



bernd KLEIN

Numerisches PYTHON

Arbeiten mit NumPy,
Matplotlib und Pandas

HANSER

Klein Numerisches Python



Bleiben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:

www.hanser-fachbuch.de/newsletter



Bernd Klein

Numerisches Python

Arbeiten mit NumPy,
Matplotlib und Pandas

HANSER

Der Autor:

Bernd Klein, bernd@python-kurs.eu

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autoren und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2019 Carl Hanser Verlag München, www.hanser-fachbuch.de

Lektorat: Brigitte Bauer-Schiewek

Copy editing: Jürgen Dubau, Freiburg/Elbe

Layout: le-tex publishing services, Leipzig

Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München

Umschlagrealisation: Max Kostopoulos

Druck und Bindung: Kösel, Krugzell

Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702

Printed in Germany

Print-ISBN: 978-3-446-45076-9

E-Book-ISBN: 978-3-446-45363-0

E-Pub-ISBN: 978-3-446-46161-1

Inhalt

Vorwort	XV
----------------------	-----------

Danksagung	XVI
-------------------------	------------

1 Einleitung	1
------------------------------	----------

1.1 Die richtige Wahl	1
-------------------------------	---

1.2 Aufbau des Buches	2
-------------------------------	---

1.3 Python-Installation	3
---------------------------------	---

1.4 Download der Beispiele	3
------------------------------------	---

1.5 Anregungen und Kritik	3
-----------------------------------	---

2 Numerisches Programmieren mit Python	5
--	----------

2.1 Definition von numerischer Programmierung	5
---	---

2.2 Zusammenhang zwischen Python, NumPy, Matplotlib, SciPy und Pandas	6
---	---

2.3 Python, eine Alternative zu Matlab	7
--	---

Teil I Kurze Einführung in Python.....	9
---	----------

3 Kurze Einführung in Python	11
--	-----------

3.1 Datenstrukturen	11
-----------------------------	----

3.1.1 Zahlen und Variablen	11
------------------------------------	----

3.1.2 Zeichenketten.....	12
----------------------------	----

3.1.3 Listen.....	15
---------------------	----

3.1.4 Tupel	16
---------------------	----

3.1.5 Frozensets und Mengen in Python.....	17
--	----

3.1.6 Dictionaries	18
----------------------------	----

3.2 Kontrollstrukturen	19
--------------------------------	----

3.2.1 Bedingte Anweisungen	19
------------------------------------	----

3.2.2 Schleifen	20
-------------------------	----

3.2.3	Funktionen	23
3.3	Ausnahmebehandlung	26
3.3.1	Die optionale else-Klausel	29
3.3.2	Exceptions generieren	29
3.3.3	Finalisierungsaktion	29
3.4	Dateien lesen und schreiben	30
3.4.1	Datei lesen.....	30
3.4.2	Datei schreiben	31
3.5	Modularisierung	32
3.5.1	Namensräume von Modulen	32
3.5.2	Suchpfad für Module	33
3.5.3	Inhalt eines Moduls	33
3.5.4	Eigene Module	33
3.5.5	Dokumentation für eigene Module	34
3.6	Klassen-Definition	35
3.6.1	Eine einfache Klasse	35
3.6.2	Attribute	35
3.6.3	Initialisierung von Instanzen	37
3.6.4	Vererbung	37
3.6.5	Private, geschützte und öffentliche Attribute	38
3.6.6	Properties.....	39
Teil II	NumPy	41
4	NumPy Einführung	43
4.1	Überblick.....	43
4.2	Vergleich NumPy-Datenstrukturen und Python.....	44
4.3	Ein einfaches Beispiel	44
4.4	Grafische Darstellung der Werte	45
4.5	Speicherbedarf	46
4.6	Zeitvergleich zwischen Python-Listen und NumPy-Arrays	49
5	Arrays in NumPy erzeugen	51
5.1	Erzeugung äquidistanter Intervalle	51
5.1.1	arange	51
5.1.2	linspace	52
5.1.3	Nulldimensionale Arrays in NumPy	53

5.1.4	Eindimensionales Array	53
5.1.5	Zwei- und Mehrdimensionale Arrays.....	54
5.2	Shape/Gestalt eines Arrays	54
5.3	Indizierung und Teilbereichsoperator	56
5.4	Dreidimensionale Arrays	61
5.5	Arrays mit Nullen und Einsen	64
5.6	Arrays kopieren.....	65
5.6.1	numpy.copy(A) und A.copy()	65
5.6.2	Zusammenhängend gespeicherte Arrays	65
5.7	Identitätsarray	67
5.7.1	Die identity-Funktion	67
5.7.2	Die eye-Funktion	68
5.8	Aufgaben	69
5.9	Lösungen.....	70
6	Datentyp-Objekt: dtype	73
6.1	dtype.....	73
6.2	Strukturierte Arrays.....	75
6.3	Ein- und Ausgabe von strukturierten Arrays	77
6.4	Unicode-Strings in Arrays	79
6.5	Umbenennen von Spaltennamen	79
6.6	Spaltenwerte austauschen.....	80
6.7	Komplexeres Beispiel.....	80
6.8	Aufgaben	82
6.9	Lösungen.....	83
7	Dimensionsänderungen.....	85
7.1	Reduktion und Reshape von Arrays	85
7.1.1	flatten	86
7.1.2	ravel	86
7.1.3	Unterschiede zwischen ravel und flatten	87
7.1.4	reshape	88
7.2	Konkatenation von Arrays	89
7.3	Weitere Dimensionen hinzufügen.....	90
7.4	Vektoren stapeln	91
7.5	„Fliesen“ mit „tile“	92

8	Numerische Operationen auf NumPy-Arrays	95
8.1	Operatoren und Skalare	95
8.2	Arithmetische Operationen auf zwei Arrays	97
8.3	Matrizenmultiplikation und Skalarprodukt	98
8.3.1	Definition der dot-Funktion	98
8.3.2	Beispiele zur dot-Funktion	98
8.3.3	Das dot-Produkt im 3-dimensionalen Fall	100
8.4	Vergleichsoperatoren	105
8.5	Logische Operatoren	106
8.6	Broadcasting	106
8.6.1	Zeilenweises Broadcasting	107
8.6.2	Spaltenweises Broadcasting	109
8.6.3	Broadcasting von zwei eindimensionalen Arrays	110
8.7	Distanzmatrix	111
8.8	Ufuncs	112
8.8.1	Anwendung von Ufuncs	112
8.8.2	Parameter für Rückgabewerte bei Ufuncs	114
8.8.3	accumulate	115
8.8.4	reduce	116
8.8.5	outer	117
8.8.6	at	118
8.9	Aufgaben	118
8.10	Lösungen	119
9	Statistik und Wahrscheinlichkeiten	121
9.1	Einführung	121
9.2	Zufallszahlen mit Python	122
9.2.1	Die Module random und secrets	122
9.2.2	Erzeugen einer Liste von Zufallszahlen	122
9.3	Zufällige Integer-Zahlen mit Python	123
9.4	Stichproben/Auswahlen	126
9.5	Zufallsintervalle	127
9.6	Gewichtete Zufallsauswahl	128
9.7	Stichproben mit Python	130
9.8	Kartesische Auswahl	132
9.8.1	Kartesisches Produkt	132
9.8.2	Kartesische Auswahl: cartesian_choice	133
9.9	Echte Zufallszahlen	135

9.10	Seed/Startwert	136
9.11	Gauss'sche Normalverteilung	137
9.12	Übung mit Binärsender	139
9.13	Synthetische Verkaufszahlen	141
9.14	Aufgaben	143
9.15	Lösungen.....	145
10	Boolesche Maskierung und Indizierung	151
10.1	Fancy-Indizierung	153
10.2	Indizierung mit einem Integer-Array.....	153
10.2.1	Übung.....	154
10.2.2	Lösung	154
10.3	nonzero und where	154
10.3.1	Übung.....	155
10.3.2	Lösung	156
10.3.3	Flatnonzero und count_nonzero.....	156
11	Lesen und Schreiben von Datendateien	157
11.1	Textdateien speichern mit savetxt	157
11.2	Textdateien laden mit loadtxt	159
11.2.1	loadtxt ohne Parameter.....	159
11.2.2	Spezielle Trenner	159
11.2.3	Selektives Einlesen von Spalten	159
11.2.4	Datenkonvertierung beim Einlesen	160
11.3	tofile	161
11.4	fromfile	162
11.5	Best Practice, um Daten zu laden und zu speichern	164
11.6	Und noch ein anderer Weg: genfromtxt	164

Teil III Matplotlib.....165

12	Einführung	167
12.1	Ein erstes Beispiel.....	168
12.2	Der Formatparameter von pyplot.plot.....	169
12.3	Bezeichnungen für die Achsen.....	172
12.4	Abfragen und Ändern des Wertebereichs der Achsen.....	174
12.5	„linspace“ zur Definition von X-Werten	175
12.6	Linienstil ändern.....	177
12.7	Flächen einfärben.....	178

13	Achsen- und Skalenteilung	181
13.1	Achsenverschiebungen und Achsenbezeichnungen	181
13.2	Verändern der Achsenbeschriftungen	186
13.3	Justierung der Tick-Beschriftungen	187
14	Legenden und Kommentare hinzufügen	189
14.1	Legende hinzufügen	189
14.2	Kommentare	194
15	Mehrfache Plots und Doppelachsen	199
15.1	Mehrere Abbildungen und Achsen	199
15.2	Unterdiagramm mit gridspec	208
15.3	Arbeiten mit Objekten	211
15.4	Ein Plot innerhalb eines anderen Plots	213
15.5	Setzen des Plotbereichs	214
15.6	Logarithmische Darstellung	215
15.7	Sekundäre Y-Achse	216
15.8	Gitterlinien	217
15.9	Abbildungen speichern	218
15.10	Aufgaben	219
15.11	Lösungen	219
16	Konturplots	223
16.1	Erstellen eines Maschengitters	223
16.2	Berechnung der Werte	225
16.3	Linienstil und Farben anpassen	227
16.4	Gefüllte Konturen	228
16.5	Individuelle Farben	229
16.6	Schwellen	230
16.7	Andere Grids	231
16.7.1	Meshgrid genauer	231
16.7.2	mgrid	235
16.7.3	ogrid	235
16.8	Aufgaben	236
16.9	Lösungen	237
17	Balken-, Säulendiagramme und Histogramme	241
17.1	Histogramme	241
17.2	Säulendiagramm	247

17.3	Balkendiagramme	249
17.4	Aufgaben	250
17.5	Lösung.....	250

Teil IV Pandas251

18 Einführung in Pandas 253

18.1	Datenstrukturen	253
18.2	Series	254
18.2.1	Indizierung.....	256
18.2.2	pandas.Series.apply	257
18.2.3	Zusammenhang zu Dictionaries	258
18.3	NaN – Fehlende Daten	258
18.3.1	Die Methoden isnull() und notnull()	259
18.3.2	Zusammenhang zwischen NaN und None	260
18.3.3	Fehlende Daten filtern.....	260
18.3.4	Fehlende Daten auffüllen	261

19 DataFrame 263

19.1	Zusammenhang zu Series	263
19.2	Manipulation der Spaltennamen	264
19.3	Zugriff auf Spalten	265
19.4	DataFrames aus Dictionaries	266
19.5	Index ändern	267
19.5.1	Umsortierung der Spalten	267
19.5.2	Spalte in Index umfunktionieren.....	269
19.6	Selektion von Zeilen	270
19.7	Summen und kumulative Summen	271
19.8	Spaltenwerte ersetzen	271
19.9	Sortierung.....	273
19.10	Spalten einfügen	274
19.11	DataFrame und verschachtelte Dictionaries.....	275
19.12	Aufgaben	276
19.13	Lösungen.....	278

20 Dateien lesen und schreiben 283

20.1	Trennerseparierte Werte	283
20.2	CSV- und DSV-Dateien lesen	284
20.3	Schreiben von CSV-Dateien	285

20.4	Lesen und Schreiben von Excel-Dateien	287
20.5	Aufgaben	288
20.6	Lösungen.....	288
21	Umgang mit NaN	291
21.1	'nan' in Python	291
21.2	NaN in Pandas	292
21.2.1	Beispiel mit NaNs	294
21.3	dropna() verwenden.....	295
21.4	Aufgaben	297
21.5	Lösungen.....	297
22	Binning	299
22.1	Einführung	299
22.2	Binning mit Pandas	301
22.2.1	Von Pandas verwendete Bins	301
22.2.2	Andere Wege, um Bins zu definieren	302
22.2.3	Bins und Werte zählen.....	303
22.2.4	Bins benennen.....	305
23	Mehrstufige Indizierung	307
23.1	Einführung	307
23.2	Mehrstufig indizierte Series	307
23.3	Zugriffsmöglichkeiten	309
23.4	Zusammenhang zu DataFrames	310
23.5	Dreistufige Indizes	312
23.6	Vertauschen mehrstufiger Indizes	314
23.7	Aufgaben	315
23.8	Lösungen.....	316
24	Datenvisualisierung mit Pandas	317
24.1	Einführung	317
24.2	Liniendiagramme in Pandas	318
24.2.1	Series	318
24.2.2	DataFrames	320
24.2.3	Sekundärachsen (Twin Axes)	322
24.2.4	Mehrere Y-Achsen	323
24.3	Ein komplexeres Beispiel	325
24.3.1	Spalten mit Zeichenketten (Strings) in Floats wandeln	328

24.4	Balkendiagramme in Pandas	329
24.4.1	Ein einfaches Beispiel.....	329
24.4.2	Balkengrafik für die Programmiersprachennutzung	330
24.4.3	Farbgebung einer Balkengrafik	331
24.5	Kuchendiagramme in Pandas	332
24.5.1	Ein einfaches Beispiel.....	332
25	Zeit und Datum	337
25.1	Einführung.....	337
25.2	Python-Standardmodule für Zeitdaten	338
25.2.1	Die date-Klasse	338
25.2.2	Die time-Klasse	339
25.3	Die datetime-Klasse	340
25.4	Unterschied zwischen Zeiten	342
25.4.1	Wandlung von datetime-Objekten in Strings.....	343
25.4.2	Wandlung mit strftime.....	343
25.5	Ausgabe in Landessprache	344
25.6	datetime-Objekte aus Strings erstellen	345
26	Zeitserien	347
26.1	Einführung.....	347
26.2	Zeitreihen und Python	347
26.3	Datumsbereiche erstellen	350
	Stichwortverzeichnis	353

Vorwort

Eine der treibenden Kräfte in der weltweiten Softwareentwicklung wird wohl am besten durch die beiden populären Begriffe „Big Data“ und „Maschinelles Lernen“ beschrieben. Immer mehr Institute und Firmen betätigen sich in diesen Feldern. Für diese und auch für individuelle Personen, die in diesen Bereichen tätig werden wollen, ist eine der bedeutendsten Fragen – wenn nicht gar die bedeutendste Frage –, was die geeignetste Programmiersprache zu diesem Zweck ist. In vielen Umfragen wird Python als beste oder auch als beliebteste Programmiersprache genannt.

Python war ursprünglich nicht für numerische Probleme ausgerichtet gewesen. Die Erfolgsstory von Python wurde erst möglich durch die Module NumPy, SciPy, Matplotlib und Pandas. Dieses Buch bietet eine umfassende Einführung in die Module NumPy, Matplotlib und Pandas, setzt aber grundlegende Kenntnisse von Python voraus. Somit ergänzt es in idealer Weise das Buch „Einführung in Python 3: Für Ein- und Umsteiger“ von Bernd Klein.

Brigitte Bauer-Schiewek, Lektorin

Danksagung

Zum Schreiben eines Buches benötigt es neben der nötigen Erfahrung und Kompetenz im Fachgebiet vor allem viel Zeit. Zeit außerhalb des üblichen Rahmens. Zeit, die vor allem die Familie mitzutragen hat. Deshalb gilt mein besonderer Dank meiner Frau Karola, die mich während dieser Zeit tatkräftig unterstützt hat.

Außerdem danke ich den zahlreichen Teilnehmerinnen und Teilnehmern an meinen Python-Kursen, die mir geholfen haben, meine didaktischen und fachlichen Kenntnisse kontinuierlich zu verbessern. Ebenso möchte ich den Besucherinnen und Besuchern meiner Online-Tutorials unter www.python-kurs.eu und www.python-course.eu danken, vor allem jenen, die sich mit konstruktiven Anmerkungen bei mir gemeldet haben. Ebenso danke ich Herrn Wilhelm Wall für seine wertvolle Hilfe, indem er die Kapitel fachlich auf Fehler überprüft hat.

Zuletzt danke ich auch ganz herzlich dem Hanser Verlag, der dieses Buch ermöglicht hat. Vor allem danke ich Frau Brigitte Bauer-Schiewek, Programmplanung Computerbuch, für die kontinuierliche ausgezeichnete Unterstützung. LaTeX ist ein fantastisches System, um Bücher zu schreiben, aber ohne die technische Unterstützung von Herrn Stephan Korell und Frau Irene Weilhart bei besonderen LaTeX-Problemen wäre ich manchmal vielleicht verzweifelt. Herrn Jürgen Dubau danke ich fürs Lektorat.

Bernd Klein, Singen

■ 1.1 Die richtige Wahl

Sich für die richtige Programmiersprache für die tägliche Arbeit zu entscheiden, ist von hoher Bedeutung. Diese Entscheidung hängt von vielen Faktoren ab. Oft ist es so, dass man gar keine Wahl hat. Das Institut oder die Firma geben einem bereits eine Sprache vor. Dies kann dazu führen, dass man beispielsweise mit Perl, Delphi oder VBA arbeiten muss, also eine der unbeliebtesten Programmiersprachen der Welt, wie eine Auswertung von Stackoverflow im Oktober 2017 herausgefunden hatte.¹ Vielleicht hat man aber auch das – wie wir es sehen – Glück und darf mit Python arbeiten. Schaut man sich die Umfrageergebnisse zu den beliebtesten Programmiersprachen an, so findet man Python immer an erster Stelle oder ganz vorne. Nach dem von TIOBE² berechneten Ranking der beliebtesten Programmiersprachen konnte Python im Dezember 2018 den Platz 3 erklimmen. Diese Position konnte es im Februar 2019 sogar um 2,4 % verbessern! TIOBE kürte Python im Jahr 2018 zur Programmiersprache des Jahres.³ Dabei wird der Platz zwei von C eingenommen. Wenn man bedenkt, dass der größte Teil der Python-Erweiterungen in C geschrieben ist, dann führt ein Wachstum von Python damit indirekt auch zu Zuwächsen bei C. Platz eins wird unangefochten von Java angeführt. Allerdings gedeiht und wächst Java in Gebieten⁴, in denen es in keiner Konkurrenz zu Python steht. Auch bei anderen Portalen zeichnen sich ähnliche oder auch noch schönere Bilder für Python ab: Beispielsweise sieht PYPL PopularitY Python auf dem ersten Platz mit 25,95 %.⁵

Sicherlich ist es toll zu wissen, dass die Programmiersprache, die man selbst nutzt, sich auch bei anderen großer Beliebtheit erfreut und von vielen oder gar den meisten im eigenen Arbeitsgebiet genutzt wird. Aber eine der wichtigsten Fragen lautet: Lassen sich mit Python eigene Projekte einfacher und besser als mit anderen Programmiersprachen

¹ <https://stackoverflow.blog/2017/10/31/disliked-programming-languages/>

² Bei dem TIOBE-Index des niederländischen Unternehmens TIOBE Software BV handelt es sich um ein seit 2001 publiziertes und monatlich aktualisiertes Ranking von Programmiersprachen nach ihrer Popularität. Der Index wird jeden Monat aktualisiert. Der Listenplatz einer Sprache ergibt sich aus der Häufigkeit von Treffern bei der Suche nach dem Namen dieser Programmiersprache in den wichtigsten Suchmaschinen wie Google, Bing, Yahoo! und so weiter. Dies bedeutet also nicht, dass es sich um ein Ranking der „besten“ Programmiersprachen oder der Sprachen mit den meisten Codezeilen handelt!

³ <https://www.tiobe.com/tiobe-index/>

⁴ So ist die Laufzeitumgebung des Android-Betriebssystems (ART, Android Runtime) in Java geschrieben.

⁵ <http://pypl.github.io/PYPL.html>

lösen? Unter „einfacher und besser“ verbergen sich natürlich Begriffe wie „Entwicklungszeit“, „Laufzeit“, „Wartbarkeit“ und so weiter.

Programmiersprachen sind wie Schuhe. Es gibt nicht den Schuh für alle Fälle. Einen Schuh, den man sowohl für feierliche Anlässe oder im Büro als auch beim Sport oder bei Wanderungen tragen kann. Python ist jedoch eine Sprache, die sich universell in den meisten Gebieten einsetzen lässt.

Man kann sich nun die berechtigte Frage stellen, worauf dieser große Erfolg von Python beruht. Zu den Hauptgründen für die Beliebtheit und die Verbreitung von Python zählt auf jeden Fall die Benutzung von Python bei „Big Data“ und dem „Maschinellen Lernen“. Zwei Gebiete, in denen es um die Lösungen von numerischen Problemen geht. Betrachtet man jedoch das reine Python, so wie es Anfang der 90er Jahre designt und implementiert worden ist, stellt man rasch fest, dass diese Sprache zur Lösung von numerischen Problemen eigentlich denkbar ungeeignet ist. Anfänglich lag der Fokus von Python keinesfalls auf der numerischen Programmierung. Was natürlich nicht bedeutet, dass man keine numerischen Probleme lösen konnte. Python stellte mit Listen und Dictionaries fantastische Datenstrukturen zur effizienten Lösung von vielfältigen allgemeinen Problemen dar. Allerdings sind diese Datenstrukturen völlig ungeeignet, numerische Berechnung, wie man sie beispielsweise im maschinellen Lernen benötigt, effizient zu implementieren. Listen sind für diese Aufgaben im Vergleich zu effizienten Array-Implementierungen zum einen extrem langsam und verschwenden zum anderen extreme Mengen von Speicherplatz. So können Array-Implementierungen beispielsweise fünfzig- bis hundertmal so langsam sein und gleichzeitig einen etwa dreißigfach größeren Speicherplatzbedarf haben. Die Lösung hierfür bieten die Module NumPy, SciPy, Matplotlib und Pandas. So stellt NumPy Datenstrukturen zur Verfügung, die um den Faktor 10 bis 100 schneller sind als Implementierungen in reinem Python oder anderen numerisch ungeeigneten Programmiersprachen.

■ 1.2 Aufbau des Buches

In diesem Buch geht es um Python und seine hervorragenden Möglichkeiten zum Einsatz bei numerischen Problemen. Also damit um die Module, die für „Big Data“ und das „Maschinelle Lernen“ unabdinglich sind. Auch wenn das Buch mit einer kompletten, aber sehr knapp gehaltenen Einführung in die Grundlagen von Python beginnt, sind Grundkenntnisse in Python beim Lesen dieses Buches von großem Vorteil. Grundkenntnisse, wie man sie beispielsweise in meinem Buch „Einführung in Python 3: Für Ein- und Umsteiger“⁶ erwerben kann. Die Schwerpunkte des vorliegenden Buches liegen auf den Modulen NumPy, Matplotlib und Pandas. Genau die Module, die einen wesentlichen Anteil zum steilen Aufstieg von Python im Ranking der beliebtesten Programmiersprachen beigetragen haben.

⁶ Bernd Klein, Einführung in Python 3: Für Ein- und Umsteiger, Carl Hanser Verlag GmbH & Co. KG; Auflage: 3., überarbeitete (6. November 2017)

■ 1.3 Python-Installation

Wir gehen davon aus, dass Python bei den Leserinnen und Lesern des Buches bereits installiert ist, insbesondere mit den genannten Modulen NumPy, Matplotlib und Pandas. Sollte dies nicht der Fall sein, empfehlen wir die Installation von Anaconda⁷, die sich auf allen Betriebssystemen äußerst einfach gestaltet. Damit ist alles installiert, was im Laufe des Buches benötigt wird.

■ 1.4 Download der Beispiele

Alle im Buch verwendeten Beispiele finden Sie zum Download unter

http://www.python-kurs.eu/buecher/numerical_python/

Dort findet sich auch ein Korrekturverzeichnis.

■ 1.5 Anregungen und Kritik

Falls Sie glauben, eine Ungenauigkeit oder einen Fehler im Buch gefunden zu haben, können Sie auch gerne eine E-Mail direkt an den Autor schicken: klein@python-kurs.eu.

Natürlich gilt dies auch, wenn Sie Anregungen oder Wünsche zum Buch geben wollen. Leider können wir jedoch – so gerne wir es auch tun würden – keine individuellen Hilfen zu speziellen Problemen geben.

Wir werden versuchen, Fehler und Anmerkungen in kommenden Auflagen zu berücksichtigen. Selbstverständlich aktualisieren wir damit auch unsere Informationen unter

http://www.python-kurs.eu/buecher/numerical_python/

Ich wünsche Ihnen viel Spaß und Erfolg beim Durcharbeiten dieses Buches!

Bernd Klein, Februar 2019

⁷ <https://www.anaconda.com/distribution/>

2

Numerisches Programmieren mit Python

■ 2.1 Definition von numerischer Programmierung

Der Titel unseres Buches lautet „Numerisches Python“, was eine Anspielung auf „Numerisches Programmieren“ ist.

Der Ausdruck „numerisches Programmieren“ – auch bekannt unter dem Begriff „wissenschaftliches Programmieren“ – ist irreführend. Man könnte es als eine Programmierung ansehen, die mit Zahlen statt mit z.B. Texten zu tun hat. Letztendlich haben die meisten Programme, auch wenn sie scheinbar nichts mit Zahlen zu tun haben, einen numerischen Kern. Denkt man beispielsweise an den Google-Algorithmus und an die Art, wie er einem auf eine Suchanfrage Vorschläge zu Webseiten offeriert, dann könnte man glauben, dass es sich bei dem zugrunde liegenden Algorithmus um reine Textverarbeitung handelt. Dennoch ist auch in diesem Fall der Kern bzw. der wesentliche Teil des Algorithmus ein numerisches Problem. Um seinen PageRanking-Algorithmus (d.h. die Bewertung der Webseiten) durchzuführen, lässt Google die größte jemals von Menschen erdachte Matrix berechnen.

So könnte man denken, dass es sich letztendlich bei jedem Programm um numerische Programmierung handelt, aber es gibt auch eine engere Definition.

Unter numerischer Programmierung versteht man das Gebiet der Informatik und der Mathematik, in dem es um Approximationsalgorithmen geht, d.h. die numerische Approximation von mathematischen Problemen oder numerischer Analysis. In anderen Worten: Probleme mit stetigen Variablen.

In unserem Buch haben wir insbesondere die numerischen Verfahren im Fokus, die in den Gebieten „Data Science“ und „Maschinelles Lernen“ besonders benötigt werden.

Python gehört zu den wichtigsten und am häufigsten benutzten Programmiersprachen in diesem Gebiet. Allerdings würde Python keine Rolle spielen, wenn es nicht mächtige Module zur numerischen Programmierung zur Verfügung stellte, die wir im Folgenden beschreiben werden.

■ 2.2 Zusammenhang zwischen Python, NumPy, Matplotlib, SciPy und Pandas

The image is a collage of mathematical formulas. At the top, it shows a vector $\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ and a partial derivative $\frac{\partial E}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \frac{1}{2} \sum_{j=1}^n (t_j - o_j)^2$. Below this, it shows a vector $\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$ and a partial derivative $\frac{\partial E}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_{j=1}^n (t_j - o_j)^2$. At the bottom, it shows a probability distribution formula: $\frac{1}{P(c|d)} = \frac{P(c_1) \prod_{j=1}^d P(c_j)}{P(c) \prod_{j=1}^d P(c_j)}$.

Python ist eine universelle Programmiersprache, die sich in den unterschiedlichsten Gebieten einsetzen lässt. So zum Beispiel in der Systemadministration, als Tool zur Erzeugung und zum Betrieb von dynamischen Webseiten und in der Computerlinguistik. Da Python ein universelle Programmiersprache ist, lässt sie sich natürlich auch zum Lösen numerischer Probleme einsetzen. So weit so gut, aber die Crux bei der Sache liegt in der Laufzeit und auch im Speicherverbrauch. Reines Python – also ohne den Einsatz irgendwelcher numerischer Spezialmodule – würde sich nicht eignen für Aufgaben, für die Matlab und R geschaffen worden sind. Sobald es um die Lösung numerischer Probleme geht, ist die Leistungsfähigkeit von Algorithmen von höchster Wichtigkeit, sowohl was die Geschwindigkeit als auch den Speicherverbrauch betrifft.

Nutzen wir Python in Kombination mit seinen Modulen NumPy, SciPy, MATPLOTLIB und Pandas, dann gehört die Sprache zu den führenden numerischen Programmiersprachen. Sie ist so effizient, wenn nicht gar effizienter, als Matlab und R.

NumPy ist ein Modul, welches die grundlegenden Datenstrukturen zur Verfügung stellt, die auch von MATPLOTLIB, SciPy und Pandas benutzt werden. NumPy implementiert mehrdimensionale Arrays und Matrizen. Außerdem gibt es den Nutzerinnen und Nutzern auch die wesentlichen Funktionalitäten an die Hand, mit denen sich diese Datenstrukturen erzeugen und manipulieren lassen.

SciPy baut auf NumPy auf, d.h. es benutzt die Datenstrukturen, die NumPy bereitstellt. Es erweitert die Leistungsfähigkeit von NumPy mit weiteren nützlichen Funktionalitäten wie beispielsweise Minimierung, Regression, Fourier-Transformation und viele andere.

Die von Python-Programmen – mit oder ohne Verwendung von NumPy und SciPy – erzeugten Daten möchte man häufig gerne grafisch darstellen. Für diesen Zweck wurde das Modul MATPLOTLIB geschaffen.

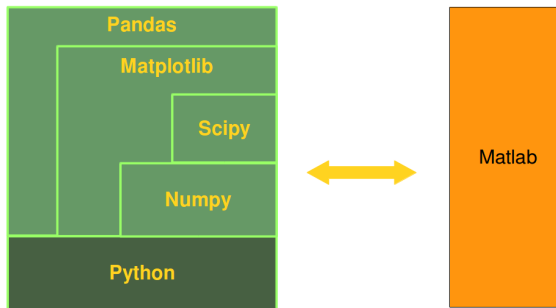
Das jüngste Kind in dieser Modulfamilie ist Pandas. Pandas benutzt alle bisher genannten Module und ist auf diesen aufgebaut. Der Fokus von Pandas besteht darin, Datenstrukturen und Operationen zur Manipulation von Tabellen und Zeitreihen bereitzustellen. Der Name ist von „panel data“ abgeleitet. Pandas ist bestens geeignet, mit Tabellendaten zu arbeiten, wie sie beispielsweise von Excel erzeugt werden.

■ 2.3 Python, eine Alternative zu Matlab

Python entwickelt sich mehr und mehr zur Hauptprogrammiersprache von Data Scientists. Dennoch plagt viele Wissenschaftler und Ingenieure die Frage, welche Sprache sie nutzen sollten, um ihre Probleme zu lösen. Die Hauptkonkurrenten im Gebiet des Maschinellen Lernens sind sicherlich Python, R und MATLAB.

Bei der Entwicklung von R hatte man Statistiker und Data Scientists im Visier, wollte aber keinesfalls eine Sprache entwickeln, die sich generell einsetzen lässt. Dies gilt ebenso für MATLAB. Python hingegen wurde von Anfang an als universelle Programmiersprache ausgerichtet. Zusätzlich eignet sich Python in Kombination mit den Modulen NumPy, SciPy, MATPLOTLIB und Pandas bestens, um R oder MATLAB zu ersetzen.

Einer der wesentlichen Nachteile von MATLAB gegenüber Python sind wohl die Kosten. Python mit all seinen Modulen ist kostenlos, wohingegen MATLAB recht teuer ist und je nach eingesetzter Toolbox extrem teuer sein kann. Bei Python handelt es sich aber nicht nur um kostenlose, sondern auch um „freie“ Software, d.h. ihr Einsatz ist nicht durch prohibitive Lizenzmodelle eingeschränkt.



Teil I

Kurze Einführung in Python

3

Kurze Einführung in Python

■ 3.1 Datenstrukturen

3.1.1 Zahlen und Variablen

Wenn man die interaktive Shell von Python oder ipython startet, kann man sofort arithmetische Ausdrücke eingeben:

```
>>> 5 * 8.6 - 4 ** 2
27.0
```

Möchte man Werte speichern, so kann man dies in der Shell ebenso wie in einem Programm mittels Variablenzuweisungen bewerkstelligen. Dies geschieht ebenso wie in den meisten anderen Programmiersprachen durch ein Gleichheitszeichen:

```
>>> x = 42
>>> y = 10
>>> print(x - y)
32
```

Wie wir im vorigen Beispiel gesehen haben, ist eine Typdeklaration dazu nicht nur nicht erforderlich, sondern auch nicht möglich. Python kennt keine Typdeklarationen. Python-Variablen stellen Referenzen auf beliebige Objekte dar.

Auch wenn den Variablennamen keine Typen zugeordnet sind, so entspricht jedes in Python definierte Objekt einem Typ oder genauer gesagt der Instanz einer Klasse.

```
>>> x = 42
>>> type(x)
<class 'int'>
```

Wir sehen also, dass eine Integer-Zahl „42“ angelegt worden ist. Diese Integer-Zahl ist ein Objekt der Integer-Klasse. Als Integers oder ganze Zahlen bezeichnet man in der Mathematik die Zahlen

..., -3, -2, -1, 0, 1, 2, 3, ...

Das heißt, die Zahlen gehen von minus unendlich bis unendlich. Selbstverständlich können wir „unendlich“ in „int“ nicht darstellen, aber die Zahlen in Python können extrem groß bzw. extrem klein werden:


```
>>> x = 2 ** 64
>>> x
18446744073709551616
>>> len(str(x))
20
>>> x = 2 ** (2 ** 22)
>>> len(str(x))
1262612
```

Benutzt man einen Variablennamen, der nicht definiert ist, erzeugt man eine Ausnahme:

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

Im folgenden Beispiel zeigen wir, dass Variablen Referenzen auf Objekte darstellen. Dazu benutzen wir die Funktion „id“, deren Rückgabewert ein Integer-Wert ist, der die Objekte eindeutig identifiziert. Wir sehen, dass das Integer-Objekt 10 nur einmal erzeugt wird und dass x und y beide dieses Objekt nach der Zuweisung „y = x“ referenzieren.

```
>>> x = 10
>>> id(x)
10106112
>>> y = x
>>> id(y)
10106112
>>> y = 12
>>> id(y)
10106176
>>> id(x)
10106112
```

Neben `int` kennt Python auch Float-Zahlen (`float`), die in Ausdrücken auch mit Integer-Zahlen verknüpft werden können:

```
>>> f = 3.5
>>> type(f)
<class 'float'>
>>> f2 = 878.323
>>> x = f * 10
>>> x
35.0
```

3.1.2 Zeichenketten/Strings

Ein weiterer wichtiger Datentyp in Python sind die Zeichenketten, die man meist auch als Strings bezeichnet. Strings können auf verschiedene Arten definiert werden: mit einfachen Anführungszeichen, mit doppelten Anführungszeichen oder mit drei einfachen bzw. drei doppelten Anführungszeichen. Wir demonstrieren diese Varianten im folgenden Beispiel. Wir verwenden auch das Nummernzeichen „#“, um Kommentare einzuleiten:

```
>>> 'ein String in einfachen Anführungszeichen'
'ein String in einfachen Anführungszeichen'
>>> 'Miller\s son' # der Rückwärtsschrägstrich fungiert wie üblich als Escape-
Zeichen
'Miller\\s son'
```

```
>>> "Miller's son" # jetzt brauchen wir kein Escape-Zeichen
"Miller's son"
>>> print("""Strings mit drei Anführungszeichen können
... über mehrere
... Zeilen gehen und enden erst, wenn
... drei Anführungszeichen kommen""")
Strings mit drei Anführungszeichen können
über mehrere
Zeilen gehen und enden erst, wenn
drei Anführungszeichen kommen
>>>
```

Bei den Strings in dreifachen Anführungszeichen wartet die Shell noch mit einer Besonderheit auf, wenn wir während der Definition eines solchen Strings die Return-Taste tippen. Die nächste Zeile wird dann nicht mit dem üblichen Prompt „>>> ” eingeleitet, sondern mit einem aus drei Punkten bestehenden Prompt „... ”. Damit signalisiert uns die Shell, dass der String noch nicht fertig ist. Der String ist erst fertig, wenn wieder entsprechend drei einfache (' ') bzw. doppelte (" ") Anführungszeichen eingegeben werden.

Strings können indiziert werden, dabei entspricht das erste Zeichen dem Index 0. Das letzte Zeichen eines Strings können wir mit dem Index - 1 ansprechen, d.h. mit negativen Zahlen erhält man eine Indizierung von rechts:

```
>>> language = "Python"
>>> language[0] # 1. Zeichen des Strings an der Stelle 0
'p'
>>> language[2] # 3. Zeichen des Strings an der Stelle 2
't'
>>> language[-1] # letztes Zeichen
'n'
>>> language[-2] # vorletztes Zeichen
'o'
```

Während man mit dem Indizieren nur einzelne Zeichen, also Strings der Länge 1, aus einem String erhalten kann, ermöglicht der Teilbereichsoperator (Slicing) das „Heraus-schneiden“ von beliebigen Teilstrings:

```
>>> s = "Ein grüner Fisch singt nie schräg!"
>>> s[4:10] # von Position 4 (inklusive) bis Position 10 (exklusive)
'grüner'
>>> s[-7:] # von Position -7 bis zum Ende des Strings
'schräg!'
>>> s[:16] # von Anfang bis zur Position 16
'Ein grüner Fisch'
```

Finden von Teilstrings in Strings:

```
>>> s = "A horse, a horse! "
>>> s += "My kingdom for a horse!"
>>> s
'A horse, a horse! My kingdom for a horse!'
>>> s.find("horse")
2
>>> s.find("horse", 3)
11
>>> s.find("horse", 16)
35
```

```
>>> s.find("horse", 36)
-1
>>> s.rfind("horse")    # Suche beginnt von hinten
35
```

Die Methode `index` liefert die gleichen Ergebnisse, außer wenn der Suchstring nicht im String vorkommt:

```
>>> s.index("horse")
2
>>> s.index("horse", 3)
11
>>> s.index("cow")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Ein String kann mit Methode `split` auch in eine Liste von Teilstrings aufgespalten werden. Ohne Parameter dient jede Folge von Leerzeichen (auch `\n`, `\t`, `\r` usw. sind Leerzeichen) als Trenner. Wird ein String als Parameter übergeben, wird dieser als Trenner verwendet. Außerdem kann man noch einen weiteren Parameter übergeben, mit dem man die Anzahl der zu spaltenden Teilstrings angeben kann:

```
>>> s = "The    is \n a\tstring."
>>> s.split()
['The', 'is', 'a', 'string.']
>>> t = "34,12.3,90,0,,1"
>>> t.split(",")
['34', '12.3', '90', '0', '', '1']
>>> t.split(",", 2)
['34', '12.3', '90,0,,1']
>>> t.split(",", maxsplit=2)
['34', '12.3', '90,0,,1']
>>> t.split(sep=" ", maxsplit=2)
['34', '12.3', '90,0,,1']
>>> s.split(maxsplit=2)
['The', 'is', 'a\tstring.']
>>> # aufspalten von hinten:
...
>>> t.rsplit(sep=" ", maxsplit=2)
['34,12.3,90,0', '', '1']
>>> s.rsplit(maxsplit=2)
['The    is', 'a', 'string.']
```

Mit den Methoden `lstrip`, `rstrip` und `strip` ist es möglich, Leerzeichen oder auch andere Zeichen vom linken bzw. rechten Rand eines Strings oder beidseitig zu entfernen:

```
>>> s = "\n2000 Hamburg\r\n"
>>> s.strip()
'2000 Hamburg'
>>> s.strip("\n\r0123456789 ")
'Hamburg'
>>> s.rstrip()
'\n2000 Hamburg'
>>> s.lstrip()
'2000 Hamburg\r\n'
```

3.1.3 Listen

Listen werden mittels eckiger Klammern in Python erzeugt. Eine Liste kann beliebige Python-Objekte enthalten, die durch Komma getrennt sind:

```
>>> lst = ["rot", "grün", "blau"]
>>> lst2 = ["rot", 12, [3, 6.78]]
```

Auf Listenelemente kann man wie bei Strings über Indices zugreifen oder man greift auf mehrere Listenelemente mit dem Teilbereichsoperator zu. Außerdem können wir mittels Indizierung der Liste auch neue Werte zuweisen:

```
>>> lst = ["rot", 12, "gelb", 123, [3, "Noch ein String"]]
>>> lst[0]
'rot'
>>> lst[4]
[3, 'Noch ein String']
>>> lst[4][1]
'Noch ein String'
>>> lst[1:4]
[12, 'gelb', 123]
>>> lst[0] = "orange"
>>> lst
['orange', 12, 'gelb', 123, [3, 'Noch ein String']]
```

Wir können prüfen, ob ein Element in einer Liste vorhanden ist:

```
>>> farben = ["rot", "grün", "blau", "gelb"]
>>> "rot" in farben
True
>>> "braun" in farben
False
>>> "grau" not in farben
True
```

Anhängen von Objekten an Listen:

```
>>> farben = ["rot", "grün", "blau", "gelb"]
>>> farben.append("silber")
>>> farben
['rot', 'grün', 'blau', 'gelb', 'silber']
```

Kopieren und Konkatenieren von Listen:

```
>>> farben1 = ["rot", "grün"]
>>> farben2 = ["blau", "gelb"]
>>> farben = farben1 + farben2
>>> farben
['rot', 'grün', 'blau', 'gelb']
>>>
>>> farben1 = ["rot", "grün"]
>>> farben2 = ["blau", "gelb"]
>>> farben = farben1.copy()
>>> farben.extend(farben2)
>>> farben
['rot', 'grün', 'blau', 'gelb']
```

Häufig muss man auch Elemente mit einem bestimmten Index aus Listen entfernen, dazu nutzt man meist die Methode pop:

```
>>> farben = ["rot", "grün", "blau", "gelb"]
>>> letztes_element = farben.pop() # das letzte Element wird entfernt
>>> letztes_element
'gelb'
>>> farben
['rot', 'grün', 'blau']
>>>
>>> erstes_element = farben.pop(0) # erstes Element wird entfernt
>>> erstes_element
'rot'
>>> farben
['grün', 'blau']
```

Möchte man ein bestimmtes Element entfernen, kann man dazu die Methode `remove` nutzen. Falls ein Objekt entfernt werden soll, welches nicht in der Liste enthalten ist, erhalten wir eine Fehlermeldung:

```
>>> farben = ["rot", "grün", "rot", "blau"]
>>> farben.remove("rot") # erstes Vorkommen von "rot" wird entfernt
>>> farben
['grün', 'rot', 'blau']
>>> farben.remove("rot")
>>> farben
['grün', 'blau']
>>> farben.remove("rot")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

3.1.4 Tupel

Tupel werden mit runden Klammern definiert. Allerdings können die Klammern auch weggelassen werden, wie wir in den folgenden Beispielen sehen:

```
>>> tage = ("Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag",
            "Sonntag")
>>> tage
('Montag', 'Dienstag', 'Mittwoch', 'Donnerstag', 'Freitag', 'Samstag', 'Sonntag')
>>> monate = "Frühling", "Sommer", "Herbst", "Winter" # auch ein Tupel
>>> monate
('Frühling', 'Sommer', 'Herbst', 'Winter')
>>>
>>> wochentage = tage[:5]
>>> wochentage
('Montag', 'Dienstag', 'Mittwoch', 'Donnerstag', 'Freitag')
```

Tupel werden auch für Mehrfachzuweisungen verwendet:

```
>>> x, y = 11, 33
>>> x, y
(11, 33)
>>> x, y = y, x # Werte werden vertauscht
>>> x, y
(33, 11)
```

Dabei werden zuerst die Ausdrücke auf der rechten Seite ausgewertet und dann den Variablen auf der linken Seite zugewiesen.

3.1.5 Frozensets und Mengen in Python

Ein set-Objekt enthält eine ungeordnete Sammlung von einmaligen und unveränderlichen Elementen, d.h. es gibt keine Mehrfachvorkommen von Elementen, d.h. Elemente können in set-Objekten nicht mehrfach vorkommen. Beim Datentyp „set“ handelt es sich um die Python-Implementierung von Mengen, wie wir sie aus der mathematischen Mengenlehre kennen.

Will man eine Menge erzeugen, so ist dies in Python sehr einfach. Wir bedienen uns der geschweiften Klammern, wie dies in der Mathematik üblich ist.

```
>>> staedte = {'Hamburg', 'München', 'Frankfurt', 'Berlin'}
>>> print(staedte)
{'München', 'Berlin', 'Frankfurt', 'Hamburg'}
>>> 'Berlin' in staedte
True
>>> 'Köln' in staedte
False
```

Bei der Ausgabe der Variablen `staedte` erkennen wir, dass die Elemente nicht in der Reihenfolge ausgegeben werden, in der wir Städte eingegeben hatten. Sets haben keine definierte Reihenfolge. Es gibt auch keinerlei Methoden, mit denen man Elemente von Mengen in Abhängigkeit einer Position bearbeiten könnte.

Sets lassen sich ideal einsetzen, wenn wir mehrfache Elemente aus Listen und Tupeln entfernen wollen. Allerdings macht dies nur Sinn, wenn uns die ursprüngliche Reihenfolge nicht mehr interessiert:

```
>>> t = ("a", "b", "b", "a", "c", "d", "a")
>>> s = set(t)
>>> s
{'d', 'a', 'c', 'b'}
>>> t = tuple(s)
>>> t
('d', 'a', 'c', 'b')
>>>
>>> s = "Dies ist ein String mit vielen Buchstaben"
>>> buchstaben = set(s.lower())
>>> buchstaben
{'m', 'e', 'h', 'u', 'c', 'b', 'v', 'r', 'i', 'n', 'd', ' ', 'a', 'l', 'g', 't', 's'}
```

Für die Elemente einer Menge gelten die gleichen Einschränkungen wie für die Schlüssel eines Dictionaries: Sets können nur unveränderliche Elemente enthalten, also Objekte vom Typ „int“, „float“, „complex“, „str“, „byte“, „frozen_set“ oder „tuple“.

Die Klasse „set“ bietet eine Vielzahl von Operationen und die aus der Mathematik üblichen Operationen auf Mengen. Wir demonstrieren die wichtigsten in den folgenden Beispielen:

```
>>> s = {"a", "b", "c"}
>>> s2 = {"b", "c", "e"}
>>> s3 = s - s2          # Differenzmenge von s und s2
>>> s3
{'a'}
>>> s | s2              # Vereinigungsmenge
{'e', 'c', 'b', 'a'}
>>> s & s2              # Schnittmenge
{'c', 'b'}
```

```

>>> s2.remove("e")      # Entfernen eines Elementes
>>> s2
{'c', 'b'}
>>> s2.add("x")         # Hinzufügen eines Elementes
>>> s2
{'x', 'c', 'b'}
>>> s2 < s              # Prüfung, ob s2 echte Teilmenge von s ist
False

```

Die Klasse „frozenset“ dient zur Erzeugung von unveränderlichen Mengen. Frozensets lassen sich deshalb auch als Schlüssel in Dictionaries und in Mengen verwenden. Sie werden erzeugt, indem man frozenset auf iterierbare Objekte wie z.B. Listen, Tupel oder Sets anwendet:

```

>>> x = frozenset({3, 5, 12})
>>> y = frozenset([33, 15, 112])
>>> m = {x, y}
>>> m
{frozenset({112, 33, 15}), frozenset({3, 12, 5})}
>>> d = {x:300, y:700}
>>> d
{frozenset({112, 33, 15}): 700, frozenset({3, 12, 5}): 300}

```

3.1.6 Dictionaries

Dictionaries gehören wohl zu den Datenstrukturen, die Python besonders attraktiv machen. Während bei Listen der Zugriff auf ein Element über die Position, also den Index erfolgt, geschieht dies bei Dictionaries über Schlüssel (engl. „keys“). Jedem Schlüssel ist ein Wert zugeordnet. Man kann sich ein Dictionary also als eine Menge von Schlüssel-Wert-Paaren vorstellen.

Die Definition und Arbeitsweise mit Dictionaries erläutern wir mit den folgenden Beispielen:

```

>>> empty_dict = {}      # leeres Dictionary
>>> len(empty_dict)      # Anzahl der Elemente
0
>>> haeufigkeit = {"a":4, "b":12, "c":20}
>>> haeufigkeit
{'a': 4, 'c': 20, 'b': 12}
>>> haeufigkeit["a"]      # Zugriff auf den Schlüssel "a"
4
>>> haeufigkeit.get("a")  # Zugriff auf den Schlüssel "a"
4
>>> list(haeufigkeit.items()) # Wandlung in Schlüssel-Wert-Liste
[('a', 4), ('c', 20), ('b', 12)]
>>> list(haeufigkeit.keys()) # Liste mit den Schlüssel
['a', 'c', 'b']
>>> list(haeufigkeit.values()) # Erzeugen einer Liste mit den Werten
[4, 20, 12]
>>> for key in haeufigkeit.keys(): # Iteration über die Schlüssel
...     print(y)
...
a
c
b

```

Die Schlüssel eines Dictionaries können beliebige unveränderliche Objekte sein, also Integers, Floats, Strings und Tupel. Als Werte können beliebige Objekte genutzt werden:

```
>>> staedte_GPS = { (52.520007, 13.404954):"Berlin", (47.767097, 8.872239):"
    Singen am Hohentwiel" }
>>> ports = {21:"File Transfer Protocol (FTP)", 22:"Secure Shell (SSH)", 23:"
    Telnet remote login service"}
>>> adressen = {"Henry":[ ("Henry", "Peterson"), 20016, "Hamburg"]}
```

■ 3.2 Kontrollstrukturen

3.2.1 Bedingte Anweisungen

Bedingte Anweisungen dienen dazu, den Programmfluss unter bestimmten Bedingungen verzweigen zu lassen:

```
x = int(input("Bitte eine ganze Zahl eingeben: "))

if x < -10:
    print("Die eingegebene Zahl ist kleiner als -10")
elif x < 0:
    print("Wert kleiner als 0 aber nicht kleiner als -10")
elif x == 0:
    print("0? Warum nicht :-)")
elif x < 10:
    print("Naja, so richtig groß ist die Zahl nicht!")
else:
    print("Die Zahl könnte ganz schön groß sein!")
    print("Aber ich habe es nicht getestet!")
```

Das Schlüsselwort „elif“ ist eine Kurzform für „else if“ und verhindert eine „ausufernde“ Verschachtelungstiefe, wie wir leicht sehen können, wenn wir das obige Programm ohne „elif“ umschreiben:

```
x = int(input("Bitte eine ganze Zahl eingeben: "))

if x < -10:
    print("Die eingegebene Zahl ist kleiner als -10")
else:
    if x < 0:
        print("Wert kleiner als 0 aber nicht kleiner als -10")
    else:
        if x == 0:
            print("0? Warum nicht :-)")
        else:
            if x < 10:
                print("Naja, so richtig groß ist die Zahl nicht!")
            else:
                print("Die Zahl könnte ganz schön groß sein!")
                print("Aber ich habe es nicht getestet!")
```


In Python-Programmen findet man häufig auch das ternäre if:

```
>>> temperatur = 25
>>> wertung = "warm" if temperatur > 20 else "frisch"
>>> wertung
'warm'
```

In diesem Beispiel ist das ternäre if eine abgekürzte Schreibweise für folgenden Code:

```
>>> if temperatur > 20:
...     wertung = "warm"
... else:
...     wertung = "frisch"
```

3.2.2 Schleifen

Schleifen werden benötigt, um einen Codeblock, also eine oder mehrere Python-Anweisungen, wiederholt auszuführen. Einen solchen Codeblock bezeichnet man auch als Schleifenkörper oder Body. Python kennt zwei Schleifentypen: die while- und die for-Schleife.

3.2.2.1 while-Schleife

Die Syntax der while-Schleife sieht wie folgt aus:

```
while <Ausdruck>:
    <Folge1>
else:
    <Folge2>
```

<Folge1> steht für eine beliebige Anweisungsfolge in gleicher Einrückungstiefe. Diese Anweisungsfolge wird solange ausgeführt, wie der Ausdruck „<Ausdruck>“ den Wert `True` liefert. Zuerst wird „<Ausdruck>“ ausgewertet, und im `True`-Fall wird „<Folge1>“ ausgeführt. Falls „<Ausdruck>“ den Wert `False` liefert, wird „<Folge2>“ ausgeführt, falls vorhanden. Der `else`-Teil ist optional. Eine while-Schleife kann auch durch ein `break`-Statement innerhalb von „<Folge1>“ abgebrochen werden. In diesem Fall wird jedoch „<Folge2>“ übersprungen, falls der `else`-Teil überhaupt vorhanden ist.

Das folgende Skript benutzt eine while-Schleife, um die Zahlen von 1 bis 4, gefolgt von ihrem jeweiligen Quadrat, auszugeben:

```
>>> i = 1
>>> while i <= 4:
...     print(i, i**2)
...     i += 1
...
1 1
2 4
3 9
4 16
```

Das nächste Beispiel benutzt ein `break`:

```
>>> numbers = [4, 5, 12, 9, -1, 8, 9]
>>> sum_numbers = 0
>>> while numbers != []:
...     number = numbers.pop()
...     if number >= 0:
...         sum_numbers += number
...     else:
...         break
...
>>> print(sum_numbers, numbers)
17 [4, 5, 12, 9]
```

3.2.2.2 for-Schleife

Die `for`-Schleife dient in Python dazu, über beliebige iterierbare Objekte zu iterieren. Iterierbare Objekte sind beispielsweise Listen, Tupel, Strings und Dictionaries.

```
einkaufsliste = ["Butter", "Milch", "Brot", "Salat", "Spam"]

for artikel in einkaufsliste:
    if artikel == "Spam":
        print("Spam mag ich nicht!")
    else:
        print("Ich werde " + artikel + " kaufen!")
```

Die Ausgabe lautet dann:

```
Ich werde Butter kaufen!
Ich werde Milch kaufen!
Ich werde Brot kaufen!
Ich werde Salat kaufen!
Spam mag ich nicht!
```

Weitere selbsterklärende Beispiele von `for`-Schleifen:

```
>>> wort = "Python"
>>> for buchstabe in wort:
...     print(buchstabe)
...
p
y
t
h
o
n
>>>
>>> liste = [(4, 5, 9.1), "Python", [4, ["abc", "xyz"]]]
>>> for element in liste:
...     print(element)
...
(4, 5, 9.1)
Python
[4, ['abc', 'xyz']]
>>>
```

Ein wichtiges iterierbares Objekt für die for-Anweisung stellt die range-Klasse¹ zur Verfügung. Mit den von range erzeugten Objekten lassen sich for-Schleifen im Stil von C und Java simulieren. Das Verhalten von range erklären wir durch die folgenden Beispiele:

```
>>> range(4)
range(0, 4)
>>> for i in range(4):
...     print(i, end=" ")
...
0, 1, 2, 3, >>>
>>>
>>> lst = [34, 55, 2, 10]
>>> for i in range(len(lst)):
...     print(i, lst[i])
...
0 34
1 55
2 2
3 10
>>>
>>> von, bis, schrittweite = 3, 21, 4
>>> lst = list(range(von, bis, schrittweite))
>>> print(lst)
[3, 7, 11, 15, 19]
```

Im folgenden Beispiel finden wir noch die break- und die continue-Anweisung im Einsatz. Mit „break“ erzeugen wir einen vorzeitigen Abbruch der Schleife, während mit „continue“ nur der aktuelle Durchlauf beendet wird, um dann mit dem nächsten Objekt fortzufahren. In diesem Beispiel finden wir auch noch eine Besonderheit der for-Schleife in Python: Die for-Schleife kann auch ein else-Statement enthalten. Nur wenn die Schleife nicht mit einem „break“ verlassen worden ist, werden die Anweisungen unter dem else-Teil bearbeitet.

```
string = input("Bitte einen String eingeben: ")

for buchstabe in string:
    zaehler = 0
    if buchstabe.isalpha():
        print(buchstabe)
    else:
        if buchstabe == ".":
            print("Punkt wurde gelesen!")
            print("Die restlichen Zeichen werden nicht mehr bearbeitet!")
            break
        else:
            print("Sonderzeichen wird ignoriert!")
            continue
    zaehler += 1
else:
    print("Der String enthielt keinen Punkt")
```

Starten wir das Programm, erhalten wir folgende Ausgabe:

```
bernd@moon:~$ python3 for_with_break_continue_else.py
Bitte einen String eingeben: Hi, you!
```

¹ range ist keine Funktion, sondern eine Klasse.

```

H
i
Sonderzeichen wird ignoriert!
Sonderzeichen wird ignoriert!
y
o
u
Sonderzeichen wird ignoriert!
Der String enthielt keinen Punkt
bernd@moon:~$ python3 for_with_break_continue_else.py
Bitte einen String eingeben: Hi, you.
H
i
Sonderzeichen wird ignoriert!
Sonderzeichen wird ignoriert!
y
o
u
Punkt wurde gelesen!
Die restlichen Zeichen werden nicht mehr bearbeitet!

```

3.2.3 Funktionen

3.2.3.1 Einfache Funktionen

Funktionen werden mit dem Schlüsselwort „def“ eingeleitet. Danach folgt der Name der Funktion, eine Folge von formalen Parametern in runden Klammern und am Ende der Zeile ein Doppelpunkt. Der Funktionskörper besteht aus beliebigen eingerückten Anweisungen. Unmittelbar nach der eben beschriebenen Kopfzeile kann noch optional ein String-Literal stehen. Dies ist ein Dokumentationsstring, docstring, der in der help-Funktionalität verwendet wird.

```

>>> def f():
...     """
...     Diese Funktion hat keine Parameter und liefert None zurück,
...     wenn sie aufgerufen wird!
...     """
...     pass
...
>>> print(f())
None

```

„pass“ ist eine leere Anweisung, die als Platzhalter fungiert. Diese Anweisung wird verwendet, wenn in einem Kontext eine Anweisung erforderlich ist, aber keine Aktion ausgeführt werden soll. In Funktionen muss mindestens eine Codezeile nach der Kopfzeile stehen, weswegen wir ein pass verwendet haben.

Ein wesentliches Merkmal von Funktionen ist die Rückgabe von Werten. Unsere Beispiel-funktion hat automatisch das None-Objekt zurückgegeben. Man kann auch explizit mit der return-Anweisung Rückgaben definieren. Trifft der Programmablauf innerhalb einer Funktion auf eine return-Anweisung, wird die Funktion verlassen und das Objekt zurückgeliefert, das in dem Ausdruck der return-Anweisung erzeugt wird.

```
>>> def poly(x):
...     return 3 * x**2 + 0.9 * x - 9
...
>>> poly(1)
-5.1
>>> poly(2)
4.800000000000001
```

Eine Funktion kann zwar nur ein Objekt zurückliefern, aber man kann natürlich auch mehrere Objekte, die man zurückgeben möchte, in einem Tupel vereinen. Die folgende Funktion liefert den Umfang und die Fläche eines Rechtecks zurück, dessen Länge und Breite als Parameter an die Funktion übergeben werden:

```
>>> def umfang_flaeche(laenge, breite):
...     return (2*(laenge+breite), laenge*breite)
...
>>> umfang_flaeche(3, 4)
(14, 12)
>>> umfang, flaeche = umfang_flaeche(3, 4)
>>> umfang, flaeche
(14, 12)
```

Funktionsnamen sind – wie andere Variablen auch – Referenzen auf Objekte, in diesem Fall Funktionsobjekte. Wir können also unserer Funktion `umfang_flaeche` einen kürzeren zusätzlichen Namen geben:

```
>>> uf = umfang_flaeche
>>> uf(1, 2)
(6, 2)
```

3.2.3.2 Default-Parameter und Schlüsselwortparameter

Man hat auch die Möglichkeit, die Parameter einer Funktion mit Standardwerten – meist auch als Default-Werte bezeichnet – zu versehen. Diese Parameter sind dann optional beim Aufruf der Funktion, d.h. man kann, aber muss für einen solchen Parameter kein Argument zur Verfügung stellen. Wird kein Wert angegeben, wird der Standardwert für diesen Parameter eingesetzt:

```
>>> def umfang(laenge=2, breite=1):
...     return 2 * (laenge + breite)
...
>>>
>>> umfang(5, 3)
16
>>> umfang(5)          # für breite wird der Standardwert „1“ benutzt
12
>>> umfang()           # es werden nun beide Standardwerte benutzt
6
```

Wie sieht es aber aus, wenn wir nur einen Wert für die Breite `breite`, aber nicht für die Länge `laenge` übergeben wollen? Übergibt man nur ein Argument, bedeutet das automatisch, dass dies einen Wert für den ersten Parameter darstellt, also in unserem Fall für `laenge`. Die Schlüsselwortparameter stellen eine Lösung für dieses Problem dar:

```
>>> umfang(breite=1.5)
7.0
```

3.2.3.3 Lokale Funktionen

Man kann innerhalb einer Funktion eine oder mehrere andere Funktionen definieren, die dann lokal in dieser Funktion sind. Sie können also nur innerhalb der Funktion benutzt werden:

```
def f(x):
    def g(x):
        return 2.3 * x

    return g(x) - 2

print(f(1))
```

Das obige Programm gibt 0.2999999999999998 als Ergebnis zurück. Die obige Funktion ist allerdings nicht sehr sinnvoll. Wir hatten festgestellt, dass Funktionen beliebige Objekte zurückliefern können. Dies bedeutet, dass Funktionen auch Referenzen auf Funktionen zurückliefern können. Im Folgenden definieren wir eine Funktion, die eine Referenz auf ein Polynom zweiten Grades zurückliefert:

```
def polynomial_creator(a, b, c):
    def polynomial(x):
        return a * x**2 + b * x + c
    return polynomial

p1 = polynomial_creator(2, 3, -1)
p2 = polynomial_creator(-1, 2, 1)

for x in range(-2, 2, 1):
    print(x, p1(x), p2(x))
```

Im obigen Programm erzeugen wir mithilfe des `polynomial_creator` zwei Polynome `p1` und `p2`. Das Programm liefert die folgende Ausgabe zurück:

```
-2 1 -7
-1 -2 -2
0 -1 1
1 4 2
```

3.2.3.4 Globale und lokale Variablen in Funktionen

Globale Variablen können, auch wenn man es normalerweise nicht tun sollte, innerhalb des Funktionsrumpfs einer Funktion „lesend“ benutzt werden. Man spricht dann von freien Variablen:

```
def f():
    print(s)
s = "Ich bin ein String!"
f()
```

Der Funktionsrumpf von `f()` besteht nur aus der `print(s)`-Anweisung. Weil es keine lokale Variable `s` gibt, d.h. keine Zuweisung an `s` innerhalb des Funktionsrumpfs von `f`, wird der Wert der globalen Variablen `s` benutzt. Dieser Wert kann allerdings nicht verändert werden, wie wir weiter unten in diesem Kapitel sehen werden. Es wird also der String „Ich bin ein String!“ ausgegeben.

Wird innerhalb einer Funktion eine Variable definiert, so ist diese lokal, auch wenn es im aufrufenden Kontext eine Variable mit gleichem Namen gibt:

```
def f():
    s = "Ich bin ein lokaler String!"
    print(s)

s = "Ich bin ein String!"
f()
print(s)
```

Starten wir das Skript, erhalten wir folgende Ausgabe:

```
$ python3 global_lokal2.py
Ich bin ein lokaler String!
Ich bin ein String!
```

Man kann auch den Wert von globalen Variablen innerhalb einer Funktion verändern. Dazu muss man sie jedoch explizit mittels des Schlüsselworts `global` als global deklarieren:

```
def f():
    global s
    print(s)
    s = "Ich bin das globale s!"
    print(s)

s = "f wird mich ändern!"
f()
print(s)
```

Als Ausgabe erhalten wir:

```
$ python3 global_lokal4.py
f wird mich ändern!
Ich bin das globale s!
Ich bin das globale s!
```

■ 3.3 Ausnahmebehandlung

Unter einer Ausnahmebehandlung² versteht man ein Verfahren, die Zustände, die während einer Fehlersituation herrschen, an andere Programmebenen weiterzuleiten. Dadurch ist es möglich, per Programm einen Fehlerzustand gegebenenfalls zu „reparieren“, um anschließend das Programm weiter auszuführen. Ansonsten würden solche Fehlerzustände in der Regel zu einem Abbruch des Programms führen.

Die Ausnahmebehandlung in Python ist sehr ähnlich zu derjenigen in Java. Der Code, der das Risiko für eine Ausnahme beherbergt, wird in einen `try`-Block eingebettet. Aber während in Java Ausnahmen durch `catch`-Konstrukte abgefangen werden, geschieht dies in Python durch das `except`-Schlüsselwort. Semantisch funktioniert es aber genauso.

² engl. „exception handling“

Ausnahmen entstehen beispielsweise, wenn man versucht, durch 0 zu dividieren, eine nicht definierte Variable anzusprechen oder eine nicht unterstützte Operation auszuführen versucht, wie wir im Folgenden demonstrieren:

```
>>> x = 10 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> z = x + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 10 + "42"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Diese Fehler kann man in einem Programm wie folgt abfangen:

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Es ist nicht möglich durch 0 zu dividieren!")

try:
    z = x + 1
except NameError:
    print("Die Variable 'x' ist nicht definiert!")

try:
    x = 10 + "42"
except TypeError:
    print("Integer und Strings können nicht addiert werden!")
```

Startet man obiges Programm, erhält man folgende Ausgaben:

```
Es ist nicht möglich durch 0 zu dividieren!
Die Variable 'x' ist nicht definiert!
Integer und Strings können nicht addiert werden!
```

Eine sinnvolle Anwendung stellt das folgende Programmstück dar, in dem wir eine robuste Eingabeaufforderung programmieren. Wir stellen sicher, dass das Programm erst weitermachen kann, wenn eine gültige Integer-Zahl eingegeben wurde:

```
while True:
    try:
        zahl = input("Zahl eingeben: ")
        zahl = int(zahl)
        break
    except ValueError as e:
        print("error message: ", e)
        print("Keine Integer-Zahl!")
```

Statt für jede Ausnahmesituation einen eigenen Exception-Zweig zu programmieren, kann man auch mit einer Exception mehrere Ausnahmen abfangen:

```
while True:
    prompt = """Choose your error:
    1 ZeroDivisionError
```



```

        2 TypeError
        3 NameError
        4 exit
    """
    try:
        error_type = input(prompt)
        if error_type == "1":
            x = 10 / 0
        elif error_type == "2":
            x = 10 + "34"
        elif error_type == "3":
            print(x)
        elif error_type == "4":
            exit()
    except Exception as e:
        print("String-Darstellung der Ausnahme: ", str(e))
        print(type(e))
        print(e.args[0])

```

In folgender beispielhafter Nutzung zeigen wir die Ergebnisse für die verschiedenen Fälle:

```
bernd@marvin ~/tmp $ python3 choose_your_error.py
```

```
Choose your error:
```

```

    1 ZeroDivisionError
    2 TypeError
    3 NameError
    4 exit

```

```
1
```

```
String-Darstellung der Ausnahme: division by zero
```

```
<class 'ZeroDivisionError'>
```

```
ZeroDivisionError
```

```
division by zero
```

```
Choose your error:
```

```

    1 ZeroDivisionError
    2 TypeError
    3 NameError
    4 exit

```

```
2
```

```
String-Darstellung der Ausnahme: unsupported operand type(s) for +: 'int' and 'str'
```

```
<class 'TypeError'>
```

```
TypeError
```

```
unsupported operand type(s) for +: 'int' and 'str'
```

```
Choose your error:
```

```

    1 ZeroDivisionError
    2 TypeError
    3 NameError
    4 exit

```

```
3
```

```
String-Darstellung der Ausnahme: name 'x' is not defined
```

```
<class 'NameError'>
```

```
NameError
```

```
name 'x' is not defined
```

```
Choose your error:
```

```

    1 ZeroDivisionError
    2 TypeError
    3 NameError
    4 exit

```

```
4
```

```
bernd@marvin ~/tmp $
```

3.3.1 Die optionale else-Klausel

Auch das try ... except-Sprachkonstrukt verfügt über eine optionale else-Klausel, die – falls vorhanden – hinter allen except-Klauseln stehen muss. Dort befindet sich Code, der ausgeführt wird, wenn die try-Klausel keine Ausnahme auslöst.

Im Folgenden verlangen wir solange die Eingabe eines Dateinamens, bis sich dieser zum Lesen öffnen lässt. Der else-Teil der try-Anweisung wird nur ausgeführt, wenn es keinen Ausnahmefehler gegeben hat. Deshalb dürfen wir dann auch auf das Datei-Handle f zugreifen:

```
while True:
    filename = input("Dateiname: ")
    try:
        f = open(filename, 'r')
    except IOError:
        print(filename, " lässt sich nicht öffnen")
    else:
        print(filename, ' hat ', len(f.readlines()), ' Zeilen ')
        f.close()
        break
```

3.3.2 Exceptions generieren

Man kann auch eigene Exceptions generieren:

```
>>> raise SyntaxError("Sorry, mein Fehler!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: Sorry, mein Fehler!
```

Außerdem lassen sich auch eigene Exception-Klassen definieren. Da wir Klassen noch nicht definiert haben, könnte das folgende Beispiel für einige noch nicht verständlich sein:

```
class MyException(Exception):
    pass

raise MyException("Was falsch ist, ist falsch!")
```

Startet man dieses Programm, erhält man folgende Ausgabe:

```
$ python3 exception_eigene_klasse.py
Traceback (most recent call last):
  File "exception_eigene_klasse.py", line 4, in <module>
    raise MyException("Was falsch ist, ist falsch!")
__main__.MyException: Was falsch ist, ist falsch!
```

3.3.3 Finalisierungsaktion

Neben except und else hat ein try-Konstrukt noch den finally-Zweig zu bieten. Man bezeichnet diese Form auch als Finalisierungs- oder Terminierungsaktionen, weil sie immer unter allen Umständen ausgeführt werden müssen, und zwar unabhängig davon, ob eine Ausnahme im try-Block aufgetreten ist oder nicht.

```

try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("Ich werde immer ausgegeben, ob Fehler oder nicht")

print("Mich sieht man nur, wenn es keinen Fehler gab!")

```

In den folgenden Programmläufen demonstrieren wir einen Fehlerfall und einen Durchlauf ohne Ausnahmen:

```

bernd@saturn:~/bodenseo/python/beispiele$ python3 finally_exception.py
Your number: 42
Ich werde immer ausgegeben, ob Fehler oder nicht
Mich sieht man nur, wenn es keinen Fehler gab!
bernd@saturn:~/bodenseo/python/beispiele$ python3 finally_exception.py
Your number: 0
Ich werde immer ausgegeben, ob Fehler oder nicht
Traceback (most recent call last):
  File "finally_exception.py", line 3, in <module>
    inverse = 1.0 / x
ZeroDivisionError: float division by zero
bernd@saturn:~/bodenseo/python/beispiele$

```

■ 3.4 Dateien lesen und schreiben

3.4.1 Datei lesen

Unser erstes Beispiel zeigt, wie man Daten aus einer Datei ausliest. Um dies tun zu können, muss man zuerst die Datei zum Lesen öffnen. Dazu benötigt man die `open()`-Funktion. Mit der `open`-Funktion erzeugt man ein Dateiobjekt, genau genommen ein `TextIOWrapper`-Objekt, und liefert eine Referenz auf dieses Objekt als Ergebniswert zurück.

Um die Datei `bundeslaender.txt` zum Lesen zu öffnen, genügt die folgende Anweisung:

```
fobj = open("bundeslaender.txt")
```

Alternativ kann man auch den Parameter-Mode auf „r“ setzen:

```

fobj = open("bundeslaender.txt", "r")
# oder über Schlüsselwort:
fobj = open("bundeslaender.txt", mode="r")

```

Nach dem obigen Befehl kann man über den `TextIOWrapper` `fobj` verschiedene Methoden aufrufen:

- `readline`: eine Zeile lesen

```

>>> fobj = open("bundeslaender.txt")
>>> fobj.readline() # lesen einer Zeile
'Land Flaeche maennlich weiblich\n'
>>> fobj.readline() # noch eine Zeile
'Baden-Württemberg 35751.65 5271 5465\n'

```

- `read`: ganzen Text in einen String lesen

```
>>> fobj = open("bundeslaender.txt")
>>> txt = fobj.read()
>>> txt[:60]
'Land Flaechе maennlich weiblich\nBaden-Württemberg 35751.65 5'
```

- `readlines`: Text in eine Liste mit den Zeilen einlesen

```
>>> fobj = open("bundeslaender.txt")
>>> zeilen = fobj.readlines()
>>> zeilen[:4] # die ersten vier Zeilen
['Land Flaechе maennlich weiblich\n', 'Baden-Württemberg 35751.65 5271 5465\n',
 'Bayern 70551.57 6103 6366\n', 'Berlin 891.85 1660 1736\n']
```

Häufig werden Dateien jedoch mittels einer `for`-Schleife Zeile für Zeile durchlaufen. Dies geschieht dann innerhalb eines `with`-Konstrukts, welches nach Beendigung des Blocks automatisch die Datei schließt:

```
with open("bundeslaender.txt") as fh:
    for zeile in fh:
        print(zeile)
```

3.4.2 Datei schreiben

Will man in eine Datei schreiben, benutzt man „w“ statt „r“ beim `open`. Mit der Methode `write` kann man einen String in die geöffnete Datei schreiben:

```
>>> fh = open("beispiel.txt", "w")
>>> fh.write("1. Zeile\n")
9
>>> fh.write("2. Zeile\n")
9
>>> fh.close()
>>> txt = open("beispiel.txt").read()
>>> print(txt)
1. Zeile
2. Zeile

>>>
```

Im folgenden Programm lesen wir die Datei `bundeslaender.txt` ein und schreiben die Daten in einem gewandelten Format in eine Datei `bundeslaender2.txt`.

```
with open("bundeslaender.txt") as fh_in, \
    open("bundeslaender2.txt", "w") as fh_out:
    fh_in.readline() # lesen der headerzeile
    fh_out.write("Land Flaechе Einwohner Dichte\n")
    for zeile in fh_in:
        land, flaeche, maennlich, weiblich = zeile.split()
        flaeche = float(flaeche)
        einwohner = int(maennlich) + int(weiblich)
        dichte = round(einwohner * 1000 / flaeche, 2)
        fh_out.write(land + " " + str(flaeche) + " " + str(einwohner) + " " + str(
            dichte) + "\n")
```

Wir zeigen im Folgenden die ersten sieben Zeilen der durch das obige Programm erzeugten Datei `bundeslaender2.txt`:

```
Land Flaeche Einwohner Dichte
Baden-Württemberg 35751.65 10736 300.29
Bayern 70551.57 12469 176.74
Berlin 891.85 3396 3807.82
Brandenburg 29478.61 2560 86.84
Bremen 404.28 663 1639.95
Hamburg 755.16 1743 2308.12
```

■ 3.5 Modularisierung

Module werden mit der `import`-Anweisung in ein Programm eingebunden:

```
import math
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
```

Es können auch mehrere Module mit einer `import`-Anweisung importiert werden:

```
import math, random
```

`import`-Anweisungen können an jeder Stelle des Quellcodes stehen, aber man sollte sie der Übersichtlichkeit willen an den Anfang stellen.

3.5.1 Namensräume von Modulen

Wir haben gesehen, dass die Klassen und Funktionen eines Modules nach einem Import in einem eigenen Namensraum zur Verfügung stehen, d.h. man kann beispielsweise auf die `sin()`-Funktion nur über den vollen Namen („fully qualified“) zugreifen, d.h.

```
>>> math.sin(3.1415)
9.265358966049024e-05
```

Möchte man „bequem“ auf Funktionen wie z.B. die Sinus-Funktion zugreifen können, so kann man die entsprechenden Funktionen direkt importieren:

```
>>> from math import sin, pi
>>> print(pi)
3.141592653589793
>>> sin(pi/2)
1.0
```

Die anderen Methoden der Bibliothek stehen dann nicht zur Verfügung. Auch nicht mit ihrem vollen Namen.

Man kann auch eine Bibliothek komplett in den globalen Namensraum einbinden. Dabei werden dann gegebenenfalls bereits vorhandene gleichlautende Namen überschrieben, wie dies im folgenden Beispiel dargestellt wird:

```
>>> pi = 57.898 # eigene Variable, die überschrieben wird
>>> print(pi)
57.898
>>> from math import *
>>> print(pi)
3.141592653589793
```

Außerdem ist es möglich, beim Import einer Bibliothek einen neuen Namen für den Namensraum zu wählen. Im Folgenden importieren wir math als m:

```
>>> import math as m
>>> m.pi
3.141592653589793
>>> m.sin(m.pi)
```

3.5.2 Suchpfad für Module

Wenn man ein Modul importiert, z.B. xyz, sucht der Interpreter nach xyz.py in der folgenden Reihenfolge:

- im aktuellen Verzeichnis
- in der Umgebungsvariablen PYTHONPATH, die auf Betriebssystemebene gesetzt werden muss
- Falls PYTHONPATH nicht gesetzt ist, wird installationsabhängig im Default-Pfad gesucht, also unter Linux/Unix z.B. in /usr/lib/python3.7.

3.5.3 Inhalt eines Moduls

Mit der built-in-Funktion dir() kann man sich die in einem Modul definierten Namen ausgeben lassen:

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2',
 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
 'tanh', 'trunc']
```

3.5.4 Eigene Module

Jede Datei, die gültigen Python-Code enthält und die Dateiendung .py hat, ist ein Modul. Die beiden folgenden Funktionen fib(), die den n-ten Fibonacci-Wert zurückliefert, und die Funktion fiblist() werden in einer Datei fibonacci.py gespeichert:

```
def fib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

```
def fiblist(n):
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib
```

Von einem anderen Programm oder von der interaktiven Shell kann man nun, falls fibonacci.py innerhalb des Suchpfads zu finden ist, die Datei mit den beiden Fibonacci-Funktionen als Modul aufrufen:

```
>>> import fibonacci
>>> fibonacci.fib(10)
55
>>> fibonacci.fiblist(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fibonacci.__name__
'fibonacci'
>>>
```

3.5.5 Dokumentation für eigene Module

Rufen wir help für unser fibonacci-Modul auf, erhalten wir folgende Ausgabe:

```
Help on module fibonacci:

NAME
    fibonacci

FUNCTIONS
    fib(n)

    fiblist(n)

FILE
    /home/data/bodenseo/python/fibonacci.py
```

Die help-Informationen können wir sehr einfach erweitern. Eine allgemeine Beschreibung des Moduls kann man in einem Docstring zu Beginn einer Moduldatei verfassen. Die Funktionen dokumentiert man wie üblich mit einem Docstring unterhalb der ersten Funktionszeile:

```
""" Modul mit wichtigen Funktionen zur Fibonacci-Folge """

def fib(n):
    """ Iterative Fibonacci-Funktion """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    """ produziert Liste der Fibo-Zahlen """
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib
```

Die help-Ausgabe sieht nun zufriedenstellend aus:

```
Help on module fibonacci:

NAME
    fibonacci - Modul mit wichtigen Funktionen zur Fibonacci-Folge

FUNCTIONS
    fib(n)
        Iterative Fibonacci-Funktion

    fiblist(n)
        produziert Liste der Fibo-Zahlen

FILE
    /home/data/bodenseo/python/fibonacci.py
```

■ 3.6 Klassen-Definition

3.6.1 Eine einfache Klasse

Die einfachst mögliche Definition einer Klasse sieht wie folgt aus:

```
class A:
    pass
```

Auch wenn in dieser Klasse keine Methoden definiert worden sind, kann man Instanzen erzeugen, was man auch als Instanziierung bezeichnet. Im folgenden Beispiel erzeugen wir die Instanzen `a1` und `a2`:

```
>>> class A:
...     pass
...
>>> a1 = A()
>>> a2 = A()
>>> print(a1)
<__main__.A object at 0x7fbbb508def0>
```

3.6.2 Attribute

Die meisten Python-Objekte können attribuiert werden:³

```
>>> import math
>>> math.f = 2.3
>>>
>>> def f():
...     pass
...
>>> f.counter = 0
```

³ Ausnahmen sind beispielsweise Instanzen der `int`-, `float`- und `str`-Klassen.

Auch eigene Klassen und deren Instanzen lassen sich attribuieren:

```
>>> class Robot:
...     pass
...
>>>
>>> r1 = Robot()
>>> r2 = Robot()
>>>
>>> Robot.name = "Marvin"
>>> r1.name = "Henry"
>>> print(r1.name, r2.name, Robot.name)
Henry Marvin Marvin
```

Bei `Robot.name` im vorigen Beispiel handelt es sich um ein sogenanntes Klassenattribut, während es sich bei `r1.name` um ein Instanzattribut handelt. Klassenattribute werden auch von allen Instanzen geteilt, falls sie kein Instanzattribut mit dem gleichen Namen haben wie im Falle von `r2.name`. Instanzattribute sind individuell für jede Instanz.

Definiert man Variablen oder Funktionen innerhalb einer `class`-Definition, also innerhalb des eingerückten Blocks, so werden diese zu Klassenattributen:

```
class Robot:

    name = "Marvin"

    def say_hi(self):
        return "Hi, I am " + self.name
```

Attribute sind Referenzen auf beliebige Objekte, im obigen Fall also auf ein String-Objekt und ein Funktionsobjekt.

Im folgenden Beispiel nehmen wir an, dass obige Klassendefinition als `simple_class.py` abgespeichert worden ist:

```
>>> from simple_class import Robot
>>> r = Robot()
>>> r.say_hi()
'Hi, I am Marvin'
>>> r.name
'Marvin'
>>> Robot.name
'Marvin'
>>> Robot.say_hi
<function Robot.say_hi at 0x7fdc61aeaea0>
```

`say_hi` im vorigen Beispiel ist eine gewöhnliche Funktion, die allerdings an die Klasse `Robot` gebunden ist. Der „volle“ Name der Funktion lautet damit `Robot.say_hi`. Der Aufruf `r.say_hi()` ist gewissermaßen eine abgekürzte Schreibweise für den Aufruf `Robot.say_hi(r)`. Man sieht dann auch, dass `r` zum Argument für den Parameter `self` wird:

```
>>> from simple_class import Robot
>>> r = Robot()
>>> Robot.say_hi(r)
'Hi, I am Marvin'
```

Außerdem wird damit auch klar, dass man statt des Namens `self` auch einen beliebigen anderen Namen hätte nehmen können. Im OOP-Jargon bezeichnet man solche Funktionen üblicherweise als Methoden.

3.6.3 Initialisierung von Instanzen

Bei der Instanziierung, d.h. beispielsweise beim Aufruf `r = Robot()`, wurde bisher ein leeres Objekt erzeugt. Bei den meisten Klassen möchte man jedoch nach der Erzeugung einer Instanz diese initialisieren, d.h. bestimmte Attribute auf einen bestimmten Initialwert setzen. Zu diesem Zweck definiert man in einer Klasse die Methode `__init__`. Diese Methode wird automatisch nach der Instanziierung einer Instanz aufgerufen. In unserem Fall könnten wir beispielsweise einen Roboter auf einen individuellen Namen setzen:

```
class Robot:

    def __init__(self, name="Marvin"):
        self.name = name

    def say_hi(self):
        return "Hi, I am " + self.name

r = Robot("Henry")
print(r.say_hi())
```

Die Ausgabe lautet in obigem Beispiel nun `Hi, I am Henry`.

3.6.4 Vererbung

Klassen können auch von anderen Klassen erben. So erbt im Folgenden die Klasse `MedicalRobot` von `Robot`:

```
class Robot:

    def __init__(self, name="Marvin"):
        self.name = name

    def say_hi(self):
        return "Hi, I am " + self.name

class MedicalRobot(Robot):

    def heal(self, x):
        return x.name + " is healed now, or maybe not!"

mr = MedicalRobot()
print(mr.say_hi())

y = "A String"
print(mr.heal(y))
```

Die Methoden der Oberklasse, auch Basisklasse genannt, also `Robot` in unserem Fall, werden von der abgeleiteten Klasse geerbt. In unserem Fall übernimmt die abgeleitete Klasse, auch Unterklasse oder Kindklasse genannt, `MedicalRobot` die Methoden `__init__` und `say_hi` von `Robot`. Außerdem wird die Kindklasse um eine Methode `heal` erweitert.

Es ist auch möglich, eine bereits definierte Methode zu überlagern. Im folgenden Beispiel überlagern wir die Methode `say_hi`:

```
class Robot:

    def __init__(self, name="Marvin"):
        self.name = name

    def say_hi(self):
        return "Hi, I am " + self.name

class MedicalRobot(Robot):

    def heal(self, x):
        return "x is healed now, or maybe not!"

    def say_hi(self):
        return "Hi, I am " + self.name + ", your personal nurse!"

mr = MedicalRobot()
print(mr.say_hi())

r = Robot("Henry")
print(r.say_hi())
```

Wir erhalten die folgende Ausgabe:

```
Hi, I am Marvin, your personal nurse!
Hi, I am Henry
```

3.6.5 Private, geschützte und öffentliche Attribute

In der Objektorientierung werden prinzipiell drei Attributarten unterschieden:

- **Public** oder öffentliche Attribute sind Attribute, auf die man von überall, d.h. innerhalb der eigenen Klasse, von anderen Klassen oder von Anwendungen, welche die Klasse nutzen, lesend und schreibend zugreifen kann und darf.
- **Protected** oder geschützte Attribute sind Attribute, auf die man nur aus der Klasse selbst oder von abgeleiteten Klassen zugreifen darf. In Python werden solche Attribute mit einem führenden Unterstrich versehen. Allerdings sind diese Attribute nicht „geschützt“, wie es der Name vermuten lässt. Der Unterstrich ist eine Konvention, um kenntlich zu machen, dass diese Klassen als `protected` anzusehen sind und nicht außerhalb der Vererbungshierarchie benutzt werden dürfen.
- **Private** Attribute dürfen und können nur innerhalb der Klasse, in der sie definiert worden sind, benutzt werden. Private Attribute führen zwei führende Unterstriche im Namen.

Wir demonstrieren dies im folgenden Beispiel:

```
class A:

    def __init__(self):
        self.pub = "Ich bin ein public-Attribut"
        self._prot = "Ich bin ein geschütztes Attribut"
        self.__priv = "Ich bin privat!"

a = A()
print(a.pub)
print(a._prot)
print(a.__priv)
```

In der Ausgabe des obigen Programms können wir sehen, dass wir auf das öffentliche und das geschützte Attribut von außen zugreifen können, aber nicht auf das private Attribut:

```
Ich bin ein public-Attribut
Ich bin ein geschütztes Attribut
Traceback (most recent call last):
  File "attribut_arten.py", line 12, in <module>
    print(a.__priv)
AttributeError: 'A' object has no attribute '__priv'
```

3.6.6 Properties

Sollte es notwendig sein, public-Attribute zu kapseln, so geschieht dies in Python mithilfe von Properties. Üblicherweise, d.h. in vielen anderen OO-Programmiersprachen, werden private Attribute (Daten) einer Klasse mithilfe von speziellen Zugriffsfunktionen, häufig als Getter und Setter bezeichnet, gekapselt. Im folgenden Beispiel zeigen wir dies mit der Klasse Robot. Wir definieren ein privates Klassenattribut für nicht-erlaubte Namen `__forbidden_names`. Der Name eines Roboters wird in einem privaten Attribut `__name` versteckt, d.h. man kann von außen nicht mehr direkt darauf zugreifen, sondern nur noch über die Methoden `get_name` und `set_name`. Die Methode `set_name` verhindert nun, dass ein Roboter einen unzulässigen Namen erhält:

```
class Robot:

    __forbidden_names = {"Henry", "Oscar"}

    def __init__(self, name="Marvin"):
        self.set_name(name)

    def get_name(self):
        return self.__name

    def set_name(self, name):
        if name in Robot.__forbidden_names:
            self.__name = "Marvin"
        else:
            self.__name = name
```

Dies ist aber nicht „pythonisch“. Nach den offiziellen Stilrichtlinien von Python⁴ implementiert man Dateninhalte, die man Benutzern der Klasse zugänglich machen will, in public-Attributen. Benötigt ein solches Attribut später eine spezielle Behandlung, wie wir es oben in der `set_name`-Methode hatten, dann können wir eine `property` einführen. Das Interface-Verhalten wird dadurch für die Benutzer der Klassen nicht verändert.

```
class Robot:

    __forbidden_names = {"Henry", "Oscar"}

    def __init__(self, name="Marvin"):
        self.name = name

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        if name in Robot.__forbidden_names:
            self.__name = "Marvin"
        else:
            self.__name = name

x = Robot("Isidor")
print(x.name)
x.name = "Henry"
print(x.name)
```

⁴ PEP 8 – Style Guide for Python Code, <https://www.python.org/dev/peps/pep-0008/>

Teil II

NumPy

4

NumPy Einführung

■ 4.1 Überblick

Wie wir bereits im Kapitel „[Definition von numerischer Programmierung](#)“ beschrieben haben, handelt es sich bei NumPy um ein Modul, welches grundlegende Datenstrukturen, d.h. mehrdimensionale Arrays und Matrizen, zur Verfügung stellt, die auch von Matplotlib, SciPy und Pandas benutzt werden. Der Name NumPy stellt ein Akronym für den englischen Begriff „Numerical Python“ dar. Bei dem NumPy-Modul wurde von Anfang an besonderer Wert auf speicherschonende und schnelle Implementierungen gelegt, weshalb auch der größte Teil des Moduls in C geschrieben worden ist.

Dadurch wird sichergestellt, dass die kompilierten mathematischen und numerischen Funktionen und Funktionalitäten eine größtmögliche Ausführungsgeschwindigkeit garantieren. Python wird damit um mächtige Datenstrukturen erweitert und bereichert, die das effiziente Rechnen mit großen Arrays und Matrizen ermöglichen. Die Implementierung zielt sogar auf extrem große („big data“) Matrizen und Arrays. Ferner bietet das Modul eine riesige Anzahl von hochwertigen mathematischen Funktionen, um mit diesen Matrizen und Arrays zu arbeiten.

SciPy (Scientific Python) wird oft im gleichen Atemzug wie NumPy genannt. SciPy erweitert die Leistungsfähigkeit von NumPy um weitere nützliche Funktionen, wie zum Beispiel Minimierung, Regression, Fouriertransformation und viele andere.

Sowohl NumPy als auch SciPy sind üblicherweise bei einer Standardinstallation von Python nicht installiert. NumPy sowie all die anderen erwähnten Module sind jedoch Bestandteil der Anaconda-Distribution.

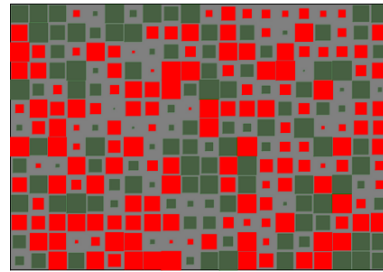


Bild 4.1 Visualisierung einer Matrix als Hinton-Diagramm

■ 4.2 Vergleich NumPy-Datenstrukturen und Python

Die Datenstrukturen des reinen Python, also ohne NumPy und andere, bieten große Vorteile:

Vorteile von Python-Datenstrukturen:

- Integers und Floats sind als mächtige Klassen implementiert. So können Integer-Zahlen beinahe „unendlich“ groß oder klein werden.¹
- 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

■ 4.3 Ein einfaches Beispiel

Wie bei allen anderen Modulen müssen wir auch NumPy importieren, bevor wir mit dem Modul arbeiten können.

```
import numpy
```

NumPy wird aber nur selten in dieser Form importiert. Meistens wird es beim Import 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))
```

Ausgabe:

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

Nun wollen wir die obigen Temperaturwerte in Grad Fahrenheit umrechnen.

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

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

¹ Sie sind letztendlich begrenzt durch die Größe des Speichers und immer noch unendlich weit von „unendlich“ entfernt!

Ausgabe:

```
[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)
```

Ausgabe:

```
[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 mithilfe einer Listenabstraktion in eine Liste mit Fahrenheit-Temperaturen wandelt, als umständlich dar!

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

Ausgabe:

```
[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)
```

Ausgabe:

```
numpy.ndarray
```

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

■ 4.4 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:

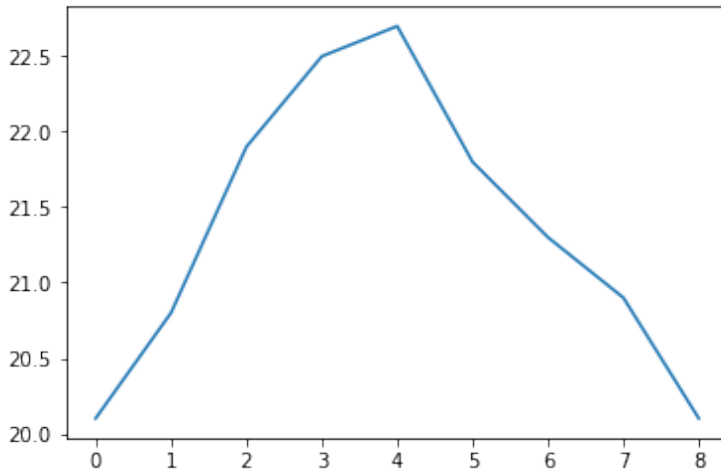
```
%matplotlib inline
```

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

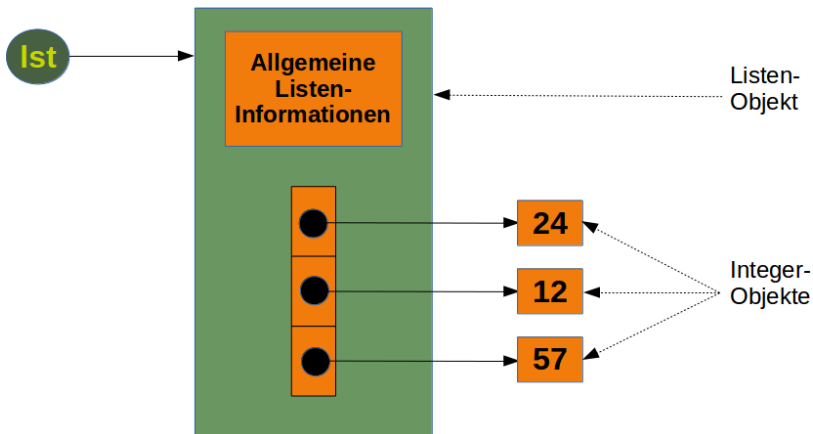
plt.plot(C)
plt.show()
```



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.

■ 4.5 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)

```

Ausgabe:

```

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))

```

Ausgabe:

```

Größe ohne Größe der Elemente: 96
Größe aller Elemente: 112
Gesamtgröße der Liste: 208
Speicherbedarf einer leeren Liste: 64

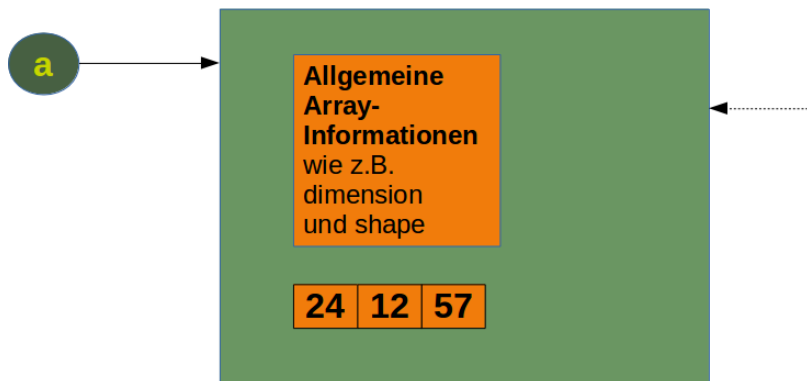
```

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))
```

Ausgabe:

120

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

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

Ausgabe:

96

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

$$96 + n * 8 \text{ Bytes}$$

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

$$64 + 8 \text{ len(lst)} + \text{len(lst)} * 28$$

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)
```

Ausgabe:

```
3
6
12
24
```

■ 4.6 Zeitvergleich zwischen Python-Listen und NumPy-Arrays

Einer der Hauptvorteile von NumPy ist sein Zeitvorteil gegenüber Standard-Python. 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!")
```

Ausgabe:

```
0.00022339820861816406 6.556510925292969e-05
NumPy is in this example 3.407272727272727 faster!
```

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

Dem Konstruktor eines Timer-Objekts 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))
```

Ausgabe:

```
0.0019844429998556734
0.00026837800032808445
```

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))
```

Ausgabe:

```
[0.0020292469998821616, 0.0019671789996209554,
 0.00211222899976651]
[0.00014969599942560308, 0.0001108839996959432,
 0.00042726500032586046]
```

5

Arrays in NumPy erzeugen

Nachdem wir im vorigen Kapitel gelernt haben, wie man NumPy-Arrays aus Listen und Tupeln erzeugt, werden wir nun 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 Abstandswert, 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.



Bild 5.1 Symbolisches Array

■ 5.1 Erzeugung äquidistanter Intervalle

5.1.1 `arange`

Die Syntax von `arange`:

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

`arange` liefert gleichmäßig verteilte Werte innerhalb eines gegebenen Intervalls zurück. Die Werte werden innerhalb des halb-offenen Intervalls `[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 Output-Arrays beeinflussen. Der Abstand zwischen zwei benachbarten Werten des Output-Arrays 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 Output-Arrays 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)
```

Ausgabe:

```
[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]
```

5.1.2 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 Intervall ['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 (Default) 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))
```

Ausgabe:

```
[ 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 Abstands 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)

```

Ausgabe:

```

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

```

5.1.3 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))

```

Ausgabe:

```

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

```

5.1.4 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))
```

Ausgabe:

```
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
```

5.1.5 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)
```

Ausgabe:

```
[[ 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.

■ 5.2 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 (Dimen-

sion). In unserem Beispiel ist die Shape gleich (6, 3). Das bedeutet, dass wir sechs Zeilen und drei Spalten haben.¹

```
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))
```

Ausgabe:

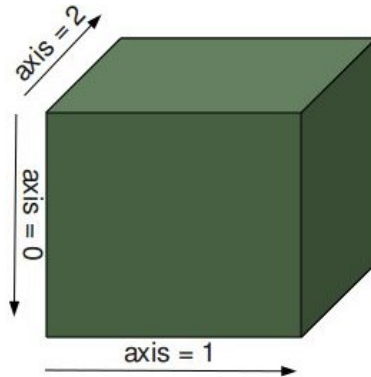
(6, 3)

Es gibt auch eine äquivalente Array-Property:

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

Ausgabe:

(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)
```

Ausgabe:

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

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

¹ In der Mathematik benutzt man neben ‚Gestalt‘ auch den Begriff ‚Typ‘ einer Matrix. Man spricht von einer $m \times n$ (sprich: m-mal-n- oder m-Kreuz-n-Matrix) und meint damit eine Matrix mit m Zeilen und n Spalten.

Ausgabe:

```
[[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))
```

Ausgabe:

```
()
```

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)
```

Ausgabe:

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

■ 5.3 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.

Das Indizieren einzelner Elemente 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])
```

Ausgabe:

```
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])
```

Ausgabe:

```
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])
```

Ausgabe:

```
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])
```

Ausgabe:

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

Wir nehmen an, dass Sie mit den Teilbereichsoperatoren (slicing) von den Listen und Tupeln bereits vertraut sind.

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 Standard-Python 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[:])
```

Ausgabe:

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

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

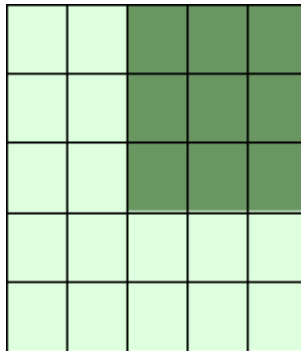
Die Anwendung des Teilbereichsoperators auf mehrdimensionale Arrays illustrieren wir in den folgenden Beispielen. Die Bereiche für jede Dimension werden durch Kommas getrennt:

```
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:])
```

Ausgabe:

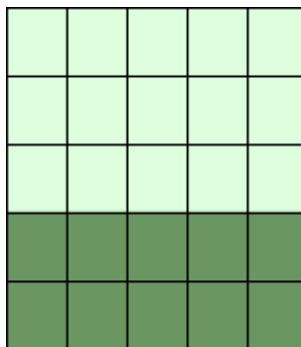
```
[[13 14 15]
 [23 24 25]
 [33 34 35]]
```



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

Ausgabe:

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



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

Ausgabe:

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


Die folgenden beiden Beispiele benutzen 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)
```

Ausgabe:

```
[[ 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])
```

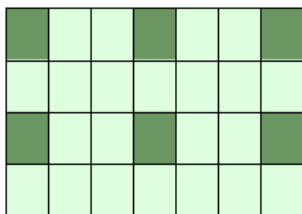
Ausgabe:

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


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

Ausgabe:

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

Falls die Zahl der Objekte in dem Auswahltuplel 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, :]
```

Ausgabe:

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

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

Achtung: Während der Teilbereichsoperator bei Listen und Tuplel 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)
```

Ausgabe:

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

Wenn wir das analog bei Listen tun, sehen wir, dass wir eine Kopie erhalten. Genau genommen müssten wir sagen, eine flache Kopie.

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

Ausgabe:

```
[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)
```

Ausgabe:

```
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))
```

Ausgabe:

```
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.

■ 5.4 Dreidimensionale Arrays

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

```
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)
```

Ausgabe:

```
(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(X[i,:,:])

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

print("\nDimension 2 with size ", X.shape[2])
for i in range(X.shape[2]):
    print(X[:, :,i])

```

Ausgabe:

```

Dimension 0 with size 4
[[3 1 2]
 [4 2 2]]
[[-1  0  1]
 [ 1 -1 -2]]
[[3 2 2]
 [4 4 3]]
[[2 2 1]
 [3 1 3]]

Dimension 1 with size 2
[[ 3  1  2]
 [-1  0  1]
 [ 3  2  2]
 [ 2  2  1]]
[[ 4  2  2]
 [ 1 -1 -2]
 [ 4  4  3]
 [ 3  1  3]]

Dimension 2 with size 3
[[ 1  2]
 [ 0 -1]
 [ 2  4]
 [ 2  1]]
[[ 1  2]
 [ 0 -1]
 [ 2  4]
 [ 2  1]]
[[ 1  2]
 [ 0 -1]
 [ 2  4]
 [ 2  1]]

```

Die folgenden Beispiele erläutern dies noch weiter:

```
[[[ 3  1  2]
   [ 4  2  2]]] x[0, :, :]
```

```
[[[-1  0  1]
   [ 1 -1 -2]]] x[1, :, :]
```

```
[[[ 3  2  2]
   [ 4  4  3]]] x[2, :, :]
```

```
[[[ 2  2  1]
   [ 3  1  3]]] x[3, :, :]
```

x.shape[0]: 4

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

```
[[[-1  0  1]
   [ 1 -1 -2]]] x[:, 0, :]
```

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

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

x.shape[1]: 2

```
[[[ 3  1  2]
   [ 4  2  2]]] x[:, :, 0]
```

```
[[[-1  0  1]
   [ 1 -1 -2]]] x[:, :, 1]
```

```
[[[ 3  2  2]
   [ 4  4  3]]] x[:, :, 2]
```

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

x.shape[2]: 3

■ 5.5 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. Default-mäßig wird es mit Float-Einsen gefüllt. Wenn man Integer-Einsen benötigt, kann man den optionalen Parameter `dtype` auf `int` setzen:

```
import numpy as np

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

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

Ausgabe:

```
[[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:

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

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

Ausgabe:

```
[[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)
```

Ausgabe:

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

5.6 Arrays kopieren

5.6.1 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 Default-Wert des optionalen Parameters. Bei `numpy.copy(obj)` steht der Default-Wert 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 Speicherlayout 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.

5.6.2 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:

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}

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

a_{11}	a_{12}	a_{13}	a_{14}	a_{21}	a_{22}	a_{23}	a_{24}	a_{31}	a_{32}	a_{33}	a_{34}
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

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

a_{11}	a_{21}	a_{31}	a_{12}	a_{22}	a_{32}	a_{13}	a_{23}	a_{33}	a_{14}	a_{24}	a_{34}
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

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 C2 'C continuous?': ", C2.flags['C_CONTIGUOUS'])
```

Ausgabe:

```
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 C2 '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)
```

Ausgabe:

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

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

Ausgabe:

```
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 Array-element 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 liegen zwischen den Werten von `F[1, 1]` und `F[2, 1]` 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'])
```

Ausgabe:

```
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
```

■ 5.7 Identitätsarray

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:

- `identity`
- `eye`

5.7.1 Die `identity`-Funktion

Wir können Identitätsarrays mit der Funktion `identity` generieren:

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

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

Die Ausgabe von `identity` ist ein $n \times n$ -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))
```

Ausgabe:

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

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

Ausgabe:

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

5.7.2 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 Ausgabearrays 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 und 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.

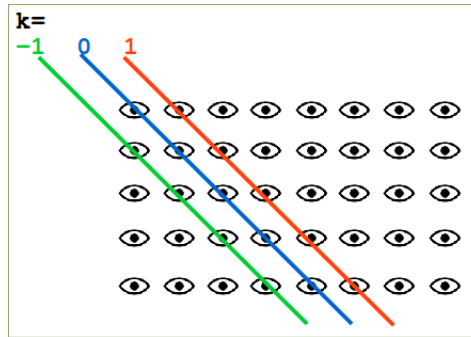
```
import numpy as np

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

Ausgabe:

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

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



5.8 Aufgaben



1. Aufgabe:

Erzeugen Sie ein beliebiges eindimensionales Array, welches Sie „v“ nennen.



2. Aufgabe:

Erzeugen Sie nun ein neues Array, das aus den ungeraden Indizes des vorher erzeugten Arrays „v“ besteht.



3. Aufgabe:

Erzeugen Sie aus „v“ ein Array in umgekehrter Reihenfolge.



4. Aufgabe:

Wie sieht die Ausgabe des folgenden Codes aus?

```
a = np.array([1, 2, 3, 4, 5])
b = a[1:4]
b[0] = 200
print(a[1])
```

**5. Aufgabe:**

Erzeugen Sie ein zweidimensionales Array mit dem Namen „m“.

**6. Aufgabe:**

Erzeugen Sie nun ein neues Array aus „m“, in dem die Elemente von jeder Reihe in umgekehrter Reihenfolge sind.

**7. Aufgabe:**

Ein weiteres, in dem die Zeilen in vertauschter Reihenfolge sind.

**8. Aufgabe:**

Erzeugen Sie ein Array aus m, in dem sowohl die Zeilen als auch die Spalten in umgekehrter Reihenfolge sind.

**9. Aufgabe:**

Schneiden Sie die erste und die letzte Reihe und Spalte ab.

■ 5.9 Lösungen

Lösung zur 1. Aufgabe:

```
import numpy as np
a = np.array([3,8,12,18,7,11,30])
```

Lösung zur 2. Aufgabe:

```
odd_elements = a[1::2]
```

Lösung zur 3. Aufgabe:

```
reverse_order = a[::-1]
```

Lösung zur 4. Aufgabe:

Die Ausgabe lautet 200, weil der Teilbereichsoperator in NumPy Sichten (Views) und keine Kopien liefert.

Lösung zur 5. Aufgabe:

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

Lösung zur 6. Aufgabe:

```
m[:, ::-1]
```

Lösung zur 7. Aufgabe:

```
m[::-1]
```

Lösung zur 8. Aufgabe:

```
m[:, :-1, ::-1]
```

Lösung zur 9. Aufgabe:

```
m[1:-1, 1:-1]
```

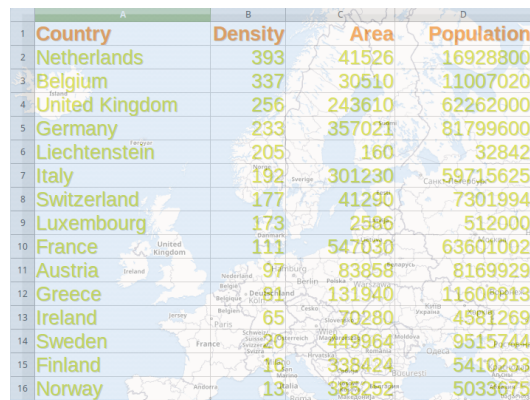

6.1 dtype

In den vorigen Kapiteln haben wir uns mit den ndarrays beschäftigt. Im zweidimensionalen Fall kann man diese als eine rechteckige Anordnung von Daten gleichen Typs sehen. In den NumPy-Arrays, die wir in unseren bisherigen Beispielen verwendet haben, benutzten wir nur grundlegende Datentypen wie Integer und Floats. Zweidimensionale NumPy-Arrays entsprechen den mathematischen Matrizen. Wir haben gesehen, dass wir mit ihnen auch wie mit Matrizen rechnen können.

Matrizen sind ein Schlüsselkonzept der linearen Algebra und werden in fast allen Gebieten der Mathematik benutzt.

Schauen wir allerdings auf die Datendarstellung, wie sie von Tabellenkalkulationsprogrammen wie beispielsweise Excel verwendet werden, erkennen wir schnell, dass das bisherige Konzept nicht flexibel genug ist. Um auch solche Daten in NumPy modellieren zu können, wird uns das Datentyp-Objekt `dtype` zur Verfügung gestellt. Dabei handelt es sich um eine Instanz der `numpy.dtype`-Klasse.

`dtype`-Objekte werden aus einer Kombination von grundlegenden Datentypen wie Integer oder Floats erzeugt. Mithilfe von `dtype` sind wir in der Lage, „strukturierte Arrays“ zu erzeugen, auch bekannt als „record arrays“. Strukturierte Arrays geben uns die Möglichkeit, verschiedene Datentypen in verschiedenen Spalten zu haben. Strukturierte Arrays haben somit – wie bereits erwähnt – Ähnlichkeiten zu Excel- oder CSV-Dokumenten. Dies ermöglicht es uns somit, Daten wie in der folgenden Tabelle mit `dtype` zu erzeugen:



	A	B	C	D
1	Country	Density	Area	Population
2	Netherlands	393	41526	16928800
3	Belgium	337	30510	11007020
4	United Kingdom	256	243610	62262000
5	Germany	233	357021	81799600
6	Liechtenstein	205	160	32842
7	Italy	192	301230	59715625
8	Switzerland	177	41290	7301994
9	Luxembourg	173	2586	512000
10	France	111	547030	63601002
11	Austria	97	83858	8169929
12	Greece	81	131940	11606813
13	Ireland	65	70280	4581269
14	Sweden	20	449964	9515744
15	Finland	16	338424	5410233
16	Norway	13	385252	5033675

Bild 6.1 Europa mit Daten

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)
```

Ausgabe:

```
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 Programms erhöhen. Dies hat noch nichts mit „strukturierten Arrays“ zu tun, die wir am Anfang dieses Kapitels erwähnt hatten.

6.2 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))
```

Ausgabe:

```
[(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'])
```

Ausgabe:

```
[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)
```

Ausgabe:

```
[(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 ebenso gut '`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)
```

Ausgabe:

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

Das Gleichheitszeichen '=' steht für die natürliche Byte-Reihenfolge¹, definiert durch das Betriebssystem. In unserem Fall bedeutet dies Little-Endian, weil wir uns auf einem Linux-Rechner befinden.

Noch ein weiterer Sachverhalt in unserem Density-Array könnte für Verwirrung sorgen. Wir definierten das Array mit einer Liste, die 1-Tupels enthält. Man könnte nun leicht annehmen, dass man auch Listen mit einem Element statt der 1-Tupel verwenden könnte. Dem ist aber nicht so. Es funktioniert nur mit Tupels. Die Tupels werden verwendet, um die Records zu definieren – in unserem 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),
```

¹ englisch: „native byte ordering“

```
( '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])
```

Ausgabe:

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

Wir können auf jedes Element individuell zugreifen:

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

Ausgabe:

```
[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))
```

Ausgabe:

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

■ 6.3 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 11 ([Lesen und Schreiben von Datendateien](#)).

```
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)
```

Ausgabe:

```
[("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)
```

Ausgabe:

```
[("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)]
```

■ 6.4 Unicode-Strings in Arrays

Wir hatten ja bereits darauf hingewiesen, dass die Strings in unserem vorigen Beispiellarray 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])
```

Ausgabe:

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

■ 6.5 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)
```

Ausgabe:

```
('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'])
```

Ausgabe:

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

■ 6.6 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)
```

Ausgabe:

```
[('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)]
```

■ 6.7 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 eine Liste benutzen, die mittels des Moduls `pickle` erzeugt worden ist. Wenn wir die gepickelte Datei `cities_and_times.pkl` einlesen, erhalten wir eine Liste mit Tupeln, die jeweils einen Städtenamen, gefolgt von einem Tag und einem Zweiertupel mit einer Uhrzeit, enthält.

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])
```

Ausgabe:

```
('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])
```

Ausgabe:

```
[('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])
```

Ausgabe:

```
[('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")
```

■ 6.8 Aufgaben

Bevor wir weitermachen, wollen wir den erlernten Stoff mit Übungsaufgaben vertiefen.



1. Aufgabe:

Definiere ein strukturiertes Array mit zwei Spalten. Die erste Spalte soll die ID eines Produktes enthalten, die sich als int32 darstellen lässt. Die zweite Spalte soll den Preis dieses Produkts enthalten.

Wie kann man die Spalte mit den Produkt-IDs ausgeben?

Wie kann man die erste Zeile ausgeben?

Wie kann man auf den dritten Artikel zugreifen, also den Artikel in der 3. Zeile?



2. Aufgabe:

Erzeuge ein Array mit der gleichen Länge wie das eben erzeugte. Die Inhalte dieses Arrays sind Integers, die der Anzahl von verkauften Einheiten für die entsprechenden Produkt-IDs entsprechen. Berechnen Sie die Verkaufserlöse pro Item und die Gesamtsumme.



3. Aufgabe:

Definiere ein strukturiertes Array für Zeiteinträge mit Stunden, Minuten und Sekunden. Die Indizes sollen „h“, „min“ und „sec“ lauten.



4. Aufgabe:

Füge zu den Zeiteinträgen aus der vorigen Aufgabe noch jeweils einen Temperaturwert hinzu.



5. Aufgabe:

Erzeugen Sie eine csv-Datei, in der eine Zeile jeweils eine Zeit und einen Temperaturwert enthält, z.B. 14:56:19 23.8 16:08:04 32.8

6.9 Lösungen

Lösung zur 1. Aufgabe:

```
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)
```

Ausgabe:

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

Lösung zur 2. Aufgabe:

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

erlöse = produkte["preise"] * verkaufszahlen

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

Ausgabe:

```
Erlöse pro Item: [1811.28 2199.65 688.38 129.39]
Gesamterlös: 4828.7000000000001
```

Lösung zur 3. Aufgabe:

```
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'])
```



Ausgabe:

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

Aus der obigen dtype-Struktur können wir auch ein zweidimensionales Array erzeugen, in dem die Spalten den Stunden, Minuten und Sekunden entsprechen:

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

Ausgabe:

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

Lösung zur 4. Aufgabe:

```
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'])
```

Ausgabe:

```
[[(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]
```

Lösung zur 5. Aufgabe:

```
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
```

Ausgabe:

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

Arbeitet man mit Arrays, kommt man früher oder später an den Punkt, dass man die Gestalt, also die Shape eines Arrays oder die Dimension von Arrays verändern will bzw. muss. Die dazu nötigen Funktionalitäten lernen wir in diesem Kapitel kennen. Wir werden auch zeigen, wie man Arrays zusammenhängt bzw. konkateniert. Weiterhin werden wir die Möglichkeiten demonstrieren, wie man existierende Arrays um weitere Dimensionen erweitern 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.

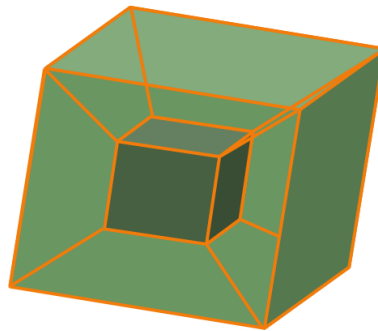


Bild 7.1 Tesseract

In dem Bild geht es um die Veranschaulichung des vierdimensionalen Raumes. Es wird ein Tesseract dargestellt, den man auch als vierdimensionalen Hyperwürfel bezeichnet. Ein Tesseract kann man als die Übertragung des Konzeptes eines dreidimensionalen Würfels in den vierdimensionalen Raum sehen. Ein Tesseract verhält sich zum Würfel wie ein Würfel zum Quadrat.

■ 7.1 Reduktion und Reshape von Arrays

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

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

7.1.1 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"))
```

Ausgabe:

```
[ 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]
```

7.1.2 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"))
```

Ausgabe:

```
[ 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]
```

7.1.3 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))
```

Ausgabe:

```
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
```

7.1.4 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)
```

Ausgabe:

```
[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)
```

Ausgabe:

```
[[[ 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)
```

Ausgabe:

```
True
```

■ 7.2 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)
```

Ausgabe:

```
[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` zusammengefügt 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)
```

Ausgabe:

```
[[ 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)
```

Ausgabe:

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

■ 7.3 Weitere Dimensionen hinzufügen

Weitere Dimensionen können zu einem Array mithilfe 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)
```

Ausgabe:

```
[[ 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)
```

Ausgabe:

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

7.4 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=" ")
```

Ausgabe:

```
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])
```

Ausgabe:

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

Ausgabe:

```
[[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)))
```

Ausgabe:

```
[[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]]
```


Wenn wir uns das vorige Beispiel anschauen, sehen wir, dass wir ein neues Array durch Ver-
vielfachung des Arrays A in horizontaler und vertikaler Richtung geschaffen haben. Würde
man A als eine Fliese ansehen, haben wir folgendes Muster geschaffen:

```
array([[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]])
```

Apropos Fliese: Fliese heißt auf Englisch „tile“, und so heißt auch die nächste Funktion, die
wir im nächsten Unterkapitel besprechen werden. Im Prinzip können wir mit `tile` das ma-
chen, was wir eben umständlich mittels `column_stack` und `row_stack` erzeugt haben.

■ 7.5 „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.

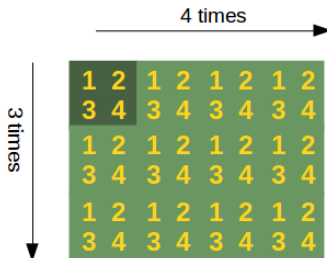
Beispiel:

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

Ausgabe:

```
[[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)
```

Ausgabe:

```
[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))
```

Ausgabe:

```
[[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)))
```

Ausgabe:

```
[[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)))
```

Ausgabe:

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


Numerische Operationen auf NumPy-Arrays

In den bisherigen Kapiteln haben wir uns im Wesentlichen auf die Erzeugung von NumPy-Datenstrukturen konzentriert. Was wäre jedoch ein Datentyp ohne Operatoren? So möchte man beispielsweise zwei Arrays addieren oder multiplizieren können. Wir werden sehen, dass man mit Arrays ebenso einfach rechnen kann wie mit Integers, Floats oder Strings. Bei zwei Arrays stellt sich auch die Frage, wie man aus diesen das Matrizenprodukt bilden kann. Letzteres natürlich nur unter der Voraussetzung, dass die Shapes der beiden Arrays passend sind.

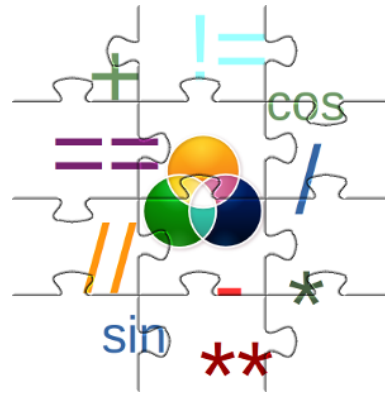


Bild 8.1 Operatoren fraktale Darstellung

Wir werden sehen, dass man in NumPy alle von der Mathematik gewohnten Operatoren auf die NumPy-Arrays anwenden kann. Auch die Verknüpfung von NumPy-Arrays mit Skalaren ist möglich, d.h. man kann zu einem Array einfach eine Integer- oder eine Float-Zahl addieren, und die Addition erfolgt dann komponentenweise.

■ 8.1 Operatoren und Skalare

Beginnen wir mit einem simplen Beispiel, was bei vielen auch im täglichen Leben von Belang ist. Man hat eine Tabelle von Lebensmitteln, deren Nährwert oder genauer deren „quantitativer Nährwert“, also die verwertbare Energie, in Kalorien angegeben ist. Nun will man aber gerne die entsprechenden Werte in die Einheit Joule umrechnen.

Lebensmittel	kcal pro 100 g
Wassermelone	30
Apfel	52
Banane	88
Karotte	36
Kartoffel	86

Lebensmittel	kcal pro 100 g
Avocado	160
Bratwurst	375
Rinderfilet	115
Bier	43
Wein	71

Im folgenden Beispiel haben wir die Kalorienwerte der obigen Tabelle in einem Array `kalorientabelle` aufgenommen. Eigentlich handelt es sich bei der Einheit Kalorie (cal) um eine veraltete Maßeinheit der Energie. Deshalb wollen wir die Werte in Joule umrechnen, genau genommen Kilojoule. Eine Kalorie entspricht ca. 4,1868 Joule.

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

jouletabelle = kalorientabelle * 4.1868

print(jouletabelle)
```

Ausgabe:

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

Wir sehen, dass alle Werte des Arrays `kalorientabelle` mit dem Faktor 4.1868 multipliziert worden sind. Da es sich bei den Kalorienwerten um vage Werte gehandelt hat, macht es keinen Sinn, die Nachkommastellen zu erhalten, denn sie würden eine trügerische Genauigkeit vortäuschen. Wir könnten also statt obiger Berechnung auch wie folgt vorgehen:

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

Ausgabe:

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

Außerdem macht es Sinn, dieses Array als ein Integer-Array zu definieren:

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

Ausgabe:

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

Eben haben wir erfolgreich ein Array (oder mathematisch betrachtet einen „Vektor“) komponentenweise mit einem Skalar multipliziert. Ebenso leicht lassen sich auch Additionen, Subtraktionen und Divisionen durchführen:

```
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)
```

Ausgabe:

```
[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. ]
```

■ 8.2 Arithmetische Operationen auf zwei Arrays

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

```
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)
```

Ausgabe:

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

Wenn wir wissen wollen, wieviele Kalorien unsere gesundheitsbewusste Person insgesamt zu sich genommen hat, können wir dies mittels der Funktion `sum` bestimmen:

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

Ausgabe:

```
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)
```

Ausgabe:

```
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!

■ 8.3 Matrizenmultiplikation und Skalarprodukt

8.3.1 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.

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

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]`.

8.3.2 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))
```

Ausgabe:

```
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))
```

Ausgabe:

```
[[ 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)
```

Ausgabe:

```
[[ 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])
```

Ausgabe:

```
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.

8.3.3 Das dot-Produkt im 3-dimensionalen Fall

Es wird ziemlich verzwickelt, 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)
```

Ausgabe:

```
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)
```

Ausgabe:

```
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))
```

Ausgabe:

```
[[[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)
```

Ausgabe:

```
(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)
```

Ausgabe:

```
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))
```

Ausgabe:

```
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])
```

Ausgabe:

```
[[[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)) )
```

Ausgabe:

True

Es gilt also, was wir eingangs behauptet haben:

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

■ 8.4 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
```

Ausgabe:

```
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))
```

Ausgabe:

False
True

■ 8.5 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))
```

Ausgabe:

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

■ 8.6 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.

Mithilfe 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 sich die beiden Arrays in ihrer Anzahl von Dimensionen 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)
```

Ausgabe:

```
[12 20  4]
```

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

```
import numpy as np

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

Ausgabe:

```
[12 20  4]
```

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

8.6.1 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)
```

Ausgabe:

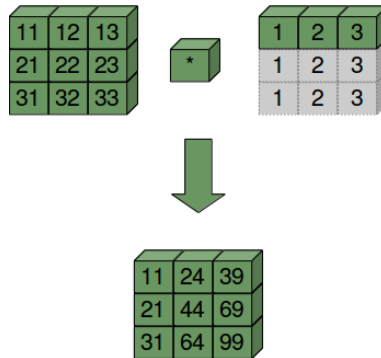
```
(3, 3)
(3,)

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

Ausgabe:

```
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)
```

Ausgabe:

```
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)
```

Ausgabe:

```
(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)
```

Ausgabe:

```
(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]]]
```

8.6.2 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)))
```

Ausgabe:

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

Alternativ können wir auch `newaxis` verwenden:

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

Ausgabe:

```
[[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]
```

Ausgabe:

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

B wird benutzt, als wäre es wie folgt aufgebaut:

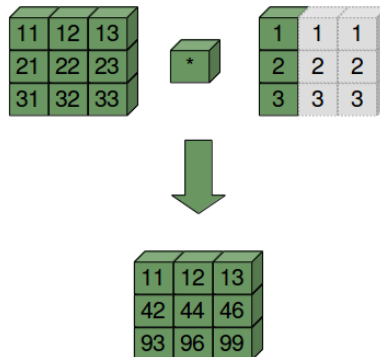
```
B = np.array([1, 2, 3])
B = B.reshape((B.shape[0], 1))
print("Shape nach Anwendung der ersten Regel: ", B.shape)
B = np.tile(B, (1, 3))
print("Shape nach Anwendung der zweiten Regel: ", B.shape)
print()

print(B)
```

Ausgabe:

```
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]]
```



8.6.3 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:

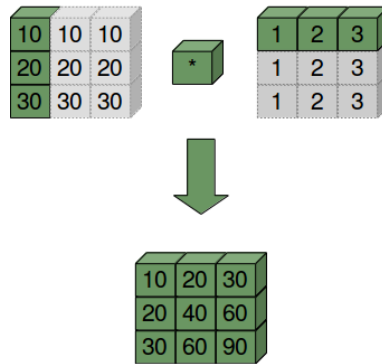
```
A = np.array([10, 20, 30])
B = np.array([1, 2, 3])

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

print(A * B)
```

Ausgabe:

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



8.7 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 $n \times n$, 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]))

```

Ausgabe:

```

[ 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]]

```

■ 8.8 Ufuncs

Bisher haben wir diverse Operatoren wie Addition, Subtraktion, Multiplikation usw. 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 `+`, `-`, `*` usw. 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 `<`, `<=`, `==`, `>`, `>=`, `!=` usw. 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.

8.8.1 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 Möglichkeit
print(add(x, y))
print("Typ der 'add'-Funktion:", type(add))
```

Ausgabe:

```
[ 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)))
```

Ausgabe:

```
[ 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))
```

Ausgabe:

```
[ 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))
```

Ausgabe:

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

8.8.2 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)
```

Ausgabe:

```
[46.08 52.74 55.62]
```

In den obigen Fällen wurde zuerst ein neues ndarray-Array mit dem Ergebnis erzeugt und dann 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)
```

Ausgabe:

```
[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)
```

Ausgabe:

```
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
```

8.8.3 accumulate

Auf die Ufunc-Funktionen 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))
```

Ausgabe:

```
[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))
```

Ausgabe:

```
[[ 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]]
```

8.8.4 reduce

Die Arbeitsweise von `reduce` im eindimensionalen Fall kann wie folgt beschrieben 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``
```

Ausgabe:

```
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))
```

Ausgabe:

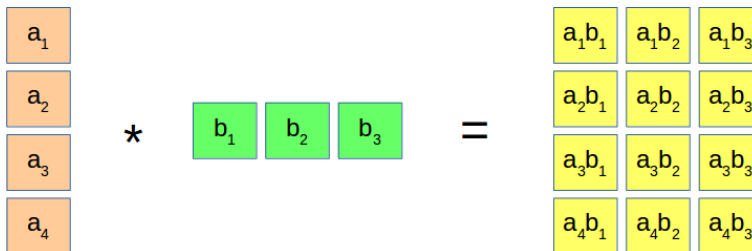
```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Reduce über Zeilen, d.h. axis = 0:
[12 15 18 21]
Reduce über Spalten, d.h. axis = 1:
[ 6 22 38]
Reduce über Zeilen und Spalten:
66
Letzteres geht einfacher mit axis = None:
66
```

8.8.5 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 spezielle Produkt zweier Vektoren auch als das „dyadische Produkt“ oder „tensorielle Produkt“ bekannt. Das Ergebnis ist eine Matrix.

Die Arbeitsweise von `outer` verdeutlichen wir im folgenden Diagramm:



Im folgenden Beispiel finden wir obiges Diagramm mit realen Werten dargestellt:

```
import numpy as np

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

Ausgabe:

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

8.8.6 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)
```

Ausgabe:

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

8.9 Aufgaben



1. Aufgabe:

Gegeben sei ein Array `B = np.array([1, 2, 3])`.

Erzeugen Sie aus diesem Array automatisch ein Array der Form:

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

Dieses Array hat die Shape (3, 1, 3).



2. Aufgabe:

Erzeugen Sie ein Integer-Array `A` mit Shape (3, 4). Erzeugen Sie ein eindimensionales Array `Z` mit den jeweiligen Zeilenminima und ein eindimensionales Array `S` mit den Spaltenminima. Wie lautet das absolute Minimum des Arrays?

■ 8.10 Lösungen

Lösung zur 1. Aufgabe:

```
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)
```

Ausgabe:

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

 [[2 2 2]]

 [[3 3 3]]]
```

Lösung zur 2. Aufgabe:

```
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))
```

Ausgabe:

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


■ 9.1 Einführung

„Jeder Amerikaner sollte über dem Durchschnittseinkommen liegen, und meine Regierung wird einen Blick darauf haben, dass sie es auch bekommen.“ (Bill Clinton)¹

Statistik und Wahrscheinlichkeitsrechnung begegnen 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



Bild 9.1 Würfel

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.

¹ Im Original: „Every American should have above average income, and my administration is going to see they get it.“ Obwohl dieses Zitat extrem häufig zitiert wird, konnten wir leider keine zitierfähige Quelle finden!

■ 9.2 Zufallszahlen mit Python

9.2.1 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 einen 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`-Moduls:

Warnung: Beachten Sie bitte, dass der Pseudo-Zufallsgenerator 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.²

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)
```

Ausgabe:

```
0.6402066778674983
```

Nun wollen wir eine kryptografisch starke Zufallszahl erzeugen:

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

Ausgabe:

```
0.7494133391755209
```

9.2.2 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 = []
```

² 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.“

```

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))

```

Ausgabe:

```
[0.04038461684304784, 0.04435712980972839, 0.33758943699957844]
```

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

```

%%timeit
random_list(100, secure=True)

```

Ausgabe:

```
352 µs ± 3.78 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

```

%%timeit
random_list(100, secure=False)

```

Ausgabe:

```
6.72 µs ± 49.5 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`-Untermodule von `numpy` erzeugen:

```

import numpy as np

np.random.random(10)

```

Ausgabe:

```

array([0.36529563, 0.70867079, 0.83847536, 0.57047906, 0.81767584,
       0.75697413, 0.46364052, 0.39767732, 0.2459246 , 0.3925199 ])

%%timeit
np.random.random(100)

```

Ausgabe:

```
1.58 µs ± 3.82 ns per loop (mean ± std. dev. of 7 runs,
1000000 loops each)
```

Allerdings werden auch hier keine sicheren Zufallszahlen erzeugt!

■ 9.3 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)]
```

Ausgabe:

```
[5, 6, 6, 3, 5, 3, 5, 5, 3, 6, 4, 4, 4, 3, 6, 4, 3, 6, 4, 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)
```

Ausgabe:

```
5

import secrets

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

Ausgabe:

```
2
```

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}")
```

Ausgabe:

```
Counter({3: 16765, 6: 16757, 1: 16659, 2: 16643, 5: 16642, 4: 16534})
Augenzahl: 6,           relative Häufigkeit: 0.16757
Augenzahl: 5,           relative Häufigkeit: 0.16642
Augenzahl: 3,           relative Häufigkeit: 0.16765
Augenzahl: 1,           relative Häufigkeit: 0.16659
Augenzahl: 2,           relative Häufigkeit: 0.16643
Augenzahl: 4,           relative Häufigkeit: 0.16534
```

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}")
```

Ausgabe:

```
Augenzahl: 3,           relative Häufigkeit: 0.16782
Augenzahl: 2,           relative Häufigkeit: 0.16795
Augenzahl: 1,           relative Häufigkeit: 0.16421
Augenzahl: 4,           relative Häufigkeit: 0.16683
Augenzahl: 6,           relative Häufigkeit: 0.16648
Augenzahl: 5,           relative Häufigkeit: 0.16671
```

Sicherlich haben Sie bemerkt, dass wir in `random.randint` beim zweiten Parameter eine 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“ (inklusive) 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)))
```

Ausgabe:

Eine Integer-Zahl: 3

Ein eindimensionales Array mit einem Element:
[5]

Ein eindimensionales Array mit zehn Elementen:
[6 6 1 2 5 2 5 1 2 4]

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

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

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 Würfel 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.

■ 9.4 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, aber auch Iteratoren sein. 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)))
```

Ausgabe:

```
c
scientist
cherries
2
```

`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.

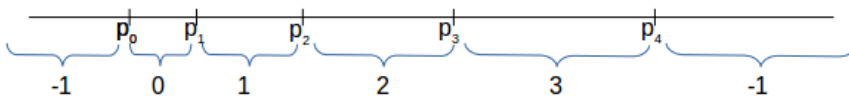
Wir definieren nun eine Funktion `weighted_choice`, die wie `random.choice` ein zufälliges Element einer Sequenz zurückliefert, jedoch sind die Elemente der Sequenz gewichtet.

■ 9.5 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 $p_0, p_1, p_2, \dots, p_n$

Die Funktion liefert i zurück, wenn $p_i < x < p_{i+1}$. -1 wird zurückgeliefert, wenn x kleiner als p_0 ist oder x größer oder gleich p_n 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=" ",)
```

Ausgabe:

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

■ 9.6 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)
```

Ausgabe:

```
[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 $[0, \text{cum_weights}[0])$ liegt, 1/5. Die Wahrscheinlichkeit, dass x im Intervall $[\text{cum_weights}[0], \text{cum_weights}[1])$ liegt, ist 1/2. Letztlich ist die Wahrscheinlichkeit, dass x im Intervall $[\text{cum_weights}[1], \text{cum_weights}[2])$ liegt, 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

$$1 - P(1) - P(6) = 4 \times p$$

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 Würfeln 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()))

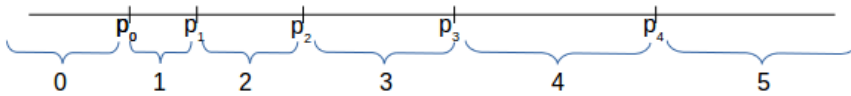
```

Ausgabe:

```
[0.0855, 0.1618, 0.1633, 0.1655, 0.1691, 0.2548]
```

Die Werte der Aufteilungsliste definieren die Intervalle, in denen wir den Wert x erwarten. Wenn der Wert x kleiner als p_0 oder größer/gleich p_n , liefern wir -1 zurück.

Wir könnten unsere erste Unterteilung als ein Intervall von $-\infty$ bis p_0 definieren und 0 zurückliefern. Die letzte Unterteilung wäre dann ein Intervall von p_n bis ∞ .



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=" ")

```

Ausgabe:

```
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,
```

■ 9.7 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)
```

Ausgabe:

```
[[0.6510252  0.53118781 0.52933256 0.27697436]
 [0.89423487 0.29696093 0.64437663 0.9924553 ]
 [0.73435037 0.67296525 0.16193815 0.42498575]]
```

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)
```

Ausgabe:

```
[0.38854095 0.87111203 0.03884354 0.26760445 0.11338149 0.04831298
 0.8127288 ]
[0.85162568 0.38649672 0.80057558 0.09931203 0.54104196 0.21070834
 0.01742418]
```

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)
```

Ausgabe:

```
[[-2.90298201 -0.19131702 -1.88645141 -0.8777939 ]
 [-2.19279986 -2.08277676 -0.95616961 -3.06663613]
 [ 1.12826615 -2.61598991  1.57970249 -0.37743384]]
```

Das Standardmodul `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 Lottoziehung in Deutschland:

```
import random

print(random.sample(range(1, 50), 6))
```

Ausgabe:

```
[14, 17, 18, 4, 12, 33]
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))
```

Ausgabe:

```
['pink', 'orange', 'green']
['orange', 'black', 'green']
['blue', 'black', 'red']
['black', 'pink', 'orange']
['pink', 'black', 'orange']
['white', 'red', 'yellow']
['blue', 'orange', 'yellow']
['blue', 'black', 'orange']
['blue', 'orange', 'green']
['blue', 'red', 'yellow']
```

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)
```

Ausgabe:

```
0.7153
0.7134
```

■ 9.8 Kartesische Auswahl

Die Funktion `cartesian_choice`, die wir im Folgenden definieren werden, ist nach dem Kartesischen Produkt aus der Mengenlehre benannt.³

9.8.1 Kartesisches Produkt

Das kartesische Produkt, auch Mengenprodukt genannt, ist ein Konstruktionsprinzip, um aus Mengen eine neue Menge zu konstruieren.

³ Das kartesische Produkt ist nach dem französischen Mathematiker René Descartes benannt.

Für zwei Mengen A und B ist das kartesische Produkt $A \times B$ die Menge aller geordneten Paare (a, b) , für die $a \in A$ und $b \in B$ gilt:

$$A \times B = \{ (a, b) \mid a \in A \text{ and } b \in B \}$$

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 A_1, A_2, \dots, A_n , können wir das kartesische Produkt entsprechend wie folgt bilden:

$$A_1 \times A_2 \times \dots \times A_n = \{ (a_1, a_2, \dots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n \}$$

Das kartesische Produkt aus n Mengen wird manchmal auch n -faches kartesisches Produkt genannt.

9.8.2 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"])
```

Ausgabe:

```
['A', 'green', 'light', 'smells']
```

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) + ".")

```

Ausgabe:

```

It may or may not be true:
No red programming language thinks of happiness.
A green programming language thinks of air.
The blue light thinks of chocolate.
A blue Python smells of happiness.
A blue light consists of happiness.
Every green programming language consists of happiness.
The red Python dreams of wisdom.
No grey light thinks of air.
The green programming language dreams of happiness.
Each yellow door dreams of happiness.

```

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)

```

Ausgabe:

```

It may or may not be true:
dreams 0.2939
smells 0.4014
thinks 0.2008
is made of 0.1039

```

■ 9.9 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 entweder 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 verwen-

dete Algorithmus deterministisch ist. Pseudozufallszahlen sind für viele Situationen gut genug, aber sie sind nicht „wirklich“ zufällig, wenn man beispielsweise Würfel- oder Lotterieziehungen simulieren will.

Die Webseite **RANDOM.ORG**⁴ erhebt den Anspruch, „echte“ Zufallszahlen zu generieren. Sie nutzen die Zufälligkeit aus atmosphärischen Störungen. Die daraus gewonnenen numerischen Werte sind für viele Anwendungen besser geeignet als die pseudozufälligen Zahlen aus einem Computerprogramm.

■ 9.10 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 Zufallswerts. 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 Zufallsgenerator zum ersten Mal aufgerufen wird, so muss der erste „Zufalls“-Wert irgendwie erzeugt werden.

Setzt man den Startwert (also die Seed) eines Pseudo-Zufallszahlengenerators, 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.

⁴ <http://www.random.org>

Für Python trifft dies ebenfalls zu. `help(random.seed)` sagt, dass wenn die Funktion mit `None` oder keinem Argument aufgerufen wird, der Seed-Wert aus der aktuellen Systemzeit oder einer anderen systemspezifischen Zufallsquelle generiert wird.

Ruft man `seed` ohne Parameter auf, wird der Startwert entweder aus der aktuellen Zeit oder aus einer systemspezifischen Zufallsquelle gewonnen.⁵ Übergibt man einen Wert, so wird dieser benutzt, um den Startwert zu berechnen.

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=" ", )
```

Ausgabe:

```
2, 1, 5, 4, 4, 3, 2, 9, 2, 10,
Nochmals die selben Zufallszahlen:
2, 1, 5, 4, 4, 3, 2, 9, 2, 10,
```

■ 9.11 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 Gauss'sche Verteilung aufweisen mit dem Hauptwert $\mu = 550$ und einer Standardabweichung von $\sigma = 30$.

Wenn wir uns die `help`-Information anschauen, sehen wir, dass der wesentliche Unterschied darin besteht, dass `gauss` etwas schneller und nicht Thread-sicher ist, während `random.normalvariate` Thread-sicher ist.

```
import random

help(random.normalvariate)
help(random.gauss)
```

Ausgabe:

```
Help on method normalvariate in module random:
```

⁵ freie Übersetzung aus der `help`-Ausgabe von `seed`: „None or no argument seeds from current time or from an operating system specific randomness source if available.“

`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 Zufallszahlen 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]
```

Ausgabe:

```
[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()

<Figure size 640x480 with 1 Axes>
```

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:
```

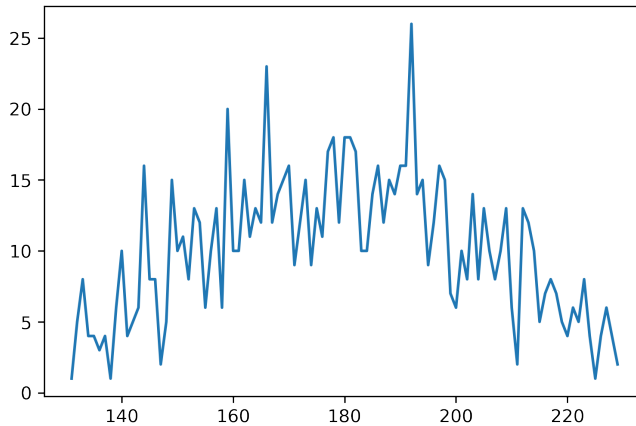
```

frequencies[value] = frequencies.get(value, 0) + 1
values.append(value)

freq = list(frequencies.items())
freq.sort()

plt.plot(*list(zip(*freq)))
plt.show()

```



■ 9.12 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

```

Ausgabe:

```
0.800606
```


Eine weitere gute Idee ist es, die Aufgabe mit einem Generator zu implementieren.

```
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
```

Ausgabe:

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.⁶

```
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)
```

⁶ Ich bedanke mich bei Dr. Hanno Baehr, der mich auf das Problem der „Zufallsextrahierung“ aufmerksam gemacht hatte. Hanno hat den theoretischen Rahmen entworfen. Während einer Nacht-Session nach einem Python-Seminar im Januar 2014 in der Bar „Zeit & Raum“ (engl. „Time & Space“) in Nürnberg habe ich ein Python-Programm implementiert, um seine theoretische Lösung empirisch zu unterstützen.

```
n = 1000000
sum(x for x in firstn(ebitter(random_ones_and_zeros(0.8)), n)) / n
```

Ausgabe:

```
0.499749
```

```
n = 1000000
sum(x for x in firstn(ebitter2(random_ones_and_zeros(0.8)), n)) / n
```

Ausgabe:

```
0.500011
```

Grundlagen der Theorie:

Unser erster Generator erzeugt einen Bitstrom B_0, B_1, B_2, \dots

Wir prüfen nun ein beliebiges Paar aufeinanderfolgender Bits B_i, B_{i+1}, \dots

Ein solches Paar kann die Werte 01, 10, 00 oder 11 haben. Die Wahrscheinlichkeit $p(01) = (p-1) \times p$ und die Wahrscheinlichkeit $p(10) = p \times (p-1)$ ergibt eine kombinierte Wahrscheinlichkeit, dass die aufeinanderfolgenden Bits entweder 01 oder 10 sind, von $2 \times (p-1) \times p$.

Betrachten wir ein anderes Bit B_{i+2} . Wie ist die Wahrscheinlichkeit dieser beiden

$$B_i + B_{i+1} = 1$$

und

$$B_{i+1} + B_{i+2} = 1 ?$$

Die möglichen Ausgaben passen auf die Bedingungen, und die zugehörigen Wahrscheinlichkeiten sind in der folgenden Tabelle aufgeführt:

Wahrscheinlichkeit	B_i	B_{i+1}	B_{i+2}
$p^2 \times (1-p)$	0	1	0
$p \times (1-p)^2$	1	0	1

Wir bezeichnen das Ergebnis $\text{sum}(B_i, B_{i+1})=1$ als X_1 und entsprechend dazu $\text{sum}(B_{i+1}, B_{i+2})=1$ als X_2 .

Die verknüpfte Wahrscheinlichkeit $P(X_1, X_2) = p^2 \times (1-p) + p \times (1-p)^2$ kann zu $p \times (1-p)$ umgestellt werden.

Die bedingte Wahrscheinlichkeit von X_2 gibt X_1 :

$$P(X_2 | X_1) = P(X_1, X_2) / P(X_2)$$

$$P(X_2 | X_1) = p \times (1-p) / 2 \times p \times (1-p) = 1 / 2$$

9.13 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 Prozentwert (`min_percent`) und einem maximalen Prozentwert (`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)
```

Ausgabe:

```
[0.53787857 0.66884249 0.72129381 0.12525482 0.19077876 0.85915589
 0.97208126 0.17522868 0.53061524 0.65013535 0.12985828 0.77437537]
[1.02303029 1.0335074 1.0377035 0.99002039 0.9952623 1.04873247
 1.0577665 0.99401829 1.02244922 1.03201083 0.99038866 1.04195003]
```

Für die neuen Verkaufszahlen nach einem Jahr multiplizieren wir das `sales`-Array mit dem `growthrates`-Array:

```
sales * growthrates
```

Ausgabe:

```
array([1274.58320255, 2294.38642652, 1697.44426183, 1916.92697158,
       997.28268363, 2201.42579308, 765.81236903, 984.44589829,
       553.59490533, 1821.4991113 , 1785.51229614, 2082.85810987])
```

Um eine nachhaltige Verkaufsentwicklung zu erhalten, verändern wir die Wachstumsraten 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.

9.14 Aufgaben



1. Aufgabe:

Wir starten mit einer kleinen Aufgabe mit Würfeln. Beweisen Sie empirisch, indem Sie ein Simulationsprogramm schreiben, dass die Wahrscheinlichkeit für das kombinierte Ereignis „Egal welche Zahl gewürfelt wurde“ (E) und „Eine Zahl größer als 2 wurde gewürfelt“ $1/3$ ist.



2. Aufgabe:

Die Datei „`universities_uk.txt`“ beinhaltet eine Liste der Universitäten im Vereinigten Königreich zur Einschreibung zwischen 2013-2014. (Quelle: [Wikipedia](#)):

Rank	Institution	Undergraduates	Postgraduates	Total students
1	Open University in England	112,535	10,955	123,490
2	University of Manchester	26,485	11,440	37,925
3	University of Nottingham	24,885	8,385	33,270
4	Sheffield Hallam University	25,985	7,115	33,100
5	University of Birmingham	19,185	13,150	32,335
6	Manchester Metropolitan University	26,635	5,525	32,160
7	University of Leeds	23,265	7,710	30,975
8	Cardiff University	21,495	8,685	30,180
9	University of South Wales	23,890	5,310	29,195
...				

Schreiben Sie eine Funktion, die ein Tupel (`universities`, `enrollments`, `total_number_of_students`) zurückliefert mit:

`universities`: Liste der Namen der Universitäten

`enrollments`: zugehörige Liste mit Einschreibungen

`total_number_of_students`: Über alle Universitäten

Jetzt können Sie 100.000 fiktive Studenten immatrikulieren mit einer Wahrscheinlichkeit, die den „echten“ Einschreibungen entspricht.



3. Aufgabe:



In dieser Aufgabe wollen wir eine Zeitreise unternehmen. Wir begeben uns zurück in das antike Pythonia (Πηθωνία). Es war in der Zeit, in der der König Pysseus als gütiger Diktator regierte. Jene Zeit, als Pysseus seine Botschafter aussandte, um durch die Welt zu reisen und zu verkünden, dass es für seine Prinzen Anacondos (Ανακονδος), Cobrion (Κομπριον), Boatos (Μποατος) und Addokles (Ανδοκλης) an der Zeit sei zu heiraten. Um die geeigneten Kandidatinnen zu finden, veranstaltete Pysseus einen Programmierwettbewerb zwischen den ebenso holden wie tapferen Amazonen, besser bekannt als Py-

thanier aus Pythonia. 11 Amazonen gelang es, in diesem Programmierwettbewerb zu bestehen, wohl auch weil sie die damals noch junge Programmiersprache Python benutzt hatten:

1. Die ätherische Airla (Αίρλα)
2. Barbara (Βαρβαρα), die Eine aus dem fremden Land
3. Eos (Ηως), sieht in der Dämmerung göttlich aus
4. Die süße Glykeria (Γλυκερία)
5. Die anmutige Hanna (Αννα)
6. Helen (Ελενη), das Licht in der Dunkelheit
7. Der gute Engel Agathangelos (Αγαθαγγελος)
8. Die violett getönte Wolke Iokaste (Ιοκαστη)
9. Medousa (Μεδουσα), die Wächterin
10. Die selbstbestimmende Sofronia (Σωφρονια)
11. Andromeda (Ανδρομεδα), die eine, die wie eine Mann oder ein Krieger denkt

Das Los sollte nun entscheiden, welche vier zu Prinzessinnen würden. Zu Beginn hatten alle die gleiche Chance gezogen zu werden. Aus Gründen, die heute nicht mehr bekannt sind, wusste Pysseus jedoch, dass sich die Wahrscheinlichkeiten mit jedem neuen Tag änderten: Sie sank um $1/13$ für die ersten 7 Amazonen und stieg um $1/12$ für die letzten 4 Amazonen. Da die letzten vier der Liste auch seinen persönlichen Wünschen entsprachen, beschloss er, die Lotterie ein paar Tage in die Zukunft zu verschieben.

Wie lange musste der König warten, bis er zu 90 % sicher sein konnte, dass seine Prinzen Anacondas, Cobrion, Boatos und Addokles die Amazonen Iokaste, Medousa, Sofronia und Andromeda heiraten würden?

■ 9.15 Lösungen

Lösung zur 1. Aufgabe:

```
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))
```

Ausgabe:

```
0.5057 0.6716 0.3397
```

Lösung zur 2. Aufgabe:

Wir schreiben zuerst die Funktion „process_datafile“, um die Daten aus der Datei zu verarbeiten:

```
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:

```
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)
```

Ausgabe:

```

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:

```

normalized_enrollments = [ students / total_students \
                           for students in enrollments]

# enrolling a virtual student:
print(weighted_choice(universities, normalized_enrollments))

```

Ausgabe:

```
University of Leeds
```

Die Aufgabe war, 100.000 fiktive Studenten zu „immatrikulieren“. Dies kann mit einer Schleife einfach durchgeführt werden:

```

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)

```

Ausgabe:

```

[ ('Open University in England', 5215),
  ('University of Manchester', 1603),
  ('University of Birmingham', 1423),
  ('University of Nottingham', 1414),
  ('Sheffield Hallam University', 1391),
  ('Manchester Metropolitan University', 1366),
  ('University of Leeds', 1352),
  ('Cardiff University', 1338),
  ('University of South Wales', 1226),
  ('University of Glasgow', 1204),
  ('University College London', 1194),
  ('University of Edinburgh', 1178),

```

```
( 'Northumbria University', 1177),
( 'University of Central Lancashire', 1175),
( 'University of the West of England', 1173),
( 'University of Sheffield', 1162),
( "King's College London", 1128),
( 'University of Oxford', 1126),
( 'Ulster University', 1126),
( 'University of Plymouth', 1117)]
```

Lösung zur 3. Aufgabe:

Die Sammlung der Amazonen ist als Liste implementiert, während wir für die Menge aus Pysseusses Favoritinnen auswählen. Die Gewichtung liegt zu Beginn bei 1/11 für alle, d.h. $1/\text{len}(\text{amazons})$.

Jeder Schleifendurchlauf 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_sample` und `weighted_sample_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)

```

Ausgabe:

```

1.7027899649983738
Number of days, he has to wait: 33

```

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)

```

Ausgabe:

```

18.861382681003306
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)
```

Ausgabe:

```
1.8446880190022057
Number of days, he has to wait: 32
```


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)
```

Ausgabe:

```
[ 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)
```

Ausgabe:

```
[ 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],
```



Bild 10.1 Maskierte Matrix

```
[55, 42, 17, 18]])
```

```
print(B >= 42)
```

Ausgabe:

```
[[ 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)
```

Ausgabe:

```
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.

■ 10.1 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)
```

Ausgabe:

```
[ 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 $A \leq 5$ 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)
```

Ausgabe:

```
[ True False  True False False False  True]
[123 190 100]
```

■ 10.2 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]
```

Ausgabe:

```
array([123, 190, 99, 188, 77, 188])
```

Wie wir sehen, können Indizes mehrfach und in beliebiger Reihenfolge auftreten!

10.2.1 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.

10.2.2 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)
```

Ausgabe:

```
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]
```

■ 10.3 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()])
```

Ausgabe:

```
(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 nonzero-Elements in der Form [Zeile, Spalte]:

```
np.transpose(A.nonzero())
```

Ausgabe:

```
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)
```

Ausgabe:

```
[[ 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)
```

Ausgabe:

```
(array([0, 0, 0, 0, 1, 1, 1, 2, 2]),
 array([0, 1, 2, 3, 0, 1, 2, 0, 1]))
```

10.3.1 Übung

Berechnen Sie die Primzahlen zwischen 0 und 100 mithilfe eines Booleschen Arrays.

10.3.2 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))
```

Ausgabe:

```
(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]),)
```

10.3.3 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.

Python bietet viele Möglichkeiten, Daten aus Dateien zu lesen und in Dateien zu schreiben. Wie man dies ohne zusätzliche Module mit reinem Python machen kann, haben wir im Kapitel 3.4 ([Dateien lesen und schreiben](#)) gesehen.

NumPy bietet jedoch optimierte Methoden für die speziellen NumPy-Datenstrukturen, die es einem ermöglichen, mit einem Befehl komplette Arrays einzulesen oder herauszuschreiben. In unserem Kapitel 6 ([Datentyp-Objekt: dtype](#)) haben wir bereits die NumPy-Funktionen `genfromtxt`, `loadtxt` und `savetxt` kennengelernt. In diesem Kapitel wollen wir diese und andere Funktionen von NumPy näher betrachten.

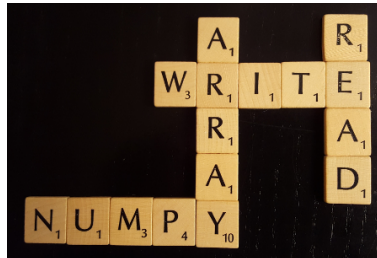


Bild 11.1 Read und Write als Scrabble

11.1 Textdateien 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 Textdatei 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 Textdatei, deren Inhalt wie folgt aussieht, wenn wir ihn uns in einem Linux-Terminal anschauen:¹

¹ Unter Windows kann man das Kommando 'type test.txt' benutzen.

```
$ more test.txt
1.0000000000000000e+00 2.0000000000000000e+00
3.0000000000000000e+00 4.0000000000000000e+00
5.0000000000000000e+00 6.0000000000000000e+00
```

Da das Array aus Integers 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. Wir können das Verhalten anpassen, indem wir dem Parameter `delimiter` einen String mitgeben. In den meisten Fällen wird dies ein einzelnes Zeichen sein. Jedoch kann ebenso eine ganze Zeichensequenz übergeben werden, z.B. ein Smiley „:-)“:

```
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
```

Parameter	Bedeutung
X	Array-ähnliche Daten, die in einer Textdatei gespeichert werden sollen.
fmt	String oder Sequenz von Strings, optional. Ein einzelner Format-String (%10.5f), eine Sequenz aus String-Formaten oder ein Multi-Format-String, z.B. 'Iteration %d – %10.5f', wobei hier der „delimiter“ ignoriert wird. Für komplexe „X“ sind folgende Optionen für „fmt“ erlaubt: a) Ein einzelnes Spezifikationsymbol, „fmt='%4e'“, liefert eine Zahlenformatierung wie „' (%s+%sj)' % (fmt, fmt)“. b) Ein vollständiger Spezifikations-String, der alle reellen und vorstellbaren Fälle umfasst, z.B. „' %4e %+4j %4e %+4j %4e %+4j'“, für 3 Spalten. c) Eine Liste mit Spezifikationen, eine pro Spalte – in diesem Fall müssen die reellen und vorstellbaren Teile getrennte Spezifikatoren haben, d.h. „['%3e + %3ej', '(%15e%+15ej)']“ für 2 Spalten.
delimiter	Ein String, der für die Separierung der Spalten genutzt wird.
newline	Ein String, der eine Zeile abschließt statt der Standardzeilenendung.
header	Ein String, der an den Beginn der Datei geschrieben wird.
footer	Ein String, der an das Ende der Datei geschrieben wird.
comments	Ein String, der vor „header“ und „footer“ gestellt wird, um diese als Kommentar zu markieren. Als Default wird hier das Hash-Tag „#“ benutzt.

Die komplette Syntax von `savetxt` sieht folgendermaßen aus:

```
savetxt(fname,
        X,
        fmt='%.18e',
        delimiter=' ',
        newline='\n',
        header='',
        footer='',
        comments='# ')
```

■ 11.2 Textdateien laden mit loadtxt

11.2.1 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)
```

Ausgabe:

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

11.2.2 Spezielle Trenner

```
y = np.loadtxt("test2.txt", delimiter=",")
print(y)
```

Ausgabe:

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

Auch nichts Neues erhalten wir, wenn wir die Datei einlesen, indem wir als Separator einen Smiley verwenden:

```
y = np.loadtxt("test3.txt", delimiter=" :-) ")
print(y)
```

Ausgabe:

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

11.2.3 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=" ")
```

Ausgabe:

```
[[ 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)
```

Ausgabe:

```
[[ 8.  5. -8.]
 [ 0.  8.  8.]
 [-6. -10. -8.]
 [ 1.  2.  2.]]
```

11.2.4 Datenkonvertierung beim Einlesen

Häufig liegen Daten in der einzulesenden Datei in einem Format vor, das wir wandeln müssen. So könnte beispielsweise eine Datei Werte in Fahrenheit-Temperaturen enthalten, die wir aber für unsere Berechnungen in Celsius benötigen. Liest man diese Daten ein, so wie wir es eben getan haben, so müsste man sie in einem Folgeschritt in Celsius wandeln.

Wir zeigen dies an der Datei `temperatures.txt`:

```
Chicago New York Boston Dallas
74.3 69.7 76.1 77.5
80.2 73.8 82.4 84.2
86.3 79.4 89.2 90.2
```

Die erste Zeile enthält die Beschreibung der Spalten. Wir müssen diese beim Einlesen überspringen. Dazu stellen wir den Parameter `skiprows` auf 1.

```
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)
```

Ausgabe:

```
[[23.5          20.94444444 24.5          25.27777778]
 [26.77777778 23.22222222 28.          29.          ]
 [30.16666667 26.33333333 31.77777778 32.33333333]]
```

Das folgende Beispiel empfehlen wir Ihnen, selbst zu versuchen. Es geht darum, die Datei „times_and_temperatures.txt“ einzulesen. Jede Zeile enthält eine Zeitangabe im Format hh:mm:ss und eine zufällige Temperatur 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))
```

Ausgabe:

```
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 Schlüsselwortparameter 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)
```

Ausgabe:

```
[ [ 360.    20.1]
  [ 361.5   16.1]
  [ 363.    16.9]
  ...
  [1375.5   22.5]
  [1377.    11.1]
  [1378.5   15.2]]
```

■ 11.3 tofile

tofile ist eine Funktion, die es ermöglicht, den Inhalt eines Arrays sowohl im Binärformat als auch im Textformat 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 Dateiojekt sein oder ein String mit einem Dateinamen.
sep	Der String „sep“ definiert den Separator, der für die Textausgabe zwischen den Elementen verwendet wird. Übergibt man diesem Parameter einen leeren String, wird eine Binär-Datei geschrieben, genau wie file.write(a.tostring()).
format	Format-String für die Textausgabe. Jeder Eintrag im Array wird so formatiert, indem dieser in den nächsten Python-Typ konvertiert 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)
```

Ausgabe:

```
[((10, 0), 98.25)]
```

11.4 fromfile

fromfile liest Daten, die mit tofile geschrieben wurden. Es ist möglich, Binärdaten 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='')
```

Parameter	Bedeutung
file	'file' kann entweder ein offenes Dateiojekt sein oder ein String mit einem Dateinamen, der gelesen werden soll.
dtype	definiert den Datentyp des Arrays, der aus der Datendatei konstruiert wird. Bei Binärdateien dient es zur Festlegung der Größe und Byte-Reihenfolge der Elemente der Datei.
count	definiert die Anzahl der Elemente, die gelesen werden sollen. -1 bedeutet, dass alle gelesen werden.
sep	Der String „sep“ gibt den Separator an, der verwendet wird, wenn die Datei eine Textdatei ist. Wenn der String leer ist (""), so wird die Datei wie eine Binärdatei behandelt. Ein Leerzeichen (' ') in einem Separator steht für 0 oder mehrere Leerzeichen. Ein Separator, der nur Leerzeichen enthält, muss mindestens einem Leerzeichen entsprechen.

```
fh = open("test4.txt", "rb")

print(np.fromfile(fh, dtype=dt))
```

Ausgabe:

```
[(( 4294967296, 12884901890), 1.06099790e-313)
 (( 30064771078, 38654705672), 2.33419537e-313)
 (( 55834574860, 64424509454), 3.60739285e-313)
 (( 81604378642, 90194313236), 4.88059032e-313)
 ((107374182424, 115964117018), 6.15378780e-313)
 ((133143986206, 141733920800), 7.42698527e-313)
 ((158913789988, 167503724582), 8.70018274e-313)
 ((184683593770, 193273528364), 9.97338022e-313)]
```

```
import numpy as np
import os
```

```
data = np.arange(50, dtype=np.int32)
data.tofile("test4.txt")
```

```
fh = open("test4.txt", "rb")
# 4 * 32 = 128
fh.seek(128, os.SEEK_SET)
```

```
x = np.fromfile(fh, dtype=np.int32)
print(x)
```

Ausgabe:

```
[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 Binärdateien sind nicht plattformunabhängig. Über `tofile` werden keine Informationen zur Byte-Reihenfolge oder Datentypen abgespeichert. Daten können im `.npy`-Format plattformunabhängig gespeichert werden, wenn stattdessen `save` und `load` verwendet werden.

■ 11.5 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)
```

Ausgabe:

```
[32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49]
```

Ausgabe:

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

■ 11.6 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 Textdatei sein. Die Datei kann ebenfalls eine Archivdatei sein. `genfromtxt` kann die Archivformate `gzip` und `bzip2` verarbeiten. Der Typ des Archivs wird durch die Dateierweiterung 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 Datentyp konvertiert. Auf der anderen Seite arbeitet `loadtxt` mit nur einem Schritt, wodurch es schneller ist.

Teil III

Matplotlib

Matplotlib ist eine Bibliothek zum Plotten wie GNUplot. Der Hauptvorteil gegenüber GNUplot ist die Tatsache, dass es sich bei Matplotlib um ein Python-Modul handelt. Aufgrund des wachsenden Interesses an der Programmiersprache Python steigt auch die Popularität von Matplotlib kontinuierlich.

Ein anderer Grund für die Attraktivität von Matplotlib liegt in der Tatsache, dass es als gute Alternative, wenn es ums Plotten geht, für MATLAB angesehen wird, wenn es in Verbindung mit NumPy und SciPy benutzt wird. Während es sich bei MATLAB um kostspielige Closed-

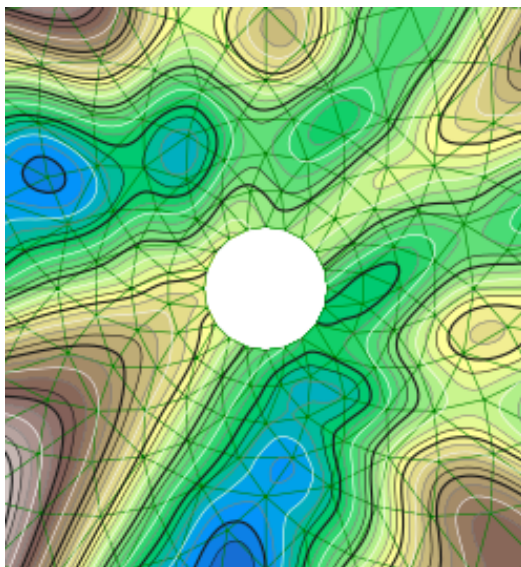


Bild 12.1 Tricontouring

Source-Software handelt, ist die Software von Matplotlib frei, kostenlos und quelloffen. Außerdem kann in Matplotlib objektorientiert programmiert werden. Es kann auch in allgemeinen GUIs wie wxPython, Qt und GTK+ verwendet werden. Mit der „pylab“-Erweiterung wird die Möglichkeit geboten, noch MATLAB-ähnlicher zu programmieren. Davon wird jedoch im Allgemeinen abgeraten, da dies zu einem unsauberen Programmierstil führt, auch wenn es dadurch MATLAB-Nutzern extrem leicht gemacht wird zu wechseln.

Mittels Matplotlib kann man Diagramme und Darstellungen in verschiedenen Formaten erzeugen, die dann in Veröffentlichungen verwendet werden können.

Eine andere Besonderheit besteht in der steilen Lernkurve, was sich darin zeigt, dass die Benutzerinnen und Benutzer sehr schnelle Fortschritte bei der Einarbeitung machen. Auf der offiziellen Webseite steht dazu Folgendes: „Matplotlib versucht, Einfaches einfach und Schweres möglich zu machen. Man kann mit nur wenigen Codezeilen Plots, Histogramme,

Leistungsspektren, Balkendiagramme, Fehlerdiagramme, Streudiagramme (Punktwolken) und so weiter erzeugen.“¹

■ 12.1 Ein erstes Beispiel

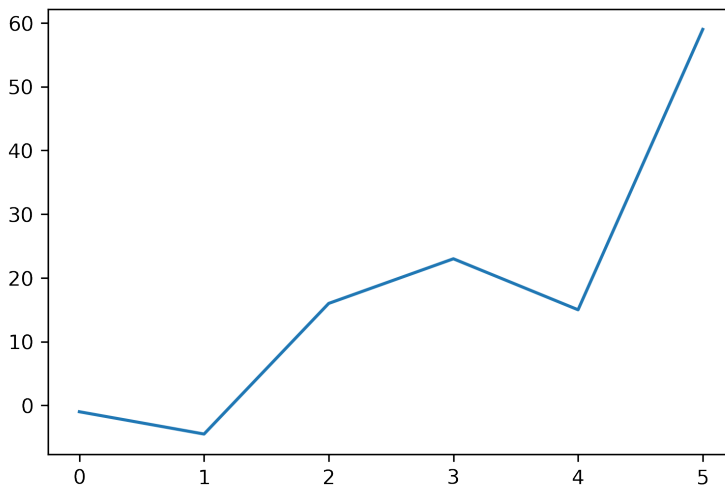
Wir werden mit einem einfachen Graphen beginnen. So einfach, dass es nicht mehr einfacher geht. Ein Graph in Matplotlib ist eine zwei- oder dreidimensionale Zeichnung, die mithilfe von Punkten, Kurven, Balken oder anderem einen Zusammenhang herstellt. Es gibt zwei Achsen: die horizontale x-Achse für die unabhängigen Werte und die vertikale y-Achse für die abhängigen Werte.

Wir werden im Folgenden das Untermodul `pyplot` verwenden. `pyplot` stellt eine prozedurale Schnittstelle zur objektorientierten Plot-Bibliothek von Matplotlib zur Verfügung. Die Kommandos von `pyplot` sind so gewählt, dass sie sowohl in den Namen als auch in ihren Argumenten MATLAB ähnlich sind.

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()
```



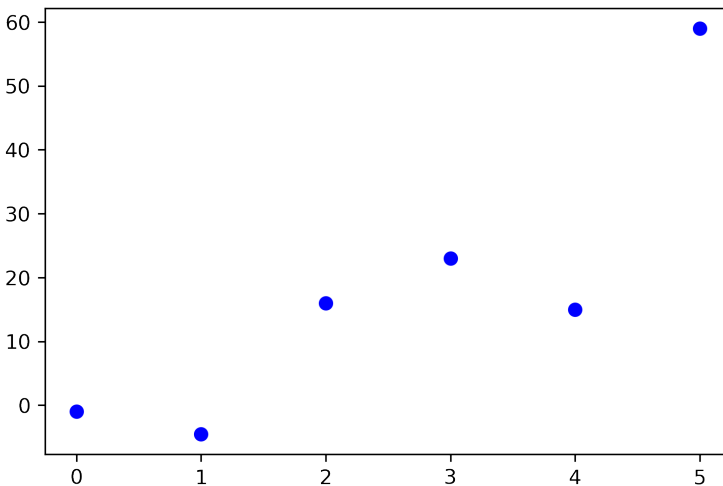
¹ „Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code.“

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 mit übergeben, können wir einen Graphen mit diskreten Werten erzeugen, in unserem Fall mit blauen Vollkreisen. Der Formatstring definiert, wie die diskreten Punkte darzustellen sind:

```
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23, 15, 59], "ob")
plt.show()
```



■ 12.2 Der Formatparameter 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 Default-Wert 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 rechts
'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 Linie

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:

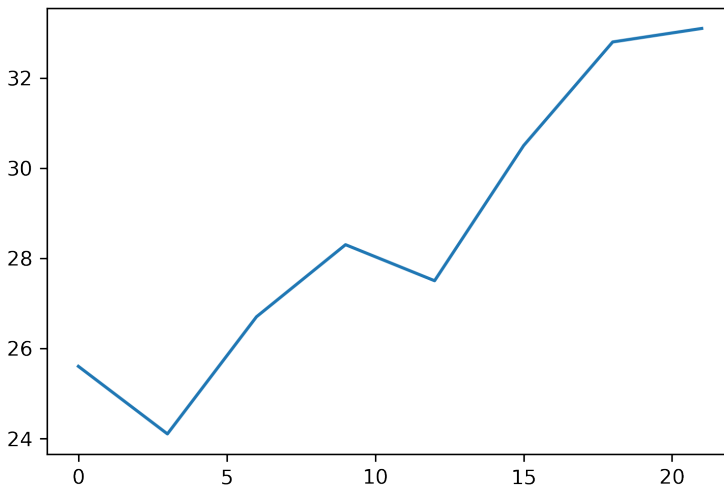
```
import matplotlib.pyplot as plt

# 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()
```

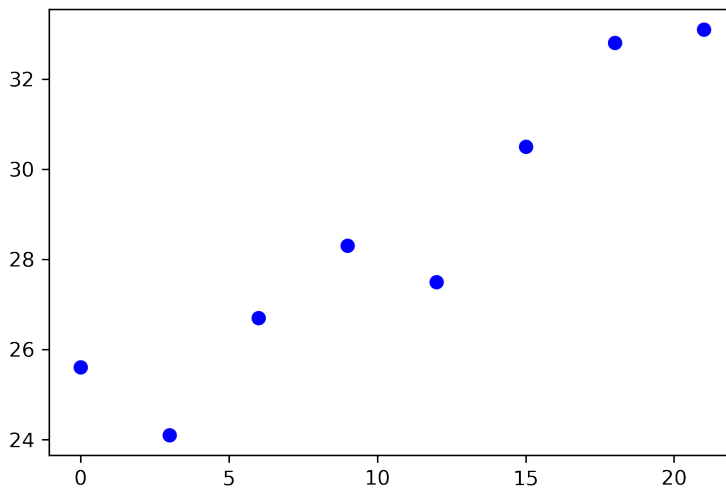
Ausgabe:

[0, 3, 6, 9, 12, 15, 18, 21]



... und das Ganze wieder mit diskreten Werten:

```
plt.plot(days, celsius_values, 'bo')
plt.show()
```

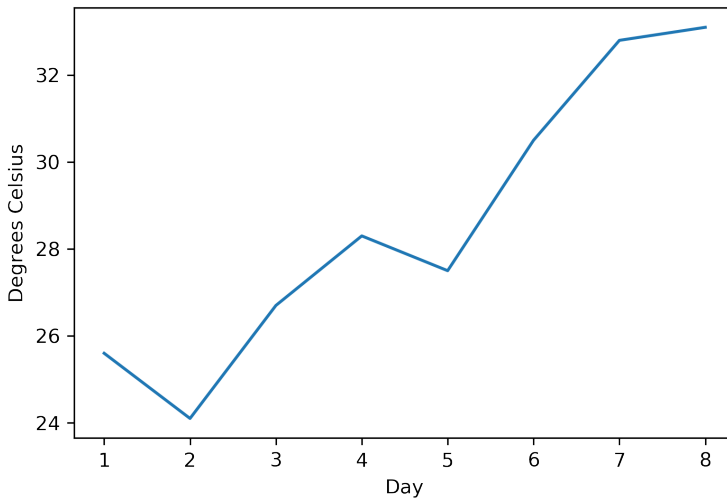
■ 12.3 Bezeichnungen für die Achsen

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:

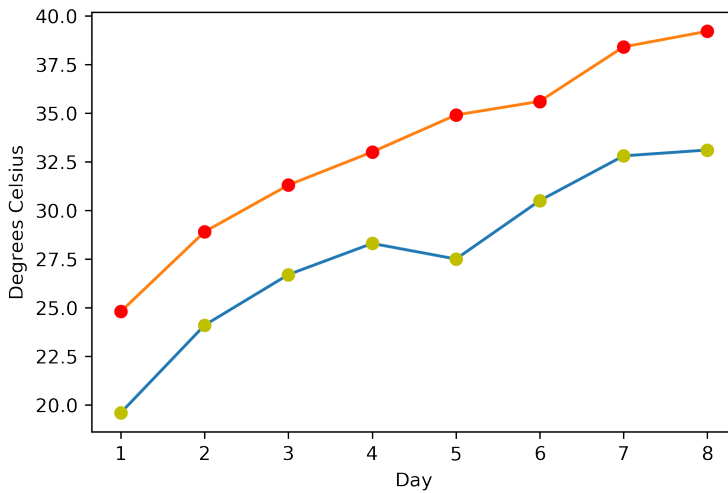
```
import matplotlib.pyplot as plt

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()
```



12.4 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]

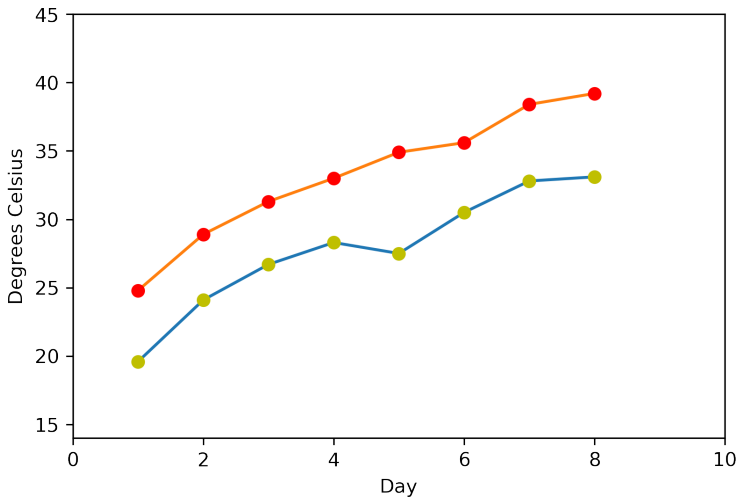
plt.xlabel('Day')
plt.ylabel('Degrees Celsius')

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()
```

Ausgabe:

```
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
```

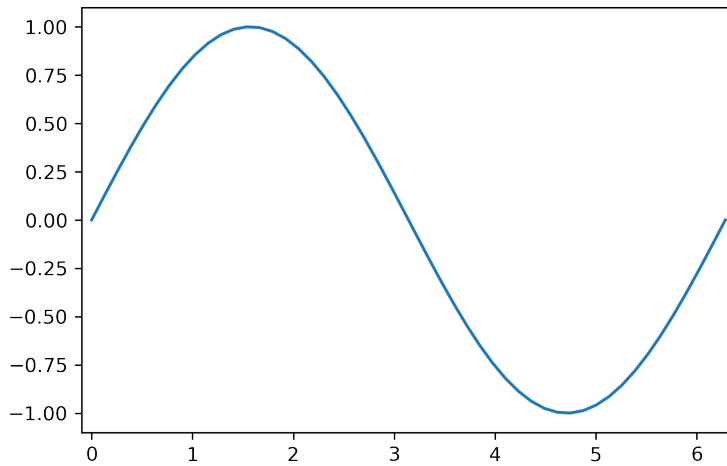


■ 12.5 „linspace“ zur Definition von X-Werten

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()
```

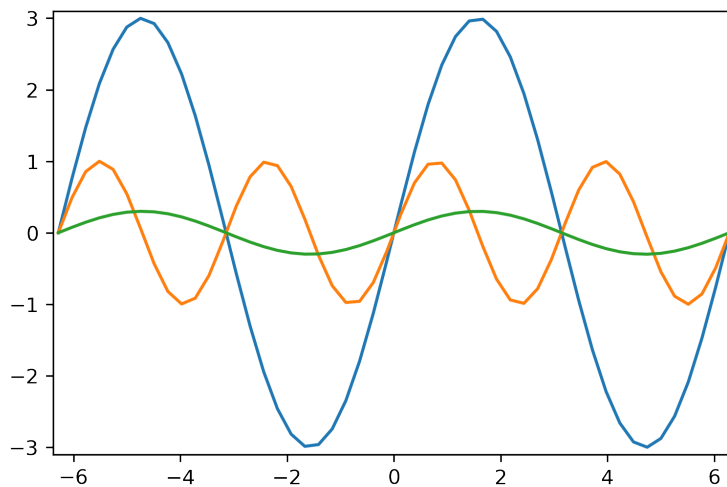


```
import numpy as np
import matplotlib.pyplot as plt

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()
```

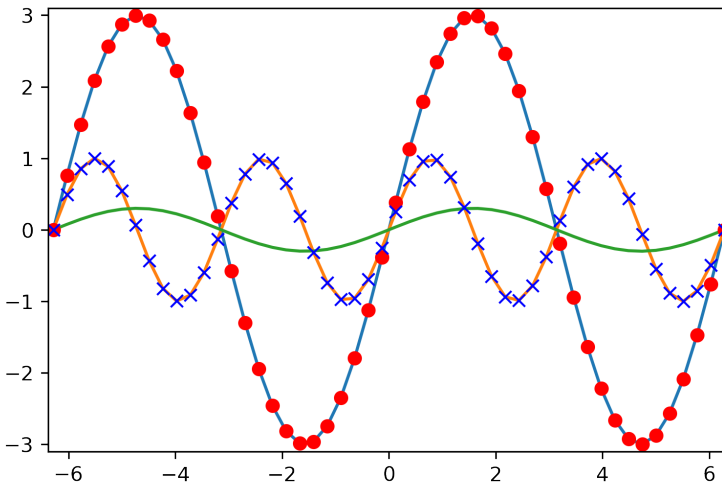


Das nächste Beispiel ist im Prinzip nichts Neues. Wir fügen lediglich zwei weitere Plots mit diskreten Punkten hinzu:

```
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()
```



12.6 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

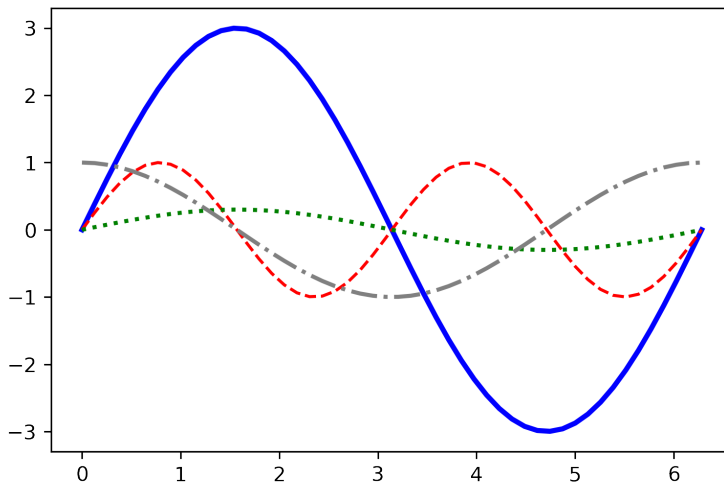
X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
```

```

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()

```



■ 12.7 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 $\sin(2 \cdot X)$:

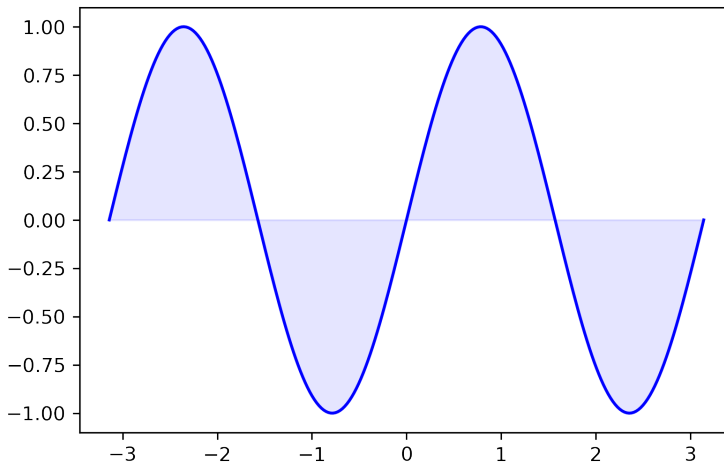
```

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)
```

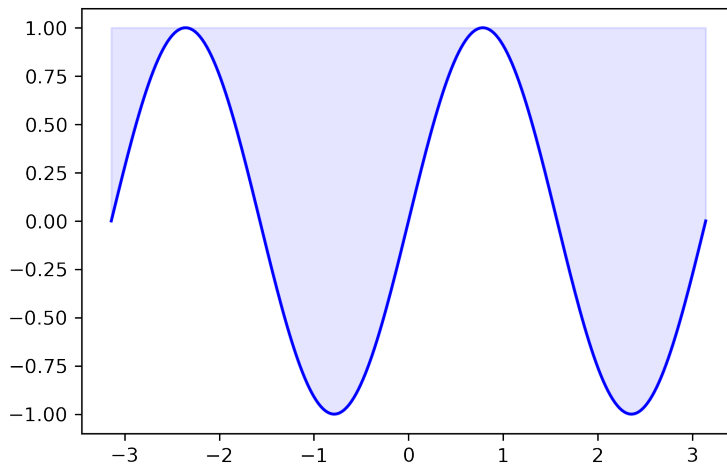
Die Parameter von `fill_between`:

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 <code>where==True</code> 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	Schlüsselwortargumente, 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()
```

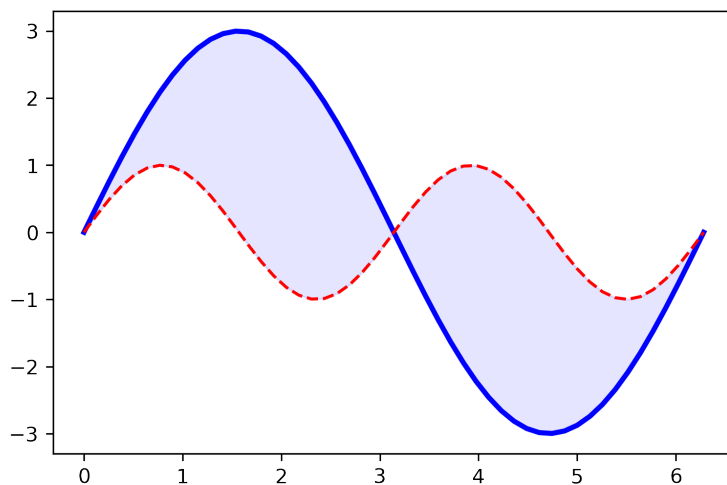



Im nächsten Beispiel füllen wir die Flächen zwischen den Funktionen F1 und F2:

```
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()
```



■ 13.1 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 Achsenmarkierungen verbinden unter Berücksichtigung des Datenbereichs. Im Deutschen sprechen wir von Achsen.

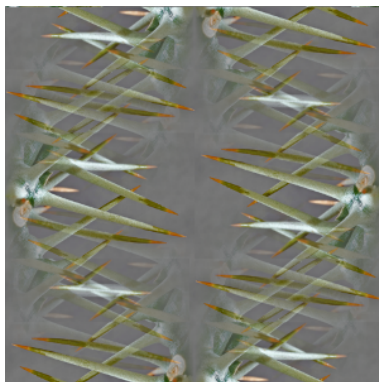


Bild 13.1 Spines, Brustkorb, Stachel und Kakteen

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.¹

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()
```

¹ Ein Plot wird im Englischen auch als „axes“ bezeichnet. Axes bedeutet normalerweise auch „Achsen“.

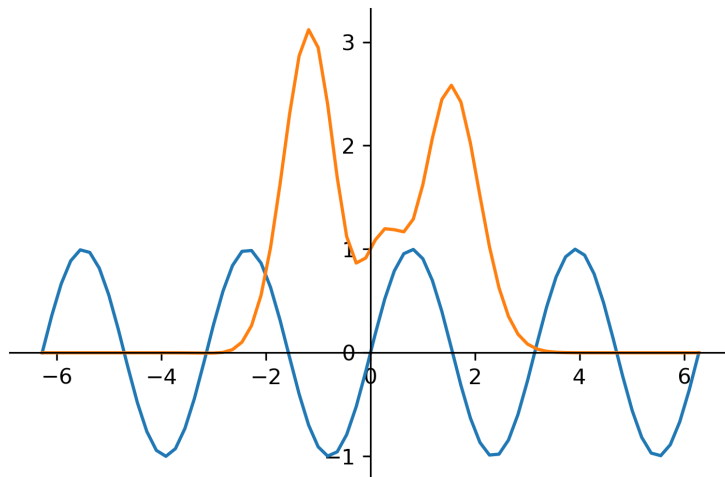
```
# Obere und rechte Achse unsichtbar werden lassen:
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')

# 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`.

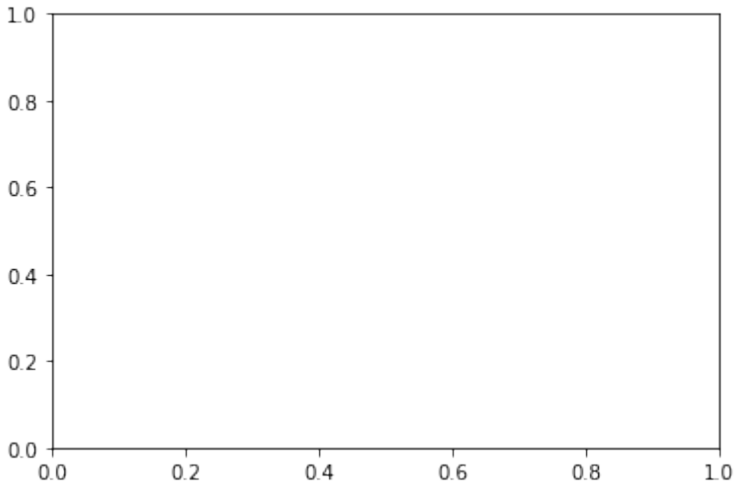
```
ax = plt.gca()

locs, labels = plt.xticks()
print(locs, labels)

locs, labels = plt.yticks()
print(locs, labels)
```

Ausgabe:

```
[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>
```



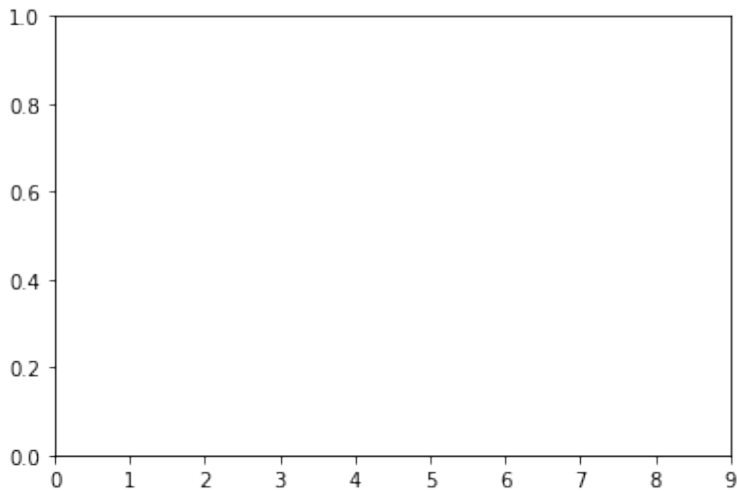
Wie bereits erwähnt, kann `xticks` ebenso dazu verwendet werden, um die Position der Ticks auf der X-Achse zu verändern:

```
plt.xticks( np.arange(10) )

locs, labels = plt.xticks()
print(locs, labels)
```

Ausgabe:

```
[0 1 2 3 4 5 6 7 8 9] <a list of 10 Text xticklabel objects>
```

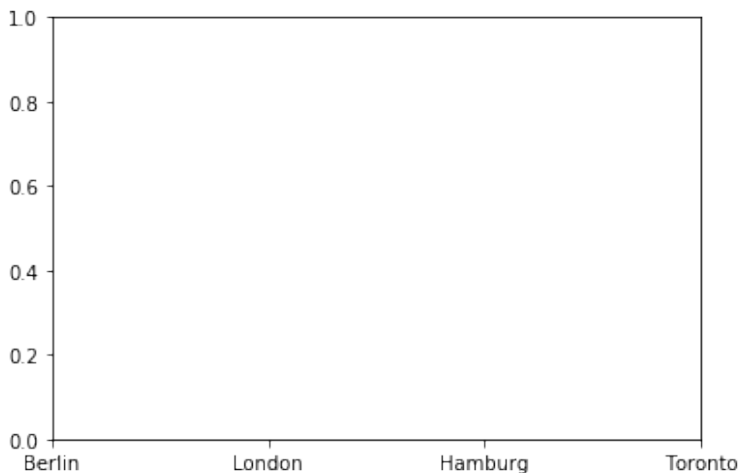


Jetzt verändern wir sowohl die Position als auch die Beschriftung der Ticks auf der X-Achse:

```
plt.xticks( np.arange(4),
            ('Berlin', 'London', 'Hamburg', 'Toronto') )
```

Ausgabe:

```
([<matplotlib.axis.XTick at 0x7f720ca91b00>,
 <matplotlib.axis.XTick at 0x7f720ca41240>,
 <matplotlib.axis.XTick at 0x7f720ca41470>,
 <matplotlib.axis.XTick at 0x7f720cb5c668>],
 <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 π 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')

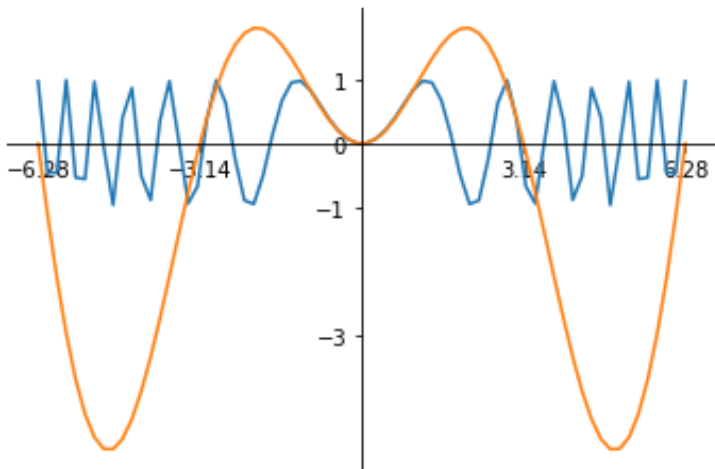
# der untere Spine wird in Position y=0 gebracht:
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))

# 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()

```



■ 13.2 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)

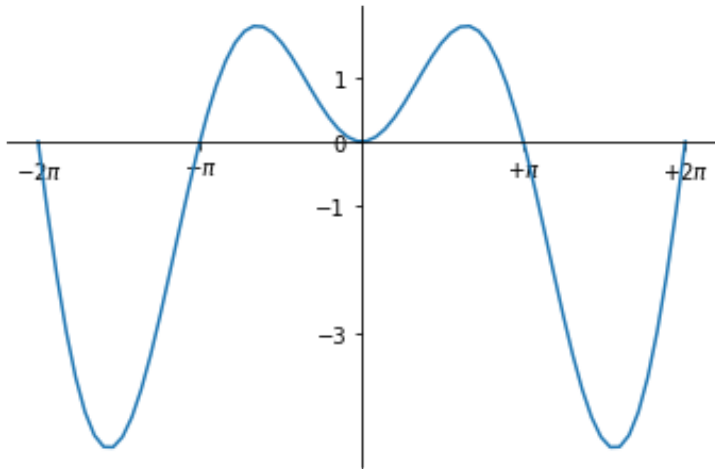
F1 = X * np.sin(X)

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()
```



■ 13.3 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())
```

Ausgabe:

```
<a list of 4 Text xticklabel objects>
for xtick in ax.get_xticklabels():
    print(xtick)
```

Ausgabe:

```
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])

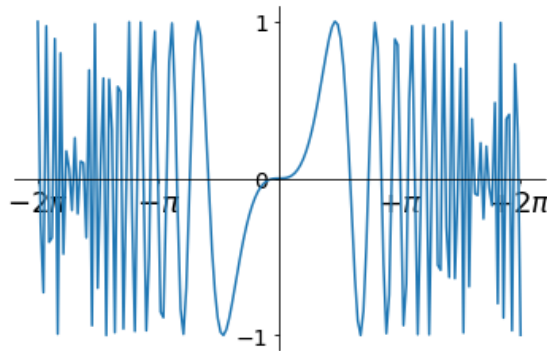
for xtick in ax.get_xticklabels():
    xtick.set_fontsize(18)
    xtick.set_bbox(dict(facecolor='white', edgecolor='None',
                        alpha=0.7 ))

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="$\sin(x)$")
```

Ausgabe:

[<matplotlib.lines.Line2D at 0x7f720cd6d9e8>]



■ 14.1 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()
```

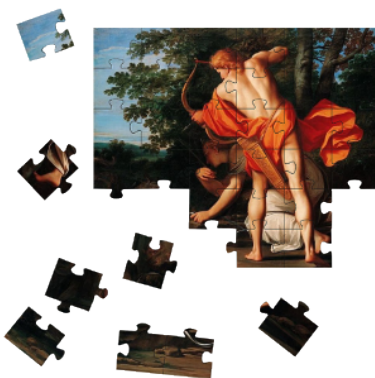
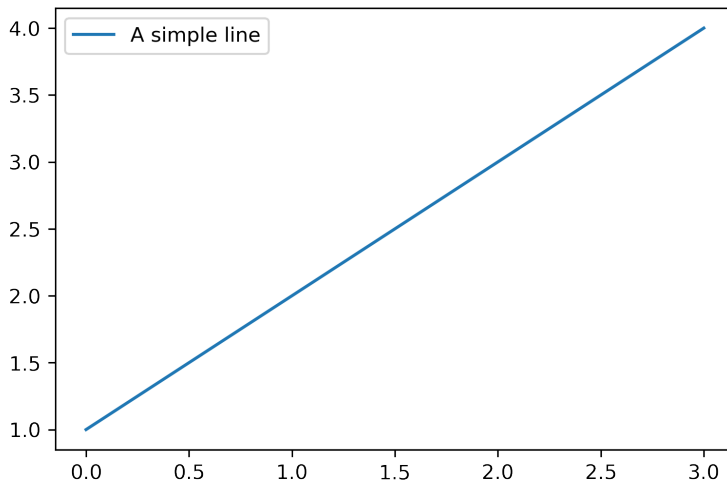


Bild 14.1 Apollo und Diana erschlagen den Python, Marcantonio Bassetti (1648-1729)

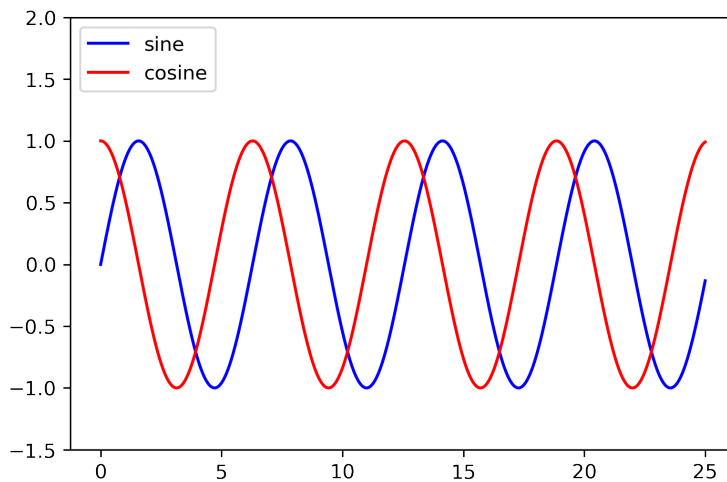


Wenn wir eine Beschriftung hinzufügen, um die Funktion zu zeichnen (plotten), wird der Wert automatisch als Beschriftung im legend-Kommando verwendet. Das einzige Argument, welches die legend-Funktion braucht, ist das Positionsargument `loc`:

```
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()
```



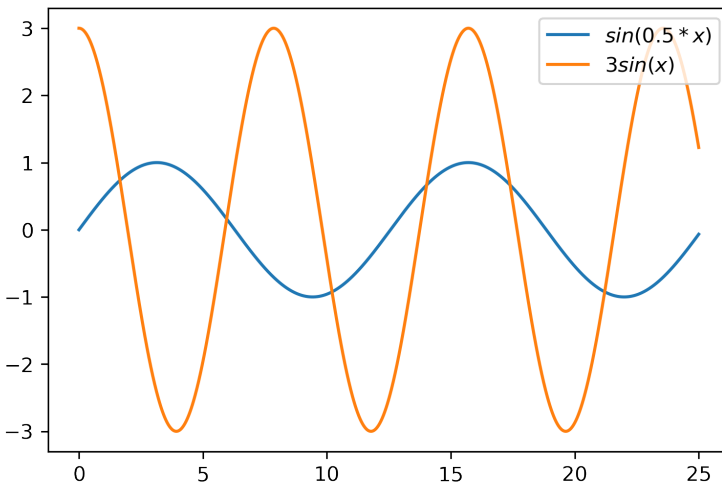
Es folgt nun ein Beispiel mit der Legende in der rechten oberen Ecke:

```
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()
```



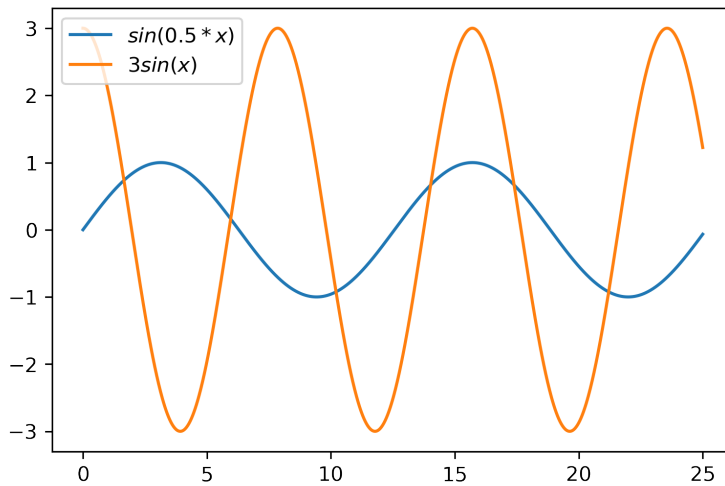
In den meisten Fällen weiß man jedoch nicht, wie das Ergebnis aussehen wird, bevor es nicht ausgegeben wurde. Möglicherweise könnte die Legende einen wichtigen Teil des Plots überdecken. Wenn Sie nicht wissen, wie die Daten aussehen werden, können Sie `best` als Argument für `loc` verwenden. Matplotlib 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()
```



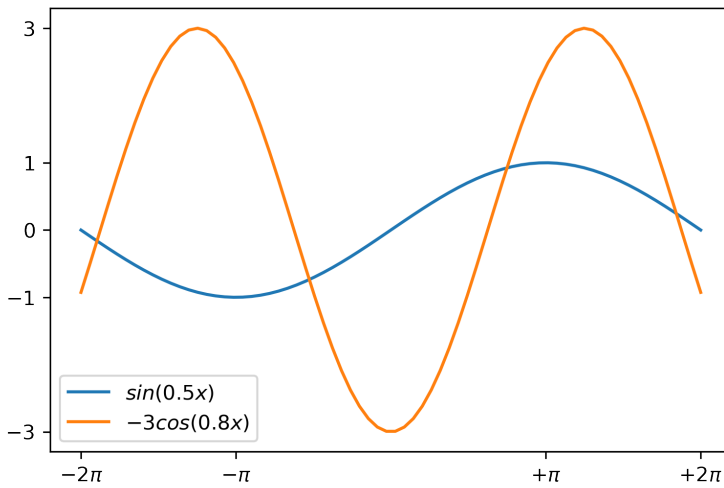
In den folgenden beiden Beispielen kann man sehen, dass `loc='best'` sehr gut funktioniert:

```
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()
```

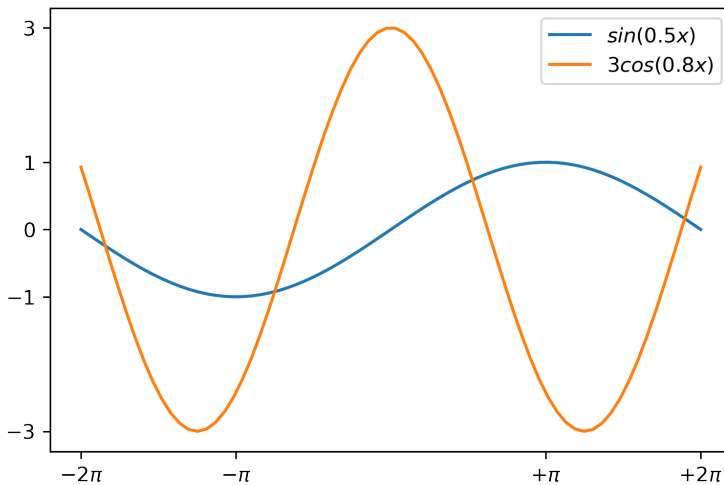


```
import numpy as np
import matplotlib.pyplot as plt

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()
```



■ 14.2 Kommentare

Natürlich hat die Sinus-Funktion „langweilige“ und „interessante“ Werte. Nehmen wir an, dass uns speziell der Wert von $3 \cdot \sin(3 \cdot \frac{\pi}{4})$ interessiert.

```
import numpy as np

print(3 * np.sin(3 * np.pi/4))
```

Ausgabe:

```
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])

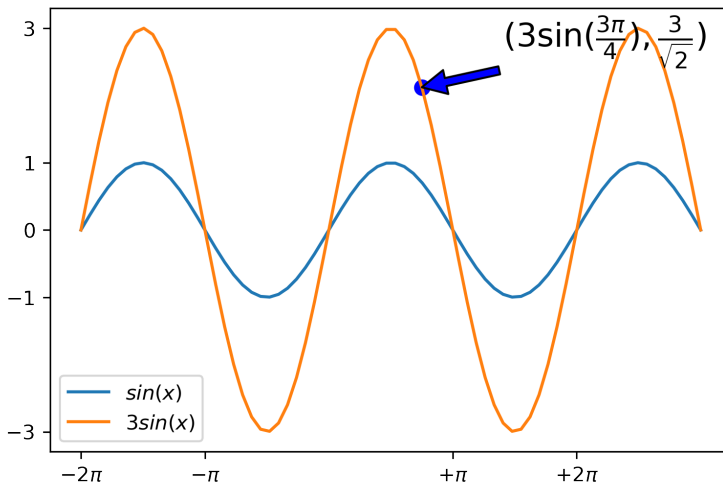
x = 3 * np.pi / 4

# 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
<code>xy</code>	Koordinaten der Pfeilspitze
<code>xytext</code>	Koordinaten der Textposition

Die `xy`- und `xytext`-Positionen sind in unserem Beispiel in den Datenkoordinaten enthalten. Es gibt weitere Koordinatensysteme, aus denen wir wählen können. Das Koordinatensystem von `xy` und `xytext` kann über String-Werte mit `xycoords` und `textcoords` spezifiziert werden. Der Default-Wert ist „data“:

String-Wert	Koordinaten-System
<code>figure points</code>	Punkte von der linken unteren Ecke der Abbildung
<code>figure pixels</code>	Pixel von der linken unteren Ecke der Abbildung
<code>figure fraction</code>	0,0 ist links unten und 1,1 ist rechts oben in der Abbildung
<code>axes points</code>	Punkte von der linken unteren Ecke der Achsen
<code>axes pixels</code>	Pixel von der linken unteren Ecke der Achsen
<code>axes fraction</code>	proportionaler Anteil, 0,0 ist links unten und 1,1 rechts oben
<code>data</code>	Verwende das Achsen-Daten-Koordinatensystem

Zusätzlich können die Eigenschaften des Pfeils spezifiziert werden. Dafür müssen wir ein Dictionary mit Pfeileigenschaften 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 Pfeilspitzenbasis in Punkten
shrink	Verschiebe die Spitze und Basis um ein paar Prozent vom Kommentarpunkt 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])

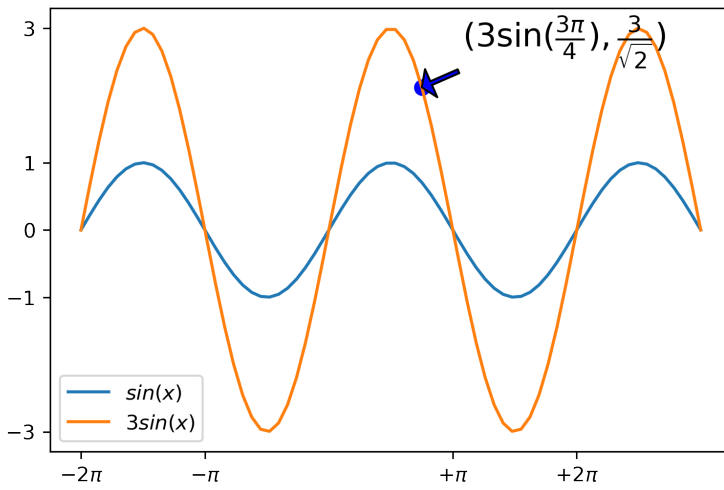
x = 3 * np.pi / 4

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()
```



In zahlreichen Beispielen haben wir bisher gesehen, wie man Diagramme und Graphen erzeugt. Nun wollen wir uns der Frage widmen, wie man mehrere Plots in einem Diagramm unterbringen kann.

Im einfachsten Fall heißt das, dass wir eine Kurve haben und 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.

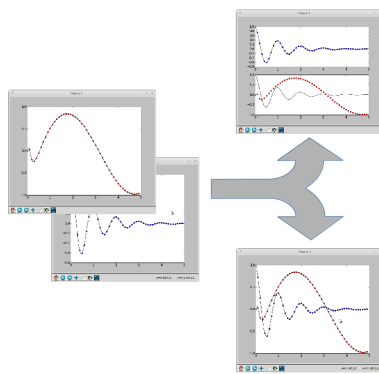


Bild 15.1 Mehrfachplots

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.

■ 15.1 Mehrere Abbildungen und Achsen

Die Funktion `subplot` hat folgende Parameter:

```
subplot(nrows, ncols, plot_number)
```

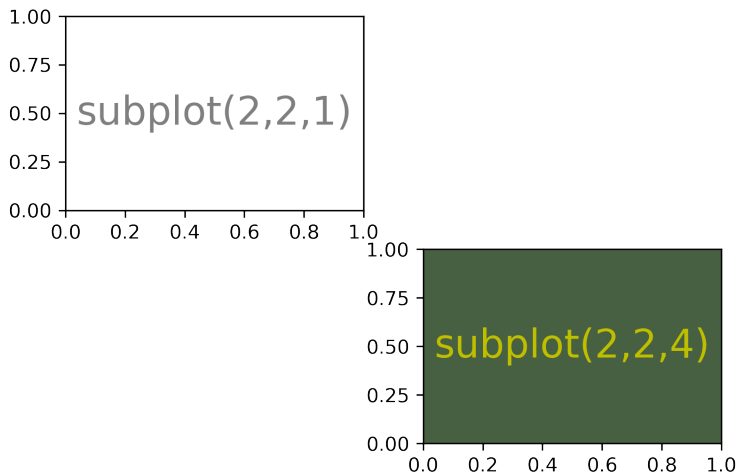
Wenn ein Unterdiagramm auf eine Abbildung angewendet wird, so wird die Abbildung theoretisch aufgeteilt in `nrows * ncols` Unterachsen. Der Parameter `plot_number` bezeichnet den Subplot, den der Funktionsaufruf erstellen muss. `plot_number` kann einen Wert zwischen 1 und dem Maximum von `nrows * ncols` 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=0.5 # float (0.0 transparent 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:

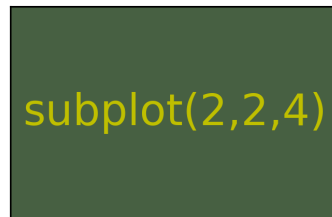
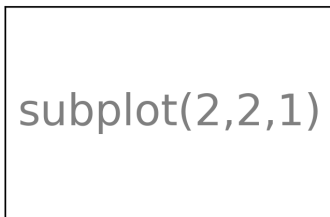
```
plt.xticks(())
plt.yticks(())
```

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) # äquivalent 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.

Die überarbeitete Version des Codes sieht wie folgt aus:

```
import matplotlib.pyplot as plt

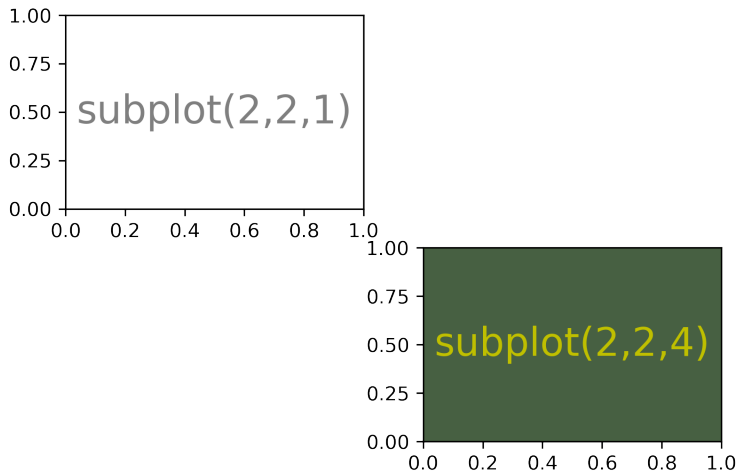
python_course_green = "#476042"
fig = plt.figure(figsize=(6, 4))
sub1 = fig.add_subplot(221) # alternativ: plt.subplot(2, 2, 1)

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', # Abkürzung '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 stattdessen die Methoden `set_xticks()` und `set_yticks()` benutzen.

```

import matplotlib.pyplot as plt

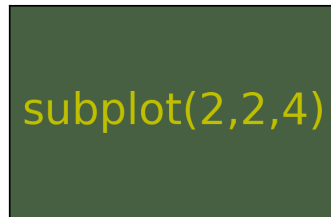
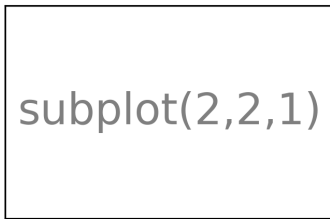
python_course_green = "#476042"
fig = plt.figure(figsize=(6, 4))
sub1 = fig.add_subplot(221) # alternativ: plt.subplot(2, 2, 1)
sub1.set_xticks([])
sub1.set_yticks([])
sub1.text(0.5, # x-Koordinate: 0 ganz links, 1 ganz rechts
0.5, # y-Koordinate: 0 ganz oben, 1 ganz unten
'subplot(2,2,1)', # der Text der ausgegeben wird
horizontalalignment='center', # Abkürzung 'ha'
verticalalignment='center', # Abkürzung '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()

```


subplot(2,2,1)

subplot(2,2,2)

subplot(2,2,3)

subplot(2,2,4)

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)
```

```

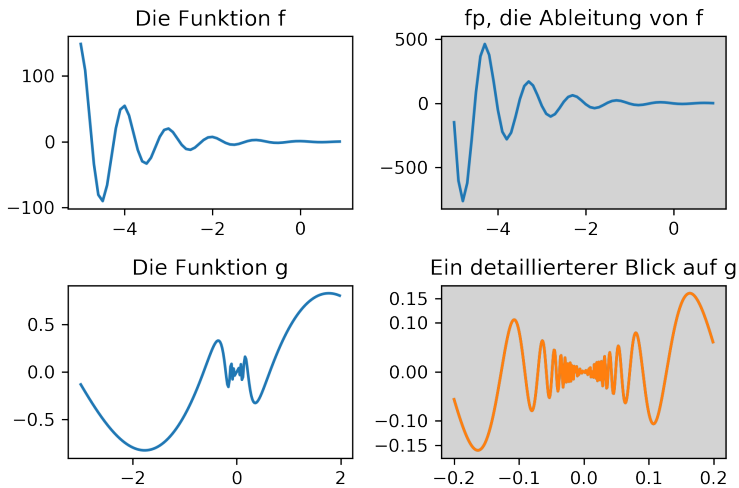
sub3.set_title('Die Funktion g')
sub3.plot(t, g(t))

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()

```



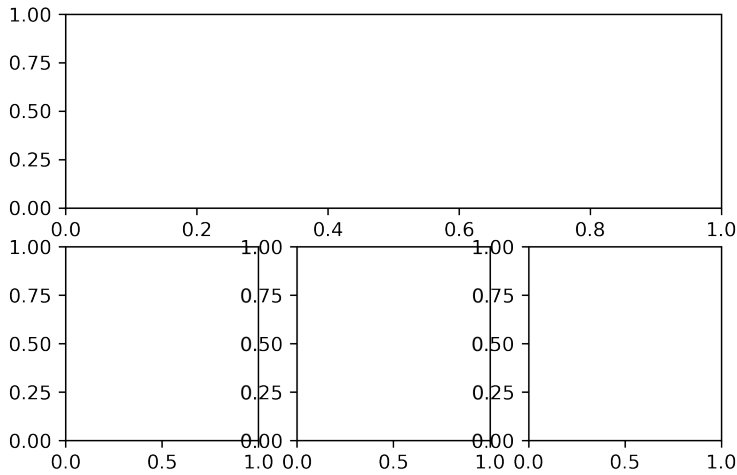
Weiteres Beispiel:

```

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)

```

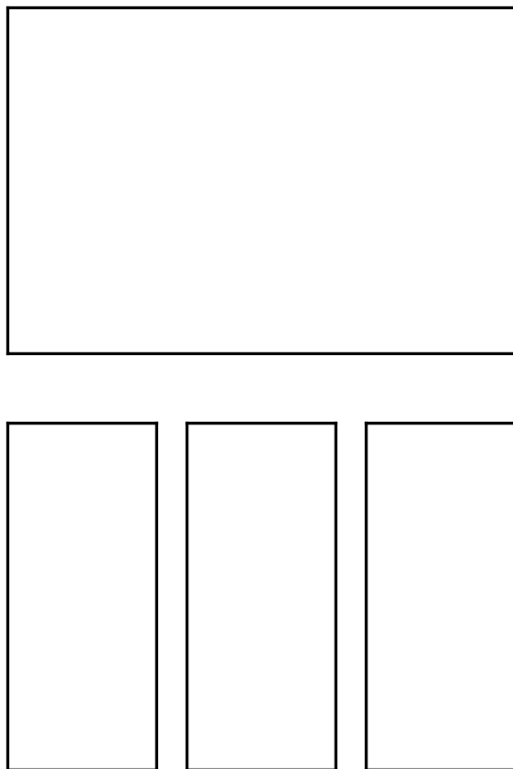


Das folgende Beispiel zeigt nichts Besonderes. Wir entfernen die `xticks` und experimentieren etwas mit der Größe der Abbildung und der Unterplots. Dafür führen wir nun den Schlüsselwortparameter `figsize` für „figure“ ein. `figsize` erwartet ein Tupel mit einem Wert für Breite und Höhe in Inches. Außerdem nutzen wir die Funktion `subplot_adjust` mit deren Schlüsselwortparametern `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([])
```



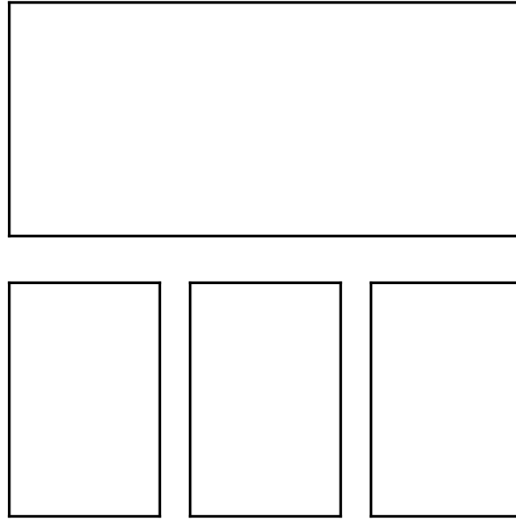
Alternative Lösung:

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([])
```



■ 15.2 Unterdiagramm mit gridspec

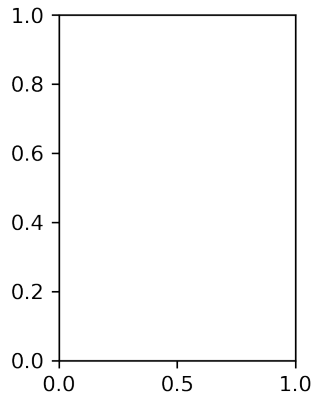
`matplotlib.gridspec` beinhaltet die Klasse `GridSpec`. Diese kann als Alternative zu `subplot` verwendet werden, um die Geometrie der Unterplots zu spezifizieren. Die Grundidee hinter `GridSpec` ist ein „Gitter“, englisch „grid“. Daher kommt auch der Name „grid-spec“. Ein Gitter wird erstellt, indem die Anzahl der benötigten Zeilen und Spalten angegeben wird. Anschließend definiert man innerhalb 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()
```

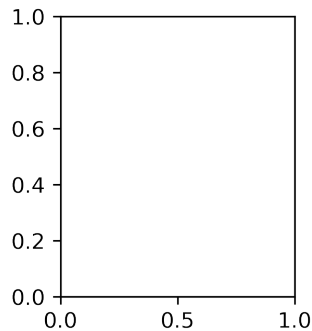


Wir könnten noch einige der Parameter aus GridSpec benutzen, z.B. können wir definieren, dass der Graph erst 20 % von unten und 15 % von links der verfügbaren Abbildungsfläche beginnen soll:

```
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 Gitterdesign:

```
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt

plt.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 Gitterspezifikation aus dem vorigen Beispiel verwenden wir nun, um sie mit Funktionsgraphen zu bestücken:

```

import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt
import numpy as np

pl.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)

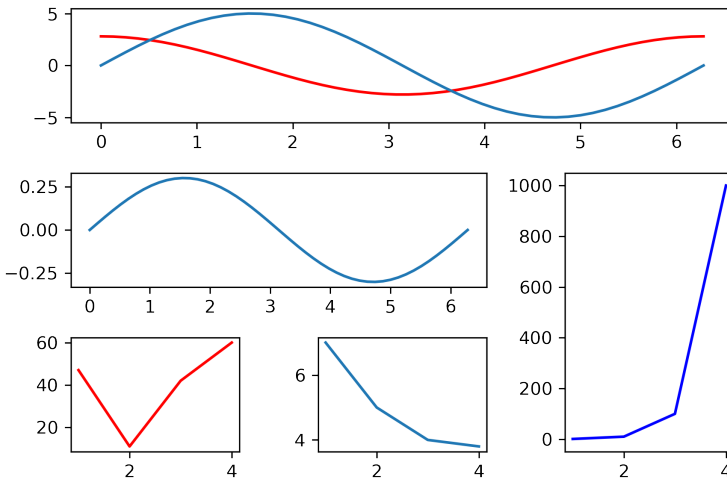
axes_3 = plt.subplot(G[1:, -1])
axes_3.plot([1,2,3,4], [1,10,100,1000], 'b-')

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()

```



15.3 Arbeiten mit Objekten

Matplotlib ist komplett objektorientiert design und programmiert – wie Python selbst. Die bis jetzt gezeigten Beispiele 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-Objekts. 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

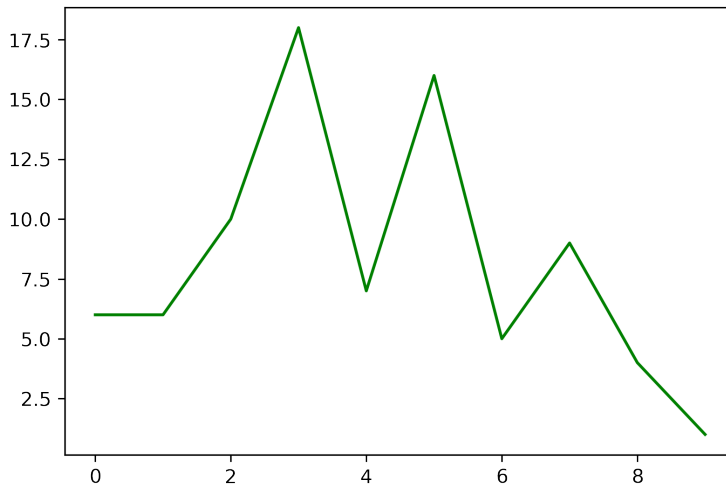
fig = plt.figure()

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)
```



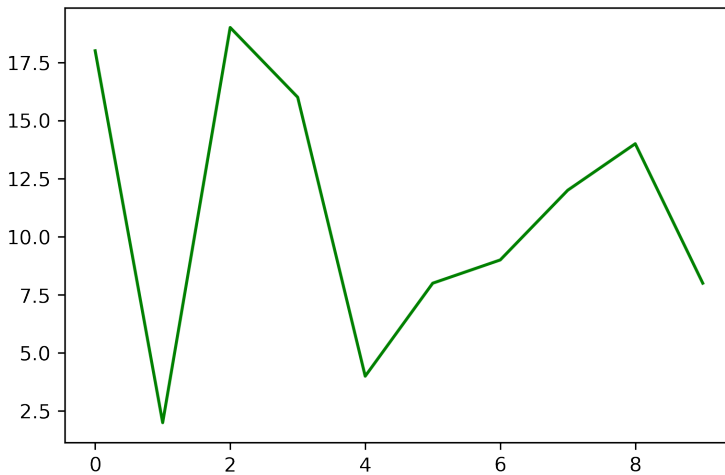
Ohne explizite Instanzen zu verwenden, sieht der Code folgendermaßen aus:

```
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()
```



■ 15.4 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]

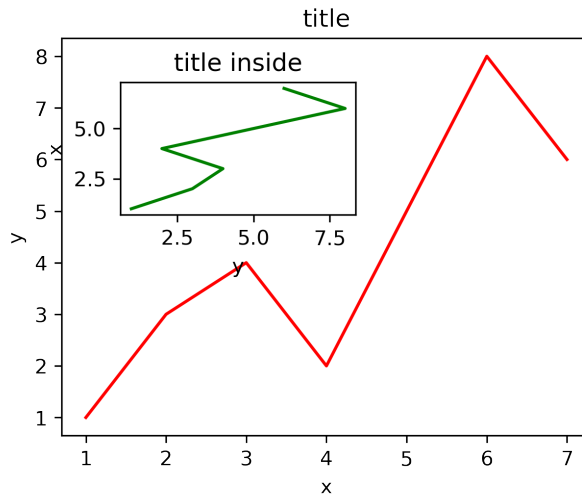
axes1 = fig.add_axes([0.1, 0.1, 0.9, 0.9]) # main axes
axes2 = fig.add_axes([0.2, 0.6, 0.4, 0.3]) # inset axes

# 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()
```



15.5 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-Objekts. Mit `axis('tight')` erstellen wir automatisch „eng anliegende“ Plot-Bereiche:

```
import numpy as np
import matplotlib.pyplot as plt

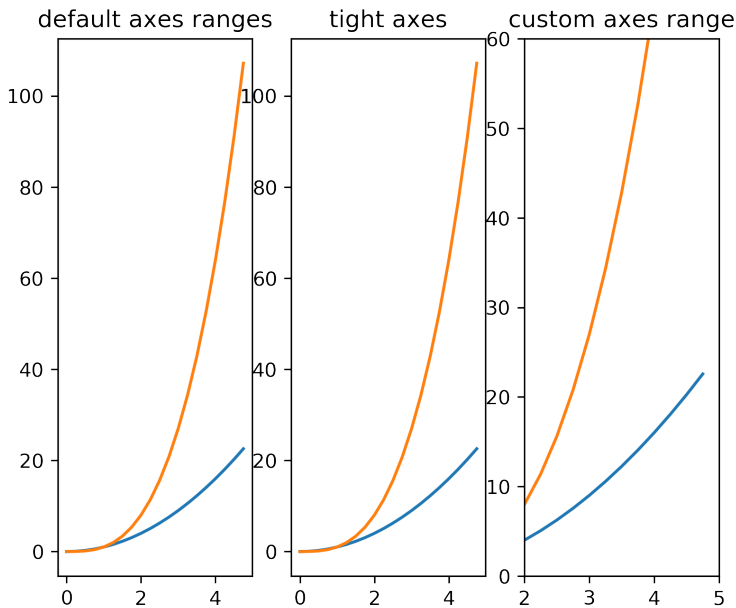
fig, axes = plt.subplots(1, 3, figsize=(6, 5))

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");
```



15.6 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 Transformationssystems in Matplotlib. Jede der Achsen-Darstellungen 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

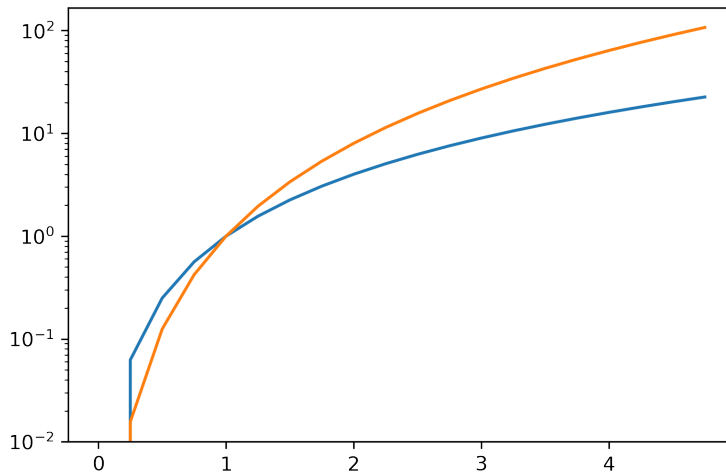
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

x = np.arange(0, 5, 0.25)

ax.plot(x, x**2, x, x**3)

ax.set_yscale("log")

plt.show()
```



■ 15.7 Sekundäre Y-Achse

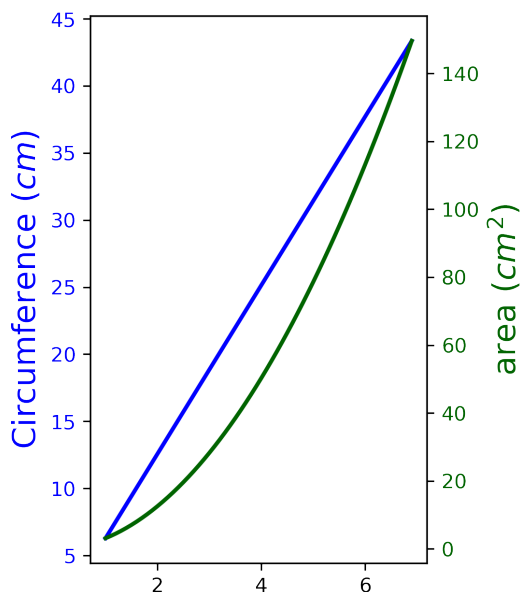
Bisher haben wir schon öfters zwei verschiedene Graphen in einem Plot dargestellt. Allerdings war bisher immer der Wertebereich, also die Einheit auf der y-Achse, der gleiche gewesen. Im Folgenden haben wir zwei Wertebereiche, einmal „cm“ und einmal „qcm“. Wir erzeugen die zweite y-Achse, die man üblicherweise 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")
```



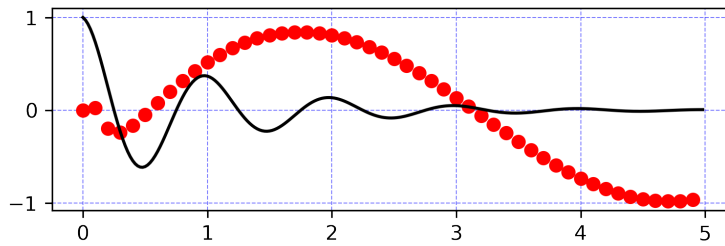
Die folgenden Themen stehen nicht im direkten Zusammenhang mit Unterplots. Trotzdem wollen wir sie hier erwähnen, um die Einführung der Basismöglichkeiten von Matplotlib abzurunden. Das erste zeigt, wie Gitterlinien definiert werden. Das zweite, sehr wichtige Thema befasst sich mit der Speicherung von Plots in Bilddateien.

15.8 Gitterlinien

```
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()
```



■ 15.9 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.

■ 15.10 Aufgaben



1. Aufgabe:

Wie kann man ein Unterplot-Design eines 3x2-Gitters erstellen, bei dem die erste Spalte verbunden ist?



2. Aufgabe:

Erstelle ein Unterplot-Design für das folgende Design:



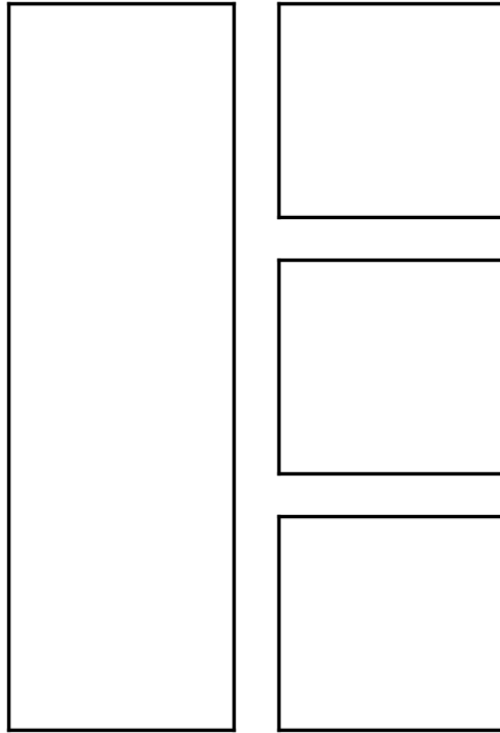
3. Aufgabe:

Erzeuge nun das gleiche Design mit GridSpec.

■ 15.11 Lösungen

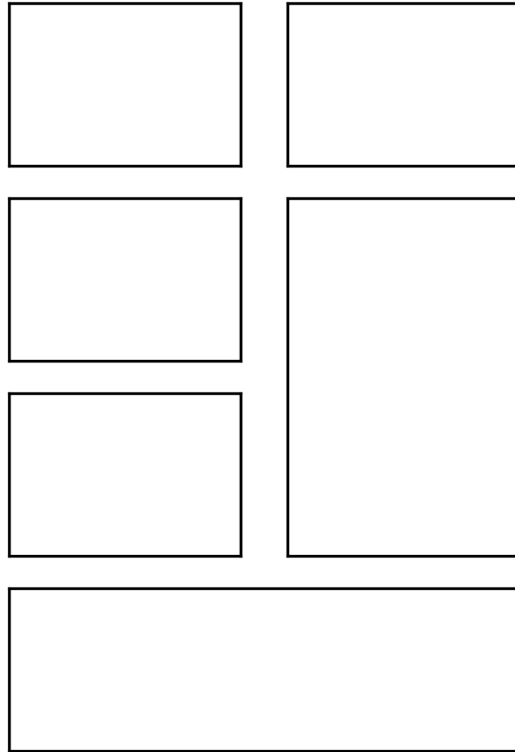
Lösung zur 1. Aufgabe:

```
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([])
```


**Lösung zur 2. Aufgabe:**

```
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()
```



Lösung zur 3. Aufgabe:

```
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt

plt.figure(figsize=(2, 3))
G = gridspec.GridSpec(4, 2)

fontsize = 12
alpha = 0.5
axes_1 = plt.subplot(G[0, 0])
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'area 1', ha='center',
         va='center', size=fontsize, alpha=alpha)

axes_2 = plt.subplot(G[0, 1])
plt.xticks(())
plt.yticks(())
plt.text(0.5, 0.5, 'area 2', ha='center',
         va='center', size=fontsize, alpha=alpha)

axes_3 = plt.subplot(G[1, 0])
plt.xticks(())
plt.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()

```



Eine Konturlinie oder Isolinie einer Funktion aus zwei Variablen ist eine Kurve entlang des konstanten Werts 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.

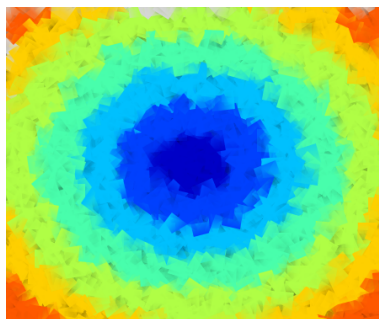



Bild 16.1 Konturplot künstlerisch verfremdet

■ 16.1 Erstellen eines Maschengitters

Ein Maschengitter (Meshgrid) ist ein rechteckiges Gitter (Datengitter), das aus zwei eindimensionalen Arrays erzeugt wird, d.h. den x - und den y -Werten. Im weiteren Verlauf dieses Kapitels werden wir nochmals auf die Funktion `meshgrid` und ihre Alternativen zurückkommen.

```
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.]]
```

corresponds to the following coordinate points:

```
(-3,-3) (0,-3) (3, -3)
(-3,-1) (0,-1) (3, -1)
(-3, 1) (0, 1) (3, 1)
(-3, 3) (0, 3) (3, 3)
```

```
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)
```

Ausgabe:

```
[-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.]]
```

■ 16.2 Berechnung der Werte

Nun berechnen wir die Funktionswerte zu den Wertepaaren des Maschengitters:

```
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]]
```

```
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)
```

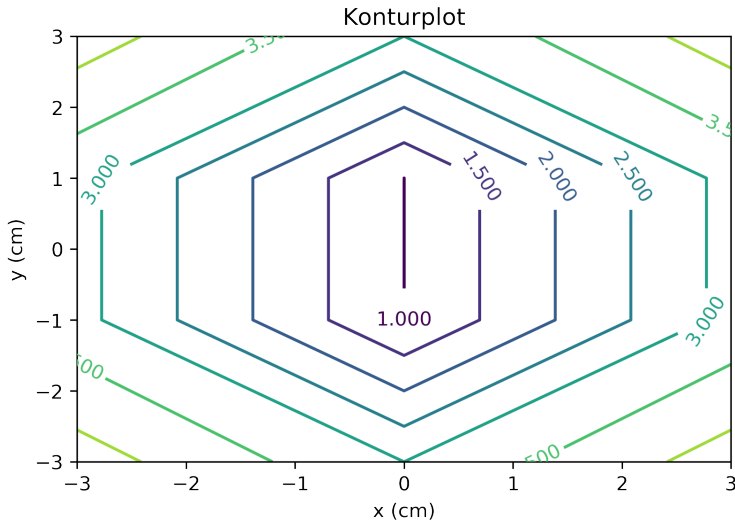
Ausgabe:

```
[[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()
```



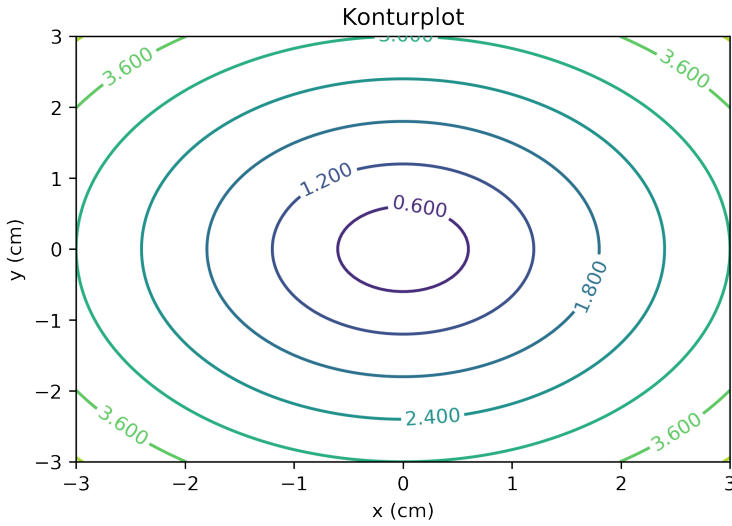
Unser Konturplot sieht sehr kantig aus, weil unser Maschengitter nur aus 12 Punkten besteht. Im Folgenden verfeinern wir unser Maschengitter:

```
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()
```

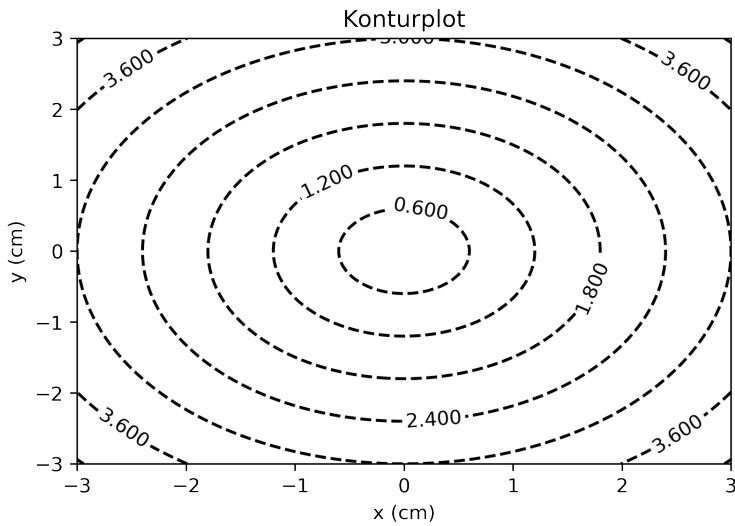


16.3 Linienstil und Farben anpassen

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()
```

■ 16.4 Gefüllte Konturen

Wir können auch den Zwischenraum zwischen den Konturlinien einfärben:

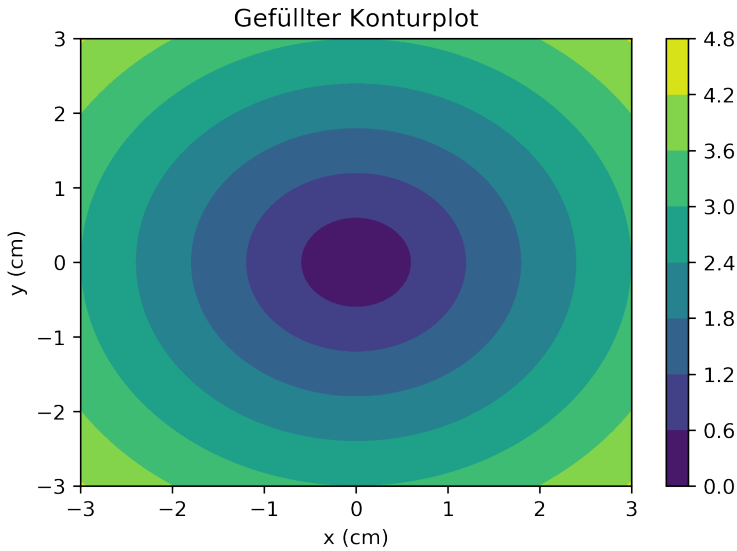
```
import numpy as np
import matplotlib.pyplot as plt

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()
```



■ 16.5 Individuelle Farben

Die Farben für die Flächen können wir natürlich auch selbst bestimmen, wie wir im folgenden Beispiel sehen:

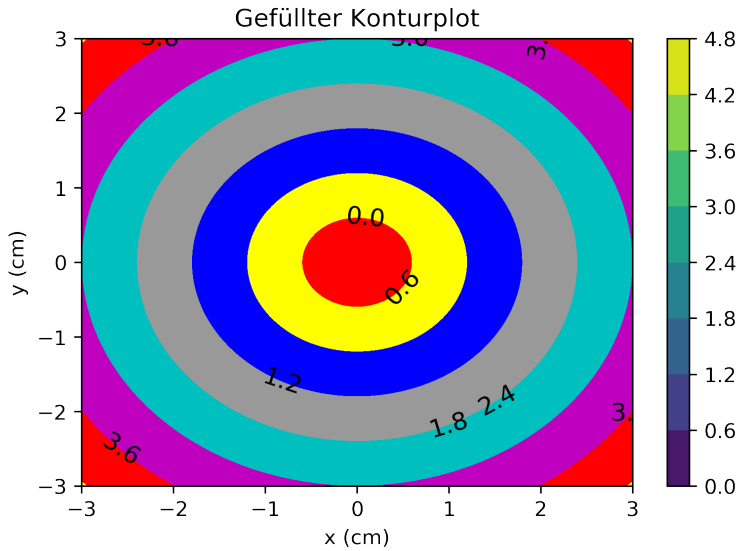
```
import numpy as np
import matplotlib.pyplot as plt

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.contourf(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)

plt.title('Gefüllter Konturplot')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.show()
```



■ 16.6 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.

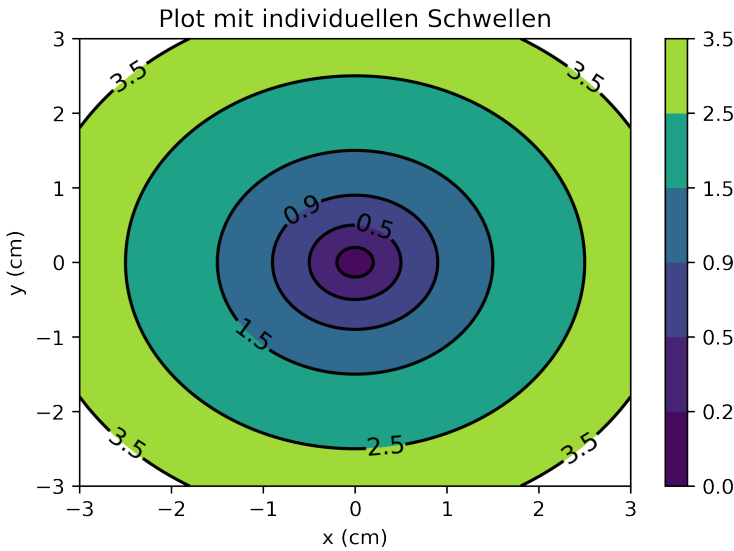
```
import numpy as np
import matplotlib.pyplot as plt

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()
```



■ 16.7 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`

16.7.1 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)
```

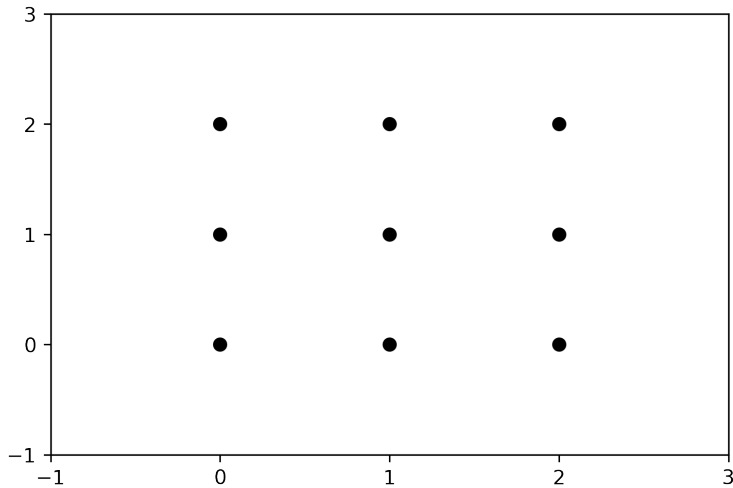
Ausgabe:

```
[[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 $0 \leq i \leq 2$ und $0 \leq j \leq 2$. 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()
```



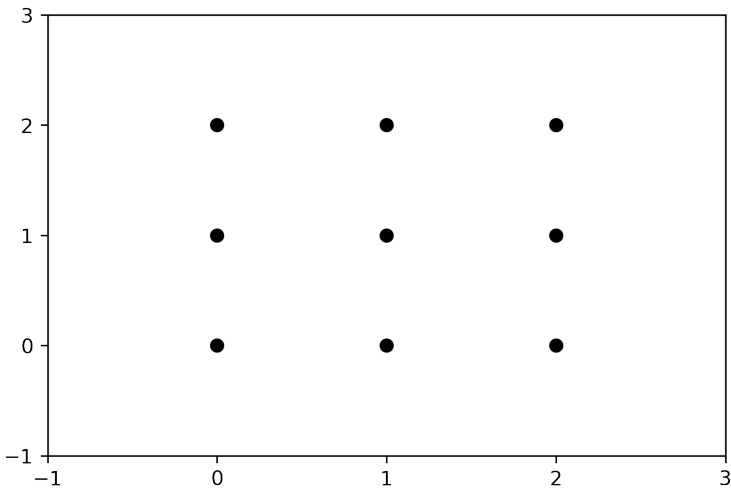
Dies war der umständliche direkte Weg, und mit `meshgrid` geht es deutlich schneller und leichter:

```
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()
```



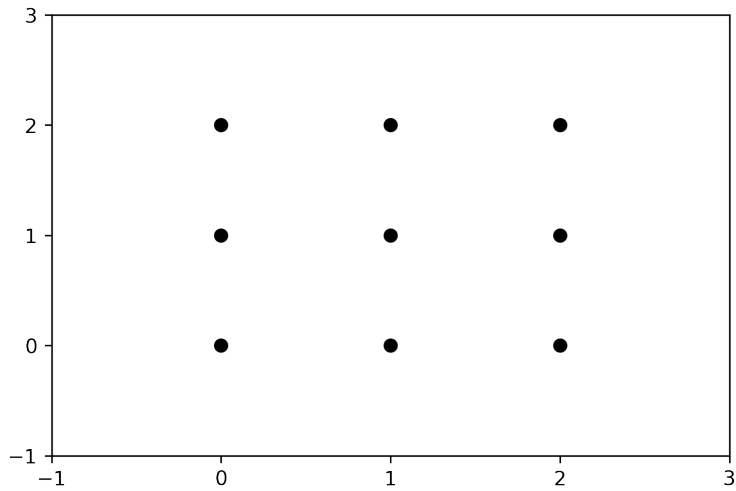
Im gewissen Sinne ist `meshgrid` überflüssig, da man das gleiche Resultat auch mittels Broadcasting erreichen kann:

```
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()
```



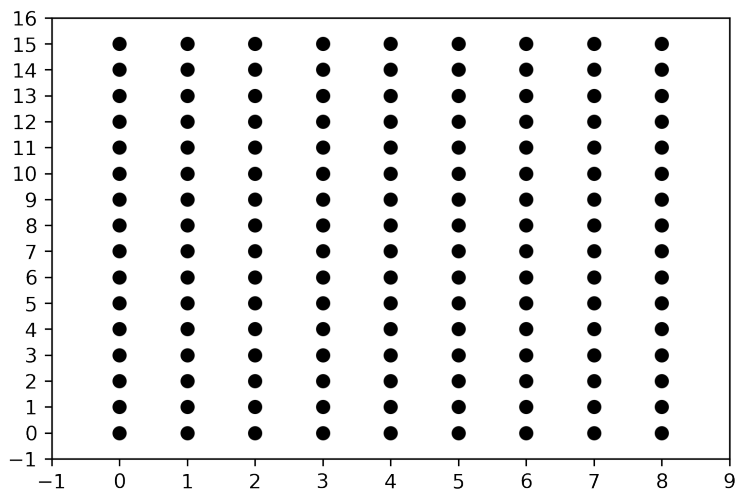
Wir hatten mit `meshgrid` eine quadratische Gitterstruktur erzeugt. Selbstverständlich können wir auch beliebige rechteckige Strukturen erzeugen:

```
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()
```



16.7.2 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 dasselbe 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)
```

Ausgabe:

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

16.7.3 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.

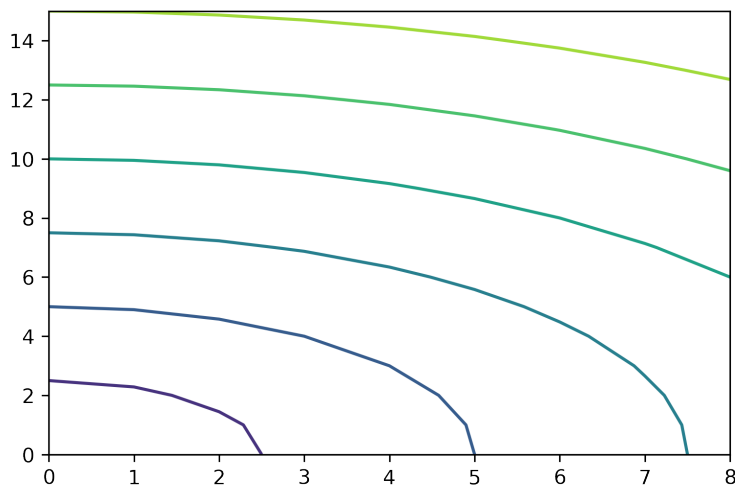
```
import numpy as np

n = 5
X_ogrid, Y_ogrid = np.ogrid[0:n, 0:n]
print(X_ogrid)
print(Y_ogrid)
```

Ausgabe:

```
[[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)
```

16.8 Aufgaben



1. Aufgabe:

Erzeuge einen Plot der Funktion

$$z = \sin(x^3) + \cos(y^2)$$



2. Aufgabe:

Bei dieser Aufgabe handelt es sich um eine herzige Angelegenheit. Einfach für die folgende Funktion einen Konturplot erzeugen, und es wird sich zeigen, warum:

$$x^2 + (y - (x^2)^{\frac{1}{5}})^2$$



3. Aufgabe:

Fülle den obigen Konturplot mit `contourf` und experimentiere mit dem Parameter `cmaps`. Mögliche Werte sind zum Beispiel

```
'Greys', 'Purples', 'Blues', 'Greens', 'Oranges',
'Reds', 'YlOrBr', 'YlOrRd', 'OrRd', 'PuRd', 'RdPu',
'BuPu', 'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn', 'BuGn',
'YlGn', 'viridis', 'plasma', 'inferno', 'magma',
'cividis', 'binary', 'gist_yarg', 'gist_gray', 'gray',
'bone', 'pink', 'spring', 'summer', 'autumn', 'winter',
'cool', 'Wistia', 'hot', 'afmhot', 'gist_heat', 'copper'
```

■ 16.9 Lösungen

Lösung zur 1. Aufgabe:

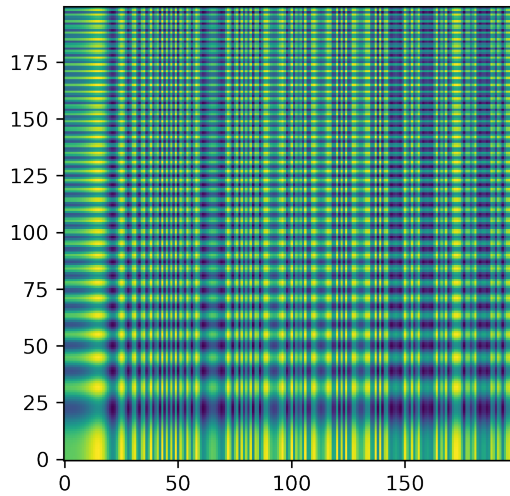
```
# progr4book

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()
```

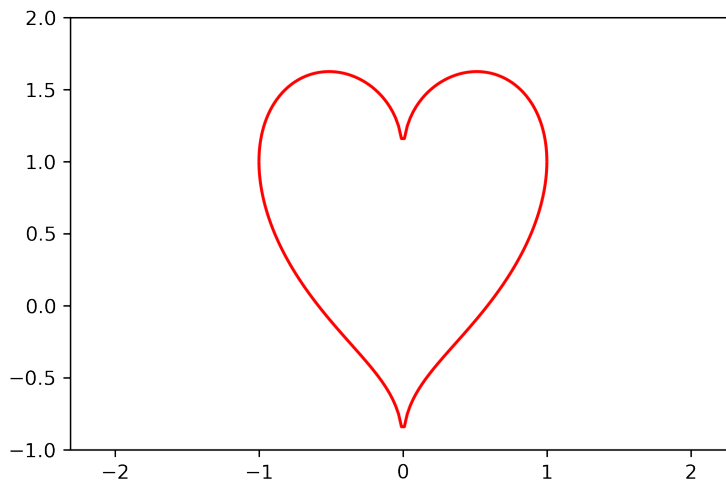


Lösung zur 2. Aufgabe:

```
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()
```



```
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()
```



Balken-, Säulen- diagramme und Histogramme

Gegenstand dieses Kapitels sind Balken- und Säulendiagramme sowie Histogramme. Sie begegnen uns tagtäglich unter anderem in den Medien. Sie bieten uns quantitativ basierte Informationen zu unterschiedlichsten Themen. Balken- und Säulendiagramme zeigen uns anschaulich, wo unsere Spitzenpolitikerinnen und -politiker gerade in der Gunst oder Missgunst der Wählerschaft stehen. Sie informieren auch über Konsequenzen von bestimmtem Verhalten: Rauchen oder nicht rauchen. Vor- und Nachteile von diversen Tätigkeiten. Einkommensverteilungen und so weiter. Einerseits dienen sie uns als Informationsquelle, um unser eigenes Denken und Handeln im statistischen Vergleich mit anderen zu sehen, andererseits verändern sie auch – dadurch, dass wir sie wahrnehmen – unser Denken und Handeln in vielen Fällen.

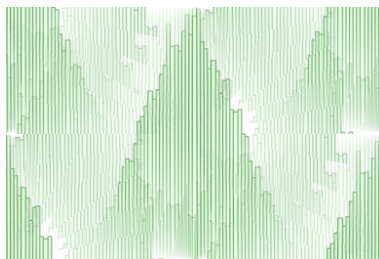


Bild 17.1 Histogramme künstlerisch verfremdet

In erster Linie interessiert uns jedoch in diesem Kapitel, wie man sie erzeugt.

Wir werden mit Histogrammen beginnen. 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 Frequenzen bzw. Häufigkeiten entsprechen.

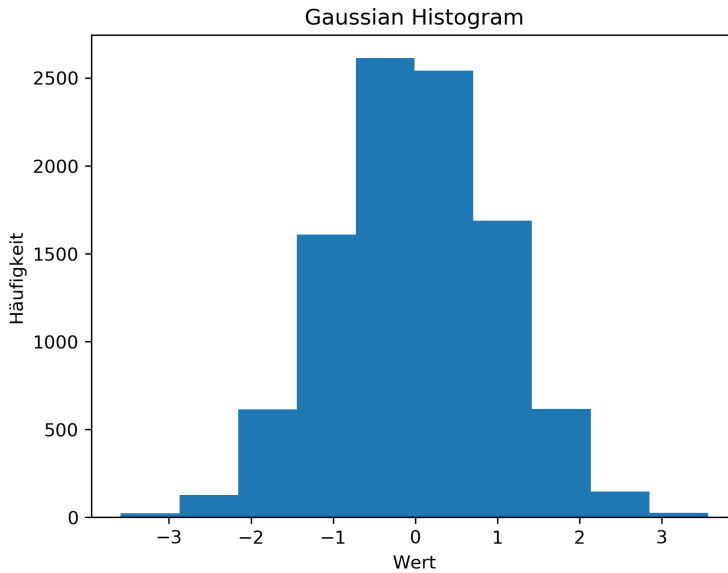
■ 17.1 Histogramme

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 generieren:

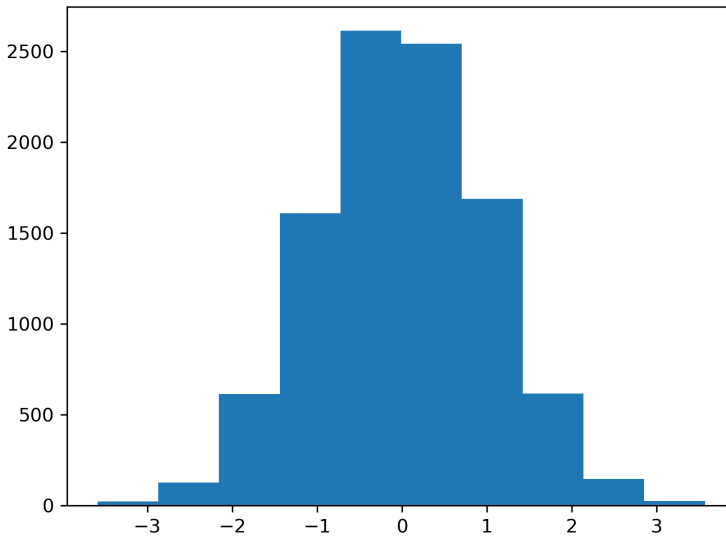
```
import matplotlib.pyplot as plt
import numpy as np

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 gleich große 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])
```

Ausgabe:

```
Die ersten drei 'bins': [-3.98158932 -3.21499646 -2.4484036 ]
```

`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))
```

Ausgabe:

```
n: [ 7. 57. 363. 1319. 2626. 2891. 1887. 679. 150.
21.] 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])
```

Ausgabe:

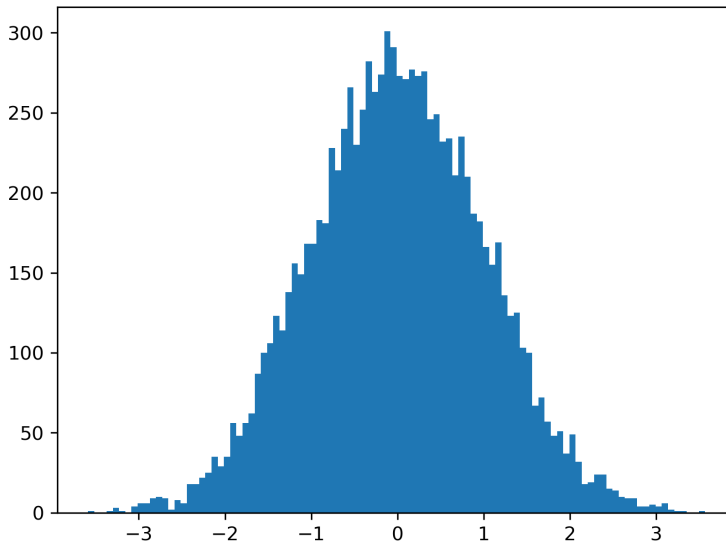
```
patches: <a list of 10 Patch objects>
Rectangle(xy=(-3.98159, 0), width=0.766593, height=7, angle=0)
Rectangle(xy=(-3.215, 0), width=0.766593, height=57, angle=0)
Rectangle(xy=(-2.4484, 0), width=0.766593, height=363, angle=0)
Rectangle(xy=(-1.68181, 0), width=0.766593, height=1319, angle=0)
Rectangle(xy=(-0.915218, 0), width=0.766593, height=2626, angle=0)
Rectangle(xy=(-0.148625, 0), width=0.766593, height=2891, angle=0)
Rectangle(xy=(0.617968, 0), width=0.766593, height=1887, angle=0)
```



```
Rectangle(xy=(1.38456, 0), width=0.766593, height=679, angle=0)
Rectangle(xy=(2.15115, 0), width=0.766593, height=150, angle=0)
Rectangle(xy=(2.91775, 0), width=0.766593, height=21, angle=0)
```

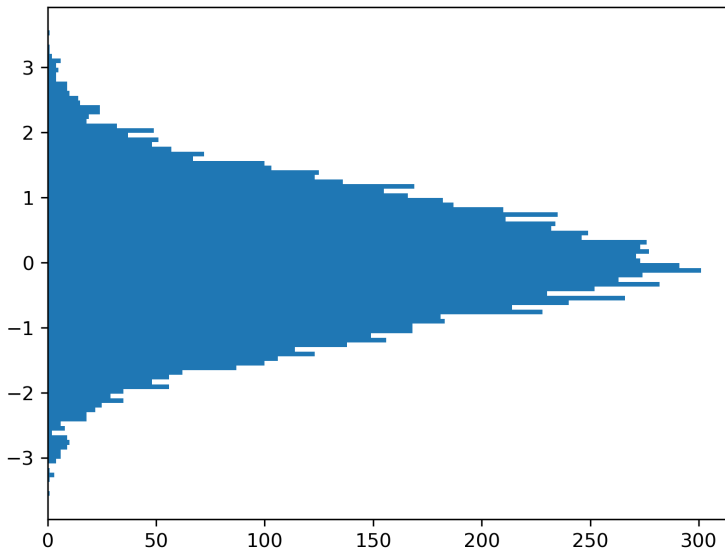
10 Bins sind nicht viel, wenn wir von 10.000 Zufallszahlen sprechen. Deshalb erhöhen wir im folgenden Programm die Anzahl der Klassen. Dazu setzen wir den Schlüsselwortparameter `bins` auf 100:

```
plt.hist(gaussian_numbers, bins=100)
plt.show()
```



Indem wir den Parameter `orientation` auf `vertical` setzen, können wir das Histogramm auch seitwärts ausgeben:

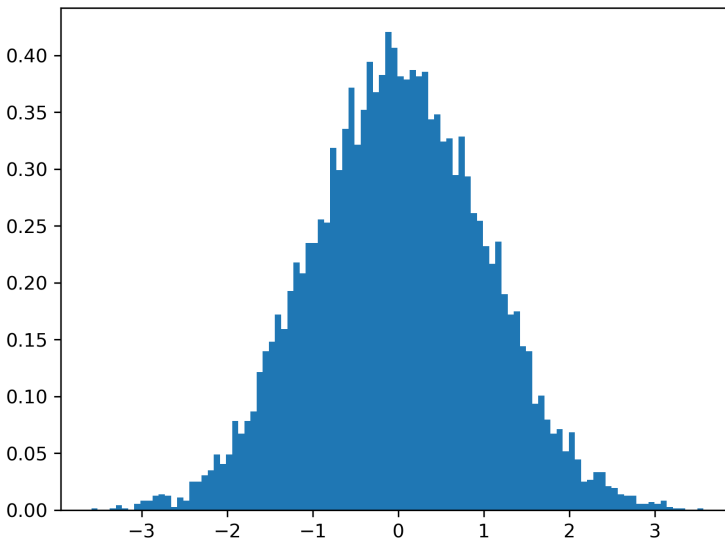
```
plt.hist(gaussian_numbers, bins=100, orientation="horizontal")
plt.show()
```



Ein weiterer wichtiger Schlüsselwortparameter von `hist` ist `density`, der den veralteten Parameter `normed` ablöst. Falls er auf `True` gesetzt wird, wird die erste Komponente – also die Häufigkeiten – des Rückgabepfels 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)))
```



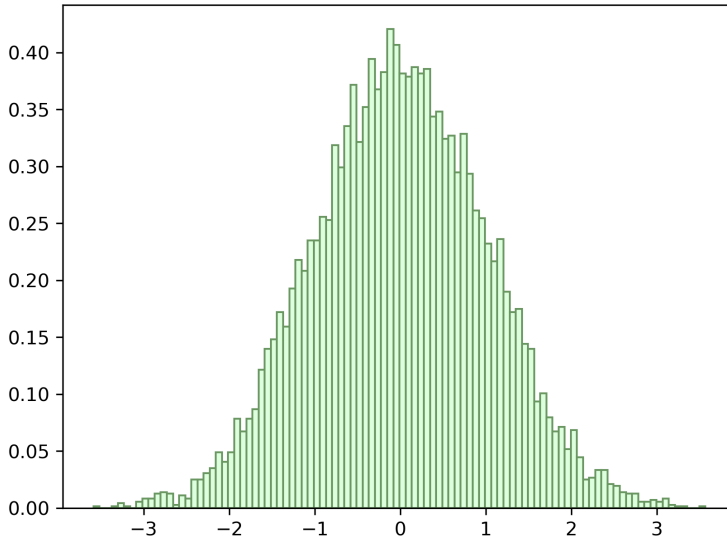
Ausgabe:

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")

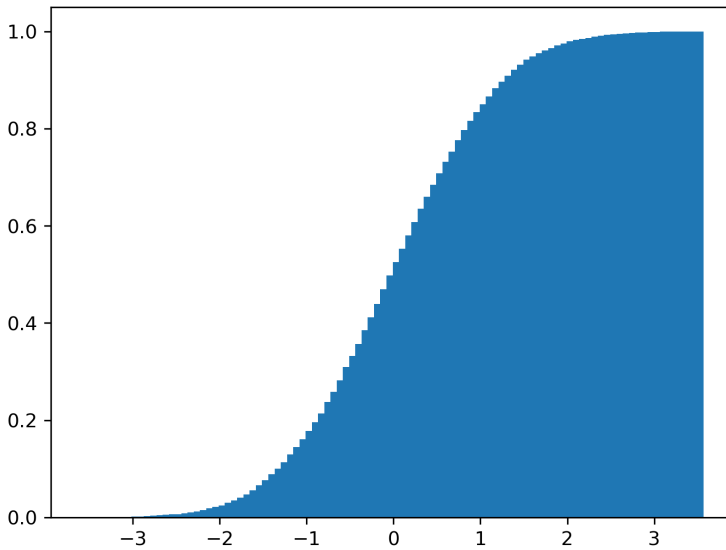
plt.show()
```



Okay, Sie möchten eine Darstellung von kumulierten Werten sehen? Wir können eine kumulative Verteilungsfunktion ausgeben, indem wir den Parameter `cumulative` setzen:

```
plt.hist(gaussian_numbers,
         bins=100,
         density=True,
         stacked=True,
         cumulative=True)

plt.show()
```



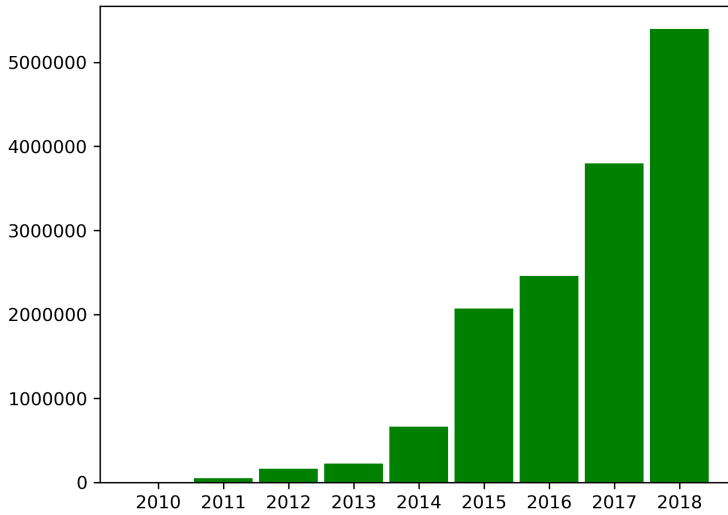
■ 17.2 Säulendiagramm

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()
```

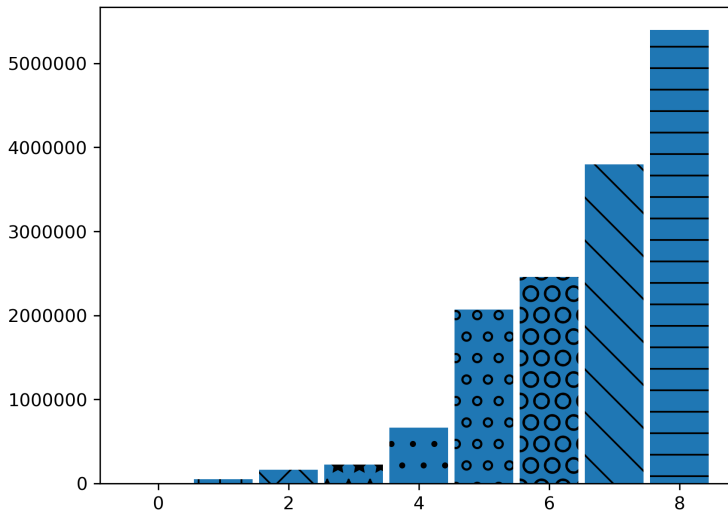


Wenn man keine Farben ausgeben kann (wie beispielsweise in diesem Buch), kann es sinnvoll sein, Schraffierungen¹ zu verwenden. Wie man dies erreichen kann, zeigen wir in folgendem Beispiel:

```
bars = plt.bar(index, visitors, bar_width)

patterns = ('-', '+', 'x', '*', '.', 'o', 'O', '\\\\', '-')
for bar, pattern in zip(bars, patterns):
    bar.set_hatch(pattern)

plt.show()
```



¹ engl. „hatchings“

■ 17.3 Balkendiagramme

Häufig werden Säulendiagramme auch fälschlicherweise als Balkendiagramme bezeichnet, da sie diesen sehr ähnlich sind. Beim Balkendiagramm sind die Rechtecke allerdings waagrecht orientiert.

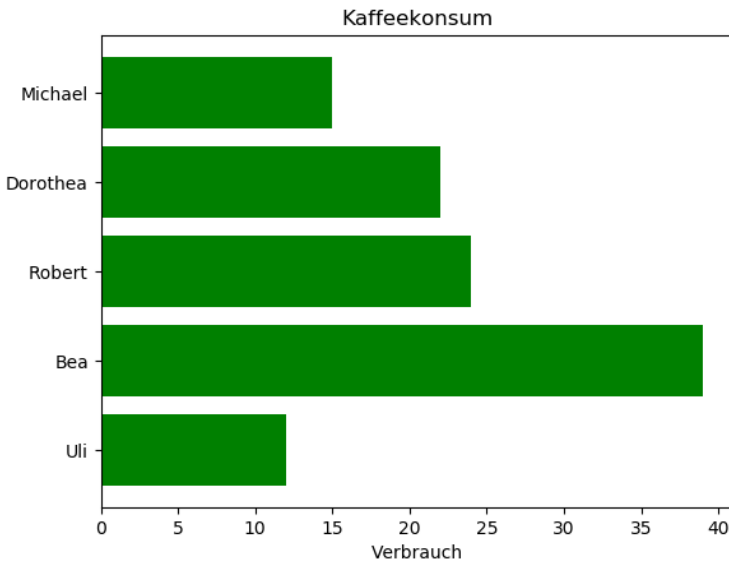
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt

# original default Parameter werden wiederhergestellt:
plt.rcParams()
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()
```



■ 17.4 Aufgaben



1. Aufgabe:

Bei der Bundestagswahl in Deutschland gab es im Jahre 2017 folgende prozentuale Verteilungen für die einzelnen Parteien:

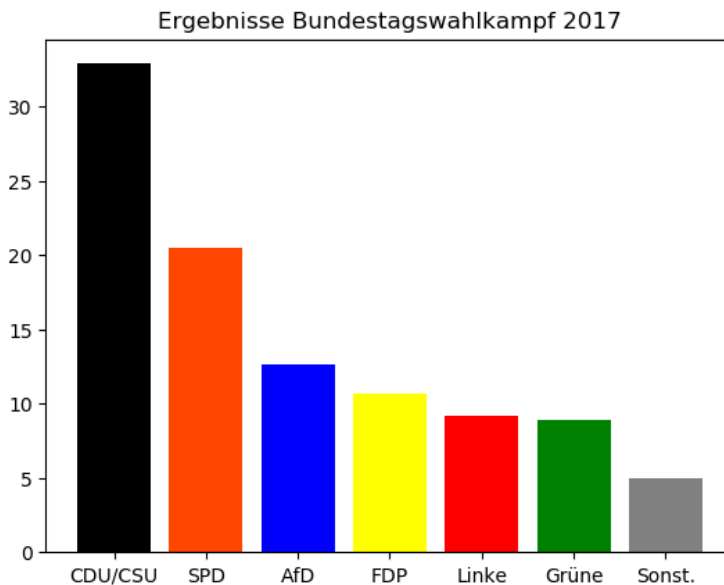
CDU/CSU: 32,9, SPD: 20,5, AfD: 12,6, FDP: 10,7, Linke: 9,2, Grüne: 8,9, Sonstige: 5
Erzeuge ein Säulendiagramm mit diesen Daten.

■ 17.5 Lösung

Lösung zur 1. Aufgabe:

```
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()
```



Teil IV

Pandas

Auch wenn Panda-Bären allgemein als süß und niedlich angesehen werden, bleiben wir in diesem Kapitel bei der Programmierung. Bei Pandas handelt es sich um eine Bibliothek zur Datenanalyse mit Python. Der Name Pandas ist ein Akronym für „panel data“. In der Ökonometrie versteht man darunter Datensätze, die sowohl eine zeitliche als auch eine nichtzeitliche Dimension aufweisen. Man kann diese Daten auch als Matrix betrachten, in der beispielsweise die Spaltenrichtung als Individualdimension und die Zeilenrichtung als Zeitdimension angesehen wird. Dies ist aber bereits ein sehr spezieller Anwendungsfall. Besser ist es, Pandas als ein System zu sehen, das auf Tabellen, so wie sie bei der Tabellenkalkulation verwendet werden, beruht, so wie etwa in dem bekanntesten Tabellenkalkulationsprogramm „Excel“. Eine besondere Stärke von Pandas liegt auch darin, dass es direkt CSV-, DSV- und Excel-Dateien einlesen und schreiben kann.

Oft gibt es Verwirrung darüber, ob Pandas nicht eine Alternative zu NumPy, SciPy und Matplotlib sei. Die Wahrheit ist aber, dass Pandas auf NumPy aufbaut. Das bedeutet auch, dass NumPy für Pandas Voraussetzung ist. SciPy und Matplotlib werden von Pandas nicht grundlegend benötigt, sind aber eine wertvolle Ergänzung. Deshalb listet das Pandas-Projekt diese auch als „optionale Abhängigkeiten“.



	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

Bild 18.1 Pandabären und DataFrame

■ 18.1 Datenstrukturen

Die wichtigen Datenstrukturen von Pandas sind:

- Series und
- DataFrame

Da der Datentyp `DataFrame` auf dem Typ `Series` basiert, beginnen wir mit `Series`.

■ 18.2 Series

Ein Series-Objekt kann man wie die Spalte in einer Excel-Tabelle plus dem zugehörigen Index sehen. Anders ausgedrückt: Eine Series ist ein eindimensionales Array-ähnliches Objekt mit einem Index. Während bei einem Array der Index den natürlichen Zahlen von 0 bis zur Länge des Arrays (exklusive) entspricht, kann der Index einer Series beliebig sein, solange er hashable ist.

Sowohl der Index als auch die Werte einer Series müssen einen einheitlichen Datentyp aufweisen, also beispielsweise nur Integers, Floats, Strings usw.

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 instanziiieren. 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)
```

Ausgabe:

```
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)
```

Ausgabe:

```
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))
```

Ausgabe:

```
[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)
```

Ausgabe:

```
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))
```

Ausgabe:

```
apples      37
oranges     46
cherries    83
pears       42
dtype: int64
Summe aus S:  115
```

Die Indizes müssen für die Addition von Series-Typen nicht identisch sein. 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)
```

Ausgabe:

```
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:

```
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)
```

Ausgabe:

```
apples      NaN
armut       NaN
cherries    NaN
elma        NaN
kiraz       NaN
oranges     NaN
pears       NaN
portakal    NaN
dtype: float64
```

18.2.1 Indizierung

Es ist möglich, auf einzelne Werte eines Series-Objekts zuzugreifen:

```
print(S['apples'])
```

Ausgabe:

```
20
```

Man kann auch auf mehrere Indizes gleichzeitig zugreifen, wenn man ein Listen- oder ein Array-ähnliches Objekt übergibt:

```
print(S[['apples', 'oranges', 'cherries']])
```

Ausgabe:

```
apples      20
oranges     33
cherries    52
dtype: int64
```

Filterung mit einem Booleschen Array:

```
S[S>30]
```

Ausgabe:

```
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))
```

Ausgabe:

```
apples      92
oranges     144
cherries    220
pears       52
```

```
dtype: int64
=====
apples      0.912945
oranges     0.999912
cherries    0.986628
pears       -0.544021
dtype: float64
```

18.2.2 pandas.Series.apply

Series.apply(func, convert_dtype=True, args=(), **kwds)

Die Funktion „func“ wird auf das Series-Objekt angewendet und liefert, in Abhängigkeit von „func“, entweder ein Series-Objekt oder ein DataFrame-Objekt zurück.

Parameter	Bedeutung
func	Eine Funktion, die auf das gesamte Series-Objekt (NumPy-Funktion) oder nur auf einzelne Werte des Series (Python-Funktion) angewendet wird.
convert_dtype	Ein Boolescher Wert. Wenn dieser auf True gesetzt wird (Standard), so wird versucht, bei der Anwendung einen besseren dtype für die elementweisen Funktionsergebnisse zu finden. Wenn der Parameter auf False gesetzt wird, so wird dtype=objekt verwendet.
args	Positionsargumente, die an die Funktion „func“ übergeben werden, zusätzlich zu den Werten des Series-Objektes.
**kwds	Zusätzliche Schlüsselwortargumente, die als Schlüsselworte an die Funktion übergeben werden.

Beispiel:

```
S.apply(np.log)
```

Ausgabe:

```
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 )
```

Ausgabe:

```
apples      30
oranges     43
cherries    52
pears       20
dtype: int64
```

18.2.3 Zusammenhang zu Dictionaries

Ein Series-Objekt kann wie ein geordnetes Python-Dictionary mit einer festen Länge angesehen werden.

Wir können bei der Erstellung eines Series-Objekts ein Dictionary übergeben. Wir erhalten ein Series-Objekt mit den Schlüsseln des Dictionarys als Indizes. Die Indizes werden sortiert.

```
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)
```

Ausgabe:

```
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
```

■ 18.3 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 Dictionarys übereinstimmen, 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 Dictionarys 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 Dictionarys 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)
```

Ausgabe:

```
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
```

Ausgabe:

```
London      8615246
Paris       2273305
Berlin      3562166
Hamburg     1760433
dtype: int64
```

18.3.1 Die Methoden isnull() und notnull()

Wir sehen, dass die Städte, die nicht im Dictionary existieren, den Wert NaN zugewiesen bekommen. NaN steht für „not a number“. Es kann in unserem Beispiel auch als „fehlt“ verstanden werden.

Wir können mit den Methoden `isnull` und `notnull` fehlende Werte prüfen:

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]
my_city_series = pd.Series(cities, index=my_cities)
print(my_city_series.isnull())
```

Ausgabe:

```
London      False
Paris       False
Zurich       True
Berlin      False
Stuttgart    True
Hamburg     False
dtype: bool
print(my_city_series.notnull())
```

Ausgabe:

```
London      True
Paris       True
Zurich      False
```



```
Berlin      True
Stuttgart   False
Hamburg     True
dtype: bool
```

18.3.2 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)
```

Ausgabe:

```
a    23.0
b    45.0
c     NaN
d     0.0
dtype: float64
pd.isnull(S)
```

Ausgabe:

```
a    False
b    False
c     True
d    False
dtype: bool
pd.notnull(S)
```

Ausgabe:

```
a    True
b    True
c    False
d    True
dtype: bool
```

18.3.3 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())
```

Ausgabe:

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
```

18.3.4 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))
```

Ausgabe:

```
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
```

Ausgabe:

```
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)
```

Ausgabe:

London	8615246
Paris	2273305
Zurich	378884
Berlin	3562166
Stuttgart	597939
Hamburg	1760433

dtype: int64

Im vorigen Kapitel haben wir gesehen, dass der Datentyp `Series` logisch gesehen einer Spalte mit Index einer Excel-Tabelle entspricht. 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 Datentyp – 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.

	A	B	C
1	Country	City	Population
2	England	London	8615246
3	Germany	Berlin	3562166
4	Spain	Madrid	3165235
5	Italy	Rome	2874038
6	France	Paris	2273305
7	Austria	Vienna	1805681
8	Romania	Bucharest	1803425
9	Germany	Hamburg	1760433
10	Hungary	Budapest	1754000
11	Poland	Warsaw	1740119
12	Spain	Barcelona	1602386
13	Germany	Munich	1493900
14	Italy	Milan	1350680

Bild 19.1 Spreadsheet und DataFrames

■ 19.1 Zusammenhang zu Series

Ein `DataFrame` hat einen Zeilen- und einen Spaltenindex. 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:

```
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])
```

Ausgabe:

```
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)
```

Ausgabe:

```
      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))
```

Ausgabe:

```
<class 'pandas.core.frame.DataFrame'>
```

Das bedeutet, dass wir `Series`-Objekte durch Konkatenierung in `DataFrame`-Objekte wandeln können!

■ 19.2 Manipulation der Spaltennamen

Wenn wir uns das eben erzeugte `DataFrame` anschauen, stört uns, dass die Spaltennamen durch wenig aussagekräftige Nummern bezeichnet werden, also 0, 1 und 2.

```
shops_df.columns
```

Ausgabe:

```
RangeIndex(start=0, stop=3, step=1)
shops_df.columns.values
```

Ausgabe:

```
array([0, 1, 2])
```

Wir sehen, dass das `DataFrame`-Objekt eine Property `columns` zur Verfügung stellt, um auf die Spalten zuzugreifen. Nehmen wir nun an, dass sich unsere Shops in Zürich, Winterthur

und Freiburg befinden. Dann wäre es sicherlich von Vorteil, diese Namen auch als Spaltennamen zu verwenden. Dazu müssen wir unsere Spaltennamen umbenennen, was wir über die Property columns bewerkstelligen können:

```
cities = ["Zürich", "Winterthur", "Freiburg"]
shops_df.columns = cities
print(shops_df)
```

Ausgabe:

	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 benannt 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)
```

Ausgabe:

	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

■ 19.3 Zugriff auf Spalten

Auf die Spalten eines DataFrames können wir einfach durch Indizierung zugreifen:

```
print(shops_df["Zürich"])
```

Ausgabe:

```
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"]))
```

Ausgabe:

```
<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)
```

Ausgabe:

```
2014    2409.14
2015    2941.01
2016    3496.83
2017    3119.55
Name: Zürich, dtype: float64
```

■ 19.4 DataFrames aus Dictionaries

Die Zugriffe auf die Spalten eines DataFrames erinnern an den Zugriff bei Dictionaries. Spaltenname wäre der Key, und die Werte der Spalte entsprechen dann den Werten eines Dictionary. So ist es nicht verwunderlich, dass man aus folgendem Dictionary `cities` auf direkte Art ein DataFrame erzeugen kann:

```
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)
```

Ausgabe:

	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

19.5 Index ändern

Bei der Erzeugung des DataFrames `city_frame` wurde automatisch der Index 0,1,2,... erzeugt. Bei der Erzeugung können wir aber auch einen eigenen Index verwenden:

```
ordinals = ["first", "second", "third", "fourth",
            "fifth", "sixth", "seventh", "eighth",
            "ninth", "tenth", "eleventh", "twelvth",
            "thirteenth"]
city_frame = pd.DataFrame(cities, index=ordinals)
print(city_frame)
```

Ausgabe:

	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
eighth	Hamburg	1760433	Germany
ninth	Budapest	1754000	Hungary
tenth	Warsaw	1740119	Poland
eleventh	Barcelona	1602386	Spain
twelvth	Munich	1493900	Germany
thirteenth	Milan	1350680	Italy

19.5.1 Umsortierung der Spalten

Die Reihenfolge der Spalten kann zum Zeitpunkt der Erstellung des DataFrames explizit festgelegt werden. Dazu dient der Schlüsselwortparameter `columns`:

```
city_frame = pd.DataFrame(cities,
                          columns = ["name",
                                    "country",
                                    "population"])
print(city_frame)
```

Ausgabe:

	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 als auch 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'])
```

Ausgabe:

	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 DataFrames 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)
```

Ausgabe:

	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

19.5.2 Spalte in Index umfunktionieren

Der Index des vorigen Beispiels ist nicht besonders informationsträchtig. Im Prinzip zählt er lediglich die Zeilen auf. Man könnte sich auch beispielsweise den Ländernamen als Index wünschen. Nun zeigen wir, wie man direkt bei der Erzeugung eines DataFrames aus einem Dictionary den Index definieren kann. Dazu weisen wir dem Schlüsselwortparameter `index` den Wert von `cities['country']` zu.

```
city_frame = pd.DataFrame(cities,
                           columns=['name', 'population'],
                           index=cities['country'])

print(city_frame)
```

Ausgabe:

	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

Wie sieht es aber aus, wenn ein DataFrame bereits existiert und wir den Index neu setzen wollen bzw. müssen? Zu diesem Zweck steht die Methode `set_index` zur Verfügung. Mit ihr lässt sich beispielsweise eine Spalte in einen Index wandeln:

```
city_frame = pd.DataFrame(cities)
city_frame2 = city_frame.set_index("country")
print(city_frame2)
```

Ausgabe:

	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 ver-

sehen, 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)
```

Ausgabe:

	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

■ 19.6 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"])
```

Ausgabe:

	name	population
Germany	Berlin	3562166
Germany	Hamburg	1760433
Germany	Munich	1493900

Will man mehrere Indexwerte angeben, übergibt man diese als Liste an `loc`:

```
print(city_frame.loc[["Germany", "France"]])
```

Ausgabe:

	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, wenn also in unserem Beispiel die Bevölkerungsanzahl größer als zwei Millionen ist:

```
print(city_frame.loc[city_frame.population > 2000000])
```

Ausgabe:

	name	population
England	London	8615246
Germany	Berlin	3562166
Spain	Madrid	3165235
Italy	Rome	2874038
France	Paris	2273305

■ 19.7 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())
```

Bei den Bevölkerungszahlen ergibt die Summation mehr Sinn.

```
city_frame["population"].sum()
```

Ausgabe:

```
33800614
```

Mit `cumsum` berechnen wir die kumulative Summe:

```
x = city_frame["population"].cumsum()
print(x)
```

Ausgabe:

```
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
```

■ 19.8 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 Fol-

genden 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())
```

Ausgabe:

	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 ursprünglichen DataFrame anfügen:

```
city_frame = pd.DataFrame(cities,
                           columns=["country",
                                    "population",
                                    "cum_population"],
                           index=cities["name"])

print(city_frame.head())
```

Ausgabe:

	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())
```

Ausgabe:

	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())
```

Ausgabe:

	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())
```

Ausgabe:

	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

19.9 Sortierung

DataFrames lassen sich anhand von bestimmten Kriterien sortieren. Im folgenden Beispiel sortieren wir den Inhalt des DataFrame-Objekts anhand der area-Werte in absteigender Größe:

```
city_frame = city_frame.sort_values(by="area", ascending=False)
print(city_frame)
```

Ausgabe:

	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)
```

Ausgabe:

	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

■ 19.10 Spalten einfügen

In vorherigen Beispielen haben wir Spalten bei der Erstellung des DataFrames 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)
```

Ausgabe:

	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

■ 19.11 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)
```

Ausgabe:

	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())
```


Ausgabe:

	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)
```

Ausgabe:

	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)

```

Ausgabe:

	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

19.12 Aufgaben



1. Aufgabe:

Erzeugen Sie ein DataFrame, das wie folgt aussieht:

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

**2. Aufgabe:**

Vertauschen Sie die Indices der vorigen Series.

**3. Aufgabe:**

Erzeugen Sie ein beliebiges DataFrame mit einem Index, der aus Vornamen besteht, sowie einer Spalte für das Gewicht und einer Spalte für die Körpergröße.

Extrahieren Sie dann alle Zeilen, deren BMI¹ im normalen Bereich liegt, d.h. zwischen 18.5 und 25.

Es gilt:

$$BMI = \frac{g}{h * * 2}$$

**4. Aufgabe:**

Geben Sie die Zeilen aus, in deren Namen ein kleines „i“ vorkommt.

**5. Aufgabe:**

Fügen Sie an den in Aufgabe 3 erzeugten DataFrame eine Zeile mit dem BMI an.

**6. Aufgabe:**

Geben Sie das in der letzten Aufgabe erzeugte DataFrame absteigend sortiert nach dem BMI aus.

**7. Aufgabe:**

Geben Sie nun alle Zeilen aus, deren BMI-Werte innerhalb von 18.5 und 23 liegen und deren Vornamen ein „a“ enthalten.

**8. Aufgabe:**

Erzeugen Sie ein DataFrame mit einem Index, der aus den Monatsnamen besteht, und die Spaltennamen Vornamen entsprechen. Füllen Sie nun die Spalten mit zufälligen ganzen Zahlen zwischen 120 und 200.

**9. Aufgabe:**

Kehten Sie das eben erzeugte DataFrame um, sodass der Index den Vornamen und die Spaltennamen den Monatsnamen entsprechen.

■ 19.13 Lösungen

Lösung zur 1. Aufgabe:

```
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)
```

Ausgabe:

```
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
```

Lösung zur 2. Aufgabe:

```
city_series = city_series.sort_index()

city_series = city_series.swaplevel()
city_series.sort_index(inplace=True)
print(city_series)
```

Ausgabe:

```
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

```

Lösung zur 3. Aufgabe:

```

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, 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)

```

Ausgabe:

	Gewicht	Größe
Henry	65	179
Sarah	58	165
Maria	68	176

Lösung zur 4. Aufgabe:

```
pdf.loc[pdf.index.str.contains("i")]
```

Ausgabe:

	Gewicht	Größe
Toni	99	189
Maria	68	176
Chris	60	175

Lösung zur 5. Aufgabe:

```

pdf.insert(loc=len(pdf.columns),
           column="BMI",
           value=(pdf.Gewicht / ((pdf.Größe/100) ** 2)))
pdf

```

Ausgabe:

	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

Lösung zur 6. Aufgabe:

```
pdf.sort_values(by="BMI", ascending=False)
```

Ausgabe:

	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

Lösung zur 7. Aufgabe:

```
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])
```

Ausgabe:

	Gewicht	Größe	BMI
Sarah	58	165	21.303949
Vera	43	150	19.111111
Maria	68	176	21.952479

Lösung zur 8. Aufgabe:

```
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)
```

Ausgabe:

	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]

Lösung zur 9. Aufgabe:

```
new_df = df.transpose()
print(new_df)
```

Ausgabe:

	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]

Pandas mit seinen mächtigen Datenstrukturen „Series“ und „DataFrame“ wäre ohne seine mächtigen Möglichkeiten, Daten aus Dateien zu lesen und zu schreiben, lediglich eine nette Spielerei. Natürlich könnte man Dateien auch mittels der Ein-/Ausgabemöglichkeiten des reinen Python, also ohne Zusatzmodule, bearbeiten, aber dann wäre zum einen der Code umständlich zu schreiben, und zum anderen wäre es nicht effizient genug. Pandas bietet deshalb für alle wichtigen Datenformate Highlevel-Funktionalitäten zum Lesen und Schreiben dieser Datenformate. So bietet es unter anderem Funktionen zum Lesen von JSON, HTML, HDF5-Format, Feather-Format, SAS und SQL.

In diesem Kapitel beschäftigen wir uns allerdings nur mit „MS Excel“- und CSV-Dateien, da dies auch die wichtigsten und am häufigsten benutzten Datenformate sind. Wir werden sehen, dass man diese Dateiformate mühelos in DataFrames wandeln und ebenso leicht DataFrames in diesen Dateiformaten abspeichern kann.

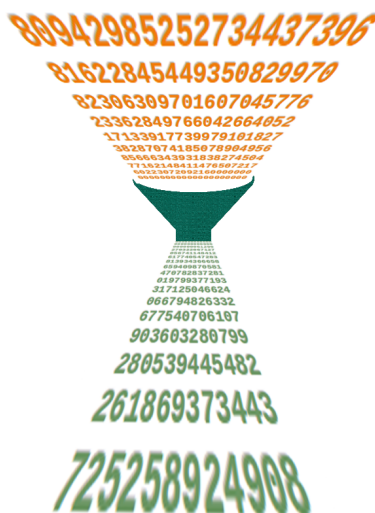


Bild 20.1 Digits as File Input and Output

■ 20.1 Trennerseparierte Werte

Die meisten Menschen verwenden den Namen „CSV-Datei“ als Synonym für eine trennerseparierte Datei. Sie beachten nicht die Tatsache, dass CSV ein Akronym für „comma separated values“ ist (also auf 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 Art 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` im Verzeichnis `data1` handelt es sich um eine DSV-Datei, die Tabulatoren (`\t`) als Trennzeichen benutzt.

■ 20.2 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)
```

Ausgabe:

	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 Spaltennamen 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())
```

Ausgabe:

	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

20.3 Schreiben von CSV-Dateien

DataFrames können wir mit der Methode `to_csv` in CSV-Dateien schreiben. Wir werden dies an einem Beispiel demonstrieren. Zuerst erzeugen wir jedoch Daten, die wir dann ausschreiben werden. Im Verzeichnis `data1` liegen die beiden Dateien

- `countries_male_population.csv` und
- `countries_female_population.csv`

die entsprechend die männlichen und weiblichen Bevölkerungszahlen von Ländern enthalten. Wir werden eine neue Datei mit der Summe von beiden, also der Gesamtbevölkerung, erzeugen.

```
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()
```

Ausgabe:

	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]]
```

Ausgabe:

		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)
```

Ausgabe:

		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")
```

■ 20.4 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 die 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
```

Ausgabe:

	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])
```

Ausgabe:

```
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
```

■ 20.5 Aufgaben



1. Aufgabe:

Die Datei „countries_population.csv“ ist eine CSV-Datei, die die Bevölkerungszahlen aller Länder enthält (July 2014). Als Trennzeichen fungiert ein Leerzeichen, und Kommas trennen die Tausenderpositionen in Zahlen. Lesen Sie die Datei in einen DataFrame ein und geben Sie dann die ersten fünf Zeilen des DataFrame aus.



2. Aufgabe:

- Lesen Sie die csv-Datei (csv) `bundeslaender.txt` ein. Erzeugen Sie eine neue Datei mit den Spaltennamen 'land', 'area', 'female', 'male', 'population' und 'density', also die Anzahl der Einwohner pro Quadratkilometer.
- Geben Sie alle Zeilen aus mit einer Fläche größer als 30.000 und deren Bevölkerung größer als 10.000 ist.
- Geben Sie alle Zeilen aus, deren Dichte größer als 30.000 ist.



3. Aufgabe:

Im Verzeichnis `data1` befindet sich eine Datei `person_data.txt` mit Personendaten. Jede Zeile hat folgenden Aufbau:

Vorname Nachname Größe Gewicht Geschlecht

Lesen Sie die Datei in ein DataFrame ein.

Erzeugen Sie dann eine Spalte mit dem BMI¹, die Sie als Letztes an das DataFrame anhängen.

Es gilt:

$$BMI = \frac{g}{h * *2}$$

■ 20.6 Lösungen

Lösung zur 1. Aufgabe:

```
pop = pd.read_csv("data1/countries_population.csv",
                  header=None,
                  names=["Country", "Population"],
                  index_col=0,
                  quotechar=" ",
                  sep=" ",
```

```

        thousands=",")
print(pop.head(5))

```

Ausgabe:

Country	Population
China	1355692576
India	1236344631
European Union	511434812
United States	318892103
Indonesia	253609643

Lösung zur 2. Aufgabe:

```

lands = pd.read_csv('data1/bundeslaender.txt', sep=" ")
print(lands.columns.values)

```

Ausgabe:

```

['land' 'area' 'male' 'female']
# swap the columns of our DataFrame:
lands = lands.reindex(columns=['land', 'area', 'female', 'male'])
lands[:2]

```

Ausgabe:

	land	area	female	male
0	Baden-Württemberg	35751.65	5465	5271
1	Bayern	70551.57	6366	6103

```

lands.insert(loc=len(lands.columns),
             column='population',
             value=lands['female'] + lands['male'])
lands[:3]

```

Ausgabe:

	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

```

lands.insert(loc=len(lands.columns),
             column='density',
             value=(lands['population'] * 1000 /
                    lands['area']).round(0))

lands[:4]

```

Ausgabe:

	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]
print(lands.loc[(lands.area>30000) & (lands.population>10000)])

```

Ausgabe:

	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]

Lösung zur 3. Aufgabe:

```
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)
```

Ausgabe:

	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

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 Fließkommaimplementierungen gefunden hatte, welche es schwierig gemacht haben, diese einfach und übergreifend zu verwenden.



Bild 21.1 Umgang mit NaN

Der Standard fügte NaN zu den arithmetischen Formaten – Mengen aus binären und dezimalen Fließkommaformaten – hinzu.

■ 21.1 '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))
```

Ausgabe:

```
nan nan nan nan
<class 'float'>
```

nan ist seit Python 3.5 auch Teil des `math`-Moduls:

```
import math
n1 = math.nan
```



```
print(n1)
print(math.isnan(n1))
```

Ausgabe:

```
nan
True
```

Achtung: Führen Sie keine Vergleiche 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)
```

Ausgabe:

```
False
False
False
False
```

■ 21.2 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())
```

Ausgabe:

	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 den Durchschnitt über den Messtag berechnet.

```
df.mean()
```

Ausgabe:

```
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
```

Ausgabe:

```
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])
```

Ausgabe:

```
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])
```

Ausgabe:

```
           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
```

21.2.1 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 nun zu Übungszwecken eine solche Datei 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 Temperaturenbeispiel weitermachen, möchten wir die Arbeitsweise der `where`-Methode an einfachen Beispielen demonstrieren:

```
s = pd.Series(range(5))
s.where(s > 1)
```

Ausgabe:

```
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)
```

Ausgabe:

	Foo	Bar
0	-15	6
1	-25	24
2	4	-11
3	-1	2

Für unser Temperaturenbeispiel 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`-Werte. Dazu erstellen wir ein `DataFrame`-Objekt mit Zufallswerten zwischen 0 und 1 mit der An-

weisung `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])
```

Ausgabe:

	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])
```

Ausgabe:

	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

■ 21.3 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)
```

Ausgabe:

	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]
```

Ausgabe:

```
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 Minimalwert 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])
```

Ausgabe:

	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])
```

Ausgabe:

	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

■ 21.4 Aufgaben



1. Aufgabe:

In diesem Kapitel hatten wir das DataFrame `disturbed_data` mit gestörten Temperaturmesswerten benutzt. Wir hatten es gefiltert, indem wir nur Zeilen zugelassen hatten, die höchstens ein NaN enthalten. Erzeugen Sie nun ein etwas fehlertoleranteres DataFrame, in dem maximal zwei NaN-Werte erlaubt sind.



2. Aufgabe:

Erzeugen Sie nun ein DataFrame, in dem es nur noch eine Spalte mit den Durchschnittswerten gibt.

■ 21.5 Lösungen

Lösung zur 1. Aufgabe:

Alles, was wir tun müssen, ist, den Schlüsselwortparameter `thresh` auf 4 zu setzen, d.h. wir verlangen, dass mindestens 4 gültige Werte in einer Zeile vorkommen. Dies bedeutet andersherum, dass nun zwei Werte NaN sein können:

```
cleansed_df = disturbed_data.dropna(thresh=4, axis=0)
print(cleansed_df[:7])
```

Ausgabe:

	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

Lösung zur 2. Aufgabe:

```
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])
```

Ausgabe:

	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

■ 22.1 Einführung

Binning ist eine Technik der Statistik, die in der Datenvorverarbeitung bzw. Datenaufbereitung angewendet wird. Unter Binning versteht man eine Klassenbildung in der Vorverarbeitung bei der Datenanalyse. Eine gegebene Menge von Werten, die sortiert sind, wird anhand ihrer Größe in die passenden Intervalle aufgeteilt. Diese Intervalle bezeichnet man im Englischen als „bins“ (deutsch „Behälter“). Jedes dieser Intervalle (Bins) wird dann durch einen Repräsentanten gekennzeichnet. Man bezeichnet diese auch als Intervalllabels. Binning wird häufig angewendet, wenn es mehr mögliche Daten gibt, als man eigentlich braucht. So können zum Beispiel Körpergrößen oder Gewichtsangaben von Menschen in Intervallen oder Kategorien eingeteilt werden.

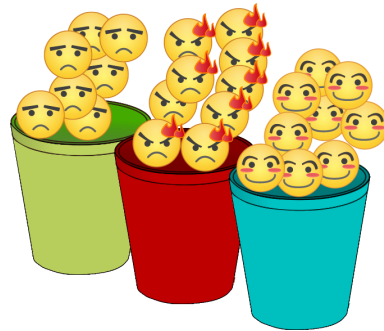


Bild 22.1 Binning

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 Testgruppe 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 Originaldaten werden also 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 $k \times k$ 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)
```

Ausgabe:

```
[(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 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)

```

Ausgabe:

```

[(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})

```

■ 22.2 Binning mit Pandas

Das Pandas-Modul bietet starke Funktionalitäten für die Einteilung von Daten. Wir demonstrieren dies anhand der vorigen Daten.

22.2.1 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 Struktur sein, wie z.B. Listen, Tupel, nd-arrays usw.:

```
categorical_object = pd.cut(weights_of_persons, bins2)
print(categorical_object)
```

Ausgabe:

```
[(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 zwischen 70 (exklusive) und 74 (inklusive) beinhaltet. Mathematisch handelt es sich dabei um ein halb-offenes Intervall. Das heißt, dass ein Endpunkt des Intervalls inklusive ist, der andere 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` auf `'left'` setzen:

```
bins2 = pd.IntervalIndex.from_tuples(bins, closed="left")
categorical_object = pd.cut(weights_of_persons, bins2)
print(categorical_object)
```

Ausgabe:

```
[(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))
```

22.2.2 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 Maximumwerte 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])
```

Ausgabe:

```
[(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])
```

Ausgabe:

```
[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)
```

22.2.3 Bins und Werte zählen

Die nächste und interessanteste Frage ist, wie wir denn die aktuellen Werte eines Bins sehen können. Das erreichen wir mit der Funktion `value_counts()`:

```
print(pd.value_counts(categorical_object))
```

Ausgabe:

```
[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)
```

Ausgabe:

```
[ 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
```

Ausgabe:

```
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] )
```

Ausgabe:

```
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)
```

22.2.4 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))
```

Ausgabe:

```
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] )
```

Ausgabe:

```
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
```


■ 23.1 Einführung

In den bisherigen Kapiteln haben wir in unseren Beispielen zu Pandas-Series und -DataFrames diverse Indizes kennengelernt. Diese Indizes hatten alle gemeinsam, dass

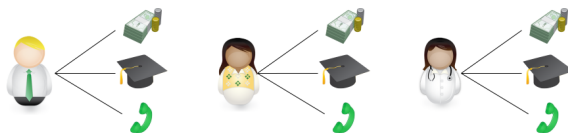


Bild 23.1 Mehrstufige Indizierung

sie einstufig waren. Für viele Anwendungen genügt dies aber nicht. Nehmen wir beispielsweise an, dass es zu mehreren Filialen einer Ladenkette Umsatzzahlen gibt, die nach Jahren sortiert sind. Dann könnte man diese Daten so aufbereiten, dass die Namen der Filiale den obersten Index abbilden und die Jahre den Unterindex.

■ 23.2 Mehrstufig indizierte Series

Mehrstufige Indizierung ist sowohl für Series als auch für DataFrames verfügbar. Es ist eine faszinierende Möglichkeit, in höheren Datendimensionen 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)
```

Ausgabe:

```
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

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
```

Ausgabe:

```
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
```

■ 23.3 Zugriffsmöglichkeiten

Wir können über folgenden Weg auf die Daten, die mit dem ersten Index bezeichnet sind, zugreifen:

```
print(city_series["Vienna"])
```

Ausgabe:

```
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"])
```

Ausgabe:

```
414.6
```

Zur Vervollständigung der zweite Weg:

```
print(city_series["Vienna", "area"])
```

Ausgabe:

```
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"])
```

Ausgabe:

```
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 Städtenamen als Schlüssel verwendet:

```
city_series[["Vienna", "Berlin"]]
```

Ausgabe:

```

Berlin  area          891.85
        country       Germany
        population    3562166
Vienna  area          414.6
        country       Austria
        population    1805681
dtype: object

```

Im nächsten Beispiel zeigen wir, wie mittels Slicing auf die inneren Schlüssel zugegriffen werden kann:

```
print(city_series[:, "area"])
```

Ausgabe:

```

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])

```

Ausgabe:

```

Oberste Hierarchiestufe:
Index(['Berlin', 'Hamburg', 'Vienna', 'Zürich'], dtype='object')
Untere Hierarchiestufe:
Index(['area', 'country', 'population'], dtype='object')

```

■ 23.4 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 erzeugen

kann. Man kann dies zwar mit folgendem Code erreichen, aber wir werden danach einen direkteren Weg zeigen.¹

```
city_df = pd.DataFrame([], index=index[0])
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)
```

Ausgabe:

	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])
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)
```

Ausgabe:

	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 DataFrames.
- `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)
```

¹ Der Parameter „`sort`“ sollte gesetzt werden, obwohl bis zur Version Python 3.7 der Index automatisch sortiert wird. In Zukunft muss dieser Parameter jedoch gesetzt werden.

```
print("\nErgebnis für level=0:")
print(city_df)
```

Ausgabe:

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()
```

Ausgabe:

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

■ 23.5 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. Genau genommen erhielten wir eine zweistufige Series. Im folgenden Beispiel zeigen wir einen 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)
```

Ausgabe:

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 mithilfe der Methode `unstack` ein DataFrame erzeugen. Wir zeigen die verschiedenen Möglichkeiten für den Parameter `level`:

```
print(S3_series.unstack(level=-1)) # entspricht 'level=2'
```

Ausgabe:

		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))
```

Ausgabe:

		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)
```

Ausgabe:

	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"])
```

Ausgabe:

```
cold      8368
hot       16031
Name: (red, right), dtype: int64
x = S3_series.unstack(level=[2, 1])
print(x)
```

Ausgabe:

```

      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"]

```

Ausgabe:

```

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.

■ 23.6 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

```

Ausgabe:

```

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

```

Ausgabe:

```

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 ---
```

Ausgabe:

```

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

```

■ 23.7 Aufgaben



1. Aufgabe:

Wandeln Sie das folgende Dictionary in eine Series mit zweistufigem Index, wobei die Ländernamen den primären und die Jahreszahlen den sekundären Index bilden:

```

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}

```



2. Aufgabe:

Vertauschen Sie den mehrstufigen Index der in der vorigen Aufgabe erzeugten Series, d.h. dass die Jahreszahlen den primären und die Länder den sekundären bilden.

■ 23.8 Lösungen

Lösung zur 1. Aufgabe:

```
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
```

Ausgabe:

```
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
```

Lösung zur 2. Aufgabe:

```
growth_rates_series = growth_rates_series.swaplevel()
growth_rates_series.sort_index(inplace=True)
growth_rates_series
```

Ausgabe:

```
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
```

■ 24.1 Einführung

Unter Datenvisualisierung versteht man die Darstellung von Daten in einem bildlichen oder grafischen Format. Es ist selten eine gute Idee, wenn man wissenschaftliche oder geschäftliche Daten nur rein textuell bzw. als Zahlenwerk dem jeweiligen Zielpublikum präsentiert. Die geeignete Visualisierung der Daten ermöglicht es Entscheidungs-

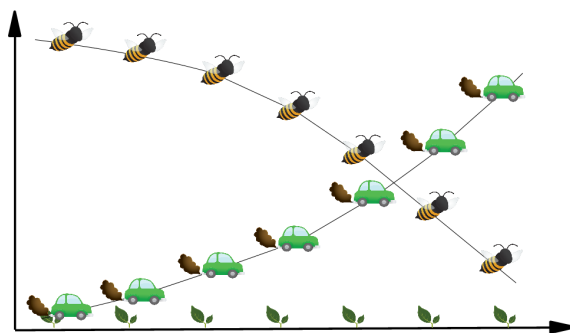


Bild 24.1 Autos und Bienen

trägern, schwierige Konzepte verstehen oder neue Muster erkennen zu können. 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. Grafiken können noch durch Basiselemente wie beispielsweise Legenden, Bezeichnungen usw. erweitert werden.

Auch wenn sich all dies bereits mit Matplotlib realisieren lässt, so handelt es sich dabei um ein Low-Level-Werkzeug, d.h. dass sich die gewünschten Repräsentationen nur umständlich oder schwer umsetzen lassen. Pandas bietet eigene Möglichkeiten, Visualisierungen leichter umzusetzen. Dennoch setzen die Visualisierungstools von Pandas auf Matplotlib auf.

Wir beginnen mit einem einfachen Beispiel eines Liniendiagramms.

■ 24.2 Liniendiagramme in Pandas

24.2.1 Series

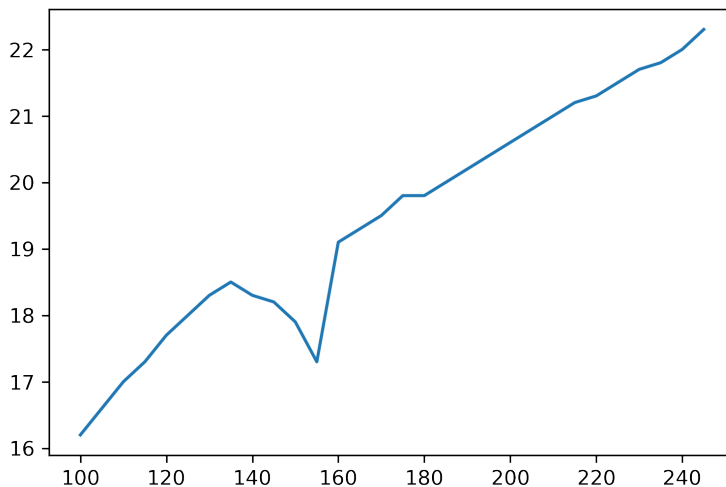
Sowohl die Series als auch die DataFrames bieten eine Plot-Funktionalität.

Im Folgenden präsentieren wir ein einfaches Beispiel eines Liniendiagramms für ein Series-Objekt:

```
import matplotlib.pyplot as plt
import pandas as pd

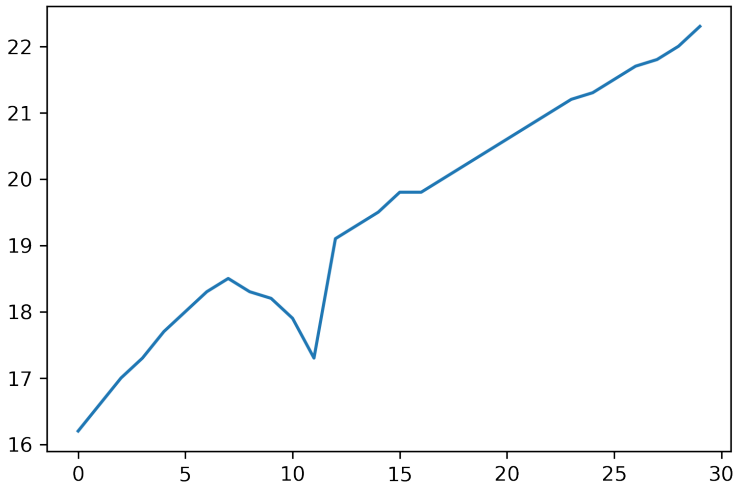
data = [16.2, 16.6, 17.0, 17.3, 17.7, 18.0,
        18.3, 18.5, 18.3, 18.2, 17.9, 17.3,
        19.1, 19.3, 19.5, 19.8, 19.8, 20.0,
        20.2, 20.4, 20.6, 20.8, 21.0, 21.2,
        21.3, 21.5, 21.7, 21.8, 22.0, 22.3]
s = pd.Series(data, index=range(100, 250, 5))

s.plot()
plt.show()
```



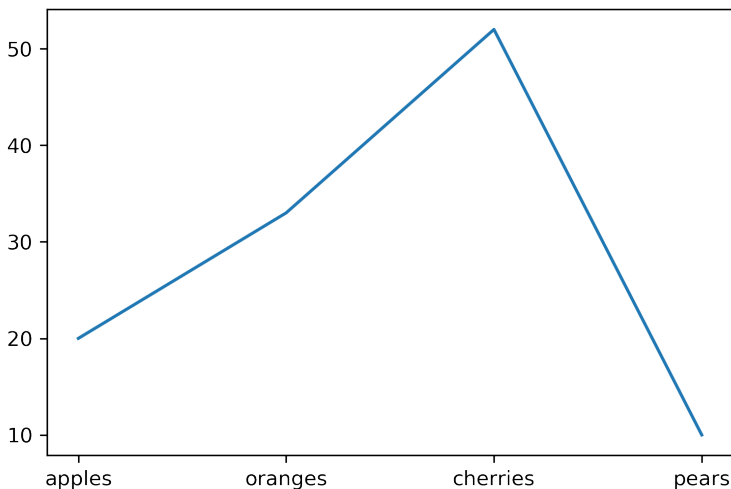
Es ist möglich, die Verwendung des Indexes zu unterdrücken, indem der Parameter `use_index` auf `False` gesetzt wird:

```
s.plot(use_index=False)
plt.show()
```



Wir experimentieren jetzt mit einem Series-Objekt, welches einen Index mit alphabetischen Werten hat. Wir mussten hier `plt.plot(S)` statt `S.plot()` schreiben, da sonst die Früchtenamen nicht erschienen wären.¹

```
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()
```



¹ Vor der Version 0.23.4 von Pandas genügte es, 'S.plot()' zu schreiben, und man erhielt dennoch die Früchtenamen.

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.

24.2.2 DataFrames

Selbstverständlich ermöglicht Pandas auch das Plotten von DataFrames. Wir wollen dies an einem Beispiel demonstrieren. Dazu konstruieren wir ein DataFrame-Objekt aus den monatlichen Temperatur- und Niederschlagsdaten der Insel Kreta. Für dieses DataFrame-Objekt werden wir dann ein Liniendiagramm erzeugen:

```
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))
```

Ausgabe:

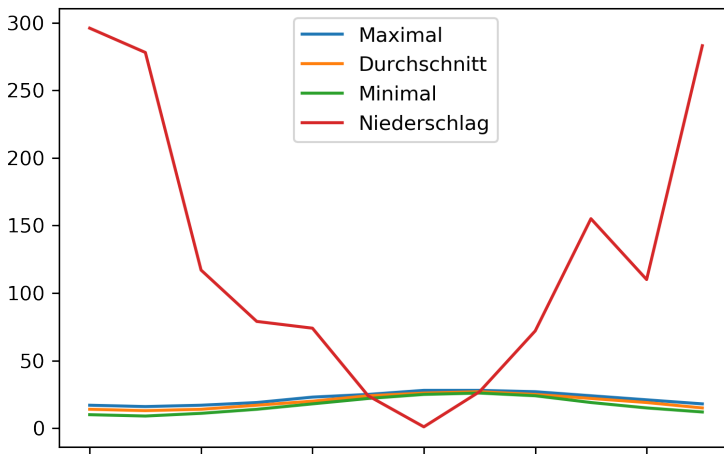
	Maximal	Durchschnitt	Minimal	Niederschlag
Januar	17	14	10	296
Februar	16	13	9	278
März	17	14	11	117
April	19	17	14	79
Mai	23	20	18	74

Im Folgenden erzeugen wir nun das Liniendiagramm für das eben erzeugte DataFrame:

```
kreta_df.plot()
```

Ausgabe:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4b79e5940>
```



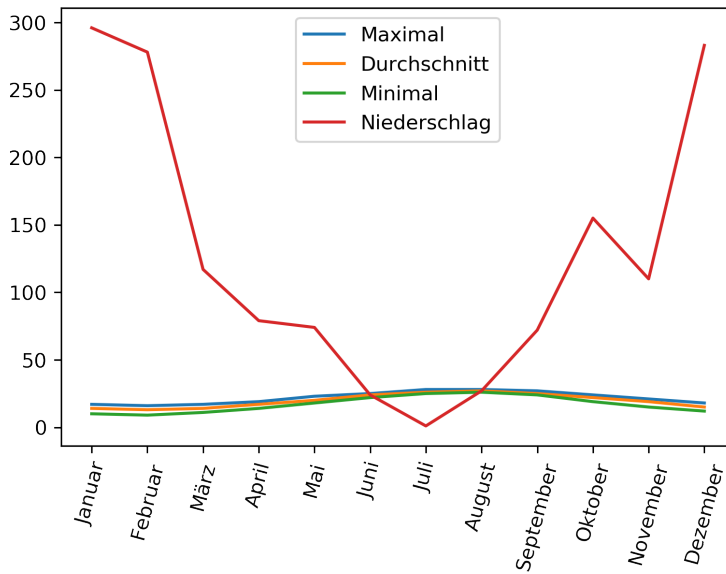
Mit der Beschriftung der Achsen können wir noch nicht zufrieden sein. Zum einen ist die X-Achse unbeschriftet, und zum anderen dient die Y-Achse sowohl zur Darstellung der Temperatur- als auch der Niederschlagswerte. Letzteres führt dazu, dass die Temperaturlinien kaum unterscheidbar sind, da die Y-Achse einen großen Wertebereich wegen der Niederschlagsmenge in „mm“ abdecken muss.

Zunächst möchten wir uns jedoch um die Beschriftung der X-Achse kümmern. Die X-Achse können wir mit den Monatsnamen versehen, indem wir explizit den Parameter `xticks` auf den Wert `range(len(kreta_df.index))` setzen, d.h. wir erhalten so viele Markierungen („ticks“) wie Monatsnamen. Der Parameter `rot` ist hierbei auch von besonderer Wichtigkeit. Mit ihm haben wir den Schrägdruck der Monatsnamen erreicht, d.h. wir haben sie bezogen auf die Horizontale um 75 Grad gedreht. Lässt man diesen Parameter weg oder setzt ihn auf Null, dann würden die Monatsnamen überlappend gedruckt und damit unlesbar.

```
kreta_df.plot(xticks=range(len(kreta_df.index)),
              rot=75)
```

Ausgabe:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4bc43e9b0>
```



24.2.3 Sekundärachsen (Twin Axes)

Wie wir bereits festgestellt haben, dient die Beschriftung der Y-Achse sowohl für die Temperatur- als auch für die Niederschlagswerte in mm. Deshalb wollen wir nun eine weitere vertikale Achse einführen, um die beiden Werte separat darzustellen.

Wir erzeugen zuerst die linke Achse, d.h. die Achse mit den Temperaturbeschriftungen. Dazu rufen wir die `plot`-Methode auf `kreta_df['Maximal']` auf und erhalten ein `AxesSubplot`-Objekt `ax` zurück, das wir in den folgenden Temperaturplots als Parameter benutzen. Gleichzeitig setzen wir mit dem ersten `plot` den Titel der Grafik mit `title='Jahresklima auf Kreta'`. Die Achse für die Niederschlagswerte erzeugen wir mittels `ax2 = ax.twinx()`

```
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()
```

```

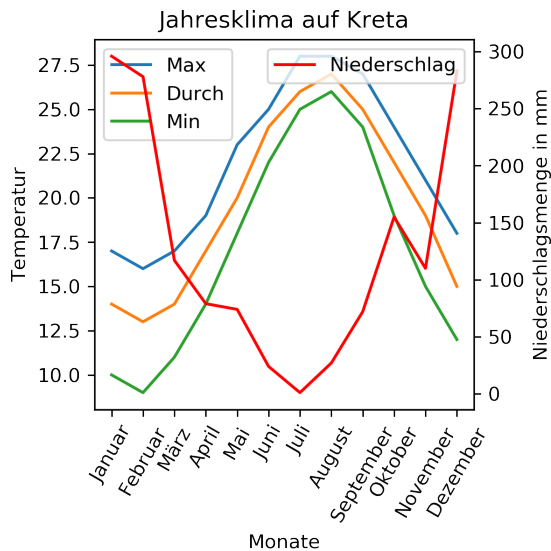
kreta_df['Niederschlag'].plot(ax=ax2,
                             style="r-")

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()

```



24.2.4 Mehrere Y-Achsen

Es lassen sich noch weitere Achsen hinzufügen.

Da wir in unserem DataFrame `kreta_df` keine weitere Spalte zum Plotten haben, fügen wir als Erstes eine neue Spalte mit den mittleren Sonnenstunden pro Tag ein:

```

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))

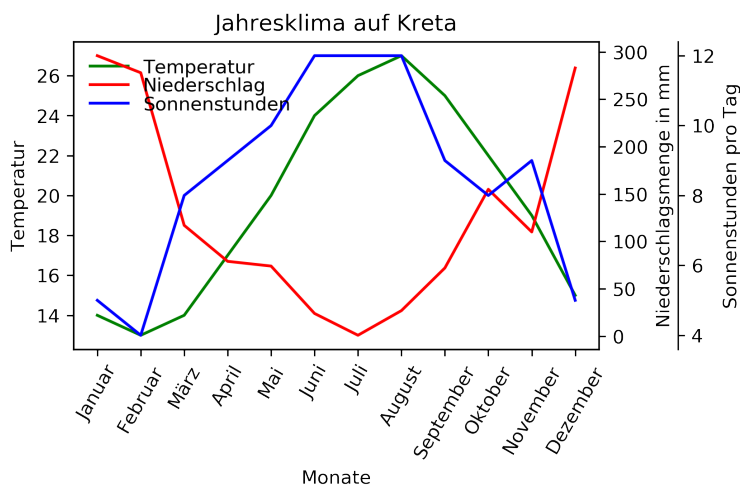
ax.legend(['Temperatur'],
          loc='upper left',
          bbox_to_anchor=(0.0, 1.0),
          frameon=False)
ax_regen.legend(loc='upper left',
               bbox_to_anchor=(0.0, 0.94),
               frameon=False)
ax_sonne.legend(loc='upper left',
               bbox_to_anchor=(0.0, 0.88),
               frameon=False)

#help(ax_sonne.legend)

#bbox_to_anchor=(1.5, 1)

plt.show()

```



■ 24.3 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()
```

Ausgabe:

	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]

```

Ausgabe:

	Visitors	Visits	Pages	Hits
Month				
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()

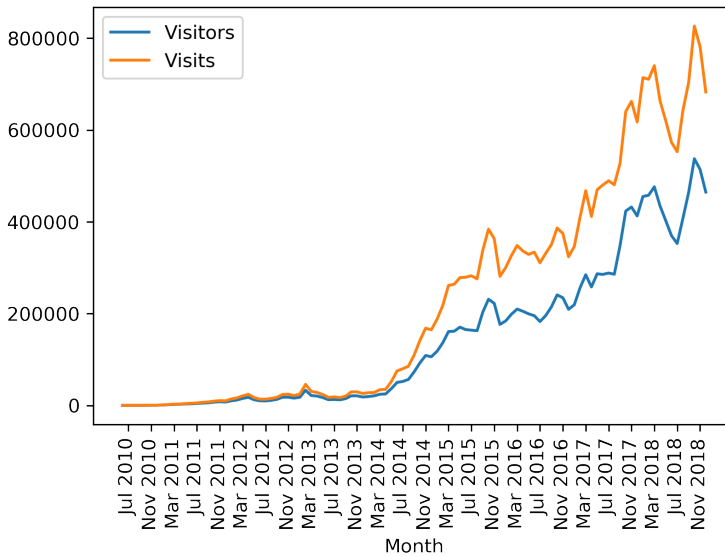
```

Ausgabe:

```

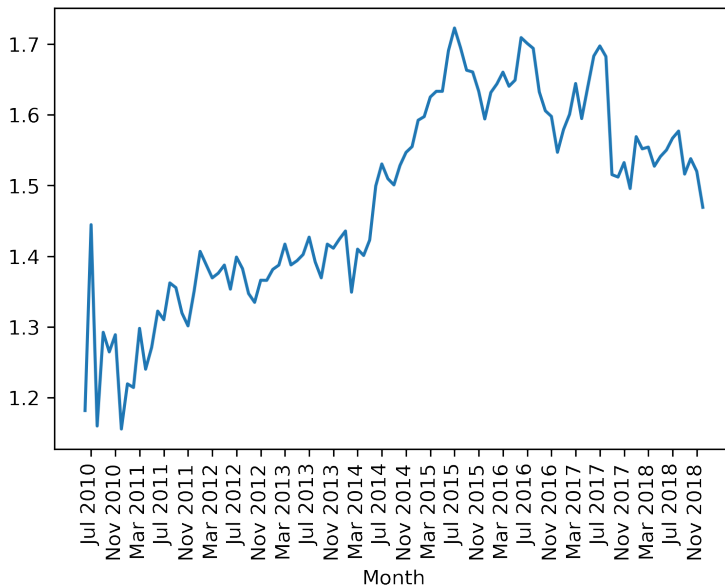
[]

```



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()
```



24.3.1 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 von TIOBE im November 2018 gesammelt und aufbereitet.

Die Datei hat folgenden Inhalt:

Die Prozentspalte enthält Strings mit dem Prozentzeichen. Das Zeichen können wir nutzen (oder loswerden), indem wir die Funktion `read_csv` verwenden. Alles, was wir dafür tun müssen, ist, eine Konverterfunktion zu definieren. Diese Konverterfunktion übergeben wir dann an `read_csv` mit dem Parameter `converters`:

```
def strip_percentage_sign(x):
    return float(x.strip('%'))

data_path = "data1/"
fname = "tiobe_programming_language_usage_nov2018.txt"
progs = pd.read_csv(data_path + fname,
                    quotechar='"',
                    thousands=",",
                    index_col=1,
                    converters={'Percentage': strip_percentage_sign},
                    delimiter=r"\s+")

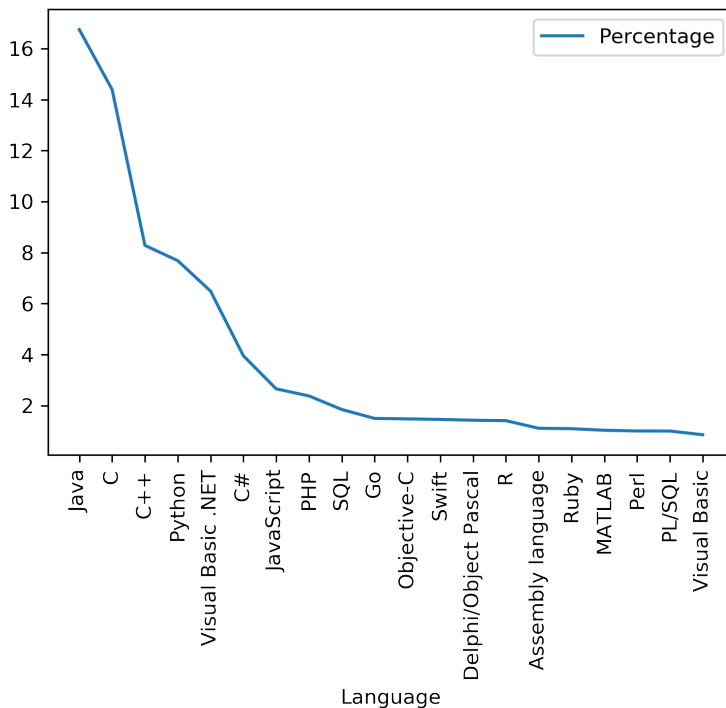
del progs["Position"]

print(progs)
progs.plot(xticks=range(len(progs.index)),
           use_index=True,
           rot=90)

plt.show()
```

Ausgabe:

	Percentage
Language	
Java	16.748
C	14.396
C++	8.282
Python	7.683
Visual Basic .NET	6.490
C#	3.952
JavaScript	2.655
PHP	2.376
SQL	1.844
Go	1.495
Objective-C	1.476
Swift	1.455
Delphi/Object Pascal	1.423
R	1.407
Assembly language	1.108
Ruby	1.091
MATLAB	1.030
Perl	1.001
PL/SQL	1.000
Visual Basic	0.854



■ 24.4 Balkendiagramme in Pandas

Balkendiagramme mit Pandas zu erstellen, ist ebenso einfach wie Liniendiagramme. Dazu fügen wir den Schlüsselwortparameter `kind` zur `plot`-Methode hinzu und setzen den Wert auf `bar`.

24.4.1 Ein einfaches Beispiel

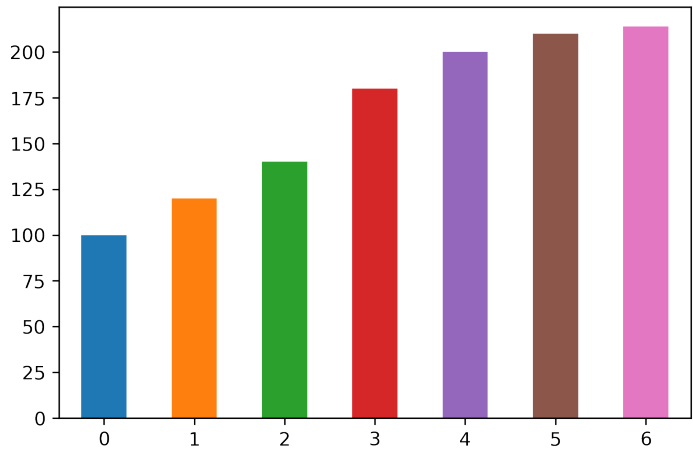
```
import pandas as pd

data = [100, 120, 140, 180, 200, 210, 214]
s = pd.Series(data, index=range(len(data)))

s.plot(kind="bar", rot=0)
plt.plot()
```

Ausgabe:

```
[]
```



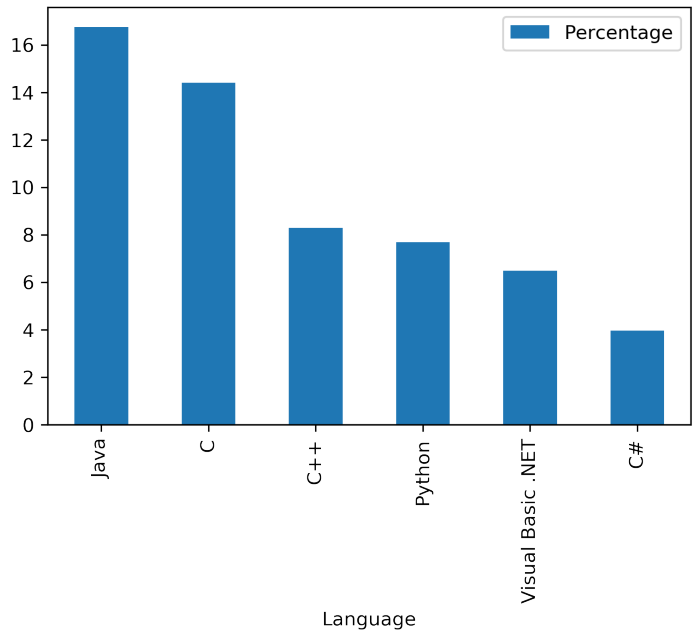
24.4.2 Balkengrafik für die Programmiersprachennutzung

Wir gehen zurück zum Beispiel des Programmiersprachen-Rankings. Jetzt generieren wir eine Balkengrafik der sechs meistverwendeten Programmiersprachen:

```
progs[:6].plot(kind="bar")
```

Ausgabe:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4b80e1898>
```

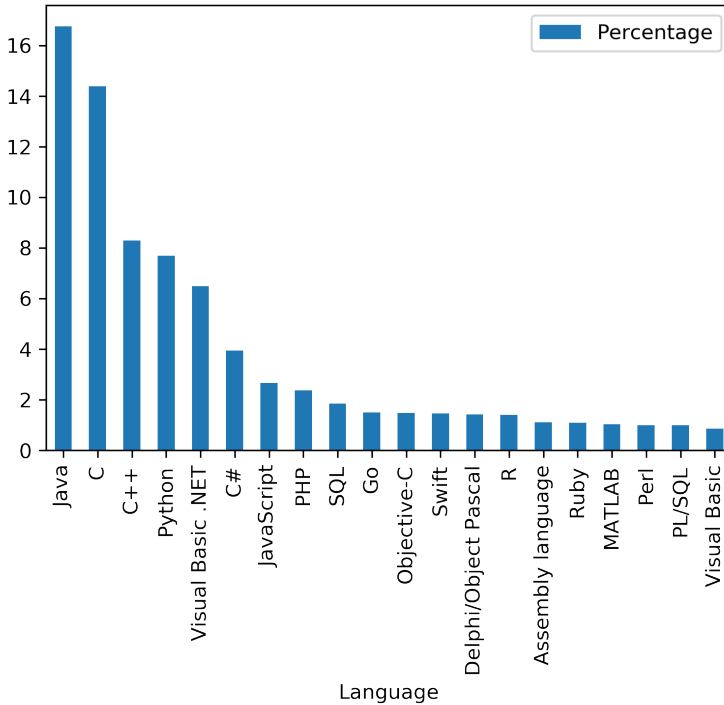


Nun die Balkendiagramme mit allen Programmiersprachen:

```
progs.plot(kind="bar")
```

Ausgabe:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4b80ec710>
```



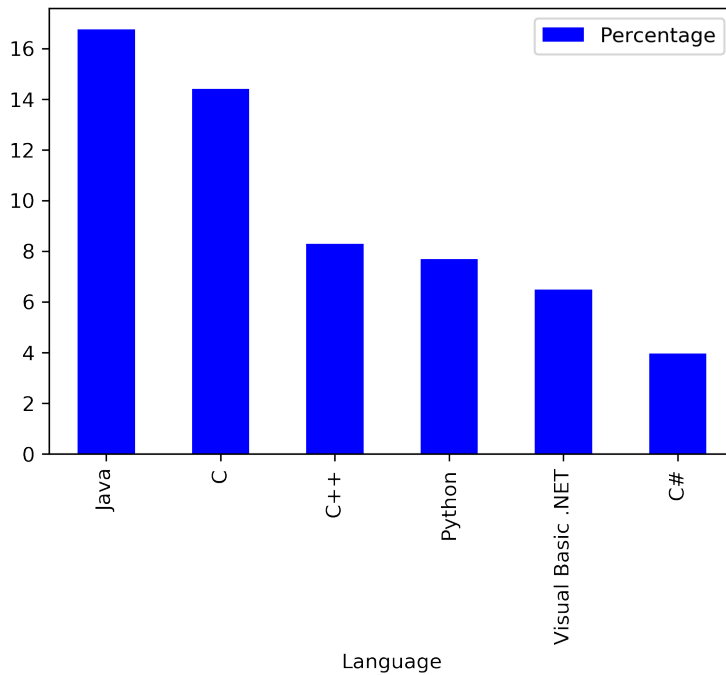
24.4.3 Farbgebung einer Balkengrafik

Es ist möglich, die Balken individuell zu färben, indem eine Liste dem Schlüsselwortparameter `color` zugewiesen wird:

```
my_colors = ['b', 'r', 'c', 'y', 'g', 'm']
progs[:6].plot(kind="bar",
               color=my_colors)
```

Ausgabe:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4b7d22710>
```

■ 24.5 Kuchendiagramme in Pandas

24.5.1 Ein einfaches Beispiel

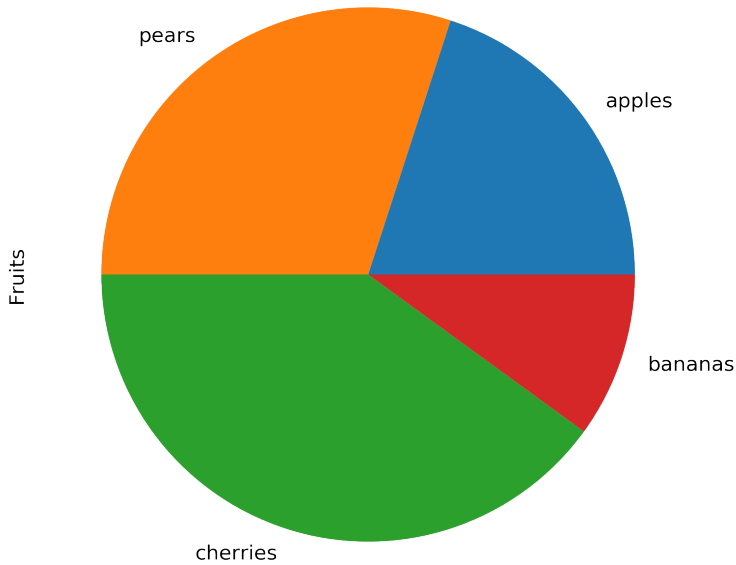
```
import pandas as pd

fruits = ['apples', 'pears', 'cherries', 'bananas']
series = pd.Series([20, 30, 40, 10],
                  index=fruits,
                  name='Fruits')

series.plot.pie(figsize=(6, 6))
```

Ausgabe:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4b7e2e0b8>
```



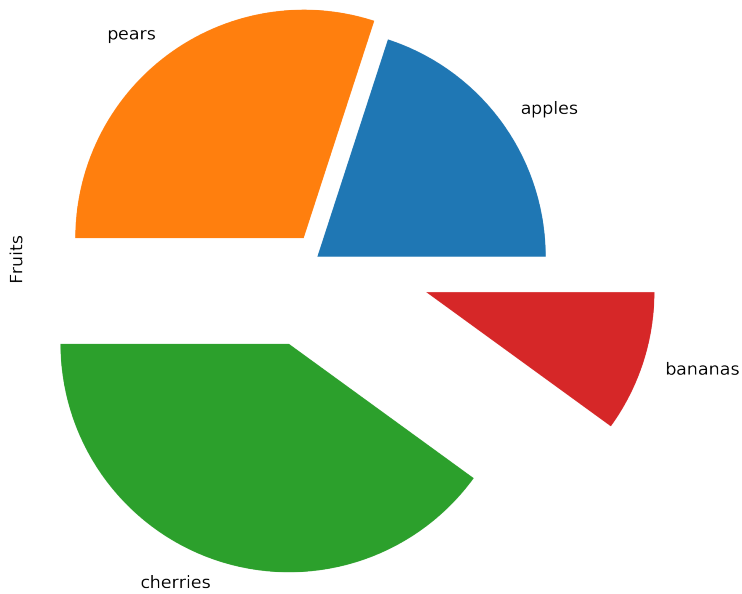
Obiges Kreisdiagramm lässt einen an einen Kuchen denken, und wie bei einem Kuchen kann man einzelne „Stücke“ herausziehen, was sich mit dem Parameter `explode` bewerkstelligen lässt. Diesem kann man eine Array-ähnliche Struktur, wie z.B. ein Tupel oder eine Liste, übergeben, die den Abstand vom Kreismittelpunkt 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)
```

Ausgabe:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4b816ed68>
```



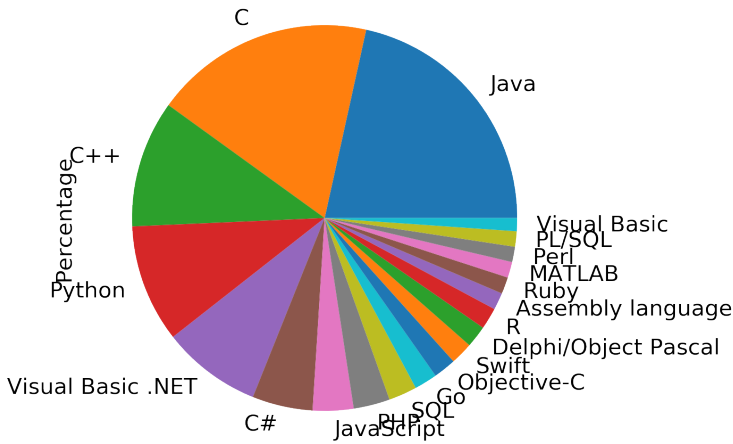
Wir generieren die vorherige Balkengrafik (Programmiersprachen) nun als Kuchendia-
gramm:

```
import matplotlib.pyplot as plt

my_colors = ['b', 'r', 'c', 'y', 'g', 'm']
progs.plot.pie(subplots=True,
               legend=False)
```

Ausgabe:

```
array([<matplotlib.axes._subplots.AxesSubplot object at
0x7ff4b7d8b898>], 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()`:

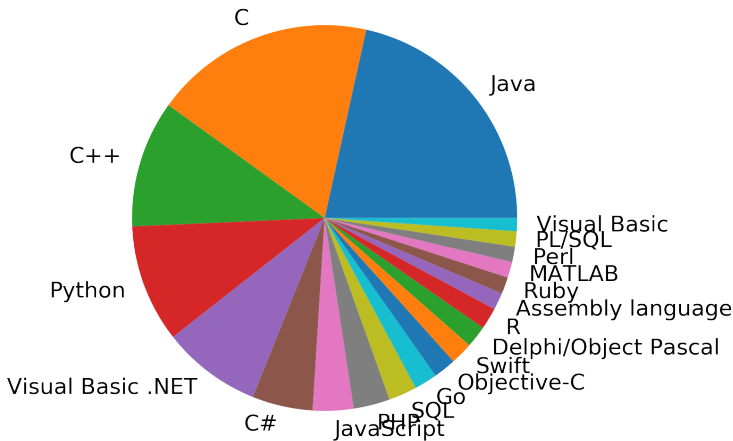
```
import matplotlib.pyplot as plt

my_colors = ['b', 'r', 'c', 'y', 'g', 'm']
progs.plot.pie(subplots=True,
               legend=False)

plt.ylabel('')
```

Ausgabe:

```
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

■ 25.1 Einführung

Egal in welcher Programmiersprache, sobald es um Kalenderdaten und Uhrzeiten geht, beginnen die Probleme. Die Probleme mit der Verarbeitung von Kalenderdaten rühren unter anderem daher, dass es auch in der schriftlichen Sprache allzu viele Varianten gibt, z.B. 24. Dezember 2018, 24. Dez. 2018, 24.12.2018, 24/12/2018, 12/24/18 usw. Eine Schreibweise, die viele wahrscheinlich gar nicht vermisst haben, besteht in der Form 2018-12-24. Eigentlich sollte dies die offiziell benutzte Schreibweise sein. Die Norm DIN 5008, die Schreib- und Gestaltungsregeln für die Textverarbeitung festlegt, besagt, dass für die numerische Schreibweise des Datums nur noch das internationale Datumsformat gemäß ISO 8601 zulässig sein soll, also JJJJ-MM-TT.¹

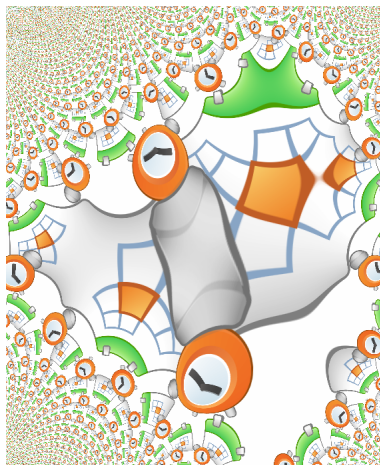


Bild 25.1 Datum und Zeiten

Python bietet reichlich Funktionalität, um mit Datums- und Zeitdaten umzugehen. Die Standardbibliotheken enthalten folgende Module:

- `time`
- `calendar`
- `datetime`

Diese Module bieten Klassen zur Manipulation von simplen und komplexen Datums- und Zeitdaten.

Speziell die `datetime`-Klasse ist sehr wichtig für die Time-Series in Pandas.

¹ Normenausschuss Informationstechnik und Anwendungen (NIA), Arbeitsausschuss NA 043-03-01 AA „Textverarbeitung“

■ 25.2 Python-Standardmodule für Zeitdaten

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 Zeitintervallen. 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 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-Objekte.

Wir starten mit dem `date`-Objekt.

25.2.1 Die `date`-Klasse

```
from datetime import date

x = date(1993, 12, 14)
print(x)
```

Ausgabe:

```
1993-12-14
```

Wir können Datums-Objekte zwischen dem 01. Januar 0001 und dem 31. Dezember 9999 instanziiieren. Über die Attribute `min` und `max` kann dies ermittelt werden:

```
from datetime import date

print(date.min)
print(date.max)
```

Ausgabe:

```
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())
```

Ausgabe:

```
1
2
2019-02-21
737111
```

Aus einem Ordinal kann das Datum mit der Klassenmethode `fromordinal` wieder herausgerechnet werden:

```
print(date.fromordinal(726952))
```

Ausgabe:

```
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())
```

Ausgabe:

```
1
print(date.today())
```

Ausgabe:

```
2019-02-21
```

Über die Attribute können wir auf den Tag, den Monat und das Jahr eines Date-Objekts zugreifen:

```
print(x.day)
print(x.month)
print(x.year)
```

Ausgabe:

```
2
1
1
```

25.2.2 Die time-Klasse

Die `time`-Klasse ist gleich organisiert wie die `date`-Klasse.

```
from datetime import time

t = time(15, 6, 23)
print(t)
```

Ausgabe:

```
15:06:23
```

Die möglichen Zeiten liegen zwischen:

```
print(time.min)
print(time.max)
```


Ausgabe:

```
00:00:00
23:59:59.999999
```

Der Zugriff auf Stunde, Minute und Sekunde:

```
print(t.hour, t.minute, t.second)
```

Ausgabe:

```
15 6 23
```

Jede Komponente eines Zeit-Objekts kann durch `replace()` geändert werden:

```
t = t.replace(hour=11, minute=59)
print(t)
```

Ausgabe:

```
11:59:23
```

Wir können einen Datums-String in C-Style generieren, entsprechend der `ctime`-Funktion in C:

```
print(x.ctime())
```

Ausgabe:

```
Tue Jan  2 00:00:00 0001
```

■ 25.3 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 Programms 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)
```

Ausgabe:

```
2017-04-19 16:31:00
```

t ist naive, weil folgender Ausdruck True ist:

```
print(t.tzinfo == None)
```

Ausgabe:

```
True
```

Wir erstellen ein 'aware' datetime-Objekt vom aktuellen Datum. Dafür benötigen wir das Modul pytz. pytz ist ein Modul, welches die 'Olsen-Zeitzone-Datenbank' in Python bereitstellt. Die Olsen-Zeitzone 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))
```

Ausgabe:

```
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
```

Ausgabe:

```
[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)]
```

■ 25.4 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))
```

Ausgabe:

```
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)
```

Ausgabe:

```
959
t1 = datetime(2017, 1, 31, 14, 17)
t2 = datetime(2015, 12, 15, 16, 59)
delta = t1 - t2
print(delta.days, delta.seconds)
```

Ausgabe:

```
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)
```

Ausgabe:

```
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)
```

Ausgabe:

```
1991-05-10 00:01:40
10 days, 0:01:40
```

25.4.1 Wandlung von datetime-Objekten in Strings

Der einfachste Weg, ein `datetime`-Objekt als String darzustellen, ist die `str`-Methode.

```
s = str(d1)
print(s)
```

Ausgabe:

```
1991-04-30 00:00:00
```

25.4.2 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.

```
print(d1.strftime('%Y-%m-%d'))
print("Wochentag: " + d1.strftime('%a'))
print("Wochentag geschrieben: " + d1.strftime('%A'))

# Weekday as a decimal number, where 0 is Sunday
# and 6 is Saturday
print("Wochentag als Dezimalzahl: " + d1.strftime('%w'))
```

Ausgabe:

```
1991-04-30
Wochentag: Tue
Wochentag geschrieben: 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'))
```

Ausgabe:

```
30
Apr
April
04
```

■ 25.5 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}!")
```

Ausgabe:

```
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}!")
```

Ausgabe:

```
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
```

■ 25.6 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 geparkt werden soll, und die Formatspezifikation.

```

from datetime import datetime
t = datetime.strptime("30 12 1999", "%d %m %Y")
print(t)

```

Ausgabe:

```

1999-12-30 00:00:00
dt = "2007-03-04T21:08:12"
datetime.strptime( dt, "%Y-%m-%dT%H:%M:%S" )

```

Ausgabe:

```

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

```

Ausgabe:

```
datetime.datetime(1957, 12, 24, 4, 3, 29)
```

Auf einer Linux-Maschine können wir mit dem Kommando `LC_ALL=en_EN.utf8 date` einen englischen Datumsstring generieren:

```

dt = 'Wed Apr 12 20:29:53 CEST 2017'
dt = datetime.strptime(dt, '%a %b %d %H:%M:%S %Z %Y')
print(dt)

```

Ausgabe:

```
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')
```

Ausgabe:

```
datetime.datetime(2011, 1, 3, 0, 0)
parse('Wed Apr 12 20:29:53 CEST 2017')
```

Ausgabe:

```
datetime.datetime(2017, 4, 12, 20, 29, 53, tzinfo=tzlocal())
```

■ 26.1 Einführung

In diesem Kapitel geht es nun um Zeitreihen, d.h. die Time-Series des Pandas-Moduls. Dabei handelt es sich eigentlich nur um eine spezielle Variante des Series-Datentyps, den wir bereits ausgiebig behandelt haben. Mit diesem Datentyp lassen sich auch Zeitreihen repräsentieren. Generell versteht man unter einer Zeitreihe eine Reihe oder Folge von Datenpunkten, welche in chronologischer Reihenfolge angeordnet sind. Für gewöhnlich sind die Abstände zwischen den Werten einer Zeitreihe äquidistant.

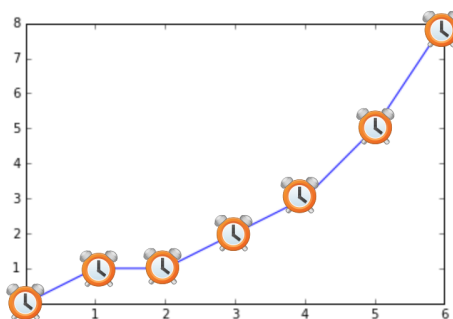


Bild 26.1 Zeitreihen

Alle gemessenen Daten, die jeweils mit einem bestimmten Zeitpunkt in Verbindung stehen, können als Zeitreihe angesehen werden. Messungen können durchaus unregelmäßig sein, haben aber in den meisten Fällen eine feste Frequenz bzw. Regelmäßigkeit. Daten können beispielsweise alle 5 Millisekunden, alle 10 Sekunden oder jede Stunde erhoben werden. Zur Darstellung von Zeitreihen werden häufig Linien-Charts verwendet.

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.

■ 26.2 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)

```

Ausgabe:

```

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

```

Wir ermitteln den Typ der soeben erstellten Time-Series:

```
print(type(ts))
```

Ausgabe:

```
<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)
```

Ausgabe:

```

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)
```

Ausgabe:

```

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)
```

Ausgabe:

```

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)

```

Ausgabe:

```

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

```

■ 26.3 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)
```

Ausgabe:

```
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 Enddatum an die `date_range`-Methode übergeben. Ebenso ist es möglich, nur einen Start oder nur ein Ende zu übergeben. In diesem Fall muss jedoch über den Schlüsselwortparameter `periods` die Anzahl der Perioden angegeben werden:

```
index = pd.date_range(start='12/24/1970', periods=7)
print(index)
```

Ausgabe:

```
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)
```

Ausgabe:

```
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)
```

Ausgabe:

```
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="M")
print(index)
```

Ausgabe:

```
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)
```

Ausgabe:

```
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')
```


Stichwortverzeichnis

A

accumulate 115
Achsenbeschriftung
– ändern 186
Achsenbezeichnung 181
Achsenbezeichnungen 172
Achsenteilung 181
Achsenverschiebung 181
add_subplot 201
annotate 194
arange 51
Arrays
– Broadcasting 106
– concatenate 89
– Diagonal-Array 68
– Erzeugen mit Einsen 64
– Erzeugen mit Nullen 64
– eye 68
– flatten 86
– Identitäts-Array 67
– identity 67
– Indizierung 56
– Konkatenation 89
– kopieren 65
– Matrizenmultiplikation 98
– may_share_memory 60
– mehrdimensional 43
– numerische Operationen 95
– ones 64
– ones_like 64
– Operatoren 95
– ravel 86
– reshape 88
– Skalare 95
– Summenprodukte 104
– Teilbereichsoperator 57
– Vergleichsoperatoren 105
– zeros 64
– zeros_like 64

at 118

atmosphärische Störungen 136
Attribuierung 35
Attribute 35
Ausnahmebehandlung
– finally 30
Auswahlen 126

B

Balkendiagramme 241, 249
Binarisierung 151
Binning 299
bins 300
Boolesche Indizierung 151
Boolesche Masken 151
Boolesche Maskierung 151
Broadcasting 106

C

C-zusammenhängend 65
calendar-Modul 337
cartesian_choice 132, 133
C_CONTIGUOUS 65
choice 126
coercion 112
column_stack 91
concatenate 89
contour 230
contourf 230
count_nonzero 156
csv-Dateien lesen 157, 283, 284
csv-Dateien schreiben 157, 285

D

DataFrame 253, 263
– Erzeugung aus Series-Objekten 263
– Index ändern 267
– reindex 268
– Spalte in Index wandeln 269

- Spaltennamen 264
- Spaltenreihenfolge 267
- Zusammenhang zu Series 263
- DataFrames
 - Sortierung 273
 - Spalten einfügen 274
 - Zeilenselektion 270
 - Zugriff auf Spalten 265
 - Zusammenhang mit mehrstufig indizierten Series 310

Datei

- lesen 30
- öffnen 30
- schreiben 30
- Datenkonvertierung beim Einlesen 160
- datetime 337
- datetime-Modul 340
- datetime-Objekt in String wandeln 343
- datetime.timedelta 342
- date.toordinal 338
- dateutil 346
- dateutil.parser 346
- Datum 337
- DIN 5008 337
- Schreibweisen 337
- Datum in Landessprache 344
- Datumsbereiche erstellen 350
- Datumsdifferenzen 342
- Diagonal-Array 68

Dimension

- Konkatenation 85
- Reduktion 85
- Summation 85
- Dimensionsänderungen 85
- DIN 5008 337

Diagonal-Array 68

Dimension

- Konkatenation 85
- Reduktion 85
- Summation 85
- Dimensionsänderungen 85
- DIN 5008 337

DIN 5008 337

Distanzmatrix 111

dot-Produkt 98

dropna 260

dsv-Dateien 283

dsv-Dateien lesen 284

dtype 52, 73

dtypes

- Spaltennamen umbenennen 79

dyadisches Produkt 117

E

- echte Zufallszahlen 135
- Einfärben von Flächen 178
- Eingabeaufforderung
 - robust 27
- erzwungene Typumwandlung 112

Excel

- openpyxl 287
- xldr 287
- Excel-Dateien lesen 287
- Excel-Dateien schreiben 287
- Excel-Tabelle 254
- Exceptions
 - eigene generieren 29
- eye 68

F

- Fancy-Indizierung 153
- F_CONTIGUOUS 65
- figure 206
 - figsize 206
- fill_between 178
- fillna 261
- Filtern fehlender Daten 260
- Finalisierungsaktion 29
- finally 29, 30
- find_interval 127
- flatnonzero 156
- flatten 86
- Fortran-zusammenhängend 65
- fully qualified 32

G

- Gauss'sche Normalverteilung 137
- gca 181
- genfromtxt 77, 164
- Gestalt eines Arrays 54
- getsizeof 46
- gezinkter Würfel 128
- Gitterlinien 218
- gnuplot 167
- gridspec 199, 208

H

- hist 242
- Histogramme 241

I

- Identitäts-Array 67
- identity 67
- import-Anweisung 32
- Indizierung
 - mehrstufig 307
- insert
 - DataFrames 274
- Instanzattribut 36
- Instanzen 35
- Instanziierung 35

IntervallIndex 301

isnull 259

Isolinie 223

K

Kalenderdaten 337

Kartesische Auswahl 132

Kartesisches Produkt 132

Klasse 35

– erzeugen 35

Klassenattribut 36

Kommentare 189

Konkatenation

– Pandas 263

Konturen

– gefüllt 228

Konturlinie 223

Konturplot 223, 225

– colors 227

– Farben ändern 227

– individuelle Farben 229

– linestyle 227

– Linienstil 227

– Schwellen 230

Konvertierung von Daten beim Einlesen 160

Kopieren von Arrays 65

Kuchendiagramme

– Pandas 332

L

Legende

– positionieren 190

Legenden 189

Lesen von Dateien mit NumPy 157

Liniendiagramm

– Pandas 318

Liniendiagramm in Pandas 320

Linienstil 169

Linienstil ändern 177

linspace 52

linstyle 177

linewidth 177

loadtxt 78, 159

locale-Modul 344

locale.setlocale 344

logarithmische Darstellung 216

Lotto-Ziehung 131

M

Maschengitter 223

Maskierung 151

math-Modul 32

matplotlib 43, 45, 167

– Beispiel Temperaturwerte 45

– Einfacher Plot 45

Matrizen 43, 73

Matrizenmultiplikation 98

may_share_memory 60

Mehrdimensionale Arrays 54

mehrdimensionale Arrays 43

Mehrstufige Indizes

– Vertauschen 314

mehrstufige Indizierung 307

Mengen

– add 17

Mengenprodukt 132

meshgrid 223, 231

mgrid 231, 235

Module

– eigene Module schreiben 33

– Inhalt 33

– Namensraum 32

– Suchpfad 33

N

Namensraum

– umbenennen 33

NaN 255, 258, 291

– Filtern fehlender Daten 260

– in Dateien 292

– math-Modul 291

– Ursprung 291

– Zusammenhang zu None 260

ndarray.strides 66

newaxis 90

nonzero 154

Norm DIN 5008 337

Normalverteilte Zufallszahlen 137

Normalverteilung 137

notnull 259

NumPy 43

– Akronym 43

– Array-Indizierung 56

– Array-Teilbereichsoperator 57

– Beispiel Dreidimensionales Array 61

– Eindimensionale Arrays 53

– Erzeugen eines einfachen NumPy-Arrays 44

– Erzeugung äquidistanter Intervalle 51

– Erzeugung von Arrays 51

– Gestalt 54

– Matrizen vs. Zweidimensionale Arrays 73

– may_share_memory 60

– Mehrdimensionale Arrays 54

– Nulldimensionale Arrays 53

- `numpy.ndarray` 53
- Seicherbedarf Arrays 46
- `Shape` 54
- Zeitvergleich zw. Python-Listen und NumPy-Arrays 49
- Zweidimensionale Arrays 54

O

- `ogrid` 231, 235
- Olsen-Zeitzone 341
- `open()` 30
- `openpyxl` 287
- `outer` 117

P

- Pandas 43, 253
 - Balkendiagramme 329
 - Dateien einlesen 283
 - Dateien schreiben 283
 - Konkatenation 263
 - Kuchendiagramme 332
 - Liniendiagramm 318, 320
 - Time-Series 347
- Panel data 253
- Parzen von Datums- und Zeitstrings 345
- Plot
 - Achsenbezeichnungen 172
 - `annotate` 194
 - Kommentare 189, 194
 - Legende 189
 - Legende positionieren 190
 - logarithmische Darstellung 216
 - mehrere in einem Diagramm 199
 - Plotbereiche setzen 215
 - speichern 218
- `plot` 169
- `plot-Funktion` 168
- `plt`
 - Alias für `matplotlib.pyplot` 168
 - `xlabel` 172
 - `ylabel` 172
- Population 131
- proleptischer Gregorianischer Kalender 338
- proleptisches Gregorianisches Ordinal 338
- Pseudozufallszahlen 122, 136
- `pyplot` 168
- `pyplot.plot` 169

R

- `randint` 124
- random-Modul 122
- `random_sample` 130

- `random.SystemRandom` 122
- `ravel` 86
- `read_csv` 284
- record arrays 73
- `reduce` 116
- `reshape` 88
- `row_stack` 91

S

- Säulendiagramm 247
- Säulendiagramme 241
- `savefig` 218
- `savetxt` 77, 157
- Schleifen 20
 - `for` 20
 - Schleifenkörper 20
 - `while` 20
- Schreiben von Dateien mit NumPy 157
- SciPy 43
- secrets-Modul 122
- secrets.SystemRandom 122
- Seed 136
- Seed-Key 136
- Sekundäre Y-Achse 216
- Series 253, 254
 - `apply` 257
 - `dropna` 260
 - Erstellung aus Dictionary 258
 - `fillna` 261
 - Filterung mit Booleschem Array 256
 - Indizierung 256
 - `isnull` 259
 - Konkatenation 263
 - `notnull` 259
 - Vergleich mit NumPy 254
 - Zusammenhang mit Dictionaries 258
- `set` 17
- `setlocale` 344
- sets
 - `add` 17
 - Operationen auf sets 17
- `set_xscale` 216
- `set_yscale` 216
- Shape eines Arrays 54
- Skalenteilung 181
- Spaltennamen
 - DataFrame 264
- Spaltenreihenfolge 267
- Spaltenzugriff
 - DataFrames 265
- spines 181
- `squeeze` 102

stack 312
Startwert einer Zufallsfolge 136
Statistik 121
Stichproben 126, 130
strftime 343
Strichproben 131
strides 66
strptime 345
strukturierte Arrays 73, 75
– Ein- und Ausgabe 77
Subplot
– add_subplot 201
subplot 199
Summenprodukte 104
Synthetische Verkaufszahlen 141

T

Teilbereichsoperator 13, 57
– Listen 13
– NumPy-Arrays 57
tensorielles Produkt 117
Tesserakt 85
tile 92
time-Klasse des datetime-Moduls 339
time-Modul 337
Time-Series 347
timedelta 342
timeit 49
Timer 49
to_csv 285
tofile 161
toordinal 338
transpose 155
type coercion 112

U

Ufuncs 112
– accumulate 115
– at 118
– outer 117
– reduce 116
– Rückgabewert 114
Uhrzeiten 337
universelle Funktionen 112
unstack 311
Unterdiagramm 212
urandom 122

V

Variable
– freie 25
– globale 25
– lokale 25
Vertauschen mehrstufiger Indizes 314

W

Wahrscheinlichkeit 121
weighted_choice 127, 128
where 154
Würfel, gezinkt 128
Würfelsimulation 126

X

xlabel 172
xldr 287
xticks 182
– Beschriftung ändern 184
– get_xticklabels 187
– Schriftgröße verändern 187

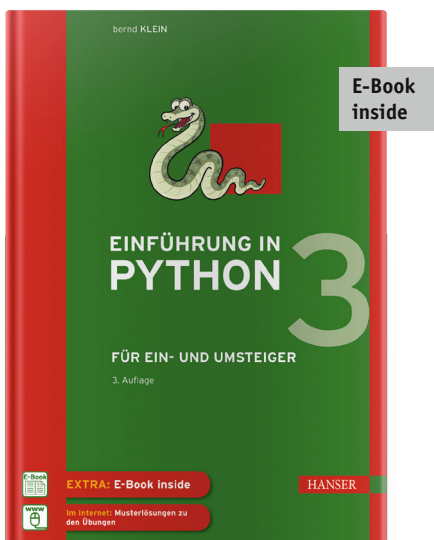
Y

Y-Achse
– sekundäre 216
ylabel 172
yticks 182

Z

Zeilenselektion
– DataFrame 270
Zeitdifferenzen 342
Zeitmessung 49
Zeitreihen 347
– arithmetische Operationen 348
Zeitreihen 347
Zufallsintervalle 127
Zufallsstichproben 130
Zufallswert 136
Zufallszahlen 122, 136
– echte 135
– kryptografisch stark 122
– normalverteilt 137
Zufallszahlengenerator 136
zusammenhängend 65
Zweidimensionale Arrays 54

Die Schlange bändigen



Klein

**Einführung in Python 3
Für Ein- und Umsteiger**

3., überarbeitete Auflage

555 Seiten. Inklusive E-Book

€ 25,-. ISBN 978-3-446-45208-4

Auch einzeln als E-Book erhältlich

€ 19,99. E-Book-ISBN 978-3-446-45387-6

Als idealer Einstieg für Programmieranfänger wie für Umsteiger behandelt dieses Buch alle grundlegenden Sprachelemente von Python 3. Aber auch Python-Kennern bietet das Buch viele weiterführende Themen wie Systemprogrammierung, Threads, Forks, Ausnahmehandlungen und Modultests.

- Besonders geeignet für Programmieranfänger, aber auch für Umsteiger von anderen Sprachen wie C, C++, Java oder Perl
- Systematische und praxisnahe Einführung in die Kunst der Programmierung
- Praxisnahe Übungen mit ausführlich dokumentierten Musterlösungen

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

Python als erste Programmiersprache



Woyand

Python für Ingenieure und Naturwissenschaftler
Einführung in die Programmierung, mathematische
Anwendungen und Visualisierungen

2., überarbeitete und erweiterte Auflage

302 Seiten. Inklusive E-Book

€ 26,-. ISBN 978-3-446-45792-8

Auch einzeln als E-Book erhältlich

Dieses Buch bietet Ihnen einen Einstieg in die Programmierung und mathematische Anwendungen mit Python. Es eignet sich besonders für Studierende im Nebenfach Informatik, z. B. Ingenieure, und setzt keine Vorkenntnisse voraus. Schwerpunkte des Buches sind die mathematischen Anwendungen sowie die Arbeit mit Numpy, Matplotlib, SYMPY und VPython.

Die zweite Auflage des Buches enthält eine Einführung in die Python-basierte Softwarebibliothek Scipy, mit deren Hilfe Probleme der Ingenieur- und Naturwissenschaften gelöst werden können. Beispiele zur numerischen Integration, Interpolation, Signalanalyse sowie Optimierung verdeutlichen den Einsatz.

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

Interdisziplinäres Know-how



Frochte

Maschinelles Lernen

Grundlagen und Algorithmen in Python

406 Seiten, 146 Abb. 22 Tab. Inklusive E-Book

€ 38,-. ISBN 978-3-446-45291-6

Auch einzeln als E-Book erhältlich

- Liefert den Hintergrund, um zu verstehen, wie und warum welche Algorithmen eingesetzt werden
- Anschauliche Umsetzung der Algorithmen mittels NumPy und SciPy
- Inklusive Kickstart in Python 3
- Geht auf Methoden des überwachten, unüberwachten und bestärkenden Lernens ein, u. a. Random Forest, DBSCAN und Q-Learning
- Für die Support Vector Machines und das Deep Learning wird auf scikit-learn bzw. Keras zurückgegriffen

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

Für alle, die weiterkommen wollen



Lo Iacono, Wiefling, Schneider

Programmieren trainieren

Mit über 120 Workouts in Java und Python

576 Seiten. Inklusive E-Book

€ 30,-. ISBN 978-3-446-45486-6

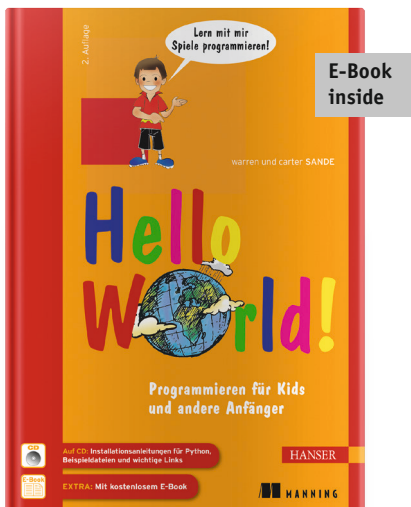
Auch einzeln als E-Book erhältlich

In diesem Übungsbuch trainierst du anhand von kurzweiligen und praxisnahen Aufgaben deine Programmierfähigkeiten. Jedes Kapitel beginnt mit einem kurzen Warmup zum behandelten Programmierkonzept; die Umsetzung übst du dann anhand von zahlreichen Workout-Aufgaben. So lernst du z. B. einen BMI-Rechner oder einen PIN-Generator zu programmieren oder wie du eine Zeitangabe mit einer analogen Uhr anzeigen kannst.

Die kommentierten Lösungen liegen in den Programmiersprachen Java und Python vor. Für ein möglichst ballastfreies Training wird für die elementaren Programmierkonzepte die Entwicklungsumgebung Processing eingesetzt.

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

Programmieren kinderleicht



Sande, Sande

Hello World!

Programmieren für Kids und andere Anfänger

2., aktualisierte und erweiterte Auflage

501 Seiten. Mit CD. Komplett in Farbe.

Inklusive E-Book

€ 29,99. ISBN 978-3-446-43806-4

Auch einzeln als E-Book erhältlich

- Alle Erklärungen der Konzepte in einfacher Sprache
- Sehr viele Bilder, Cartoons und lustige Beispiele
- Umfassende Fragen und Aufgaben zum Üben und Lernen
- Nutzt die kostenlos verfügbare Programmiersprache Python
- Von Vater und Sohn geschrieben und von Pädagogen geprüft – garantiert Kind-gerecht
- Auch für erwachsene Anfänger ohne Vorkenntnisse geeignet

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

NUMERISCHES PYTHON //

- Grundlagen der Lösung numerischer Probleme mit Python
- Verarbeitung großer Datenmengen (»Big Data«) mit NumPy, wie sie beispielsweise im maschinellen Lernen Anwendung finden
- Zielgruppe sind Personen, die in der Wissenschaft, im Ingenieurwesen und in der Datenanalyse tätig sind
- Datenvisualisierung mit Matplotlib
- Ideal zum Umstieg von Matlab auf Python

In diesem Buch stehen die numerischen Verfahren im Fokus, die im Gebiet »Data Science« und »Maschinelles Lernen« besonders benötigt werden. Python gehört zu den wichtigsten und am häufigsten benutzten Sprachen in diesem Gebiet und wird in Kombination mit seinen Modulen NumPy, SciPy, Matplotlib und Pandas häufiger verwendet als Matlab und R.

Der erste Teil des Buchs enthält eine kompakte Einführung in Python, eine ideale Zusammenfassung für diejenigen, die Python bereits kennen oder mit dem Buch »Einführung in Python 3« von Bernd Klein gelernt haben.

NumPy ist das zentrale Thema des zweiten Teils. Der Aufbau und das Arbeiten mit NumPy-Arrays bilden den Ausgangspunkt dieses Kapitels. Danach wird auf die besonderen Aspekte des dtype-Datentyps eingegangen. In einem weiteren Kapitel stehen die Numerischen Operationen, Broadcasting und Ufuncs von NumPy im Mittelpunkt. Einigen Fragestellungen der Statistik und der Wahrscheinlichkeitsrechnung wurde ebenfalls ein Kapitel gewidmet. Auch auf die Boolesche Maskierung und Indizierung von NumPy-Arrays wird eingegangen. Der NumPy-Teil des Buchs schließt mit dem File-Handling von Daten.

Der Diplom-Informatiker **Bernd KLEIN** genießt internationales Ansehen als Python-Dozent. Bisher hat er über 350 Python-Kurse in Firmen und Forschungsinstituten in Deutschland, Frankreich, der Schweiz, Österreich, den Niederlanden, Luxemburg, Rumänien und Kanada durchgeführt. Er ist Gründer und Inhaber des Schulungsanbieters Bodenseo. Besondere Anerkennung findet er wegen seiner Python-Webseiten www.pythonkurs.eu und www.python-course.eu, die jährlich über 6 Millionen Besucher verzeichnen. Seit 2016 ist er Lehrbeauftragter für Python und maschinelles Lernen an der Universität Freiburg.

AUS DEM INHALT //

- NumPy
 - Numerische Operationen auf mehrdimensionalen Arrays
 - Broadcasting
 - Ufuncs
- Matplotlib
 - Diskrete und kontinuierliche Graphen
 - Balken- und Säulendiagramme
 - Histogramme
 - Konturplots
- Pandas
 - Series
 - DataFrames
 - Lesen, Schreiben und Bearbeiten von Excel- und csv-Dateien
 - Umgang mit unvollständigen Daten
 - Datenvisualisierung
 - Zeitserien

HANSER

