2. Auflage

Bernd **KLEIN**

Numerisches PYTHON

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

HANSER

Klein

Numerisches Python

Bernd Klein

Numerisches Python

Arbeiten mit NumPy, Matplotlib und Pandas

2., aktualisierte und erweiterte Auflage

HANSER

Der Autor:
Bernd Klein, bernd@python-kurs.eu

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.

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
Titelmotiv: © stock.adobe.com/ARTvektor
Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Göttingen
Printed in Germany

Print-ISBN: 978-3-446-47170-2
E-Book-ISBN: 978-3-446-47366-9
E-Pub-ISBN: 978-3-446-47957-9

Inhalt

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. Mein besonderer Dank für die zweite Auflage gilt Herrn Tobias Habermann, der dafür gesorgt hat, dass alle Python-Beispiele dieser Auflage automatisch mittels „Pythontex“ getestet und ausgeführt werden, und der auch ansonsten bei der Fehlerkorrektur sehr hilfreich war.

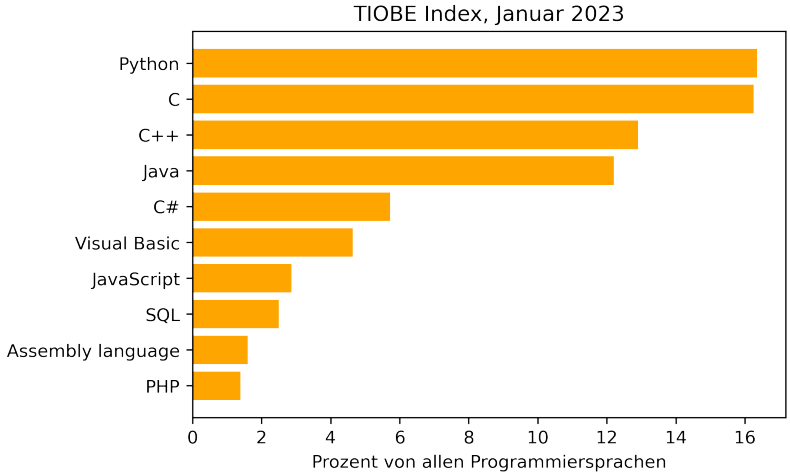
Zuletzt danke ich auch ganz herzlich dem Hanser Verlag, der dieses Buch nun auch in der zweiten, deutlich erweiterten und farbigen Auflage ermöglicht hat. Vor allem danke ich Frau Brigitte Bauer-Schiewek, Programmplanung Computerbuch, und Frau Kristin Rothe, Lektoratsassistenz Computerbuch, für die kontinuierliche 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 Korrektorat.

Bernd Klein, Singen

1 Einleitung

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. Vielleicht hat man auch das Glück – wie wir es sehen – und darf mit Python arbeiten. Schaut man sich die Umfrageergebnisse zu den beliebtesten Programmiersprachen an, so findet man Python immer an erster oder einer der ersten Stellen. Nach dem von TIOBE[1] berechneten Ranking führt Python deutlich vor allen anderen Programmiersprachen. In der ersten Auflage dieses Buches (also 2018) stand Python noch auf Platz 3.



[1] 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!

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 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, im Büro, 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 „Maschinellem Lernen“. Zwei Gebiete, in denen es um die Lösungen von numerischen Problemen geht. Der überwältigende Erfolg von Python beruht unter anderem auf den Modulen 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. Da die Module weitestgehend in C geschrieben sind, kann man sagen, dass sie so schnell wie C-Programme laufen.

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 „Maschinelles 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“[2] 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.

1.3 Python-Installation

Wir gehen davon aus, dass bei den Leserinnen und Lesern des Buches Python bereits installiert ist, insbesondere mit den genannten Modulen NumPy, Matplotlib und Pandas. Sollte dies nicht der Fall sein, empfehlen wir die Installation von Anaconda[3], die sich äußerst einfach auf allen Betriebssystemen gestaltet. Damit ist alles installiert, was im Laufe des Buches benötigt wird.

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

[3] *https://www.anaconda.com/distribution/*

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: bernd.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, April 2023

2 Numerisches Programmieren mit Python

2.1 Begriffsbestimmung

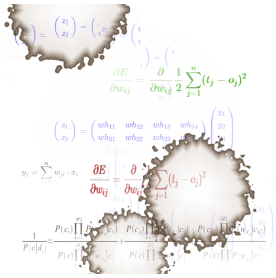


Bild 2.1 Mathematische Formeln und Kaffeeklecke

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 würde dies dann für fast alle Programme gelten. Denn auch wenn ein Programm scheinbar nichts mit Zahlen zu tun hat, birgt es einen numerischen Kern. Denkt man beispielsweise an den Google-Algorithmus und an die Art, wie er einem Vorschläge zu Webseiten auf eine Suchanfrage 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.

Allgemein versteht man unter numerischer Programmierung Software zur Lösung von Problemen, die oft große Datenmengen und/oder komplexe Gleichungen enthalten. Es geht also darum, mathematische Algorithmen zu implementieren, die sich auf großen Datensätzen anwenden lassen. Vor allen Dingen **effizient** anwenden lassen.

In unserem Buch haben wir insbesondere 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 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

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 eine 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 wie MATLAB und R, wenn nicht gar effizienter.

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. R und MATLAB, die Hauptkonkurrenten von Python, verlieren aus verschiedenen Gründen mehr und mehr an Bedeutung.

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. Zusätzliche eignet sich Python in Kombination mit den Modulen NumPy, SciPy, Matplotlib und Pandas bestens, um R oder MATLAB zu ersetzen.

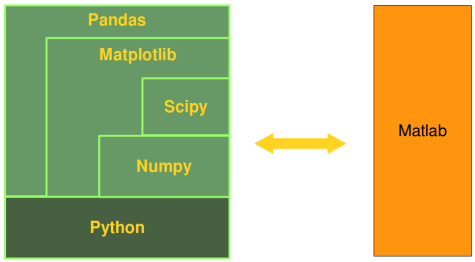


Bild 2.2 Zusammenhang zu MATLAB

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

NumPy

3 NumPy – Einführung

3.1 Überblick

Bei NumPy handelt es sich um ein Modul, welches grundlegende Datenstrukturen, – d. h. mehrdimensionale Arrays und Matrizen, – und Funktionalitäten zur Verfügung stellt, die von zahlreichen anderen Modulen, unter anderem von Matplotlib, SciPy und Pandas, benutzt werden. Der Name NumPy stellt ein Akronym für den englischen Begriff „Numerical Python“ dar, also „numerisches Python“. 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.

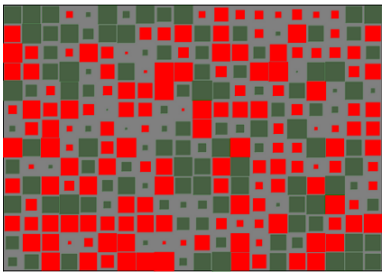


Bild 3.1 Visualisierung einer Matrix als Hinton-Diagramm

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, also wissenschaftliches Python) wird oft im gleichen Atemzug wie NumPy genannt. SciPy erweitert die Leistungsfähigkeit von NumPy um weitere nützliche Funktionen wie zum Beispiel Minimierung, Regression, Fourier-Transformation 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. Will man NumPy manuell installieren, sollte man beachten, dass es als Erstes installiert wird, also vor SciPy.

Das Diagramm in Bild 3.1 wurde übrigens auch mit Python unter Benutzung von NumPy und Matplotlib erzeugt. Das Bild stellt eine grafische Visualisierung einer Matrix mit 14 Reihen und 20 Spalten dar. Es handelt sich um ein sogenanntes Hinton-Diagramm. Die Größe eines Quadrates innerhalb dieses Diagramms korrespondiert zu der Größe des ent-

sprechenden Wertes in der darzustellenden Matrix. Die Farbe bestimmt dabei, ob es sich um einen positiven oder negativen Wert handelt. In unserem Beispiel: Die Farbe Rot bezeichnet die negativen Werte, und die Farbe Grün bezeichnet die positiven 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.

3.2 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

3.3 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, um sich bei der Benutzung ein wenig Schreibarbeit zu sparen:

```
import numpy as np
```

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

```
cvalues = [20.8, 21.9, 22.5, 22.7, 22.3, 21.0, 21.2, 20.9, 20.1]
```

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

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

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

Ausgabe:

```
[20.8 21.9 22.5 22.7 22.3 21.  21.2 20.9 20.1]
```

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)
```

Ausgabe:

```
[69.44 71.42 72.5  72.86 72.14 69.8  70.16 69.62 68.18]
```

Das Array C selbst wurde dabei jedoch nicht verändert:

```
print(C)
```

Ausgabe:

```
[20.8 21.9 22.5 22.7 22.3 21.  21.2 20.9 20.1]
```

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

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

Ausgabe:

```
[69.44, 71.42, 72.5, 72.86, 72.14, 69.8, 70.16, 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`“:

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

Ausgabe:

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

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

3.4 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 ar-

[2] Also Python ohne Benutzung des NumPy-Moduls

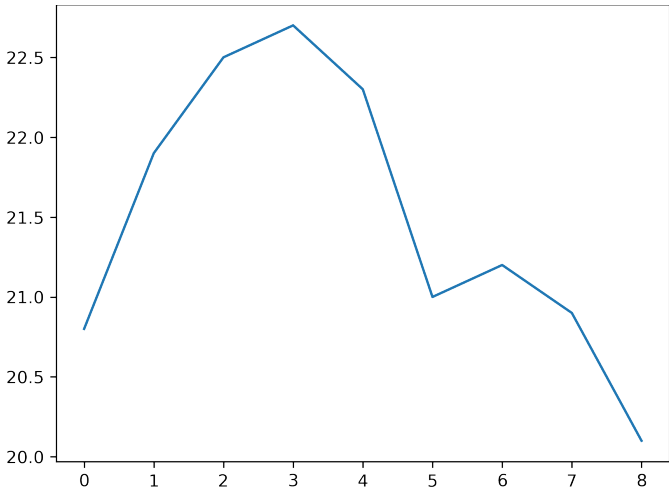
beitet[3], empfiehlt es sich, die folgende Codezeile zu verwenden, falls der Plot nicht so wie im Buch erscheint:

```
%matplotlib inline
```

Dies ist meistens die Default-Einstellung. Sollen die Plots in einem Notebook 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:

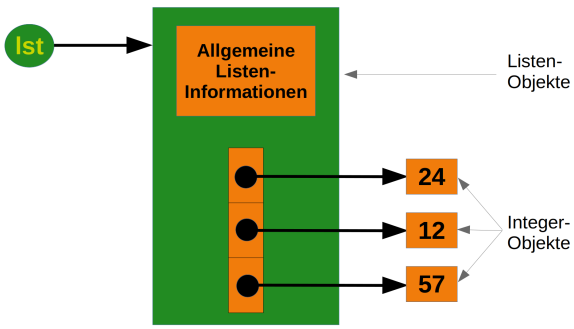
```
import matplotlib.pyplot as plt
plt.plot(C)
```



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.

3.5 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.



[3] Nur dort oder in der ipython-Shell, denn es handelt sich nicht um Python-Befehle!

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)
```

Ausgabe:

```
Größe ohne Größe der Elemente:  120
Größe aller Elemente:  84
Gesamtgröße der Liste:  204
```

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))
```

Ausgabe:

```
Größe ohne Größe der Elemente:  120
Größe aller Elemente:  112
Gesamtgröße der Liste:  232
Speicherbedarf einer leeren Liste:  56
```

Aus den Ausgaben des vorigen Codes[4] können wir folgern, dass wir für jedes Integer-Element 16 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 + 16 * len(lst)
```

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

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:

```
128
```

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

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

Ausgabe:

```
104
```

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:

```
104 + n * 8 Bytes
```

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

```
56 + 16 * len(lst) + len(lst) * 28
```

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

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

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

```
a = np.array([24, 12, 57], np.int8)
print(size(a) - size(e))
a = np.array([24, 12, 57], np.int16)
print(size(a) - size(e))
a = np.array([24, 12, 57], np.int32)
print(size(a) - size(e))
a = np.array([24, 12, 57], np.int64)
print(size(a) - size(e))
```

Ausgabe:

```
3
6
12
24
```

3.6 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 = 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!')
```

Ausgabe:

```
0.6928548812866211 0.017958402633666992
NumPy ist in diesem Fall 38.58-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 per Default auf `pass` 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 wiederholen 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:

```
0.0043389090005803155
0.00010923299851128832
```

Die `repeat`-Methode 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:

```
[0.005344285998944542, 0.007209806999526336, 0.005345828998542856]
[0.0001395979998051189, 0.0001086129996110685 2, 0.00010834400018211454]
```

4 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 Abstandwert, um innerhalb von gegebenen Intervallgrenzen entsprechende Werte zu generieren, während `linspace` eine bestimmte Anzahl von Werten innerhalb gegebener Intervallgrenzen berechnet. Den Abstand berechnet `linspace` automatisch.

Bild 4.1 Symbolisches Array

4.1 Erzeugung äquidistanter Intervalle

4.1.1 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. 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)

# 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)
```

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]
```

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)
```

Ausgabe:

```
[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)
```

Ausgabe:

```
[ 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 nächstkleinere Integer-Zahl 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. Die Moral von der Geschicht': Wenn man als Ergebnis ganze Zahlen haben will, sollte man auch bei den Parametern ganze Zahlen verwenden.

4.1.2 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
# Die nächste Zeile dient nur dazu, die Buchausgabe zu verschönern:
np.set_printoptions(linewidth=65, precision=3)

# 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.     1.184  1.367  1.551  1.735  1.918  2.102  2.286  2.469
  2.653  2.837  3.02   3.204  3.388  3.571  3.755  3.939  4.122
  4.306  4.49   4.673  4.857  5.041  5.224  5.408  5.592  5.776
  5.959  6.143  6.327  6.51   6.694  6.878  7.061  7.245  7.429
  7.612  7.796  7.98   8.163  8.347  8.531  8.714  8.898  9.082
  9.265  9.449  9.633  9.816 10.   ]
[ 1.   2.5  4.   5.5  7.   8.5 10. ]
[1.    2.286 3.571 4.857 6.143 7.429 8.714]
```

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

```
import numpy as np

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

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

Ausgabe:

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

Vielleicht fragen sich hier einige, wozu dies notwendig sein sollte. Kann man den Abstand nicht einfach als Differenz zweier benachbarter Array-Elemente berechnen? Dazu schauen wir uns folgendes Beispiel an:

```
import numpy as np
a, spacing = np.linspace(4, 23, endpoint=False, retstep=True)
print(a[:6])

# Abstände zwischen den ersten 6 Array-Elementen:
for i in range(6):
    print(a[i + 1] - a[i])

print(f"{spacing=}")
```

Ausgabe:

```
[4.   4.38 4.76 5.14 5.52 5.9 ]
0.3799999999999999
0.3799999999999999
0.3800000000000008
0.379999999999999
0.3800000000000008
0.3799999999999999
spacing=0.38
```

Aus der Ausgabe können wir ersehen, dass sich die berechneten Werte unterscheiden und dass keiner dieser Werte dem „wahren“ Wert, also dem von `linspace` zurückgelieferten Wert `spacing`, entspricht. Es handelt sich um Rundungsprobleme bei den Float-Operationen.

4.1.3 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
```

4.1.4 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 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
```

4.1.5 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.

4.2 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 die 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, 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:

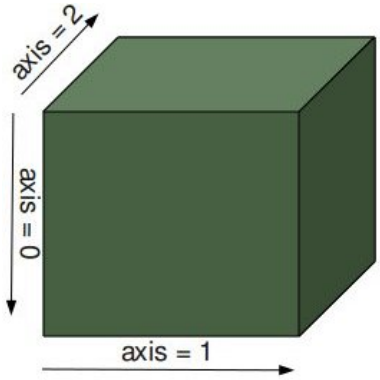
```
(6, 3)
```

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)
```

[1] In der Mathematik benutzt man neben „Gestalt“ auch den Begriff „Typ“ einer Matrix. Man spricht von einer m × 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]]
```

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

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)
```

4.3 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])
# 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.

Alternativ können wir auch nur ein Klammernpaar benutzen, 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 bereits vertraut sind mit den Teilbereichsoperatoren (slicing) von den Listen und Tupeln. Das englische Verb „to slice“ bedeutet in Deutsch „schneiden“ oder auch „in Scheiben schneiden“. Letztere Bedeutung entspricht auch der Arbeitsweise des Teilbereichsoperators in Python und NumPy. Man schneidet sich gewissermaßen eine „Scheibe“ aus einem sequentiellen Datentyp oder einem Array heraus. Die Syntax in NumPy ist analog zu den Listen 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]) # 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]
```

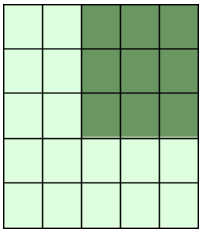
Die Anwendung des Teilbereichsoperators auf mehrdimensionale Arrays illustrieren wir in den folgenden Beispielen. Die Bereiche für jede Dimension werden durch Kommas getrennt:

```
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:

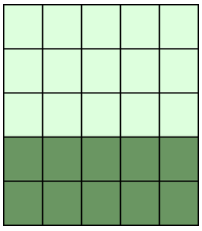
```
[[13 14 15]
 [23 24 25]
 [33 34 35]]
```



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

Ausgabe:

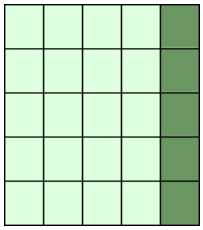
```
[[41 42 43 44 45]
 [51 52 53 54 55]]
```



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

Ausgabe:

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



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

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

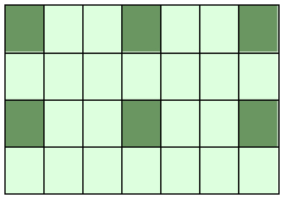
Ausgabe:

```
[[ 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:

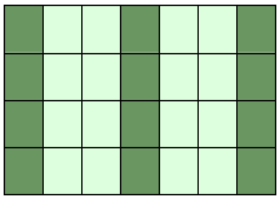
```
[[ 0  3  6]
 [14 17 20]]
```



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

Ausgabe:

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



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

Ausgabe:

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

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

Achtung: Während der Teilbereichsoperator bei Listen und Tupel neue Objekte erzeugt, generiert er bei NumPy nur eine Sicht (engl. „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. Genaugenommen 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]

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

Ausgabe:

```
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))
```

Ausgabe:

```
True
```

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

4.4 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)
```

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(f"\nAusgabe von X[{i:1},:,:]:")
    print(X[i, :, :])

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

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

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: