KAPITEL 4

Grundlagen von NumPy: Arrays und vektorisierte Berechnung

NumPy, die Kurzform von Numerical Python, ist das elementare Paket, das für wissenschaftliches Rechnen mit hoher Leistung und Datenanalyse gebraucht wird. Auf dieser Grundlage bauen so gut wie alle höher entwickelten Werkzeuge in diesem Buch auf. Hier ein paar Dinge, die es zu bieten hat:

- `ndarray`, ein schnelles und platzsparendes mehrdimensionales Array, das neben vektorisierten arithmetischen Operationen über ein ausgeklügeltes Broadcasting verfügt.
- Mathematische Standardfunktionen für schnelle Operationen auf ganzen Arrays, ohne Schleifen programmieren zu müssen.
- Werkzeuge zum Lesen und Schreiben von Daten auf Festplatte sowie für die Arbeit mit Memory-mapped Dateien.
- Lineare Algebra, Erzeugen von Zufallszahlen sowie Fourier-Transformation.
- Werkzeuge zur Anbindung von Code, der in C, C++ oder Fortran geschrieben wurde.

Aus der Sicht eines Ökosystems ist der letzte Punkt auch einer der wichtigsten. Weil NumPy eine einfach zu verwendende C-API bereitstellt, ist es sehr leicht, Daten an externe, in niederen Sprachen geschriebene Bibliotheken zu übergeben und ebenso die Daten von solchen Bibliotheken wieder als NumPy-Arrays zurückzugeben. Diese Fähigkeit hat Python zur Sprache der Wahl gemacht, um Wrapper für vorhandenen Code in C/C++/Fortran zu erzeugen und ihnen ein dynamisches, leicht zu verwendendes Interface zu geben.

Wenn auch NumPy von sich aus nicht viele hoch entwickelte Funktionalitäten zur Datenanalyse besitzt, so hilft ein gutes Grundverständnis von NumPy-Arrays und Array-orientiertem Rechnen doch erheblich beim Umgang mit Tools wie pandas. Wenn Sie neu in Python sind und erst einmal nur ein Gefühl für die Handhabung von Daten durch pandas bekommen möchten, können Sie dieses Kapitel auch erst mal querlesen. Für die besonderen Eigenschaften von NumPy wie Broadcasting gibt es Kapitel 12.

Für die meisten Applikationen zur Datenanalyse sind folgende die Hauptbereiche, mit denen wir es zu tun haben:

- Schnelle, vektorisierte Array-Operationen zum Bereinigen von Daten, zum Bilden und Filtern von Teilmengen, zum Umwandeln und für jede Menge anderer Berechnungen.
- Verbreitete Array-Algorithmen wie Sortieren, Eindeutigkeit und Mengenoperationen.
- Effiziente deskriptive Statistik und Aggregation/Summierung von Daten.
- Datenausrichtung und Manipulation relationaler Daten zum Mischen und Verbinden heterogener Datenmengen.
- Ausdrücken logischer Bedingungen als Array-Ausdruck anstelle von Schleifen mit `if-elif-else`-Verzweigungen.
- Gruppenweise Datenmanipulation (Aggregation, Transformation, Funktionsanwendung). Viel mehr davon in Kapitel 5.

Während NumPy die rechnerische Grundlage für diese Vorgänge bereitstellt, werden Sie wohl doch eher pandas für die meisten Arten der Datenanalyse einsetzen wollen (insbesondere für strukturierte oder tabellarische Daten), da es ein reiches und hoch angesiedeltes Interface zur Verfügung stellt, das die typischen Anwendungen sehr kompakt und einfach macht. pandas bietet obendrein etwas speziellere Funktionalität wie die Manipulation von Zeitreihen, womit NumPy gar nicht aufwarten kann.

In diesem Kapitel wie auch im Rest des Buchs setze ich als Konvention stets die Anweisung `import numpy as np` voraus. Natürlich kann ich Sie nicht daran hindern, `from numpy import *` in Ihren Code zu schreiben, um dieses ewige `np.` zu vermeiden, aber ich rate dringend davon ab, dies als schlechte Angewohnheit zu kultivieren.

Das ndarray von NumPy: ein mehrdimensionales Array-Objekt

Eine der zentralen Eigenschaften von NumPy ist sein N-dimensionales Array-Objekt oder auch ndarray, das einen schnellen, flexiblen Container für große Datensätze in Python bietet. Arrays ermöglichen Ihnen, mathematische Operationen auf ganzen Blöcken von Daten durchzuführen, wobei die Syntax den äquivalenten skalaren Operationen immer noch ähnlich bleibt:

```
In [8]: data
Out[8]:
array([[ 0.9526, -0.246 , -0.8856],
       [ 0.5639,  0.2379,  0.9104]])

In [9]: data * 10                          In [10]: data + data
Out[9]:                                    Out[10]:
```

```
array([[ 9.5256, -2.4601, -8.8565],       array([[ 1.9051, -0.492 , -1.7713],
       [ 5.6385,  2.3794,  9.104 ]])             [ 1.1277,  0.4759,  1.8208]])
```

Ein ndarray ist ein generischer mehrdimensionaler Behälter für homogene Daten; das bedeutet, alle Elemente müssen den gleichen Datentyp aufweisen. Jedes Array hat einen `shape`, ein Tupel, das die Größe in jeder Dimension angibt, sowie einen `dtype`, ein Objekt, das den Datentyp des Arrays anzeigt:

```
In [11]: data.shape
Out[11]: (2, 3)

In [12]: data.dtype
Out[12]: dtype('float64')
```

Dieses Kapitel wird Sie in die Grundzüge von Arrays in NumPy einführen und sollte ausreichen, mit dem Rest des Buchs zurechtzukommen. Für viele Anwendungen der Datenanalyse brauchen Sie kein tiefes Verständnis von NumPy. Wenn Sie aber ein wissenschaftlicher Python-Guru werden möchten, ist ein profundes Wissen über Array-orientiertes Programmieren und entsprechendes Denken eine notwendige Voraussetzung.

Wann immer Sie auf ein »Array«, »NumPy-Array« oder »ndarray« im Text stoßen, bezieht sich dies bis auf wenige Ausnahmen immer auf das Gleiche: das ndarray-Objekt.

Erzeugen von ndarrays

Der einfachste Weg, ein Array zu erzeugen, geht über die Funktion `array`. Sie akzeptiert jedes Sequenzobjekt (einschließlich anderer Arrays) und produziert ein neues NumPy-Array mit den übergebenen Daten. Ein guter Kandidat zur Umwandlung ist beispielsweise eine Liste:

```
In [13]: data1 = [6, 7.5, 8, 0, 1]

In [14]: arr1 = np.array(data1)

In [15]: arr1
Out[15]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

Verschachtelte Sequenzen wie etwa eine Liste von Listen gleicher Länge werden in ein mehrdimensionales Array umgeformt:

```
In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [17]: arr2 = np.array(data2)

In [18]: arr2
Out[18]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

In [19]: arr2.ndim
Out[19]: 2
```

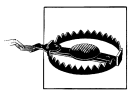
```
In [20]: arr2.shape
Out[20]: (2, 4)
```

Wenn nicht explizit angegeben (später dazu mehr), versucht `np.array`, einen guten Datentyp für das zu kreierende Array zu finden. Dieser Datentyp wird in einem speziellen `dtype`-Objekt abgelegt; in den obigen zwei Beispielen haben wir dann:

```
In [21]: arr1.dtype
Out[21]: dtype('float64')

In [22]: arr2.dtype
Out[22]: dtype('int64')
```

Zusätzlich zu `np.array` gibt es noch einige andere Funktionen, die Arrays produzieren. `zeros` und `ones` etwa erzeugen Arrays aus Nullen und Einsen mit angegebener Länge oder Gestalt. `empty` erzeugt ein Array, ohne den Inhalt mit Werten zu besetzen. Um mit diesen Methoden höherdimensionale Arrays zu erzeugen, geben Sie ein Tupel für die Gestalt an (der erste Parameter heißt `shape`):

```
In [23]: np.zeros(10)
Out[23]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [24]: np.zeros((3, 6))
Out[24]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [25]: np.empty((2, 3, 2))
Out[25]:
array([[[  4.94065646e-324,   4.94065646e-324],
        [  3.87491056e-297,   2.46845796e-130],
        [  4.94065646e-324,   4.94065646e-324]],

       [[  1.90723115e+083,   5.73293533e-053],
        [ -2.33568637e+124,  -6.70608105e-012],
        [  4.42786966e+160,   1.27100354e+025]]])
```



Es ist unsicher, anzunehmen, dass `np.empty` ein Array mit lauter Nullen abliefert. Wie oben gesehen, kann auch beliebiger Müll darin stehen.

`arange` ist eine Version der eingebauten Python2-Funktion `range` für Arrays:

```
In [26]: np.arange(15)
Out[26]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Im vorliegenden Python 3 ist `range` inzwischen ein Generator geworden. Sie erhalten das zu `arange` passende Verhalten, wenn Sie `range` durch `list(range)` ersetzen. Tabelle 4-1 zeigt eine kurze Übersicht der Funktionen zum Generieren von Arrays. Da NumPy auf numerisches Rechnen fokussiert ist, ist der Datentyp oft `float64` (Fließkommazahl, wenn nicht explizit angegeben).

Tabelle 4-1: Array-erzeugende Funktionen

Funktion	Beschreibung
`array`	Konvertiert die angegebenen Daten (Liste, Tupel, Array oder andere Sequenz) in ein ndarray. Der Datentyp wird entweder abgeleitet oder explizit angegeben. Die Daten werden standardmäßig immer kopiert, auch wenn es nicht nötig wäre.
`asarray`	Konvertiert die Eingabe in ein ndarray, aber nur, wenn sie nicht bereits ein ndarray ist.
`arange`	Wie das eingebaute `list(range)`, ergibt aber ein ndarray statt einer Liste.
`ones, ones_like`	Produziert ein Array aus lauter Einsen mit angegebener Gestalt und angegebenem Datentyp. `ones_like` übernimmt ein anderes Array und setzt bei gleicher Gestalt und gleichem Typ überall Einsen ein.
`zeros, zeros_like`	Wie `ones` und `ones_like`, aber mit Nullen statt Einsen.
`empty, empty_like`	Erzeugt neues Array durch Allozieren neuen Speichers, aber trägt keine Werte ein wie `ones` und `zeros`.
`eye, identity`	Erzeugt eine quadratische N-x-N-Einheitsmatrix (Einsen auf der Hauptdiagonalen und sonst Nullen).

Datentypen für ndarrays

Der *Datentyp* oder `dtype` ist ein spezielles Objekt mit der Information, die das ndarray braucht, um ein Stück Speicher als einen bestimmten Datentyp zu interpretieren:

```
In [27]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [28]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [29]: arr1.dtype           In [30]: arr2.dtype
Out[29]: dtype('float64')     Out[30]: dtype('int32')
```

Die `dtype`-Objekte sind ein Teil dessen, was NumPy so mächtig und flexibel macht. In den meisten Fällen entsprechen sie direkt einer zugrunde liegenden Maschinendarstellung, was es leicht macht, binäre Datenströme von und zur Platte zu übertragen sowie Code anzubinden, der in einer primitiven Sprache wie C oder Fortran geschrieben wurde. Die numerischen `dtypes` sind ähnlich benannt: Ein `dtype`-Name wie `float` oder `int` wird von einer Zahl gefolgt, die die Anzahl der Bits pro Element angibt. Ein normaler Fließkommawert in doppelter Genauigkeit (was Python übrigens auch als `float`-Objekt benutzt) verbraucht genau 8 Bytes oder 64 Bits. Mithin ist dieser Typ in NumPy als `float64` bekannt (siehe Tabelle 4-2 für die Komplettliste der von NumPy unterstützten Datentypen).

Versuchen Sie jetzt bitte nicht, sich alle `dtypes` von NumPy zu merken, insbesondere wenn Sie ein Neueinsteiger sind. Es reicht oft völlig aus, sich mit der generellen Art der Daten auseinanderzusetzen, mit der Sie umgehen, sei es Fließkommazahl, Komplex, Integer, Boolesch, String oder ein allgemeines Python-Objekt. Wenn Sie wirklich mehr Kontrolle über die Ablage im Speicher und auf Platte benötigen, insbesondere bei großen Datenmengen, haben Sie auch die Möglichkeiten dazu.

Tabelle 4-2: Datentypen von NumPy

Typ	Typencode	Beschreibung
`int8`, `uint8`	`i1`, `u1`	8-Bit-(1-Byte-)Integer-Typen mit und ohne Vorzeichen.
`int16`, `uint16`	`i2`, `u2`	16-Bit-Integer-Typen mit und ohne Vorzeichen.
`int32`, `uint32`	`i4`, `u4`	32-Bit-Integer-Typen mit und ohne Vorzeichen.
`int64`, `uint64`	`i8`, `u8`	64-Bit-Integer-Typen mit und ohne Vorzeichen.
`float16`	`f2`	Fließkommazahlen mit halber Genauigkeit.
`float32`	`f4` oder `f`	Fließkommazahlen mit einfacher Genauigkeit, kompatibel mit einem C float.
`float64`	`f8` oder `d`	Fließkommazahlen mit einfacher Genauigkeit, kompatibel mit einem C double und dem `float`-Objekt in Python.
`float128`	`f16` oder `g`	Fließkommazahlen mit erweiterter Genauigkeit.
`complex64`, `complex128`, `complex256`	`c8`, `c16`, `c32`	Komplexe Zahlen, durch je zwei Fließkommazahlen mit 32, 64 oder 128 Bit dargestellt.
`bool`	`?`	Boolescher Typ, speichert die Werte `True` und `False`.
`object`	`O`	Typ Python-Objekt.
`string_`	`S`	String-Typ fester Länge, 1 Byte pro Zeichen. Ein String `dtype` mit 10 Zeichen wäre zum Beispiel `'S10'`.
`unicode_`	`U`	Unicode-Typ fester Länge, Anzahl an Bytes ist plattformabhängig. Gleiche semantische Spezifikation wie `string_` (z. B. `'U10'`).

Mithilfe der Methode `astype` von ndarry können Sie ein Array explizit von einem `dtype` zu einem anderen umwandeln, auch *Casting* genannt:

```
In [31]: arr = np.array([1, 2, 3, 4, 5])

In [32]: arr.dtype
Out[32]: dtype('int64')

In [33]: float_arr = arr.astype(np.float64)

In [34]: float_arr.dtype
Out[34]: dtype('float64')
```

In diesem Fall wurden Integer zu Fließkommazahlen gecastet. Wenn ich eine Fließkommazahl zu einem Integer caste, wird der Nachkommateil abgeschnitten:

```
In [35]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [36]: arr
Out[36]: array([  3.7,  -1.2,  -2.6,   0.5,  12.9,  10.1])

In [37]: arr.astype(np.int32)
Out[37]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

Sollten Sie ein Array aus Strings haben, die Zahlen darstellen, können Sie astype benutzen, um diese in eine numerische Form umzuwandeln:

```
In [38]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [39]: numeric_strings.astype(float)
Out[39]: array([  1.25,  -9.6 ,  42.  ])
```

Wenn das Casting aus irgendeinem Grund fehlschlägt (wie z. B. ein String, der sich nicht in float64 umwandeln lässt), wird ein TypeError ausgelöst. Hier war ich übrigens etwas faul und schrieb einfach float anstelle von np.float64; NumPy ist clever genug, um zu erkennen, dass der Python-Typ hier den dtype anzeigen soll.

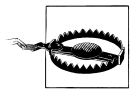
Sie können ebenfalls das dtype-Attribut eines anderen Arrays benutzen:

```
In [40]: int_array = np.arange(10)

In [41]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [42]: int_array.astype(calibers.dtype)
Out[42]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

Es gibt ein paar Abkürzungscodes, mit denen Sie sich auch auf bestimmte dtypes beziehen können:

```
In [43]: empty_uint32 = np.empty(8, dtype='u4')

In [44]: empty_uint32
Out[44]:
array([       0,        0, 65904672,        0, 64856792,        0,
       39438163,        0], dtype=uint32)
```

Der Aufruf von astype kreiert immer ein neues Array (eine Kopie der Daten), auch wenn der neue dtype identisch mit dem alten ist.



Man sollte im Kopf behalten, dass Fließkommazahlen, wie die in Arrays des Typs float64 oder float32, nur eine Annäherung von gebrochenen Zahlen speichern können. Bei komplexen Berechnungen kann dadurch einiges an Fließkommafehlern auflaufen, sodass Vergleiche nur bis zu einer begrenzten Anzahl von Dezimalstellen sinnvoll sind.

Operationen zwischen Arrays und Skalaren

Arrays sind wichtig, weil sie erlauben, viele Operationen auf Daten durchzuführen, ohne eine for-Schleife zu schreiben. Das nennt man oft auch *Vektorisierung*. Jede arithmetische Operation zwischen gleich großen Arrays führt ihre Arbeit elementweise durch:

```
In [45]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [46]: arr
Out[46]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [47]: arr * arr
Out[47]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])
```

```
In [48]: arr - arr
Out[48]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Arithmetische Operationen mit Skalaren propagieren den Skalar zu jedem Element des Arrays, wie Sie es sich vermutlich gewünscht haben:

```
In [49]: 1 / arr
Out[49]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])
```

```
In [50]: arr ** 0.5
Out[50]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

Operationen zwischen Arrays verschiedener Größe nennt man *Broadcasting*. Das wird eingehender in Kapitel 12 besprochen. Ein tiefer gehendes Verständnis des Broadcastings ist für einen Großteil dieses Buchs nicht so wichtig.

Einfaches Indizieren und Slicing

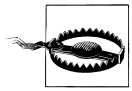
Das Indizieren von Arrays in NumPy ist ein weites Feld, weil es ziemlich viele Wege gibt, wie Sie vielleicht auf Teilmengen oder individuelle Elemente zugreifen möchten. Eindimensionale Arrays sind recht einfach; oberflächlich betrachtet, benehmen sie sich ähnlich wie Python-Listen:

```
In [51]: arr = np.arange(10)

In [52]: arr
Out[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [53]: arr[5]
Out[53]: 5

In [54]: arr[5:8]
Out[54]: array([5, 6, 7])

In [55]: arr[5:8] = 12

In [56]: arr
Out[56]: 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 bei `arr[5:8] = 12`, zuweisen, an alle Positionen propagiert (man sagt dazu auch *Broadcasting*). Ein wichtiger erster Unterschied gegenüber Listen ist, dass Teilbereiche (Slices) von Arrays sogenannte *Views* auf das ursprüngliche Array sind. Die Daten werden dabei nicht kopiert, und jede Änderung des Views schlägt sich auch im Array nieder:

```
In [57]: arr_slice = arr[5:8]

In [58]: arr_slice[1] = 12345

In [59]: arr
Out[59]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,     9])

In [60]: arr_slice[:] = 64

In [61]: arr
Out[61]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

Wenn Sie neu bei NumPy sind, mag Sie das überraschen – vor allem wenn Sie andere Sprachen für Arrays kennen, die eifriger im Kopieren sind. Weil NumPy für große Datenmengen konzipiert wurde, können Sie sich vorstellen, was für Rechenzeit- und Speicherprobleme entstünden, wenn NumPy darauf beharrte, alle naselang etwas kopieren zu müssen.



Wenn Sie auf einer echten Kopie eines Slice bestehen, müssen Sie explizit das Array kopieren, zum Beispiel `arr[5:8].copy()`.

Bei höherdimensionalen Arrays haben Sie gleich eine Menge mehr Optionen. Im zweidimensionalen Fall sind die Elemente an jedem Index nicht mehr Skalare, sondern eindimensionale Arrays:

```
In [62]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [63]: arr2d[2]
Out[63]: array([7, 8, 9])
```

Mithin kann man einzelne Elemente rekursiv ansprechen. Aber das ist etwas viel Aufwand, besser ist es, Sie übergeben eine kommaseparierte Liste von Indizes zum Selektieren einzelner Elemente. Folgendes ist also äquivalent:

```
In [64]: arr2d[0][2]
Out[64]: 3

In [65]: arr2d[0, 2]
Out[65]: 3
```

Beachten Sie auch Abbildung 4-1 für eine Illustration der Indizierung bei einem 2D-Array.

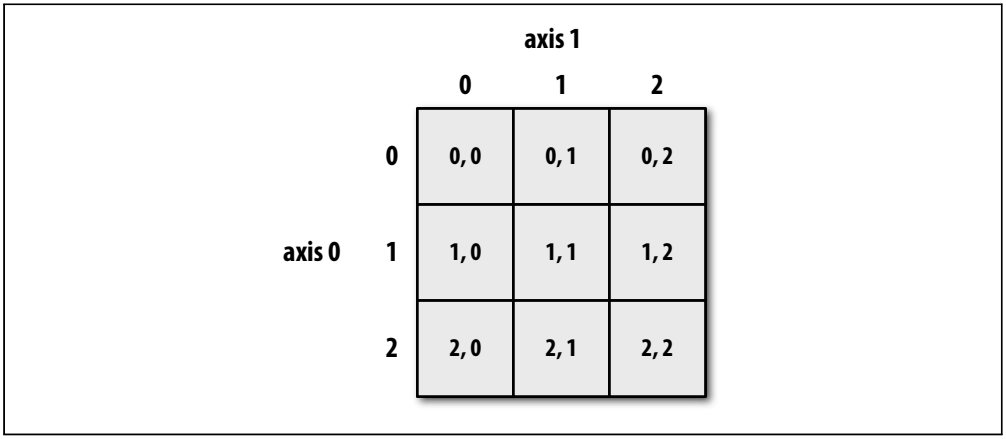


Abbildung 4-1: Elemente in einem NumPy-Array indizieren

Wenn Sie bei einem mehrdimensionalen Array spätere Indizes weglassen, entsteht ein niedrigerdimensionales ndarray mit all den Daten entlang den höheren Dimensionen. Sehen wir uns das 2 × 2 × 3-Array `arr3d` an:

```
In [66]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [67]: arr3d
Out[67]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

`arr3d[0]` ist ein 2 × 3-Array:

```
In [68]: arr3d[0]
Out[68]:
array([[1, 2, 3],
       [4, 5, 6]])
```

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

```
In [69]: old_values = arr3d[0].copy()

In [70]: arr3d[0] = 42

In [71]: arr3d
Out[71]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [72]: arr3d[0] = old_values

In [73]: arr3d
Out[73]:
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Analog gibt Ihnen `arr3d[1, 0]` alle Werte, deren Indizes mit `(1, 0)` anfangen, also ein eindimensionales Array:

```
In [74]: arr3d[1, 0]
Out[74]: array([7, 8, 9])
```

Beachten Sie, dass in allen Fällen, in denen Unterbereiche von Arrays selektiert wurden, die Ergebnisse Views waren.

Indizieren mit Slices

Wie eindimensionale Objekte (etwa Listen in Python) können auch ndarrays mit der gewohnten Syntax ausgeschnitten werden:

```
In [75]: arr[1:6]
Out[75]: array([ 1,  2,  3,  4, 64])
```

Höherdimensionale Objekte ergeben mehr Möglichkeiten zum Slicing, da man entlang einer oder mehrerer Achsen schneiden und mit Integern mixen kann. Betrachten wir obiges zweidimensionales Array, `arr2d`. Das Slicing ist bei diesem Array etwas anders:

```
In [76]: arr2d
Out[76]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [77]: arr2d[:2]
Out[77]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Wie Sie sehen, wurde entlang von Achse 0 geschnitten, der ersten Achse. Ein Slice selektiert somit einen Bereich von Elementen entlang einer Achse. Sie können wie auch bei Indizes mehrere Slices angeben:

```
In [78]: arr2d[:2, 1:]
Out[78]:
array([[2, 3],
       [5, 6]])
```

Wenn Sie in dieser Weise schneiden, bekommen Sie immer Array-Views mit der gleichen Anzahl von Dimensionen. Beim Mischen von Integer-Indizes und Slices bekommen Sie Schnitte mit niedrigerer Dimensionalität:

```
In [79]: arr2d[1, :2]
Out[79]: array([4, 5])
```

```
In [80]: arr2d[2, :1]
Out[80]: array([7])
```

Dies ist verdeutlicht in Abbildung 4-2. Beachten Sie, dass ein Doppelpunkt allein ohne Grenze die gesamte Achse übernimmt. Somit können Sie nur einen Slice der höherdimensionalen Achse erzeugen mit:

```
In [81]: arr2d[:, :1]
Out[81]:
array([[1],
```

```
       [4],
       [7]])
```

Das Zuweisen an einen Slice-Ausdruck weist in der Tat die gesamte Selektion zu:

```
In [82]: arr2d[:2, 1:] = 0
```

Boolesches Indizieren

Lassen Sie uns ein Beispiel betrachten, bei dem wir einige Daten in einem Array haben, sowie ein zweites Array mit mehrfach vorhandenen Namen. Ich werde hier die `randn`-Funktion in `numpy.random` verwenden, um ein paar normalverteilte Daten zu generieren:

```
In [83]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [84]: data = np.random.randn(7, 4)

In [85]: names
Out[85]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='|S4')

In [86]: data
Out[86]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [-0.268 ,  0.5465,  0.0939, -2.0445],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [ 2.1452,  0.8799, -0.0523,  0.0672],
       [-1.0023, -0.1698,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

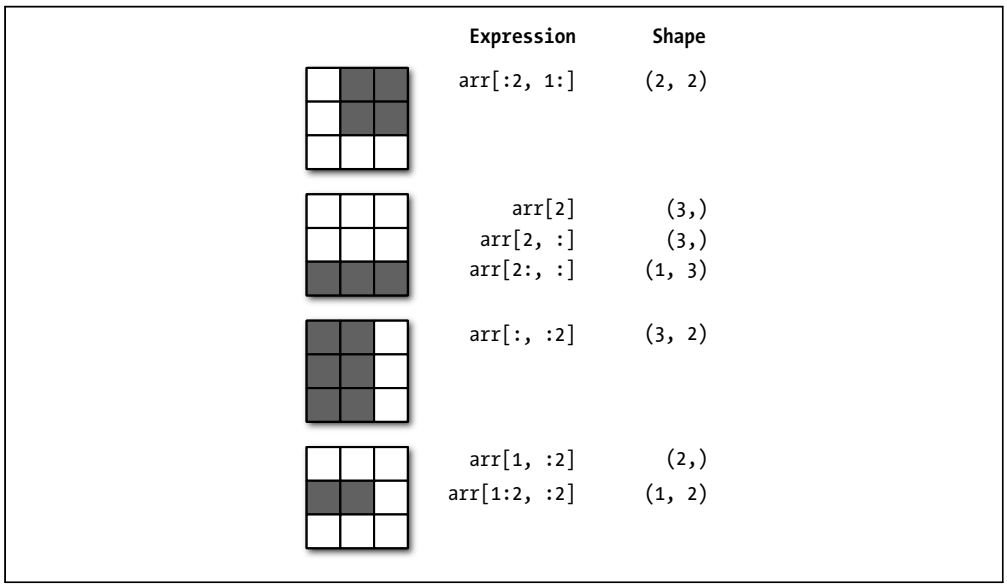


Abbildung 4-2: Zweidimensionales Slicing von Arrays

Angenommen, jeder Name gehörte zu einer Zeile im Array `data` und wir möchten diejenigen Zeilen mit dem korrespondierenden Namen `'Bob'` auswählen. Wie die arithmetischen Operationen sind auch Vergleiche (wie `==`) mit Arrays vektorisiert. Das Vergleichen der Namen mit dem String `'Bob'` produziert ein boolesches Array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

Dieses boolesche Array kann zum Indizieren verwendet werden:

```
In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

Das boolesche Array muss die gleiche Länge aufweisen wie die Achse, die indiziert werden soll. Sie können boolesche Arrays sogar mit Slices und Integern kombinieren (oder ganzen Sequenzen davon, wie wir später sehen werden):

```
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792],
       [-0.0523,  0.0672]])

In [90]: data[names == 'Bob', 3]
Out[90]: array([ 1.2792,  0.0672])
```

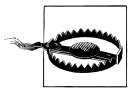
Um alles zu selektieren außer `'Bob'`, können Sie entweder `!=` benutzen oder die Bedingung durch `-` negieren:

```
In [91]: names != 'Bob'
Out[91]: array([False, True, True, False, True, True, True], dtype=bool)

In [92]: data[-(names == 'Bob')]
Out[92]:
array([[-0.268 ,  0.5465,  0.0939, -2.0445],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [-1.0023, -0.1698,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

Zum Selektieren zweier der drei Namen kombinieren wir boolesche Operatoren wie `&` (und) und `|` (oder):

```
In [93]: mask = (names == 'Bob') | (names == 'Will')

In [94]: mask
Out[94]: array([True, False, True, True, True, False, False], dtype=bool)

In [95]: data[mask]
Out[95]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [ 2.1452,  0.8799, -0.0523,  0.0672],
       [-1.0023, -0.1698,  1.1503,  1.7289]])
```

Das Auswählen von Daten mithilfe von booleschem Indizieren erstellt immer eine Kopie der Daten, auch wenn das Array unverändert bleibt.



Die Python-Schlüsselwörter `and` und `or` funktionieren nicht mit booleschen Arrays.

Das Setzen von Werten mit booleschen Arrays funktioniert nach dem gesunden Menschenverstand. Alle negativen Werte in `data` auf null zu setzen, ist unglaublich einfach:

```
In [96]: data[data < 0] = 0

In [97]: data
Out[97]:
array([[ 0.    ,  0.5433,  0.    ,  1.2792],
       [ 0.    ,  0.5465,  0.0939,  0.    ],
       [ 0.    ,  0.    ,  0.7719,  0.3103],
       [ 2.1452,  0.8799,  0.    ,  0.0672],
       [ 0.    ,  0.    ,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.    ,  0.    ]])
```

Auch ganze Zeilen oder Spalten durch ein boolesches Array zu setzen, ist sehr einfach:

```
In [98]: data[names != 'Joe'] = 7

In [99]: data
Out[99]:
array([[ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.    ,  0.5465,  0.0939,  0.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.    ,  0.    ]])
```

Fancy Indexing

Fancy Indexing (übersetzbar mit »raffiniertes Indizieren«) ist ein durch NumPy adoptierter Begriff für das Indizieren durch Integer-Arrays. Angenommen, wir hätten ein 8 × 4-Array:

```
In [100]: arr = np.empty((8, 4))

In [101]: for i in range(8):
   .....:     arr[i] = i

In [102]: arr
Out[102]:
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.]])
```

Zum Auswählen einiger Zeilen in einer bestimmten Anordnung übergeben Sie einfach eine Liste oder ein ndarray mit ganzen Zahlen, die die gewünschte Ordnung angeben:

```
In [103]: arr[[4, 3, 0, 6]]
Out[103]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

Das war hoffentlich das, was Sie erwartet haben! Mit negativen Indizes selektieren Sie vom Ende:

```
In [104]: arr[[-3, -5, -7]]
Out[104]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

Das Angeben von mehreren Index-Arrays tut etwas geringfügig anderes, es selektiert ein eindimensionales Array, das mit jedem Tupel von Indizes korrespondiert:

```
# more on reshape in Chapter 12
In [105]: arr = np.arange(32).reshape((8, 4))

In [106]: arr
Out[106]:
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 [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[107]: array([ 4, 23, 29, 10])
```

Nehmen Sie sich etwas Zeit, um das genau zu verinnerlichen: Es wurden die Elemente `(1, 0)`, `(5, 3)`, `(7, 1)` und `(2, 2)` selektiert. Das Verhalten von Fancy Indexing ist in diesem Fall etwas anders, als man es erwarten könnte (mich selbst eingeschlossen), nämlich ein rechteckiger Bereich gebildet aus einer Teilmenge an Zeilen und Spalten. Um diesen zu bekommen, können Sie so vorgehen:

```
In [108]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[108]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Ein weiterer Ansatz ist es, die Funktion `np.ix_` zu verwenden, die zwei eindimensionale Integer-Arrays zu einem Indexer für die quadratische Region umwandelt:

```
In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[109]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Behalten Sie im Hinterkopf: Fancy Indexing kopiert die Daten im Gegensatz zum Slicing immer in ein neues Array.

Arrays transponieren und Achsen vertauschen

Transponieren ist eine spezielle Art des Umformens, die einen View ergibt, ohne irgendetwas zu kopieren. Arrays besitzen die Methode `transpose` und auch das spezielle `T`-Attribut:

```
In [110]: arr = np.arange(15).reshape((3, 5))

In [111]: arr
Out[111]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
In [112]: arr.T
Out[112]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

Bei Matrixberechnungen werden Sie dies sehr oft tun, zum Beispiel zum Berechnen des inneren Matrixprodukts X^TX mit `np.dot`:

```
In [113]: arr = np.random.randn(6, 3)

In [114]: np.dot(arr.T, arr)
Out[114]:
array([[ 2.584 ,  1.8753,  0.8888],
       [ 1.8753,  6.6636,  0.3884],
       [ 0.8888,  0.3884,  3.9781]])
```

Bei höherdimensionalen Arrays eignet sich die Methode `transpose` besser, weil sie ein Tupel von zu permutierenden Achsennummern akzeptiert (möge Ihr Gehirn explodieren):

```
In [115]: arr = np.arange(16).reshape((2, 2, 4))

In [116]: arr
Out[116]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [117]: arr.transpose((1, 0, 2))
Out[117]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
```

```
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Das einfache Transponieren mit `.T` ist nur der Spezialfall des Vertauschens zweier Achsen. ndarray hat ferner die Methode `swapaxes`, die ein Paar von Achsennummern akzeptiert:

```
In [118]: arr                    In [119]: arr.swapaxes(1, 2)
Out[118]:                        Out[119]:
array([[[ 0,  1,  2,  3],        array([[[ 0,  4],
        [ 4,  5,  6,  7]],               [ 1,  5],
                                         [ 2,  6],
       [[ 8,  9, 10, 11],                [ 3,  7]],
        [12, 13, 14, 15]]])
                                        [[ 8, 12],
                                         [ 9, 13],
                                         [10, 14],
                                         [11, 15]]])
```

Auf ähnliche Weise ergibt `swapaxes` einen View auf die Daten, ohne irgendetwas zu kopieren.

Universelle Funktionen: Schnelle elementweise Array-Funktionen

Eine universelle Funktion oder auch *ufunc* ist eine Funktion, die elementweise Operationen auf Daten in ndarrays durchführt. Sie können sich diese vorstellen als schnelle vektorisierte Wrapper für einfache Funktionen, die aus einem oder mehreren Skalaren ein oder mehrere skalare Ergebnisse produzieren.

Viele ufuncs sind simple elementweise Transformationen wie etwa `sqrt` oder `exp`:

```
In [120]: arr = np.arange(10)

In [121]: np.sqrt(arr)
Out[121]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [122]: np.exp(arr)
Out[122]:
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 `add` oder `maximum` nehmen zwei Arrays (deshalb *binäre* ufuncs) und liefern ein einziges Array als Ergebnis:

```
In [123]: x = np.random.randn(8)

In [124]: y = np.random.randn(8)

In [125]: x
Out[125]:
array([ 0.0749,  0.0974,  0.2002, -0.2551,  0.4655,  0.9222,  0.446 ,
```

```
       -0.9337])

In [126]: y
Out[126]:
array([ 0.267 , -1.1131, -0.3361,  0.6117, -1.2323,  0.4788,  0.4315,
       -0.7147])

In [127]: np.maximum(x, y) # element-wise maximum
Out[127]:
array([ 0.267 ,  0.0974,  0.2002,  0.6117,  0.4655,  0.9222,  0.446 ,
       -0.7147])
```

Wenn auch weniger verbreitet, so kann eine ufunc auch mehrere Arrays abliefern. `modf` ist ein Beispiel dafür, eine vektorisierte Form der eingebauten Python-Funktion `divmod`: Sie liefert den gebrochenen und den ganzzahligen Teil eines Fließkomma-Arrays:

```
In [128]: arr = np.random.randn(7) * 5

In [129]: np.modf(arr)
Out[129]:
(array([-0.6808,  0.0636, -0.386 ,  0.1393, -0.8806,  0.9363, -0.883 ]),
 array([-2.,  4., -3.,  5., -3.,  3., -6.]))
```

Die verfügbaren ufuncs sind in Tabelle 4-3 und Tabelle 4-4 aufgeführt.

Tabelle 4-3: Unäre ufuncs

Funktion	Beschreibung
`abs, fabs`	Berechnet elementweise den Absolutwert für ganzzahlige, Fließkomma- oder komplexe Werte. Benutzen Sie `fabs` als eine schnellere Variante für nicht komplexwertige Daten.
`sqrt`	Berechnet die Quadratwurzel jedes Elements. Entspricht `arr ** 0.5`.
`square`	Berechnet das Quadrat jedes Elements. Entspricht `arr ** 2`.
`exp`	Berechnet den Exponenten e^x jedes Elements.
`log, log10, log2, log1p`	Natürlicher Logarithmus (Basis *e*), log Basis 10, log Basis 2 sowie log(1 + x).
`sign`	Berechnet das Signum jedes Elements: 1 (positiv), 0 (zero) oder -1 (negativ).
`ceil`	Rundet jedes Elements auf, also die kleinste ganze Zahl, die größer oder gleich dem Element ist.
`floor`	Rundet jedes Elements ab, also die größte ganze Zahl, die kleiner oder gleich dem Element ist.
`rint`	Rundet Elemente zur nächsten ganzen Zahl mit dem gleichen `dtype`.
`modf`	Gibt den gebrochenen und den ganzzahligen Anteil des Arrays als separate Arrays zurück.
`isnan`	Liefert ein boolesches Array, das die Werte anzeigt, die `NaN` (Not a Number) sind.
`isfinite, isinf`	Liefert ein boolesches Array, das die Werte anzeigt, die endlich (nicht-`inf`, nicht-`NaN`) bzw. unendlich sind.
`cos, cosh, sin, sinh, tan, tanh`	Reguläre und hyperbolische trigonometrische Funktionen.
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometrische Funktionen.
`logical_not`	Berechnet elementweise den Wahrheitswert von `not x`.

Tabelle 4-4: Binäre universelle Funktionen

Funktion	Beschreibung
`add`	Addiert korrespondierende Elemente in Arrays.
`subtract`	Subtrahiert Elemente des zweiten Arrays vom ersten Array.
`multiply`	Multipliziert Array-Elemente.
`divide, floor_divide`	Dividieren oder floor-Dividieren (Abschneiden des Rests).
`power`	Nimmt die Elemente im ersten Array zum Exponenten im zweiten Array hoch.
`maximum, fmax`	Elementweises Maximum.`fmax` ignoriert `NaN`.
`minimum, fmin`	Elementweises Minimum.`fmin` ignoriert `NaN`.
`mod`	Elementweiser Modulus (Rest bei Division).
`copysign`	Kopiert die Vorzeichen im zweiten Argument in die Werte des ersten Arguments.
`greater, greater_equal, less, less_equal, equal, not_equal`	Führt elementweise Vergleiche durch, ergibt ein boolesches Array. Äquivalent zu diesen Infix-Operatoren: `>, >=, <, <=, ==, !=`
`logical_and, logical_or, logical_xor`	Berechnet elementweise Wahrheitswerte der logischen Operationen. Äquivalent zu diesen Infix-Operatoren: `& | ^`

Datenverarbeitung mit Arrays

Mit NumPy-Arrays können Sie diverse Aufgaben der Datenbearbeitung als prägnante Array-Ausdrücke schreiben, die sonst viele Schleifen erfordern würden. Die Praxis, explizite Schleifen durch Ausdrücke mit Arrays zu ersetzen, nennt man auch *Vektorisierung*. Im Allgemeinen werden vektorisierte Array-Operationen ein bis zwei Größenordnungen (oder noch mehr) schneller sein als ihr reines Python-Äquivalent, dem weit größten Effekt bei jeder Art von numerischer Berechnung. Später in Kapitel 12 werde ich das *Broadcasting* erklären, ein mächtiges Hilfsmittel beim Vektorisieren von Berechnungen.

Als ein einfaches Beispiel evaluieren wir die Funktion `sqrt(x^2 + y^2)` über ein reguläres Gitter von Werten. Die Funktion `np.meshgrid` nimmt zwei eindimensionale Arrays und produziert zwei zweidimensionale Matrizen, die zu allen Paaren von `(x, y)` in den zwei Arrays korrespondieren:

```
In [130]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

In [131]: xs, ys = np.meshgrid(points, points)

In [132]: ys
Out[132]:
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 bedarf es nun nichts weiter, als den gleichen Ausdruck zu schreiben, den Sie auch für nur zwei Punkte schreiben würden:

```
In [134]: import matplotlib.pyplot as plt

In [135]: z = np.sqrt(xs ** 2 + ys ** 2)

In [136]: z
Out[136]:
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 [137]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[137]: <matplotlib.colorbar.Colorbar instance at 0x4e46d40>

In [138]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
Out[138]: <matplotlib.text.Text at 0x4565790>
```

Sehen Sie sich Abbildung 4-3 an. Hier habe ich die Funktion `imshow` aus matplotlib benutzt, um ein Bild des zweidimensionalen Arrays aus Funktionswerten zu plotten.

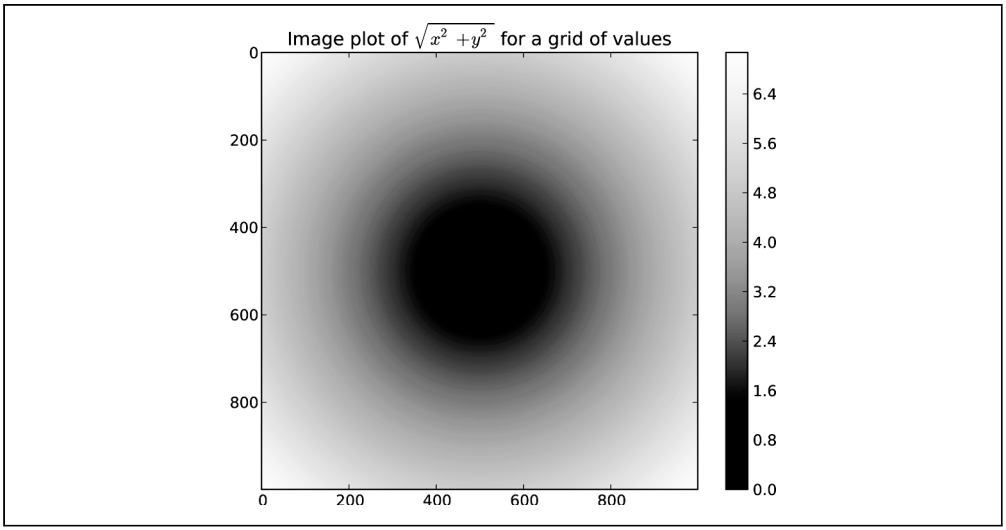


Abbildung 4-3: Plot der Funktionsauswertung am Grid

Konditionale Logik als Array-Operationen

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

```
In [140]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [141]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [142]: cond = np.array([True, False, True, True, False])
```

Wir wollen jetzt immer dann einen Wert von `xarr` nehmen, wenn der korrespondierende Wert in `cond` `True` ist, ansonsten den Wert aus `yarr`. Eine List-Comprehension, die das tut, sähe ungefähr so aus:

```
In [143]: result = [(x if c else y)
   .....:           for x, y, c in zip(xarr, yarr, cond)]

In [144]: result
Out[144]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

Leider ist dieser Ansatz problematisch. Zunächst wird dies bei großen Arrays nicht sehr schnell sein, weil die ganze Arbeit in reinem Python geschieht. Des Weiteren wird es nicht mit mehrdimensionalen Arrays funktionieren. Mit `np.where` können Sie das jedoch sehr kurz schreiben:

```
In [145]: result = np.where(cond, xarr, yarr)

In [146]: result
Out[146]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

Das zweite und das dritte Argument zu `np.where` müssen dabei keine Arrays sein, es sind Skalare erlaubt. Eine typische Anwendung von `where` in der Datenanalyse ist, ein neues Array basierend auf Werten in einem anderen Array zu erzeugen. Stellen Sie sich ein Array mit zufälligen Daten vor. Nun möchten Sie alle positiven Werte durch 2 und alle negativen durch –2 ersetzen. Das lässt sich sehr leicht mit `np.where` erreichen:

```
In [147]: arr = np.random.randn(4, 4)

In [148]: arr
Out[148]:
array([[ 0.6372,  2.2043,  1.7904,  0.0752],
       [-1.5926, -1.1536,  0.4413,  0.3483],
       [-0.1798,  0.3299,  0.7827, -0.7585],
       [ 0.5857,  0.1619,  1.3583, -1.3865]])

In [149]: np.where(arr > 0, 2, -2)
Out[149]:
array([[ 2,  2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2,  2, -2],
       [ 2,  2,  2, -2]])

In [150]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[150]:
array([[ 2.    ,  2.    ,  2.    ,  2.    ],
       [-1.5926, -1.1536,  2.    ,  2.    ],
       [-0.1798,  2.    ,  2.    , -0.7585],
       [ 2.    ,  2.    ,  2.    , -1.3865]])
```

Die an where übergebenen Arrays können mehr sein als nur gleich große Arrays oder Skalare.

Mit etwas Cleverness können Sie mit where auch weit komplexere Logik ausdrücken; hier habe ich zwei boolesche Arrays, cond1 und cond2, und möchte für jedes der vier möglichen Paare von booleschen Werten einen unterschiedlichen Wert zuweisen:

```
result = []
for i in range(n):
    if cond1[i] and cond2[i]:
        result.append(0)
    elif cond1[i]:
        result.append(1)
    elif cond2[i]:
        result.append(2)
    else:
        result.append(3)
```

Wenn auch nicht direkt offensichtlich, kann diese for-Schleife in einen geschachtelten where-Ausdruck verwandelt werden:

```
np.where(cond1 & cond2, 0,
         np.where(cond1, 1,
                  np.where(cond2, 2, 3)))
```

In diesem Beispiel können wir obendrein noch einen Vorteil daraus ziehen, dass die booleschen Werte in Berechnungen als 0 oder 1 gehandhabt werden, und das könnte alternativ als eine arithmetische Operation ausgedrückt werden (wenngelich noch ein bisschen kryptischer):

```
result = 1 * (cond1 & -cond2) + 2 * (cond2 & -cond1) + 3 * -(cond1 | cond2)
```

Mathematische und statistische Methoden

Eine Anzahl mathematischer Methoden, die Statistik über ein ganzes Array oder über die Daten entlang einer Achse berechnen, sind als Array-Methoden zugreifbar. Aggregationen (oftmals 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:

```
In [151]: arr = np.random.randn(5, 4) # normally-distributed data

In [152]: arr.mean()
Out[152]: 0.062814911084854597

In [153]: np.mean(arr)
Out[153]: 0.062814911084854597

In [154]: arr.sum()
Out[154]: 1.2562982216970919
```

Funktionen wie mean und sum haben einen optionalen axis-Parameter, der es erlaubt, die Statistik entlang einer bestimmten Achse zu rechnen. In jedem Fall ergibt sich ein Array mit einer Dimension weniger:

```
In [155]: arr.mean(axis=1)
Out[155]: array([-1.2833,  0.2844,  0.6574,  0.6743, -0.0187])

In [156]: arr.sum(0)
Out[156]: array([-3.1003, -1.6189,  1.4044,  4.5712])
```

Andere Methoden wie `cumsum` und `cumprod` aggregieren nicht, sondern produzieren ein Array mit den Zwischenergebnissen:

```
In [157]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [158]: arr.cumsum(0)
Out[158]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])
```

```
In [159]: arr.cumprod(1)
Out[159]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

Bei Interesse bietet Tabelle 4-5 eine komplette Auflistung. In späteren Kapiteln werden wir noch viele Beispiele für diese Methoden in Aktion sehen.

Tabelle 4-5: Elementare statistische Array-Methoden

Methode	Beschreibung
`sum`	Summe aller Elemente im Array oder entlang einer Achse. Arrays mit der Länge null haben die Summe 0.
`mean`	Arithmetischer Mittelwert. Arrays der Länge null haben `NaN` als Mittelwert.
`std`, `var`	Standardabweichung und Varianz mit optionaler Anpassung der Freiheitsgrade (Standardnenner `n`).
`min`, `max`	Minimum und Maximum.
`argmin`, `argmax`	Indizes der minimalen und maximalen Elemente.
`cumsum`	Kumulative Summe der Elemente, beginnend mit 0.
`cumprod`	Kumulatives Produkt der Elemente, beginnend mit 1.

Methoden für boolesche Arrays

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

```
In [160]: arr = np.random.randn(100)

In [161]: (arr > 0).sum() # Number of positive values
Out[161]: 44
```

Zwei weitere Methoden, `any` und `all`, sind besonders geeignet für boolesche Arrays. `any` testet, ob wenigstens einer der Werte in einem Array `True` ist, während bei `all` alle Werte `True` sein müssen:

```
In [162]: bools = np.array([False, False, True, False])

In [163]: bools.any()
Out[163]: True
```

```
In [164]: bools.all()
Out[164]: False
```

Diese Methoden funktionieren auch mit nicht booleschen Arrays, wobei jeder Wert ungleich 0 als `True` gezählt wird.

Sortieren

Wie Pythons eingebauter Listentyp haben auch die NumPy-Arrays eine `sort`-Methode, die die Werte an Ort und Stelle sortiert:

```
In [165]: arr = np.random.randn(8)

In [166]: arr
Out[166]:
array([ 0.6903,  0.4678,  0.0968, -0.1349,  0.9879,  0.0185, -1.3147,
       -0.5425])

In [167]: arr.sort()

In [168]: arr
Out[168]:
array([-1.3147, -0.5425, -0.1349,  0.0185,  0.0968,  0.4678,  0.6903,
        0.9879])
```

Bei mehrdimensionalen Arrays kann jeder eindimensionale Bereich an Ort und Stelle durch die Angabe der Achsennummer bei `sort` sortiert werden:

```
In [169]: arr = np.random.randn(5, 3)

In [170]: arr
Out[170]:
array([[-0.7139, -1.6331, -0.4959],
       [ 0.8236, -1.3132, -0.1935],
       [-1.6748,  3.0336, -0.863 ],
       [-0.3161,  0.5362, -2.468 ],
       [ 0.9058,  1.1184, -1.0516]])

In [171]: arr.sort(1)

In [172]: arr
Out[172]:
array([[-1.6331, -0.7139, -0.4959],
       [-1.3132, -0.1935,  0.8236],
       [-1.6748, -0.863 ,  3.0336],
       [-2.468 , -0.3161,  0.5362],
       [-1.0516,  0.9058,  1.1184]])
```

Die Methode `np.sort` produziert eine sortierte Kopie, anstatt das Array selbst zu ändern. Eine »Hauruckmethode« zum Berechnen der Quantile eines Arrays ist, es zu sortieren und dann die Werte an einem bestimmten Rang auszuwählen:

```
In [173]: large_arr = np.random.randn(1000)

In [174]: large_arr.sort()
```

```
In [175]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[175]: -1.5791023260896004
```

Weitere Details zur Sortiermethode von NumPy und zu fortgeschritteneren Techniken wie indirekte Sortierung finden Sie in Kapitel 12. Ferner gibt es in pandas jede Menge weiterer Datenmanipulationen, die mit Sortieren zu tun haben (etwa Datentabellen nach einer oder mehreren Spalten zu sortieren).

Unique und andere Mengenlogik

NumPy besitzt einige elementare Mengenoperationen auf eindimensionalen Arrays. Vermutlich die am meisten verbreitete ist `np.unique`, die die sortierten eindeutigen Werte als Array zurückgibt:

```
In [176]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [177]: np.unique(names)
Out[177]:
array(['Bob', 'Joe', 'Will'],
      dtype='|S4')

In [178]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [179]: np.unique(ints)
Out[179]: array([1, 2, 3, 4])
```

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

```
In [180]: sorted(set(names))
Out[180]: ['Bob', 'Joe', 'Will']
```

Eine andere Funktion namens `np.in1d` prüft das Vorhandensein der Werte des einen Arrays in einem anderen und liefert ein boolesches Array zurück:

```
In [181]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [182]: np.in1d(values, [2, 3, 6])
Out[182]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

In Tabelle 4-6 finden Sie eine Auflistung der Mengenfunktionen in NumPy.

Tabelle 4-6: Mengenoperationen bei Arrays

Methode	Beschreibung
`unique(x)`	Berechnet und sortiert die eindeutigen Elemente in `x`.
`intersect1d(x, y)`	Berechnet und sortiert die gemeinsamen Elemente in `x` und `y`.
`union1d(x, y)`	Berechnet und sortiert die Vereinigung der Elemente.
`in1d(x, y)`	Berechnet ein boolesches Array, das für jedes Element von`x` das Enthaltensein in `y` anzeigt.
`setdiff1d(x, y)`	Mengendifferenz, Elemente in`x` und nicht in `y`.
`setxor1d(x, y)`	Symmetrische Mengendifferenz; Elemente, die in einem der Arrays, aber nicht in beiden vorkommen.

Dateiein- und -ausgabe bei Arrays

NumPy ist in der Lage, Daten sowohl in Text- als auch Binärform auf die Platte zu laden und zu speichern. In den nachfolgenden Kapiteln werden Sie auch Werkzeuge zum Lesen tabellarischer Daten in den Speicher kennenlernen.

Arrays im binären Format auf der Platte speichern

`np.save` und `np.load` sind die zwei Arbeitspferde für effizientes Speichern und Laden von Array-Daten auf Platte. Arrays werden normalerweise in einem unkomprimierten rohen Format gespeichert mit der Endung *.npy*.

```
In [183]: arr = np.arange(10)

In [184]: np.save('some_array', arr)
```

Wenn der Dateipfad nicht bereits auf *.npy* endet, wird diese Erweiterung angehängt. Das Array auf der Platte kann dann mit `np.load` geladen werden:

```
In [185]: np.load('some_array.npy')
Out[185]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Sie können mit der Funktion `np.savez` mehrere Arrays in einem Zip-Archiv speichern, wobei Sie die Arrays als Schlüsselwortargumente angeben:

```
In [186]: np.savez('array_archive.npz', a=arr, b=arr)
```

Wenn Sie eine *.npz*-Datei laden, bekommen Sie ein Dictionary-ähnliches Objekt, das die einzelnen Arrays »lazy« (faul) lädt, also die Daten erst heranholt, wenn darauf zugegriffen wird:

```
In [187]: arch = np.load('array_archive.npz')

In [188]: arch['b']
Out[188]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Speichern und Laden von Textdateien

Das Laden von Text ist an sich keine besondere Aufgabe, aber die Fülle der Funktionen zum Lesen und Schreiben von Dateien in Python kann einen Anfänger schon ein wenig erschlagen. Deshalb möchte ich mein Augenmerk im Wesentlichen auf die Funktionen `read_csv` und `read_table` in pandas richten. Manchmal kann es von Nutzen sein, mit `np.loadtxt` oder dem spezialisierteren `np.genfromtxt` Daten direkt in NumPy-Arrays einzulesen.

Diese Funktionen besitzen jede Menge Optionen, um verschiedene Trennzeichen, Umwandlungsfunktionen für verschiedene Spalten, Überspringen von Zeilen und vieles mehr anzugeben. Nehmen Sie einen einfachen Fall einer kommaseparierten Datei (CSV) wie dieser:

```
In [191]: !cat ch04/array_ex.txt
0.580052,0.186730,1.040717,1.134411
0.194163,-0.636917,-0.938659,0.124094
-0.126410,0.268607,-0.695724,0.047428
-1.484413,0.004176,-0.744203,0.005487
2.302869,0.200131,1.670238,-1.881090
-0.193230,1.047233,0.482803,0.960334
```

Diese kann folgendermaßen in ein zweidimensionales Array geladen werden:

```
In [192]: arr = np.loadtxt('ch04/array_ex.txt', delimiter=',')

In [193]: arr
Out[193]:
array([[ 0.5801,  0.1867,  1.0407,  1.1344],
       [ 0.1942, -0.6369, -0.9387,  0.1241],
       [-0.1264,  0.2686, -0.6957,  0.0474],
       [-1.4844,  0.0042, -0.7442,  0.0055],
       [ 2.3029,  0.2001,  1.6702, -1.8811],
       [-0.1932,  1.0472,  0.4828,  0.9603]])
```

`np.savetxt` führt die umgekehrte Operation aus: ein Array in eine begrenzte Textdatei zu schreiben. `genfromtxt` ist ähnlich wie `loadtxt`, aber mehr in Richtung strukturierter Daten und Behandlung fehlender Daten hochgezüchtet. In Kapitel 12finden Sie mehr über strukturierte Arrays.

Mehr über das Lesen und Schreiben von Dateien, insbesondere tabellarischen oder arbeitsblattähnlichen Daten, finden Sie in späteren Kapiteln, die pandas und DataFrame-Objekte behandeln.

Lineare Algebra

Lineare Algebra, wie Matrizenmultiplikation, Dekomposition, Determinanten und andere Mathematik mit quadratischen Matrizen, ist ein wichtiger Teil jeder Array-Bibliothek. Anders als bei Sprachen wie MATLAB ist die Multiplikation zweier Matrizen mit * ein elementweises Produkt und kein Skalarprodukt von Matrizen. Dafür gibt es `dot`, sowohl eine Array-Methode als auch eine Funktion im Namensraum von `numpy`, für die Multiplikation von Matrizen:

```
In [194]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [195]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [196]: x                    In [197]: y
Out[196]:                      Out[197]:
array([[ 1.,  2.,  3.],       array([[  6.,  23.],
       [ 4.,  5.,  6.]])              [ -1.,   7.],
                                      [  8.,   9.]])

In [198]: x.dot(y)  # equivalently np.dot(x, y)
Out[198]:
array([[  28.,   64.],
       [  67.,  181.]])
```

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

```
In [199]: np.dot(x, np.ones(3))
Out[199]: array([  6.,  15.])
```

`numpy.linalg` besitzt einen Standardsatz an Funktionen zu Matrizenzerlegung, Invertierung, Determinanten und weiteren Dingen. Diese sind unter der Oberfläche mit den gleichen Fortran-Bibliotheken des Industriestandard implementiert wie in anderen Sprachen (etwa MATLAB und R). Vertreter sind BLAS, LAPACK oder auch (abhängig davon, wie Ihr NumPy kompiliert wurde) Intel MKL:

```
In [201]: from numpy.linalg import inv, qr

In [202]: X = np.random.randn(5, 5)

In [203]: mat = X.T.dot(X)

In [204]: inv(mat)
Out[204]:
array([[ 3.0361, -0.1808, -0.6878, -2.8285, -1.1911],
       [-0.1808,  0.5035,  0.1215,  0.6702,  0.0956],
       [-0.6878,  0.1215,  0.2904,  0.8081,  0.3049],
       [-2.8285,  0.6702,  0.8081,  3.4152,  1.1557],
       [-1.1911,  0.0956,  0.3049,  1.1557,  0.6051]])

In [205]: mat.dot(inv(mat))
Out[205]:
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.]])

In [206]: q, r = qr(mat)

In [207]: r
Out[207]:
array([[ -6.9271,   7.389 ,   6.1227,  -7.1163,  -4.9215],
       [  0.    ,  -3.9735,  -0.8671,   2.9747,  -5.7402],
       [  0.    ,   0.    , -10.2681,   1.8909,   1.6079],
       [  0.    ,   0.    ,   0.    ,  -1.2996,   3.3577],
       [  0.    ,   0.    ,   0.    ,   0.    ,   0.5571]])
```

Tabelle 4-7 enthält eine Liste von häufig genutzten Funktionen für lineare Algebra.

Die wissenschaftliche Python-Gemeinde hofft noch immer, dass eines Tages ein Infix-Operator für Matrixmultiplikation implementiert wird, sodass es eine syntaktisch schönere Schreibweise gibt als `np.dot`. Aber bis dahin gibt es nur diesen einen Weg.

Tabelle 4-7: Gebräuchliche Funktionen von numpy.linalg

Funktion	Beschreibung
`diag`	Selektiert die diagonalen (oder dazu versetzten) Elemente einer quadratischen Matrix als eindimensionales Array oder konvertiert ein eindimensionales Array in eine quadratische Matrix durch Füllen mit Nullen neben der Diagonalen.
`dot`	Matrixmultiplikation.
`trace`	Berechnet die Summe der diagonalen Elemente (die Spur).
`det`	Berechnet die Determinante.
`eig`	Berechnet die Eigenwerte und Eigenvektoren einer quadratischen Matrix.
`inv`	Berechnet die Inverse einer quadratischen Matrix.
`pinv`	Berechnet die Moore-Penrose-Pseudo-Inverse einer Matrix.
`qr`	Berechnet die QR-Dekomposition.
`svd`	Berechnet die Singular Value Dekomposition (SVD).
`solve`	Löst das lineare Gleichungssystem $Ax = b$ in x, wobei A eine quadratische Matrix ist.
`lstsq`	Berechnet die Lösung der kleinsten Quadrate für $Ax = b$.

Erzeugen von Zufallszahlen

Das Modul `numpy.random` ergänzt das eingebaute Python-Modul `random` um Funktionen zum effizienten Generieren ganzer Arrays von Zufallszahlen nach einer Vielzahl von Wahrscheinlichkeitsverteilungen. Sie können zum Beispiel ein 4 x 4-Array von Zufallszahlen mit der Standard-Normalverteilung durch `normal` bekommen:

```
In [208]: samples = np.random.normal(size=(4, 4))

In [209]: samples
Out[209]:
array([[ 0.1241,  0.3026,  0.5238,  0.0009],
       [ 1.3438, -0.7135, -0.8312, -2.3702],
       [-1.8608, -0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329, -2.3594]])
```

Das in Python eingebaute Modul `random` hingegen generiert nur eine Stichprobe zur gleichen Zeit. Wie Sie in diesem Benchmark sehen können, ist `numpy.random` hier locker eine Größenordnung schneller beim Generieren sehr großer Stichproben:

```
In [210]: from random import normalvariate

In [211]: N = 1000000

In [212]: %timeit samples = [normalvariate(0, 1) for _ in xrange(N)]
1 loops, best of 3: 1.33 s per loop

In [213]: %timeit np.random.normal(size=N)
10 loops, best of 3: 57.7 ms per loop
```

In Tabelle 4-8 haben Sie eine unvollständige Liste der in `numpy.random` verfügbaren Funktionen. Im nächsten Abschnitt gebe ich einige Beispiele dafür, wie man sehr viele Stichproben am Stück generieren kann.

Tabelle 4-8: Unvollständige Liste der Funktionen in numpy.random

Funktion	Beschreibung
`seed`	Setzt den Startwert des Zufallsgenerators.
`permutation`	Erzeugt eine zufällige Permutation einer Sequenz oder einen permutierten Bereich.
`shuffle`	Vermischt eine Sequenz an Ort und Stelle.
`rand`	Zieht Stichproben von einer Gleichverteilung.
`randint`	Zieht zufällige Ganzzahlen aus einem gegebenen Intervall.
`randn`	Zieht Stichproben aus einer Normalverteilung mit Mittelwert 0 und Standardabweichung 1 (Interface wie bei MATLAB).
`binomial`	Zieht Stichproben aus einer Binomialverteilung.
`normal`	Zieht Stichproben aus einer (Gaußschen) Normalverteilung.
`beta`	Zieht Stichproben aus einer Betaverteilung.
`chisquare`	Zieht Stichproben aus einer Chi-Quadrat-Verteilung.
`gamma`	Zieht Stichproben aus einer Gammaverteilung.
`uniform`	Zieht Stichproben aus einer [0, 1) Gleichverteilung.

Beispiel: Random Walks

Ein anschauliches Beispiel mit Operationen auf ganzen Array ist die Simulation von Random Walks, auch Irrfahrten genannt. Ziehen wir zunächst einen einfachen Random Walk in Betracht, der bei 0 anfängt und bei dem Schritte mit 1 oder 1 die gleiche Wahrscheinlichkeit haben. Ein 1.000 Schritte langer Random Walk in Python allein und mit eingebautem `random`-Modul sähe ungefähr so aus:

```
import random
position = 0
walk = [position]
steps = 1000
for i in range(steps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
```

In Abbildung 4-4 sehen Sie ein Plot der ersten 100 Schritte eines dieser Random Walks.

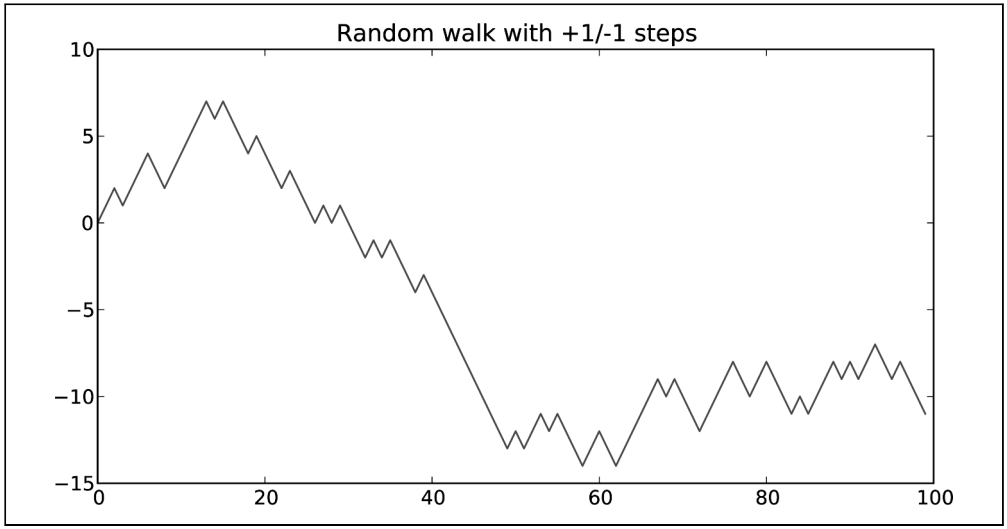


Abbildung 4-4: Eine einfache Irrfahrt

Es mag Ihnen aufgefallen sein, dass `walk` nichts weiter ist als die kumulative Summe der zufälligen Schritte und dass es als Array-Ausdruck berechnet werden könnte. Gut, dann benutze ich jetzt das Modul `np.random`, um mit einem Schlag 1.000-mal eine Münze zu werfen, setze die Werte entsprechend auf 1 oder –1 und berechne die kumulative Summe:

```
In [215]: nsteps = 1000

In [216]: draws = np.random.randint(0, 2, size=nsteps)

In [217]: steps = np.where(draws > 0, 1, -1)

In [218]: walk = steps.cumsum()
```

Wir können jetzt anfangen, Statistiken zu berechnen wie das Minimum und Maximum entlang der Bahnkurve:

```
In [219]: walk.min()        In [220]: walk.max()
Out[219]: -3                Out[220]: 31
```

Ein wenig komplizierter ist die Statistik der *ersten Durchgangszeit*, der Anzahl Schritte, bis der Random Walk einen bestimmten Wert erreicht. Hier wollen wir wissen, wie lange es dauert, bis die Irrfahrt wenigstens 10 Schritte vom Ausgangspunkt 0 entfernt ist, egal in welche Richtung. `np.abs(walk) >= 10` erzeugt ein boolesches Array, wenn der Walk 10 erreicht oder überschreitet, wir möchten aber gern den Index der *ersten* 10 oder –10 wissen. Es stellt sich heraus, dass man es mit `argmax` berechnen kann, das den ersten Index des Maximalwerts im booleschen Array sucht (`True` ist der Maximalwert):

```
In [221]: (np.abs(walk) >= 10).argmax()
Out[221]: 37
```

Man beachte, dass argmax wie hier nicht immer die effizienteste Lösung ist, weil das immer ein volles Absuchen des Arrays bedeutet. In diesem speziellen Fall wissen wir ja schon beim ersten True, dass wir das Maximum erreicht haben.

Simulation vieler Random Walks gleichzeitig

Wäre es Ihr Ziel, viele Random Walks zu simulieren – sagen wir 5.000 Stück –, bräuchten Sie nur wenig zu verändern, um alle Ergebnisse gleichzeitig zu berechnen. Wenn die Funktion numpy.random ein 2er-Tupel für den Parameter size bekommt, generiert sie ein zweidimensionales Array aus Ziehungen, und wir können die kumulative Summe über die Zeilen berechnen, um alle 5.000 Random Walks auf einen Schuss zu bekommen:

```
In [222]: nwalks = 5000

In [223]: nsteps = 1000

In [224]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1

In [225]: steps = np.where(draws > 0, 1, -1)

In [226]: walks = steps.cumsum(1)

In [227]: walks
Out[227]:
array([[  1,   0,   1, ...,   8,   7,   8],
       [  1,   0,  -1, ...,  34,  33,  32],
       [  1,   0,  -1, ...,   4,   5,   4],
       ...,
       [  1,   2,   1, ...,  24,  25,  26],
       [  1,   2,   3, ...,  14,  13,  14],
       [ -1,  -2,  -3, ..., -24, -23, -22]])
```

Wir können nun die maximalen und minimalen Werte über alle diese Walks berechnen:

```
In [228]: walks.max()        In [229]: walks.min()
Out[228]: 138                Out[229]: -133
```

Lassen Sie uns nun aus diesen Walks die erste Durchgangszeit für 30 oder –30 berechnen. Das ist ein wenig trickreich, weil nicht alle 5.000 Walks bis zur 30 kommen. Wir können das vorher abprüfen mittels der Methode any:

```
In [230]: hits30 = (np.abs(walks) >= 30).any(1)

In [231]: hits30
Out[231]: array([False, True, False, ..., False, True, False], dtype=bool)

In [232]: hits30.sum() # Number that hit 30 or -30
Out[232]: 3410
```

Mit diesem booleschen Array können wir jetzt die Zeilen aus walks nehmen, die tatsächlich den absoluten Abstand von 30 erreichen, um dann mit argmax über Achse 1 die Durchgangszeiten zu erhalten:

```
In [233]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [234]: crossing_times.mean()
Out[234]: 498.88973607038122
```

Probieren Sie ruhig noch ein wenig mit anderen Verteilungen herum als nur mit den gleichverteilten Münzwürfen. Sie brauchen lediglich einen anderen Zufallszahlengenerator, etwas wie die Funktion `normal`, die Normalverteilungen mit einstellbarem Mittelwert und Standardabweichung erzeugt:

```
In [235]: steps = np.random.normal(loc=0, scale=0.25,
   .....:                          size=(nwalks, nsteps))
```