E-BOOK inklusive

3. Auflage

Bernd KLEIN

Numerisches PYTHON

Arbeiten mit **NumPy**, **Matplotlib** und **Pandas**

HANSER

Klein

Numerisches Python

Bleiben Sie auf dem Laufenden!

Der Hanser Computerbuch-Newsletter informiert Sie regelmäßig über neue Bücher und Termine aus den verschiedenen Bereichen der IT. 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

3., aktualisierte Auflage

HANSER

Print-ISBN: 978-3-446-48549-5
E-Book-ISBN: 978-3-446-48584-6
ePub-ISBN: 978-3-446-48606-5

Die allgemein verwendeten Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

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

Vilshofener Straße 10 | 81679 München | info@hanser.de
www.hanser-fachbuch.de
Lektorat: Brigitte Bauer-Schiewek, Kristin Rothe
Herstellung: Grazyna Lada
Coverkonzept: Marc Müller-Bremer, *www.rebranding.de*, München
Covergestaltung: Thomas West
Titelmotiv: © stock.adobe.com/ARTvektor
Satz: le-tex publishing services GmbH, Leipzig
Druck: Elanders Waiblingen GmbH, Waiblingen
Printed in Germany

Inhalt

Vorwort XVII

Danksagung XVIII

1 Einleitung 1

1.1 Die richtige Wahl 1

1.2 Aufbau des Buches 2

1.3 Dieses Buch und die Werkzeuge dahinter 3

1.4 Download der Beispiele 3

1.5 Über den Autor 4

1.6 Anregungen und Kritik 4

2 Numerisches Programmieren 5

2.1 Überblick 5

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

2.3 Python – eine Alternative zu MATLAB 7

3 Installation von NumPy, Matplotlib, Pandas und JupyterLab 9

3.1 Einleitung 9

3.2 Installation mit `conda` und Miniconda 10

3.3 Installation mit `pip` 11

3.4 Starten von JupyterLab 11

3.5 Warum JupyterLab? 12

Teil I NumPy 13

4 NumPy Einführung 15
4.1 Überblick 15
4.1.1 Was ist NumPy? 15
4.1.2 Ein einfaches Beispiel 16
4.2 Vergleich NumPy-Datenstrukturen und Listen 17
4.2.1 Zentrale Unterschiede 17
4.2.2 Speicherbedarf 18
4.2.3 Zeitvergleich zwischen Listen und NumPy-Arrays 21

5 Erzeugung und Struktur von Arrays 23
5.1 Dimensionen 23
5.1.1 Nulldimensionale Arrays in NumPy 23
5.1.2 Eindimensionales Array 24
5.1.3 Zwei- und mehrdimensionale Arrays 24
5.2 Gestalt eines Arrays 25
5.3 Indizierung und Teilbereichsoperator 26
5.4 Dreidimensionale Arrays 33
5.5 Array-Erzeugungsfunktionen 35
5.5.1 arange 36
5.5.2 linspace 38
5.6 Arrays mit Nullen und Einsen 39
5.7 Einheitsmatrix 41
5.7.1 Die identity-Funktion 41
5.7.2 Die eye-Funktion 42
5.8 Datentypen 43
5.9 Arrays kopieren 45
5.9.1 numpy.copy(A) und A.copy() 45
5.9.2 Zusammenhängend gespeicherte Arrays 46
5.10 Aufgaben 48

6 Datentyp-Objekt: dtype 51
6.1 dtype 51
6.2 Strukturierte Arrays 53

6.3 Ein- und Ausgabe von strukturierten Arrays ... 56
6.4 Unicode-Strings in Arrays ... 58
6.5 Umbenennen von Spaltennamen ... 59
6.6 Spaltenwerte austauschen ... 59
6.7 Komplexeres Beispiel ... 60
6.8 Aufgaben ... 62

7 Arrays kombinieren und umformen ... 63
7.1 Reduktion und Umformung von Arrays ... 64
7.1.1 flatten ... 64
7.1.2 ravel ... 65
7.1.3 Unterschiede zwischen `ravel` und `flatten` ... 65
7.1.4 reshape ... 66
7.2 Hinzufügen von Dimensionen ... 68
7.3 Konkatenation und Stapelung von Arrays ... 69
7.3.1 `concatenate` ... 69
7.3.2 `stack` ... 72
7.3.3 `dstack` ... 74
7.3.4 `vstack` ... 78
7.3.5 `hstack` ... 79
7.4 `dsplit` ... 81
7.5 Arrays wiederholen mit `tile` ... 82
7.6 Aufgaben ... 85

8 Numerische Operationen auf NumPy-Arrays ... 87
8.1 Operationen mit Skalaren ... 87
8.2 Operationen zwischen und auf Arrays ... 89
8.3 Matrizenmultiplikation und Skalarprodukt ... 90
8.3.1 Definition der dot-Funktion ... 90
8.3.2 Beispiele zur dot-Funktion ... 91
8.3.3 Das dot-Produkt im dreidimensionalen Fall ... 92
8.4 Vergleichsoperatoren ... 97
8.5 Logische Operatoren ... 98
8.6 Broadcasting ... 99

8.6.1 Zeilenweises Broadcasting 100
8.6.2 Spaltenweises Broadcasting 103
8.6.3 Broadcasting von zwei eindimensionalen Arrays 106
8.7 Distanzmatrix 106
8.8 ufuncs 108
8.8.1 Anwendung von ufuncs 108
8.8.2 Parameter für Rückgabewerte bei ufuncs 110
8.8.3 accumulate 113
8.8.4 reduce 115
8.8.5 outer 116
8.8.6 at 117
8.9 Aufgaben 117

9 Statistik und Wahrscheinlichkeiten 119
9.1 Einführung 119
9.2 Auf dem random-Modul aufbauende Funktionen 120
9.2.1 Echte Zufallszahlen 121
9.2.2 Erzeugen einer Liste von Zufallszahlen 122
9.2.3 Zufällige Integer-Zahlen 123
9.2.4 Stichproben oder Auswahlen 124
9.2.5 Zufallsintervalle 125
9.2.6 Seed oder Startwert 125
9.2.7 Gewichtete Zufallsauswahl 127
9.2.8 Stichproben mit Python 129
9.2.9 Kartesische Auswahl 132
9.2.10 Kartesisches Produkt 132
9.2.11 Kartesische Auswahl: cartesian_choice 132
9.2.12 Gauss'sche Normalverteilung 135
9.2.13 Übung mit Binärsender 138
9.3 Das random-Untermodul von NumPy 141
9.3.1 Integers und Floats zufällig erzeugen 141
9.3.2 numpy.random.choice 143
9.3.3 numpy.random.random_sample 145
9.4 Synthetische Verkaufszahlen 147
9.5 Aufgaben 148

10 Boolesche Maskierung und Indizierung ... 151
10.1 Fancy-Indizierung ... 153
10.2 Indizierung mit einem Integer-Array ... 154
10.3 nonzero und where ... 154
10.4 Beispielanwendungen mit `np.where` ... 156
10.5 Aufgaben ... 158

11 Lesen und Schreiben von Daten-Dateien ... 159
11.1 Textdateien speichern mit savetxt ... 160
11.2 Textdateien laden mit loadtxt ... 162
11.2.1 loadtxt ohne Parameter ... 162
11.2.2 Spezielle Trenner ... 162
11.2.3 Selektives Einlesen von Spalten ... 163
11.2.4 Datenkonvertierung beim Einlesen ... 163
11.3 tofile ... 165
11.4 fromfile ... 166
11.5 Empfohlene Methoden ... 168
11.6 Eine weitere Möglichkeit: `genfromtxt` ... 169

Teil II Matplotlib ... 171

12 Einführung ... 173
12.1 Ein erstes Beispiel ... 174
12.2 Format-Parameter von plot ... 175
12.3 Mehrere Datenreihen mit Achsentiteln ... 177

13 Objektorientiert plotten ... 179
13.1 Erzeugung einer Figure und Axes ... 181
13.2 Achsenbeschriftungen und Titel ... 183
13.3 Die Plot-Methode ... 184
13.4 Wertebereiche der Achsen ... 185
13.5 Plotten mehrerer Funktionen ... 188
13.6 Streudiagramme ... 189
13.7 Flächen einfärben ... 192
13.8 Aufgaben ... 195

14 Mehrfache Plots und Doppelachsen ... 197
14.1 Subplots mit `subplot` ... 198
14.2 Flexible Layouts mit GridSpec ... 206
14.3 Doppelachsen ... 213
14.4 Aufgaben ... 215

15 Achsen- und Skalenteilung ... 217
15.1 Achsen und Spines ... 217
15.2 Achsenbeschriftungen ändern ... 222
15.3 Justierung der Tick-Beschriftungen ... 223

16 Legenden und Annotationen ... 225
16.1 Legende hinzufügen ... 225
16.2 Annotations / Anmerkungen ... 229
16.3 Aufgaben ... 236

17 Konturplots ... 237
17.1 Erstellen eines Maschengitters ... 238
17.2 Funktionen auf Meshgrids ... 240
17.3 contour ohne meshgrid ... 242
17.4 Linienstil und Farben anpassen ... 242
17.5 Gefüllte Konturen ... 244
17.6 Individuelle Farben ... 245
17.7 Schwellen ... 246
17.8 Andere Grids ... 247
17.8.1 Meshgrid genauer ... 248
17.8.2 mgrid ... 250
17.8.3 ogrid ... 250
17.9 imshow ... 252
17.10 Aufgaben ... 253

18 Histogramme und Diagramme ... 255
18.1 Histogramme ... 256
18.2 Säulendiagramm ... 260
18.3 Balkendiagramme ... 262

18.4 Gruppierte Balkendiagramme ... 263
18.5 xkcd-Modus ... 267
18.6 Tortendiagramme ... 269
18.7 Stapeldiagramme ... 270
18.8 Aufgaben ... 271

Teil III Pandas ... 273

19 Pandas:Series ... 275
19.1 Grundlagen der Series-Datenstruktur ... 276
19.2 Zugriff und Indizierung ... 279
19.3 Wertmanipulation mit apply ... 280
19.4 Series aus Dictionaries ... 281
19.5 NaN – Fehlende Daten ... 282
19.5.1 Fehlende Werte prüfen ... 283
19.5.2 Zusammenhang zwischen NaN und None ... 284
19.5.3 Fehlende Daten filtern ... 285
19.5.4 Fehlende Daten auffüllen ... 285
19.5.5 Vergleich verschiedener Interpolationsmethoden ... 288
19.6 Aufgaben ... 289

20 DataFrame ... 291
20.1 Ein erstes Beispiel ... 292
20.2 Zusammenhang zu Series ... 293
20.3 Manipulation der Spaltennamen ... 294
20.4 DataFrames aus Dictionaries ... 295
20.5 Zugriff auf Spalten ... 298
20.6 Selektion von Zeilen ... 298
20.6.1 `loc` ... 298
20.6.2 query ... 300
20.7 Modifikation von DataFrames ... 302
20.7.1 Spalten einfügen ... 303
20.7.2 Spalten austauschen ... 307

20.7.3 Zeilen austauschen ... 308
20.7.4 Einzelne Werte mittels at und iat ändern ... 308
20.8 Index ändern ... 309
20.8.1 Umsortierung von Spalten und Index ... 310
20.8.2 Spalten umbenennen ... 312
20.8.3 Spalte als Index verwenden ... 312
20.9 Summen und kumulative Summen ... 313
20.9.1 Leere Spalten und nachträgliches Befüllen ... 315
20.10 Sortierung ... 317
20.11 Aufgaben ... 318

21 Styling ... 321
21.1 Einleitung ... 321
21.2 Daten und Präsentation trennen ... 322
21.3 Die `.style`-Eigenschaft ... 323
21.3.1 Grundlegende Formatierung mit `.format` ... 323
21.4 Maximalwerten in Zeilen und Spalten ... 324
21.5 Anwenden eines Farbverlaufs ... 325
21.5.1 Balkendiagramme innerhalb von Zellen anwenden ... 326
21.6 Aufgaben ... 328

22 Dateiverarbeitung ... 331
22.1 DSV- / CSV-Dateien ... 332
22.1.1 CSV- und DSV-Dateien lesen ... 332
22.1.2 Schreiben von CSV-Dateien ... 334
22.1.3 Einfaches Beispiel ... 334
22.1.4 Umfangreicheres Beispiel ... 336
22.1.5 Beispiel mit einer Nicht-Standard-csv-Datei ... 338
22.2 Lesen und Schreiben von JSON-Dateien ... 340
22.3 Lesen und Schreiben von Excel-Dateien ... 341
22.4 Aufgaben ... 342

23 Pandas: groupby ... 345
23.1 Groupby mit Series ... 346
23.2 Arbeitsweise von `groupby` ... 348

23.3 GroupBy mit DataFrames ... 350
23.3.1 GroupBy mit Funktion ... 352
23.3.2 Beispiel mit Datei ... 354
23.4 Aufgaben ... 355

24 Pivot-Tabellen ... 359
24.1 Pivot-Funktion in Pandas ... 360
24.2 Pivot-Aufruf ohne Werte für ‚values' ... 362
24.3 Die Funktion `pivot_table` in Pandas ... 363
24.4 Pivoting auf den Titanic-Daten ... 365
24.5 Aufgaben ... 368

25 Umgang mit NaN ... 369
25.1 'nan' in Python ... 369
25.2 NaN in Pandas ... 370
25.2.1 Beispiel mit NaNs ... 373
25.3 dropna() verwenden ... 375
25.4 Aufgaben ... 377

26 Binning ... 379
26.1 Einführung ... 379
26.2 Binning mit Pandas ... 380
26.2.1 Binning mit cut ... 380
26.2.2 Erzeugen eines IntervalIndex-Objektes ... 382
26.2.3 Mehr zu pd.cut ... 383
26.2.4 Speicheroptimierung mit `Categorical` ... 384
26.2.5 Binnings mit Labels ... 385
26.3 Aufgaben ... 386

27 Mehrstufige Indizierung ... 387
27.1 Einführung ... 387
27.2 Mehrstufig indizierte Series-Objekte ... 388
27.3 Mehrstufige Indizierung durch Listen-Multiplikation ... 390
27.4 Weitere Möglichkeiten zur Index-Erzeugung ... 391
27.5 Zugriffsmöglichkeiten ... 392

27.6 Dreistufige Indizes ... 396
27.7 Zusammenhang zu DataFrames ... 397
27.7.1 Manueller Weg mit `pd.concat` ... 398
27.7.2 unstack und stack ... 399
27.8 Vertauschen mehrstufiger Indizes ... 402
27.9 Aufgaben ... 404

28 Datenvisualisierung mit Pandas ... 407
28.1 Einführung ... 407
28.2 Liniendiagramm in Pandas ... 408
28.2.1 Series ... 408
28.2.2 DataFrames ... 410
28.2.3 Sekundärachsen (Twin Axes) ... 413
28.2.4 Mehrere Y-Achsen ... 414
28.2.5 Spalten mit Zeichenketten (Strings) in Floats wandeln ... 416
28.3 Balkendiagramme in Pandas ... 417
28.3.1 Ein einfaches Beispiel ... 417
28.3.2 Balkengrafik für die Programmiersprachennutzung ... 418
28.3.3 Farbgebung einer Balkengrafik ... 420
28.4 Tortendiagramme in Pandas ... 421
28.4.1 Ein einfaches Beispiel ... 421
28.5 Flächenplot mit area ... 423
28.6 Aufgaben ... 424

29 Zeit und Datum ... 425
29.1 Einführung ... 425
29.2 Python-Standardmodule für Zeitdaten ... 426
29.2.1 Die date-Klasse ... 426
29.2.2 Die time-Klasse ... 428
29.3 Die datetime-Klasse ... 429
29.4 Unterschied zwischen Zeiten ... 431
29.4.1 Wandlung von datetime-Objekten in Strings ... 432
29.4.2 Wandlung mit strftime ... 433
29.5 Ausgabe in Landessprache ... 434
29.6 datetime-Objekte aus Strings erstellen ... 435

30 Zeitreihen ... 437
30.1 Einführung ... 437
30.2 Zeitreihen und Python ... 438
30.3 Datumsbereiche erstellen ... 440
30.4 Datumsbereiche mit Uhrzeitangaben ... 443
30.5 Aufgaben ... 444

Teil IV Anwendungen ... 445

31 Techniken der Bildverarbeitung ... 447
31.1 Einführung ... 447
31.2 Bilder laden und anzeigen ... 448
31.3 Histogramme der Farbwerte ... 450
31.4 Bildausschnitte ... 452
31.5 Geometrische Transformationen ... 452
31.6 Filterung ... 454
31.7 Aufhellen und Abtönen von Bildern ... 459
31.8 Kachelung ... 467
31.9 Wasserzeichen mit `np.where` ... 468
31.10 Ein weiteres Beispiel für Wasserzeichen mit `np.where` ... 470
31.11 Aufgaben ... 473

32 Finanzverwaltung mit Pandas ... 475
32.1 Haushaltsbuch ... 475
32.1.1 Haushaltsbuch mit CSV-Datei ... 476
32.1.2 Excel-Haushaltsbuch mit Kontenplan ... 480
32.1.3 Auswertung des Excel-Haushaltsbuches ... 482
32.2 Einnahmeüberschussrechnung ... 483
32.2.1 Journaldatei ... 484
32.2.2 Analyse und Visualisierung der Daten ... 485
32.2.3 Steuersummen ... 490

Teil V Lösungen zu den Aufgaben 493

33 Lösungen zu den Aufgaben 495

33.1 Lösungen zu Kapitel 5 (Erzeugung und Struktur von Arrays) 495
33.2 Lösungen zu Kapitel 6 (Datentyp-Objekt: dtype) 497
33.3 Lösungen zu Kapitel 7 (Arrays kombinieren und umformen) 499
33.4 Lösungen zu Kapitel 8 (Numerische Operationen auf NumPy-Arrays) 502
33.5 Lösungen zu Kapitel 9 (Statistik und Wahrscheinlichkeiten) 505
33.6 Lösungen zu Kapitel 10 (Boolesche Maskierung und Indizierung) 510
33.7 Lösungen zu Kapitel 13 (Objektorientiert plotten) 512
33.8 Lösungen zu Kapitel 14 (Mehrfache Plots und Doppelachsen) 515
33.9 Lösungen zu Kapitel 16 (Legenden und Annotationen) 517
33.10 Lösungen zu Kapitel 17 (Konturplots) 519
33.11 Lösungen zu Kapitel 18 (Histogramme und Diagramme) 523
33.12 Lösungen zu Kapitel 19 (Pandas:Series) 527
33.13 Lösungen zu Kapitel 20 (DataFrame) 531
33.14 Lösungen zu Kapitel 21 (Styling) 536
33.15 Lösungen zu Kapitel 22 (Dateiverarbeitung) 538
33.16 Lösungen zu Kapitel 23 (Pandas: groupby) 541
33.17 Lösungen zu Kapitel 24 (Pivot-Tabellen) 547
33.18 Lösungen zu Kapitel 25 (Umgang mit NaN) 548
33.19 Lösungen zu Kapitel 26 (Binning) 549
33.20 Lösungen zu Kapitel 27 (Mehrstufige Indizierung) 550
33.21 Lösungen zu Kapitel 28 (Datenvisualisierung mit Pandas) 555
33.22 Lösungen zu Kapitel 30 (Zeitreihen) 557
33.23 Lösungen zu Kapitel 31 (Techniken der Bildverarbeitung) 558

Stichwortverzeichnis 561

Vorwort

Eine der treibenden Kräfte in der heutigen Softwareentwicklung lässt sich wohl am besten mit den Begriffen „Big Data“ und „Maschinelles Lernen“ beschreiben. Immer mehr Forschungseinrichtungen und Unternehmen engagieren sich in diesen Bereichen. Für sie – und auch für Einzelpersonen, die in diesen Feldern aktiv werden wollen – stellt sich eine zentrale Frage: Welche Programmiersprache eignet sich am besten für datengetriebene Anwendungen?

In nahezu allen Umfragen der letzten Jahre wird Python als eine der besten, wenn nicht als die beliebteste Programmiersprache genannt. Python hat sich in beeindruckendem Tempo zu einem unverzichtbaren Werkzeug in der Datenanalyse, im wissenschaftlichen Rechnen und in der künstlichen Intelligenz entwickelt.

Dabei war Python ursprünglich nicht für numerische Aufgaben konzipiert. Der heutige Erfolg basiert maßgeblich auf leistungsfähigen Erweiterungen wie `NumPy`, `SciPy`, `Matplotlib` und `Pandas`. Diese Module haben Python zu einer ernstzunehmenden Alternative zu spezialisierten Softwarepaketen gemacht – sowohl in der Forschung als auch in der Industrie.

Das vorliegende Buch bietet eine fundierte Einführung in die Arbeit mit `NumPy`, `Matplotlib` und `Pandas`. Es richtet sich an Leserinnen und Leser, die bereits grundlegende Python-Kenntnisse mitbringen – zum Beispiel aus dem Buch „Einführung in Python 3: Für Ein- und Umsteiger“ von Bernd Klein. Damit bildet es eine ideale Ergänzung für alle, die den nächsten Schritt in Richtung datenorientierter Programmierung gehen möchten.

Brigitte Bauer-Schiewek, Lektorin im September 2025

Danksagung

Zum Schreiben eines Buches braucht es nicht nur Erfahrung und Fachkompetenz, sondern vor allem eines: Zeit. Zeit jenseits des Gewohnten, frei von Ablenkung, mit Raum fürs Nachdenken, Ausprobieren und das kontinuierliche und geduldige Ringen um gute Lösungen … Zeit, die zwangsläufig von der Familie mitgetragen wird. Mein besonderer Dank gilt daher meiner Frau Karola, die mich während der gesamten Entstehung dieses Buches – von der ersten bis zur dritten Auflage – mit großem Verständnis und tatkräftiger Unterstützung begleitet hat.

Mein aufrichtiger Dank gilt auch dem Hanser Verlag, der die Veröffentlichung dieses Buches – nun bereits in der dritten Auflage – ermöglicht hat. Ganz besonders danke ich Frau Brigitte Bauer-Schiewek (Programmplanung Computerbuch) für die kontinuierlich hervorragende Zusammenarbeit sowie Kristin Rothe (Lektoratsassistenz Computerbuch) für ihre Unterstützung im Produktionsprozess. Ihr gilt darüber hinaus mein besonderer Dank für ihre große Geduld bei der aufwendigen Umstellung auf Barrierefreiheit sowie auf ein neues Fachbuchdesign. In diesem Zusammenhang betraten wir gemeinsam Neuland – ein wichtiger und zukunftsweisender Schritt.

Mein Dank gilt auch dem Textsatzsystem LaTeX[1], das es ermöglicht, Bücher in ansprechendem Layout und technisch präziser Form zu schreiben. Aber auch beim „tollsten“ Satzsystem gibt es gelegentlich knifflige Situationen, in denen mir Herr Stephan Korell und Frau Irene Weilhart mit ihrer technischen Unterstützung kompetent zur Seite standen. Ihnen beiden gilt mein besonderer Dank.

Was LaTeX jedoch noch faszinierender macht, ist die Möglichkeit, mit `PythonTeX`[2] die Stärken von LaTeX und Python zu verbinden: Rechenoperationen, Datenauswertungen oder Diagramme können automatisiert mit Python erzeugt und direkt sowie automatisch in das gesetzte Dokument eingebettet werden. Seit der zweiten Auflage dieses Buches nutze ich `PythonTeX`, um genau diese Möglichkeiten auszuschöpfen. Mein Dank gilt Herrn Tobias Habermann, der mich in der zweiten Auflage tatkräftig bei der Umstellung auf `PythonTeX` unterstützt hat.

1 LaTeX wurde von Leslie Lamport entwickelt und basiert auf dem von Donald E. Knuth geschaffenen System TeX. Beiden gilt mein großer Respekt für ihre herausragenden Beiträge zum wissenschaftlichen Publizieren.

2 `pythontex` ist ein LaTeX-Paket, mit dem sich Python-Code direkt im Dokument ausführen und dessen Ausgabe automatisch einfügen lässt – z. B. für Berechnungen, Tabellen oder Diagramme.

Ein herzliches Dankeschön geht auch an die zahlreichen Teilnehmerinnen und Teilnehmer meiner Python-Kurse. Durch ihre Rückfragen, ihr Feedback und ihre Anregungen konnte ich meine didaktischen und fachlichen Konzepte stetig weiterentwickeln und an den Bedürfnissen der Lernenden ausrichten. Ebenso danke ich den vielen Nutzerinnen und Nutzern meiner Online-Tutorials unter *www.python-kurs.eu* und *www.python-course.eu* – insbesondere jenen, die sich mit konstruktiven Hinweisen und Fragen direkt an mich gewandt haben.

Bernd Klein, Singen im September 2025

1 Einleitung

1.1 Die richtige Wahl

Sich für die passende Programmiersprache im Berufsalltag zu entscheiden, ist von großer Bedeutung. Diese Wahl hängt von vielen Faktoren ab – und nicht selten hat man gar keine echte Wahl: Häufig ist durch das Unternehmen, das Team oder das jeweilige Projekt bereits eine Sprache vorgegeben. Immer häufiger hat man jedoch das Glück, mit Python arbeiten zu dürfen.

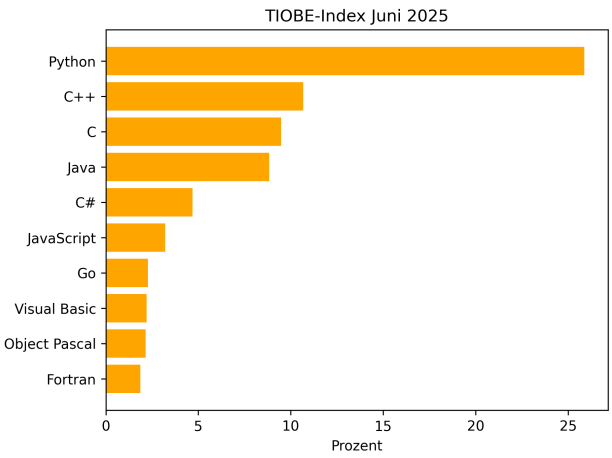


Bild 1.1 Top 10 der Programmiersprachen

Betrachtet man aktuelle Umfragen zu den beliebtesten Programmiersprachen, so findet sich Python fast immer auf den vorderen Plätzen – oft sogar ganz oben. Auch im TIOBE-Index[1)] liegt Python inzwischen deutlich vor allen anderen Sprachen. In der ersten Auflage dieses Buches

1 Der TIOBE-Index des niederländischen Unternehmens TIOBE Software BV ist ein seit 2001 monatlich publiziertes Ranking zur Popularität von Programmiersprachen. Der Rang ergibt sich aus der Häufigkeit von Suchanfragen mit ihrem Namen in Suchmaschinen wie Google, Bing oder Yahoo!. Der Index misst keine Codezeilen oder technische Qualität, sondern lediglich die Verbreitung der Sprache im Web.

im Jahr 2018 belegte Python noch Platz 3 im TIOBE-Ranking. Dennoch wurde Python schon damals in die TIOBE-Hall of Fame aufgenommen – als die Sprache mit dem größten Popularitätszuwachs des Jahres. Seit Oktober 2021 steht Python unangefochten auf Platz 1 im TIOBE-Index.

Natürlich ist es schön zu wissen, dass die eigene Sprache der Wahl auch bei vielen anderen beliebt ist – und womöglich sogar in der eigenen Branche führend eingesetzt wird. Aber eine der wichtigsten Fragen lautet: Lassen sich mit Python eigene Projekte einfacher und besser als mit anderen Programmiersprachen lösen? Unter „einfacher und besser“ verbergen sich natürlich Begriffe wie „Entwicklungszeit“, „Laufzeit“, „Wartbarkeit“ und so weiter.

Programmiersprachen sind wie Schuhe: Es gibt nicht den einen für alle Zwecke. Kein Schuh passt zugleich zu feierlichen Anlässen, ins Büro, zum Sport und beim Wandern. Python hingegen ist eine Sprache, die sich in den meisten Bereichen flexibel einsetzen lässt – einer der Gründe für ihren großen Erfolg. Aber vor allen Dingen verdankt Python seinen Siegeszug den leistungsfähigen Modulen NumPy, SciPy, Matplotlib und Pandas. Mit deren Hilfe lassen sich numerische Probleme dank der klaren und leicht verständlichen Syntax besonders einfach lösen. NumPy stellt darüber hinaus Datenstrukturen bereit, die um den Faktor 10 bis 100 schneller sind als Implementierungen in reinem Python oder vielen anderen Programmiersprachen. Da die Module größtenteils in C geschrieben sind, erreichen sie nahezu die Geschwindigkeit nativer C-Programme.

1.2 Aufbau des Buches

In diesem Buch geht es um Python und seine hervorragenden Möglichkeiten zum Einsatz bei numerischen Problemen – also um jene Module, die für Themen wie „Big Data“ und „Maschinelles Lernen“ mittlerweile unverzichtbar geworden sind. Der Fokus liegt auf der Anwendung der Bibliotheken `NumPy`, `Matplotlib` und `Pandas`.

Das Buch setzt grundlegende Kenntnisse in Python voraus, also an Lesende, die bereits erste Erfahrungen mit der Sprache gesammelt haben – etwa durch ein Einführungsbuch wie „Einführung in Python 3“[2].

Dieses Buch bietet eine praxisorientierte Einführung in die numerische Programmierung mit Python und ist in mehrere logisch aufeinander aufbauende Teile gegliedert:

Teil I: NumPy Beginnt mit den Grundlagen der numerischen Programmierung und zeigt, wie man mit `NumPy` effizient mit Arrays arbeitet. Themen wie Array-Erzeugung, Indexierung, Datentypen, mathematische Operationen, Broadcasting und statistische Auswertungen werden systematisch behandelt.

2 Bernd Klein, *Einführung in Python 3: Für Ein- und Umsteiger*, ISBN: 978-3-446-46379-0, 4. Auflage, 06/2021, 600 Seiten, fester Einband, komplett in Farbe.

Teil II: Matplotlib Vermittelt die Grundlagen der Datenvisualisierung mit `Matplotlib`. Von einfachen Plots bis hin zu komplexen Diagrammen mit mehreren Achsen oder Konturplots wird die Bandbreite der Darstellungsmöglichkeiten aufgezeigt.

Teil III: Pandas Führt in die Arbeit mit tabellarischen Daten ein. Die Datenstrukturen `Series` und `DataFrame`, der Umgang mit fehlenden Werten, Gruppierungen, Pivot-Tabellen sowie Zeit- und Datumsfunktionen stehen im Fokus. Ebenso wird gezeigt, wie Daten aus verschiedenen Quellen wie `CSV`-, `Excel`- oder `JSON`-Dateien gelesen und gespeichert werden können.

Teil IV: Anwendungen Zeigt konkrete Anwendungsfälle mit `pandas` – aus der Bildverarbeitung, der Finanzanalyse und der Analyse der Energiegewinnung in Deutschland.

Teil V: Lösungen Enthält die Lösungen zu den im Buch gestellten Übungsaufgaben zur Selbstkontrolle.

Ergänzt wird das Buch durch ein einführendes Kapitel zur Installation und Einrichtung der benötigten Bibliotheken.

1.3 Dieses Buch und die Werkzeuge dahinter

Dieses Buch wurde vollständig mit LaTeX und `PythonTeX` erstellt. Damit verbinden sich die typografischen Stärken von LaTeX – wie präzise Formatierung, konsistentes Layout, automatische Inhaltsverzeichnisse, Querverweise und Literaturverwaltung – mit den dynamischen Möglichkeiten von Python. Durch `PythonTeX` kann Python-Code direkt in das Dokument eingebettet und bei der Compilierung ausgeführt werden. Ergebnisse wie Rechenausgaben, Tabellen, Diagramme oder interaktive Inhalte erscheinen automatisch an der richtigen Stelle im Buch.

Dank Syntaxhervorhebung, reproduzierbarer Codeausführung und direkter Integration in den Text entsteht ein didaktisch besonders transparenter und konsistenter Workflow – ideal für ein Buch über Datenanalyse und Visualisierung mit Python.

1.4 Download der Beispiele

Alle im Buch verwendeten Beispiele befinden sich zum Download unter

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

Dort findet sich auch ein Korrekturverzeichnis.

1.5 Über den Autor

Bernd Klein schloss 1988 sein Studium der Informatik mit dem Diplom an der Universität des Saarlandes ab. Bis 2007 war er als Softwareentwickler in der Industrie tätig. Seitdem ist er international als Dozent und Trainer für Softwareentwicklung und Programmiersprachen aktiv – mit Schwerpunkt auf der Programmiersprache Python seit 2009.

Er arbeitet mit Universitäten, Forschungseinrichtungen und Unternehmen im In- und Ausland zusammen. Seit 2008 betreibt er die Online-Lernplattformen *https://python-kurs.eu* und *https://python-course.eu*, die jährlich von Millionen Lernenden weltweit genutzt werden.

Veröffentlichte Bücher:

- *Einführung in Python 3*. 4. Auflage, Hanser Verlag, München 2023.
  ISBN 978-3-446-46379-0
- *Python-Grundlagen | eLearning*. 1. Auflage, Hanser Verlag, 2023.
  ISBN 978-3-446-47992-0
- *Numerisches Python*. 2. Auflage, Hanser Verlag, 2022.
  ISBN 978-3-446-47170-2
- *Funktionale Programmierung mit Python*. 1. Auflage, Hanser Verlag, 2025.
  ISBN 978-3-446-48191-6

1.6 Anregungen und Kritik

Bei Hinweisen auf Ungenauigkeiten oder Fehler im Buch kann gerne eine E-Mail direkt an den Autor geschickt werden: `bernd.klein@python-kurs.eu`.

Dies gilt natürlich auch für Anregungen, Anmerkungen oder Wünsche zum Buch.

Fehler und Anmerkungen werden wir in kommenden Auflagen berücksichtigen.

Ich wünsche allen beim Lesen ebenso viel Freude, wie ich sie beim Schreiben hatte. Viel Spaß beim Ausprobieren, Programmieren und Entdecken!

Bernd Klein, September 2025

2
Numerisches Programmieren

2.1 Überblick

Bild 2.1 Analoge Datenanalyse im Büro

Der Titel dieses Buches lautet „Numerisches Python“, in Anlehnung an den Begriff „numerisches Programmieren“ – ein im Alltag häufig unscharf und manchmal auch missverständlich verwendeter Begriff. Man könnte annehmen, es ginge um jede Programmierung, die mit Zahlen arbeitet – was praktisch auf fast alle Programme zutreffen würde. Selbst Anwendungen, die scheinbar rein textbasiert sind, wie z. B. Suchmaschinenalgorithmen, enthalten im Kern numerische Verfahren. So basiert etwa der ursprüngliche PageRank-Algorithmus von Google auf der Berechnung einer extrem großen Matrix mit Milliarden von Zeilen und Spalten.[1)]

1 Die enorme Größenordnung der im PageRank-Algorithmus verwendeten Matrix wird in einer Vorlesung der Cornell University veranschaulicht: „Aus mathematischer Sicht ist es – zumindest theoretisch – eine einfache Aufgabe, die Eigenvektoren zum Eigenwert 1 zu berechnen, sobald man die Matrix M hat. Wie in Vorlesung 1: Man löst einfach das Gleichungssystem $Ax = x$! Aber wenn die Matrix M eine Größe von 30 Milliarden hat (wie im echten Web-Graphen), sind selbst Programme wie MATLAB oder Mathematica damit eindeutig überfordert.“ Originalzitat: “From the mathematical point of view, once we have M, computing the eigenvectors corresponding to the eigenvalue 1 is, at least in theory, a straightforward task. As in Lecture 1, just solve the system Ax = x! But when the matrix M has size 30 billion (as it does for the real Web graph), even mathematical software such as Matlab or Mathematica are clearly overwhelmed.” Quelle: *https://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html*

Unter numerischer Programmierung versteht man die rechnergestützte Lösung mathematischer Probleme durch numerische Verfahren wie beispielsweise das Lösen linearer und nichtlinearer Gleichungssysteme, numerische Integration, Matrixoperationen, statistische Berechnungen oder Optimierungsprobleme, also Methoden, die sich in nahezu allen naturwissenschaftlichen und technischen Disziplinen wiederfinden. Numerisches Programmieren ist somit ein Teilbereich des wissenschaftlichen Programmierens. Letzteres umfasst sämtliche Arten von Softwareentwicklung im Forschungs- und Analysekontext, z. B. auch Visualisierung, Simulation oder Datenaufbereitung.

Ziel dieses Buches ist es, die zentralen Werkzeuge vorzustellen, die zur Umsetzung numerischer Verfahren in den Bereichen Data Science, Statistik und maschinelles Lernen mit Python benötigt werden. Im Mittelpunkt stehen praxisnahe und effiziente Bibliotheken wie NumPy, Pandas und Matplotlib, die die Grundlage für datenintensive Anwendungen bilden.

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

Python ist eine universelle Programmiersprache, die in verschiedensten Bereichen eingesetzt wird – z. B. in der Systemadministration, im Web Development, in der Computerlinguistik und, wie schon mehrfach erwähnt, in der numerischen Programmierung, wo Geschwindigkeit und Speicherverbrauch entscheidend sind. Reines Python – also ohne den Einsatz optimierter Bibliotheken – ist für aufwendige numerische Aufgaben nicht konkurrenzfähig zu spezialisierten Werkzeugen wie MATLAB oder R. Die Leistungsfähigkeit der eingesetzten Algorithmen ist bei numerischen Problemen von höchster Bedeutung. Deshalb stützt sich Python auf seine leistungsstarken Module `NumPy`, `SciPy`, `Matplotlib` und `Pandas`. Dadurch gehört Python heute zu den führenden Sprachen im Bereich der numerischen Programmierung – und ist dabei oft effizienter als MATLAB oder R.

- **NumPy** stellt die grundlegenden Datenstrukturen bereit, insbesondere mehrdimensionale Arrays (ndarrays) und Matrizen. Es enthält elementare Funktionen zur Erzeugung, Manipulation und Auswertung dieser Strukturen und bildet die Basis für viele andere Pakete.
- **SciPy** baut auf NumPy auf und ergänzt es durch umfangreiche Funktionalitäten aus der wissenschaftlichen Mathematik wie numerische Integration, Interpolation, lineare Algebra, Optimierung und Fourier-Transformation.
- **Matplotlib** ermöglicht die grafische Darstellung von Daten, wie sie in vielen wissenschaftlichen Kontexten nötig ist. Es unterstützt einfache Diagramme ebenso

wie komplexe Visualisierungen mit mehreren Achsen, Subplots oder interaktiven Elementen.

- **Pandas** ist auf die Arbeit mit tabellarischen Daten (DataFrames) spezialisiert. Es bietet leistungsfähige Werkzeuge für Zeitreihenanalyse, Gruppierungen, Pivot-Tabellen, Umgang mit fehlenden Werten und vieles mehr. Darüber hinaus erlaubt Pandas das Einlesen und Schreiben vieler gängiger Formate wie CSV, Excel oder JSON.

2.3 Python – eine Alternative zu MATLAB

Python entwickelt sich zunehmend zur bevorzugten Programmiersprache von Data Scientists, Analysten und wissenschaftlichen Programmiererinnen und Programmierern. Während früher Werkzeuge wie MATLAB und R in Forschung, Technik und Statistik dominierten, verlagert sich der Schwerpunkt heute zunehmend auf Python. Diese Entwicklung ist auf eine Vielzahl von Faktoren zurückzuführen – unter anderem auf die hohe Flexibilität, eine sehr aktive Community, die modulare Erweiterbarkeit sowie die Tatsache, dass Python als Open-Source-Software kostenlos verfügbar ist.

MATLAB wurde ursprünglich für numerische Berechnungen im technischen und ingenieurwissenschaftlichen Bereich entwickelt. Es bietet eine spezialisierte Umgebung mit zahlreichen Funktionen für lineare Algebra, Signalverarbeitung, Systemsimulation und Optimierung. Die Sprache ist in sich geschlossen, proprietär und basiert auf einem kostenpflichtigen Lizenzmodell. Für viele wissenschaftliche Einrichtungen, Studierende oder Start-ups können die Lizenzkosten – insbesondere für spezialisierte Toolboxes – eine erhebliche Hürde darstellen.

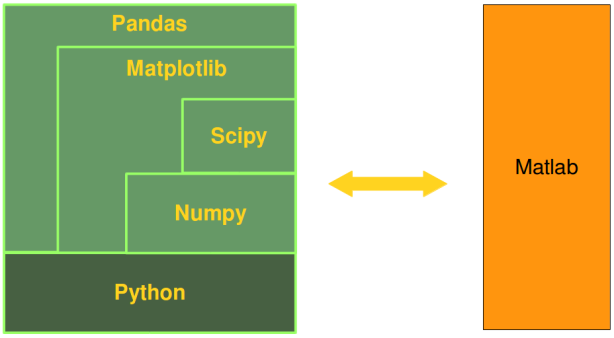


Bild 2.2 Zusammenhang zu MATLAB

Python hingegen wurde von Beginn an als universelle Programmiersprache konzipiert. Mit Bibliotheken wie `NumPy`, `SciPy`, `Matplotlib`, `Pandas`, `SymPy` oder `scikit-learn` kann Python heute nahezu alle Aufgaben abdecken, die zuvor exklusiv MATLAB vorbehalten waren – und darüber hinaus viele weitere, etwa in den Bereichen Webentwicklung, Automatisierung oder Machine Learning.

Ein wesentlicher Vorteil von Python liegt auch in der nahtlosen Integration moderner Entwicklungsumgebungen wie Jupyter Notebooks. Diese Umgebung erlaubt eine Kombination aus Code, Visualisierung und erklärendem Text – ein ideales Format für explorative Datenanalyse, wissenschaftliches Rechnen und die Lehre.

Ein weiterer Pluspunkt ist die Offenheit des Ökosystems: Python erlaubt einfache Anbindungen an C/C++-Bibliotheken, Fortran-Routinen oder externe Tools. Es existieren zudem viele Möglichkeiten für paralleles und verteiltes Rechnen (z. B. mit `Dask` oder `joblib`) sowie für GPU-beschleunigtes Rechnen (z. B. `CuPy`, `PyTorch`, `TensorFlow`).

Nicht zuletzt profitiert Python von seiner enormen Nutzerbasis. Die offene Entwicklungskultur führt zu einer schnellen Verbesserung der Werkzeuge, umfangreicher Dokumentation, Tutorials, Konferenzen und einer Vielzahl an frei verfügbaren Ressourcen.

3
Installation von NumPy, Matplotlib, Pandas und JupyterLab

3.1 Einleitung

In diesem Kapitel wird beschrieben, wie die für die folgenden Kapitel benötigten Bibliotheken NumPy, Matplotlib, Pandas sowie JupyterLab installiert werden können.

JupyterLab ist die moderne, browserbasierte Oberfläche für interaktive Python-Notebooks und wird im weiteren Verlauf dieses Buches verwendet, da sie sich besonders für das Experimentieren mit Code, Datenanalyse und Visualisierungen eignet.

Bild 3.1 Speed Up!

Wir stellen zwei Methoden vor:

- Installation mit `pip`, dem Standard-Paketmanager von Python
- Installation mit `conda`, dem Paketmanager von Anaconda und Miniconda

Beide Methoden haben ihre Vorzüge: Während `pip` direkt mit jeder Python-Installation genutzt werden kann, erleichtert `conda` das Management von Abhängigkeiten und stellt optimierte Versionen der Pakete bereit. Je nach System und Anforderungen kann die bevorzugte Methode gewählt werden.

Wichtiger Hinweis zu Anaconda: Seit 2020 ist sie jedoch für kommerzielle Nutzung nicht mehr uneingeschränkt kostenlos.[1] Unternehmen und Organisationen mit 200 oder mehr Mitarbeitenden benötigen eine kostenpflichtige Lizenz. Für Privatpersonen, Studierende, Lehrkräfte sowie kleinere Unternehmen und nichtkommerzielle akademische Einrichtungen bleibt die Nutzung hingegen weiterhin kostenlos – insbesondere zu rein privaten Lern- und Studienzwecken.[2] Die zugrunde liegenden Python-Pakete in Anaconda (wie `NumPy`, `Pandas`, `Matplotlib` u. a.) sind Open Source und grundsätzlich kostenlos verfügbar. Die Lizenzgebühren beziehen sich nicht auf diese Software selbst, sondern auf die kommerzielle Bereitstellung, Wartung und Integration durch Anaconda Inc. sowie ggf. auf den zugehörigen Support. Wer auf Nummer sicher gehen möchte, kann alternativ auf die frei verfügbare Miniconda oder eine individuell konfigurierte Python-Umgebung mit `pip` zurückgreifen.

Miniconda ist eine schlankere Variante von Anaconda, die ebenfalls `conda` als Paketmanager enthält, jedoch keine vorinstallierten Pakete mitbringt. Dabei ist zu beachten, dass der Standard-Channel („defaults“) von Anaconda Inc. betrieben wird. Organisationen, die unter die kommerzielle Lizenzregelung fallen, sollten daher auf alternative Paketquellen wie conda-forge ausweichen, um mögliche Lizenzkosten zu vermeiden.[3]

3.2 Installation mit conda und Miniconda

Falls die Anaconda- oder Miniconda-Distribution verwendet wird, steht der Paketmanager `conda` zur Verfügung. Dieser Paket- und Umgebungsmanager ermöglicht die einfache Installation und Verwaltung zahlreicher wissenschaftlicher Python-Pakete. Benutzt man Anaconda, sind die meisten der benötigten Pakete bereits installiert und müssen nicht wie im Folgenden gezeigt manuell installiert werden. Bei Miniconda erfolgt die Installation der benötigten Pakete mit:

```
conda install numpy matplotlib pandas jupyterlab
# oder alternativ, Community-basiert und lizenzfrei:
conda install -c conda-forge numpy matplotlib pandas jupyterlab
```

Ein Vorteil von `conda` ist, dass es automatisch alle Abhängigkeiten berücksichtigt und in vielen Fällen optimierte, vorcompilierte Versionen der Pakete bereitstellt – insbesondere für numerisch anspruchsvolle Anwendungen.

1 Siehe Lizenzinformationen unter *https://www.anaconda.com/pricing*

2 Vor einer Nutzung im beruflichen oder institutionellen Kontext sollten die aktuellen Lizenzbedingungen unbedingt geprüft werden.

3 Bitte beachten Sie, dass wir für die rechtliche Korrektheit keine Gewähr übernehmen können – insbesondere, da sich die Lizenzbedingungen jederzeit ändern können.

3.3 Installation mit pip

Alternativ zur Nutzung von conda ist pip das offizielle, standardisierte Werkzeug zur Paketinstallation und ist bei aktuellen Python-Versionen üblicherweise bereits vorinstalliert. Die Installation von Python-Paketen über pip ist aus lizenzrechtlicher Sicht die sicherste und unkomplizierteste Methode. pip greift auf das zentrale Python-Paketverzeichnis PyPI (Python Package Index) zu, in dem nahezu alle verbreiteten Pakete unter freien Open-Source-Lizenzen (z. B. MIT, BSD oder Apache) veröffentlicht sind. Diese Lizenzen erlauben sowohl private als auch kommerzielle Nutzung ohne gesonderte Genehmigung oder Lizenzgebühren.

Für die im Buch behandelten Beispiele aus Datenanalyse und Visualisierung können die benötigten Pakete wie folgt installiert werden:

```
pip install numpy matplotlib pandas jupyterlab
pip install xlrd dataframe-image openpyxl
```

Falls pip nicht auf dem aktuellen Stand ist, kann es mit folgendem Befehl aktualisiert werden:

```
python -m pip install --upgrade pip
```

Weitere Pakete werden im Laufe des Buches benötigt. Wir weisen an den entsprechenden Stellen darauf hin. Sie können diese dann bei Bedarf problemlos mit pip nachinstallieren.

3.4 Starten von JupyterLab

Nach der Installation kann JupyterLab gestartet werden. Der genaue Befehl hängt vom Betriebssystem ab:

- *Linux/macOS*: Öffnen Sie ein Terminal und geben Sie folgenden Befehl ein:

  ```
  jupyter lab
  ```

- *Windows (Eingabeaufforderung oder PowerShell)*: Öffnen Sie die Eingabeaufforderung (cmd.exe) oder PowerShell und geben Sie ein:

  ```
  jupyter lab
  ```

- *Windows (Anaconda-Nutzer)*: Falls JupyterLab über Anaconda installiert wurde, kann es auch über die grafische Oberfläche gestartet werden:

  1. Öffnen Sie die Anaconda Navigator-App.
  2. Wählen Sie im Menü „JupyterLab“ und klicken Sie auf „Start“.

Nach dem Start öffnet sich eine Browser-Oberfläche, über die neue Notebooks erstellt und bearbeitet werden können. Falls sich der Browser nicht automatisch öffnet, kann folgende Adresse manuell eingegeben werden:

```
http://localhost:8888/lab
```

Falls JupyterLab nach der Installation nicht gefunden wird, kann es notwendig sein, die entsprechende Umgebung zu aktivieren (bei Nutzung von `conda`):

```
conda activate my_env
jupyter lab
```

3.5 Warum JupyterLab?

Im weiteren Verlauf dieses Buches wird JupyterLab häufig verwendet, da es sich hervorragend für interaktive Programmierung eignet. Es bietet:

- die schrittweise Ausführung von Code,
- die direkte Visualisierung von Diagrammen,
- die Kombination von Code, Formeln und Erklärungen in einem Dokument,
- eine moderne Oberfläche mit Datei-Explorer, Tabs und integrierten Terminals.

Dies erleichtert das Testen von Code und die Analyse von Daten erheblich. Falls JupyterLab nicht genutzt werden soll, können alle Codebeispiele auch in einer regulären Python-Umgebung oder einer IDE wie VS Code oder PyCharm ausgeführt werden.

Teil I

NumPy

4
NumPy Einführung

4.1 Überblick

4.1.1 Was ist NumPy?

Bei NumPy handelt es sich um ein Modul, das grundlegende Datenstrukturen – mehrdimensionale Arrays und Matrizen – sowie wichtige Funktionalitäten bereitstellt. Diese werden von anderen Modulen wie beispielsweise Matplotlib, SciPy und Pandas genutzt.

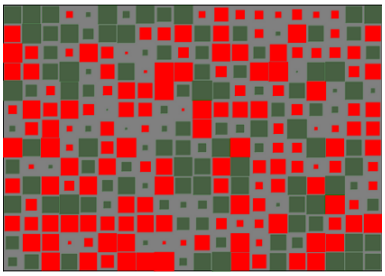


Bild 4.1 Visualisierung einer Matrix als Hinton-Diagram

Der Name NumPy ist ein Akronym für „Numerical Python“, also „numerisches Python“. Von Anfang an wurde bei der Entwicklung von NumPy besonderer Wert auf eine speicherschonende und effiziente Implementierung gelegt. Daher ist der Großteil des Moduls in C geschrieben, was eine hohe Ausführungsgeschwindigkeit der numerischen und mathematischen Funktionen sicherstellt.[1)]

Durch NumPy wird Python um leistungsfähige Datenstrukturen erweitert, die effiziente Berechnungen mit großen Arrays und Matrizen ermöglichen – sogar für extrem große Datenmengen („Big Data“). Zudem stellt das Modul eine Vielzahl hochwertiger mathematischer Funktionen zur Verfügung, die speziell für die Arbeit mit diesen Strukturen optimiert sind. SciPy („Scientific Python“, also „wissenschaftliches Python“) wird oft in einem Atemzug mit NumPy genannt. Es erweitert NumPy um zusätzliche Funktionen wie Minimierung, Regression, Fouriertransformation und viele weitere Werkzeuge.

1 Siehe *https://stackoverflow.com/questions/1825857/how-much-of-numpy-and-scipy-is-in-c*

Das Diagramm in Bild 4.1 wurde mit Python unter Verwendung von NumPy und Matplotlib erzeugt. Es zeigt ein sogenanntes Hinton-Diagramm zur Visualisierung einer 14×20-Matrix: Die Größe der Quadrate repräsentiert den Betrag der Matrixwerte, die Farbe deren Vorzeichen – rot für negative und grün für positive Werte.

NumPy basiert auf zwei früheren Python-Modulen, die mit Arrays zu tun hatten. Eines von diesen ist Numeric. Numeric ist wie NumPy ein Python-Modul für leistungsstarke numerische Berechnungen, aber es ist heute überholt. Ein anderer Vorgänger von NumPy ist Numarray, bei dem es sich um eine vollständige Überarbeitung von Numeric handelt, aber auch dieses Modul ist heute veraltet. NumPy ist die Verschmelzung dieser beiden, d. h. es ist auf dem Code von Numeric und den Funktionalitäten von Numarray aufgebaut.

4.1.2 Ein einfaches Beispiel

Um mit NumPy arbeiten zu können, müssen wir es zuerst importieren:

```
import numpy
# oder deutlich gebräuchlicher, um Schreibarbeit zu sparen:
import numpy as np
```

In unserem ersten einfachen NumPy-Beispiel definieren wir ein eindimensionales NumPy-Array:

```
C = np.array([20.8, 21.9, 22.5, 22.7, 22.3, 21.0, 21.2, 20.9])
print(C)
```

Dies ist das Ergebnis des Codes:

```
[20.8 21.9 22.5 22.7 22.3 21.  21.2 20.9]
```

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)
```

Der Code ergibt folgendes Resultat:

```
[69.44 71.42 72.5  72.86 72.14 69.8  70.16 69.62]
```

Verglichen zu diesem Vorgehen stellt sich die bestmöglich reine Python-Lösung[2], die eine Liste mithilfe einer Listen-Abstraktion in eine Liste mit Fahrenheit-Temperaturen wandelt, als umständlich dar:

```
cvalues = [20.8, 21.9, 22.5, 22.7, 22.3, 21.0, 21.2, 20.9]
fvalues = [x * 9 / 5 + 32 for x in cvalues]
print(fvalues)
```

Das Ergebnis erscheint wie folgt:

```
[69.44, 71.42, 72.5, 72.86, 72.14, 69.8, 70.16, 69.62]
```

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`“:

```
print(type(C))
```

Das Resultat ist:

```
<class 'numpy.ndarray'>
```

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

4.2 Vergleich NumPy-Datenstrukturen und Listen

4.2.1 Zentrale Unterschiede

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.[3]
- Listen bieten effiziente Methoden zum Einfügen, Anhängen und Löschen von Elementen.
- Dictionaries bieten einen schnellen Lookup.

Vorteile von NumPy-Datenstrukturen gegenüber Python:

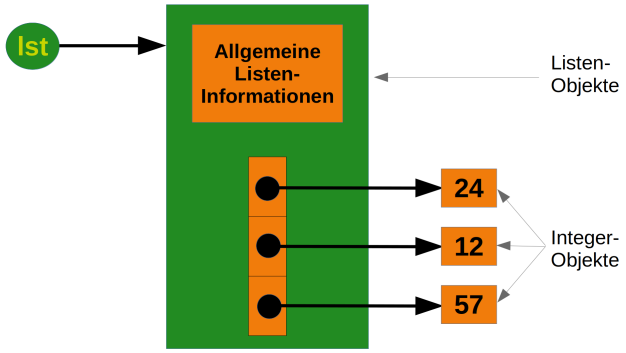
- Array-basierte Berechnungen
- Effizient implementierte mehrdimensionale Arrays
- Entworfen für wissenschaftliche Berechnungen

2 Also Python ohne Benutzung des NumPy-Moduls

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

4.2.2 Speicherbedarf

Die wesentlichen Vorteile von NumPy-Arrays sind ein geringer Speicherverbrauch und ein optimales Laufzeitverhalten. Wir wollen uns den Speicherverbrauch von NumPy-Arrays in diesem Kapitel unseres Tutorials näher 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)   # nur die grüne 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)
```

Das ergibt sich aus dem Code:

```
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)
empty_lst = []
print("Speicherbedarf einer leeren Liste: ", size(empty_lst))
```

Das ergibt sich aus dem Code:

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

Aus den Ausgaben des vorigen Codes[4)] 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:

```
56 + 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:

4 Je nach Python Version können die Werte auch unterschiedlich sein, was aber am Prinzip nichts ändert!

Wir bestimmen nun den Speicherbedarf des Arrays aus dem vorherigen Bild:

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

Nach der Ausführung erhalten wir:

```
136
```

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

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

Das Skript liefert:

```
112
```

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

`112 + n * 8 Bytes`

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

`56 + 8 * len(lst) + len(lst) * 28`

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

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

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

```
a8 = np.array([24, 12, 57], np.int8)
a16 = np.array([24, 12, 57], np.int16)
a32 = np.array([24, 12, 57], np.int32)
a64 = np.array([24, 12, 57], np.int64)
print(size(a8), size(a16), size(a32), size(a64))
```

Die Ausführung des Codes ergibt:

```
115 118 124 136
```

4.2.3 Zeitvergleich zwischen 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 = 1000000
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(f'NumPy ist in diesem Fall {t1 / t2:5.2f}-mal schneller!')
```

Hier ist die Ausgabe:

```
0.2208847999572754 0.011658430099487305
NumPy ist in diesem Fall 18.95-mal schneller!
```

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

Dem Konstruktor eines Timer-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:

```
0.0017026760033331811
5.411202437244356e-05
```

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

```
print(timer_obj1.repeat(repeat=2, number=10))
print(timer_obj2.repeat(repeat=2, number=10))
```

Was wir erhalten, ist:

```
[0.001727322000078857, 0.001787365006748587]
[3.801099956035614e-05, 3.414400271140039e-05]
```

5
Erzeugung und Struktur von Arrays

Nachdem wir im vorigen Kapitel gelernt haben, wie man NumPy-Arrays aus Listen und Tupeln erstellt, werden wir uns nun systematisch mit der internen Struktur von Arrays befassen. Darüber hinaus werden Sie weitere zentrale Funktionen zur Erzeugung und Initialisierung von Arrays kennenlernen – darunter Arrays mit vorgegebenem Inhalt wie Nullen oder Einsen sowie solche mit automatisch erzeugten Zahlenfolgen oder Zufallswerten.

Bild 5.1 Symbolisches Array

5.1 Dimensionen

5.1.1 Nulldimensionale Arrays in NumPy

In NumPy können Arrays beliebiger Dimension erstellt werden – einschließlich nulldimensionaler Arrays. Ein Skalar, also ein einzelner Zahlenwert ohne Achse oder Richtung, wird in NumPy als nulldimensionales Array dargestellt.

Im folgenden Beispiel erzeugen wir ein solches nulldimensionales Array mit dem Wert 42. Wenn wir die Methode `ndim` auf das Array anwenden, erhalten wir dessen Dimension. Darüber hinaus lässt sich der Typ des Objekts als `numpy.ndarray` identifizieren.

```
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))
```

Nach der Ausführung erhalten wir:

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

5.1.2 Eindimensionales Array

Wir haben bereits in unserem anfänglichen Beispiel ein eindimensionales Array – besser als Vektor bekannt – gesehen. Was wir bis jetzt noch nicht erwähnt haben, aber naheliegt, 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=}\n{V=}")
print(f"Typ von F: {F.dtype}, Typ von V: {V.dtype}")
```

Wir erhalten diese Ausgabe:

```
F=array([ 1,  1,  2,  3,  5,  8, 13, 21])
V=array([ 3.4,  6.9, 99.8, 12.8])
Typ von F: int64, Typ von V: float64
```

5.1.3 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(f"{A}\n{A.ndim=}")
```

Das Ergebnis des Codes ist:

```
[[ 3.4  8.7  9.9]
 [ 1.1 -7.8 -0.7]
 [ 4.1 12.3  4.8]]
A.ndim=2
```

5.2 Gestalt eines Arrays

Die Funktion `shape` liefert die Größe bzw. die Gestalt eines Arrays in Form eines Integer-Tupels zurück. Diese Zahlen bezeichnen die Längen der entsprechenden Array-Dimensionen, d. h. im zweidimensionalen Fall den Zeilen und Spalten. In anderen Worten: Die Gestalt oder Shape eines Arrays ist ein Tupel mit der Anzahl der Elemente pro Achse (Dimension). In unserem Beispiel ist die Shape gleich (6, 3). Das bedeutet, das wir sechs Zeilen und drei Spalten haben.[1)]

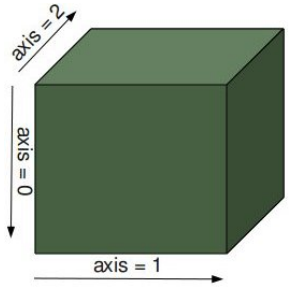
```
x = np.array([[67, 63, 87],
              [77, 69, 59],
              [85, 87, 99],
              [79, 72, 71],
              [63, 89, 93],
              [68, 92, 78]])
print(np.shape(x))
```

Dies ist das Ergebnis des Codes:

```
(6, 3)
```

Es gibt auch eine äquivalente Array-Property `print(x.shape)`, die das gleiche Ergebnis liefert.

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)
```

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

Das Ergebnis erscheint wie folgt:

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

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

Dies ist das Ergebnis des Codes:

```
[[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)` eingeben würde.

Die Shape eines Skalars ist ein leeres Tupel:

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

Ergebnis:

```
()
```

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

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

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])
```

Das Ergebnis erscheint wie folgt:

```
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. Alternativ können wir auch nur ein Klammernpaar benutzen, und alle Indizes werden mit Kommas separiert:

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

Hier ist die 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])
```

Nach der Ausführung erhalten wir:

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

Das englische Verb „to slice“ bedeutet in Deutsch „(in Scheiben) schneiden“. Dies entspricht der Bedeutung 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 den Listen im Falle von eindimensionalen Arrays. Allerdings können wir „Slicing“ auch auf mehrdimensionale Arrays anwenden. Slicing unterstützt ein bis drei Parameter in der Form `[start:stop:step]`.

Wir demonstrieren die Arbeitsweise des Teilbereichsoperators an einigen Beispielen mit einem eindimensionalen Array:

```
S = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(S[2:5]) # Die Elemente von Position 2 (inkl.) bis 5 (exkl.)
print(S[:4])  # Die Elemente von Anfang bis zur Pos. 4 (exklusive)
print(S[6:])  # von Pos 6 (inkl.) bis zum Ende
print(S[:])   # von Anfang bis Ende
```

Ausgabe:

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

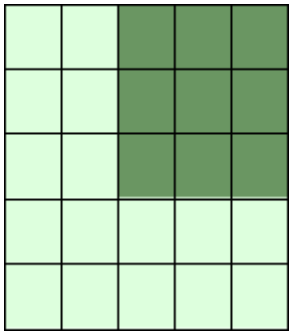
Bei mehrdimensionalen Arrays werden die Slicing-Bereiche für jede Dimension durch Kommata getrennt angegeben:

```
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:])
```

Der Code ergibt folgendes Resultat:

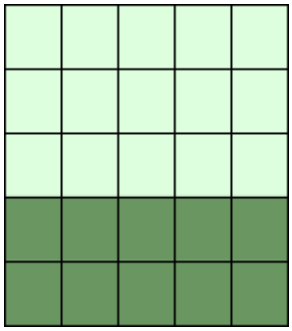
```
[[13 14 15]
 [23 24 25]
 [33 34 35]]
```



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

Wir erhalten diese Ausgabe:

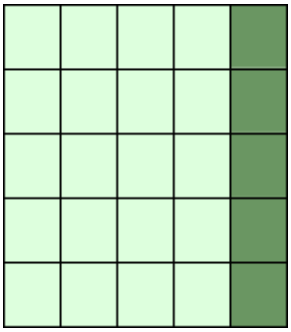
```
[[41 42 43 44 45]
 [51 52 53 54 55]]
```



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

Es folgt das Ergebnis:

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



Die folgenden beiden Beispiele benutzten auch noch den dritten Parameter `step`.

```
X = 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, 24, 25, 26, 27]])
print(X)
```

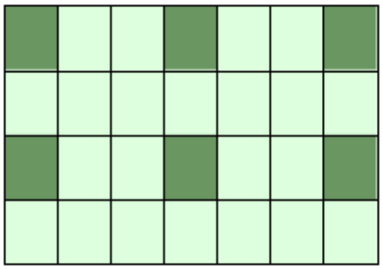
Die Auswertung ergibt:

```
[[ 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:

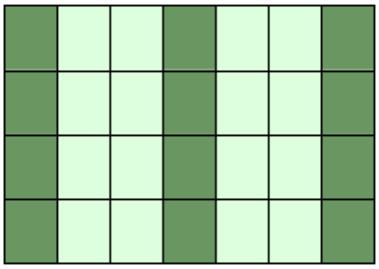
```
[[ 0  3  6]
 [14 17 20]]
```



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

Die Ausführung führt zu dieser Ausgabe:

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



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

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

print(A[1:3, 0:2])  # equivalent zu print(A[1:3, 0:2, :])
```

Ergebnis:

```
[[[46 14  4]
  [45 14  5]]

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

Achtung: Der Teilbereichsoperator erzeugt bei Listen und Tupeln neue Objekte – bei NumPy hingegen nur eine Sicht (englisch: „view“) auf das Originalarray. Änderungen an dieser Sicht wirken sich daher direkt auf das Originalarray aus.

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

Das Ergebnis des Codes ist:

```
[ 0  1  2 23  4  5  6  7  8  9]
```

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

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

Die Verarbeitung ergibt:

```
[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]

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

Das ergibt sich aus dem Code:

```
True
```

Auch wenn es in den meisten Beispielen so aussieht, als würden sich die Arrays immer Elemente teilen, wenn die Funktion `True` zurückliefert, ist dies nicht immer so. Wir zeigen dies im folgenden Beispiel. `B1` und `B2` haben 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))
```

Das Resultat ist:

```
True
```

`may_share_memory` liefert im vorigen Beispiel `True` zurück, obwohl die Arrays keinen Speicher teilen.

5.4 Dreidimensionale Arrays

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

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

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

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

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

print(X.shape)
```

Der Code ergibt folgendes Resultat:

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

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

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

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

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

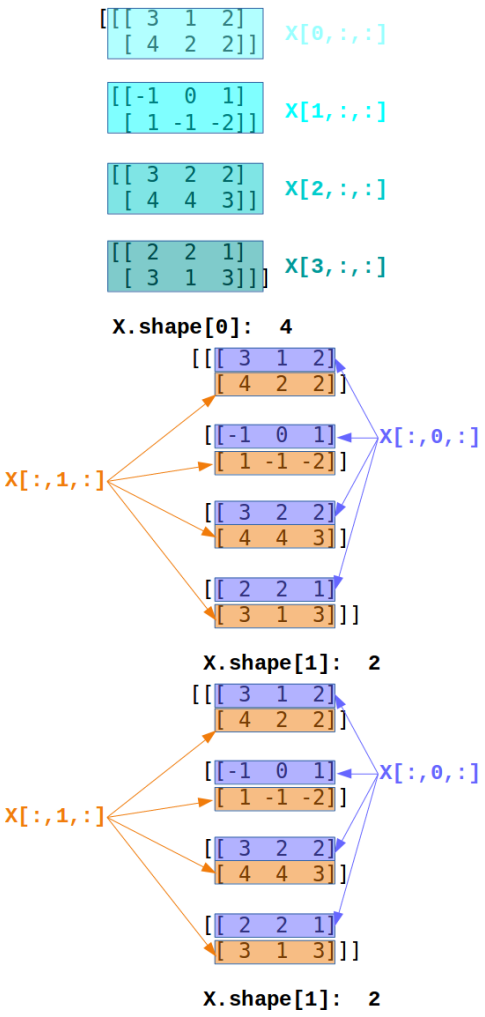
Hier ist die Ausgabe:

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

Dimension 1 with size  2
Ausgabe von X[:,0,:]:
[[ 3  1  2]
 [-1  0  1]
 [ 3  2  2]
 [ 2  2  1]]
Ausgabe von X[:,1,:]:
[[ 4  2  2]
 [ 1 -1 -2]
 [ 4  4  3]
 [ 3  1  3]]

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

Die folgenden Bilder erläutern dies noch weiter:



5.5 Array-Erzeugungsfunktionen

Viele numerische Algorithmen erfordern zu Beginn Arrays mit festen Werten oder vordefinierter Struktur. NumPy bietet hierfür Funktionen, mit denen sich solche Arrays einfach und effizient erzeugen lassen.

So bietet NumPy zwei 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.

Ein häufiges Szenario ist die Erzeugung von Arrays mit gleichmäßig verteilten Werten innerhalb eines bestimmten Intervalls. Dafür bietet NumPy zwei zentrale Funktionen:

- `arange` verwendet einen festen Schrittwert, um Werte innerhalb eines offenen Intervalls zu generieren.
- `linspace` erzeugt eine definierte Anzahl an Werten über ein Intervall mit automatisch berechnetem Abstand.

5.5.1 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. Als Argumente können sowohl Integer als auch Float-Werte übergeben werden. 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` ein range-Objekt zurückliefert. Ein range-Objekt erlaubt uns, über einen großen Zahlenbereich zu iterieren, wobei die Zahlen nur bei Bedarf erzeugt werden. Falls der `start`-Parameter bei `arange` nicht übergeben wird, wird `start` auf 0 gesetzt. Das Ende des Intervalls wird durch den Parameter `stop` bestimmt. Üblicherweise wird das Intervall diesen Wert nicht beinhalten, außer in den Fällen, in denen `step` keine Ganzzahl ist und floating-point-Effekte die Länge des Ausgabearrays beeinflussen. Der Abstand zwischen zwei benachbarten Werten des Ausgabearrays kann mittels des optionalen Parameters `step` gesetzt werden. Der Default-Wert für `step` ist 1.

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

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

```
import numpy as np

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

x = range(1, 7)
print(x)    # x ist ein Iterator
print(list(x))

x = np.arange(7.3)
print(x)
```

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

Der Code ergibt folgendes Resultat:

```
[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]
```

Man muss vorsichtig sein, wenn man einen Float-Wert für den Step-Parameter verwendet, wie wir im folgenden Beispiel sehen können:

```
x = np.arange(12.04, 12.84, 0.08)
print(x)
```

Das ergibt sich aus dem Code:

```
[12.04 12.12 12.2  12.28 12.36 12.44 12.52 12.6  12.68 12.76 12.84]
```

Die Hilfe von `arange` sagt für den Stop-Parameter Folgendes aus: „Ende des Intervalls“. Das Intervall schließt diesen Wert nicht ein, außer in einigen Fällen, in denen step keine ganze Zahl ist und die Abrundung der Fließkommazahl die Länge von out beeinflusst. Dies ist in unserem Beispiel der Fall.

Die folgende Verwendung von `arange` ist ein wenig abwegig. Warum sollten wir Fließkommazahlen verwenden, wenn wir Ganzzahlen als Ergebnis haben wollen? Trotzdem könnte das Ergebnis verwirrend sein.

```
x = np.arange(0.6, 10.4, 0.71, int)
print(x)
```

Nach der Ausführung erhalten wir:

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13]
```

Dieses Ergebnis scheint sich allen logischen Erklärungen zu entziehen. Es lässt sich aber dadurch erklären, dass `arange`, bevor es startet, den Wert für den Startwert auf die nächst-kleinere Integerzahl abschneidet, dann berechnet es die Anzahl der Schritte. In unserem Fall `(10.4 - 0.6) / 0.71`, was 13.802816901408452 ergibt. Auch dieser Wert wird abgeschnitten, also auf 13. Lediglich bei der Schrittweite wird gerundet, also auf 1 aufgerundet. Dann werden dreizehn „Schritte“ ab 0 durchgeführt. Moral von der Geschicht’: Wenn man als Ergebnis ganze Zahlen haben will, sollte man auch bei den Parametern ganze Zahlen verwenden.