitt möchten wir zeigen, wie man sinnvoll mit NaN-Werten in Pandas umgehen kann. Wir werden eine Datei mit Messwerten auswerten, die vereinzelt NaN-Werte aufweist. Doch bevor wir mit NaN-Werten arbeiten, bearbeiten wir zunächst eine Datei ohne jegliche NaN-Werte. Die Datei temperatures.csv beinhaltet die Temperaturen von sechs Sensoren, die alle 15 Minuten zwischen 6:00 Uhr und 19:15 Uhr gemessen wurden.

Die Daten aus dieser Datei können mit der Funktion `read_csv` eingelesen werden:

```
import pandas as pd

df = pd.read_csv("data1/temperatures.csv",
                 sep=";",
                 index_col=0,
                 decimal=",")
print(df.head())
```

```
         sensor1  sensor2  sensor3  sensor4  sensor5  sensor6
time
06:00:00     14.3     13.7     14.2     14.3     13.5     13.6
06:15:00     14.5     14.5     14.0     15.0     14.5     14.7
06:30:00     14.6     15.1     14.8     15.3     14.0     14.2
06:45:00     14.8     14.5     15.6     15.2     14.7     14.6
07:00:00     15.0     14.9     15.7     15.6     14.0     15.3
```

Wir wollen pro Messzeitpunkt die Durchschnittstemperatur berechnen. Dazu können wir die DataFrame-Methode `mean` verwenden. Bei Verwendung der Methode `mean` ohne Parameter werden die Spalten aufsummiert. Auch wenn dies nicht das ist, was wir wollen, ist es aber trotzdem interessant, denn damit haben wir die Durchschnitt über den Messtag berechnet.

```
df.mean()
```

```
sensor1    19.775926
sensor2    19.757407
sensor3    19.840741
sensor4    20.187037
sensor5    19.181481
```

```
sensor6    19.437037
dtype: float64
```

Was wir eigentlich bestimmen wollen, ist die Durchschnittstemperatur über alle sechs Sensoren. Dazu setzen wir den Parameter `axis` auf den Wert `1`:

```
average_temp_series = df.mean(axis=1)
print(average_temp_series[:8]) # die ersten 8 Zeilen
```

```
time
06:00:00    13.933333
06:15:00    14.533333
06:30:00    14.666667
06:45:00    14.900000
07:00:00    15.083333
07:15:00    15.116667
07:30:00    15.283333
07:45:00    15.116667
dtype: float64
```

```
sensors = df.columns.values
# all columns will be removed:
df = df.drop(sensors, axis=1)
print(df[:5])
```

```
Empty DataFrame
Columns: []
Index: [06:00:00, 06:15:00, 06:30:00, 06:45:00, 07:00:00]
```

Nun fügen wir die Werte der Durchschnittstemperaturen dem DataFrame als neue Spalte `temperature` hinzu:

```
# best practice:
df = df.assign(temperature=average_temp_series)  # inplace option not available

# alternatively:
#df.loc[:,"temperature"] = average_temp_series
print(df[:5])
```

```
          temperature
time
06:00:00    13.933333
06:15:00    14.533333
06:30:00    14.666667
06:45:00    14.900000
07:00:00    15.083333
```

Beispiel mit NaNs Stellen wir uns vor, die Datei `temperatures.csv` enthielte in den Sensorspalten NaN-Werte. Ein NaN-Wert bedeutet, dass das Messgerät zu diesem Zeitpunkt keine Messung liefern konnte.

Da wir keine solche Datei haben, werden wir eine solche Datei nun zu Übungszwecken künstlich erzeugen. Wir werden die Werte aus der Datei `temperatures.csv` nutzen, um ein DataFrame zu erzeugen. Dann erzeugen wir zufallsgesteuert NaN-Werte in dieser Datenstruktur:

```
temp_df = pd.read_csv("data1/temperatures.csv",
                      sep=";",
                      index_col=0,
                      decimal=",")
```

Nun weisen wir dem DataFrame-Objekt per Zufall NaN-Werte zu. Dazu verwenden wir die `where`-Methode des DataFrame. Wenn `where` auf ein DataFrame-Objekt `df` angewendet wird, d.h. `df.where(cond, other_df)`, wird ein Objekt mit dem identischen Muster (Shape) wie `df` zurückgeliefert, dessen Werte aus `df` stammen und das korrespondierende Element aus `cond = True` ist. Ansonsten stammt der Wert aus `other_df`.

Bevor wir mit unserem Temperaturen-Beispiel weitermachen, möchten wir die Arbeitsweise der `where`-Methode an einfachen Beispielen demonstrieren:

```
s = pd.Series(range(5))
s.where(s > 1)
```

```
0    NaN
1    NaN
2    2.0
3    3.0
4    4.0
dtype: float64
```

```
import numpy as np

A = np.random.randint(1, 30, (4, 2))

df = pd.DataFrame(A, columns=['Foo', 'Bar'])
m = df % 2 == 0
df.where(m, -df, inplace=True)
print(df)
```

```
   Foo  Bar
0  -15    6
1  -25   24
2    4  -11
3   -1    2
```

Für unser Temperaturen-Beispiel brauchen wir ein DataFrame `nan_df`, welches nur NaN-Werte beinhaltet und dasselbe Muster (Shape) wie unser Temperaturen-DataFrame `temp_df` aufweist. Dieses DataFrame verwenden wir dann in der `where`-Methode. Zusätzlich brauchen wir ein DataFrame `df_bool` mit den Bedingungen als `True`-Werten. Dazu erstellen wir ein DataFrame-Objekt mit Zufallswerten zwischen 0 und 1 mit der Anweisung `random_df < 0.8`. Damit erhalten wir das DataFrame-Objekt `df_bool`, in dem ca. 80 % der Werte `True` sind:

```
random_df = pd.DataFrame(np.random.random(size=(54, 6)),
                         columns=temp_df.columns.values,
                         index=temp_df.index)
nan_df = pd.DataFrame(np.nan,
                      columns=temp_df.columns.values,
                      index=temp_df.index)
```

```
df_bool = random_df<0.8

print(df_bool[:5])
```

```
          sensor1  sensor2  sensor3  sensor4  sensor5  sensor6
time
06:00:00    False    False     True     True    False    False
06:15:00     True     True     True     True     True    False
06:30:00     True    False     True     True     True     True
06:45:00     True     True     True     True    False     True
07:00:00    False     True     True     True     True     True
```

Wir haben nun alles zusammen, um unser DataFrame mit unvollständigen Messungen mittels `where` zu erstellen und dieses DataFrame dann mit der Methode `to_csv` in der Datei `temperatures_with_NaN.csv` abzuspeichern:

```
disturbed_data = temp_df.where(df_bool, nan_df)

disturbed_data.to_csv("data1/temperatures_with_NaN.csv")
print(disturbed_data[:10])
```

```
          sensor1  sensor2  sensor3  sensor4  sensor5  sensor6
time
06:00:00      NaN      NaN     14.2     14.3      NaN      NaN
06:15:00     14.5     14.5     14.0     15.0     14.5      NaN
06:30:00     14.6      NaN     14.8     15.3     14.0     14.2
06:45:00     14.8     14.5     15.6     15.2      NaN     14.6
07:00:00      NaN     14.9     15.7     15.6     14.0     15.3
07:15:00     15.2     15.2     14.6     15.3     15.5     14.9
07:30:00     15.4     15.3     15.6     15.6     14.7     15.1
07:45:00     15.5     14.8     15.4     15.5     14.6     14.9
08:00:00     15.7     15.6     15.9     16.2     15.4     15.4
08:15:00      NaN     15.8     15.9     16.9     16.0     16.2
```

dropna() verwenden

`dropna` ist eine DataFrame-Methode. Wenn wir diese Methode ohne Argumente verwenden, wird ein Objekt zurückgegeben, bei dem jede Zeile entfernt wurde, in der Daten gefehlt haben, also NaN-Werte waren:

```
df = disturbed_data.dropna()
print(df)
```

```
          sensor1  sensor2  sensor3  sensor4  sensor5  sensor6
time
07:15:00     15.2     15.2     14.6     15.3     15.5     14.9
07:30:00     15.4     15.3     15.6     15.6     14.7     15.1
07:45:00     15.5     14.8     15.4     15.5     14.6     14.9
08:00:00     15.7     15.6     15.9     16.2     15.4     15.4
08:30:00     16.1     15.7     16.1     15.9     14.9     15.2
09:45:00     18.4     19.0     19.0     19.4     18.4     18.3
10:30:00     20.4     19.4     20.0     21.0     20.2     19.8
12:00:00     24.0     23.1     23.1     24.8     22.5     22.7
12:15:00     23.8     23.7     24.8     25.1     22.2     22.4
```

```
12:30:00     23.6     24.2     23.6     24.1     22.1     22.5
13:30:00     22.9     21.9     22.9     24.3     22.9     23.0
14:30:00     22.1     21.9     22.3     22.2     21.2     22.1
15:15:00     21.6     21.3     21.7     21.7     21.9     21.1
15:30:00     21.4     21.3     21.7     21.9     21.0     21.7
16:30:00     20.8     20.7     20.7     20.4     20.2     19.6
17:15:00     20.3     20.7     19.6     21.3     19.8     19.0
17:30:00     20.1     20.5     19.7     19.7     18.7     19.7
18:15:00     19.6     19.9     19.2     19.9     20.0     18.6
19:15:00     19.0     19.7     18.9     19.2     18.5     19.4
```

`dropna` kann auch verwendet werden, um alle Spalten zu entfernen, in denen einige Werte NaN sind. Dafür muss lediglich der Parameter `axis = 1` gesetzt werden. Wie wir im vorherigen Beispiel gesehen haben, ist der Default-Wert dafür `False`. Sollte jede Spalte der Sensoren NaN-Werte enthalten, so werden auch alle Spalten ausgeblendet:

```
df = disturbed_data.dropna(axis=1)
df[:5]
```

```
Empty DataFrame
Columns: []
Index: [06:00:00, 06:15:00, 06:30:00, 06:45:00, 07:00:00]
```

Wir ändern unsere Aufgabe: Wir sind nun nur an den Zeilen interessiert, welche mehr als einen NaN-Wert enthalten. Dafür ist der Parameter `thresh` ideal. Dieser kann auf einen Minimal-Wert gesetzt werden. `thresh` wird auf den Integer-Wert gesetzt, der die minimale Anzahl an Nicht-NaN-Werten angibt. Wir haben sechs Temperaturwerte in jeder Zeile. Mit `thresh = 5` stellen wir sicher, dass mindestens 5 von `NaN` verschiedene Werte in jeder Zeile enthalten sind:

```
cleansed_df = disturbed_data.dropna(thresh=5, axis=0)
print(cleansed_df[:7])
```

```
          sensor1  sensor2  sensor3  sensor4  sensor5  sensor6
time
06:15:00     14.5     14.5     14.0     15.0     14.5      NaN
06:30:00     14.6      NaN     14.8     15.3     14.0     14.2
06:45:00     14.8     14.5     15.6     15.2      NaN     14.6
07:00:00      NaN     14.9     15.7     15.6     14.0     15.3
07:15:00     15.2     15.2     14.6     15.3     15.5     14.9
07:30:00     15.4     15.3     15.6     15.6     14.7     15.1
07:45:00     15.5     14.8     15.4     15.5     14.6     14.9
```

Jetzt berechnen wir erneut die Durchschnittswerte, aber diesmal aus `cleansed_df`, d.h. das DataFrame, aus dem bereits alle Zeilen entfernt wurden, die mehr als einen NaN-Wert hatten:

```
average_temp_series = cleansed_df.mean(axis=1)
sensors = cleansed_df.columns.values
df = cleansed_df.drop(sensors, axis=1) # nicht unbedingt notwendig

df = df.assign(temperature=average_temp_series)  # inplace option not available
print(df[:6])
```

```
          temperature
```

```
time
06:15:00    14.500000
06:30:00    14.580000
06:45:00    14.940000
07:00:00    15.100000
07:15:00    15.116667
07:30:00    15.283333
```

```
#prog4book

cleansed_df = disturbed_data.dropna(thresh=4, axis=0)
print(cleansed_df[:7])
```

```
          sensor1  sensor2  sensor3  sensor4  sensor5  sensor6
time
06:15:00     14.5     14.5     14.0     15.0     14.5      NaN
06:30:00     14.6      NaN     14.8     15.3     14.0     14.2
06:45:00     14.8     14.5     15.6     15.2      NaN     14.6
07:00:00      NaN     14.9     15.7     15.6     14.0     15.3
07:15:00     15.2     15.2     14.6     15.3     15.5     14.9
07:30:00     15.4     15.3     15.6     15.6     14.7     15.1
07:45:00     15.5     14.8     15.4     15.5     14.6     14.9
```

```
#prog4book

average_temp_series = cleansed_df.mean(axis=1)
sensors = cleansed_df.columns.values
df = cleansed_df.drop(sensors, axis=1)

df = df.assign(temperature=average_temp_series)
print(df[:6])
```

```
          temperature
time
06:15:00    14.500000
06:30:00    14.580000
06:45:00    14.940000
07:00:00    15.100000
07:15:00    15.116667
07:30:00    15.283333
```

25 Pandas Python Binning

```
# invisible
import numpy as np
import pandas as pd
np.core.arrayprint._line_width = 60


pd.set_option('display.max_colwidth', 65)
pd.set_option('display.max_columns', 5)
```

25.0.1 Binning in Python und Pandas

Einführung

Binning ist eine Technik, die in der Datenverarbeitung und Statistik verwendet wird. Unter Binning versteht man eine Klassenbildung in der Vorverarbeitung bei der Datenanalyse. Eine gegebene Menge von Werten, die sortiert sind, werden in Intervalle aufgeteilt. Diese Intervalle bezeichnet man im Englischen als “bins” (deutsch “Behälter”). Jedes dieser Intervalle (Bins) wird dann durch einen Repräsentanten bezeichnet. Man bezeichnet diese auch als Intervallabels. Binning wird häufig angewendet, wenn es mehr mögliche Daten gibt, als man eigentlich braucht. So können bspw. Körpergrössen von Menschen in Intervallen oder Kategorien eingeteilt werden.

Nehmen wir an, wir messen die Körpergrößen von 30 Menschen. Die Größenwerte können, grob geschätzt, zwischen 1,30 Meter und 2,50 Meter liegen. Theoretisch gibt es nun 120 verschiedene cm-Werte, jedoch werden wir meistens nur 30 verschiedene Werte aus der Test-Gruppe beobachten.

Eine Möglichkeit um sie zu gruppieren wäre die Einteilung in die Bereiche von 1,30 - 1,50 Meter, 1,50 - 1,70 Meter, 1,70 - 1,90 Meter, usw. Die Original-Daten werden also zu den passenden “Eimern” (Buckets) zugeordnet. Weiterhin werden die Originaldaten durch die korrespondierenden Intervalle ersetzt. Binning ist also eine Form der Quantifizierung.

Einteilungen (Bins) müssen nicht unbedingt numerisch sein. Die Kategorisierung kann beispielsweise von der Art “Hunde”, “Katzen”, “Hamster” usw. sein, also von jeder erdenklichen Art.

Binning wird auch in der Bildverarbeitung verwendet, um die Datenmengen zu reduzieren, indem benachbarte Pixel zu einzelnen Pixeln kombiniert werden. Man nennt das Verfahren auch kxk-binning, weil Bereiche von k x k Pixel auf einen Pixel reduziert werden.

Pandas bietet einfache Wege, um Bins zu erstellen und so Daten einzuteilen. Bevor wir auf die Pandas-Funktionalität eingehen, möchten wir noch die Basisfunktionen von Python vorstellen, um Bins zu verwenden. Im folgenden Beispiel kommen Listen und Tupel zum Einsatz:

```
def create_bins(lower_bound, width, quantity):
    """ create_bins gibt eine Partitionierung mit gleichen Abständen
        zurück. Es handelt sich um eine aufsteigende Liste von Tupeln,
        die die Werte der Intervallgrenzen darstellen.
        A tuple bins[i], i.e. (bins[i][0], bins[i][1])  with i > 0
        and i < quantity, satisfies the following conditions:
            (1) bins[i][0] + width == bins[i][1]
            (2) bins[i-1][0] + width == bins[i][0] and
                bins[i-1][1] + width == bins[i][1]
    """


    bins = []
    for low in range(lower_bound,
                     lower_bound + quantity * width + 1, width):
        bins.append((low, low+width))
    return bins
```

Wir erstellen nun 5 bins (quantity=5) mit einer Breite von 10 (width=10) und beginnen bei 10 (lower_bound=10):

```
bins = create_bins(lower_bound=10,
                   width=10,
                   quantity=5)

print(bins)
```

```
[(10, 20), (20, 30), (30, 40), (40, 50), (50, 60), (60, 70)]
```

Die nächste Funktion `find_bin()` wird mit einer Liste oder einem Tupel `bins` aufgerufen, welches 2er-Tupel oder Listen mit zwei Elementen enthalten muss. Die Funktion ermittelt den Index des Intervalls, der zu dem Wert `value` gehört:

```
def find_bin(value, bins):
    """ 'bins' ist ist eine List von Tupeln, wie beispielsweise
        [(0,20), (20, 40), (40, 60)]
        binning gibt den kleinsten Index i von 'bins' zurück,
        sodass gilt:
        bin[i][0] <= value < bin[i][1]
    """

    for i in range(0, len(bins)):
        if bins[i][0] <= value < bins[i][1]:
            return i
    return -1
```

```
from collections import Counter

bins = create_bins(lower_bound=50,
                   width=4,
                   quantity=10)

print(bins)
```

```
weights_of_persons = [73.4, 69.3, 64.9, 75.6, 74.9, 80.3,
                      78.6, 84.1, 88.9, 90.3, 83.4, 69.3,
                      52.4, 58.3, 67.4, 74.0, 89.3, 63.4]

binned_weights = []

for value in weights_of_persons:
    bin_index = find_bin(value, bins)
    print(value, bin_index, bins[bin_index])
    binned_weights.append(bin_index)

frequencies = Counter(binned_weights)
print(frequencies)
```

```
[(50, 54), (54, 58), (58, 62), (62, 66), (66, 70), (70, 74), (74, 78), (78, 82),
↪  (82, 86), (86, 90), (90, 94)]
73.4 5 (70, 74)
69.3 4 (66, 70)
64.9 3 (62, 66)
75.6 6 (74, 78)
74.9 6 (74, 78)
80.3 7 (78, 82)
78.6 7 (78, 82)
84.1 8 (82, 86)
88.9 9 (86, 90)
90.3 10 (90, 94)
83.4 8 (82, 86)
69.3 4 (66, 70)
52.4 0 (50, 54)
58.3 2 (58, 62)
67.4 4 (66, 70)
74.0 6 (74, 78)
89.3 9 (86, 90)
63.4 3 (62, 66)
Counter({4: 3, 6: 3, 3: 2, 7: 2, 8: 2, 9: 2, 5: 1, 10: 1, 0: 1, 2: 1})
```

Binning mit Pandas

Das Pandas-Modul bietet starke Funktionalitäten für die Einteilung von Daten. Wir demonstrieren dies anhand der vorigen Daten.

Von Pandas verwendete Bins Als Bins haben wir im vorigen Beispiel eine Liste von Tupeln verwendet. Diese Liste müssen wir nun in eine Datenstruktur wandeln, welche von der Pandas-Funktion `cut()` verwendet werden kann. Diese Datenstruktur ist ein `IntervalIndex`. Wir können das mit dem Befehl `pd.IntervalIndex.from_tuples()` tun:

```
import pandas as pd

bins2 = pd.IntervalIndex.from_tuples(bins)
```

`cut` ist der Name der Pandas-Funktion, die wir brauchen, um Daten in Bins einzuteilen. `cut` erwartet einige Parameter. Die wichtigsten sind aber `x` für die aktuellen Werte und `bins`, welcher den IntervalIndex definiert. `x` kann jede eindimensionale Array-ähnliche Sruktur

sein, wie z.B. Listen, Tupel, nd-arrays usw.:

```
categorical_object = pd.cut(weights_of_persons, bins2)
print(categorical_object)
```

```
[(70, 74], (66, 70], (62, 66], (74, 78], (74, 78], ..., (58, 62], (66, 70], (70,
↪  74], (86, 90], (62, 66]]
Length: 18
Categories (11, interval[int64]): [(50, 54] < (54, 58] < (58, 62] < (62, 66] ...
↪  (78, 82] < (82, 86] < (86, 90] < (90, 94]]
```

Das Ergebnis der Funktion `cut()` ist ein sogenanntes "Kategorisches Objekt" (Categorical object). Jedes "bin" entspricht einer Kategorie. Die Kategorien sind in einer mathematischen Notation beschrieben. `(70, 74]` bedeutet, dass dieses "bin" Werte beinhaltet zwischen 70 (exklusive) und 74 (inklusive). Mathematisch handelt es sich dabei um ein halb-offenes Intervall. Das heißt, dass ein Endpunkt des Intervalls inklusive ist, der anderes Endpunkt dagegen nicht. Manchmal wird es auch halb-geschlossenes Intervall genannt. In unserem vorherigen Kapitel haben wir auch ein halb-offenes Intervall definiert, jedoch anders herum, d.h. dass die linke Seite offen und die rechte geschlossen war. Wenn wir `pd.IntervalIndex.from_tuples` verwenden, können wir die Öffnung der "bins" definieren, indem wir den Parameter `closed` auf einen der folgenden Werte setzen:

- 'left': linke Seite geschlossen und rechte Seite geöffnet
- 'right': (Default) linke Seite geöffnet und rechte Seite geschlossen
- 'both': beide Seiten geschlossen
- 'neither': beide Seiten geöffnet

Für das gleiche Verhalten wie im vorherigen Kapitel, setzen wir den Parameter `closed = 'left'`:

```
bins2 = pd.IntervalIndex.from_tuples(bins, closed="left")
categorical_object = pd.cut(weights_of_persons, bins2)
print(categorical_object)
```

```
[[70, 74), [66, 70), [62, 66), [74, 78), [74, 78), ..., [58, 62), [66, 70), [74,
↪  78), [86, 90), [62, 66)]
Length: 18
Categories (11, interval[int64]): [[50, 54) < [54, 58) < [58, 62) < [62, 66) ...
↪  [78, 82) < [82, 86) < [86, 90) < [90, 94)]
```

Andere Wege um Bins zu definieren Wir haben `IntervalIndex` verwendet, um die Gewichtsdaten in "bins" einzuteilen. Die Funktion `cut()` kann noch mit zwei weiteren Arten der bin-Repräsentation umgehen:

- Integer-Werte: Angabe eines Integer-Wertes für die gleiche Breite aller "Bins" im Bereich der Werte `x`. Die Range von `x` wird auf jeder Seite um 0.1% erweitert, um die Minimum- und Maximum-Werte zu inkludieren.
- Skalare Sequenzen: Definiert die "bin"-Kanten, was eine ungleiche Breite der "bins" erlaubt. Die Range von `x` wird nicht erweitert.

```
categorical_object = pd.cut(weights_of_persons, 18)

print(categorical_object[:5])
```

```
[(71.35, 73.456], (69.244, 71.35], (62.928, 65.033], (75.561, 77.667], (73.456,
↪  75.561]]
Categories (18, interval[float64]): [(52.362, 54.506] < (54.506, 56.611] <
↪  (56.611, 58.717] < (58.717, 60.822] ... (81.878, 83.983] < (83.983, 86.089] <
↪  (86.089, 88.194] < (88.194, 90.3]]
```

```
sequence_of_scalars = [ x[0] for x in bins]
sequence_of_scalars.append(bins[-1][1])
print(sequence_of_scalars)
categorical_object = pd.cut(weights_of_persons,
                            sequence_of_scalars,
                            right=False)
for i in range(len(categorical_object)):
    print(categorical_object[i])
```

```
[50, 54, 58, 62, 66, 70, 74, 78, 82, 86, 90, 94]
[70, 74)
[66, 70)
[62, 66)
[74, 78)
[74, 78)
[78, 82)
[78, 82)
[82, 86)
[86, 90)
[90, 94)
[82, 86)
[66, 70)
[50, 54)
[58, 62)
[66, 70)
[74, 78)
[86, 90)
[62, 66)
```

Bins und Werte zählen Die nächste und interessanteste Frage ist, wie wir denn die aktuellen Werte eines Bins sehen können. Das können wir mit der Funktion `value_counts()` erreichen:

```
print(pd.value_counts(categorical_object))
```

```
[74, 78)    3
[66, 70)    3
[86, 90)    2
[82, 86)    2
[78, 82)    2
[62, 66)    2
[90, 94)    1
[70, 74)    1
[58, 62)    1
[50, 54)    1
[54, 58)    0
dtype: int64
```

`categorical_object.codes` bietet eine Beschriftung der Eingangswerte in die "binning"-Kategorien:

```
labels = categorical_object.codes
print(labels)
```

```
[ 5  4  3  6  6  7  7  8  9 10  8  4  0  2  4  6  9  3]
```

`categories` ist der IntervalIndex der Kategorien der Label-Indizes:

```
categories = categorical_object.categories
categories
```

```
IntervalIndex([[50, 54), [54, 58), [58, 62), [62, 66), [66, 70) ... [74, 78),
↪  [78, 82), [82, 86), [86, 90), [90, 94)]
              closed='left',
              dtype='interval[int64]')
```

Zusammenhang zwischen Gewichtsdaten und "bins":

```
for index in range(len(weights_of_persons)):
    label_index = labels[index]
    print(weights_of_persons[index],
          label_index,
          categories[label_index] )
```

```
73.4 5 [70, 74)
69.3 4 [66, 70)
64.9 3 [62, 66)
75.6 6 [74, 78)
74.9 6 [74, 78)
80.3 7 [78, 82)
78.6 7 [78, 82)
84.1 8 [82, 86)
88.9 9 [86, 90)
90.3 10 [90, 94)
83.4 8 [82, 86)
69.3 4 [66, 70)
52.4 0 [50, 54)
58.3 2 [58, 62)
67.4 4 [66, 70)
74.0 6 [74, 78)
89.3 9 [86, 90)
63.4 3 [62, 66)
```

Bins benennen Stellen wir uns vor, wir hätten eine Universität, die in Abhängigkeit der Durchschnittsnote (GPA - Grade point average) drei Stufen der "Latin honors" verleiht:

- "summa cum laude" setzt einen GPA über 3.9 voraus
- "magna cum laude", wenn der GPA über 3.8 ist
- "cum laude", wenn der GPA 3.6 oder höher ist

```
degrees = ["none",
           "cum laude",
           "magna cum laude",
           "summa cum laude"]
```

```
student_results = [3.93, 3.24, 2.80,
                   2.83, 3.91, 3.698,
                   3.731, 3.25, 3.24,
                   3.82, 3.22]

student_results_degrees = pd.cut(student_results,
                                 [0, 3.6, 3.8, 3.9, 4.0],
                                 labels=degrees)
print(pd.value_counts(student_results_degrees))
```

```
none               6
summa cum laude    2
cum laude          2
magna cum laude    1
dtype: int64
```

Schauen wir uns die einzelnen Bewertungen/Benotungen der Studenten an:

```
labels = student_results_degrees.codes
categories = student_results_degrees.categories

for index in range(len(student_results)):
    label_index = labels[index]
    print(student_results[index],
          label_index,
          categories[label_index] )
```

```
3.93 3 summa cum laude
3.24 0 none
2.8 0 none
2.83 0 none
3.91 3 summa cum laude
3.698 1 cum laude
3.731 1 cum laude
3.25 0 none
3.24 0 none
3.82 2 magna cum laude
3.22 0 none
```

26 Pandas Multi Level Indexing

```
# invisible
import pandas as pd

pd.set_option('display.max_colwidth', 65)
pd.set_option('display.max_columns', 65)
import numpy as np
np.core.arrayprint._line_width = 65
```

26.0.1 Mehrstufige Indizierung

Einführung

Die Basiskonzepte von Pandas haben wir im vorherigen Kapitel gelernt. Dabei haben wir uns die Daten-Strukturen - Series und - DataFrame

angeschaut.

Ebenso haben wir gelernt, wie man Series- und DataFrame-Objekte in numerischen Python-Programmen erstellt und manipuliert.

Jetzt wollen wir weitere Aspekte dieser Datenstrukturen betrachen. Wir beginnen mit den fortgeschrittenen Indizierungsmöglichkeiten in Pandas.

Mehrstufig indizierte Series

Mehrstufige Indizierung ist sowohl für Series als auch für DataFrame verfügbar. Es ist eine faszinierende Möglichkeit, in höheren Daten-Dimensionen mit den Pandas-Datenstrukturen zu arbeiten. Ein effizienter Weg, um beliebig hoch dimensionierte Daten zu speichern und zu manipulieren und damit in 1-dimensionalen (Series) oder 2-dimensionalen (DataFrames) Strukturen zu arbeiten. Mit anderen Worten können wir mit höher-dimensionierten Daten in niedrigeren Dimensionen arbeiten. Es ist Zeit für ein Beispiel in Python:

```
import pandas as pd

cities = ["Vienna", "Vienna", "Vienna",
          "Hamburg", "Hamburg", "Hamburg",
          "Berlin", "Berlin", "Berlin",
          "Zürich", "Zürich", "Zürich"]
index = [cities, ["country", "area", "population",
                  "country", "area", "population",
                  "country", "area", "population",
                  "country", "area", "population"]]

data = ["Austria", 414.60,    1805681,
```

```
        "Germany", 755.00,    1760433,
        "Germany", 891.85,    3562166,
        "Switzerland", 87.88, 378884]

city_series = pd.Series(data, index=index)
print(city_series)
```

```
Vienna   country           Austria
         area              414.6
         population       1805681
Hamburg  country           Germany
         area              755.0
         population       1760433
Berlin   country           Germany
         area               891.85
         population       3562166
Zürich   country       Switzerland
         area               87.88
         population        378884
dtype: object
```

```
cities_data = { ("Vienna", "country"): "Austria",
                ("Vienna", "area"): 414.6,
                ("Vienna", "population"): 1805681,
                ("Hamburg", "country"): "Germany",
                ("Hamburg", "area"): 755,
                ("Hamburg", "population"): 1760433,
                ("Berlin", "country"): "Germany",
                ("Berlin", "area"): 891.85,
                ("Berlin", "population"): 3562166,
                ("Zürich", "country"): "Switzerland",
                ("Zürich", "area"): 87.88,
                ("Zürich", "population"): 378884 }

city_series = pd.Series(cities_data)

city_series
```

```
Vienna   country           Austria
         area              414.6
         population       1805681
Hamburg  country           Germany
         area                 755
         population       1760433
Berlin   country           Germany
         area               891.85
         population       3562166
Zürich   country       Switzerland
         area               87.88
         population        378884
dtype: object
```

Zugriffsmöglichkeiten

Wir können über folgenden Weg auf die Daten, die mit dem ersten Index bezeichnet sind, zugreifen:

```
print(city_series["Vienna"])
```

```
country       Austria
area            414.6
population    1805681
dtype: object
```

Ebenso kann auf die Information über das Land (country), Gebiet (area) oder Bevölkerung (population) einer Stadt zugegriffen werden. Dazu gibt es zwei Möglichkeiten:

```
print(city_series["Vienna"]["area"])
```

```
414.6
```

Zur Vervollständigung der zweite Weg:

```
print(city_series["Vienna", "area"])
```

```
414.6
```

Wenn der Index geordnet ist, kann auch die Slicing-Operation angewendet werden:

```
city_series = city_series.sort_index()
print("city_series with sorted index:")
print(city_series)

print("\nSlicing the city_series:")
print(city_series["Berlin":"Vienna"])
```

```
city_series with sorted index:
Berlin   area               891.85
         country           Germany
         population        3562166
Hamburg  area                   755
         country           Germany
         population        1760433
Vienna   area                 414.6
         country           Austria
         population        1805681
Zürich   area                 87.88
         country       Switzerland
         population         378884
dtype: object

Slicing the city_series:
Berlin   area           891.85
         country       Germany
         population    3562166
Hamburg  area              755
         country       Germany
```

```
         population    1760433
Vienna   area            414.6
         country       Austria
         population    1805681
dtype: object
```

Ebenso können dann auch die Inhalte mehrerer Städte selektiv ausgegeben werden, indem man eine Liste der Stadtnamen als Schlüssel verwendet:

```
city_series[["Vienna", "Berlin"]]
```

```
Vienna  area            414.6
        country       Austria
        population    1805681
Berlin  area            891.85
        country       Germany
        population    3562166
dtype: object
```

Im nächsten Beispiel zeigen wir, wie mittels Slicing auf die inneren Schlüssel zugegriffen werden kann:

```
print(city_series[:, "area"])
```

```
Berlin     891.85
Hamburg       755
Vienna      414.6
Zürich      87.88
dtype: object
```

Mit `city_series.index.levels` kann man auf die einzelnen Stufen des mehrstufigen Indexes zugreifen. Bei diesem Objekt handelt es sich um eine FrozenList, über die wir hier iterieren:

```
for i in range(len(city_series.index.levels)):
    if i == 0:
        print("Oberste Hierarchiestufe:")
    elif i == 1:
        print("Untere Hierarchiestufe:")
    print(city_series.index.levels[i])
```

```
Oberste Hierarchiestufe:
Index(['Berlin', 'Hamburg', 'Vienna', 'Zürich'], dtype='object')
Untere Hierarchiestufe:
Index(['area', 'country', 'population'], dtype='object')
```

Zusammenhang zu DataFrames

Einige werden sicherlich bemerkt haben, dass man obige mehrstufige Series auch als DataFrame-Objekte darstellen könnte. Ein DataFrame ist ja bereits zweidimensional, während eine Series nur eindimensional ist, sofern man keinen mehrstufigen Index verwendet. Nun stellt sich die Frage, wie man aus der Series `city_series` ein DataFrame erzeigen kann. Man kann dies zwar mit folgendem Code erreichen, aber wir werden danach einen direkteren Weg zeigen.

```
city_df = pd.DataFrame([], index=index[0][::3])
for key in index[1][:3]:
    city_df = pd.concat([city_df,
                         city_series[:, key]],
                         axis=1,
                         sort=False)

city_df.columns = ["country", "population", "area"]
print(city_df)
```

```
            country population     area
Vienna       Austria      414.6  1805681
Hamburg      Germany        755  1760433
Berlin       Germany     891.85  3562166
Zürich   Switzerland      87.88   378884
```

Setzt man `sort` auf `False`, erhält man einen unsortierten Index, in unserem Fall:

```
city_df = pd.DataFrame([], index=index[0][::3])
for key in index[1][:3]:
    city_df = pd.concat([city_df,
                         city_series[:, key]],
                         axis=1,
                         sort=False)

city_df.columns = ["country", "population", "area"]
print(city_df)
```

```
            country population     area
Vienna       Austria      414.6  1805681
Hamburg      Germany        755  1760433
Berlin       Germany     891.85  3562166
Zürich   Switzerland      87.88   378884
```

Obiges können wir einfacher haben, indem wir die von der Series-Klasse zur Verfügung gestellte Methode `unstack` benutzen. `unstack` bietet zwei optionale Parameter:

- `level`, der per Default auf -1 gesetzt ist, bestimmt, welcher Teil des mehrstufigen Indexes als Spaltenbezeichner verwendet wird. -1 bedeutet, dass der innere Index verwendet wird. Das entspricht in unserem Beispiel city_series.index.levels[-1], also die Städtenamen. Setzen wir `level` auf 0, so werden die Städtenamen zum Index des DataFrame.
- `fill_value` ist per Default auf None gesetzt. Mit diesem Parameter kann man den Wert bestimmen, auf den `NaN`-Werte umgesetzt werden, falls diese sich in den Daten befinden.

```
city_df = city_series.unstack()
print("Für level wurde der Default-Wert -1 genutzt:")
print(city_df)

city_df = city_series.unstack(level=0)
print("\nErgebnis für level=0:")
print(city_df)
```

```
Für level wurde der Default-Wert -1 genutzt:
           area      country population
Berlin   891.85      Germany    3562166
Hamburg     755      Germany    1760433
Vienna    414.6      Austria    1805681
Zürich    87.88  Switzerland     378884

Ergebnis für level=0:
             Berlin  Hamburg   Vienna       Zürich
area         891.85      755    414.6        87.88
country     Germany  Germany  Austria  Switzerland
population  3562166  1760433  1805681       378884
```

Die DataFrame-Methode `stack` entspricht der Umkehrfunktion, d.h. aus einem DataFrame-Objekt erzeugt sie ein Series-Objekt mit mehrstufigem Index:

```
city_df.stack()
```

```
area        Berlin          891.85
            Hamburg            755
            Vienna           414.6
            Zürich           87.88
country     Berlin         Germany
            Hamburg        Germany
            Vienna         Austria
            Zürich     Switzerland
population  Berlin         3562166
            Hamburg        1760433
            Vienna         1805681
            Zürich          378884
dtype: object
```

Dreistufige Indizes

Zu Anfang dieses Kapitels haben wir gesehen, wie wir eine Series mit einem mehrstufigen Index direkt durch die Angabe einer Liste mit zwei oder mehr Index-Arrays oder Listen erzeugen können. Wir hatten ein Dictionary `cities` und die verschachtelte Liste `index` zu einer mehrstufigen `Series` gewandelt. Genaugenommen erhielten wir eine zweistufige Series. Im folgenden Beispiel zeigen wir ein Beispiel mit einem dreistufigen Index:

```
import pandas as pd

index = [ ["hot"] * 6 + ["cold"] * 6,
          (["red"] * 2  + ["green"] * 2 + ["blue"] * 2) * 2,
          ["right", "wrong"] * 6]

data = np.random.randint(100, 100000, size=(12,))

S3_series = pd.Series(data, index=index)
print(S3_series)
```

```
hot   red    right    16031
             wrong    71724
      green  right    51514
             wrong    56067
```

```
      blue   right    51977
             wrong    28134
cold  red    right     8368
             wrong    54434
      green  right     6894
             wrong    82765
      blue   right    34937
             wrong    58604
dtype: int64
```

Auch im Falle von dreistufigen Indizes können wir mit Hilfe der Methode `unstack` ein DataFrame erzeugen. Wir zeigen die verschiedenen Möglichkeiten für den Parameter `level`:

```
print(S3_series.unstack(level=-1))   # entspricht 'level=2'
```

```
            right  wrong
cold blue   34937  58604
     green   6894  82765
     red     8368  54434
hot  blue   51977  28134
     green  51514  56067
     red    16031  71724
```

```
print(S3_series.unstack(level=-0))
```

```
              cold    hot
blue  right  34937  51977
      wrong  58604  28134
green right   6894  51514
      wrong  82765  56067
red   right   8368  16031
      wrong  54434  71724
```

```
x = S3_series.unstack(level=[1, 2])
print(x)
```

```
        red         green          blue
      right  wrong  right  wrong  right  wrong
cold   8368  54434   6894  82765  34937  58604
hot   16031  71724  51514  56067  51977  28134
```

```
print(x["red", "right"])
```

```
cold     8368
hot     16031
Name: (red, right), dtype: int64
```

```
x = S3_series.unstack(level=[2, 1])
print(x)
```

```
      right  wrong  right  wrong  right  wrong
        red    red  green  green   blue   blue
cold   8368  54434   6894  82765  34937  58604
hot   16031  71724  51514  56067  51977  28134
```

```
x["right"]
```

```
        red  green   blue
cold   8368   6894  34937
hot   16031  51514  51977
```

Die Daten hätten aber auch wie in folgendem Dictionary organisiert gewesen sein können. Auch dann können wir diese direkt in ein mehrstufiges Dictionary wandeln:

Vertauschen mehrstufiger Indizes

Es ist möglich, die Ebenen eines mehrstufigen Index mit der Methode `swaplevel` zu vertauschen:

```
S3_swapped = S3_series.swaplevel()
S3_swapped.sort_index(inplace=True)
S3_swapped
```

```
cold  right  blue     34937
             green     6894
             red       8368
      wrong  blue     58604
             green    82765
             red      54434
hot   right  blue     51977
             green    51514
             red      16031
      wrong  blue     28134
             green    56067
             red      71724
dtype: int64
```

```
print(city_series)
city_series = city_series.swaplevel()
city_series.sort_index(inplace=True)
print("\n--- vertauscht ---")
city_series
```

```
Berlin   area               891.85
         country           Germany
         population        3562166
Hamburg  area                 755
         country           Germany
         population        1760433
Vienna   area                 414.6
         country           Austria
         population        1805681
Zürich   area                 87.88
         country       Switzerland
         population         378884
dtype: object

--- vertauscht ---
```

```
area        Berlin          891.85
            Hamburg            755
            Vienna           414.6
            Zürich           87.88
country     Berlin         Germany
            Hamburg        Germany
            Vienna         Austria
            Zürich     Switzerland
population  Berlin         3562166
            Hamburg        1760433
            Vienna         1805681
            Zürich          378884
dtype: object
```

```
growth_rates = {("Afghanistan", 2015): 1.31,
                ("Afghanistan", 2016): 2.37,
                ("Afghanistan", 2017): 2.60,
                ("Ägypten", 2015): 4.37,
                ("Ägypten", 2016): 4.35,
                ("Ägypten", 2017): 4.18,
                ("Albanien", 2015): 2.22,
                ("Albanien", 2016): 3.35,
                ("Albanien", 2017): 3.84}

growth_rates_series = pd.Series(growth_rates)

growth_rates_series
```

```
Afghanistan  2015    1.31
             2016    2.37
             2017    2.60
Ägypten      2015    4.37
             2016    4.35
             2017    4.18
Albanien     2015    2.22
             2016    3.35
             2017    3.84
dtype: float64
```

```
growth_rates_series = growth_rates_series.swaplevel()
growth_rates_series.sort_index(inplace=True)
growth_rates_series
```

```
2015  Afghanistan    1.31
      Albanien       2.22
      Ägypten        4.37
2016  Afghanistan    2.37
      Albanien       3.35
      Ägypten        4.35
2017  Afghanistan    2.60
      Albanien       3.84
      Ägypten        4.18
dtype: float64
```

27 Daten Visualisierung Mit Pandas Und Python

27.0.1 Datenvisualisierung mit Pandas

Einführung

Es ist selten eine gute Idee, wenn man wissenschaftliche oder geschäftliche Daten nur rein textuell dem jeweiligen Zielpublikum präsentiert. Zur Visualisierung der Daten werden meistens verschiedene Arten von Diagrammen verwendet. Dadurch wird die Kommunikation der Information effizienter und macht die Daten greifbarer. Mit anderen Worten macht es komplexe Daten zugänglicher und verständlicher. So können numerische Daten beispielsweise in Punkt-, Linien-, Säulen- und Balkendiagrammen sowie in Kreis-, Kuchen und Tortendiagrammen grafisch repräsentiert werden.

Wir haben bereits die starken Möglichkeiten von Matplotlib kennengelernt, um öffentlichkeitstaugliche Grafiken zu erstellen. Matplotlib ist ein Low-Level-Werkzeug, um dieses Ziel zu erreichen. Grafiken können noch durch Basiselemente erweitert werden wie beispielsweise Legenden, Bezeichnungen und so weiter. Mit Pandas können all diese Möglichkeiten leichter umgesetzt werden.

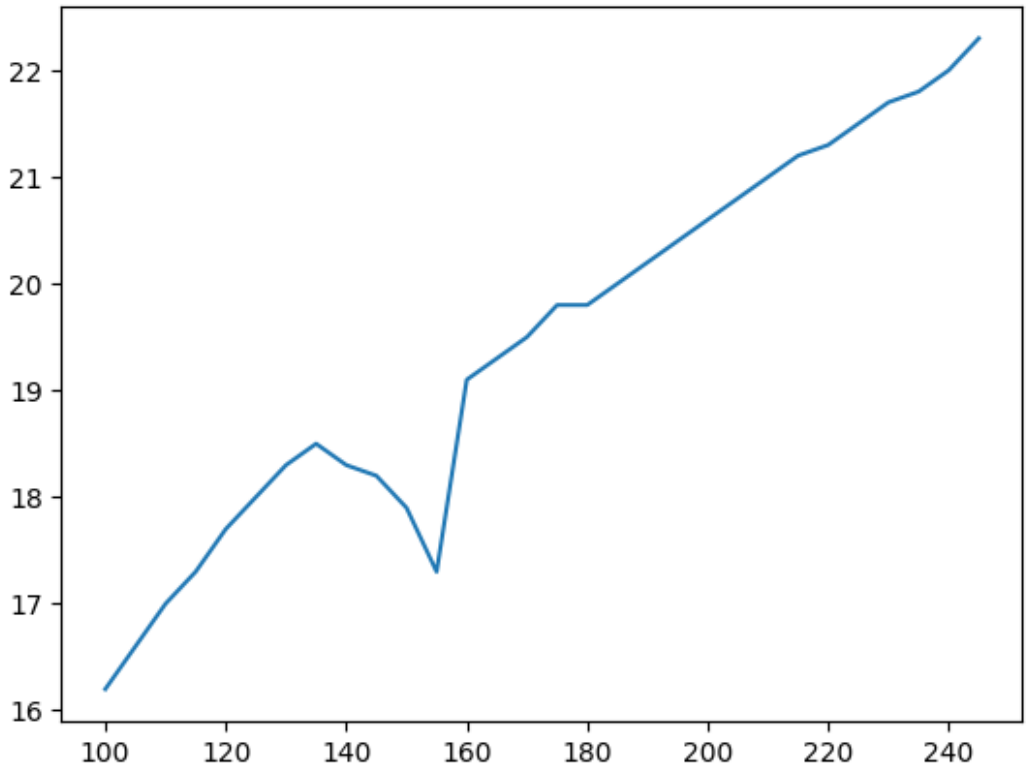
Wir beginnen mit einem einfachen Beispiel eines Liniendiagrammes.

Liniendiagramm

Series Sowohl die Pandas-Series, als auch die DataFrames, unterstüzen die Plot-Methode.

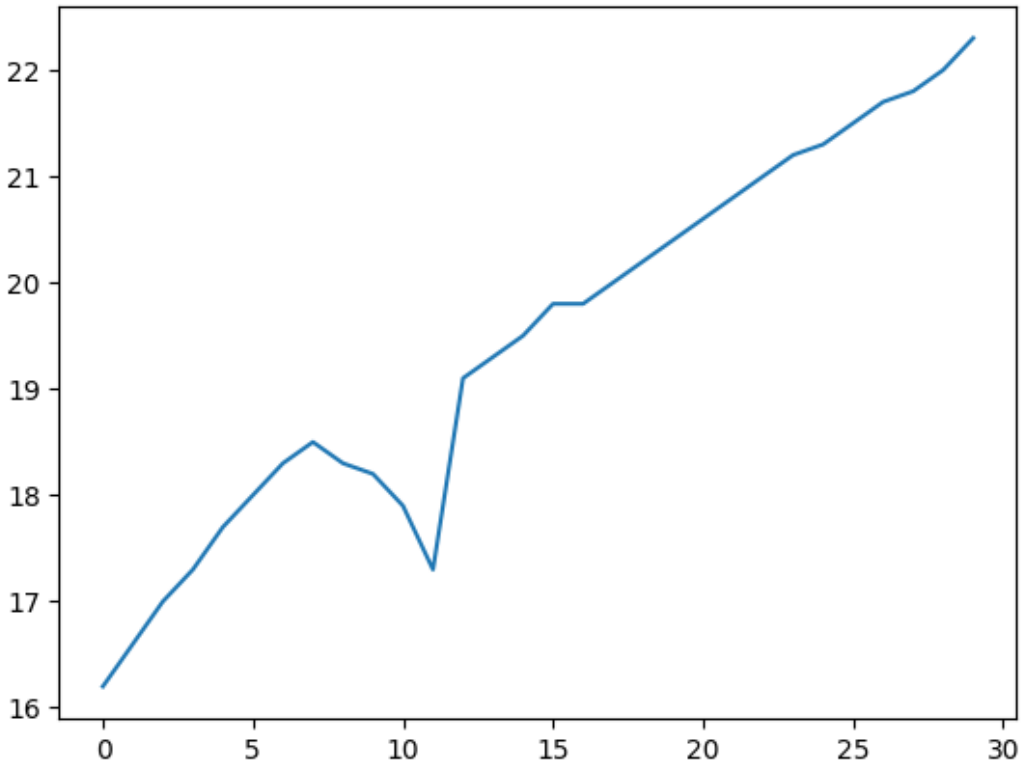
Sie können im folgenden einfaches Beispiel einer Linien-Grafik für ein Series-Objekt sehen. Wir verwenden eine einfache Python-Liste `data` für die Daten. Der Index wird für die x-Werte verwendet, oder die Domäne.

```
import matplotlib.pyplot as plt
import pandas as pd

data = [16.2, 16.6, 17.0, 17.3, 17.7, 18.0,
        18.3, 18.5, 18.3, 18.2, 17.9, 17.3,
        19.1, 19.3, 19.5, 19.8, 19.8, 20.0,
        20.2, 20.4, 20.6, 20.8, 21.0, 21.2,
        21.3, 21.5, 21.7, 21.8, 22.0, 22.3]
s = pd.Series(data, index=range(100, 250, 5))

s.plot()
plt.show()
```



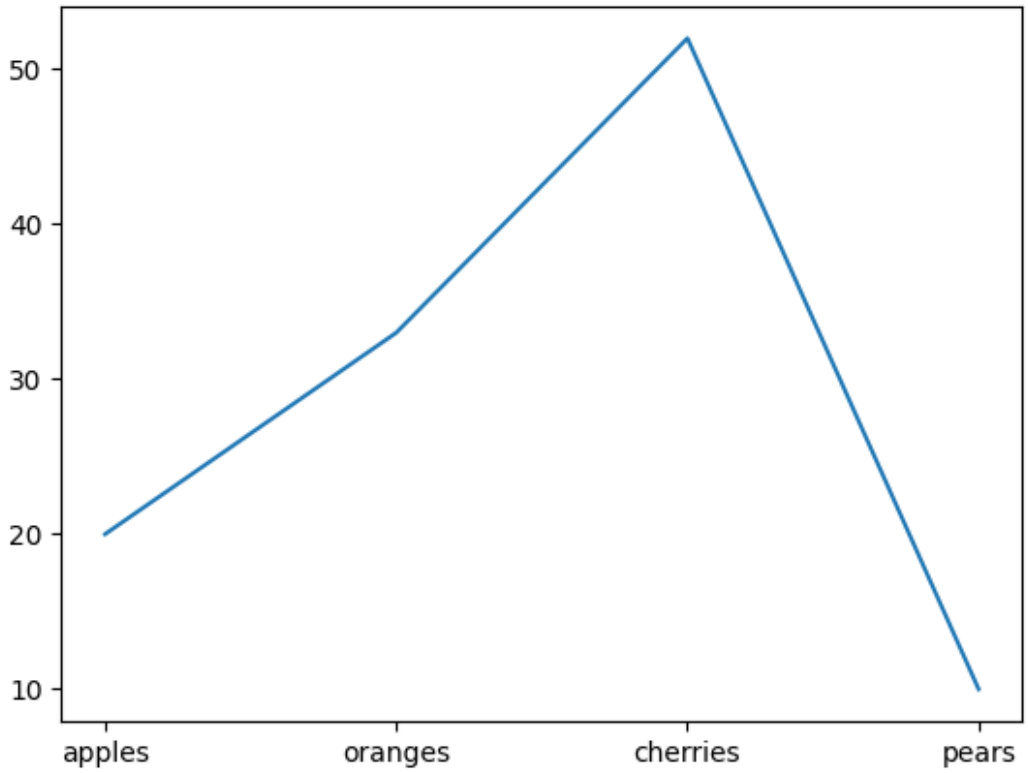
Es ist möglich die Verwendung des Indexes zu unterdrücken, indem der Parameter `use_index` auf `False` gesetzt wird:

```
s.plot(use_index=False)
plt.show()
```



Im vorigen Beispiel bestand der Index aus numerischen Werten. In vielen Fällen besteht der Index jedoch aus Stringwerten. Wir spielen nun mit einem solchen Beispiel:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
plt.plot(S)  # notwendig ab Pandas-Version 0.23.4
#S.plot()    # funktioniert bis 0.23.4
plt.show()
```



Natürlich ergibt es wenig Sinn in diesem Fall, d.h. bei kategorialen Daten, ein Liniendiagramm zu benutzen, da die Daten ja keine kontinuierliche Funktion beschreiben. Hier empfiehlt es sich zum Beispiel, ein Säulen-, Balken oder Kreisdiagramm zu benutzen.

DataFrames Wir stellen nun die Plot-Methode für DataFrames vor. Dazu definieren wir ein Dictionary mit Bevölkerungswerten und Flächenangaben. Dieses Dictionary verwenden wir dann für die Erstellung eine DataFrame-Objektes, welches wir anschliessend für die Grafik verwenden wollen:

```
# prog4book
monate = ["Januar", "Februar", "März", "April", "Mai", "Juni",
          "Juli", "August", "September", "Oktober", "November",
          "Dezember"]
max_temperatur = [17, 16, 17, 19, 23, 25, 28, 28, 27, 24, 21, 18]
durchschnitts_temperatur = [14, 13, 14, 17, 20, 24, 26, 27,
                            25, 22, 19, 15]
min_temperatur = [10, 9, 11, 14, 18, 22, 25, 26, 24, 19, 15, 12]
niederschlag_mm = [296, 278, 117, 79, 74, 24, 1, 27, 72, 155,
                   110, 283]

kreta_dict = {"Maximal": max_temperatur,
              "Durchschnitt":durchschnitts_temperatur,
              "Minimal": min_temperatur,
              "Niederschlag": niederschlag_mm}

kreta_df = pd.DataFrame(kreta_dict, index=monate)
print(kreta_df.head(5))
```

```
         Maximal  Durchschnitt  Minimal  Niederschlag
```

```
Januar        17            14        10           296
Februar       16            13         9           278
März          17            14        11           117
April         19            17        14            79
Mai           23            20        18            74
```

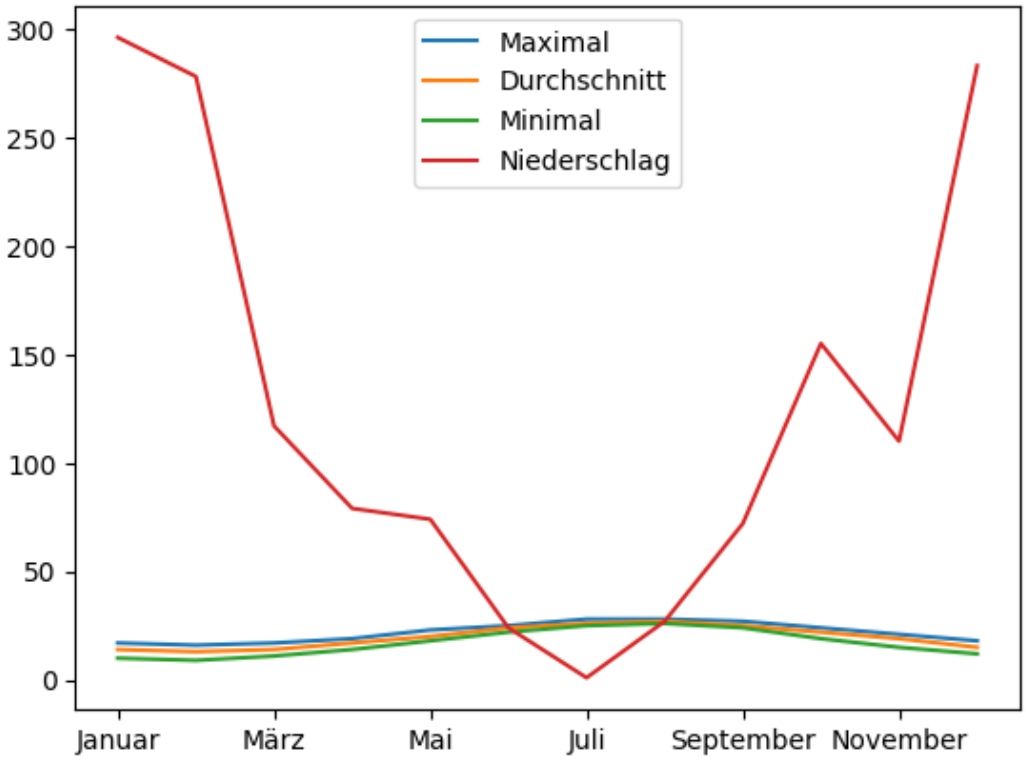
```
# prog4web
import pandas as pd

cities = {"Name": ["London", "Berlin", "Madrid", "Rom",
                   "Paris", "Wien", "Bukarest", "Hamburg",
                   "Budapest", "Warschau", "Barcelona",
                   "München", "Mailand"],
          "Bevölkerung": [8615246, 3562166, 3165235, 2874038,
                          2273305, 1805681, 1803425, 1760433,
                          1754000, 1740119, 1602386, 1493900,
                          1350680],
          "Fläche" : [1572, 891.85, 605.77, 1285,
                      105.4, 414.6, 228, 755,
                      525.2, 517, 101.9, 310.4,
                      181.8]
}

city_frame = pd.DataFrame(cities,
                          columns=["Bevölkerung", "Fläche"],
                          index=cities["Name"])
print(city_frame)
```

```
           Bevölkerung   Fläche
London         8615246  1572.00
Berlin         3562166   891.85
Madrid         3165235   605.77
Rom            2874038  1285.00
Paris          2273305   105.40
Wien           1805681   414.60
Bukarest       1803425   228.00
Hamburg        1760433   755.00
Budapest       1754000   525.20
Warschau       1740119   517.00
Barcelona      1602386   101.90
München        1493900   310.40
Mailand        1350680   181.80
```

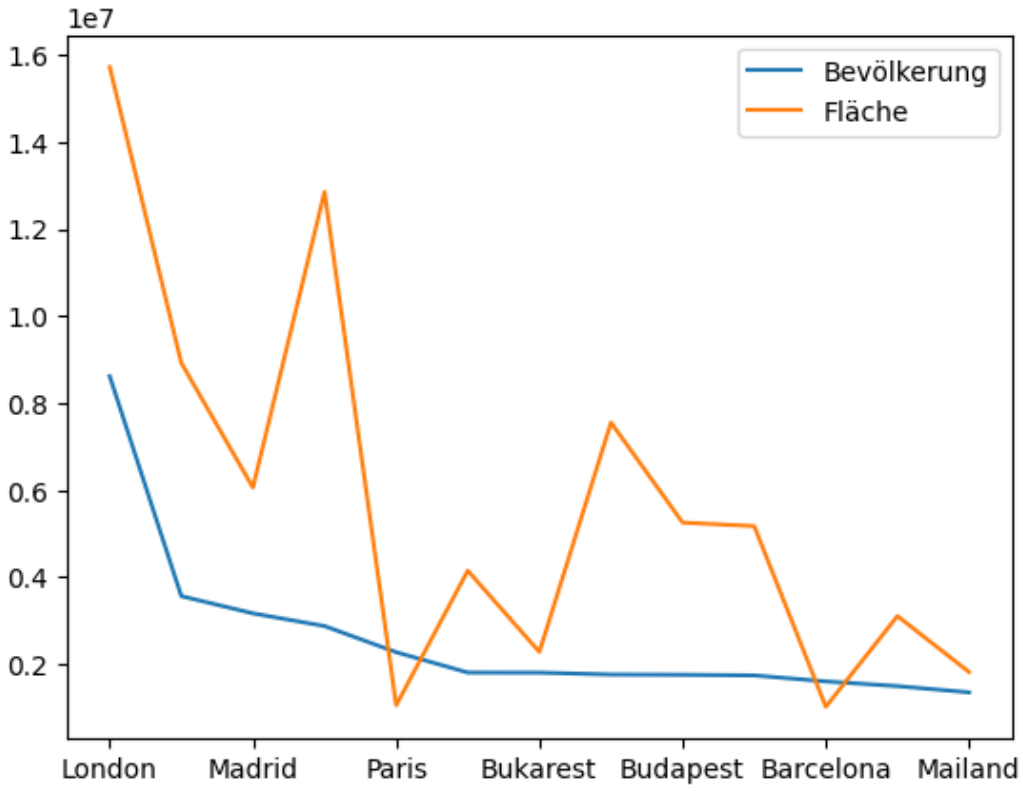
Der folgende Code erstellt die Grafik unseres DataFrames. Die Flächen-Werte werden noch mit 1000 multipliziert, da die “Flächen”-Linie sonst unsichtbar bzw. durch die x-Achse verdeckt würde:

```
# prog4book
kreta_df.plot()
```

```
<Axes: >
```



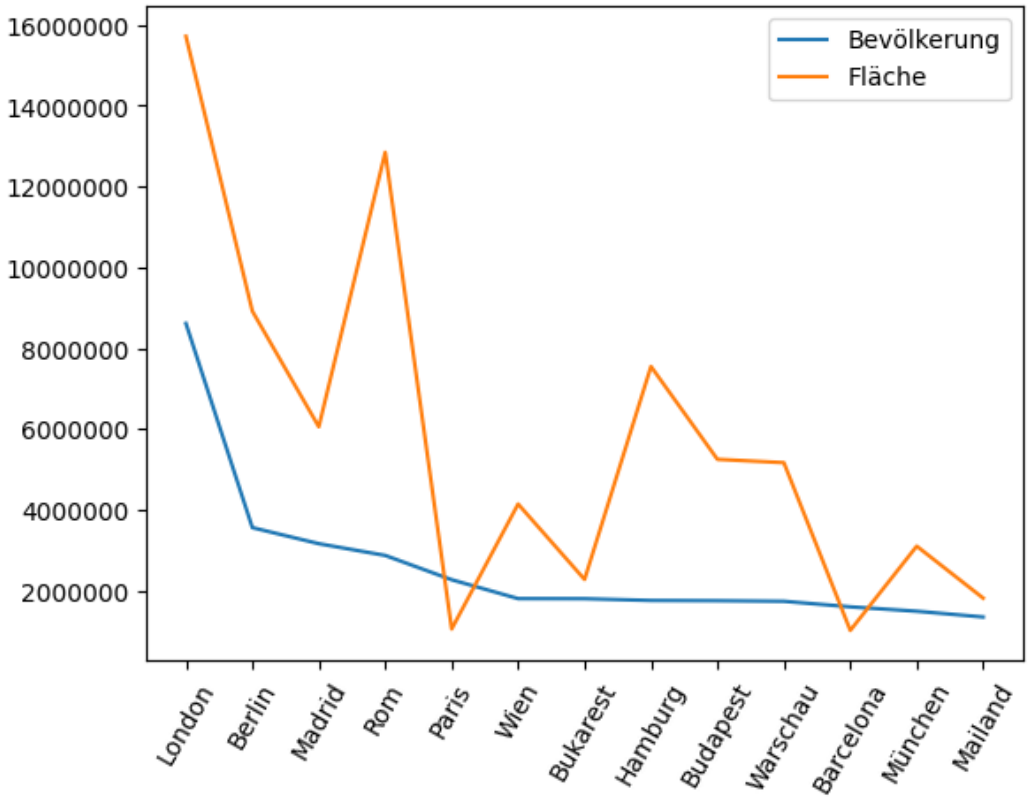

```
# prog4web
city_frame["Fläche"] *= 10000
city_frame.plot()
plt.show()
```



Die Beschriftung der Achsen entspricht nicht unseren Wünschen: Zum einen ist die X-Achse unbeschriftet und zum anderen sind die Werte der Y-Achse in wissenschaftlicher Notation angegeben. Letzeres erkennt man an der Beschriftung `1e7`, die sich oben links oberhalb der Linie befindet. `1e7` bedeutet, dass die Werte der Y-Achse mit 10^7 multipliziert werden müssen.

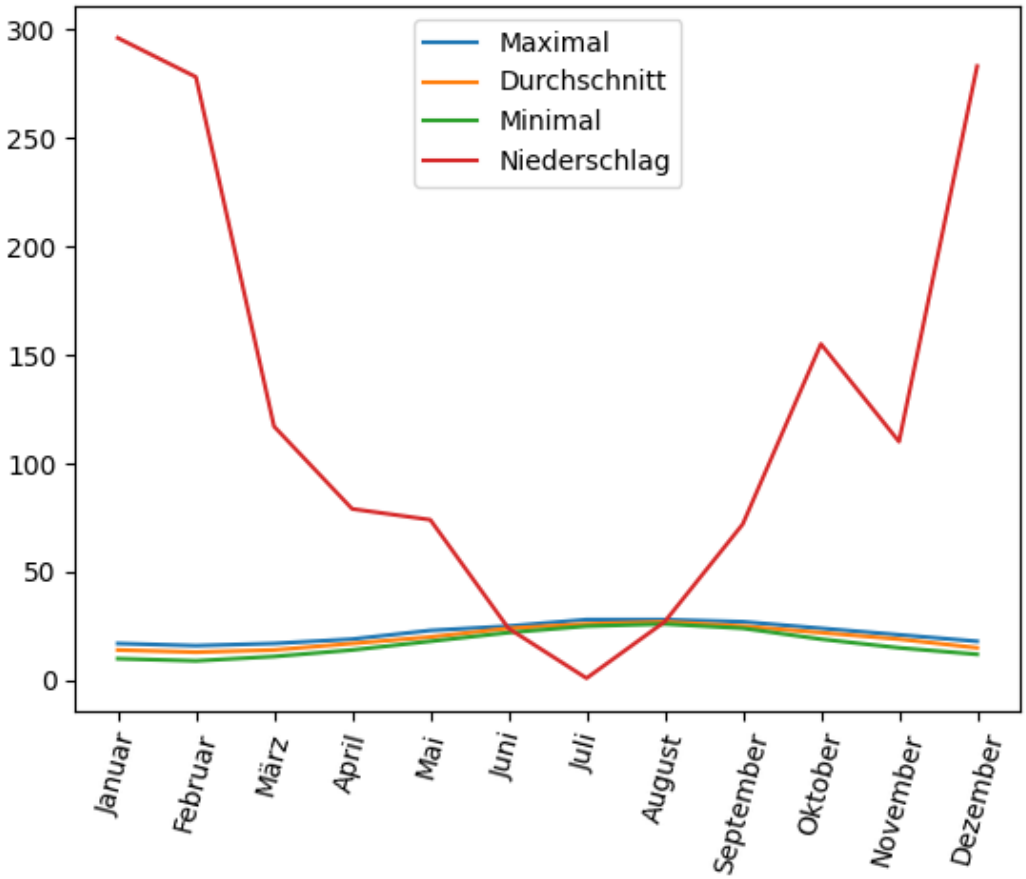
Die X-Achse können wir mit den Städtenamen versehen, indem wir explizit xticks auf den Wert range(len(city_frame.index) setzen. Der Parameter `ret` ist hierbei auch von besonderer Wichtigkeit. Mit ihm haben wir den Schrägdruck der Städtenamen erreicht, d.h. wir haben die Städtenamen um 60 Grad bezogen auf die Horizontale gedreht. Lässt man ihn weg oder setzt ihn auf Null, dann werden die Städtnamen überlappend gedruckt und damit unlesbar.

```
# prog4web
import matplotlib
city_frame.plot(xticks=range(len(city_frame.index)),
                rot=60)
plt.gca().yaxis.set_major_formatter(matplotlib.ticker.FormatStrFormatter('%d'))

plt.show()
```




```
# prog4book
kreta_df.plot(xticks=range(len(kreta_df.index)),
              rot=75)
```

```
<Axes: >
```



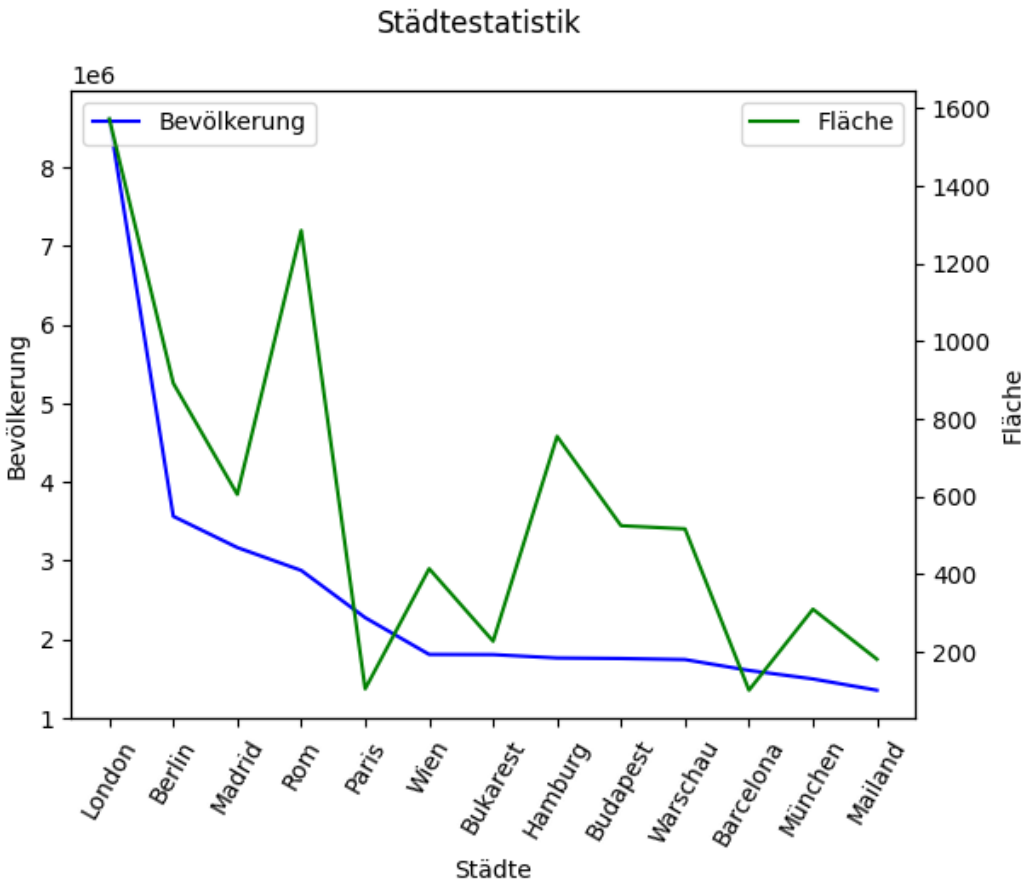
Sekundärachsen (Twin Axes) Die Flächen-Spalte hatten wir mit 10000 multipliziert, damit sie besser dargestellt werden kann. Die bessere Lösung besteht in der Verwendung von Sekundärachsen (engl. Twin Axes). Wir demonstrieren ihre Anwendung im nächsten Beispiel. Zunächst dividieren wir die Spalte `Fläche` durch 10000, um die ursprünglichen Werte wiederherzustellen:

```
# prog4web
city_frame["Fläche"] /= 10000
```

Um die Darstellung mit Sekundärachsen zu erreichen, benötigen wir subplots aus dem Modul matplotlib und die Funktion “twinx”:

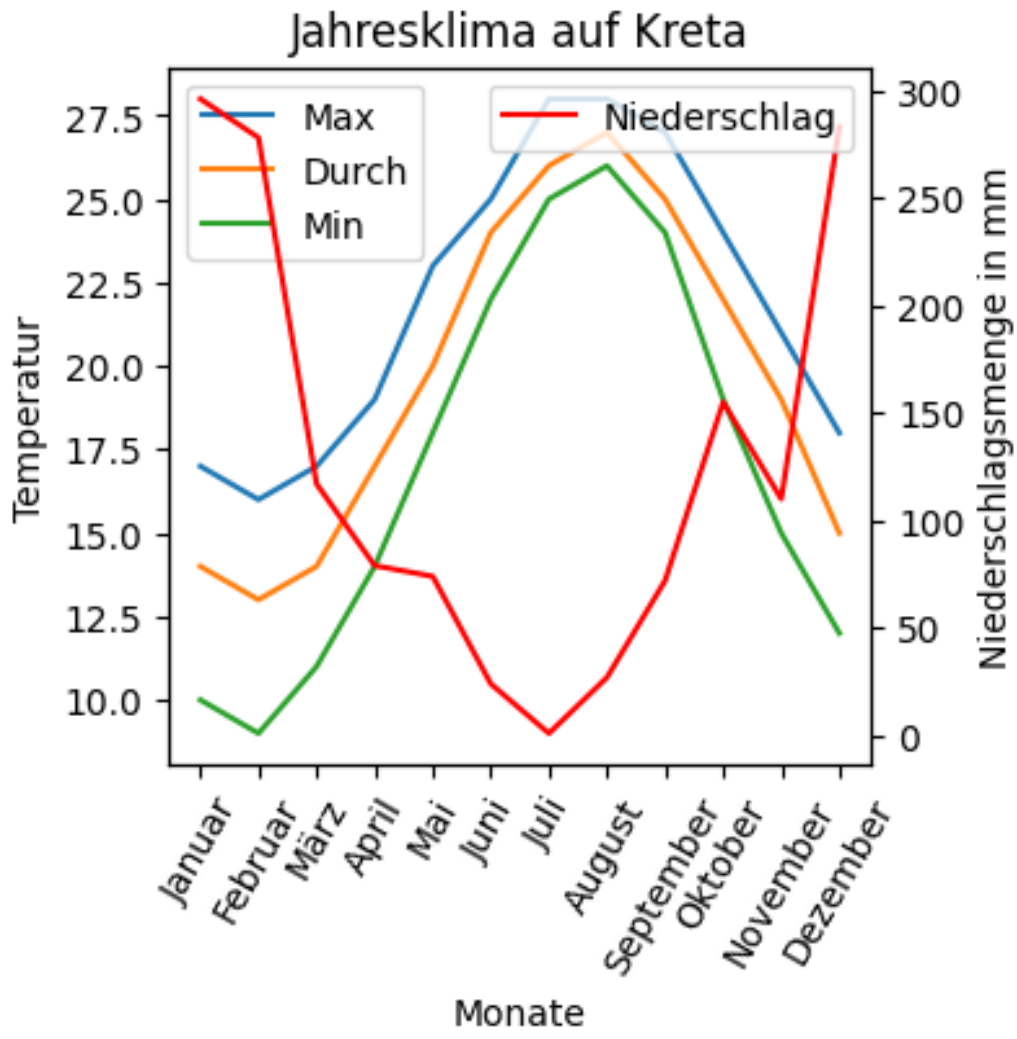
```
# prog4web
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
fig.suptitle("Städtestatistik")
ax.set_ylabel("Bevölkerung")
ax.set_xlabel("Städte")

ax2 = ax.twinx()
ax2.set_ylabel("Fläche")

line1 = city_frame["Bevölkerung"].plot(ax=ax,
                                       style="b-",
```

```
                                    xticks=range(len(city_frame.index)),
                                    use_index=True,
                                    rot=60)

line2 = city_frame["Fläche"].plot(ax=ax2,
                                  style="g-")


ax.legend(["Bevölkerung"], loc=2)
ax2.legend(loc=1)


plt.show()
```




```
# prog4book
import matplotlib.pyplot as plt

ax = kreta_df['Maximal'].plot(xticks=range(len(kreta_df.index)),
                              figsize=(3.5, 3.5),
                              use_index=True,
                              title='Jahresklima auf Kreta',
                              rot=60)

kreta_df['Durchschnitt'].plot(ax=ax,
                              xticks=range(len(kreta_df.index)),
                              use_index=True,
                              rot=60)
```

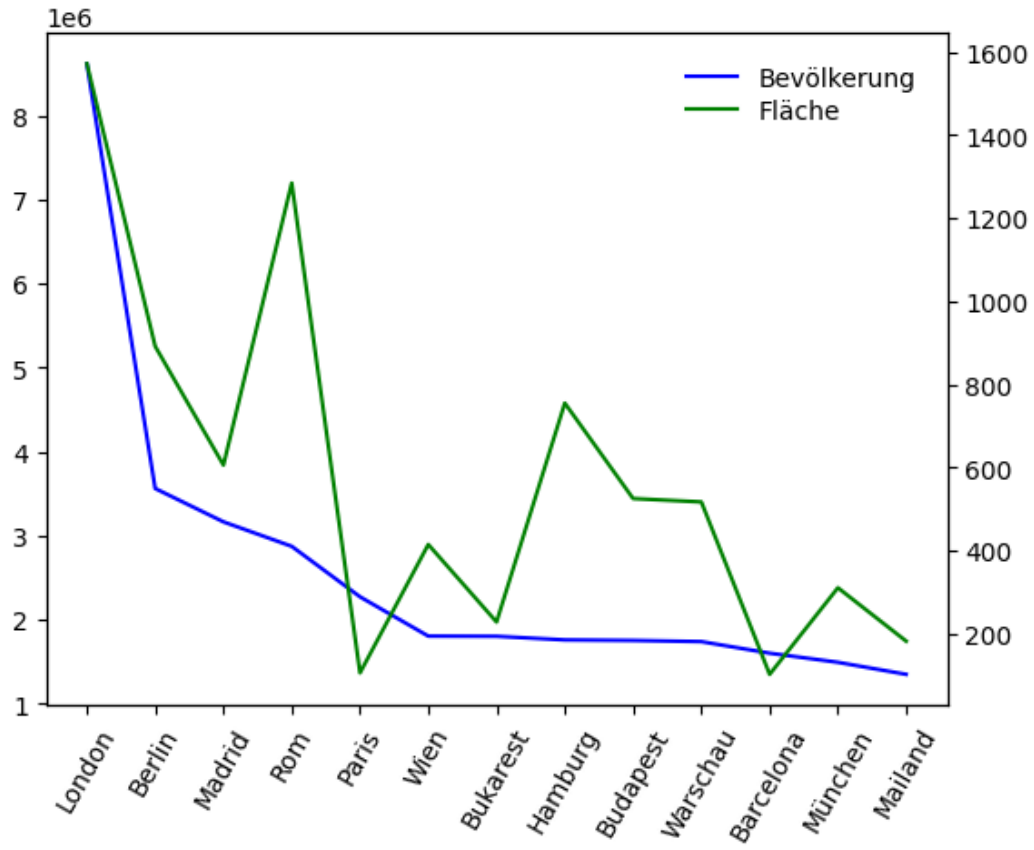
```
kreta_df['Minimal'].plot(ax=ax,
                         xticks=range(len(kreta_df.index)),
                         use_index=True,
                         rot=60)

ax2 = ax.twinx()

kreta_df['Niederschlag'].plot(ax=ax2,
                              style="r-")


ax.set_ylabel('Temperatur')
ax.set_xlabel('Monate')
ax2.set_ylabel('Niederschlagsmenge in mm')

ax.legend(['Max', 'Durch', 'Min'], loc=2)
ax2.legend(loc=1)


plt.show()
```



Wir können Sekundärachsen auch direkt in Pandas ohne Zuhilfenahme von Subl lots erzeugen, wie wir im Code des folgenden Programmes sehen:

```
# prog4web
import matplotlib.pyplot as plt

ax1 = city_frame["Bevölkerung"].plot(style="b-",
                                     xticks=range(len(city_frame.index)),
                                     use_index=True,
                                     rot=60)
ax2 = ax1.twinx()
#ax2.spines['right'].set_position(('axes', 1.0))
city_frame["Fläche"].plot(ax=ax2,
                          style="g-")



ax1.legend(loc=(.7,.9), frameon = False)
ax2.legend( loc=(.7, .85), frameon = False)
```

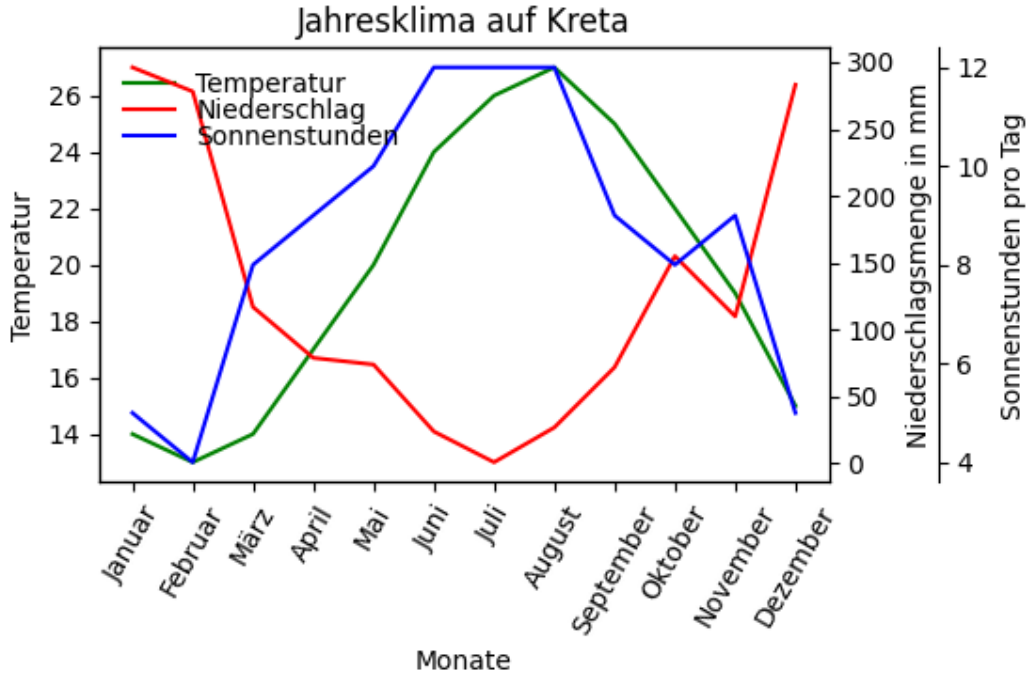
```
<matplotlib.legend.Legend at 0x74254c28f890>
```



Mehrere Y-Achsen Es lassen sich noch weitere Achsen hinzufügen.

Wir fügen eine weitere Achse zum `city_frame` hinzu, indem wir eine neue Spalte mit der Bevölkerungsdichte erstellen, d.h. die Anzahl der Menschen pro Quadratkilometer:

```
# prog4book

import matplotlib.pyplot as plt

sonnenstunden = [5, 4, 8, 9, 10, 12, 12, 12, 9, 8, 9, 5]
kreta_dict = {"Maximal": max_temperatur,
              "Durchschnitt":durchschnitts_temperatur,
              "Minimal": min_temperatur,
              "Niederschlag": niederschlag_mm,
              "Sonnenstunden": sonnenstunden}
kreta_df = pd.DataFrame(kreta_dict, index=monate)

ax = kreta_df['Durchschnitt'].plot(xticks=range(len(kreta_df.index)),
                                   figsize=(5, 3),
                                   style="g-",
                                   use_index=True,
                                   title='Jahresklima auf Kreta',
                                   rot=60)


ax_regen, ax_sonne = ax.twinx(), ax.twinx()

kreta_df['Niederschlag'].plot(ax=ax_regen,
                              style="r-")

kreta_df['Sonnenstunden'].plot(ax=ax_sonne,
                               style="b-")

ax.set_ylabel('Temperatur')
ax.set_xlabel('Monate')
ax_regen.set_ylabel('Niederschlagsmenge in mm')
ax_sonne.set_ylabel('Sonnenstunden pro Tag')

ax_regen.spines['right'].set_position(('axes', 1.0))
ax_sonne.spines['right'].set_position(('axes', 1.15))

ax.legend(['Temperatur'],
          loc='upper left',
          bbox_to_anchor=(0.0, 1.0),
          frameon=False)
ax_regen.legend(loc='upper left',
                bbox_to_anchor=(0.0, 0.94),
                frameon=False)
ax_sonne.legend(loc='upper left',
                bbox_to_anchor=(0.0, 0.88),
                frameon=False)

#help(ax_sonne.legend)

#bbox_to_anchor=(1.5, 1)

plt.show()
```



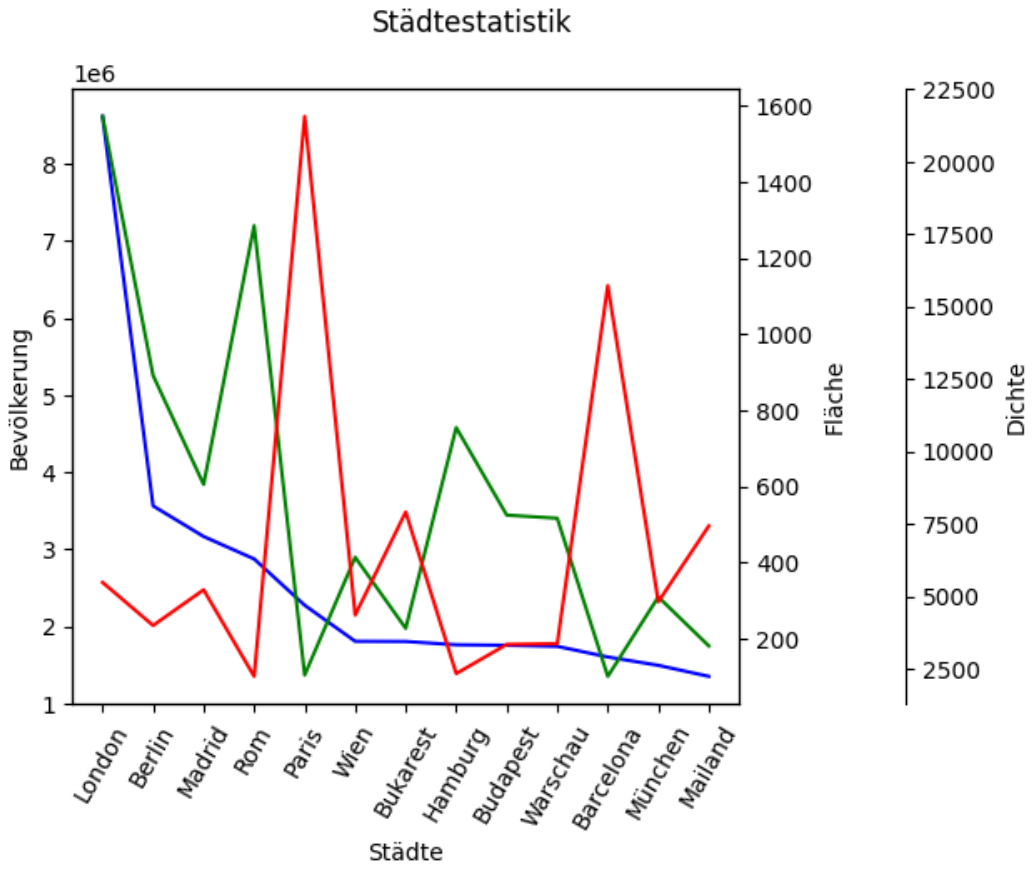

```
# prog4web
city_frame["Dichte"] = city_frame["Bevölkerung"] / city_frame["Fläche"]

print(city_frame.head(3))
```

```
        Bevölkerung    Fläche        Dichte
London      8615246   1572.00   5480.436387
Berlin      3562166    891.85   3994.131300
Madrid      3165235    605.77   5225.143206
```

Damit gibt es drei Spalten zu zeichnen. Für diesen Zweck erstellen wir drei Achsen um die Werte darzustellen:

```
# prog4web
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
fig.suptitle("Städtestatistik")
ax.set_ylabel("Bevölkerung")
ax.set_xlabel("Städte")

ax_area, ax_density = ax.twinx(), ax.twinx()
ax_area.set_ylabel("Fläche")
ax_density.set_ylabel("Dichte")

rspine = ax_density.spines['right']
rspine.set_position(('axes', 1.25))
ax_density.set_frame_on(True)
ax_density.patch.set_visible(False)
fig.subplots_adjust(right=0.75)

city_frame["Bevölkerung"].plot(ax=ax,
                               style="b-",
                               xticks=range(len(city_frame.index)),
```

```
                                use_index=True,
                                rot=60)

city_frame["Fläche"].plot(ax=ax_area,
                          style="g-")

city_frame["Dichte"].plot(ax=ax_density,
                          style="r-")

plt.show()
```



Ein komplexeres Beispiel

Wir nutzen das zuvor erlangte Wissen im nächsten Beispiel. Wir verwenden dazu eine Datei mit Besucherstatistiken der Webseite python-course.eu. Der Inhalt der Datei sieht folgendermaßen aus:

```
Month Year 'Unique visitors' 'Number of visits' Pages Hits Bandwidth Unit
Jun 2010 11 13 42 290 2.63 MB
Jul 2010 27 39 232 939 9.42 MB
Aug 2010 75 87 207 1,096 17.37 MB

...
```

```
Aug 2018 407,512 642,542 1,223,319 7,829,987 472.37 GB
Sep 2018 463,937 703,327 1,187,224 8,468,723 514.46 GB
Oct 2018 537,343 826,290 1,403,176 10,013,025 620.55 GB
Nov 2018 514,072 781,335 1,295,594 9,487,834 642.16 GB
Dec 2018 464,763 682,741 1,209,298 8,348,946 544.44 GB
```

Das Einlesen dieser Datei können wir mittels `read_csv` bewerkstelligen. Die Daten sind durch Whitespaces getrennt. Damit auch eine Folge von Whitespaces als ein Delimiter erkannt wird, benutzen wir einen regulären Ausdruck, d.h. `r'\s+'`. Die Tausenderstellen sind mit "," getrennt.

```
import pandas as pd

data_path = 'data1/'
fname = 'python_course_monthly_history.txt'
webstats = pd.read_csv(data_path + fname,
                       quotechar='"',
                       thousands=',',
                       delimiter=r'\s+')
# umbenennen der Spaltennamen:
webstats.columns = ['Month', 'Year', 'Visitors', 'Visits',
                    'Pages', 'Hits', 'Bandwidth', 'Unit']

webstats.head()
```

	Month	Year	Visitors	Visits	Pages	Hits	Bandwidth	Unit
0	Jun	2010	11	13	42	290	2.63	MB
1	Jul	2010	27	39	232	939	9.42	MB
2	Aug	2010	75	87	207	1096	17.37	MB
3	Sep	2010	171	221	480	2373	39.63	MB
4	Oct	2010	238	301	552	2872	52.13	MB

Wir erkennen, dass die Angaben für die Bandbreite (Bandwidth) nicht in einer festen Einheit angegeben sind, sondern von der letzten Spalte abhängen. Sie können in Bytes, Megabytes oder in Gigabytes sein. Die Funktion `unit_convert` wandelt ein Tupel, bestehend aus einem Wert und einer Einheit, in einen Bytewert um. Wir wenden diese Funktion nun auf die beiden letzten Spalten an. Der Parameter `axis` muss auf 1 gesetzt sein, damit die Funktion jeweils auf einen Wert und eine Einheit angewendet wird:

```
def unit_convert(x):
    value, unit = x
    if unit == 'MB':
        value *= 1024
    elif unit == 'GB':
        value *= 1048576 # i.e. 1024 **2
    return value

cols = ['Bandwidth', 'Unit']
bandwidth = webstats[cols].apply(unit_convert,
                                 axis=1)
```

Als Nächstes löschen wir die `'unit'`-Spalte und ersetzen die `'Bandwidth'`-Spalte mit den neu berechneten Werten `bandwidth`. Außerdem ersetzen wir die Monatsangaben durch einen neuen Stringwert, bestehend aus Monat und Jahr. Die Spalte `Year` löschen wir dann.

```
del webstats['Unit']
webstats['Bandwidth'] = bandwidth

def concat(x):
    """concatenates month and year"""
    return x[0] + " " + str(x[1])

month_year = webstats[['Month', 'Year']]
month_year = month_year.apply(concat,
                              axis=1)
webstats['Month'] = month_year
del webstats['Year']

webstats.set_index('Month', inplace=True)
del webstats['Bandwidth']


webstats[:10]
```

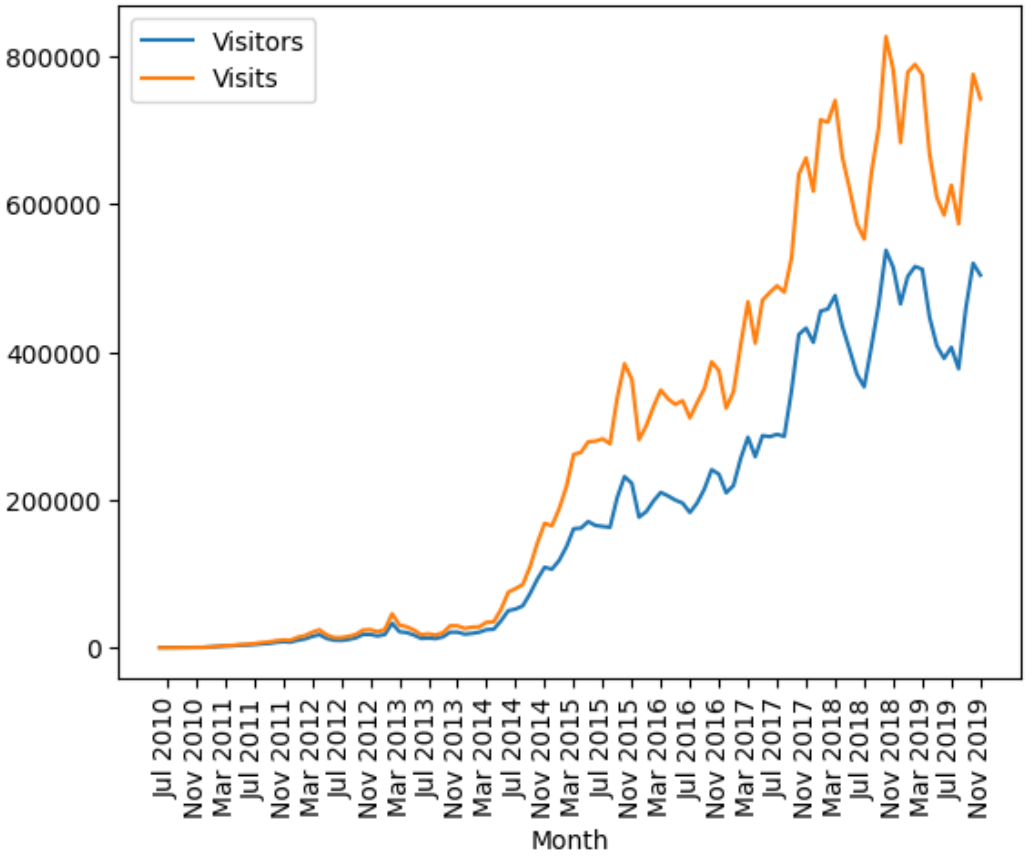
```
/tmp/ipykernel_55009/2292131772.py:6: FutureWarning: Series.__getitem__ treating
↪  keys as positions is deprecated. In a future version, integer keys will
↪  always be treated as labels (consistent with DataFrame behavior). To access a
↪  value by position, use `ser.iloc[pos]`
  return x[0] + " " + str(x[1])
```

Month	Visitors	Visits	Pages	Hits
Jun 2010	11	13	42	290
Jul 2010	27	39	232	939
Aug 2010	75	87	207	1096
Sep 2010	171	221	480	2373
Oct 2010	238	301	552	2872
Nov 2010	353	455	912	5086
Dec 2010	366	423	944	4884
Jan 2011	911	1111	2321	11442
Feb 2011	1488	1807	4139	22291
Mar 2011	2128	2762	6009	32407

Nun sind wir bereit für den Plot des DataFrames:

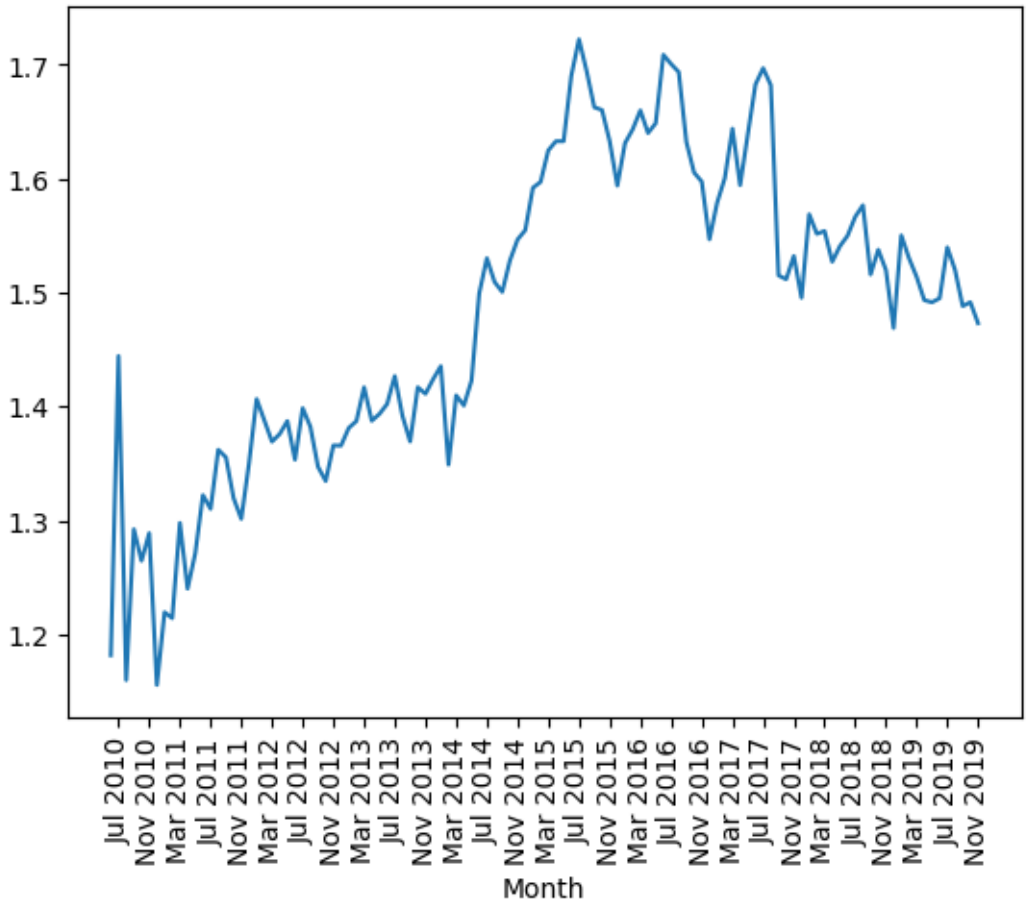
```
xticks = range(1, len(webstats.index), 4)
webstats[['Visitors', 'Visits']].plot(use_index=True,
                                      rot=90,
                                      xticks=xticks)

plt.plot()
```

```
[]
```



Wir berechnen nun die durchschnittliche Anzahl der Besuche pro Besucher.

```
ratio = pd.Series(webstats["Visits"] / webstats["Visitors"],
                  index=webstats.index)
ratio.plot(use_index=True,
           xticks=range(1, len(ratio.index), 4),
           rot=90)
plt.show()
```



Spalten mit Zeichenketten (Strings) in Floats wandeln Im Verzeichnis data1 haben wir die Datei

tiobe_programming_language_usage_nov2018.txt

mit einem Nutzungs-Ranking von Programmiersprachen. Die Daten wurden gesammelt und aufbereitet von TIOBE im November 2018.

Die Datei hat folgenden Inhalt:

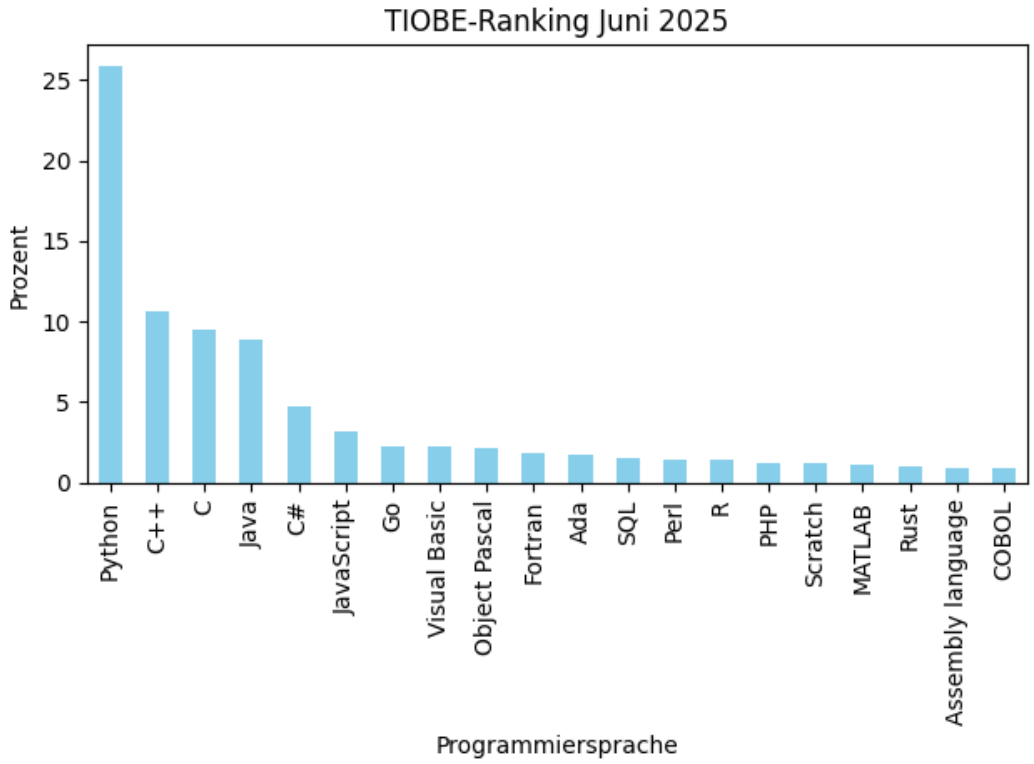
Die Prozent-Spalte enthält Strings mit dem Prozent-Zeichen. Das Zeichen können wir nutzen (oder loswerden), indem wir die Funktion `read_csv` verwenden. Alles, was wir dafür tun müssen, ist, eine Konverter-Funktion zu definieren. Diese Konverter-Funktion übergeben wir dann an `read_csv` mit dem Parameter `converters`.

```
import pandas as pd
def strip_percentage_sign(x):
    return float(x.strip('%'))
data_path = "data1/"
fname = "tiobe_programming_language_usage_jun2025.csv"
progs = pd.read_csv(
    data_path + fname,
    quotechar="'",
    index_col=2,
    converters={'Ratings': strip_percentage_sign,
                'Change': strip_percentage_sign},
    delimiter=r",")
```

```
print(progs)
```

Programmiersprache	Jun 2024	Jun 2025	Ratings	Change
Python	1	1	25.87	10.48
C++	2	2	10.68	0.65
C	3	3	9.47	0.24
Java	4	4	8.84	0.44
C#	5	5	4.69	-1.96
JavaScript	6	6	3.21	-0.11
Go	7	7	2.28	0.35
Visual Basic	8	9	2.20	0.54
Object Pascal	9	11	2.15	0.62
Fortran	10	10	1.86	0.33
Ada	11	25	1.70	0.91
SQL	12	8	1.55	-0.21
Perl	13	27	1.47	0.77
R	14	21	1.39	0.43
PHP	15	15	1.25	0.03
Scratch	16	16	1.19	0.02
MATLAB	17	14	1.13	-0.13
Rust	18	17	0.97	-0.20
Assembly language	19	13	0.91	-0.35
COBOL	20	20	0.89	-0.08

```
progs['Ratings'].plot(kind="bar",
                      use_index=True,
                      rot=90,
                      color='skyblue')

plt.ylabel("Prozent")
plt.title("TIOBE-Ranking Juni 2025")
plt.tight_layout()
```



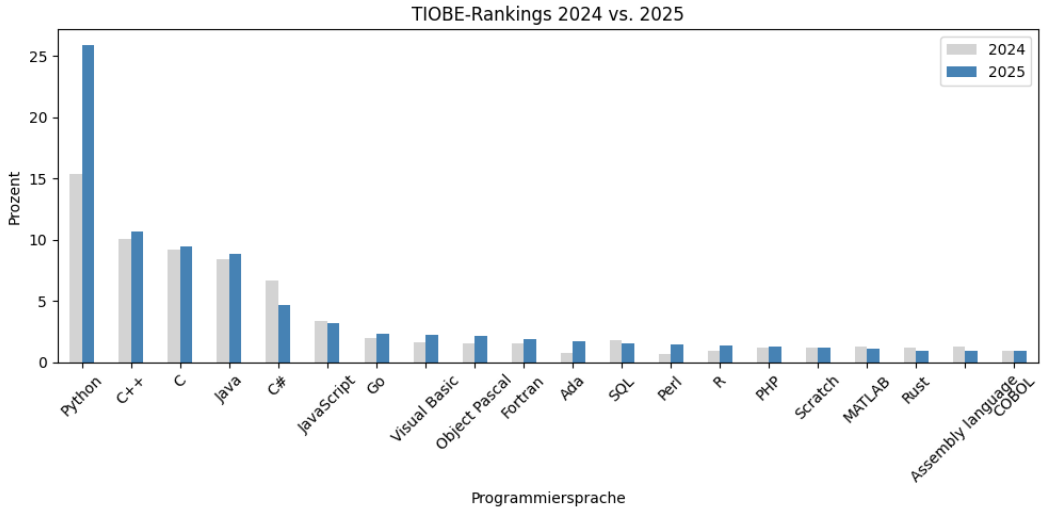

```
# Rating 2024 berechnen
progs["Ratings2024"] = progs["Ratings"] - progs["Change"]
progs
```

Programmiersprache	Jun 2024	Jun 2025	Ratings	Change	Rating2024	\
Python	1	1	25.87	10.48	15.39	
C++	2	2	10.68	0.65	10.03	
C	3	3	9.47	0.24	9.23	
Java	4	4	8.84	0.44	8.40	
C#	5	5	4.69	-1.96	6.65	
JavaScript	6	6	3.21	-0.11	3.32	
Go	7	7	2.28	0.35	1.93	
Visual Basic	8	9	2.20	0.54	1.66	
Object Pascal	9	11	2.15	0.62	1.53	
Fortran	10	10	1.86	0.33	1.53	
Ada	11	25	1.70	0.91	0.79	
SQL	12	8	1.55	-0.21	1.76	
Perl	13	27	1.47	0.77	0.70	
R	14	21	1.39	0.43	0.96	
PHP	15	15	1.25	0.03	1.22	
Scratch	16	16	1.19	0.02	1.17	
MATLAB	17	14	1.13	-0.13	1.26	
Rust	18	17	0.97	-0.20	1.17	
Assembly language	19	13	0.91	-0.35	1.26	
COBOL	20	20	0.89	-0.08	0.97	

Programmiersprache	Ratings2024
Python	15.39
C++	10.03
C	9.23

```
Java                    8.40
C#                      6.65
JavaScript              3.32
Go                      1.93
Visual Basic            1.66
Object Pascal           1.53
Fortran                 1.53
Ada                     0.79
SQL                     1.76
Perl                    0.70
R                       0.96
PHP                     1.22
Scratch                 1.17
MATLAB                  1.26
Rust                    1.17
Assembly language       1.26
COBOL                   0.97
```

```python
# Neue DataFrame mit beiden Jahren
ratings_df = progs[["Ratings2024", "Ratings"]].copy()
ratings_df.columns = ["2024", "2025"]

# Optional: sortieren nach 2025
ratings_df = ratings_df.sort_values("2025", ascending=False)

# Balkendiagramm
ax = ratings_df.plot(kind="bar",
                     rot=45,
                     color=["lightgray", "steelblue"],
                     figsize=(10, 5))

ax.set_ylabel("Prozent")
ax.set_title("TIOBE-Rankings 2024 vs. 2025")
fig = ax.get_figure()
fig.tight_layout()
```



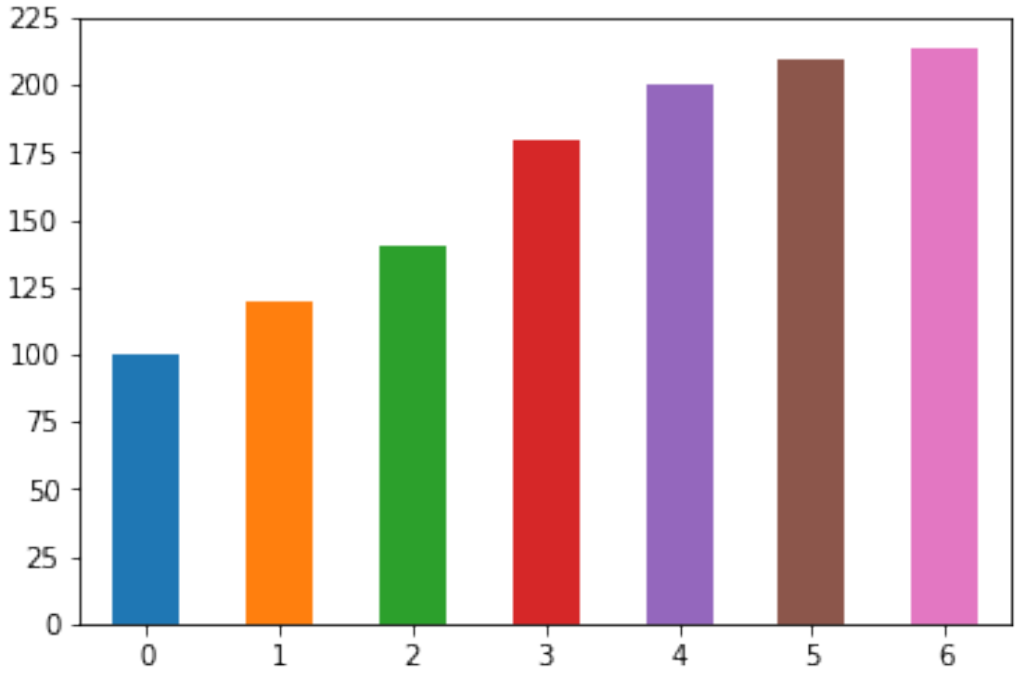
Balkendiagramme in Pandas

Balkendiagramme mit Pandas zu erstellen, ist ebenso einfach wie Liniendiagramme. Dazu fügen wir den Schlüsselwort-Parameter `kind` zu plot-Methode hinzu und setzen den Wert auf `bar`.

Ein einfaches Beispiel

```
import pandas as pd

data = [100, 120, 140, 180, 200, 210, 214]
s = pd.Series(data, index=range(len(data)))

s.plot(kind="bar", rot=0)
plt.plot()
```

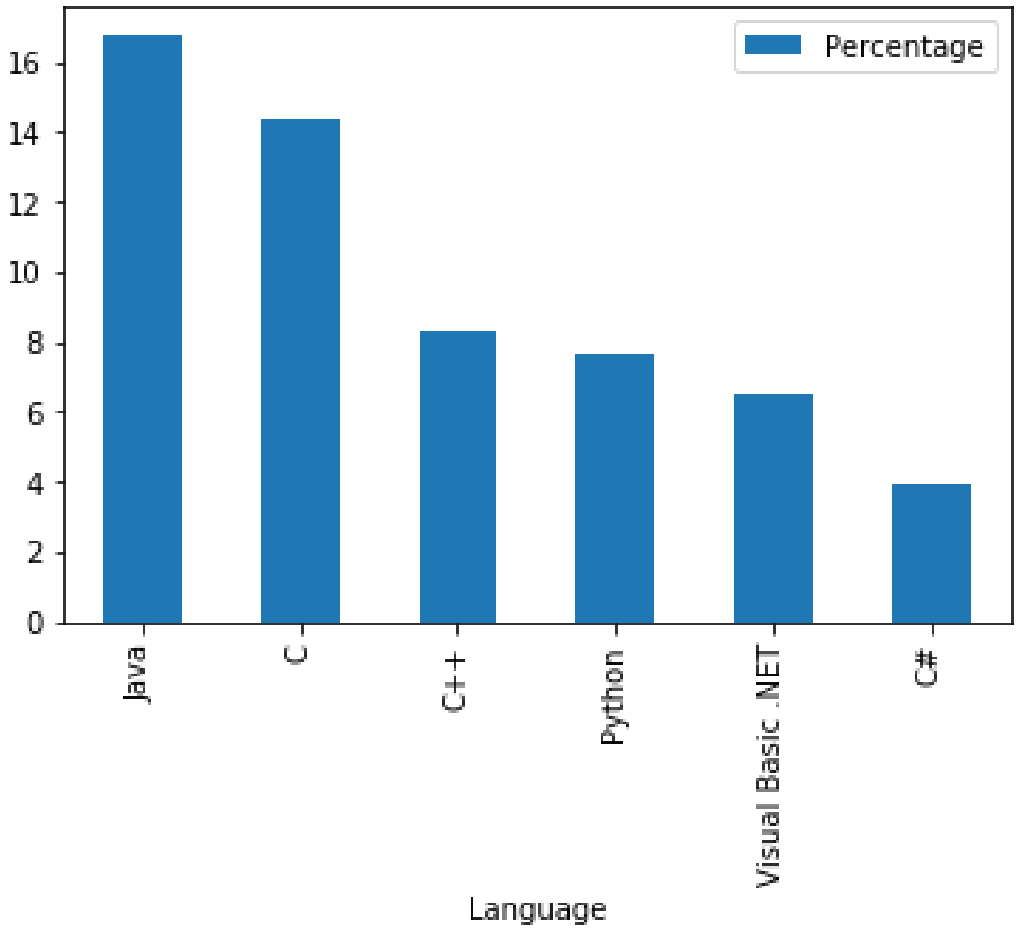
```
[]
```



Balken-Grafik für die Programmiersprachennutzung Wir gehen zurück zum Beispiel des Programmiersprachen-Rankings. Jetzt generieren wir eine Balkengrafik der sechs meistverwendeten Programmiersprachen:

```
progs[:6].plot(kind="bar")
```

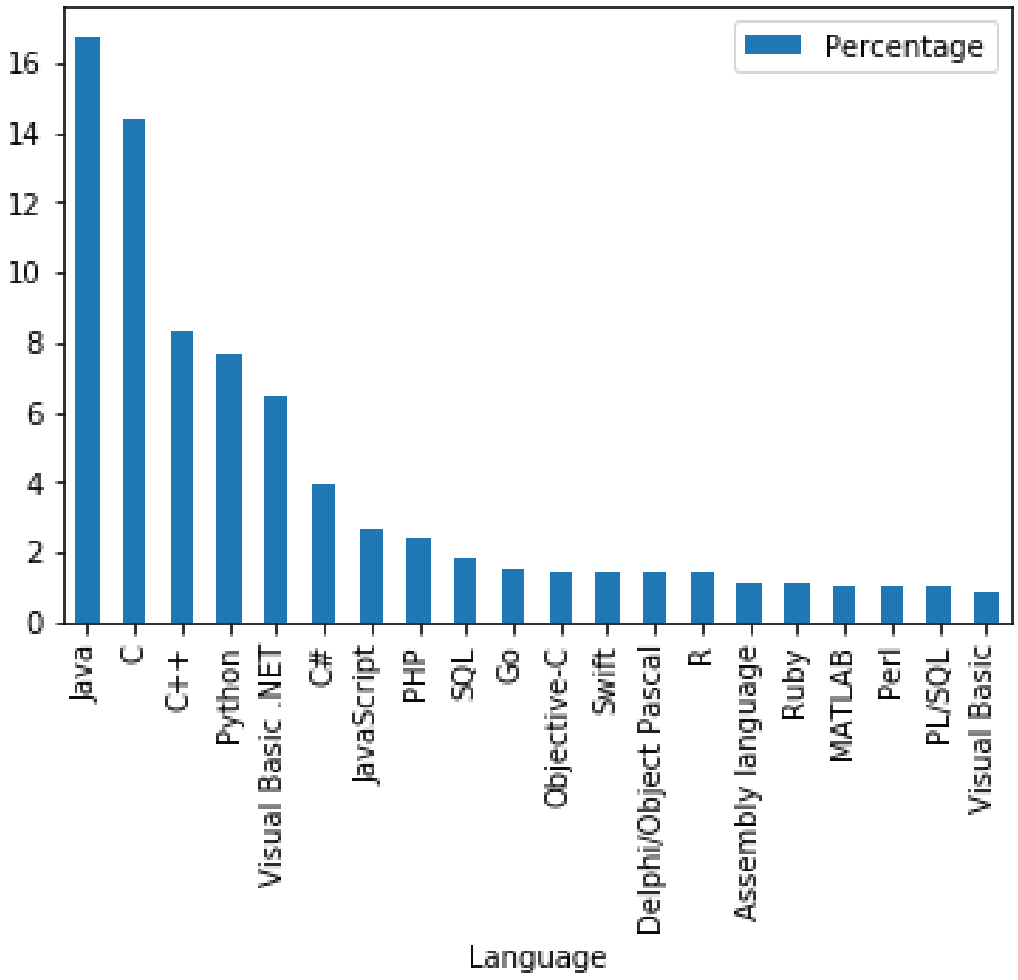
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7339668748>
```



Nun die Balkendiagramme mit allen Programmiersprachen:

```
progs.plot(kind="bar")
```

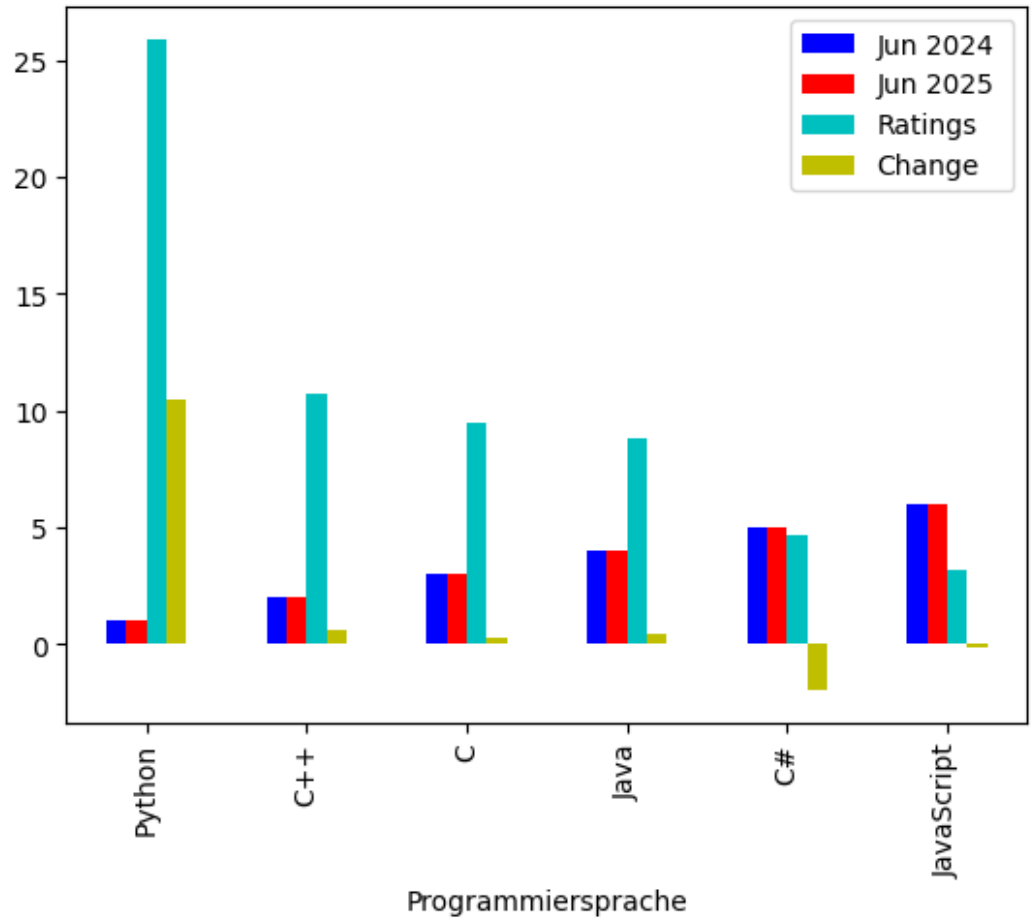
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f73397f5668>
```



Farbgebung einer Balken-Grafik Es ist möglich, die Balken individuell zu färben, indem eine Liste dem Schlüsselwort-Parameter `color` zugewiesen wird:

```
my_colors = ['b', 'r', 'c', 'y', 'g', 'm']
progs[:6].plot(kind="bar",
               color=my_colors)
```

```
<Axes: xlabel='Programmiersprache'>
```

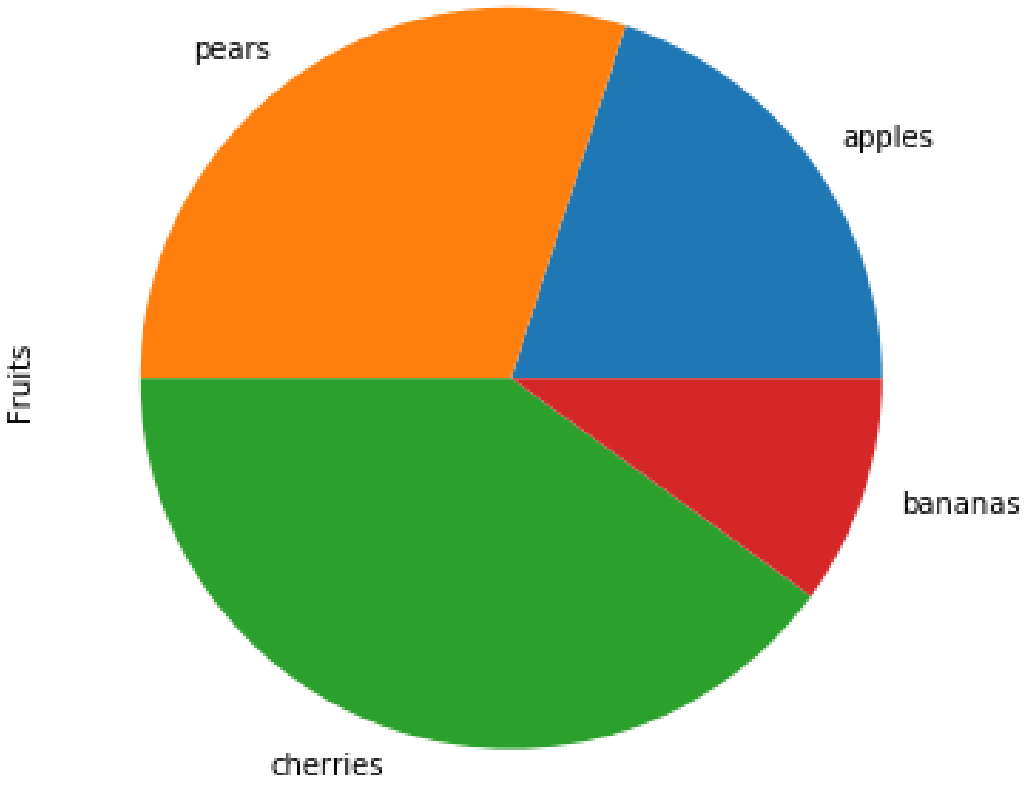


Kuchen-Diagramme in Pandas

Ein einfaches Beispiel

```
import pandas as pd

fruits = ['apples', 'pears', 'cherries', 'bananas']
series = pd.Series([20, 30, 40, 10],
                   index=fruits,
                   name='Fruits')

series.plot.pie(figsize=(6, 6))
```

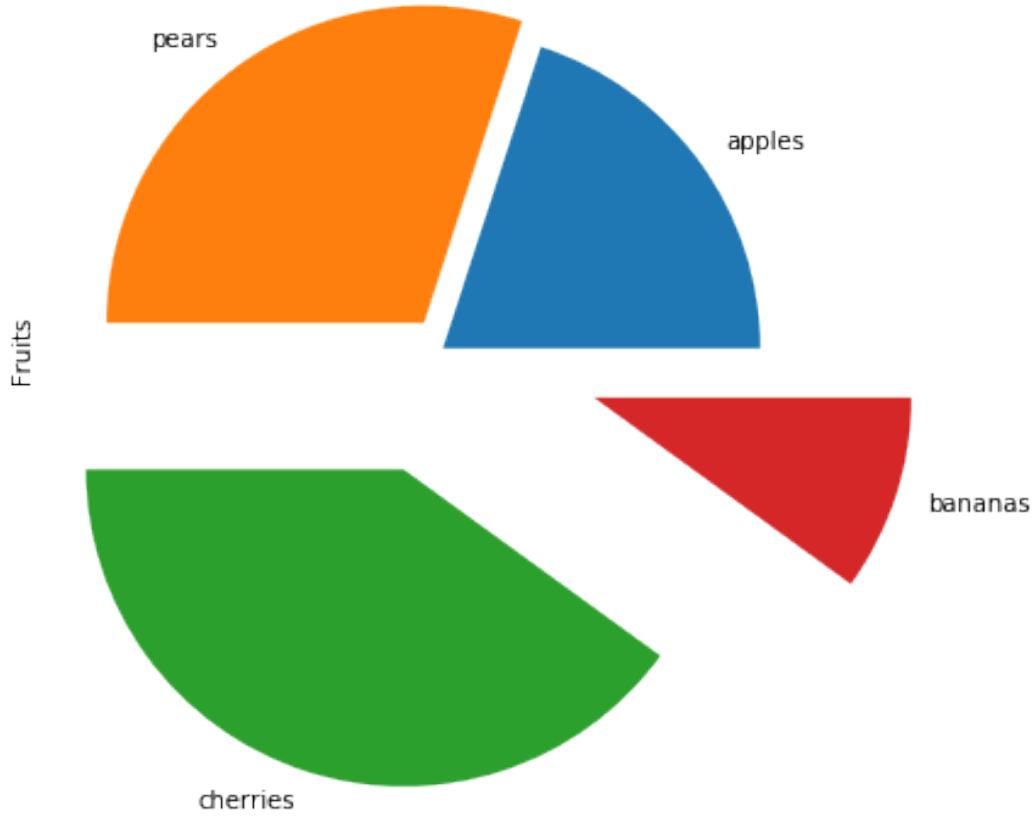
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7338e9e588>
```



Obiges Kreisdiagramm lässt einen an einen Kuchen denken, und wie bei einem Kuchen, kann man einzelne "Stücke" herausziehen, was sich mit dem Parameter `explode` bewerkstelligen lässt. Diesem kann man eine Array-ähnliche Struktur, wie z.B. ein Tupel oder Liste, übergeben, die den Abstand vom Kreismittelpunkt beschreibt. Die Default-Werte sind 0, d.h. die "Kuchenstücke" sind nicht herausgezogen:

```
fruits = ['apples', 'pears', 'cherries', 'bananas']

series = pd.Series([20, 30, 40, 10],
                   index=fruits,
                   name='Fruits')
explode = [0, 0.10, 0.40, 0.5]
series.plot.pie(figsize=(6, 6),
                explode=explode)
```

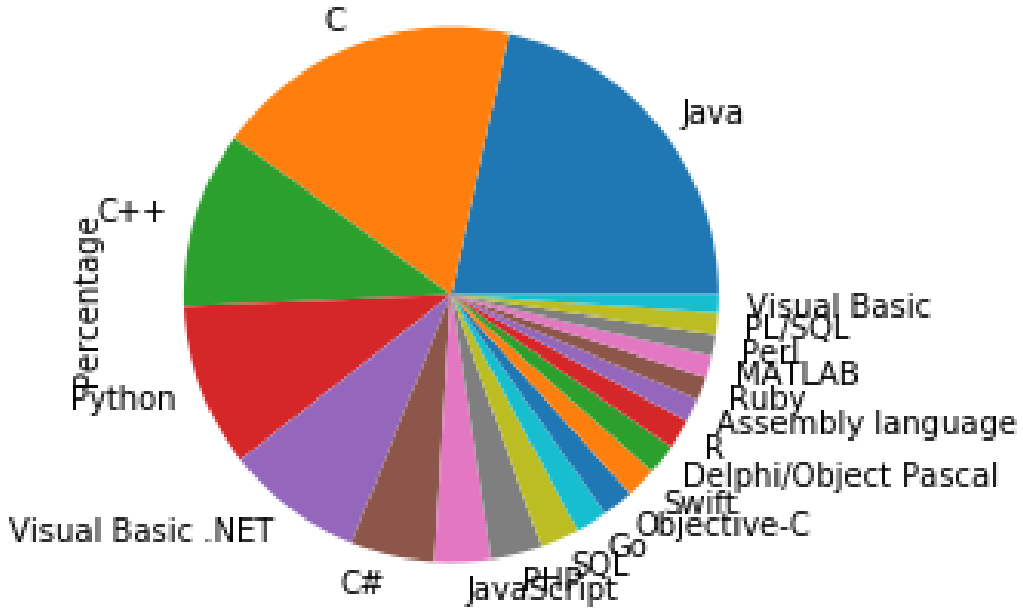
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7338e5efd0>
```



Wir generieren die vorherige Balken-Grafik (Programmiersprachen) nun als Kuchendiagramm:

```
import matplotlib.pyplot as plt

my_colors = ['b', 'r', 'c', 'y', 'g', 'm']
progs.plot.pie(subplots=True,
               legend=False)
```

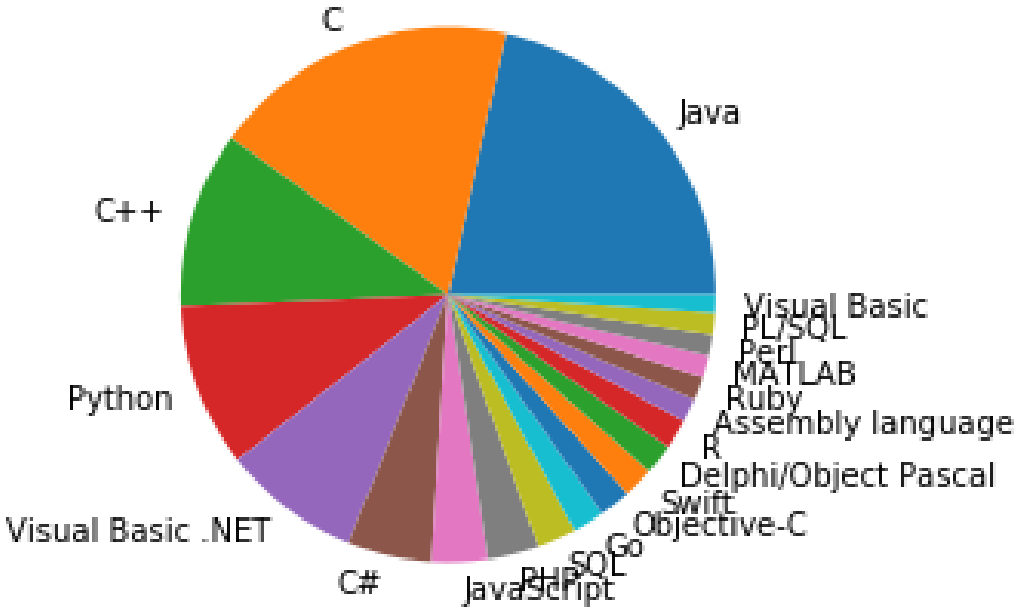
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f7338e1a9e8>],
      dtype=object)
```



Es ist nicht schön, dass das y-Label `Percentage` innerhalb der Grafik angezeigt wird. Wir entfernen die Bezeichnung mit der Funktion `plt.ylabel`.

```
import matplotlib.pyplot as plt

my_colors = ['b', 'r', 'c', 'y', 'g', 'm']
progs.plot.pie(subplots=True,
               legend=False)

plt.ylabel('')
```

```
Text(0, 0.5, '')
```



Alle Alternativen für den Parameter `kind` im Überblick:

- `line`: Liniendiagramm (Default-Wert)
- `bar`: Säulendiagramm
- `hist`: Histogramm
- `box`: Kastengrafik (engl. Box Plot)
- `kde`: Kernel Density Estimation plot
- `density`: wie kde
- `area`: Flächendiagramm
- `pie`: Kreisdiagramm

28 Python3 Time And Date

```
# invisible
import numpy as np
np.core.arrayprint._line_width = 65
```

28.0.1 Python, Datum und Zeit

Python Standard-Module für Zeit-Daten

Python bietet reichlich Funktionalität, um mit Datums- und Zeit-Daten umzugehen. Die Standard-Bibliotheken enthalten folgende Module:

- `time`
- `calendar`
- `datetime`

Diese Module bieten Klassen zur Manipulation von simplen und komplexen Datums- und Zeit-Daten.

Speziell die `datetime`-Klasse ist sehr wichtig für die Timeseries in Pandas.

Die wichtigsten Module in Python, um mit Zeiten zu arbeiten, sind `time`, `calendar` und `datetime`.

Das `datetime`-Modul bietet diverse Klassen, Methoden und Funktionen für die Arbeit mit Daten, Zeiten und Zeit-Intervallen. Dafür stehen folgende Klassen zur Verfügung:

- Die Instanzen der `date`-Klasse können Daten innerhalb der Jahre zwischen 1 und 9999 abbilden.
- Eine Instanz der `datetime`-Klasse besteht sowohl aus dem Datum, als auch aus der Zeit.
- Die `time`-Klasse implementiert Zeit-Objekte.
- Die `timedelta`-Klasse wird verwendet zur Differenzbildung zwischen zwei Zeit- oder Datums-Objekten.
- Die `tzinfo`-Klasse dient der Implementierung von Zeitzonen für Zeit- und Datums-Objekten.

Wir starten mit dem date-Objekt.

Die date-Klasse

```
from datetime import date

x = date(1993, 12, 14)
print(x)
```

```
1993-12-14
```

Wir können Datums-Objekte zwischen dem 01. Januar 0001 und dem 31. Dezember 9999 instanziieren. Über die Attribute `min` und `max` kann dies ermittelt werden:

```
from datetime import date

print(date.min)
print(date.max)
```

```
0001-01-01
9999-12-31
```

Wir können diverse Methoden auf das obige `date`-Objekt anwenden. Das proleptische Gregorianische Ordinal liefert die Methode `toordinal`. Der proleptische gregorianische Kalender besteht aus der Erweiterung des gregorianischen Kalenders rückwärts über seine Einführung im Jahre 1582 hinaus. In der Ordinalen Numerierung entspricht somit dem Tag 1 der 1. Januar des Jahres 1:

```
x = date(1, 1, 1) # 1. Januar 1
print(x.toordinal())
x = date(1, 1, 2) # 2. Januar 1
print(x.toordinal())
print(x.today())
print(x.today().toordinal())
```

```
1
2
2019-02-21
737111
```

Aus einem Ordinal kann das Datum mit der Klassen-Methode `fromordinal` wieder herausgerechnet werden:

```
print(date.fromordinal(726952))
```

```
1991-04-30
```

Wenn Sie den Wochentag eines bestimmten Tages wissen möchten, kann dies mit der Methode `weekday` berechnet werden. `weekday` liefert Zahlen zwischen `0` (Montag) und `6` (Sonntag) zurück.

```
print(x.weekday())
```

```
1
```

```
print(date.today())
```

```
2019-02-21
```

Über die Attribute können wir auf den Tag, den Monat und das Jahr eines Date-Objektes zugreifen:

```
print(x.day)
```

```
print(x.month)
print(x.year)
```

```
2
1
1
```

Die time-Klasse Die `time`-Klasse ist gleich organisiert wie die `date`-Klasse.

```
from datetime import time

t = time(15, 6, 23)
print(t)
```

```
15:06:23
```

Die möglichen Zeiten liegen zwischen:

```
print(time.min)
print(time.max)
```

```
00:00:00
23:59:59.999999
```

Der Zugriff auf Stunde, Minute und Sekunde:

```
print(t.hour, t.minute, t.second)
```

```
15 6 23
```

Jede Komponente eines Zeit-Objektes kann durch `replace()` geändert werden:

```
t = t.replace(hour=11, minute=59)
print(t)
```

```
11:59:23
```

Wir können einen Datums-String in C-Style generieren, entsprechend der `ctime`-Funktion in C:

```
print(x.ctime())
```

```
Tue Jan  2 00:00:00 0001
```

Die datetime-Klasse

Das `datetime`-Modul bietet uns Funktionen und Methoden zur Manipulation von Datums- und Zeit-Objekten. Weiterhin stellt es Funktionalitäten zur Verfügung für arithmetische Operationen für Datums- und Zeit-Objekte, z.B. Addition und Subtraktion. Ein weiterer Fokus der Implementierung liegt auf der Extrahierung von Attributen.

Es gibt zwei Arten von Datums- und Zeit-Objekten:

- naive
- aware

Wenn ein Zeit-Objekt 'naive' ist, enthält es keine Informationen für den Vergleich oder die Lokalisation gegenüber anderen Datums- oder Zeit-Objekten. Die Semantik, falls das 'naive'-Objekt einer bestimmten Zeitzone entspricht (wie beispielsweise UTC, lokale Zeit, etc.), ist in der Logik des Programs verankert.

Auf der anderen Seite hat ein 'aware'-Objekt Informationen über die Zeitzone. Somit kann es gegenüber anderen 'aware'-Objekten lokalisiert werden.

Wie können Sie herausfinden, ob ein datetime-Objekt `t` 'aware' ist?

`t` ist 'aware', wenn `t.tzinfo` nicht None ist und `t.tzinfo.utcoffset(t)` nicht None ist. Beide Bedingungen müssen erfüllt sein.

Demgegenüber ist das Objekt `t` 'naive', wenn `t.tzinfo` oder `t.tzinfo.utcoffset(t)` None ist.

Erstellen wir ein datetime-Objekt:

```
from datetime import datetime
t = datetime(2017, 4, 19, 16, 31, 0)
print(t)
```

```
2017-04-19 16:31:00
```

`t` ist naive, weil folgender Ausdruck `True` ist:

```
print(t.tzinfo == None)
```

```
True
```

Wir erstellen ein 'aware' datetime-Objekt vom aktuellen Datum. Dafür benötigen wir das Modul `pytz`. `pytz` ist ein Modul, welches die 'Olsen-Zeitzonen-Datenbank' in Python bereitstellt. Die Olsen-Zeitzonen werden nahezu komplett durch dieses Modul unterstützt.

```
from datetime import datetime
import pytz
t = datetime.now(pytz.utc)
```

Wir sehen, dass sowohl `t.tzinfo` als auch `t.tzinfo.utcoffset(t)` nicht `None` sind und `t` somit ein 'aware'-Objekt ist:

```
print(t.tzinfo, t.tzinfo.utcoffset(t))
```

```
UTC 0:00:00
```

```
from datetime import datetime, timedelta as delta
ndays = 15
start = datetime(1991, 4, 30)
dates = [start - delta(days=x) for x in range(0, ndays)]
dates
```

```
[datetime.datetime(1991, 4, 30, 0, 0),
 datetime.datetime(1991, 4, 29, 0, 0),
 datetime.datetime(1991, 4, 28, 0, 0),
 datetime.datetime(1991, 4, 27, 0, 0),
 datetime.datetime(1991, 4, 26, 0, 0),
 datetime.datetime(1991, 4, 25, 0, 0),
 datetime.datetime(1991, 4, 24, 0, 0),
 datetime.datetime(1991, 4, 23, 0, 0),
 datetime.datetime(1991, 4, 22, 0, 0),
 datetime.datetime(1991, 4, 21, 0, 0),
 datetime.datetime(1991, 4, 20, 0, 0),
 datetime.datetime(1991, 4, 19, 0, 0),
 datetime.datetime(1991, 4, 18, 0, 0),
 datetime.datetime(1991, 4, 17, 0, 0),
 datetime.datetime(1991, 4, 16, 0, 0)]
```

Unterschied zwischen Zeiten

Schauen wir was passiert, wenn wir `datetime`-Objekte voneinander subtrahieren:

```
from datetime import datetime

delta = datetime(1993, 12, 14) - datetime(1991, 4, 30)
print(delta, type(delta))
```

```
959 days, 0:00:00 <class 'datetime.timedelta'>
```

Das Ergebnis der Subtraktion der beiden `datetime`-Objekte ist ein `timedelta`-Objekt. Über das Attribut `days` können wir die Tage der Differenz auslesen:

```
print(delta.days)
```

```
959
```

```
t1 = datetime(2017, 1, 31, 14, 17)
t2 = datetime(2015, 12, 15, 16, 59)
delta = t1 - t2
print(delta.days, delta.seconds)
```

```
412 76680
```

Es ist möglich, ein `timedelta`-Objekt von einem anderen `datetime`-Objekt zu subtrahieren oder es zu addieren (in Tagen), um ein neues `datetime`-Objekt zu berechnen:

```
from datetime import datetime, timedelta
d1 = datetime(1991, 4, 30)
d2 = d1 + timedelta(10)
print(d2)
print(d2 - d1)

d3 = d1 - timedelta(100)
print(d3)
d4 = d1 - 2 * timedelta(50)
```

```
print(d4)
```

```
1991-05-10 00:00:00
10 days, 0:00:00
1991-01-20 00:00:00
1991-01-20 00:00:00
```

Ebenso können `timedelta`-Objekte auch in Tagen und Minuten zu `datetime`-Objekten addiert oder voneinander subtrahiert werden:

```
from datetime import datetime, timedelta
d1 = datetime(1991, 4, 30)
d2 = d1 + timedelta(10,100)
print(d2)
print(d2 - d1)
```

```
1991-05-10 00:01:40
10 days, 0:01:40
```

Wandlung von datetime-Objekten in Strings Der einfachste Weg ein `datetime`-Objekt als String darzustellen ist die `str`-Methode.

```
s = str(d1)
print(s)
```

```
1991-04-30 00:00:00
```

Wandlung mit strftime Der Methodenaufruf `datetime.strftime(format)` liefert einen String zurück, welcher die Zeit und das Datum repräsentiert, jedoch durch ein explizites Format bestimmt wird. Ein komplette Liste von möglichen Formatierungen kann hier (strftime) eingesehen werden:

```
print(d1.strftime('%Y-%m-%d'))
print("Wochentag: " + d1.strftime('%a'))
print("Wochentag ausgeschrieben: " + d1.strftime('%A'))

# Weekday as a decimal number, where 0 is Sunday
# and 6 is Saturday
print("Wochentag als Dezimalzahl: " + d1.strftime('%w'))
```

```
1991-04-30
Wochentag: Tue
Wochentag ausgeschrieben: Tuesday
Wochentag als Dezimalzahl: 2
```

Formatierung von Monaten:

```
# Day of the month as a zero-padded decimal number.
# 01, 02, ..., 31
print(d1.strftime('%d'))

# Month as locale's abbreviated name.
```

```
# Jan, Feb, ..., Dec (en_US);
# Jan, Feb, ..., Dez (de_DE)
print(d1.strftime('%b'))

# Month as locale's full name.
# January, February, ..., December (en_US);
# Januar, Februar, ..., Dezember (de_DE)
print(d1.strftime('%B'))

# Month as a zero-padded decimal number.
# 01, 02, ..., 12
print(d1.strftime('%m'))
```

```
30
Apr
April
04
```

Ausgabe in Landessprache

Wir haben bereits gesehen, dass die Datumsausgaben in Englisch erfolgt sind. Im Folgenden geben wir ein Datum auf die im Vereinigten Königreich gebräuchlichste Art aus:

```
from datetime import datetime, timedelta
d1 = datetime(1993, 12, 14)

print(d1.strftime('%d %B %Y'))

print("Nur in Zahlen:")
print(d1.strftime('%d/%m/%Y'))
print("Auf US Art:")
print(d1.strftime('%m/%d/%Y'))

print(f"It was a {d1.strftime('%A'):s}!")
```

```
14 December 1993
Nur in Zahlen:
14/12/1993
Auf US Art:
12/14/1993
It was a Tuesday!
```

Eine häufig gestellte Frage lautet, wie man diese Ausgaben in Landessprache ausgeben kann. Zunächst ist es wichtig, das `locale`-Modul zu importieren:

```
from datetime import datetime, timedelta
import locale

# Umstellung auf Deutsch:
locale.setlocale(locale.LC_ALL, 'de_DE.utf8')
d1 = datetime(1993, 12, 14)

print(d1.strftime('%d. %B %Y'))

print("Nur in Zahlen:")
```

```
print(d1.strftime('%d.%m.%Y'))

print(f"Der {d1.strftime('%d.%m.%Y'):s} war ein {d1.strftime('%A'):s}!")

# und nun in Französisch:

locale.setlocale(locale.LC_ALL, 'fr_FR.utf8')
d1 = datetime(1993, 12, 14)

print(d1.strftime('%d %B %Y'))

print("Seulement en nombre:")
print(d1.strftime('%d.%m.%Y'))

print(f"Le {d1.strftime('%d %m %Y'):s} était un {d1.strftime('%A'):s}!")
```

```
14. Dezember 1993
Nur in Zahlen:
14.12.1993
Der 14.12.1993 war ein Dienstag!
14 décembre 1993
Seulement en nombre:
14.12.1993
Le 14 12 1993 était un mardi!
```

Anmerkung:

Die länderspezifischen Ausgaben der obigen Beispiele funktionieren nur, falls 'de_DE.utf8' und 'fr_FR.utf8' im Betriebssystem installiert sind. Unter Ubuntu kann man diese wie folgt installieren:

`sudo locale-gen fr_FR.UTF-8`

und

`sudo locale-gen de_DE.UTF-8`

datetime-Objekte aus Strings erstellen

Wir können `strptime` nutzen, um neue `datetime`-Objekte aus Strings zu erstellen, welche Datum und Zeit beinhalten. Die Argumente von `strptime` sind der String, welcher geparst werden soll, und die Format-Spezifikation.

```
from datetime import datetime
t = datetime.strptime("30 12 1999", "%d %m %Y")
print(t)
```

```
1999-12-30 00:00:00
```

```
dt = "2007-03-04T21:08:12"
datetime.strptime( dt, "%Y-%m-%dT%H:%M:%S" )
```

```
datetime.datetime(2007, 3, 4, 21, 8, 12)
```

```
import locale
```

```
locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')

dt = '12/24/1957 4:03:29 AM'
dt = datetime.strptime(dt, '%m/%d/%Y %I:%M:%S %p')
dt
```

```
datetime.datetime(1957, 12, 24, 4, 3, 29)
```

Auf einer Linux-Maschine können wir einen englischen Datumsstring generieren mit dem Kommando `LC_ALL=en_EN.utf8 date`.

```
dt = 'Wed Apr 12 20:29:53 CEST 2017'
dt = datetime.strptime(dt, '%a %b %d %H:%M:%S %Z %Y')
print(dt)
```

```
2017-04-12 20:29:53
```

Obwohl `datetime.strptime()` eine einfache Möglichkeit ist, ein Datum mit einem bekannten Format zu parsen, kann es doch kompliziert sein, für neue Datumsformate jedes Mal eine neue Spezifikation zu erstellen.

Für das Parsen ist die Nutzung der Methode `dateutil.parser` besser:

```
from dateutil.parser import parse

parse('2011-01-03')
```

```
datetime.datetime(2011, 1, 3, 0, 0)
```

```
parse('Wed Apr 12 20:29:53 CEST 2017')
```

```
datetime.datetime(2017, 4, 12, 20, 29, 53, tzinfo=tzlocal())
```

29 Pandas Time Series

```
# invisible
import numpy as np
np.core.arrayprint._line_width = 65
```

29.0.1 Python, Pandas und Zeitserien

Einführung

In unserem nächsten Kapitel des Pandas-Tutorial behandeln wir Time Series. Eine Time Series ist eine Reihe von Datenpunkten, welche in chronologischer (zeitlicher) Reihenfolge gelistet (indiziert) sind. Für gewöhnlich ist eine Time Series eine Sequenz von Werten, mit gleichen zeitlichen Abständen.

Alle gemessenen Daten, die auch mit einem bestimmten Zeitpunkt in Verbindung stehen, können als Time Series angesehen werden. Messungen können durchaus unregelmäßig sein, haben aber in den meisten Fällen eine feste Frequenz bzw. Regelmässigkeit. D.h. dass Daten bspw. alle 5 Millisekunden, alle 10 Sekunden oder jede Stunde erhoben werden. Time Series werden oft in Liniencharts dargestellt.

Bevor Sie fortfahren möchten wir ihnen noch unser Tutorial empfehlen zum Thema Time Processing mit Standard Python-Modulen, wie z.B. datetime, time und calendar.

Wir wollen in diesem Kapitel die Pandas-Tools vorstellen, um mit Time Series umzugehen. Sie werden also lernen, mit großen Time Series zu arbeiten und diese zu modifizieren:

Zeitreihen und Python

Wir können eine Pandas-Series definieren, welche als Index eine Reihe von Zeitstempeln enthält:

```
import numpy as np
import pandas as pd
from datetime import datetime, timedelta as delta

ndays = 10
start = datetime(2018, 12, 1)
dates = [start - delta(days = x) for x in range(0, ndays)]

values = [25, 50, 15, 67, 70, 9, 28, 30, 32, 12]

ts = pd.Series(values, index = dates)
print(ts)
```

```
2018-12-01    25
```

```
2018-11-30    50
2018-11-29    15
2018-11-28    67
2018-11-27    70
2018-11-26     9
2018-11-25    28
2018-11-24    30
2018-11-23    32
2018-11-22    12
dtype: int64
```

Datumsfolgen mit pd.date_range

Mit `pd.date_range()` lassen sich regelmäßige Datums- und Zeitindizes erzeugen. Die wichtigsten Parameter sind `start`, `end`, `periods`, `freq` und `inclusive`.

```
pd.date_range(start="2026-01-01", periods=5, freq="D")
```

Für Monatsenden verwendet aktuelles pandas den Alias `MME"` (*month end*). Ältere Beispiele verwenden häufig `MM"`; dieser Alias wurde in pandas 3.0 entfernt. Analog wurden beispielsweise `"Q"` durch `"QE"` und `"Y"` durch `"YE"` ersetzt.

Mit `inclusive=lleft"`, `rright"`, `"both"` oder `nneither"` lässt sich steuern, welche Randwerte in einem durch `start` und `end` definierten Bereich enthalten sind.

Wir ermitteln den Typ der soeben erstellten Time-Series:

```
print(type(ts))
```

```
<class 'pandas.core.series.Series'>
```

Was wir erzeugt haben, ist eine Zeitreihe oder Time-Series, weil es auf den Series von Pandas basiert. Wie sieht der Index dieser Time-Series aus? Wir sehen es hier:

```
print(ts.index)
```

```
DatetimeIndex(['2018-12-01', '2018-11-30', '2018-11-29', '2018-11-28',
               '2018-11-27', '2018-11-26', '2018-11-25', '2018-11-24',
               '2018-11-23', '2018-11-22'],
              dtype='datetime64[ns]', freq=None)
```

Wir erstellen eine weitere Time-Series:

```
values2 = [32, 54, 18, 61, 72, 19, 21, 33, 29, 17]

ts2 = pd.Series(values2, index=dates)
```

Es ist möglich, arithmetische Operationen auf Zeitreihen durchzuführen, wie bei anderen Series-Objekten auch. Als Beispiel addieren wir die beiden zuvor erstellten Time-Series:

```
print(ts + ts2)
```

```
2018-12-01     57
2018-11-30    104
2018-11-29     33
2018-11-28    128
2018-11-27    142
2018-11-26     28
2018-11-25     49
2018-11-24     63
2018-11-23     61
2018-11-22     29
dtype: int64
```

Arithmetischer Durchschnitt der beiden Series-Objekte:

```
print((ts + ts2) / 2)
```

```
2018-12-01    28.5
2018-11-30    52.0
2018-11-29    16.5
2018-11-28    64.0
2018-11-27    71.0
2018-11-26    14.0
2018-11-25    24.5
2018-11-24    31.5
2018-11-23    30.5
2018-11-22    14.5
dtype: float64
```

Dies kann auch mit Series-Objekten gemacht werden, die eine andere Indexierung haben.

```
import pandas as pd

from datetime import datetime, timedelta as delta

ndays = 10

start = datetime(2018, 6, 1)
dates = [start - delta(days=x) for x in range(0, ndays)]

start2 = datetime(2018, 5, 28)
dates2 = [start2 - delta(days=x) for x in range(0, ndays)]

values = [25, 50, 15, 67, 70, 9, 28, 30, 32, 12]
values2 = [32, 54, 18, 61, 72, 19, 21, 33, 29, 17]

ts = pd.Series(values, index = dates)
ts2 = pd.Series(values2, index = dates2)

print(ts + ts2)
```

```
2018-05-19     NaN
2018-05-20     NaN
2018-05-21     NaN
2018-05-22     NaN
2018-05-23    31.0
2018-05-24   104.0
2018-05-25    91.0
```

```
2018-05-26     46.0
2018-05-27     63.0
2018-05-28    102.0
2018-05-29      NaN
2018-05-30      NaN
2018-05-31      NaN
2018-06-01      NaN
dtype: float64
```

Datumsbereiche erstellen

Die Methode `date_range()` aus dem Pandas-Modul kann für die Erstellung eines Datumsstempel-Index verwendet werden:

```
import pandas as pd

index = pd.date_range('12/24/1970', '01/03/1971')
print(index)
```

```
DatetimeIndex(['1970-12-24', '1970-12-25', '1970-12-26', '1970-12-27',
               '1970-12-28', '1970-12-29', '1970-12-30', '1970-12-31',
               '1971-01-01', '1971-01-02', '1971-01-03'],
              dtype='datetime64[ns]', freq='D')
```

Wir haben ein Start- und ein Ende-Datum an die `date_range`-Methode übergeben. Ebenso ist es möglich, nur einen Start oder nur ein Ende zu übergeben. In diesem Fall muss jedoch die Anzahl der Perioden, über den Schlüsselwort-Parameter `periods`, angegeben werden:

```
index = pd.date_range(start='12/24/1970', periods=7)
print(index)
```

```
DatetimeIndex(['1970-12-24', '1970-12-25', '1970-12-26', '1970-12-27',
               '1970-12-28', '1970-12-29', '1970-12-30'],
              dtype='datetime64[ns]', freq='D')
```

```
index = pd.date_range(end='12/24/1970', periods=7)
print(index)
```

```
DatetimeIndex(['1970-12-18', '1970-12-19', '1970-12-20', '1970-12-21',
               '1970-12-22', '1970-12-23', '1970-12-24'],
              dtype='datetime64[ns]', freq='D')
```

Ebenso ist es möglich Zeitreihen zu erstellen, welche nur die Arbeitstage beinhalten. Dazu muss der Schlüsselwortparameter `freq` auf `B` gesetzt werden:

```
index = pd.date_range('2017-04-07', '2017-04-13', freq="B")
print(index)
```

```
DatetimeIndex(['2017-04-07', '2017-04-10', '2017-04-11', '2017-04-12',
               '2017-04-13'],
              dtype='datetime64[ns]', freq='B')
```

Im nächsten Beispiel generieren wir eine Zeitreihe, welche die Monatsenden zwischen zwei

Zeitpunkten enthält. Dabei sehen wir, dass das Jahr 2016 den 29. Februar hatte, weil es ein Schaltjahr war:

```
index = pd.date_range('2016-02-25', '2016-07-02', freq="ME")
print(index)
```

```
DatetimeIndex(['2016-02-29', '2016-03-31', '2016-04-30', '2016-05-31',
               '2016-06-30'],
              dtype='datetime64[ns]', freq='M')
```

Weitere Abkürzungen:

Alias	Description
B	business day frequency
C	custom business day frequency (experimental)
D	calendar day frequency
W	weekly frequency
M	month end frequency
BM	business month end frequency
MS	month start frequency
BMS	business month start frequency
Q	quarter end frequency
BQ	business quarter end frequency
QS	quarter start frequency
BQS	business quarter start frequency
A	year end frequency
BA	business year end frequency
AS	year start frequency
BAS	business year start frequency
H	hourly frequency
T	minutely frequency
S	secondly frequency
L	milliseconds
U	microseconds

```
index = pd.date_range('2017-02-05', '2017-04-13', freq="W-Mon")
print(index)
```

```
DatetimeIndex(['2017-02-06', '2017-02-13', '2017-02-20', '2017-02-27',
               '2017-03-06', '2017-03-13', '2017-03-20', '2017-03-27',
               '2017-04-03', '2017-04-10'],
              dtype='datetime64[ns]', freq='W-MON')
```

30 Haushaltsbuch Mit Pandas

30.0.1 Haushaltsbuch

In diesem Kapitel unseres Pandas-Tutorials werden wir uns mit einfachen Ausgaben- und Einkommenstabellen für den privaten Gebrauch befassen. Einige sagen, wenn man Geld erfolgreich verwalten möchte, muss man die Einnahmen und Ausgaben genau verfolgen. Die Überwachung des Geldflusses ist wichtig, um die eigene finanzielle Situation besser zu verstehen. Man muss genau wissen, wie viel ein- und ausgeht. Dazu bedient man sich üblicherweise eines Haushaltsbuches, indem alle Ein- und Ausgaben protokolliert sind. Dieser Artikel will niemanden von der Notwendigkeit überzeugen, dies zu tun. Das Hauptaugenmerk liegt vielmehr auf den Möglichkeiten, die Python und Pandas bieten, um die erforderlichen Tools zu programmieren. Wir zeigen Verfahren, um eine Haushaltsbuch auszuwerten.

Wir beginnen mit einem kleinen Beispiel, das für private Zwecke geeignet ist. Im folgenden Kapitel unseres Pandas-Tutorials wird ein ausführlicheres Beispiel behandelt, welches für kleine Unternehmen geeignet ist.

Private Haushaltsplanung mit Python und Pandas

Nehmen wir an, Sie haben bereits eine CSV-Datei, die Ihre Ausgaben und Einnahmen über einen bestimmten Zeitraum enthält. Dieses Tagebuch könnte folgendermaßen aussehen:

```
Datum;Beschreibung;Kategorie;Ausgaben;Einnahmen
2020-06-02;Gehalt Frank;Einkommen;0;4896.44
2020-06-03;Supermarkt;Essen und Getränke;132.40;0
2020-06-04;Gehalt Laura;Einkommen;0;4910.14
2929-06-04;Stadtwerke (Strom);Betriebskosten;87.34;0
2020-06-09;Stadtwerke (Wasser und Abwasser);Betriebskosten;60.56;0
2020-06-10;Fitnessstudio, Jane;Gesundheit und Sport;19.00;0
2020-06-11;Bank;Kredittilgung;1287.43;0
2020-06-12;LeGourmet Restaurant;Restaurants und Hotels;145.00;0
2020-06-13;Supermarkt;Essen und Getränke;197.42;0
2020-06-13;Pizzeria da Pulcinella;Restaurants und Hotels;60.00;0
2020-06-26;Supermarkt;Essen und Getränke;155.42;0
2020-06-27;Theatertickets;education and culture;125;0
2020-07-02;Gehalt Frank;Einkommen;0;4896.44
2020-07-03;Supermarkt;Essen und Getränke;147.90;0
2020-07-05;Gehalt Laura;Einkommen;0;4910.14
2020-07-08;Golf Club, jährliche Zahlung;Gesundheit und Sport;612.18;0
2020-07-09;Hausversicherung;Versicherungen;167.89;0
2020-07-10;Fitnessstudio, Jane;Gesundheit und Sport;19.00;0
2020-07-10;Supermarkt;Essen und Getränke;144.12;0
2020-07-11;Banküberweisung;Kredittilgung;1287.43;0
2020-07-18;Supermarkt;Essen und Getränke;211.24;0
```

```
2020-07-13;Pizzeria da Pulcinella;Restaurants und Hotels;33.00;0
2020-07-23;Kinobesuch;Kultur und Weiterbildung;19;0
2020-07-25;Supermarkt;Essen und Getränke;186.11;0
```

Die oben genannte CSV-Datei wird unter dem Namen `ein_und_ausgaben.csv` im Ordner `data` gespeichert. Es ist einfach, sie mit Pandas einzulesen, wie wir in unserem Kapitel Dateien lesen und schreiben mit Pandas sehen können:

```
import pandas as pd

import os
print(os.getcwd())

exp_inc = pd.read_csv("data1/ein_und_ausgaben.csv", sep=";")
exp_inc
```

```
/data/Dropbox (Bodenseo)/notebooks/pandas_de
```

	Datum	Beschreibung	Kategorie	\
0	2020-06-02	Gehalt Frank	Einkommen	
1	2020-06-03	Supermarkt	Essen und Getränke	
2	2020-06-04	Gehalt Laura	Einkommen	
3	2929-06-04	Stadtwerke (Strom)	Betriebskosten	
4	2020-06-09	Stadtwerke (Wasser und Abwasser)	Betriebskosten	
5	2020-06-10	Fitnessstudio, Jane	Gesundheit und Sport	
6	2020-06-11	Bank	Kredittilgung	
7	2020-06-12	LeGourmet Restaurant	Restaurants und Hotels	
8	2020-06-13	Supermarkt	Essen und Getränke	
9	2020-06-13	Pizzeria da Pulcinella	Restaurants und Hotels	
10	2020-06-26	Supermarkt	Essen und Getränke	
11	2020-06-27	Theatertickets	Kultur und Weiterbildung	
12	2020-07-02	Gehalt Frank	Einkommen	
13	2020-07-03	Supermarkt	Essen und Getränke	
14	2020-07-05	Gehalt Laura	Einkommen	
15	2020-07-08	Golf Club, jährliche Zahlung	Gesundheit und Sport	
16	2020-07-09	Hausversicherung	Versicherungen und Steuern	
17	2020-07-10	Fitnessstudio, Jane	Gesundheit und Sport	
18	2020-07-10	Supermarkt	Essen und Getränke	
19	2020-07-11	Banküberweisung	Kredittilgung	
20	2020-07-18	Supermarkt	Essen und Getränke	
21	2020-07-13	Pizzeria da Pulcinella	Restaurants und Hotels	
22	2020-07-23	Kinobesuch	Kultur und Weiterbildung	
23	2020-07-25	Supermarkt	Essen und Getränke	

	Ausgaben	Einnahmen
0	0.00	4896.44
1	132.40	0.00
2	0.00	4910.14
3	87.34	0.00
4	60.56	0.00
5	19.00	0.00
6	1287.43	0.00
7	145.00	0.00
8	197.42	0.00
9	60.00	0.00
10	155.42	0.00
11	125.00	0.00

```
12      0.00    4896.44
13    147.90       0.00
14      0.00    4910.14
15    612.18       0.00
16    167.89       0.00
17     19.00       0.00
18    144.12       0.00
19   1287.43       0.00
20    211.24       0.00
21     33.00       0.00
22     19.00       0.00
23    186.11       0.00
```

Durch Lesen der CSV-Datei haben wir ein DataFrame-Objekt erstellt. Was können wir damit machen oder mit anderen Worten: Welche Informationen interessieren Frank und Laura? Natürlich interessieren sie sich für den Kontostand. Sie wollen wissen, wie hoch das Gesamteinkommen war, und sie wollen die Summe aller Ausgaben sehen.

Die Salden ihrer Ausgaben und Einnahmen können leicht berechnet werden, indem die Funktion `sum` auf das DataFrame`exp_inc[['Out', 'In']]`-Objekt angewendet:

```
exp_inc[['Ausgaben', 'Einnahmen']].sum()
```

```
Ausgaben      5097.44
Einnahmen    19613.16
dtype: float64
```

Welche weiteren Informationen möchten sie aus den Daten gewinnen? Sie könnten daran interessiert sein, die Ausgaben nach den verschiedenen Kategorien zusammengefasst zu sehen. Dies lässt sich mittels groupby und sum bewerkstelligen:

```
category_sums = exp_inc.groupby("Kategorie").sum()
category_sums
```

```
                          Ausgaben  Einnahmen
Kategorie
Betriebskosten              147.90       0.00
Einkommen                     0.00   19613.16
Essen und Getränke         1174.61       0.00
Gesundheit und Sport        650.18       0.00
Kredittilgung              2574.86       0.00
Kultur und Weiterbildung    144.00       0.00
Restaurants und Hotels      238.00       0.00
Versicherungen und Steuern  167.89       0.00
```

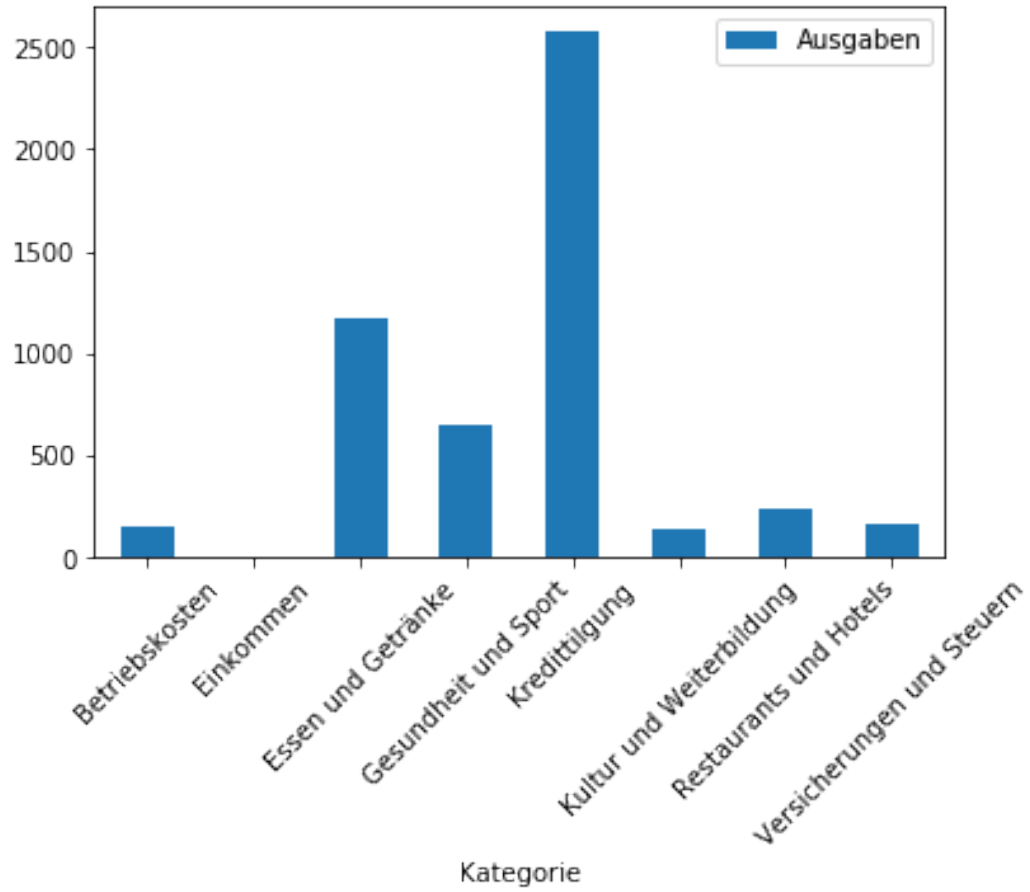
```
category_sums.index
```

```
Index(['Betriebskosten', 'Einkommen', 'Essen und Getränke',
       'Gesundheit und Sport', 'Kredittilgung', 'Kultur und Weiterbildung',
       'Restaurants und Hotels', 'Versicherungen und Steuern'],
      dtype='object', name='Kategorie')
```

```
import matplotlib.pyplot as plt

ax = category_sums.plot.bar(y="Ausgaben")
```

```
plt.xticks(rotation=45)
```

```
(array([0, 1, 2, 3, 4, 5, 6, 7]), <a list of 8 Text xticklabel objects>)
```



Wir können dies auch als Tortendiagramm darstellen:

```
ax = category_sums.plot.pie(y="Ausgaben")
ax.legend(loc="upper left", bbox_to_anchor=(1.5, 1))
```

```
<matplotlib.legend.Legend at 0x7fc2c20966d0>
```

Alternativ können wir dasselbe Kreisdiagramm mit dem folgenden Code erstellen:

```
ax = category_sums["Ausgaben"].plot.pie()
ax.legend(loc="upper left", bbox_to_anchor=(1.5, 1))
```

```
<matplotlib.legend.Legend at 0x7fc2c2011ed0>
```

If you imagine that you will have to type in all the time category names like “household goods and service” or “rent and mortgage interest”, you will agree that it is very likely to have typos in your journal of expenses and income.

So it will be a good idea to use numbers (account numbers) for your categories. The following categories are available in our example.

The following categories are provided:

Wenn man sich vorstellt, dass man die ganze Zeit Kategorienamen wie “Essen und Getränke” oder “Restaurants und Hotels” eingeben muss, kann man sich leicht vorstelltn, dass sich leicht Tippfehler im Ausgaben- und Einnahmenjournal einschleichen können.

Es ist daher eine gute Idee, Nummern (Kontonummern) für die Kategorien zu verwenden. Die folgenden Kategorien sind in unserem Beispiel verfügbar:

Kategorie	Kontonummer
Kredittilgung	200
Versicherungen und Steuern	201
Essen und Getränke	202
Kultur und Weiterbildung	203
Transport	204
Gesundheit und Sport	205
Haushalt und Dienstleistungen	206
Kleidung	207
Kommunikationskosten	208
Restaurants und Hotels	209
Betriebskosten (Heizung, Strom, Wasser, Müll)	210
andere Ausgaben	211
Einkommen	400

Wir können dies als Python `dictionary` implementieren, das Kategorien in Kontonummern abbildet:

```
category2account = {"Kredittilgung": "200",
                    "Versicherungen und Steuern": "201",
                    "Essen und Getränke": "202",
                    "Kultur und Weiterbildung": "203",
                    "Transport": "204",
                    "Gesundheit und Sport": "205",
                    "Haushalt und Dienstleistungen": "206",
                    "Kleidung": "207",
                    "Kommunikationskosten": "208",
                    "Restaurants und Hotels": "209",
                    "Betriebskosten": "210",
                    "andere Ausgaben": "211",
                    "Einkommen": "400"}
```

Der nächste Schritt besteht darin, unsere "unhandlichen" Kategorienamen durch die Kontonummern zu ersetzen. Die Ersetzungsmethode `replace` von `DataFrame` ist für diesen Zweck ideal. Wir können alle Vorkommen der Kategorienamen in unserem `DataFrame` durch die entsprechenden Kontonamen ersetzen:

```
exp_inc.replace(category2account, inplace=True)
exp_inc.rename(columns={"Category": "Accounts"}, inplace=True)
exp_inc
```

```
         Datum                      Beschreibung Kategorie  Ausgaben  \
0   2020-06-02                      Gehalt Frank       400      0.00
1   2020-06-03                        Supermarkt       202    132.40
2   2020-06-04                      Gehalt Laura       400      0.00
3   2929-06-04                Stadtwerke (Strom)       210     87.34
4   2020-06-09  Stadtwerke (Wasser und Abwasser)       210     60.56
5   2020-06-10               Fitnessstudio, Jane       205     19.00
6   2020-06-11                              Bank       200   1287.43
7   2020-06-12             LeGourmet Restaurant       209    145.00
8   2020-06-13                        Supermarkt       202    197.42
9   2020-06-13           Pizzeria da Pulcinella       209     60.00
10  2020-06-26                        Supermarkt       202    155.42
11  2020-06-27                     Theatertickets       203    125.00
12  2020-07-02                      Gehalt Frank       400      0.00
13  2020-07-03                        Supermarkt       202    147.90
14  2020-07-05                      Gehalt Laura       400      0.00
15  2020-07-08     Golf Club, jährliche Zahlung       205    612.18
16  2020-07-09                  Hausversicherung       201    167.89
17  2020-07-10               Fitnessstudio, Jane       205     19.00
18  2020-07-10                        Supermarkt       202    144.12
19  2020-07-11                    Banküberweisung       200   1287.43
20  2020-07-18                        Supermarkt       202    211.24
21  2020-07-13           Pizzeria da Pulcinella       209     33.00
22  2020-07-23                        Kinobesuch       203     19.00
23  2020-07-25                        Supermarkt       202    186.11

    Einnahmen
0     4896.44
1        0.00
2     4910.14
```

```
3        0.00
4        0.00
5        0.00
6        0.00
7        0.00
8        0.00
9        0.00
10       0.00
11       0.00
12    4896.44
13       0.00
14    4910.14
15       0.00
16       0.00
17       0.00
18       0.00
19       0.00
20       0.00
21       0.00
22       0.00
23       0.00
```

Wir werden dieses DataFrame-Objekt jetzt in einer Excel-Datei speichern. Diese Excel-Datei enthält zwei Blätter: eines mit dem Journal "Ausgaben und Einnahmen" und das andere mit der Zuordnung von Kontonummern zu Kategorienamen. Zu diesem Zweck verwandeln wir das `category2account`-Dictionary in ein Series-Objekt. Als Index dienen die Kontonummern:

We will save this `DataFrame` object now in an excel file. This excel file will have two sheets: One with the "expenses and income" journal and the other one with the mapping of account numbers to category names. We will turn the category2account dictionary into a Series object for this purpose. The index will be the account numbers:

```
account_numbers = pd.Series(list(category2account.keys()),
↪  index=category2account.values())
account_numbers.name = "Beschreibung"
account_numbers.rename("Kontonummern")
```

```
200                   Kredittilgung
201       Versicherungen und Steuern
202               Essen und Getränke
203         Kultur und Weiterbildung
204                        Transport
205             Gesundheit und Sport
206    Haushalt und Dienstleistungen
207                         Kleidung
208             Kommunikationskosten
209           Restaurants und Hotels
210                   Betriebskosten
211                  andere Ausgaben
400                        Einkommen
Name: Kontonummern, dtype: object
```

```
exp_inc.insert(1, "Kontonummern", account_numbers)
```

```
with pd.ExcelWriter('data1/expenses_and_income_2020.xlsx') as writer:
```

```
    account_numbers.to_excel(writer, "account numbers")
    exp_inc.to_excel(writer, "journal")
    writer.save()
```

31 Einnahme Ueberschuss Rechnung

31.0.1 Einnahmeüberschussrechnung

Bei der in Deutschland benutzten Einnahmenüberschussrechnung (EÜR), - in Österreich als Einnahmen-Ausgaben-Rechnung (E/A-Rechnung) bekannt - handelt es sich um eine vereinfachte Gewinnermittlungsmethode, die so vom Gesetz vorgegeben und und für bestimmte Berufsgruppen anerkannt ist. Einnahmenüberschussrechnung ist sowohl in Deutschland als auch in Österreich im § 4 Abs. 3 des jeweiligen Einkommensteuergesetzes (EStG) geregelt. Alle Stuerpflichtigen, die nicht zuur doppelten Buchführung verpflichtet sind, dürfen dieses vereinfachte Verfahren zuur Gewinnermittler verwenden. Bei der E*R gilt das Zufluss- und Abflussprinzip, das bedeutet das lediglich die Einnahmen bzw. Ausgaben zu berücksichtigen sind, die in dem entsprechenden Wirtschaftsjahr vereinnahmt bzw. gezahlt wurden. Bestandsveränderungen bleiben unberücksichtigt. Auch wenn hier und an anderen Stellen imer von “vereinfacht” oder von “einfacher” Gewinnermittlungsmethode die Rede ist, so darf das nicht darüber hinweg täuschen, dasss auch bei dieser Methode viele gesetzlichen Regeln zu beachten sind. Im Rahmen dieses Tutorials kann natürlich nicht auf die komplexe Materie eingegangen werden. Die hier vorgestellten Verfahren können für die eigene EÜr verwendet werden, aber es kann keine Garantie auf die Richtigkeit gegeben werden.

Wir sind hauptsächlich daran interessiert, die verschiedenen Möglichkeiten vorzustellen, wie Pandas und Python zur Einnahmeüberschussrechnung benutzt werden können. Wir zeigen, wie es mit Python möglich ist, den Geldfluss zu überwachen und zu visualisieren. Auf diese Weise kann man sich einen besseren Überblick über die finanzielle Situation des Betriebes oder der selbständigen Tätigkeit verschaffen. Die hier vorgestellten Algorithmen können auch steuerlich eingesetzt werden. Aber seien Sie gewarnt, dies ist eine sehr allgemeine Behandlung der Angelegenheit und muss an die tatsächliche Steuersituation angepasst werden, d.h. es kann keine Gewähr für die steuerliche Richtigkeit übernommen werden.

Im Ordner `data1` befindet sich eine Excel-Datei `net_income_method_2020.xlsx` mit den Buchhaltungsdaten eines fiktiven Unternehmens.

Journaldatei

Dieses Excel-Dokument enthält zwei Datenblätter: eines mit den tatsächlichen Daten “journal” und eines mit dem Namen “account numbers” (Kontonummern), das die Zuordnung von den Kontonummern zur Beschreibung enthält.

Wir werden diese Excel-Datei in zwei DataFrame-Objekte lesen:

```
import pandas as pd

with pd.ExcelFile("data1/net_income_method_2020.xlsx") as xl:
    accounts2descr = xl.parse("account numbers",
                              index_col=0)
    journal = xl.parse("journal",
                       index_col=0,
```

```
                      )

journal.index = pd.to_datetime(journal.index)
journal.index
```

```
DatetimeIndex(['2020-04-02', '2020-04-02', '2020-04-02', '2020-04-02',
               '2020-04-02', '2020-04-02', '2020-04-05', '2020-04-05',
               '2020-04-05', '2020-04-05', '2020-04-09', '2020-04-09',
               '2020-04-10', '2020-04-10', '2020-04-10', '2020-04-10',
               '2020-04-10', '2020-04-10', '2020-04-13', '2020-04-13',
               '2020-04-13', '2020-04-26', '2020-04-26', '2020-04-26',
               '2020-04-26', '2020-04-27', '2020-05-03', '2020-05-03',
               '2020-05-03', '2020-05-03', '2020-05-05', '2020-05-05',
               '2020-05-08', '2020-05-09', '2020-05-10', '2020-05-11',
               '2020-05-11', '2020-05-11', '2020-05-11', '2020-05-11',
               '2020-05-13', '2020-05-18', '2020-05-25', '2020-05-25',
               '2020-06-01', '2020-06-02', '2020-06-03', '2020-06-03',
               '2020-06-04', '2020-06-04', '2020-06-09', '2020-06-10',
               '2020-06-10', '2020-06-11', '2020-06-11', '2020-06-11',
               '2020-06-11', '2020-06-11', '2020-06-12', '2020-06-13',
               '2020-06-13', '2020-06-26', '2020-06-26', '2020-06-27',
               '2020-07-02', '2020-07-03', '2020-07-05', '2020-07-05',
               '2020-07-08', '2020-07-09', '2020-07-10', '2020-07-10',
               '2020-07-10', '2020-07-10', '2020-07-10', '2020-07-10',
               '2020-07-11', '2020-07-11', '2020-07-13', '2020-07-18',
               '2020-07-23', '2020-07-23', '2020-07-25', '2020-07-25',
               '2020-07-27', '2020-07-26', '2020-07-28'],
              dtype='datetime64[ns]', name='date', freq=None)
```

Die erste ist die Registerkarte “Kontonummern”, die die Zuordnung von den Kontonummern zur Beschreibung der Konten enthält:

```
accounts2descr
```

```
                   description
account
4400        revenue plant Munich
4401     revenue plant Frankfurt
4402        revenue plant Berlin
2010                   souvenirs
2020                     clothes
2030              other articles
2050                       books
2100                   insurances
2200                        wages
2300                        loans
2400                       hotels
2500                       petrol
2600           telecommunication
2610                    internet
```

Das zweite Datenblatt “journal” enthält die eigentlichen Journaleinträge:

```
journal[:10]
```

```
            account number  document number              description  \
```

```
date
2020-04-02            4402       8983233038             Zurkan, Köln
2020-04-02            2010         57550799       Birmann, Souvenirs
2020-04-02            2200         14989004                    wages
2020-04-02            2500         12766279  Filling Station, Petrol
2020-04-02            4400       3733462359       EnergyCom, Hamburg
2020-04-02            4402       7526058231       Enoigo, Strasbourg
2020-04-05            4402       1157284466        Qbooks, Frankfurt
2020-04-05            4402       7009463592             Qbooks, Köln
2020-04-05            2020         68433353          Jamdon, Clothes
2020-04-05            2010         53353169        Outleg, Souvenirs

            tax rate  gross amount
date
2020-04-02        19       4105.98
2020-04-02        19      -1890.00
2020-04-02         0     -17478.23
2020-04-02        19        -89.40
2020-04-02        19       4663.54
2020-04-02        19       2412.82
2020-04-05         7       2631.42
2020-04-05         7       3628.45
2020-04-05        19      -1900.00
2020-04-05        19      -2200.00
```

Es gibt viele Möglichkeiten, diese Daten zu analysieren. Wir können zum Beispiel alle Konten zusammenfassen:

```
account_sums = journal[["account number",
↪  "gross amount"]].groupby("account number").sum()
account_sums
```

```
                gross amount
account number
2010                -4090.00
2020               -10500.80
2030                -1350.00
2050                 -900.00
2100                 -612.00
2200               -69912.92
2300               -18791.92
2400                -1597.10
2500                   -89.40
2600                 -492.48
2610                 -561.00
4400                37771.84
4401                69610.35
4402                61593.99
```

Diagramme der Konten

Wie wäre es mit einer Visualisierung dieser Kontendaten? Dazu könnten wir beispielsweise Kreisdiagramme, auch Kuchendiagramme genannt, benutzen. Im Englischen bezeichnet man sie als “pie charts”. Nun haben wir aber ein kleines Problem: Kreisdiagramme dürfen keine negativen Werte enthalten. Dies ist jedoch kein wirkliches Problem. Wir können die Konten in Einnahmen- und Ausgabenkonten aufteilen. Das entspricht natürlich auch mehr

dem, was wir wirklich sehen wollen.

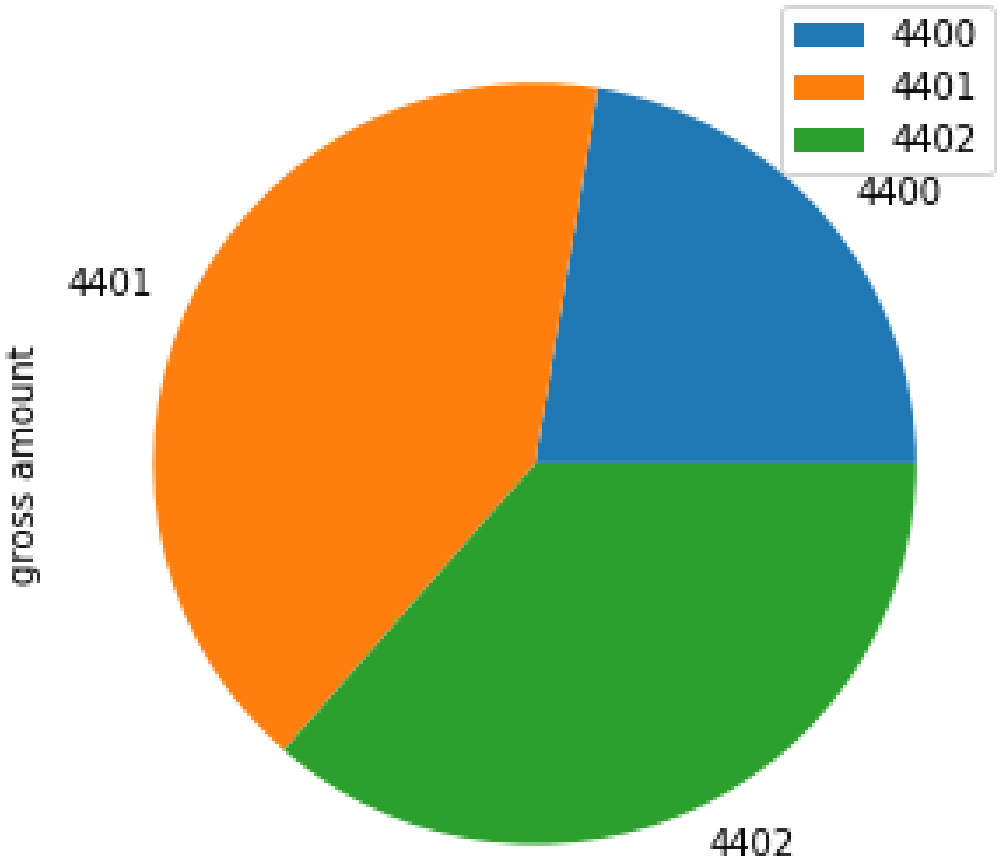
Diagramme für die Einnahmekonten (Kreditoren) Wir erstellen einen DataFrame mit den Einkommenskonten:

```
income_accounts = account_sums[account_sums["gross amount"] > 0]
income_accounts
```

```
                gross amount
account number
4400                37771.84
4401                69610.35
4402                61593.99
```

Wir können diese Werte jetzt in einem Kreisdiagramm visualisieren.

```
plot = income_accounts.plot(y='gross amount', figsize=(5, 5), kind="pie")
```



Die Position der Legende gefällt Ihnen wahrscheinlich nicht? Mit dem Parameter `bbox_to_anchor` können wir ihn an einer gewünschten Position positionieren. Wenn wir wollen, können wir es sogar außerhalb des Grundstücks verschieben. Mit den relativen Koordinatenwerten

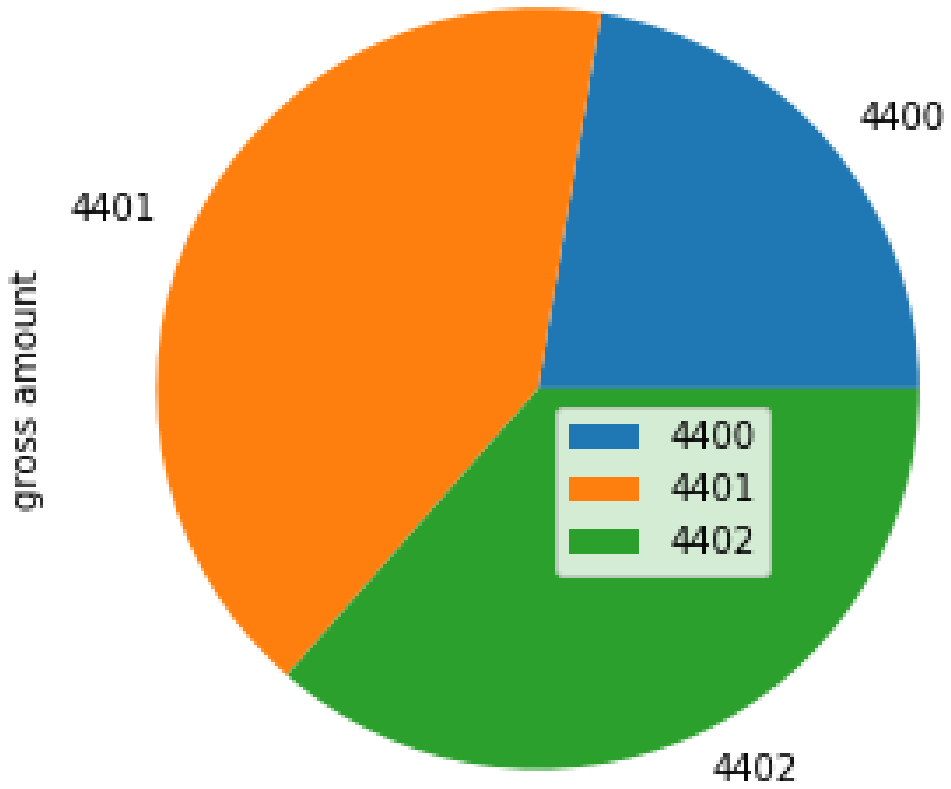
`(0.5, 0.5)` positionieren wir die Legende in der Mitte des Diagramms. Die Legende ist ein Rechteck. Die Frage ist also, was die Position (0,5, 0,5) bedeutet. Wir können dies definieren, indem wir zusätzlich den Parameter 'loc"' verwenden:

loc-Wert	Bedeutung
upper left	bbox_to_anchor bezeichnet die Position der linken oberen Ecke des Legenden-Rechtecks
upper right	entsprechend die rechte obere Ecke
lower left	die linke untere Ecke
lower left	die rechte untere Ecke

Wir verwenden dies, um die Legende mit der linken oberen Ecke in der Mitte des Plots zu positionieren:

```
plot = income_accounts.plot(y='gross amount',
                            figsize=(5, 5),
                            kind="pie")
plot.legend(bbox_to_anchor=(0.5, 0.5),
            loc="upper left")
```

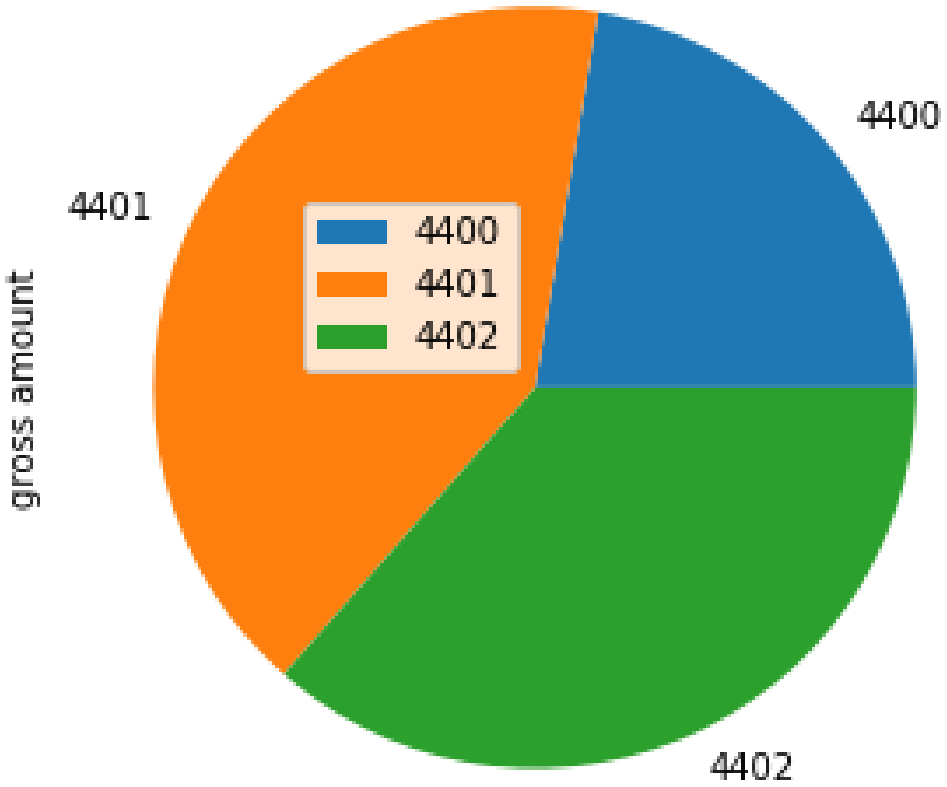
```
<matplotlib.legend.Legend at 0x7fe2da444a50>
```



Jetzt positionieren wir die untere rechte Ecke der Legende in der Mitte des Diagramms:

```
plot = income_accounts.plot(y='gross amount', figsize=(5, 5), kind="pie")
plot.legend(bbox_to_anchor=(0.5, 0.5),
            loc="lower right")
```

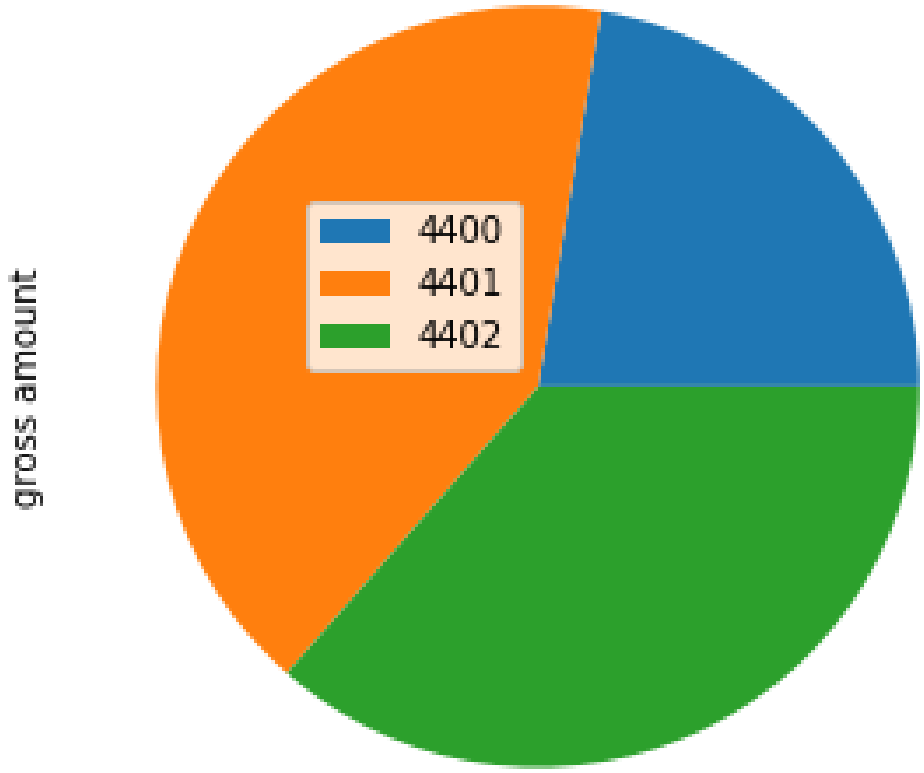
```
<matplotlib.legend.Legend at 0x7fe2da40b510>
```



Es gibt noch eine andere Sache, die wir verbessern können. Wir sehen die Bezeichnungen 4400, 4401 und 4402 neben jedem Kuchensegment. Außerdem sehen wir sie in der Legende. Dies sind hässliche und redundante Informationen. Im Folgenden werden wir die Beschriftungen im Plot deaktivieren, d.h. wir werden sie auf eine leere Zeichenfolge setzen, und wir setzen sie explizit in der Legendenmethode. Allerdings setzen wir sie nicht auf die Kontonummern sondern auf die dazugehörige Beschreibung:

```
plot = income_accounts.plot(y='gross amount',
                            figsize=(5, 5),
                            kind="pie",
                            labels=['', '', ''])
plot.legend(bbox_to_anchor=(0.5, 0.5),
            labels=income_accounts.index)
```

```
<matplotlib.legend.Legend at 0x7fe2da398e10>
```

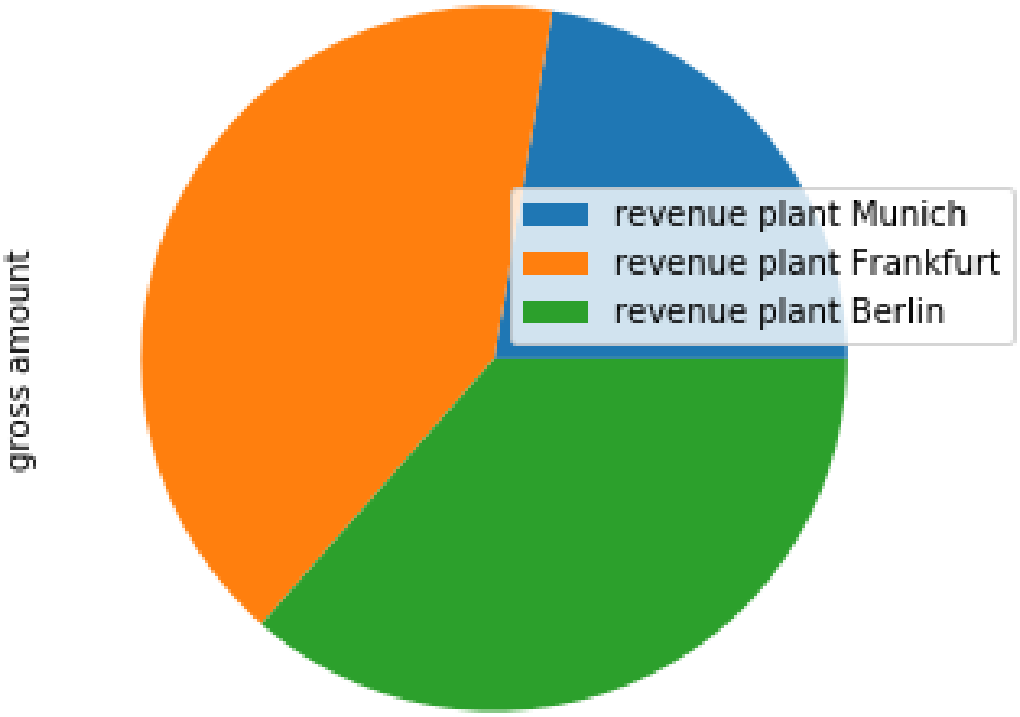


Now, we are close to perfection. Just one more tiny thing. Some might prefer to see the actual description text rather than an account number. We will cut out this information from the DataFrame `accounts2descr` by using `loc` and the list of desired numbers `[4400, 4401, 4402]`. The result of this operation will be the argument of the `set_index` method. (Atention: `reindex` is not giving the wanted results!)

```
s = accounts2descr["description"].loc[[4400, 4401, 4402]]
income_accounts.set_index(s, inplace=True)
```

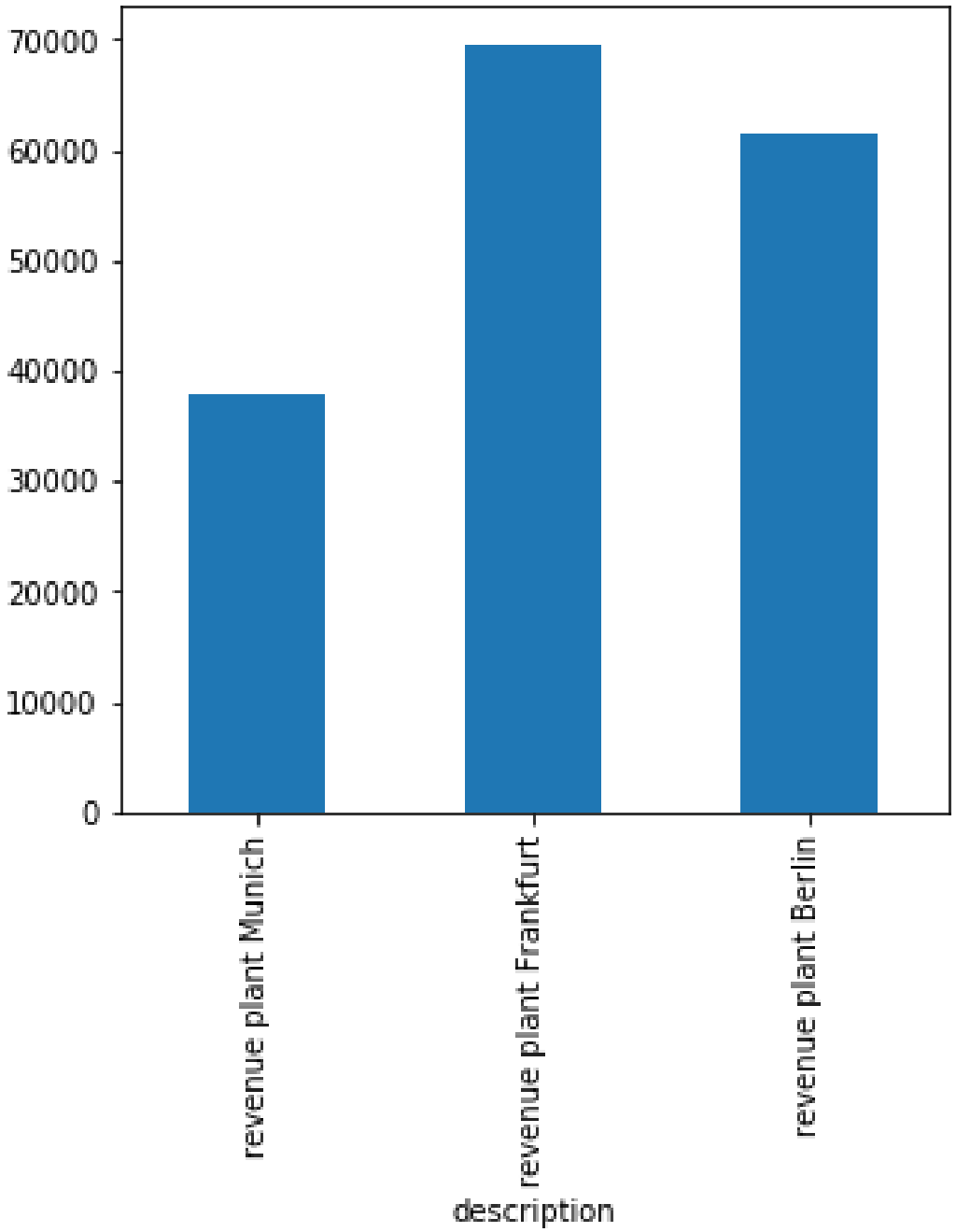
```
descriptions = accounts2descr["description"].loc[[4400, 4401, 4402]]


plot = income_accounts.plot(kind="pie",
                            y='gross amount',
                            figsize=(5, 5),
                            labels=['', '', ''])

plot.legend(bbox_to_anchor=(0.5, 0.5),
            loc="lower left",
            labels=descriptions)
```

```
<matplotlib.legend.Legend at 0x7fe2da3d2f10>
```



Soll es doch lieber ein Säulendiagramm sein? Kein Problem, wir müssen nur den Parameter `kind` auf `bar` statt auf `pie` setzen:

```
plot = income_accounts.plot(y='gross amount',
                            figsize=(5, 5),
                            kind="bar",
                            legend=False)
```



Für Balkendiagramme müssen müssen wir `kind` auf `barh` setzen. Damit es nicht zu langweilig wird können wir die Balken auch einfärben, indem wir dem Parameter `color` eine Liste mit Farben übergeben:

```
plot = income_accounts.plot(y='gross amount',
                            figsize=(5, 5),
                            kind="barh",
                            legend=False,
                            color=['green', 'orange', 'blue'])
```

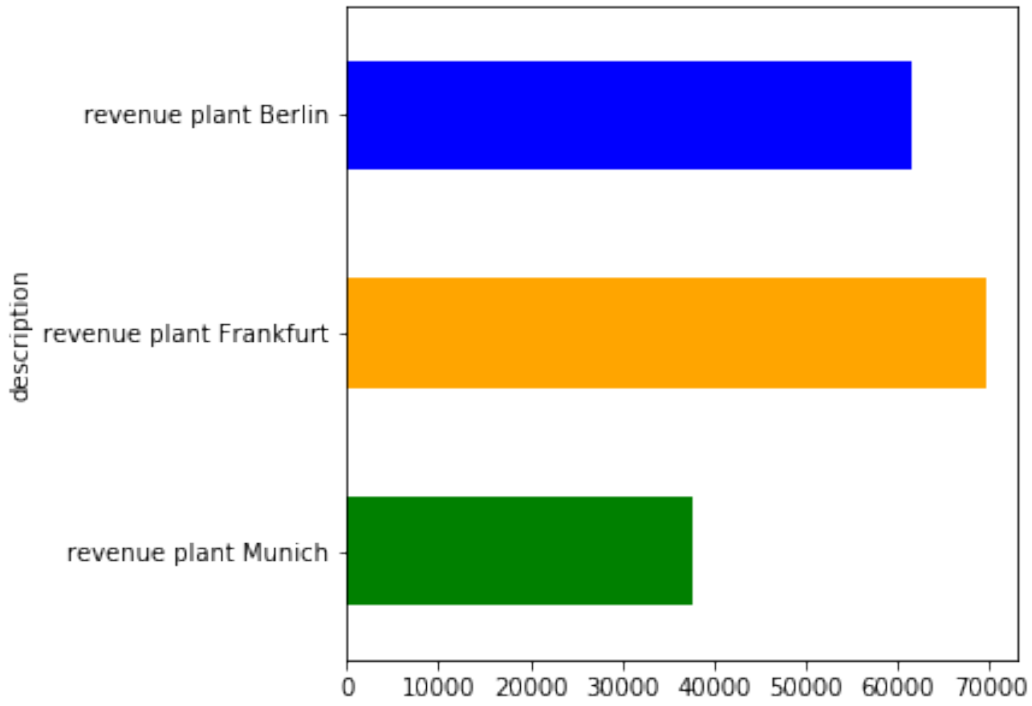


Diagramme für die Ausgabenkonten (Debitoren) Das Gleiche können wir jetzt mit unseren Debitoren (Spesenabrechnungen) tun:

```
expenses_accounts = account_sums[account_sums["gross amount"] < 0]
expenses_accounts
```

```
                gross amount
account number
2010                -4090.00
2020               -10500.80
2030                -1350.00
2050                 -900.00
2100                 -612.00
2200               -69912.92
2300               -18791.92
2400                -1597.10
2500                  -89.40
2600                 -492.48
2610                 -561.00
```

```
acc2descr_expenses = accounts2descr["description"].loc[expenses_accounts.index]
acc2descr_expenses
```

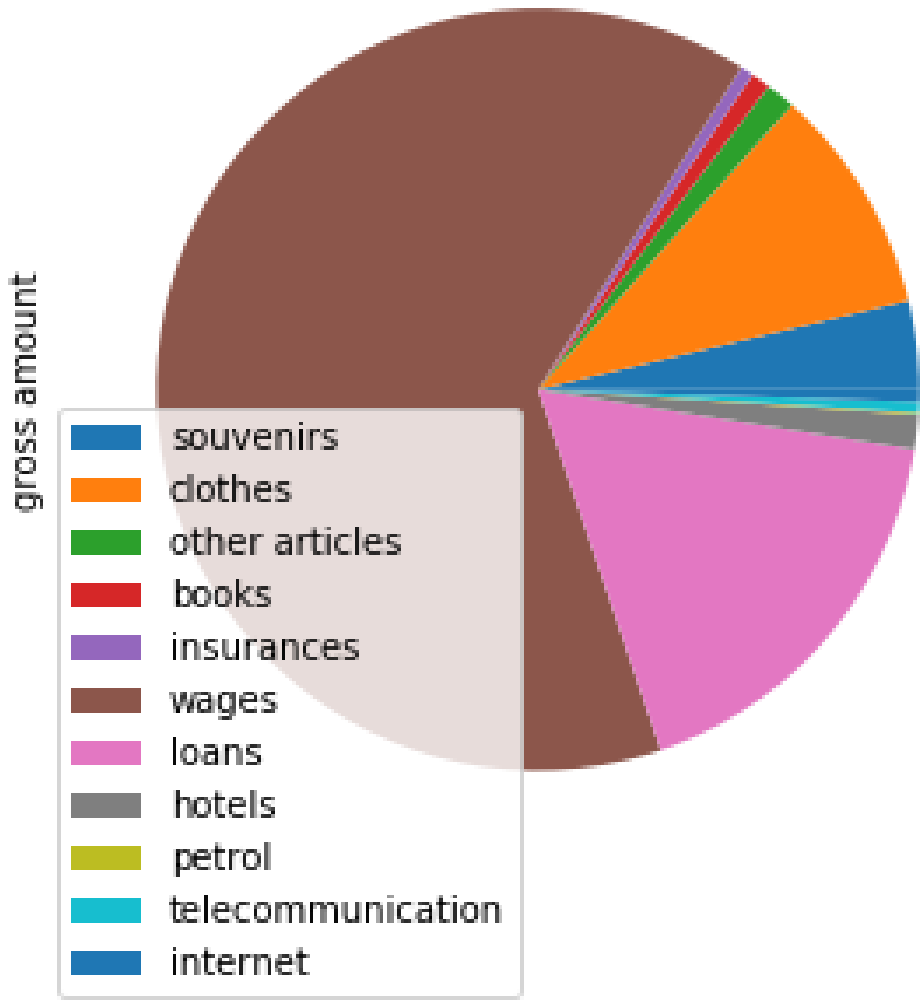
```
account number
2010            souvenirs
2020              clothes
2030       other articles
2050                books
2100           insurances
2200                wages
2300                loans
```

```
2400               hotels
2500               petrol
2600    telecommunication
2610             internet
Name: description, dtype: object
```

```
expenses_accounts.set_index(acc2descr_expenses.values, inplace=True)

expenses_accounts *= -1
```

```
labels = [''] * len(expenses_accounts)
plot = expenses_accounts.plot(kind="pie",
                              y='gross amount',
                              figsize=(5, 5),
                              labels=labels)
plot.legend(bbox_to_anchor=(0.5, 0.5),
            labels=expenses_accounts.index)
```

```
<matplotlib.legend.Legend at 0x7fe2da1f71d0>
```



Tax Sums

We will sum up the amount according to their tax rate.

```
journal.drop(columns=["account number"])
```

```
            document number                   description  tax rate  \
date
2020-04-02       8983233038                  Zurkan, Köln        19
2020-04-02         57550799            Birmann, Souvenirs        19
2020-04-02         14989004                         wages         0
2020-04-02         12766279       Filling Station, Petrol        19
2020-04-02       3733462359            EnergyCom, Hamburg        19
...                     ...                           ...       ...
2020-07-25       5204418668  BoKoData, Bodensee, Konstanz        19
2020-07-25         85241331               Hotel, Konstanz         7
2020-07-27         26865618               Hotel, Franfurt         7
2020-07-26       5892708524    Oscar Zopvar KG, Luxemburg        19
2020-07-28       1633775505    Oscar Zopvar KG, Luxemburg        19

            gross amount
date
2020-04-02       4105.98
2020-04-02      -1890.00
2020-04-02     -17478.23
2020-04-02        -89.40
2020-04-02       4663.54
...                  ...
2020-07-25       3678.38
2020-07-25       -583.00
2020-07-27       -450.00
2020-07-26       2589.80
2020-07-28       3578.46

[87 rows x 4 columns]
```

Im Folgenden definieren wir nun eine Funktion `steuersummen`, die die Mehrwertsteuersummen nach Steuersätzen aus einem Journal-DataFrame berechnet:

```
def steuersummen(journal_df, months=None):
    """ Liefert ein DataFrame mit den Umsätzen und Steuersätzen zurück-
        Wird months eine Zahl oder Liste übergeben, werden nur die Umsätze
        der entshrechenden Monate berücksichtigt.
        Beispiel: steuersummen(df, months=[3, 6]) bedeutet nur die Monate
        3 (März) und 6 (Juni)"""
    if months:
        if isinstance(months, int):
            month_cond = journal_df.index.month == months
        elif isinstance(months, (list, tuple)):
            month_cond = journal_df.index.month.isin(months)
        positive = journal_df["gross amount"] > 0
        umsatzsteuern = journal_df[positive & month_cond]
        negative = journal_df["gross amount"] < 0
        vorsteuern = journal_df[negative & month_cond]
    else:
        umsatzsteuern = journal_df[journal_df["gross amount"] > 0]
```

```
        vorsteuern = journal_df[journal_df["gross amount"] < 0]

    umsatzsteuern = umsatzsteuern[["tax rate",
    ↪  "gross amount"]].groupby("tax rate").sum()
    umsatzsteuern.rename(columns={"gross amount": "Umsaetze brutto"},
                         inplace=True)
    umsatzsteuern.index.name = 'Steuerrate'

    vorsteuern = vorsteuern[["tax rate",
    ↪  "gross amount"]].groupby("tax rate").sum()
    vorsteuern.rename(columns={"gross amount": "Ausgaben brutto"},
                      inplace=True)
    vorsteuern.index.name = 'Steuerrate'

    steuern = pd.concat([vorsteuern, umsatzsteuern], axis=1)
    steuern.insert(1,
                   column="Vorsteuer",
                   value=(steuern["Ausgaben brutto"] * steuern.index /
                   ↪  100).round(2))
    steuern.insert(3,
                   column="Umsatzsteuer",
                   value=(steuern["Umsaetze brutto"] * steuern.index /
                   ↪  100).round(2))

    return steuern.fillna(0)


steuersummen(journal)
```

Steuerrate	Ausgaben brutto	Vorsteuer	Umsaetze brutto	Umsatzsteuer
0	-90102.20	-0.00	8334.43	0.00
7	-3847.10	-269.30	11240.71	786.85
19	-14948.32	-2840.18	149401.04	28386.20

```
stsum_5 = steuersummen(journal, months=5)
stsum_6 = steuersummen(journal, months=6)
stsum_5
```

Steuerrate	Ausgaben brutto	Vorsteuer	Umsaetze brutto	Umsatzsteuer
0	-22411.53	-0.00	0.00	0.00
7	-900.00	-63.00	0.00	0.00
19	-145.00	-27.55	31328.98	5952.51

```
stsum_6
```

Steuerrate	Ausgaben brutto	Vorsteuer	Umsaetze brutto	Umsatzsteuer
0	-22400.61	-0.00	6479.47	0.00
7	0.00	0.00	4980.84	348.66
19	-123.12	-23.39	34570.25	6568.35

```
stsum_5 + stsum_6
```

Steuerrate	Ausgaben brutto	Vorsteuer	Umsaetze brutto	Umsatzsteuer
0	-44812.14	-0.00	6479.47	0.00
7	-900.00	-63.00	4980.84	348.66
19	-268.12	-50.94	65899.23	12520.86

```
steuersummen(journal, months=[5, 6])
```

Steuerrate	Ausgaben brutto	Vorsteuer	Umsaetze brutto	Umsatzsteuer
0	-44812.14	-0.00	6479.47	0.00
7	-900.00	-63.00	4980.84	348.66
19	-268.12	-50.94	65899.23	12520.85

Weitere Bücher von Bernd Klein

Die folgenden Verlagstitel sind eigenständige Veröffentlichungen beim Carl Hanser Verlag. Diese automatisch erzeugte PDF-Ausgabe von `python-kurs.eu` ist keine Hanser-Veröffentlichung.

Einführung in Python 3

Für Ein- und Umsteiger

Bernd Klein

ISBN: 978-3-446-46379-0

4. Auflage, 2021

Mehr beim Carl Hanser Verlag

Numerisches Python

Arbeiten mit NumPy, Matplotlib und Pandas

Bernd Klein

ISBN: 978-3-446-48584-6

3. Auflage, 2025

Mehr beim Carl Hanser Verlag

Funktionale Programmierung mit Python

Funktionale Konzepte praxisnah in Python einsetzen

Bernd Klein, Philip Klein

ISBN: 978-3-446-48191-6

1. Auflage, 2025

Mehr beim Carl Hanser Verlag

Hinweis zur Ausgabe

Diese Ausgabe wurde automatisiert aus den Quellen von `python-kurs.eu` erzeugt. Die Online-Fassung kann aktueller sein als diese PDF-Ausgabe.