Datenanalyse mit Python

Auswertung von Daten mit pandas, NumPy und Jupyter

KAPITEL 4

Grundlagen von NumPy: Arrays und vektorisierte Berechnung

NumPy, kurz für *Numerical Python*, ist eines der wichtigsten Pakete für numerische Berechnungen in Python. Viele wissenschaftliche Rechenpakete nutzen die Array-Objekte von NumPy als *Lingua Franca* oder Standardschnittstelle für den Datenaustausch. Viel von dem, was ich hier über NumPy erzähle, lässt sich auch auf pandas übertragen.

Hier sind einige der Dinge, die Sie in NumPy finden werden:

- ndarray, ein effizientes mehrdimensionales Array, das schnelle Array-orientierte Arithmetikoperationen und flexible *Broadcasting*-Fähigkeiten mitbringt.
- Mathematische Funktionen für schnelle Operationen auf ganzen Daten-Arrays, ohne dass man Schleifen schreiben muss.
- Tools zum Lesen und Schreiben von Array-Daten auf die Festplatte und zum Arbeiten mit Memory-mapped Dateien.
- Lineare Algebra, Generierung von Zufallszahlen und Fourier-Transformation.
- Eine C-API zum Anbinden von NumPy an Bibliotheken, die in C, C++ oder Fortran geschrieben sind.

Wegen der umfassenden und gut dokumentierten C-API von NumPy ist es einfach, Daten an externe Bibliotheken weiterzugeben, die in einer niedrigen Programmiersprache geschrieben sind. Ebenso einfach können externe Bibliotheken Daten als NumPy-Arrays an Python zurückliefern. Diese Eigenschaft macht Python zur Sprache der Wahl, wenn man Wrapper für vorhandenen Code in C, C++ oder FORTRAN schreiben und diesem eine dynamische und leicht zu verwendende Schnittstelle geben möchte.

NumPy selbst bietet zwar keine wissenschaftliche oder Modellierungsfunktionalität, dennoch hilft Ihnen ein Grundverständnis von NumPy-Arrays und Array-orientierten Berechnungen bei der effektiveren Benutzung von Tools mit Array-orientierter Semantik wie etwa pandas. Da NumPy ein so umfassendes Thema ist, werde ich diverse komplexere NumPy-Funktionen wie etwa Broadcasting später ausführlicher behandeln (siehe Anhang A). Viele dieser fortgeschritteneren Features

sind für den Rest dieses Buchs nicht erforderlich, aber sie können hilfreich sein, sobald Sie tiefer in das wissenschaftliche Rechnen mit Python einsteigen.

Für die meisten Datenanalyseanwendungen konzentriere ich mich vor allem auf folgende Bereiche:

- Schnelle Array-basierte Operationen zum Bereinigen von Daten, Bilden und Filtern von Teilmengen sowie für Transformationen und andere Arten von Berechnungen.
- Gebräuchliche Array-Algorithmen wie Sortieren, Eindeutigkeit und Mengenoperationen.
- Effiziente beschreibende Statistik und das Aggregieren/Zusammenfassen von Daten.
- Datenausrichtung und relationale Datenmanipulationen zum Mischen und Verbinden heterogener Datenmengen.
- Ausdrücken logischer Bedingungen als Array-Ausdruck statt als Schleifen mit `if-elif-else`-Verzweigungen.
- Gruppenweise Datenmanipulation (Aggregation, Transformation, Funktionsanwendung).

Auch wenn NumPy die rechnerische Grundlage für die allgemeine numerische Datenverarbeitung bietet, werden viele Leser pandas als Basis für die meisten Arten von Statistiken oder Analysen, speziell an Tabellendaten, verwenden wollen. pandas stellt darüber hinaus eine etwas speziellere Funktionalität bereit, wie die Manipulation von Zeitreihen, die es in NumPy nicht gibt.

Die Wurzeln des Array-orientierten Rechnens in Python lassen sich bis 1995 zurückverfolgen, als Jim Hugunin die Numeric-Bibliothek geschaffen hat. Im Laufe der nächsten zehn Jahre begannen viele wissenschaftliche Gemeinschaften mit der Array-Programmierung in Python, allerdings kam es mit Beginn der 2000er-Jahre zu einer Zersplitterung des Bibliotheksökosystems. 2005 schaffte es Travis Oliphant, aus den damaligen Numeric- und Numarray-Projekten das NumPy-Projekt zusammenzuschmieden, um der Community ein einziges Array-Rechenframework zur Verfügung zu stellen.

Einer der Gründe dafür, dass NumPy für numerische Berechnungen in Python so wichtig ist, besteht in seiner Effizienz bei großen Daten-Arrays. Das hat verschiedene Ursachen:

- NumPy speichert Daten intern in einem zusammenhängenden Speicherblock unabhängig von anderen eingebauten Python-Objekten. Die NumPy-Algorithmenbibliothek, die in C geschrieben ist, kann ohne irgendwelche Typprüfungen oder anderen Aufwand auf diesem Speicher arbeiten. NumPy-Arrays verwenden außerdem viel weniger Speicher als eingebaute Python-Sequenzen.

- NumPy-Operationen führen komplexe Berechnungen auf ganzen Arrays aus, ohne dass sie dazu Python-`for`-Schleifen brauchen, was für große Sequenzen langsam sein kann. NumPy ist schneller als normaler Python-Code, weil seine auf C basierenden Algorithmen den Overhead vermeiden, den dieser Code mit sich bringt.

Damit Sie eine Vorstellung von dem Leistungsunterschied bekommen, betrachten Sie ein NumPy-Array mit einer Million Integer-Werten und die entsprechende Python-Liste:

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1_000_000)

In [9]: my_list = list(range(1_000_000))
```

Multiplizieren wir nun jede Sequenz mit 2:

```
In [10]: %timeit my_arr2 = my_arr * 2
721 us +- 7.49 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)

In [11]: %timeit my_list2 = [x * 2 for x in my_list]
49 ms +- 1.02 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

NumPy-basierte Algorithmen sind im Allgemeinen 10- bis 100-mal schneller (oder mehr) als ihre reinen Python-Gegenstücke und nutzen dabei deutlich weniger Speicher.

4.1 Das ndarray von NumPy: ein mehrdimensionales Array-Objekt

Eines der zentralen Merkmale von NumPy ist sein N-dimensionales Array-Objekt oder ndarray, ein schneller, flexibler Container für große Datenmengen in Python. Arrays erlauben Ihnen, mathematische Operationen auf ganzen Datenblöcken durchzuführen, wobei die Syntax den äquivalenten Operationen zwischen skalaren Elementen ähnlich ist.

Damit Sie einen Vorgeschmack darauf bekommen, wie NumPy Stapelberechnungen mit einer ähnlichen Syntax durchführt wie bei den Skalaren auf eingebauten Python-Objekten, importiere ich zuerst NumPy und erzeuge ein kleines Array:

```
In [12]: import numpy as np

In [13]: data = np.array([[1.5, -0.1, 3], [0, -3, 6.5]])

In [14]: data
Out[14]:
array([[ 1.5, -0.1,  3. ],
       [ 0. , -3. ,  6.5]])
```

Anschließend schreibe ich mathematische Operationen mit `data`:

```
In [15]: data * 10
Out[15]:
array([[ 15.,  -1.,  30.],
       [  0., -30.,  65.]])

In [16]: data + data
Out[16]:
array([[ 3. , -0.2,  6. ],
       [ 0. , -6. , 13. ]])
```

Im ersten Beispiel wurden alle Elemente mit 10 multipliziert, im zweiten wurden die korrespondierenden Werte in jeder »Zelle« des Arrays miteinander addiert.

In diesem Kapitel und auch im Rest des Buchs verfahre ich immer nach der Konvention `import numpy as np`. Es wäre zwar möglich, `from numpy import *` in Ihrem Code zu schreiben, um das ewige `np.` zu vermeiden, allerdings rate ich dringend davon ab, sich das anzugewöhnen. Der `numpy`-Namensraum ist groß und enthält eine Reihe von Funktionen, deren Namen mit den eingebauten Python-Funktionen in Konflikt geraten (wie `min` und `max`). Es ist immer eine gute Idee, sich an solchen Standardkonventionen zu orientieren.

Ein ndarray ist ein generischer, mehrdimensionaler Container für homogene Daten, das heißt, alle Elemente müssen vom selben Typ sein. Jedes Array hat einen `shape`, ein Tupel, das die Größe jeder Dimension angibt, und einen `dtype`, ein Objekt, das den Datentyp des Arrays beschreibt:

```
In [17]: data.shape
Out[17]: (2, 3)

In [18]: data.dtype
Out[18]: dtype('float64')
```

Dieses Kapitel führt Sie in die Grundlagen der Benutzung von NumPy-Arrays ein und sollte Sie ausreichend auf den Rest des Buchs vorbereiten. Es ist für viele datenanalytische Anwendungen zwar nicht nötig, ein tiefes Verständnis von NumPy zu haben, allerdings hilft es Ihnen auf dem Weg zum Guru des wissenschaftlichen Python, wenn Sie sich mit der Array-orientierten Programmierung und Denkweise vertraut machen.

Die Begriffe »Array«, »NumPy-Array« und »ndarray« beziehen sich in diesem Buch fast immer auf dasselbe: das ndarray-Objekt.

ndarrays erzeugen

Am einfachsten erzeugen Sie ein Array mit der array-Funktion. Diese akzeptiert jedes sequenzartige Objekt (einschließlich anderer Arrays) und erzeugt ein neues NumPy-Array, das die übergebenen Daten enthält. So ist zum Beispiel eine Liste ein guter Kandidat für eine Konvertierung:

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([6. , 7.5, 8. , 0. , 1. ])
```

Verschachtelte Sequenzen, wie eine Liste aus Listen gleicher Länge, werden in ein mehrdimensionales Array umgewandelt:

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Da data2 eine Liste von Listen war, hat das NumPy-Array arr2 zwei Dimensionen mit einer Form, die aus den Daten abgeleitet wurde. Wir können das bestätigen, wenn wir die Attribute ndim und shape untersuchen:

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

Wenn nichts explizit angegeben ist (mehr dazu im Abschnitt »Datentypen für ndarrays« auf Seite 109), versucht numpy.array, einen guten Datentyp für das Array abzuleiten, das es erzeugt. Der Datentyp wird in einem speziellen dtype-Metadatenobjekt gespeichert. So haben wir etwa in den vorherigen zwei Beispielen Folgendes:

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

Neben numpy.array gibt es eine Reihe weiterer Funktionen zum Anlegen neuer Arrays. numpy.zeros und numpy.ones erzeugen zum Beispiel Arrays aus Nullen bzw. Einsen bei vorgegebener Länge oder Form. numpy.empty erzeugt ein Array, ohne dessen Werte auf einen bestimmten Wert zu initialisieren. Um mit diesen Methoden ein höherdimensionales Array zu erzeugen, übergeben Sie für die Form ein Tupel:

```
In [29]: np.zeros(10)
Out[29]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])

In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],
       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

Man darf nicht davon ausgehen, dass `numpy.empty` ein Array zurückliefert, das nur Nullen enthält. Diese Funktion gibt nicht initialisierten Speicher zurück, der daher Müll enthalten kann. Sie sollten sie nur verwenden, wenn Sie planen, das neue Array mit Daten zu füllen.

`numpy.arange` ist eine Version der eingebauten Python-Funktion `range` für Arrays:

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In Tabelle 4-1 finden Sie eine kurze Liste der Standardfunktionen zum Erzeugen von Arrays. Da sich NumPy auf das numerische Rechnen konzentriert, ist der Datentyp in vielen Fällen `float64` (Gleitkommazahl), sofern nichts angegeben ist.

Tabelle 4-1: Ein paar wichtige Funktionen zum Erzeugen von Arrays

Funktion	Beschreibung
`array`	Wandelt Eingabedaten (Liste, Tupel, Array oder ein anderer Sequenztyp) in ein ndarray um, indem entweder ein Datentyp abgeleitet oder explizit ein Datentyp angegeben wird; kopiert standardmäßig die Eingabedaten.
`asarray`	Wandelt Eingabedaten in ein ndarray um, kopiert sie aber nicht, falls die Eingabe bereits ein ndarray ist.
`arange`	Wie die eingebaute `range`-Funktion, liefert aber ein ndarray statt einer Liste zurück.
`ones, ones_like`	Erzeugt ein Array nur aus Einsen mit der angegebenen Form und dem Datentyp; `ones_like` nimmt ein Array entgegen und erzeugt ein `ones`-Array der gleichen Form und des gleichen Datentyps.
`zeros, zeros_like`	Wie `ones` und `ones_like`, erzeugt aber stattdessen Arrays aus Nullen.
`empty, empty_like`	Erzeugt neue Arrays, indem es neuen Speicher belegt, diesen aber anders als `ones` und `zeros` nicht mit Werten füllt.
`full, full_like`	Erzeugt ein Array der angegebenen Form und des Datentyps mit Werten, die auf den angegebenen »Füllwert« gesetzt sind; `full_like` nimmt ein anderes Array entgegen und erzeugt ein gefülltes Array der gleichen Form und des gleichen Datentyps.

Tabelle 4-1: Ein paar wichtige Funktionen zum Erzeugen von Arrays (Fortsetzung)

Funktion	Beschreibung
`eye, identity`	Erzeugt eine quadratische N × N-Einheitsmatrix (Einsen auf der Diagonalen und ansonsten Nullen).

Datentypen für ndarrays

Der *Datentyp* oder `dtype` ist ein besonderes Objekt, das die Informationen (oder *Metadaten*, also Daten über Daten) enthält, die das ndarray benötigt, um einen Auszug aus dem Speicher als eine bestimmte Art von Daten zu identifizieren:

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')

In [36]: arr2.dtype
Out[36]: dtype('int32')
```

Datentypen sind einer der Gründe für NumPys Flexibilität bei der Interaktion mit Daten aus anderen Systemen. In den meisten Fällen erlauben sie eine direkte Zuordnung zur zugrunde liegenden Festplatten- oder Speicherrepräsentation, was es ermöglicht, binäre Datenströme zu lesen und Anbindungen an Code in maschinennahen Sprachen wie C oder FORTRAN vorzunehmen. Die numerischen Datentypen werden auf die gleiche Weise benannt: ein Typname, wie `float` oder `int`, gefolgt von einer Zahl, die die Anzahl der Bits pro Element angibt. Ein normaler Gleitkommawert mit doppelter Genauigkeit (im Prinzip das Python-Objekt `float`) nimmt 8 Bytes oder 64 Bits ein. Deshalb wird dieser Typ in NumPy als `float64` bezeichnet. Tabelle 4-2 zeigt eine vollständige Liste der von NumPy unterstützten Datentypen.

Sie müssen sich die NumPy-Datentypen nicht merken, speziell als Anfänger nicht. Oft reicht es, sich um die allgemeine Art der Daten zu kümmern, mit denen Sie es zu tun haben, also Gleitkomma, Komplex, Integer, Boolean, String oder allgemeines Python-Objekt. Wenn Sie mehr Kontrolle darüber haben müssen, wie die Daten im Speicher und auf der Festplatte abgelegt werden, vor allem bei großen Datenmengen, ist es gut zu wissen, dass Sie den Speichertyp kontrollieren können.

Tabelle 4-2: NumPy-Datentypen

Typ	Typcode	Beschreibung
`int8, uint8`	`i1, u1`	Vorzeichenbehaftete und vorzeichenlose 8-Bit-(1-Byte-)Integer-Typen.
`int16, uint16`	`i2, u2`	Vorzeichenbehaftete und vorzeichenlose 16-Bit-Integer-Typen.

Tabelle 4-2: NumPy-Datentypen (Fortsetzung)

Typ	Typcode	Beschreibung
`int32, uint32`	`i4, u4`	Vorzeichenbehaftete und vorzeichenlose 32-Bit-Integer-Typen.
`int64, uint64`	`i8, u8`	Vorzeichenbehaftete und vorzeichenlose 64-Bit-Integer-Typen.
`float16`	`f2`	Gleitkommazahlen mit halber Genauigkeit.
`float32`	`f4 oder f`	Gleitkommazahlen mit einfacher Genauigkeit; kompatibel mit einem C float.
`float64`	`f8 oder d`	Gleitkommazahlen mit doppelter Genauigkeit; kompatibel mit einem C double und dem Python-Objekt `float`.
`float128`	`f16 oder g`	Gleitkommazahlen mit erweiterter Genauigkeit.
`complex64, complex128, complex256`	`c8, c16, c32`	Komplexe Zahlen, repräsentiert durch zwei Gleitkommazahlen mit 32, 64 bzw. 128 Bits.
`bool`	`?`	Boolescher Typ; speichert die Werte `True` und `False`.
`object`	`O`	Python-Objekttyp; Wert kann ein beliebiges Python-Objekt sein.
`string_`	`S`	ASCII-String-Typ mit fester Länge (1 Byte pro Zeichen); um zum Beispiel einen String-Datentyp der Länge 10 zu erzeugen, nutzen Sie `'S10'`.
`unicode_`	`U`	Unicode-Typ fester Länge (die Anzahl der Bytes ist plattformabhängig); die Semantik der Spezifikation ist identisch mit `string_` (z. B. `'U10'`).

Es gibt sowohl vorzeichenbehaftete (*signed*) wie auch vorzeichenlose (*unsigned*) Integer-Typen, und viele werden mit diesen Begriffen nicht vertraut sein. Ein *vorzeichenbehafteter* Integer kann sowohl positive wie auch negative Zahlen enthalten, während ein *vorzeichenloser* Integer nur Werte größer oder gleich null darstellen kann. So kann beispielsweise `int8` (Signed 8-Bit-Integer) ganze Zahlen von –128 bis (einschließlich) 127 repräsentieren, während `uint8` (Unsigned 8-Bit-Integer) ganze Zahlen von 0 bis 255 aufnimmt.

Sie können ein Array ausdrücklich von einem Datentyp in einen anderen umwandeln. Für diesen auch *Casting* genannten Vorgang nutzen Sie die ndarray-Methode `astype`:

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr
Out[40]: array([1., 2., 3., 4., 5.])

In [41]: float_arr.dtype
Out[41]: dtype('float64')
```

In diesem Beispiel wurden Integer-Werte in Gleitkommazahlen umgewandelt. Beim Umwandeln von Gleitkommazahlen in Integer wird der Nachkommateil abgeschnitten:

```
In [42]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [43]: arr
Out[43]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])

In [44]: arr.astype(np.int32)
Out[44]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

Falls Sie ein Array aus Strings haben, die Zahlen darstellen, können Sie sie mit `astype` in eine numerische Form konvertieren:

```
In [45]: numeric_strings = np.array(["1.25", "-9.6", "42"], dtype=np.string_)

In [46]: numeric_strings.astype(float)
Out[46]: array([ 1.25, -9.6 , 42.  ])
```

Seien Sie vorsichtig bei der Benutzung des Typs `numpy.string_`. String-Daten in NumPy haben nämlich eine feste Größe, und Eingaben könnten daher ohne Warnung abgeschnitten werden. pandas verhält sich bei nicht numerischen Daten deutlich intuitiver.

Sollte das Casting aus irgendeinem Grund fehlschlagen (weil etwa ein String nicht in `float64` konvertiert werden kann), wird ein `ValueError` ausgelöst. Ich war hier ein bisschen faul und habe `float` statt `np.float64` geschrieben; NumPy setzt cleverererweise für die Python-Typen seine eigenen äquivalenten Datentypen ein.

Sie können auch das `dtype`-Attribut eines anderen Arrays benutzen:

```
In [47]: int_array = np.arange(10)

In [48]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [49]: int_array.astype(calibers.dtype)
Out[49]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

Es gibt Kurzformen für die Codes, mit denen Sie sich ebenfalls auf einen `dtype` beziehen können:

```
In [50]: zeros_uint32 = np.zeros(8, dtype="u4")

In [51]: zeros_uint32
Out[51]: array([0, 0, 0, 0, 0, 0, 0, 0], dtype=uint32)
```

Der Aufruf von `astype` erzeugt *immer* ein neues Array (eine Kopie der Daten), selbst wenn der neue Datentyp identisch mit dem alten ist.

Rechnen mit NumPy-Arrays

Arrays sind wichtig, weil sie Ihnen erlauben, viele Operationen auf Daten auszuführen, ohne dass Sie `for`-Schleifen schreiben müssen. NumPy-Anwender bezeichnen dies als *Vektorisierung*. Jede arithmetische Operation zwischen Arrays gleicher Größe führt ihre Arbeit elementweise durch:

```
In [52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [53]: arr
Out[53]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [54]: arr * arr
Out[54]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

In [55]: arr - arr
Out[55]:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Arithmetische Operationen mit Skalaren propagieren das skalare Argument zu jedem Element in dem Array:

```
In [56]: 1 / arr
Out[56]:
array([[1.    , 0.5   , 0.3333],
       [0.25  , 0.2   , 0.1667]])

In [57]: arr ** 2
Out[57]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

Vergleiche zwischen Arrays derselben Größe ergeben boolesche Arrays:

```
In [58]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [59]: arr2
Out[59]:
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])

In [60]: arr2 > arr
Out[60]:
array([[False,  True, False],
       [ True, False,  True]])
```

Operationen zwischen Arrays unterschiedlicher Größe werden als *Broadcasting* bezeichnet. Wir gehen in Anhang A näher darauf ein. Für den Großteil dieses Buchs ist es nicht notwendig, ein tiefer gehendes Verständnis zum Broadcasting zu entwickeln.

Einfaches Indizieren und Slicing

Das Indizieren von Arrays in NumPy ist ein weites Feld, weil es sehr viele Möglichkeiten gibt, eine Teilmenge Ihrer Daten oder einzelne Elemente auszuwählen. Eindimensionale Arrays sind einfach; oberflächlich betrachtet, verhalten sie sich ähnlich wie Python-Listen:

```
In [61]: arr = np.arange(10)

In [62]: arr
Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [63]: arr[5]
Out[63]: 5

In [64]: arr[5:8]
Out[64]: array([5, 6, 7])

In [65]: arr[5:8] = 12

In [66]: arr
Out[66]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Wie Sie sehen können, wird ein skalarer Wert, wenn Sie ihn einem Teilbereich wie `arr[5:8] = 12` zuweisen, an die gesamte Auswahl propagiert (*Broadcasting*).

Ein erster wichtiger Unterschied zu Pythons eingebauten Listen besteht darin, dass Teilbereiche (Slices) von Arrays sogenannte *Views* auf das ursprüngliche Array sind. Das bedeutet, dass die Daten nicht kopiert werden und sich alle Modifikationen an dem View im Quell-Array niederschlagen.

Als Beispiel erzeuge ich zuerst ein Slice von `arr`:

```
In [67]: arr_slice = arr[5:8]

In [68]: arr_slice
Out[68]: array([12, 12, 12])
```

Wenn ich nun die Werte in `arr_slice` ändere, spiegeln sich die Änderungen im Original-Array `arr` wider:

```
In [69]: arr_slice[1] = 12345

In [70]: arr
Out[70]:
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
           9])
```

Das »leere« Slice `[:]` wird allen Werten in einem Array zugewiesen:

```
In [71]: arr_slice[:] = 64

In [72]: arr
Out[72]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

Sollten Sie neu bei NumPy sein, mag Sie das überraschen, vor allem wenn Sie andere Array-Programmiersprachen benutzt haben, die viel bereitwilliger Daten kopieren. Da NumPy mit sehr großen Arrays funktionieren soll, können Sie sich vorstellen, dass es zu Leistungs- und Speicherproblemen kommen könnte, wenn NumPy darauf bestehen würde, die Daten immer zu kopieren.

Falls Sie statt eines Views tatsächlich die Kopie eines Slice eines ndarray haben wollen, müssen Sie das Array explizit kopieren – zum Beispiel so: `arr[5:8].copy()`. Wie Sie sehen werden, funktioniert pandas genauso.

Bei höherdimensionalen Arrays haben Sie deutlich mehr Optionen. In einem zweidimensionalen Array sind die Elemente an jedem Index keine Skalare mehr, sondern eindimensionale Arrays:

```
In [73]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [74]: arr2d[2]
Out[74]: array([7, 8, 9])
```

Das bedeutet, dass auf einzelne Elemente rekursiv zugegriffen werden kann. Allerdings ist das ein bisschen zu viel Arbeit, Sie können deshalb eine kommaseparierte Liste mit Indizes übergeben, um einzelne Elemente auszuwählen. Diese sind also äquivalent:

```
In [75]: arr2d[0][2]
Out[75]: 3

In [76]: arr2d[0, 2]
Out[76]: 3
```

Abbildung 4-1 verdeutlicht das Indizieren eines zweidimensionalen Arrays. Ich persönlich finde es hilfreich, mir die Achse 0 als die »Zeilen« und die Achse 1 als die »Spalten« des Arrays vorzustellen.

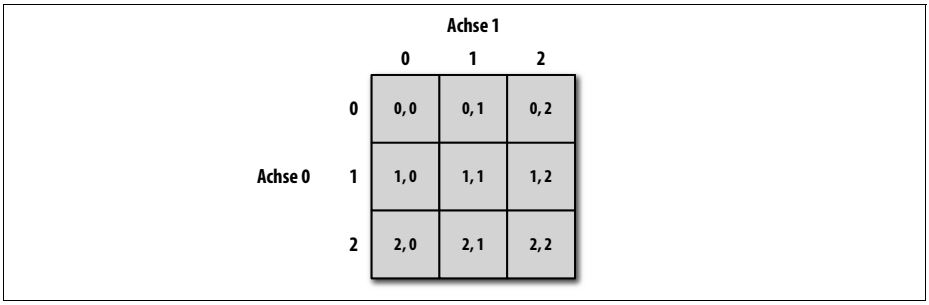

Abbildung 4-1: Indizieren der Elemente in einem NumPy-Array

Falls Sie in einem mehrdimensionalen Array die späteren Indizes weglassen, ist das zurückgelieferte Objekt ein niedrigerdimensionales ndarray, das aus all den Daten aus den höheren Dimensionen besteht. In dem 2 × 2 × 3-Array `arr3d`:

```
In [77]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [78]: arr3d
Out[78]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

ist `arr3d[0]` also ein 2 × 3-Array:

```
In [79]: arr3d[0]
Out[79]:
array([[1, 2, 3],
       [4, 5, 6]])
```

`arr3d[0]` können sowohl skalare Werte als auch Arrays zugewiesen werden:

```
In [80]: old_values = arr3d[0].copy()

In [81]: arr3d[0] = 42

In [82]: arr3d
Out[82]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [83]: arr3d[0] = old_values

In [84]: arr3d
Out[84]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In gleicher Weise liefert Ihnen `arr3d[1, 0]` alle Werte, deren Indizes mit `(1, 0)` starten, sodass ein eindimensionales Array gebildet wird:

```
In [85]: arr3d[1, 0]
Out[85]: array([7, 8, 9])
```

Hätten wir in zwei Schritten indiziert, wären wir ebenfalls auf diesen Ausdruck gekommen:

```
In [86]: x = arr3d[1]

In [87]: x
Out[87]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [88]: x[0]
Out[88]: array([7, 8, 9])
```

Beachten Sie, dass in all den Fällen, in denen Teilbereiche des Arrays ausgewählt wurden, die zurückgelieferten Arrays Views sind.

Diese mehrdimensionale Indexsyntax für NumPy-Arrays funktioniert nicht mit normalen Python-Objekten, wie zum Beispiel Listen mit Listen.

Mit Slices indizieren

Wie eindimensionale Objekte (z. B. Python-Listen) können auch ndarrays mit der bekannten Syntax zerlegt werden:

```
In [89]: arr
Out[89]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [90]: arr[1:6]
Out[90]: array([ 1,  2,  3,  4, 64])
```

Betrachten Sie das bereits bekannte zweidimensionale Array `arr2d`. Das Slicing dieses Arrays ist ein bisschen anders:

```
In [91]: arr2d
Out[91]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [92]: arr2d[:2]
Out[92]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Wie Sie sehen, wurde es entlang der Achse 0 zerteilt, der ersten Achse. Ein Slice wählt also einen Bereich von Elementen entlang einer Achse aus. Es hilft vielleicht, wenn man den Ausdruck `arr2d[:2]` als »wähle die ersten beiden Zeilen von `arr2d`« liest.

Genau wie Sie mehrere Indizes übergeben können, können Sie auch mehrere Slices übergeben:

```
In [93]: arr2d[:2, 1:]
Out[93]:
array([[2, 3],
       [5, 6]])
```

Wenn Sie das Slicing so durchführen, bekommen Sie immer Array-Views mit der gleichen Anzahl von Dimensionen. Durch das Mischen von Integer-Indizes und Slices erhalten Sie Slices mit weniger Dimensionen.

Ich kann zum Beispiel die zweite Zeile, aber nur die ersten beiden Spalten auswählen:

```
In [94]: lower_dim_slice = arr2d[1, :2]
```

Hier ist `arr2d` zweidimensional, `lower_dim_slice` ist eindimensional, und es hat die Form eines Tupels mit einer Achsengröße:

```
In [95]: lower_dim_slice.shape
Out[95]: (2,)
```

Auf ähnliche Weise kann ich die dritte Spalte, aber nur die ersten beiden Zeilen auswählen:

```
In [96]: arr2d[:2, 2]
Out[96]: array([3, 6])
```

In Abbildung 4-2 finden Sie eine grafische Veranschaulichung des Ganzen. Der allein stehende Doppelpunkt bedeutet übrigens, dass die gesamte Achse genommen wird. Sie können deshalb folgendermaßen ein Slice der höherdimensionalen Achsen erzeugen:

```
In [97]: arr2d[:, :1]
Out[97]:
array([[1],
       [4],
       [7]])
```

Das Zuweisen an einen Slice-Ausdruck weist natürlich die gesamte Auswahl zu:

```
In [98]: arr2d[:2, 1:] = 0

In [99]: arr2d
Out[99]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

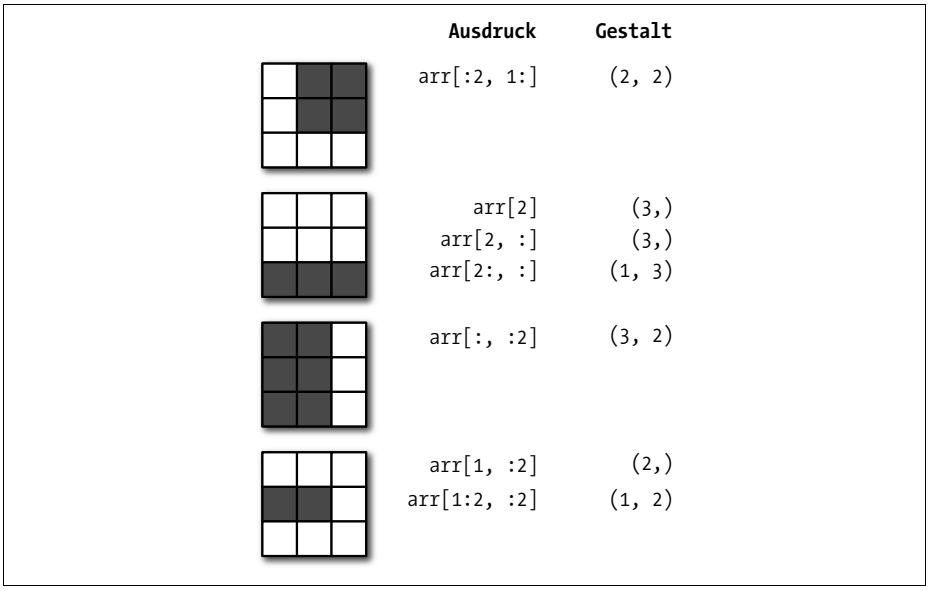

Abbildung 4-2: Zweidimensionales Slicing von Arrays

Boolesches Indizieren

Betrachten wir ein Beispiel, in dem wir einige Daten in einem Array und ein Array aus mehrfach vorhandenen Namen haben:

```
In [100]: names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])

In [101]: data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2],
   .....:                  [-12, -4], [3, 4]])

In [102]: names
Out[102]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')

In [103]: data
Out[103]:
array([[  4,   7],
       [  0,   2],
       [ -5,   6],
       [  0,   0],
       [  1,   2],
       [-12,  -4],
       [  3,   4]])
```

Nehmen wir an, jeder Name entspricht einer Zeile im Array `data`. Wir wollen alle Zeilen mit dem korrespondierenden Namen `'Bob'` auswählen. Genau wie die arithmetischen Operationen sind auch Vergleiche (wie etwa `==`) mit Arrays vektorisiert. Das bedeutet, ein Vergleich von `names` mit dem String `'Bob'` ergibt ein boolesches Array:

```
In [104]: names == "Bob"
Out[104]: array([ True, False, False,  True, False, False, False])
```

Dieses boolesche Array kann beim Indizieren des Arrays übergeben werden:

```
In [105]: data[names == "Bob"]
Out[105]:
array([[4, 7],
       [0, 0]])
```

Das boolesche Array muss die gleiche Länge haben wie die Array-Achse, die indiziert werden soll. Sie können boolesche Arrays sogar mit Slices und Integer-Werten (oder Sequenzen von Integer-Werten, mehr dazu später) kombinieren.

In diesen Beispielen wähle ich aus den Zeilen aus, in denen `names == "Bob"` ist, und indiziere außerdem die Spalten:

```
In [106]: data[names == "Bob", 1:]
Out[106]:
array([[7],
       [0]])

In [107]: data[names == "Bob", 1]
Out[107]: array([7, 0])
```

Um alles bis auf "Bob" auszuwählen, können Sie entweder != einsetzen oder die Bedingung mit ~ negieren:

```
In [108]: names != "Bob"
Out[108]: array([False,  True,  True, False,  True,  True,  True])

In [109]: ~(names == "Bob")
Out[109]: array([False,  True,  True, False,  True,  True,  True])

In [110]: data[~(names == "Bob")]
Out[110]:
array([[  0,   2],
       [ -5,   6],
       [  1,   2],
       [-12,  -4],
       [  3,   4]])
```

Der Operator ~ ist ganz nützlich, wenn Sie ein boolesches Array invertieren wollen, das von einer Variablen referenziert wird:

```
In [111]: cond = names == "Bob"

In [112]: data[~cond]
Out[112]:
array([[  0,   2],
       [ -5,   6],
       [  1,   2],
       [-12,  -4],
       [  3,   4]])
```

Zum Auswählen von zwei der drei Namen nutzen Sie boolesche Operatoren wie & (und) und | (oder):

```
In [113]: mask = (names == "Bob") | (names == "Will")

In [114]: mask
Out[114]: array([ True, False,  True,  True,  True, False, False])

In [115]: data[mask]
Out[115]:
array([[ 4,  7],
       [-5,  6],
       [ 0,  0],
       [ 1,  2]])
```

Die Auswahl von Daten aus einem Array durch boolesche Indizierung und das Zuweisen des Ergebnisses zu einer neuen Variablen erzeugt *immer* eine Kopie der Daten, selbst wenn das zurückgegebene Array unverändert ist.

Die Python-Schlüsselwörter and und or funktionieren nicht mit booleschen Arrays. Verwenden Sie stattdessen & (und) und | (oder).

Das Setzen von Werten mit booleschen Arrays funktioniert durch das Substituieren des Werts oder der Werte auf der rechten Seite an den Stellen, an denen die Werte des booleschen Arrays `True` sind. Um alle negativen Werte in `data` auf 0 zu setzen, müssen wir lediglich Folgendes tun:

```
In [116]: data[data < 0] = 0

In [117]: data
Out[117]:
array([[4, 7],
       [0, 2],
       [0, 6],
       [0, 0],
       [1, 2],
       [0, 0],
       [3, 4]])
```

Ebenso einfach ist es, ganze Zeilen oder Spalten mithilfe eines eindimensionalen booleschen Arrays zu setzen:

```
In [118]: data[names != "Joe"] = 7

In [119]: data
Out[119]:
array([[7, 7],
       [0, 2],
       [7, 7],
       [7, 7],
       [7, 7],
       [0, 0],
       [3, 4]])
```

Wie Sie später sehen werden, lassen sich diese Arten von Operationen auf zweidimensionalen Daten bequem mit pandas erledigen.

Fancy Indexing

Fancy Indexing (so etwas wie »raffiniertes Indizieren«) ist ein von NumPy übernommener Begriff, der das Indizieren mittels Integer-Arrays beschreibt. Nehmen wir an, wir hätten ein 8 × 4-Array:

```
In [120]: arr = np.zeros((8, 4))

In [121]: for i in range(8):
   .....:     arr[i] = i

In [122]: arr
Out[122]:
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
```

```
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

Um eine Teilmenge der Zeilen in einer bestimmten Reihenfolge auszuwählen, können Sie einfach eine Liste oder ein ndarray mit Integer-Werten übergeben, die die gewünschte Reihenfolge angeben:

```
In [123]: arr[[4, 3, 0, 6]]
Out[123]:
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

Dieser Code hat hoffentlich das gemacht, was Sie erwartet haben! Negative Indizes wählen Zeilen vom Ende her aus:

```
In [124]: arr[[-3, -5, -7]]
Out[124]:
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

Wenn Sie mehrere Index-Arrays übergeben, passiert etwas geringfügig anderes; dabei wird ein eindimensionales Array aus Elementen ausgewählt, die den einzelnen Tupeln der Indizes entsprechen:

```
In [125]: arr = np.arange(32).reshape((8, 4))

In [126]: arr
Out[126]:
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],
       [28, 29, 30, 31]])

In [127]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[127]: array([ 4, 23, 29, 10])
```

Wir schauen uns die `reshape`-Methode in Anhang A genauer an.

Hier wurden die Elemente `(1, 0)`, `(5, 3)`, `(7, 1)` und `(2, 2)` ausgewählt. Egal wie viele Dimensionen das Array hat – die Ergebnisse des Fancy Indexing sind immer eindimensional.

Das Fancy Indexing verhält sich in diesem Fall etwas anders, als es manche Anwender (und ich nehme mich da nicht aus) erwarten würden: die rechteckige Region, die durch die Auswahl einer Teilmenge der Zeilen und Spalten der Matrix gebildet wird. Hier ist eine Möglichkeit, das zu erreichen:

```
In [128]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[128]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Denken Sie daran, dass Fancy Indexing im Gegensatz zum Slicing die Daten immer in ein neues Array kopiert, wenn Sie das Ergebnis in eine neue Variable kopieren. Weisen Sie Werte durch Fancy Indexing zu, werden die indexierten Werte verändert:

```
In [129]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[129]: array([ 4, 23, 29, 10])

In [130]: arr[[1, 5, 7, 2], [0, 3, 1, 2]] = 0

In [131]: arr
Out[131]:
array([[ 0,  1,  2,  3],
       [ 0,  5,  6,  7],
       [ 8,  9,  0, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22,  0],
       [24, 25, 26, 27],
       [28,  0, 30, 31]])
```

Arrays transponieren und Achsen tauschen

Transponieren ist eine besondere Art des Umformens, die ebenfalls einen View der zugrunde liegenden Daten zurückliefert, ohne etwas zu kopieren. Arrays besitzen die `transpose`-Methode sowie das besondere `T`-Attribut:

```
In [132]: arr = np.arange(15).reshape((3, 5))

In [133]: arr
Out[133]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [134]: arr.T
Out[134]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

Bei Matrixberechnungen werden Sie das recht oft machen – zum Beispiel wenn Sie mit der `numpy.dot`-Funktion das innere Matrixprodukt berechnen:

```
In [135]: arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2], [-1, 0, -1], [1, 0, 1]])

In [136]: arr
Out[136]:
array([[ 0,  1,  0],
       [ 1,  2, -2],
       [ 6,  3,  2],
       [-1,  0, -1],
       [ 1,  0,  1]])

In [137]: np.dot(arr.T, arr)
Out[137]:
array([[39, 20, 12],
       [20, 14,  2],
       [12,  2, 10]])
```

Der Infix-Operator `@` ist eine weitere Variante der Matrixmultiplikation:

```
In [138]: arr.T @ arr
Out[138]:
array([[39, 20, 12],
       [20, 14,  2],
       [12,  2, 10]])
```

Das einfache Transponieren mit `.T` ist ein Sonderfall des Vertauschens von Achsen. ndarray besitzt die Methode `swapaxes`, die ein Paar aus Achsennummern entgegennimmt und die angegebenen Achsen vertauscht, um die Daten neu anzuordnen:

```
In [139]: arr
Out[139]:
array([[ 0,  1,  0],
       [ 1,  2, -2],
       [ 6,  3,  2],
       [-1,  0, -1],
       [ 1,  0,  1]])

In [140]: arr.swapaxes(0, 1)
Out[140]:
array([[ 0,  1,  6, -1,  1],
       [ 1,  2,  3,  0,  0],
       [ 0, -2,  2, -1,  1]])
```

`swapaxes` liefert ebenfalls einen View der Daten zurück, ohne eine Kopie herzustellen.

4.2 Erzeugen von Pseudozufallszahlen

Das Modul `numpy.random` ergänzt das eingebaute Python-Modul `random` um Funktionen, die effizient ganze Arrays mit Zufallswerten aus vielen Arten von Wahrscheinlichkeitsverteilungen erzeugen. So erhalten Sie zum Beispiel mit `numpy.random.standard_normal` ein 4 × 4-Array mit Zufallswerten aus der Standardnormalverteilung:

```
In [141]: samples = np.random.standard_normal(size=(4, 4))

In [142]: samples
Out[142]:
array([[-0.2047,  0.4789, -0.5194, -0.5557],
       [ 1.9658,  1.3934,  0.0929,  0.2817],
       [ 0.769 ,  1.2464,  1.0072, -1.2962],
       [ 0.275 ,  0.2289,  1.3529,  0.8864]])
```

Pythons eingebautes `random`-Modul generiert im Gegensatz dazu zeitgleich immer nur eine Stichprobe. Wie Ihnen dieser Benchmark zeigt, ist `numpy.random` beim Generieren sehr großer Stichproben deutlich mehr als eine Größenordnung schneller:

```
In [143]: from random import normalvariate

In [144]: N = 1_000_000

In [145]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.05 s +- 14.5 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [146]: %timeit np.random.standard_normal(N)
21.8 ms +- 212 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Diese Zufallszahlen sind nicht wirklich zufällig (sondern *pseudozufällig*), sie werden von einem konfigurierbaren Zufallszahlengenerator erzeugt, der deterministisch die Werte bestimmt. Funktionen wie `numpy.random.standard_normal` nutzen den Standardzufallszahlengenerator des Moduls `numpy.random`, aber Sie können Ihren Code so anpassen, dass er einen expliziten Generator verwendet:

```
In [147]: rng = np.random.default_rng(seed=12345)

In [148]: data = rng.standard_normal((2, 3))
```

Das Argument `seed` legt den initialen Status des Generators fest – dieser ändert sich jedes Mal, wenn das `rng`-Objekt zum Generieren von Daten verwendet wird. Das Generator-Objekt `rng` ist zudem von anderem Code isoliert, der ebenfalls eventuell das Modul `numpy.random` nutzt:

```
In [149]: type(rng)
Out[149]: numpy.random._generator.Generator
```

In Tabelle 4-3 finden Sie eine unvollständige Liste der in Zufallszahlengenerator-Objekten wie `rng` verfügbaren Methoden. Ich werde das oben erzeugte `rng`-Objekt im Rest des Kapitels verwenden, um Zufallszahlen zu erzeugen.

Tabelle 4-3: Methoden von NumPy-Zufallszahlengeneratoren

Methode	Beschreibung
`seed`	Setzt den Startwert des Zufallszahlengenerators.
`permutation`	Liefert eine zufällige Permutation einer Sequenz oder einen permutierten Bereich zurück.
`shuffle`	Vermischt eine Sequenz an Ort und Stelle.

Tabelle 4-3: Methoden von NumPy-Zufallszahlengeneratoren (Fortsetzung)

Methode	Beschreibung
`uniform`	Zieht Stichproben aus einer Gleichverteilung.
`integers`	Zieht zufällige Ganzzahlen aus einem gegebenen Intervall.
`standard_normal`	Zieht Stichproben aus einer Normalverteilung mit dem Mittelwert 0 und der Standardabweichung 1.
`binomial`	Zieht Stichproben aus einer Binomialverteilung.
`normal`	Zieht Stichproben aus einer Normalverteilung (einer gaussschen Verteilung).
`beta`	Zieht Stichproben aus einer Beta-Verteilung.
`chisquare`	Zieht Stichproben aus einer Chi-Quadrat-Verteilung.
`gamma`	Zieht Stichproben aus einer Gamma-Verteilung.
`uniform`	Zieht Stichproben aus einer [0, 1)-Gleichverteilung.

4.3 Universelle Funktionen: schnelle elementweise Array-Funktionen

Eine universelle Funktion oder *ufunc* ist eine Funktion, die auf den Daten in ndarrays elementweise Operationen durchführt. Stellen Sie sie sich als einen schnellen vektorisierten Wrapper für einfache Funktionen vor, der einen oder mehrere skalare Werte nimmt und ein oder mehrere skalare Ergebnisse produziert.

Viele ufuncs sind einfache elementweise Transformationen, wie `numpy.sqrt` oder `numpy.exp`:

```
In [150]: arr = np.arange(10)

In [151]: arr
Out[151]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [152]: np.sqrt(arr)
Out[152]:
array([0.    , 1.    , 1.4142, 1.7321, 2.    , 2.2361, 2.4495, 2.6458,
       2.8284, 3.    ])

In [153]: np.exp(arr)
Out[153]:
array([   1.    ,    2.7183,    7.3891,   20.0855,   54.5982,  148.4132,
        403.4288, 1096.6332, 2980.958 , 8103.0839])
```

Diese werden als *unäre* ufuncs bezeichnet. Andere, wie `numpy.add` oder `numpy.maximum`, nehmen zwei Arrays entgegen (daher *binäre* ufuncs) und liefern ein einzelnes Array als Ergebnis zurück:

```
In [154]: x = rng.standard_normal(8)

In [155]: y = rng.standard_normal(8)
```

```
In [156]: x
Out[156]:
array([-1.3678,  0.6489,  0.3611, -1.9529,  2.3474,  0.9685, -0.7594,
        0.9022])

In [157]: y
Out[157]:
array([-0.467 , -0.0607,  0.7888, -1.2567,  0.5759,  1.399 ,  1.3223,
       -0.2997])

In [158]: np.maximum(x, y)
Out[158]:
array([-0.467 ,  0.6489,  0.7888, -1.2567,  2.3474,  1.399 ,  1.3223,
        0.9022])
```

Hier hat `numpy.maximum` das elementweise Maximum der Elemente in `x` und `y` berechnet.

Es ist zwar nicht üblich, aber eine ufunc kann auch mehrere Arrays zurückgeben. `numpy.modf` ist ein Beispiel dafür, eine vektorisierte Version der eingebauten Python-Funktion `math.modf`; es liefert die gebrochenen und ganzzahligen Teile eines Gleitkomma-Arrays zurück:

```
In [159]: arr = rng.standard_normal(7) * 5

In [160]: arr
Out[160]: array([ 4.5146, -8.1079, -0.7909,  2.2474, -6.718 , -0.4084,  8.6237])

In [161]: remainder, whole_part = np.modf(arr)

In [162]: remainder
Out[162]: array([ 0.5146, -0.1079, -0.7909,  0.2474, -0.718 , -0.4084,  0.6237])

In [163]: whole_part
Out[163]: array([ 4., -8., -0.,  2., -6., -0.,  8.])
```

Ufuncs akzeptieren ein optionales `out`-Argument, das ihnen erlaubt, ihre Ergebnisse einem bestehenden Array zuzuweisen, statt ein neues zu erstellen:

```
In [164]: arr
Out[164]: array([ 4.5146, -8.1079, -0.7909,  2.2474, -6.718 , -0.4084,  8.6237])

In [165]: out = np.zeros_like(arr)

In [166]: np.add(arr, 1)
Out[166]: array([ 5.5146, -7.1079,  0.2091,  3.2474, -5.718 ,  0.5916,  9.6237])

In [167]: np.add(arr, 1, out=out)
Out[167]: array([ 5.5146, -7.1079,  0.2091,  3.2474, -5.718 ,  0.5916,  9.6237])

In [168]: out
Out[168]: array([ 5.5146, -7.1079,  0.2091,  3.2474, -5.718 ,  0.5916,  9.6237])
```

In den Tabellen 4-4 und 4-5 finden Sie einige der ufuncs aus NumPy. Es werden immer wieder neue hinzugefügt, daher sollten Sie auch einen Blick in die Online-

dokumentation von NumPy werfen, um auf dem neuesten Stand zu bleiben und eine vollständige Liste zu erhalten.

Tabelle 4-4: Einige unäre universelle Funktionen

Funktion	Beschreibung
`abs, fabs`	Berechnet elementweise den absoluten Wert für Integer-, Gleitkomma- oder komplexe Werte.
`sqrt`	Berechnet für jedes Element die Quadratwurzel (äquivalent zu `arr ** 0.5`).
`square`	Berechnet das Quadrat jedes Elements (äquivalent zu `arr ** 2`).
`exp`	Berechnet den Exponenten e^x jedes Elements.
`log, log10, log2, log1p`	Natürlicher Logarithmus (Basis *e*), log Basis 10, log Basis 2 bzw. log(1 + x).
`sign`	Berechnet das Vorzeichen jedes Elements: 1 (positiv), 0 (null) oder –1 (negativ).
`ceil`	Rundet jedes Element auf die kleinste Integer-Zahl auf, die größer oder gleich diesem Element ist.
`floor`	Rundet jedes Element auf die größte Integer-Zahl ab, die kleiner oder gleich diesem Element ist.
`rint`	Rundet Elemente auf die nächstgelegene Integer-Zahl, wobei der `dtype` beibehalten wird.
`modf`	Gibt die gebrochenen und ganzzahligen Anteile eines Arrays als separate Arrays zurück.
`isnan`	Gibt ein boolesches Array zurück, das angibt, ob die einzelnen Werte `NaN` (Not a Number, keine Zahl) sind.
`isfinite, isinf`	Gibt ein boolesches Array zurück, das angibt, ob die einzelnen Werte endlich (nicht-`inf`, nicht-`NaN`) bzw. unendlich sind.
`cos, cosh, sin, sinh, tan, tanh`	Reguläre trigonometrische und Hyperbelfunktionen.
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometrische Funktionen.
`logical_not`	Berechnet elementweise den Wahrheitswert von `not x` (äquivalent zu `~arr`).

Tabelle 4-5: Einige binäre universelle Funktionen

Funktion	Beschreibung
`add`	Addiert korrespondierende Elemente in Arrays.
`subtract`	Subtrahiert die Elemente im zweiten Array vom ersten Array.
`multiply`	Multipliziert die Array-Elemente miteinander.
`divide, floor_divide`	Dividieren oder Floor-Dividieren (Abschneiden des Rests).
`power`	Potenziert Elemente im ersten Array entsprechend dem Exponenten im zweiten Array.
`maximum, fmax`	Elementweises Maximum; `fmax` ignoriert `NaN`.
`minimum, fmin`	Elementweises Minimum; `fmin` ignoriert `NaN`.
`mod`	Elementweiser Modulus (Rest der Division).
`copysign`	Kopiert die Vorzeichen der Werte im zweiten Argument auf die Werte im ersten Argument.

Tabelle 4-5: Einige binäre universelle Funktionen (Fortsetzung)

Funktion	Beschreibung
`greater, greater_equal, less, less_equal, equal, not_equal`	Führt elementweise Vergleiche durch, die ein boolesches Array ergeben (äquivalent zu den Infix-Operatoren `>, >=, <, <=, ==, !=`).
`logical_and`	Berechnet elementweise die Wahrheitswerte von AND (&).
`logical_or`	Berechnet elementweise die Wahrheitswerte von OR (|).
`logical_xor`	Berechnet elementweise die Wahrheitswerte von XOR (^).

4.4 Array-orientierte Programmierung mit Arrays

Mit NumPy-Arrays können Sie viele Arten von Datenverarbeitungsaufgaben in Form präziser Array-Ausdrücke schreiben, für die ansonsten Schleifen erforderlich wären. Dieses Ersetzen expliziter Schleifen durch Array-Ausdrücke bezeichnet man gelegentlich als *Vektorisierung*. Im Allgemeinen sind vektorisierte Array-Operationen deutlich schneller als ihre reinen Python-Gegenstücke. Ihre größte Wirkung zeigen sie bei allen Arten von numerischen Berechnungen. Später in Anhang A erläutere ich das *Broadcasting*, eine leistungsstarke Methode für vektorisierte Berechnungen.

Als ein einfaches Beispiel nehmen wir einmal an, dass wir die Funktion `sqrt(x^2 + y^2)` über ein normales Gitter von Werten evaluieren wollten. Die Funktion `numpy.meshgrid` nimmt zwei eindimensionale Arrays entgegen und erzeugt zwei zweidimensionale Matrizen entsprechend den Paaren von `(x, y)` in den beiden Arrays:

```
In [169]: points = np.arange(-5, 5, 0.01) # 100 equally spaced points

In [170]: xs, ys = np.meshgrid(points, points)

In [171]: ys
Out[171]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Zum Auswerten der Funktion schreiben Sie einfach den gleichen Ausdruck auf, den Sie auch bei zwei Punkten schreiben würden:

```
In [172]: z = np.sqrt(xs ** 2 + ys ** 2)

In [173]: z
Out[173]:
array([[7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],
       [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],
       [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
```

```
...,
[7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],
[7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
[7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])
```

In einer Vorschau auf Kapitel 9 nutze ich matplotlib, um Visualisierungen dieses zweidimensionalen Arrays herzustellen:

```
In [174]: import matplotlib.pyplot as plt

In [175]: plt.imshow(z, cmap=plt.cm.gray, extent=[-5, 5, -5, 5])
Out[175]: <matplotlib.image.AxesImage at 0x7f7132db3ac0>

In [176]: plt.colorbar()
Out[176]: <matplotlib.colorbar.Colorbar at 0x7f713a5833a0>

In [177]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
Out[177]: Text(0.5, 1.0, 'Image plot of $\\sqrt{x^2 + y^2}$ for a grid of values')
```

In Abbildung 4-3 habe ich die matplotlib-Funktion `imshow` genommen, um aus dem zweidimensionalen Array von Funktionswerten ein Bild zeichnen zu lassen.

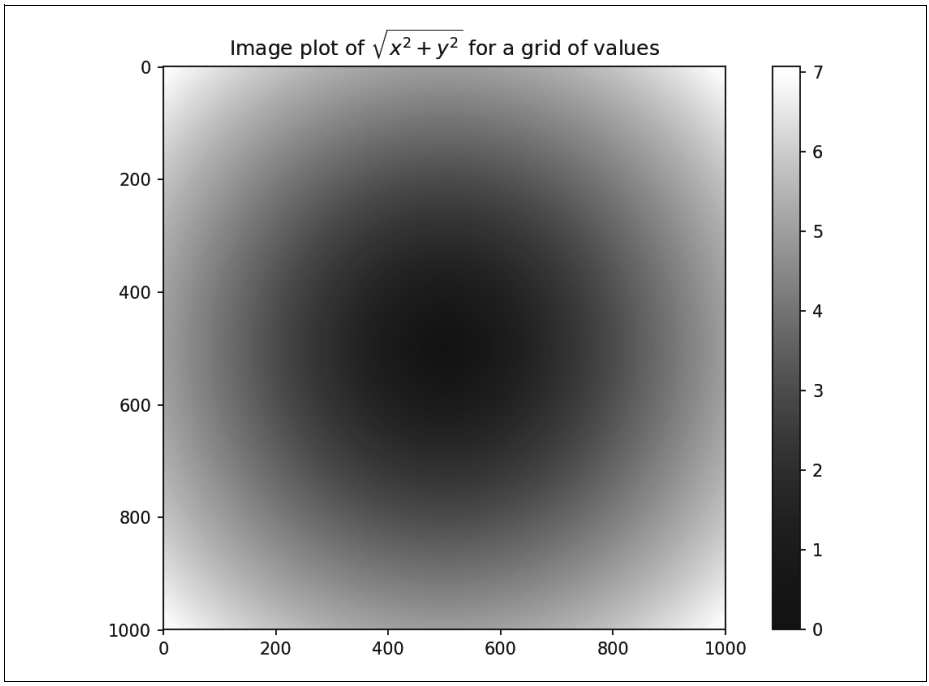

Abbildung 4-3: Plot der Funktionsauswertung am Grid

Arbeiten Sie in IPython, können Sie alle offenen Plotfenster durch `plt.close("all")` schließen:

```
In [179]: plt.close("all")
```

Der Begriff *Vektorisierung* wird verwendet, um andere Konzepte in der Informatik zu beschreiben, aber in diesem Buch nutze ich ihn für zeitgleiche Operationen auf ganzen Daten-Arrays, statt mit einer Python-`for`-Schleife jeden Wert einzeln zu verarbeiten.

Bedingte Logik als Array-Operationen ausdrücken

Die Funktion `numpy.where` ist eine vektorisierte Version des ternären Ausdrucks `x if condition else y`. Nehmen wir an, wir hätten ein boolesches Array und zwei Arrays mit Werten:

```
In [180]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [181]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [182]: cond = np.array([True, False, True, True, False])
```

Nun wollen wir einen Wert aus `xarr` nehmen, wenn der korrespondierende Wert in `cond` `True` ist. Andernfalls soll der Wert aus `yarr` genommen werden. Eine List Comprehension, die das erledigt, könnte so aussehen:

```
In [183]: result = [(x if c else y)
   .....:           for x, y, c in zip(xarr, yarr, cond)]

In [184]: result
Out[184]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

Das bringt mehrere Probleme mit sich. Zum einen ist es für große Arrays nicht besonders schnell (weil die ganze Arbeit in interpretiertem Python-Code ausgeführt wird), und zum anderen funktioniert es nicht mit mehrdimensionalen Arrays. Mit `numpy.where` können Sie dies mit einem einzelnen Funktionsaufruf erreichen:

```
In [185]: result = np.where(cond, xarr, yarr)

In [186]: result
Out[186]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Das zweite bzw. dritte Argument zu `numpy.where` muss kein Array sein, eines oder beide können Skalare sein. Eine typische Anwendung von `where` in der Datenanalyse ist es, ein neues Array aus Werten auf der Basis eines anderen Arrays zu erzeugen. Stellen Sie sich vor, Sie hätten eine Matrix aus zufällig generierten Daten und wollten alle positiven Werte durch 2 und alle negativen Werte durch –2 ersetzen. Mit `numpy.where` ist das möglich:

```
In [187]: arr = rng.standard_normal((4, 4))

In [188]: arr
Out[188]:
array([[ 2.6182,  0.7774,  0.8286, -0.959 ],
       [-1.2094, -1.4123,  0.5415,  0.7519],
       [-0.6588, -1.2287,  0.2576,  0.3129],
       [-0.1308,  1.27  , -0.093 , -0.0662]])
```

```
In [189]: arr > 0
Out[189]:
array([[ True,  True,  True, False],
       [False, False,  True,  True],
       [False, False,  True,  True],
       [False,  True, False, False]])

In [190]: np.where(arr > 0, 2, -2)
Out[190]:
array([[ 2,  2,  2, -2],
       [-2, -2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2, -2, -2]])
```

Wenn Sie `numpy.where` verwenden, können Sie Skalare und Arrays kombinieren. Zum Beispiel kann ich so alle positiven Werte in `arr` durch die Konstante 2 ersetzen:

```
In [191]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[191]:
array([[ 2.    ,  2.    ,  2.    , -0.959 ],
       [-1.2094, -1.4123,  2.    ,  2.    ],
       [-0.6588, -1.2287,  2.    ,  2.    ],
       [-0.1308,  2.    , -0.093 , -0.0662]])
```

Mathematische und statistische Methoden

Eine Reihe mathematischer Funktionen, die Statistiken über ein ganzes Array oder über die Daten entlang einer Achse berechnen, sind als Methoden der Array-Klasse verfügbar. Aggregationen (manchmal auch *Reduktionen* genannt) wie die Summe `sum`, der Mittelwert `mean` und die Standardabweichung `std` können entweder als Methoden der Array-Instanz oder auf oberster Ebene als NumPy-Funktion aufgerufen werden. Nutzen Sie die NumPy-Funktion, wie zum Beispiel `numpy.sum`, müssen Sie das zu aggregierende Array als erstes Argument übergeben.

Hier generiere ich einige normalverteilte Zufallsdaten und berechne ein paar Statistiken:

```
In [192]: arr = rng.standard_normal((5, 4))

In [193]: arr
Out[193]:
array([[-1.1082,  0.136 ,  1.3471,  0.0611],
       [ 0.0709,  0.4337,  0.2775,  0.5303],
       [ 0.5367,  0.6184, -0.795 ,  0.3   ],
       [-1.6027,  0.2668, -1.2616, -0.0713],
       [ 0.474 , -0.4149,  0.0977, -1.6404]])

In [194]: arr.mean()
Out[194]: -0.08719744457434529

In [195]: np.mean(arr)
Out[195]: -0.08719744457434529
```

```
In [196]: arr.sum()
Out[196]: -1.743948891486906
```

Funktionen wie `mean` und `sum` akzeptieren ein optionales `axis`-Argument, sodass die Statistiken entlang der angegebenen Achse berechnet werden. Das Ergebnis ist ein Array mit einer Dimension weniger:

```
In [197]: arr.mean(axis=1)
Out[197]: array([ 0.109 ,  0.3281,  0.165 , -0.6672, -0.3709])

In [198]: arr.sum(axis=0)
Out[198]: array([-1.6292,  1.0399, -0.3344, -0.8203])
```

Hier bedeutet `arr.mean(axis=1)`: »Berechne den Mittelwert über die Spalten.«, während `arr.sum(axis=0)` bedeutet: »Berechne die Summe entlang der Zeilen.«

Andere Methoden wie `cumsum` und `cumprod` aggregieren nicht, sondern erzeugen ein Array der Zwischenergebnisse:

```
In [199]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [200]: arr.cumsum()
Out[200]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

In mehrdimensionalen Arrays liefern Akkumulationsfunktionen wie `cumsum` ein Array der gleichen Größe zurück, allerdings mit Teilergebnissen entlang der Achsen entsprechend den niedriger dimensionierten Slices:

```
In [201]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [202]: arr
Out[202]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Der Ausdruck `arr.cumsum(axis=0)` berechnet die kumulative Summe entlang der Zeilen, während `arr.cumsum(axis=1)` die Summen entlang der Spalten ermittelt:

```
In [203]: arr.cumsum(axis=0)
Out[203]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [204]: arr.cumsum(axis=1)
Out[204]:
array([[ 0,  1,  3],
       [ 3,  7, 12],
       [ 6, 13, 21]])
```

In Tabelle 4-6 finden Sie eine vollständige Liste. In späteren Kapiteln werden wir dann Beispiele dieser Methoden in Aktion sehen.

Tabelle 4-6: Grundlegende statistische Array-Methoden

Methode	Beschreibung
`sum`	Summe aller Elemente in dem Array oder entlang einer Achse; Arrays der Länge null ergeben die Summe 0.
`mean`	Arithmetischer Mittelwert; ungültig für Arrays der Länge null (liefert `NaN`).
`std, var`	Standardabweichung bzw. Varianz.
`min, max`	Minimum und Maximum.
`argmin, argmax`	Indizes der minimalen bzw. maximalen Elemente.
`cumsum`	Kumulative Summe der Elemente, beginnend bei 0.
`cumprod`	Kumulatives Produkt der Elemente, beginnend bei 1.

Methoden für boolesche Arrays

Boolesche Werte werden in den vorgestellten Methoden als 1 (`True`) und 0 (`False`) behandelt. Daher wird `sum` oft als Möglichkeit genutzt, `True`-Werte in einem booleschen Array zu zählen:

```
In [205]: arr = rng.standard_normal(100)

In [206]: (arr > 0).sum() # Number of positive values
Out[206]: 48

In [207]: (arr <= 0).sum() # Number of non-positive values
Out[207]: 52
```

Die Klammern im Ausdruck `(arr > 0).sum()` sind erforderlich, um `sum()` auf dem temporären Ergebnis von `arr > 0` aufrufen zu können.

Es gibt zwei weitere Methoden, `any` und `all`, die besonders bei booleschen Arrays nützlich sind. `any` testet, ob ein oder mehrere Werte in einem Array `True` sind, während `all` prüft, ob alle Werte `True` sind:

```
In [208]: bools = np.array([False, False, True, False])

In [209]: bools.any()
Out[209]: True

In [210]: bools.all()
Out[210]: False
```

Diese Methoden funktionieren auch mit nicht booleschen Arrays, bei denen Werte, die nicht 0 sind, zu `True` ausgewertet werden.

Sortieren

Wie Pythons eingebauter Listentyp können auch NumPy-Arrays mit der Methode `sort` an Ort und Stelle sortiert werden:

```
In [211]: arr = rng.standard_normal(6)

In [212]: arr
Out[212]: array([ 0.0773, -0.6839, -0.7208,  1.1206, -0.0548, -0.0824])

In [213]: arr.sort()

In [214]: arr
Out[214]: array([-0.7208, -0.6839, -0.0824, -0.0548,  0.0773,  1.1206])
```

Bei einem mehrdimensionalen Array können Sie jeden eindimensionalen Bereich mit Werten an Ort und Stelle entlang einer Achse sortieren, wenn Sie die Achsennummer an `sort` übergeben:

```
In [215]: arr = rng.standard_normal((5, 3))

In [216]: arr
Out[216]:
array([[ 0.936 ,  1.2385,  1.2728],
       [ 0.4059, -0.0503,  0.2893],
       [ 0.1793,  1.3975,  0.292 ],
       [ 0.6384, -0.0279,  1.3711],
       [-2.0528,  0.3805,  0.7554]])
```

`arr.sort(axis=0)` sortiert die Werte in jeder Spalte, während `arr.sort(axis=1)` in jeder Zeile sortiert:

```
In [217]: arr.sort(axis=0)

In [218]: arr
Out[218]:
array([[-2.0528, -0.0503,  0.2893],
       [ 0.1793, -0.0279,  0.292 ],
       [ 0.4059,  0.3805,  0.7554],
       [ 0.6384,  1.2385,  1.2728],
       [ 0.936 ,  1.3975,  1.3711]])

In [219]: arr.sort(axis=1)

In [220]: arr
Out[220]:
array([[-2.0528, -0.0503,  0.2893],
       [-0.0279,  0.1793,  0.292 ],
       [ 0.3805,  0.4059,  0.7554],
       [ 0.6384,  1.2385,  1.2728],
       [ 0.936 ,  1.3711,  1.3975]])
```

Die Methode `numpy.sort` liefert eine sortierte Kopie eines Arrays zurück (wie die in Python eingebaute Funktion `sorted`), statt das Array direkt zu modifizieren, zum Beispiel:

```
In [221]: arr2 = np.array([5, -10, 7, 1, 0, -3])

In [222]: sorted_arr2 = np.sort(arr2)

In [223]: sorted_arr2
Out[223]: array([-10,  -3,   0,   1,   5,   7])
```

Weitere Details über die Sortiermethoden von NumPy und andere komplexere Techniken wie das indirekte Sortieren finden Sie in Anhang A. In pandas gibt es ebenfalls Arten der Datenmanipulation, die mit dem Sortieren zu tun haben (wie etwa das Sortieren einer Datentabelle nach einer oder mehreren Spalten).

Unique und andere Mengenlogik

NumPy besitzt einige elementare Mengenoperationen für eindimensionale ndarrays. Häufig benutzt wird `numpy.unique`, die die sortierten eindeutigen Werte in einem Array zurückliefert:

```
In [224]: names = np.array(["Bob", "Will", "Joe", "Bob", "Will", "Joe", "Joe"])

In [225]: np.unique(names)
Out[225]: array(['Bob', 'Joe', 'Will'], dtype='<U4')

In [226]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [227]: np.unique(ints)
Out[227]: array([1, 2, 3, 4])
```

Stellen Sie `numpy.unique` einmal der reinen Python-Alternative gegenüber:

```
In [228]: sorted(set(names))
Out[228]: ['Bob', 'Joe', 'Will']
```

In vielen Fällen ist die NumPy-Version schneller, zudem liefert sie ein NumPy-Array anstelle einer Python-Liste zurück.

Eine andere Funktion, `numpy.in1d`, prüft das Vorhandensein der Werte aus einem Array in einem anderen und liefert ein boolesches Array zurück:

```
In [229]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [230]: np.in1d(values, [2, 3, 6])
Out[230]: array([ True, False, False,  True,  True, False,  True])
```

In Tabelle 4-7 finden Sie eine Liste der Array-Mengenfunktionen in NumPy.

Tabelle 4-7: Mengenoperationen bei Arrays

Methode	Beschreibung
`unique(x)`	Berechnet die sortierten eindeutigen Elemente in `x`.
`intersect1d(x, y)`	Berechnet die sortierten gemeinsamen Elemente in `x` und `y`.
`union1d(x, y)`	Berechnet die sortierte Vereinigung der Elemente.
`in1d(x, y)`	Berechnet ein boolesches Array, das angibt, ob die einzelnen Elemente aus `x` in `y` enthalten sind.
`setdiff1d(x, y)`	Mengendifferenz, Elemente in `x`, die nicht in `y` enthalten sind.
`setxor1d(x, y)`	Symmetrische Mengendifferenzen, Elemente, die in einem der Arrays, aber nicht in beiden enthalten sind.

4.5 Dateiein- und -ausgabe bei Arrays

NumPy ist in der Lage, Daten in einigen Text- und Binärformaten auf der Festplatte zu speichern und von ihr zu laden. Ich werde in diesem Abschnitt nur über das in NumPy eingebaute Binärformat sprechen, da die meisten Anwender bevorzugt pandas und andere Werkzeuge zum Laden von Text- oder Tabellendaten benutzen werden (siehe Kapitel 6 für weitere Informationen).

`numpy.save` und `numpy.load` sind die zwei Arbeitspferde zum effizienten Speichern und Laden von Array-Daten auf und von der Platte. Arrays werden standardmäßig in einem unkomprimierten rohen Binärformat mit der Dateierweiterung *.npy* gespeichert:

```
In [231]: arr = np.arange(10)

In [232]: np.save("some_array", arr)
```

Sollte der Dateipfad noch nicht auf *.npy* enden, wird die Erweiterung angehängt. Das Array auf der Platte kann dann mit `numpy.load` geladen werden:

```
In [233]: np.load("some_array.npy")
Out[233]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Zum Speichern mehrerer Arrays in einem unkomprimierten Archiv nutzen Sie `numpy.savez` und übergeben die Arrays als Schlüsselwortargumente:

```
In [234]: np.savez("array_archive.npz", a=arr, b=arr)
```

Wenn Sie eine *.npz*-Datei laden, erhalten Sie ein Dictionary-artiges Objekt zurück, das die einzelnen Arrays dann lädt (»lazy«), wenn auf sie zugegriffen wird:

```
In [235]: arch = np.load("array_archive.npz")

In [236]: arch["b"]
Out[236]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Lassen sich Ihre Daten gut komprimieren, könnten Sie stattdessen `numpy.savez_compressed` benutzen:

```
In [237]: np.savez_compressed("arrays_compressed.npz", a=arr, b=arr)
```

4.6 Lineare Algebra

Operationen der linearen Algebra sind, genau wie Matrixmultiplikationen, Dekompositionen, Determinanten und andere Mathematik mit quadratischen Matrizen, ein wichtiger Bestandteil vieler Array-Bibliotheken. Das Multiplizieren zweier zweidimensionaler Arrays mit * ist ein elementweises Produkt, während für Matrixmultiplikationen eine Funktion zum Einsatz kommen muss. Dafür gibt es die Funktion `dot` – sowohl als Array-Methode als auch als Funktion im `numpy`-Namensraum – für die Matrixmultiplikation:

```
In [241]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [242]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [243]: x
Out[243]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [244]: y
Out[244]:
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])

In [245]: x.dot(y)
Out[245]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

`x.dot(y)` ist äquivalent zu `np.dot(x, y)`:

```
In [246]: np.dot(x, y)
Out[246]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

Ein Matrixprodukt zwischen einem zweidimensionalen Array und einem eindimensionalen Array passender Größe resultiert in einem eindimensionalen Array:

```
In [247]: x @ np.ones(3)
Out[247]: array([ 6., 15.])
```

`numpy.linalg` besitzt einen Standardsatz an Funktionen zu Matrixzerlegung, Invertierungen, Determinanten und anderen Dingen:

```
In [248]: from numpy.linalg import inv, qr

In [249]: X = rng.standard_normal((5, 5))

In [250]: mat = X.T @ X

In [251]: inv(mat)
Out[251]:
array([[  3.4993,   2.8444,   3.5956, -16.5538,   4.4733],
       [  2.8444,   2.5667,   2.9002, -13.5774,   3.7678],
       [  3.5956,   2.9002,   4.4823, -18.3453,   4.7066],
       [-16.5538, -13.5774, -18.3453,  84.0102, -22.0484],
       [  4.4733,   3.7678,   4.7066, -22.0484,   6.0525]])

In [252]: mat @ inv(mat)
Out[252]:
array([[ 1.,  0., -0.,  0., -0.],
       [ 0.,  1.,  0.,  0., -0.],
       [ 0., -0.,  1., -0., -0.],
       [ 0., -0.,  0.,  1., -0.],
       [ 0., -0.,  0., -0.,  1.]])
```

Der Ausdruck `X.T.dot(X)` berechnet das Skalarprodukt von `X` mit seiner transponierten `X.T`.

In Tabelle 4-8 finden Sie eine Liste der gebräuchlichsten Funktionen für die lineare Algebra.

Tabelle 4-8: Häufig verwendete numpy.linalg-Funktionen

Funktion	Beschreibung
`diag`	Gibt die diagonalen (oder gegendiagonalen) Elemente einer quadratischen Matrix als eindimensionales Array zurück oder konvertiert ein eindimensionales Array in eine quadratische Matrix mit Nullen auf der Gegendiagonalen.
`dot`	Matrixmultiplikation.
`trace`	Berechnet die Summe der diagonalen Elemente.
`det`	Berechnet die Matrixdeterminante.
`eig`	Berechnet die Eigenwerte und Eigenvektoren einer quadratischen Matrix.
`inv`	Berechnet die Inverse einer quadratischen Matrix.
`pinv`	Berechnet die Moore-Penrose-Pseudoinverse einer Matrix.
`qr`	Berechnet die QR-Zerlegung.
`svd`	Berechnet die Singulärwertzerlegung (Singular Value Decomposition, SVD).
`solve`	Löst das lineare System Ax = b für x, wobei A eine quadratische Matrix ist.
`lstsq`	Berechnet die Lösung der kleinsten Quadrate für `Ax = b`.

4.7 Beispiel: Random Walks

Die Simulation von *Random Walks* (*https://en.wikipedia.org/wiki/Random_walk*) (Irrfahrten) zeigt anschaulich die Anwendung von Array-Operationen. Schauen wir uns zuerst einen einfachen Random Walk an, der bei 0 beginnt und bei dem die Schritte 1 und –1 mit gleicher Wahrscheinlichkeit auftreten.

Und so würde man rein in Python einen Random Walk mit 1.000 Schritten implementieren: mit dem eingebauten `random`-Modul:

```
#! blockstart
import random
position = 0
walk = [position]
nsteps = 1000
for _ in range(nsteps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
#! blockend
```

In Abbildung 4-4 sehen Sie einen Beispielplot der ersten 100 Werte eines dieser Random Walks:

```
In [255]: plt.plot(walk[:100])
```

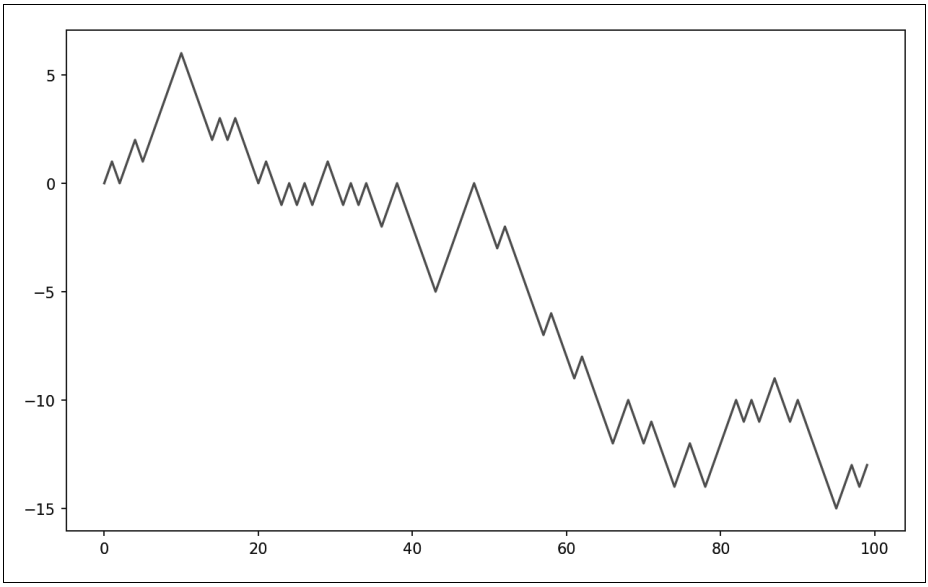

Abbildung 4-4: Eine einfache Irrfahrt

Sie werden vielleicht bemerken, dass `walk` die kumulative Summe der zufälligen Schritte ist und als Array-Ausdruck ausgewertet werden könnte. Daher benutze ich das Modul `numpy.random`, um auf einmal 1.000 Münzwürfe zu ziehen, setze diese auf 1 und –1 und berechne die kumulative Summe:

```
In [256]: nsteps = 1000

In [257]: rng = np.random.default_rng(seed=12345)  # fresh random generator

In [258]: draws = rng.integers(0, 2, size=nsteps)

In [259]: steps = np.where(draws == 0, 1, -1)

In [260]: walk = steps.cumsum()
```

Daraus können wir dann Statistiken wie das Minimum und das Maximum entlang der Bahnkurve ermitteln:

```
In [261]: walk.min()
Out[261]: -8

In [262]: walk.max()
Out[262]: 50
```

Eine kompliziertere Statistik ist die *erste Durchgangszeit*, der Schritt, an dem der Random Walk einen bestimmten Wert erreicht. Hier wollen wir vielleicht wissen, wie lang der Random Walk brauchte, um in jeder Richtung wenigstens 10 Schritte vom Ursprung 0 weg zu sein. `np.abs(walk) >= 10` liefert uns ein boolesches Array, das anzeigt, wo der Walk die 10 erreicht oder überschritten hat, allerdings wollen

wir den Index der *ersten* 10 oder –10. Das können wir mit `argmax` berechnen, das den ersten Index des Maximumwerts im booleschen Array zurückgibt (`True` ist das Maximum):

```
In [263]: (np.abs(walk) >= 10).argmax()
Out[263]: 155
```

Beachten Sie, dass `argmax` hier nicht immer effizient ist, weil es stets einen vollständigen Scan des Arrays durchführt. In diesem speziellen Fall wissen wir, dass es das Maximum ist, wenn wir einmal ein `True` festgestellt haben.

Viele Random Walks auf einmal simulieren

Wäre es unser Ziel, viele Random Walks, zum Beispiel 5.000, zu simulieren, könnten Sie mit nur geringen Änderungen an dem gezeigten Code alle Random Walks gleichzeitig generieren. Beim Übergeben eines 2-Tupels generieren die `numpy.random`-Funktionen ein zweidimensionales Array aus Ziehungen, und wir können die kumulative Summe für jede Zeile berechnen, um alle 5.000 Random Walks auf einen Schlag auszurechnen:

```
In [264]: nwalks = 5000

In [265]: nsteps = 1000

In [266]: draws = rng.integers(0, 2, size=(nwalks, nsteps)) # 0 or 1

In [267]: steps = np.where(draws > 0, 1, -1)

In [268]: walks = steps.cumsum(axis=1)

In [269]: walks
Out[269]:
array([[  1,   2,   3, ...,  22,  23,  22],
       [  1,   0,  -1, ..., -50, -49, -48],
       [  1,   2,   3, ...,  50,  49,  48],
       ...,
       [ -1,  -2,  -1, ..., -10,  -9, -10],
       [ -1,  -2,  -3, ...,   8,   9,   8],
       [ -1,   0,   1, ...,  -4,  -3,  -2]])
```

Jetzt berechnen wir die maximalen und minimalen Werte über alle Walks:

```
In [270]: walks.max()
Out[270]: 114

In [271]: walks.min()
Out[271]: -120
```

Ermitteln wir nun aus all diesen Walks die minimale Durchgangszeit bis 30 oder –30. Das ist ein bisschen schwieriger, weil nicht alle 5.000 bis 30 gekommen sind. Wir können das mit der `any`-Methode prüfen:

```
In [272]: hits30 = (np.abs(walks) >= 30).any(axis=1)

In [273]: hits30
Out[273]: array([False,  True,  True, ...,  True, False,  True])

In [274]: hits30.sum() # Number that hit 30 or -30
Out[274]: 3395
```

Mit diesem booleschen Array können wir nun die Zeilen der Walks auswählen, die tatsächlich die absolute 30 erreicht haben. Dann rufen wir `argmax` über die Achse 1 auf, damit die Durchgangszeiten festgestellt werden:

```
In [275]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(axis=1)

In [276]: crossing_times
Out[276]: array([201, 491, 283, ..., 219, 259, 541])
```

Schließlich berechnen wir die minimale durchschnittliche Durchgangszeit:

```
In [277]: crossing_times.mean()
Out[277]: 500.5699558173785
```

Experimentieren Sie selbst mit anderen Verteilungen als nur den gleichverteilten Münzwürfen für die Schritte. Sie brauchen lediglich eine andere Funktion für die Zufallszahlengenerierung, wie etwa `standard_normal`, um normalverteilte Schritte mit einstellbarem Mittelwert und Standardabweichung zu generieren:

```
In [278]: draws = 0.25 * rng.standard_normal((nwalks, nsteps))
```

Denken Sie daran, dass für diesen vektorisierten Ansatz ein Array mit `nwalks * nsteps` Elementen erstellt werden muss, was für große Simulationen viel Speicher verbrauchen kann. Ist der Speicherplatz begrenzter, wird ein anderes Vorgehen erforderlich sein.

4.8 Schlussbemerkung

Der Rest des Buchs konzentriert sich zwar vorrangig auf die Datenverarbeitung mit pandas, doch wir werden auch weiter in einem ähnlichen Array-basierten Stil arbeiten. In Anhang A steigen wir tiefer in die NumPy-Funktionalität ein, sodass Sie Ihre Fähigkeiten beim Rechnen mit Arrays weiterentwickeln können.