



Capítulo 4. Conceptos básicos de NumPy: matrices y computación vectorizada

NumPy, abreviatura de Python numérico, es uno de los paquetes fundamentales más importantes para la computación numérica en Python. La mayoría de los paquetes computacionales que proporcionan funcionalidad científica utilizan los objetos de matriz de NumPy como la *lengua franca* para el intercambio de datos.

Estas son algunas de las cosas que encontrarás en NumPy:

- ndarray, un eficiente matriz multidimensional que proporciona operaciones aritméticas rápidas orientadas a la matriz y capacidades de *transmisión* flexibles.
- Funciones matemáticas para operaciones rápidas en conjuntos completos de datos sin tener que escribir bucles.
- Herramientas para leer / escribir datos de matriz en el disco y trabajar con archivos mapeados en memoria.
- Álgebra lineal, generación de números aleatorios y capacidades de transformación de Fourier.
- API de CA para conectar NumPy con bibliotecas escritas en C, C++ o FORTRAN.

Debido a que NumPy proporciona una API de C fácil de usar, es sencillo pasar datos a bibliotecas externas escritas en un lenguaje de bajo nivel y también para que las bibliotecas externas devuelvan datos a Python como matrices NumPy. Esta característica ha hecho de Python un lenguaje de elección para envolver bases de código C / C++ / Fortran heredadas y brindarles una interfaz dinámica y fácil de usar.

Si bien NumPy por sí solo no proporciona modelado o funcionalidad científica, comprender los arreglos NumPy y la computación orientada a arreglos lo ayudará a usar herramientas con semántica orientada a arreglos, como pandas, de manera mucho más efectiva. Dado que NumPy es un tema importante, cubriré muchas características avanzadas de NumPy, como la transmisión en mayor profundidad más adelante (ver [Apéndice A](#)).

Para la mayoría de las aplicaciones de análisis de datos, las principales áreas de funcionalidad en las que me centraré son:

- Operaciones rápidas de matriz vectorizada para munging y limpieza de datos, subconjunto y filtrado, transformación y cualquier otro tipo de cálculos
- Algoritmos de matriz comunes como operaciones de clasificación, únicas y de conjuntos
- Estadísticas descriptivas eficientes y datos de agregación / resumen
- Alineación de datos y manipulaciones de datos relacionales para fusionar y unir conjuntos de datos heterogéneos
- Expresando lógica condicional como expresiones de matriz en lugar de bucles con `if-elif-else`ramas
- Manipulaciones de datos grupales (agregación, transformación, aplicación de funciones)

Si bien NumPy proporciona una base computacional para el procesamiento general de datos numéricos, muchos lectores querrán usar pandas como base para la mayoría de los tipos de estadísticas o análisis, especialmente en datos tabulares. pandas también proporciona algunas funcionalidades más específicas del dominio, como la manipulación de series temporales, que no está presente en NumPy.

NOTA

La computación orientada a matrices en Python tiene sus raíces en 1995, cuando Jim Hugunin creó la biblioteca numérica. Durante los siguientes 10 años, muchas comunidades de programación científica comenzaron a programar en matriz en Python, pero el ecosistema de la biblioteca se había fragmentado a principios de la década de 2000. En 2005, Travis Oliphant pudo forjar el proyecto NumPy a partir de los proyectos Numeric y Numarray para unir a la comunidad en torno a un marco de cómputo de matriz única.



Una de las razones por las que NumPy es tan importante para los cálculos numéricos en Python es porque está diseñado para la eficiencia en grandes conjuntos de datos. Hay un número de razones para esto:

- NumPy almacena internamente datos en un bloque contiguo de memoria, independiente de otros objetos Python integrados. La biblioteca de algoritmos de NumPy escrita en lenguaje C puede operar en esta memoria sin ningún tipo de verificación u otra sobrecarga. Las matrices NumPy también usan mucha menos memoria que las secuencias integradas de Python.
- Las operaciones NumPy realizan cálculos complejos en matrices enteras sin la necesidad de `for`bucles Python .

Para darle una idea de la diferencia de rendimiento, considere una matriz NumPy de un millón de enteros y la lista equivalente de Python:

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

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

Ahora multipliquemos cada secuencia por 2:

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 10 ms, total: 30 ms
Wall time: 31.3 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 680 ms, sys: 180 ms, total: 860 ms
Wall time: 861 ms
```

Los algoritmos basados en NumPy son generalmente de 10 a 100 veces más rápidos (o más) que sus equivalentes de Python puros y usan significativamente menos memoria.

4.1 La matriz de datos NumPy: un objeto de matriz multidimensional

Una de las claves Las características de NumPy es su objeto de matriz N-dimensional, o `ndarray`, que es un contenedor rápido y flexible para grandes conjuntos de datos en Python. Las matrices le permiten realizar operaciones matemáticas en bloques completos de datos utilizando una sintaxis similar a las operaciones equivalentes entre elementos escalares.

Para darle una idea de cómo NumPy habilita los cálculos por lotes con una sintaxis similar a los valores escalares en objetos Python incorporados, primero importo NumPy y genero una pequeña matriz de datos aleatorios:

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

# Generate some random data
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[ -0.2047,  0.4789, -0.5194],
       [-0.5557,  1.9658,  1.3934]])
```

Luego escribo operaciones matemáticas con `data`:

```
In [15]: data * 10
Out[15]:
array([[ -2.0471,   4.7894, -5.1944],
       [-5.5573,  19.6578,  13.9341]])

In [16]: data + data
Out[16]:
array([[ -0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```

En el primer ejemplo, todos los elementos se han multiplicado por 10. En el segundo, los valores correspondientes en cada "celda" en la matriz se han agregado entre sí.

NOTA

En este capítulo y en todo el libro, uso el convención estándar de NumPy de usar siempre `import numpy as np`. Por supuesto, puede poner `from numpy import *` su código para evitar tener que escribir `np.`, pero le aconsejo que no haga el hábito. El `numpy`espacio de nombres grande y contiene una serie de funciones cuyos nombres entran en conflicto con las funciones integradas de Python (como `min` y `max`).

Un `ndarray` es un contenedor multidimensional genérico para datos homogéneos; es decir, todos los elementos deben ser del mismo tipo. Cada conjunto tiene un `shape`, un tupla que indica el tamaño de cada dimensión, y a `dtype`, un objeto que describe el *tipo* de *datos* de la matriz:

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

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

Este capítulo le presentará los conceptos básicos del uso de matrices NumPy, y debería ser suficiente para seguir junto con el resto del libro. Si bien no es necesario tener un conocimiento profundo de NumPy para muchas aplicaciones de análisis de datos, convertirse en un experto en programación y pensamiento orientado a matrices es un paso clave en el camino para convertirse en un Python científicogurú.



NOTA

Siempre que vea "matriz", "matriz NumPy" o "ndarray" en el texto, con pocas excepciones, todos se refieren a lo mismo: el objeto ndarray.

Crear ndarrays

La forma más fácil de crear una matriz es usar La `array` función. Esto acepta cualquier objeto similar a una secuencia (incluidas otras matrices) y produce una nueva matriz NumPy que contiene los datos pasados. Por ejemplo, una lista es un buen candidato para la conversión:

```
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. ])
```

Las secuencias anidadas, como una lista de listas de igual longitud, se convertirán en una matriz multidimensional:

```
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]])
```

Como `data2` era una lista de listas, la matriz NumPy `arr2` tiene dos dimensiones con forma inferida de los datos. Podemos confirmar esto por inspección de los atributos `ndim` y `shape`:

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

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

A menos que se especifique explícitamente (más sobre esto más adelante), `np.array` intenta inferir un buen tipo de datos para la matriz que crea. El tipo de datos se almacena en un `dtype` objeto especial de metadatos; por ejemplo, en los dos ejemplos anteriores tenemos:

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

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

Además de `np.array`, hay una serie de otras funciones para crear nuevas matrices. Como ejemplos, `zeros` y `ones` crean matrices de 0s o 1s, respectivamente, con una longitud o forma dada. `empty` crea una matriz sin inicializar sus valores a ningún valor particular. Para crear una matriz de dimensiones superiores con estos métodos, pase una tupla para la forma:

```
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.]])
```

PRECAUCIÓN

No es seguro asumir que `np.empty` devolverá una matriz de todos los ceros. En algunos casos, puede devolver valores de "basura" no inicializados.

`arange` es una versión con valor de matriz de la función incorporada de Python `range`:

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

Consulte la [Tabla 4-1](#) para obtener una lista breve de las funciones de creación de matriz estándar. Dado que NumPy se centra en la computación numérica, el tipo de datos, si no se especifica, en muchos casos será `float64` (punto flotante).



Tabla 4-1. Funciones de creación de matriz

Función	Descripción
<code>array</code>	Convertir entrada datos (lista, tupla, matriz u otro tipo de secuencia) a un ndarray ya sea deduciendo un tipo de letra o especificando explícitamente un tipo de letra; copia los datos de entrada por defecto
<code>asarray</code>	Convertir entrada a ndarray, pero no copie si la entrada ya es un ndarray
<code>arange</code>	Como el incorporado <code>range</code> pero devuelve un ndarray en lugar de una lista
<code>ones</code> , <code>ones_like</code>	Producir una matriz de todos los 1s con la forma y el tipo dados; <code>ones_like</code> toma otra matriz y produce una matriz de unidades de la misma forma y tipo
<code>zeros</code> , <code>zeros_like</code>	Me gusta <code>ones</code> <code>ones_like</code> produciendo matrices de 0s en su lugar
<code>empty</code> , <code>empty_like</code>	Crear nuevas matrices por asignar nueva memoria, pero no llenar con ningún valor como <code>ones</code> <code>zeros</code>
<code>full</code> , <code>full_like</code>	Producir una matriz de la forma y dtype dados con todos los valores establecidos en el "valor de relleno" indicado, <code>full_like</code> toma otra matriz y produce una matriz llena de la misma forma y dtype
<code>eye</code> , <code>identity</code>	Crear un matriz de identidad cuadrada $N \times N$ (1s en la diagonal y 0s en otra parte)

Tipos de datos para ndarrays

El *tipo de datos* o `dtype` es un objeto especial que contiene la información (o *metadatos* , datos sobre datos) que el ndarray necesita para interpretar una porción de memoria como un tipo particular de datos:

```
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')
```

Los `dtypes` son una fuente de flexibilidad de NumPy para interactuar con datos provenientes de otros sistemas. En la mayoría de los casos, proporcionan un mapeo directamente en un disco subyacente o una representación de memoria, lo que facilita la lectura y escritura de secuencias binarias de datos en el disco y también la conexión al código escrito en un lenguaje de bajo nivel como C o Fortran. Los tipos numéricos se nombran de la misma manera: un nombre de tipo, like `float` `int`, seguido de un número que indica el número de bits por elemento. Un valor estándar de punto flotante de doble precisión (lo que se usa debajo del capó en el `float` objeto de Python) ocupa 8 bytes o 64 bits. Por lo tanto, este tipo se conoce en NumPy como `float64`. Consulte la [Tabla 4-2](#) para obtener una lista completa de los tipos de datos admitidos de NumPy.

NOTA

No se preocupe por memorizar los tipos NumPy, especialmente si es un usuario nuevo. A menudo solo es necesario preocuparse por el *tipo* general de datos con los que está tratando, ya sea punto flotante, complejo, entero, booleano, cadena u objeto general de Python. Cuando necesita más control sobre cómo se almacenan los datos en la memoria y en el disco, especialmente los conjuntos de datos grandes, es bueno saber que tiene control sobre el tipo de almacenamiento.



Tabla 4-2. Tipos de datos NumPy

Tipo	Código de tipo	Descripción
int8, uint8	i1, u1	Firmado y tipos enteros de 8 bits (1 byte) sin signo
int16, uint16	i2, u2	Firmado y tipos enteros de 16 bits sin signo
int32, uint32	i4, u4	Firmado y tipos enteros de 32 bits sin signo
int64, uint64	i8, u8	Firmado y tipos enteros de 64 bits sin signo
float16	f2	Media precisión punto flotante
float32	f4 or f	Estándar punto flotante de precisión simple; compatible con flotador C
float64	f8 or d	Estándar punto flotante de doble precisión; compatible con C doble y float objeto Python
float128	f16 or g	Precisión extendida punto flotante
complex64 complex128 complex256	c8, c16, c32	Complejo números representados por dos flotadores de 32, 64 o 128, respectivamente
bool	?	Booleano tipo de almacenamiento Truey Falsevalores
object	O	Pitón tipo de objeto; un valor puede ser cualquier objeto de Python
string_	S	Longitud fija Tipo de cadena ASCII (1 byte por carácter); por ejemplo, para crear un tipo de cadena con longitud 10, use 'S10'
unicode_	U	Longitud fija Tipo Unicode (número de bytes específico de la plataforma); semántica de la misma especificación que string_(por ejemplo, 'U10')

Puede convertir de forma explícita o *emitir* una serie de uno a otro mediante `dtype.astype` método de ndarray :

```
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.dtype
Out[40]: dtype('float64')
```

En este ejemplo, los enteros se convirtieron en coma flotante. Si lanzo algunos números de punto flotante para que sean de tipo entero, la parte decimal se truncará:

```
In [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

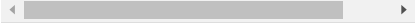
In [42]: arr
Out[42]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])

In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

Si tiene una serie de cadenas que representan números, puede usarlas `astype`para convertirlas en forma numérica:

```
In [44]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.:

In [45]: numeric_strings.astype(float)
Out[45]: array([ 1.25, -9.6 , 42.  ])
```



PRECAUCIÓN

Es importante tener cuidado al usar el `numpy.string_tipo`, ya que los datos de cadena en NumPy tienen un tamaño fijo y pueden truncar la entrada sin previo aviso. pandas tiene un comportamiento más intuitivo y listo para usar en datos no numéricos.

Si la conversión fallara por alguna razón (como una cadena que no puede ser cconvertido a float64), ValueErrorse levantará un testimonio. Aquí era un



poco vago y escribí en float lugar de np.float64; NumPy alias los tipos de Python a sus propios tipos de datos equivalentes.

También puedes usar otra matriz atributo dtype:

```
In [46]: int_array = np.arange(10)

In [47]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype

In [48]: int_array.astype(calibers.dtype)
Out[48]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.]
```

Hay cadenas de código de tipo abreviado que también puede usar para referirse a un dtype:

```
In [49]: empty_uint32 = np.empty(8, dtype='u4')

In [50]: empty_uint32
Out[50]:
array([[ 0, 1075314688,  0, 1075707904,  0,
        1075838976,  0, 1072693248], dtype=uint32)
```

NOTA

Las llamadas *astype* *siempre* crean una nueva matriz (una copia de los datos), incluso si el nuevo dtype es el mismo que el antiguo.

Aritmética con matrices NumPy

Las matrices son importantes porque le permiten expresar operaciones por lotes en datos sin escribir ningún *for* bucle. Los usuarios de NumPy llaman *Esta vectorización* . Cualquier operación aritmética entre matrices de igual tamaño aplica la operación por elementos:

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

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

In [53]: arr * arr
Out[53]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

In [54]: arr - arr
Out[54]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Aritmética Las operaciones con escalares propagan el argumento escalar a cada elemento de la matriz:

```
In [55]: 1 / arr
Out[55]:
array([[ 1.   ,  0.5   ,  0.3333],
       [ 0.25 ,  0.2   ,  0.1667]])

In [56]: arr ** 0.5
Out[56]:
array([[ 1.   ,  1.4142,  1.7321],
       [ 2.   ,  2.2361,  2.4495]])
```

Las comparaciones entre matrices del mismo tamaño producen matrices booleanas:

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

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

In [59]: arr2 > arr
Out[59]:
array([[False,  True, False],
       [ True, False,  True]], dtype=bool)
```

Las operaciones entre matrices de diferentes tamaños son llamado *radiodifusión* y será discutido con más detalle en el **Apéndice A** . Tener una comprensión profunda de la transmisión no es necesario para la mayoría de este libro.

Indexación básica y segmentación

Matriz NumPy La indexación es un tema rico, ya que hay muchas maneras en que puede querer seleccionar un subconjunto de sus datos o elementos individuales. Las matrices unidimensionales son simples; en la superficie actúan de manera similar a las listas de Python:

```
In [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])

In [64]: arr[5:8] = 12

In [65]: arr
Out[65]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Como puede ver, si asigna un valor escalar a un segmento, como en `arr[5:8] = 12`, el valor se propaga (o se *difunde* en adelante) a toda la selección. Una primera distinción importante de las listas integradas de Python es que los



segmentos de matriz son *vistas* en la matriz original. Esto significa que los datos no se copian y cualquier modificación de la vista se reflejará en la matriz de origen.

Para dar un ejemplo de esto, primero creo una porción de `arr`:

```
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])
```

Ahora, cuando cambio los valores `arr_slice`, las mutaciones se reflejan en la matriz original `arr`:

```
In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([ 0, 1, 2, 3, 4, 12, 12345,
9])
```

El segmento "desnudo" `[:]` se asignará a todos los valores de una matriz:

```
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: array([ 0, 1, 2, 3, 4, 64, 64, 8, 9])
```

Si es nuevo en NumPy, esto puede sorprenderle, especialmente si ha utilizado otros lenguajes de programación de matriz que copian datos con mayor entusiasmo. Como NumPy ha sido diseñado para poder trabajar con matrices muy grandes, podría imaginar problemas de rendimiento y memoria si NumPy insistiera en copiar siempre los datos.

PRECAUCIÓN

Si desea una copia de una porción de un ndarray en lugar de una vista, deberá copiar explícitamente la matriz:por ejemplo `arr[5:8].copy()`.

Con matrices de dimensiones superiores, tiene muchas más opciones. En una matriz bidimensional, los elementos en cada índice ya no son escalares sino más bien matrices unidimensionales:

```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [73]: arr2d[2]
Out[73]: array([7, 8, 9])
```

Por lo tanto, los elementos individuales se pueden acceder de forma recursiva. Pero eso es demasiado trabajo, por lo que puede pasar una lista de índices separados por comas para seleccionar elementos individuales. Entonces estos son equivalentes:

```
In [74]: arr2d[0][2]
Out[74]: 3

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

Consulte la [Figura 4-1](#) para ver una ilustración de indexación en una matriz bidimensional. Me resulta útil pensar en el eje 0 como las "filas" de la matriz y el eje 1 como las "columnas".

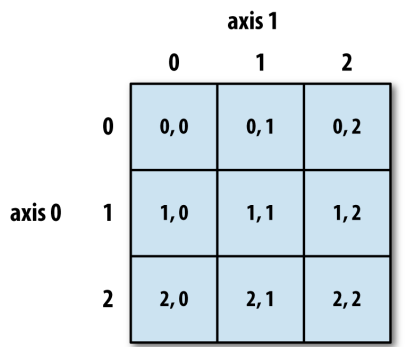


Figura 4-1. Elementos de indexación en una matriz NumPy

En matrices multidimensionales, si omite los índices posteriores, el objeto devuelto será un ndarray de dimensiones inferiores que constará de todos los datos a lo largo de las dimensiones superiores. Entonces, en la matriz $2 \times 2 \times 3$ `arr3d`:

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

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

`arr3d[0]` es una matriz de 2×3 :

```
In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
[4, 5, 6]])
```



Se pueden asignar tanto valores escalares como matrices a `arr3d[0]`:

```
In [79]: old_values = arr3d[0].copy()

In [80]: arr3d[0] = 42

In [81]: arr3d
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [82]: arr3d[0] = old_values

In [83]: arr3d
Out[83]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Del mismo modo, `arr3d[1, 0]` le ofrece todos los valores cuyos índices comienzan con `(1, 0)`, formando una matriz unidimensional:

```
In [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])
```

Esta expresión es la misma que si hubiéramos indexado en dos pasos:

```
In [85]: x = arr3d[1]

In [86]: x
Out[86]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [87]: x[0]
Out[87]: array([7, 8, 9])
```

Tenga en cuenta que en todos estos casos donde se han seleccionado subsecciones de la matriz, las matrices devueltas son vistas.

INDEXACIÓN CON RODAJAS

Al igual que los objetos unidimensionales, como las listas de Python, los ndarrays se pueden dividir con la sintaxis familiar:

```
In [88]: arr
Out[88]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [89]: arr[1:6]
Out[89]: array([ 1,  2,  3,  4, 64])
```

Considere la matriz bidimensional de antes, `arr2d`. Cortar esta matriz es un poco diferente :

```
In [90]: arr2d
Out[90]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

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

Como puede ver, se ha cortado a lo largo del eje 0, el primer eje. Una porción, por lo tanto, selecciona un rango de elementos a lo largo de un eje. Puede ser útil leer la expresión `arr2d[:2]` como "seleccione las dos primeras filas de `arr2d`".

Puede pasar varios sectores al igual que puede pasar múltiples índices:

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

Cuando se corta de esta forma, siempre obtiene vistas de matriz del mismo número de dimensiones. Al mezclar índices enteros y sectores, obtienes sectores de dimensiones más bajas.

Por ejemplo, puedo seleccionar la segunda fila pero solo las dos primeras columnas de esta manera:

```
In [93]: arr2d[1, :2]
Out[93]: array([4, 5])
```

Del mismo modo, puedo seleccionar la tercera columna, pero solo las dos primeras filas así:

```
In [94]: arr2d[:2, 2]
Out[94]: array([3, 6])
```

Vea la **Figura 4-2** para una ilustración. Tenga en cuenta que dos puntos por sí solo significa tomar todo el eje, por lo que puede cortar solo ejes de dimensiones superiores haciendo:

```
In [95]: arr2d[:, :1]
Out[95]:
array([[1],
       [4],
       [7]])
```

Por supuesto, la asignación a una expresión de división se asigna al conjunto selección:

```
In [96]: arr2d[:2, 1:] = 0

In [97]: arr2d
Out[97]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```



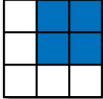
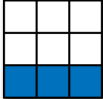

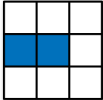
	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	<code>(3,)</code> <code>(3,)</code> <code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	<code>(2,)</code> <code>(1, 2)</code>

Figura 4-2. Corte de matriz bidimensional

Indexación booleana

Vamos considere un ejemplo donde tenemos algunos datos en una matriz y una matriz de nombres con duplicados. Voy a usar aquí la función `numpy.random` para generar algunos datos aleatorios normalmente distribuidos:

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe'])
In [99]: data = np.random.randn(7, 4)

In [100]: names
Out[100]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='<U4')

In [101]: data
Out[101]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3826,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Supongamos que cada nombre corresponde a una fila en la `data` matriz y queremos seleccionar todas las filas con el nombre correspondiente 'Bob'. Al igual que las operaciones aritméticas, las comparaciones (como `==`) con las matrices también se vectorizan. Así, Al comparar `names` con la cadena se 'Bob' obtiene una matriz booleana:

```
In [102]: names == 'Bob'
Out[102]: array([ True, False, False,  True, False, False, False], dt
```

Esta matriz booleana se puede pasar al indexar la matriz:

```
In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

La matriz booleana debe tener la misma longitud que el eje de matriz que está indexando. Incluso puede mezclar y combinar matrices booleanas con sectores o enteros (o secuencias de enteros; más sobre esto más adelante).

PRECAUCIÓN

La selección booleana no fallará si la matriz booleana no tiene la longitud correcta, por lo que recomiendo tener cuidado al usar esta función.

En estos ejemplos, selecciono de las filas donde `names == 'Bob'` e indexo las columnas, también:

```
In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])

In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464,  0.477 ])
```

Para seleccionar todo pero 'Bob', puedes usar `!=` o negar la condición usando `~`:

```
In [106]: names != 'Bob'
Out[106]: array([False,  True,  True, False,  True,  True,  True], dt

In [107]: data[~(names == 'Bob')]
Out[107]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3826,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```



El ~operador puede ser útil cuando desea invertir una condición general:

```
In [108]: cond = names == 'Bob'

In [109]: data[~cond]
Out[109]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Al seleccionar dos de los tres nombres para combinar varias condiciones booleanas, use operadores aritméticos booleanos como &(y) y |(o):

```
In [110]: mask = (names == 'Bob') | (names == 'Will')

In [111]: mask
Out[111]: array([ True, False,  True,  True,  True, False, False], dtype=bool)

In [112]: data[mask]
Out[112]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

La selección de datos de una matriz mediante indexación booleana *siempre* crea una copia de los datos, incluso si la matriz devuelta no ha cambiado.

PRECAUCIÓN

Las palabras clave de Python `and` y `or` no funcionan con matrices booleanas. Use `&` (y) y `|` (o) en su lugar.

Establecer valores con matrices booleanas funciona de manera de sentido común. Para establecer todos los valores negativos en `data0` solo necesitamos hacer:

```
In [113]: data[data < 0] = 0

In [114]: data
Out[114]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072,  0. ,  0.275 ,  0.2289],
       [ 1.3529,  0.8864,  0. ,  0. ],
       [ 1.669 ,  0. ,  0. ,  0.477 ],
       [ 3.2489,  0. ,  0. ,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0. ,  0. ,  0. ,  0. ]])
```

Configurar filas o columnas enteras utilizando una matriz booleana unidimensional también es fácil:

```
In [115]: data[names != 'Joe'] = 7

In [116]: data
Out[116]:
array([[ 7. ,  7. ,  7. ,  7. ],
       [ 1.0072,  0. ,  0.275 ,  0.2289],
       [ 7. ,  7. ,  7. ,  7. ],
       [ 7. ,  7. ,  7. ,  7. ],
       [ 7. ,  7. ,  7. ,  7. ],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0. ,  0. ,  0. ,  0. ]])
```

Como veremos más adelante, este tipo de operaciones en datos bidimensionales son convenientes para hacer con pandas

Indexación de lujo

La *indexación de fantasía* es un término adoptado por NumPy para describir la indexación utilizando matrices de enteros. Supongamos que tenemos una matriz de 8 × 4:

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
.....:     arr[i] = i

In [119]: arr
Out[119]:
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.]])
```

Para seleccionar un subconjunto de las filas en un orden particular, simplemente puede pasar una lista o un conjunto de números enteros que especifique el orden deseado:

```
In [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

¡Ojalá este código hiciera lo que esperabas! El uso de índices negativos selecciona filas desde el final:

```
In [121]: arr[[-3, -5, -7]]
Out[121]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```



Pasar múltiples matrices de índice hace algo ligeramente diferente; selecciona una matriz unidimensional de elementos correspondientes a cada tupla de índices:

```

In [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
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 [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])

```

Miraremos el `reshape` método con más detalle en el [Apéndice A](#) .

Aquí los elementos $(1, 0)$, $(5, 3)$, $(7, 1)$, y $(2, 2)$ fueron seleccionados. Independientemente de cuántas dimensiones tenga la matriz (aquí, solo 2), el resultado de la indexación elegante siempre es unidimensional.

El comportamiento de la indexación elegante en este caso es un poco diferente de lo que algunos usuarios podrían haber esperado (incluido yo mismo), que es la región rectangular formada al seleccionar un subconjunto de las filas y columnas de la matriz. Aquí hay una forma de conseguir eso:

```

In [125]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[125]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])

```

Tenga en cuenta que la indexación elegante, a diferencia de la división, siempre copia los datos en una nueva matriz.

Transposición de matrices y cambio de ejes

La transposición es un forma especial de remodelación que de manera similar devuelve una vista de los datos subyacentes sin copiar nada. Las matrices tienen el `transpose` método y también el `T` atributo especial :

```

In [126]: arr = np.arange(15).reshape((3, 5))

In [127]: arr
Out[127]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [128]: arr.T
Out[128]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])

```

Al hacer cálculos matriciales, puede hacer esto muy a menudo, por ejemplo, al calcular el producto matriz interno utilizando `np.dot`:

```

In [129]: arr = np.random.randn(6, 3)

In [130]: arr
Out[130]:
array([[ -0.8608,  0.5681, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])

In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])

```

Para matrices de dimensiones superiores, `transpose` aceptará una tupla de números de eje para permutar los ejes (para una flexión mental adicional):

```

In [132]: arr = np.arange(16).reshape((2, 2, 4))

In [133]: arr
Out[133]:
array([[[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
         [12, 13, 14, 15]])])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[[ 0,  1,  2,  3],
         [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
         [12, 13, 14, 15]])])

```

Aquí, los ejes se han reordenado con el segundo eje primero, el primer eje segundo y el último eje sin cambios.

La transposición simple con `.` es un caso especial de intercambio de ejes. `ndarray` tiene el método `swapaxes`, que toma un par de números de eje y cambia los ejes indicados para reorganizar los datos:

```

In [135]: arr
Out[135]:
array([[[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
         [12, 13, 14, 15]])])

In [136]: arr.swapaxes(1, 2)
Out[136]:
array([[[[ 0,  4],
         [ 1,  5],
         [ 2,  6],

```



```
[ 3., 7]],  
[[ 8, 12],  
[ 9, 13],  
[10, 14],  
[11, 15]]])
```

swapaxes de manera similar, devuelve una vista de los datos sin hacer un Copiar.

4.2 Funciones universales: funciones rápidas de matriz de elementos sabios

Una función universal, o *ufunc*, es una función que realiza operaciones basadas en elementos en datos en ndarrays. Puede pensar en ellos como envoltorios vectorizados rápidos para funciones simples que toman uno o más valores escalares y producen uno o más resultados escalares.

Muchos ufuncs son transformaciones simples basadas en elementos, como sqrt o exp:

```
In [137]: arr = np.arange(10)  
  
In [138]: arr  
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])  
  
In [139]: np.sqrt(arr)  
Out[139]:  
array([ 0.        ,  1.        ,  1.4142,  1.7321,  2.        ,  2.2361,  2.4495  
        2.6458,  2.8284,  3.        ])  
  
In [140]: np.exp(arr)  
Out[140]:  
array([  1.        ,  2.7183,  7.3891, 20.0855, 54.5982,  
        148.4132, 403.4288, 1096.6332, 2980.958 , 8103.0839])
```

Estos se mencionan como ufuncs *unarios*. Otros, como addo *maximum*, toman dos matrices (por lo tanto, ufuncs *binarios*) y devolver una sola matriz como resultado:

```
In [141]: x = np.random.randn(8)  
  
In [142]: y = np.random.randn(8)  
  
In [143]: x  
Out[143]:  
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584,  
        -0.6685])  
  
In [144]: y  
Out[144]:  
array([ 0.8626, -0.01 ,  0.05 ,  0.6702,  0.853 , -0.9559, -0.0235,  
        -2.3042])  
  
In [145]: np.maximum(x, y)  
Out[145]:  
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  0.7584,  
        -0.6685])
```

Aquí, se `numpy.maximum` calculó el elemento máximo sabio de los elementos en `xy` y.

Si bien no es común, un ufunc puede devolver múltiples matrices. `modf` es un ejemplo, una versión vectorizada del Python incorporado `divmod`; esodevuelve las partes fraccionarias e integrales de una matriz de punto flotante:

```
In [146]: arr = np.random.randn(7) * 5  
  
In [147]: arr  
Out[147]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  
        ])  
  
In [148]: remainder, whole_part = np.modf(arr)  
  
In [149]: remainder  
Out[149]: array([-0.2623, -0.0915, -0.663 ,  0.3731,  0.6182,  0.45  
        ])  
  
In [150]: whole_part  
Out[150]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

Ufuncs acepta un outargumento opcional que les permite operar in situ en matrices:

```
In [151]: arr  
Out[151]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  
        ])  
  
In [152]: np.sqrt(arr)  
Out[152]: array([ nan,    nan,    nan,  2.318 ,  1.9022,  1.857  
        ])  
  
In [153]: np.sqrt(arr, arr)  
Out[153]: array([ nan,    nan,    nan,  2.318 ,  1.9022,  1.857  
        ])  
  
In [154]: arr  
Out[154]: array([ nan,    nan,    nan,  2.318 ,  1.9022,  1.857  
        ])
```

Consulte las Tablas 4-3 y 4-4 para obtener una lista de ufuncs disponibles.



Tabla 4-3. Ufuncs unarios

Función	Descripción
abs, fabs	Calcular el valor absoluto por elemento para valores enteros, de coma flotante o complejos
sqrt	Calcular el raíz cuadrada de cada elemento (equivalente a <code>arr ** 0.5</code>)
square	Calcular el cuadrado de cada elemento (equivalente a <code>arr ** 2</code>)
exp	Calcular el exponente e^x de cada elemento
log, log10, log2, log1p	Natural logaritmo (base e), log base 10, log base 2 y $\log(1 + x)$, respectivamente
sign	Calcular El signo de cada elemento: 1 (positivo), 0 (cero) o -1 (negativo)
ceil	Calcular el techo de cada elemento (es decir, el entero más pequeño mayor o igual a ese número)
floor	Calcular el piso de cada elemento (es decir, el entero más grande menor o igual a cada elemento)
rint	Redondo elementos al entero más cercano, preservando el <code>dtype</code>
modf	Regreso partes fraccionales e integrales de la matriz como una matriz separada
isnan	Regreso matriz booleana que indica si cada valor es NaN(No es un número)
isfinite, isinf	Regreso matriz booleana que indica si cada elemento es finito (no <code>inf</code> , no NaN) o infinito, respectivamente
cos, cosh, sin, sinh, tan, tanh	Regular y funciones trigonométricas hiperbólicas
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverso funciones trigonométricas
logical_not	Calcular la verdad valor de not \times elemento sabio (equivalente a <code>~arr</code>).

Tabla 4-4. Funciones universales binarias

Función	Descripción
add	Añadir elementos correspondientes en matrices
subtract	Sustraer elementos en la segunda matriz de la primera matriz
multiply	Multiplicar elementos de matriz
divide, floor_divide	Dividir o división del piso (truncando el resto)
power	Elevar elementos en la primera matriz a las potencias indicadas en la segunda matriz
maximum, fmax	Elemento sabio máximo; <code>fmax</code> ignoraNaN
minimum, fmin	Elemento sabio mínimo; <code>fmin</code> ignoraNaN
mod	Elemento sabio módulo (resto de división)
copysign	Copiar signo de valores en segundo argumento a valores en primer argumento
greater, greater_equal, less, less_equal, equal, not_equal	Realizar comparación de elementos sabios, produciendo una matriz booleana (equivalente a infijo operadores <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> , <code>==</code> , <code>!=</code>)
logical_and, logical_or, logical_xor	Calcular Valor de verdad de elementos sabios de la operación lógica (equivalente a infijo operadores <code>&</code> , <code> </code> , <code>^</code>)



Usando matrices NumPy le permite expresar muchos tipos de tareas de procesamiento de datos como expresiones concisas de matriz que de otro modo podrían requerir bucles de escritura. Esta práctica de reemplazar bucles explícitos con expresiones de matriz se conoce comúnmente como *vectorización*. En general, las operaciones de matriz vectorizadas a menudo serán uno o dos (o más) órdenes de magnitud más rápido que sus equivalentes de Python puros, con el mayor impacto en cualquier tipo de cálculos numéricos. Más adelante, en el **Apéndice A**, explico la *transmisión*, un método poderoso para vectorizar cálculos.

Como un ejemplo simple, supongamos que deseamos evaluar la función $\sqrt{x^2 + y^2}$ en una cuadrícula de valores regular. La `np.meshgrid` función toma dos matrices 1D y produce dos matrices 2D correspondientes a todos los pares de (x, y) en las dos matrices:

```
In [155]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

In [156]: xs, ys = np.meshgrid(points, points)

In [157]: ys
Out[157]:
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]])
```

Ahora, evaluar la función es cuestión de escribir la misma expresión que escribirías con dos puntos:

```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)

In [159]: z
Out[159]:
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]])
```

Como vista previa del **Capítulo 9**, uso `matplotlib` para crear visualizaciones de esta matriz bidimensional:

```
In [160]: import matplotlib.pyplot as plt

In [161]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[161]: <matplotlib.colorbar.Colorbar at 0x7fa37d95c5f8>

In [162]: plt.title("Image plot of  $\sqrt{x^2 + y^2}$  for a grid of va
Out[162]: Text(0.5,1,'Image plot of  $\sqrt{x^2 + y^2}$  for a grid of '
```

Ver **Figura 4-3**. Aquí utilicé la función `matplotlib.imshow` para crear un gráfico de imagen a partir de una matriz bidimensional de valores de funciones.

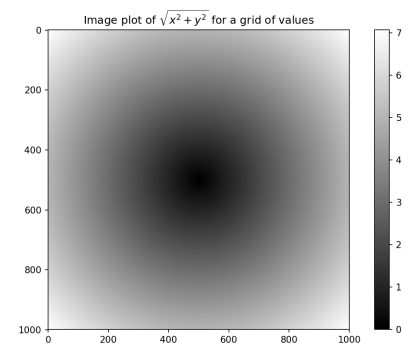


Figura 4-3. Gráfico de función evaluado en cuadrícula

Expresando lógica condicional como operaciones de matriz

La `numpy.where` función es una versión vectorizada de la expresión ternaria `x if condition else y`. Supongamos que tenemos una matriz booleana y dos matrices de valores:

```
In [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, False])
```

Supongamos que deseamos tomar un valor `xarr` cada vez que el valor correspondiente en `cond` es `True`, y de lo contrario tomar el valor de `yarr`. Una comprensión de la lista que hace esto podría verse así:

```
In [168]: result = [(x if c else y)
.....:               for x, y, c in zip(xarr, yarr, cond)]

In [169]: result
Out[169]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999]
```

Esto tiene múltiples problemas. Primero, no será muy rápido para matrices grandes (porque todo el trabajo se realiza en código Python interpretado). En segundo lugar, no funcionará con matrices multidimensionales. Con `np.where` usted puede escribir esto de manera muy concisa:

```
In [170]: result = np.where(cond, xarr, yarr)
```



```
In [171]: result
Out[171]: array([[ 1.1,  2.2,  1.3,  1.4,  2.5])
```

El segundo y el tercer argumento `np.where` necesitan ser matrices; uno o ambos pueden ser escalares. Un uso típico de `where` en el análisis de datos es producir una nueva matriz de valores basada en otra matriz. Suponga que tiene una matriz de datos generados aleatoriamente y desea reemplazar todos los valores positivos con 2 y todos los valores negativos con -2. Esto es muy fácil de hacer con `np.where`:

```
In [172]: arr = np.random.randn(4, 4)

In [173]: arr
Out[173]:
array([[ -0.5031, -0.6223, -0.9212, -0.7262],
       [  0.2229,  0.0513, -1.1577,  0.8167],
       [  0.4336,  1.0107,  1.8249, -0.9975],
       [  0.8506, -0.1316,  0.9124,  0.1882]])

In [174]: arr > 0
Out[174]:
array([[False, False, False, False],
       [ True,  True, False,  True],
       [ True,  True,  True, False],
       [ True, False,  True,  True]], dtype=bool)

In [175]: np.where(arr > 0, 2, -2)
Out[175]:
array([[ -2, -2, -2, -2],
       [  2,  2, -2,  2],
       [  2,  2,  2, -2],
       [  2, -2,  2,  2]])
```

Puede combinar escalares y matrices cuando lo use `np.where`. Por ejemplo, puedo reemplazar todos los valores positivos `arr` con la constante 2 así:

```
In [176]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[176]:
array([[ -0.5031, -0.6223, -0.9212, -0.7262],
       [  2.,      2.,    -1.1577,  2.      ],
       [  2.,      2.,      2.,    -0.9975],
       [  2.,    -0.1316,  2.,      2.      ]])
```

Las matrices pasadas `np.where` pueden ser más que simples matrices o escalares de igual tamaño.

Métodos matemáticos y estadísticos

Un conjunto de funciones matemáticas. que las estadísticas de cálculo sobre una matriz completa o sobre los datos a lo largo de un eje son accesibles como métodos de la clase de matriz. Puede usar agregaciones (a menudo llamadas *reducciones*) like `sum`, `mean` y `std` (desviación estándar) llamando al método de instancia de matriz o usando la función NumPy de nivel superior.

Aquí genero algunos datos aleatorios normalmente distribuidos y calculo algunas estadísticas agregadas:

```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.5851],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9009, -0.5095,  1.5817, -0.5287]])

In [179]: arr.mean()
Out[179]: 0.1960705119998253

In [180]: np.mean(arr)
Out[180]: 0.1960705119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

Las funciones tienen gusto `mean` y `sum` toman un `axis` argumento opcional que computa la estadística sobre el eje dado, resultando en una matriz con una dimensión menos:

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

Aquí, `arr.mean(1)` significa "calcular la media en las columnas" donde `arr.sum(0)` significa "Calcular suma abajo de las filas".

Otros métodos tienen gusto `cumsum` `cumprod` agregan, produciendo una matriz de los resultados intermedios:

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [185]: arr.cumsum()
Out[185]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

En las matrices multidimensionales, las funciones de acumulación como `cumsum` devolver una matriz del mismo tamaño, pero con los agregados parciales calculados a lo largo del eje indicado de acuerdo con cada segmento de dimensión inferior:

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [187]: arr
Out[187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [189]: arr.cumprod(axis=1)
Out[189]:
```



```
array([[ 0,  0,  0],
       [ 3, 12, 60],
       [ 6, 42, 336]])
```

Consulte la [Tabla 4-5](#) para obtener una lista completa. Veremos muchos ejemplos de estos métodos en acción en capítulos posteriores.

Tabla 4-5. Métodos estadísticos básicos de matriz

Método	Descripción
sum	La suma de todos los elementos en la matriz o a lo largo de un eje; las matrices de longitud cero tienen suma 0
mean	Significado aritmético; las matrices de longitud cero tienen una NaNmedia
std, var	Estándar desviación y varianza, respectivamente, con ajuste opcional de grados de libertad (denominador predeterminado n)
min, max	Mínimo y máximo
argmin, argmax	Índices de elementos mínimos y máximos, respectivamente
cumsum	Acumulativo suma de elementos a partir de 0
cumprod	Acumulativo producto de elementos a partir de 1

Métodos para matrices booleanas

Valores booleanos están obligados a 1 (True) y 0 (False) en los métodos anteriores. Por lo tanto, a `sum`n menudo se usa como un medio para contar Truevalores en una matriz booleana:

```
In [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum() # Number of positive values
Out[191]: 42
```

Hay dos adicionales métodos, `any` y `all`, útil especialmente para matrices booleanas. `any`prueba si uno o más valores en una matriz es `True`, mientras `all`verifica si cada valor es `True`:

```
In [192]: bools = np.array([False, False, True, False])

In [193]: bools.any()
Out[193]: True

In [194]: bools.all()
Out[194]: False
```

Estos métodos también funcionan con matrices no booleanas, donde se evalúan elementos distintos de cero `True`.

Clasificación

Al igual que el tipo de lista incorporada de Python, Las matrices NumPy se pueden ordenar en el lugar con el `sort`método:

```
In [195]: arr = np.random.randn(6)

In [196]: arr
Out[196]: array([-0.6095, -0.4938,  1.24  , -0.1357,  1.43  , -0.8469])

In [197]: arr.sort()

In [198]: arr
Out[198]: array([-0.8469, -0.4938, -0.1357,  0.6095,  1.24  ,  1.43  ])
```

Puede ordenar cada sección unidimensional de valores en una matriz multidimensional in situ a lo largo de un eje pasando el número de eje a `sort`:

```
In [199]: arr = np.random.randn(5, 3)

In [200]: arr
Out[200]:
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218  , -0.8948, -1.7415]])

In [201]: arr.sort(1)

In [202]: arr
Out[202]:
array([[ -0.2555,  0.6033,  1.2636],
       [-0.9616, -0.4457,  0.4684],
       [-1.8245,  0.6254,  1.0229],
       [-0.3501,  0.0909,  1.1074],
       [-1.7415, -0.8948,  0.218  ]])
```

El método de nivel superior `np.sort` devuelve una copia ordenada de una matriz en lugar de modificar la matriz en el lugar. Una forma rápida y sucia de calcular los cuantiles de una matriz es ordenarla y seleccionar el valor en un rango particular:

```
In [203]: large_arr = np.random.randn(1000)

In [204]: large_arr.sort()

In [205]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[205]: -1.5311513550102103
```



Para más detalles sobre el uso de métodos de clasificación de NumPy, y las técnicas más avanzadas como tipo indirectos, véase el [Apéndice A](#). También se pueden encontrar otros tipos de manipulaciones de datos relacionados con la ordenación (por ejemplo, ordenar una tabla de datos por una o más columnas) en pandas.

Lógica de conjunto única y otra

NumPy tiene algo básico establecer operaciones para ndarrays unidimensionales. Una de uso común es `np.unique`, que devuelve los valores únicos ordenados en una matriz:

```
In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe'])
In [207]: np.unique(names)
Out[207]:
array(['Bob', 'Joe', 'Will'],
      dtype='<U4')

In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

Contraste `np.unique` con la alternativa pura de Python:

```
In [210]: sorted(set(names))
Out[210]: ['Bob', 'Joe', 'Will']
```

Otra función `np.in1d`, prueba membresía de los valores en una matriz en otra, devolviendo una matriz booleana:

```
In [211]: values = np.array([5, 0, 0, 3, 2, 5, 6])
In [212]: np.in1d(values, [2, 3, 6])
Out[212]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

Consulte la [Tabla 4-6](#) para obtener una lista de las funciones establecidas en NumPy.

Tabla 4-6. Operaciones de conjunto de matrices

Método	Descripción
<code>unique(x)</code>	Calcular los elementos ordenados y únicos en <code>x</code>
<code>intersect1d(x, y)</code>	Calcule lo ordenado, elementos comunes en <code>xy</code>
<code>union1d(x, y)</code>	Calcular el ordenado unión de elementos
<code>in1d(x, y)</code>	Calcular un matriz booleana que indica si cada elemento de <code>x</code> está contenido en <code>y</code>
<code>setdiff1d(x, y)</code>	Establecer diferencia elementos en <code>x</code> que no están en <code>y</code>
<code>setxor1d(x, y)</code>	Establecer diferencias simétricas; elementos que están en cualquiera de las matrices, pero no ambos

4.4 Entrada y salida de archivos con matrices

NumPy puede guardar y cargar datos hacia y desde el disco en formato de texto o binario. En esta sección solo discuto el formato binario incorporado de NumPy, ya que la mayoría de los usuarios preferirán los pandas y otras herramientas para cargar texto o datos tabulares (consulte el [Capítulo 6](#) para obtener más información).

`np.savey` `np.load` son las dos funciones de caballo de batalla para guardar y cargar eficientemente los datos del conjunto en el disco. Las matrices se guardan de forma predeterminada en un formato binario sin comprimir con extensión `.npy`:

```
In [213]: arr = np.arange(10)
In [214]: np.save('some_array.npy', arr)
```

Si la ruta del archivo aún no termina en `.npy`, se *agregará* la extensión. La matriz en el disco se puede cargar con `np.load`:

```
In [215]: np.load('some_array.npy')
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Guarda múltiples matrices en un archivo sin comprimir usando `np.savez` y pasando las matrices como argumentos de palabras clave:

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

Al cargar un archivo `.npz`, recupera un objeto tipo dict que carga las matrices individuales de forma perezosa:

```
In [217]: arch = np.load('array_archive.npz')
In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Si sus datos se comprimen bien, puede desear utilizar `numpy.savez_compressed` en su lugar:

```
In [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```



4.5 Álgebra lineal

Álgebra lineal, como la multiplicación de matrices, descomposiciones, determinantes y otras matemáticas de matriz cuadrada, es una parte importante de cualquier biblioteca de matriz. A diferencia de algunos lenguajes como MATLAB, multiplicar dos matrices bidimensionales *es un producto basado en elementos en lugar de un producto de matriz de puntos. Por lo tanto, hay una función dot, tanto una matrizmétodo y una función en el numpyespacio de nombres, paramultiplicación matricial:

```
In [223]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
In [224]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [225]: x
Out[225]:
array([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]])

In [226]: y
Out[226]:
array([[ 6., 23.],
        [-1.,  7.],
        [ 8.,  9.]])

In [227]: x.dot(y)
Out[227]:
array([[ 28.,  64.],
        [ 67., 181.]])
```

x.dot(y)es equivalente a np.dot(x, y):

```
In [228]: np.dot(x, y)
Out[228]:
array([[ 28.,  64.],
        [ 67., 181.]])
```

Un producto matricial entre una matriz bidimensional y una matriz unidimensional de tamaño adecuado da como resultado una matriz unidimensional:

```
In [229]: np.dot(x, np.ones(3))
Out[229]: array([ 6., 15.] )
```

El @símbolo(a partir de Python 3.5) también funciona como un operador infijo que realiza la multiplicación de matrices:

```
In [230]: x @ np.ones(3)
Out[230]: array([ 6., 15.] )
```

numpy.linalgtiene un conjunto estándar de descomposiciones matriciales y cosas como inversa y determinante. Estos se implementan bajo el capó a través de las mismas bibliotecas de álgebra lineal estándar de la industria utilizadas en otros lenguajes como MATLAB y R, como BLAS, LAPACK o posiblemente (dependiendo de su compilación NumPy) el Intel patentado MKL (Biblioteca del kernel matemático):

```
In [231]: from numpy.linalg import inv, qr

In [232]: X = np.random.randn(5, 5)

In [233]: mat = X.T.dot(X)

In [234]: inv(mat)
Out[234]:
array([[ 933.1189,  871.8258, -1417.6902, -1460.4005, 1782.1391],
       [ 871.8258,  815.3929, -1325.9965, -1365.9242, 1666.9347],
       [-1417.6902, -1325.9965, 2158.4424, 2222.0191, -2711.6822],
       [-1460.4005, -1365.9242, 2222.0191, 2289.0575, -2793.422 ],
       [1782.1391, 1666.9347, -2711.6822, -2793.422 , 3409.5128]])

In [235]: mat.dot(inv(mat))
Out[235]:
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 [236]: q, r = qr(mat)

In [237]: r
Out[237]:
array([[ -1.6914,  4.38 ,  0.1757,  0.4075, -0.7838],
       [ 0. , -2.6436,  0.1939, -3.072 , -1.0702],
       [ 0. ,  0. , -0.8138,  1.5414,  0.6155],
       [ 0. ,  0. ,  0. , -2.6445, -2.1669],
       [ 0. ,  0. ,  0. ,  0. ,  0.0002]])
```

La expresión X.T.dot(X)calcula el producto punto de Xcon su transposición X.T.

Consulte la [Tabla 4-7](#) para obtener una lista de algunas de las funciones de álgebra lineal más utilizadas.



Tabla 4-7. Funciones `numpy.linalg` de uso común

Función	Descripción
<code>diag</code>	Regreso los elementos diagonales (o fuera de diagonal) de una matriz cuadrada como una matriz 1D, o convertir una matriz 1D en una matriz cuadrada con ceros en la diagonal
<code>dot</code>	Matriz multiplicación
<code>trace</code>	Calcular la suma de los elementos diagonales
<code>det</code>	Calcular el determinante de la matriz
<code>eig</code>	Calcular los valores propios y los vectores propios de una matriz cuadrada
<code>inv</code>	Calcular el inverso de una matriz cuadrada
<code>pinv</code>	Calcular el pseudo-inverso de Moore-Penrose de una matriz
<code>qr</code>	Calcular la descomposición QR
<code>svd</code>	Calcular la descomposición de valores singulares (SVD)
<code>solve</code>	Resuelve el sistema lineal $Ax = b$ para x , donde A es una matriz cuadrada
<code>lstsq</code>	Calcular la solución de mínimos cuadrados a $Ax = b$

4.6 Generación de números pseudoaleatorios

El `numpy.random` módulo complementa el Python `random` incorporado con funciones para generar de manera eficiente matrices enteras de valores de muestra a partir de muchos tipos de distribuciones de probabilidad. Por ejemplo, puede obtener una matriz 4×4 de muestras de la distribución normal estándar utilizando `normal`:

```
In [238]: samples = np.random.normal(size=(4, 4))

In [239]: samples
Out[239]:
array([[ 0.5732,  0.1933,  0.4429,  1.2796],
       [ 0.575 ,  0.4339, -0.7658, -1.237 ],
       [-0.5367,  1.8545, -0.92  , -0.1082],
       [ 0.1525,  0.9435, -1.0953, -0.144 ]])
```

El `random` módulo incorporado de Python, por el contrario, solo muestrea un valor a la vez. Como puede ver en este punto de referencia, `numpy.random` es mucho más rápido en un orden de magnitud para generar muestras muy grandes:

```
In [240]: from random import normalvariate

In [241]: N = 1000000

In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.54 s +- 81.9 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [243]: %timeit np.random.normal(size=N)
71.4 ms +- 8.53 ms per loop (mean +- std. dev. of 7 runs, 10 loops ea
```

Decimos que estos son números *pseudoaleatorios* porque son generados por un algoritmo con comportamiento determinista basado en la *semilla* del generador de números aleatorios. Puede cambiar la semilla de generación de números aleatorios de NumPy usando `np.random.seed`:

```
In [244]: np.random.seed(1234)
```

Las funciones de generación de datos `numpy.random` utilizan una semilla aleatoria global. Para evitar el estado global, se puede utilizar `numpy.random.RandomState` para crear un generador de números aleatorios aislado de otros:

```
In [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206,  0.8872,  0.8596,
       -0.6365,  0.0157, -2.2427])
```

Consulte la [Tabla 4-8](#) para obtener una lista parcial de las funciones disponibles en `numpy.random`. Daré algunos ejemplos de cómo aprovechar la capacidad de estas funciones para generar grandes matrices de muestras de una vez en la siguiente sección.



Tabla 4-8. Lista parcial de funciones numpy.random

Función	Descripción
seed	Sembrar el generador de números aleatorios
permutation	Devolver un permutación aleatoria de una secuencia, o devolver un rango permutado
shuffle	Al azar permutar una secuencia en el lugar
rand	Dibujar muestras de una distribución uniforme
randint	Dibujar enteros aleatorios de un rango bajo a alto dado
randn	Dibujar muestras de una distribución normal con media 0 y desviación estándar 1 (interfaz similar a MATLAB)
binomial	Dibujar muestras de una distribución binomial
normal	Dibujar muestras de una distribución normal (gaussiana)
beta	Dibujar muestras de una distribución beta
chisquare	Dibujar muestras de una distribución de chi-cuadrado
gamma	Dibujar muestras de una distribución gamma
uniform	Dibujar muestras de un distribución uniforme [0, 1)

4.7 Ejemplo: caminatas aleatorias

La simulación de caminatas aleatorias proporciona una aplicación ilustrativa de la utilización de operaciones de matriz. Primero consideremos una caminata aleatoria simple que comienza en 0 con pasos de 1 y -1 que ocurren con la misma probabilidad.

Aquí hay una forma pura de Python para implementar una única caminata aleatoria con 1,000 pasos usando el random módulo incorporado :

```
In [247]: 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)
.....:
```

Consulte la Figura 4-4 para ver un gráfico de ejemplo de los primeros 100 valores en una de estas caminatas aleatorias:

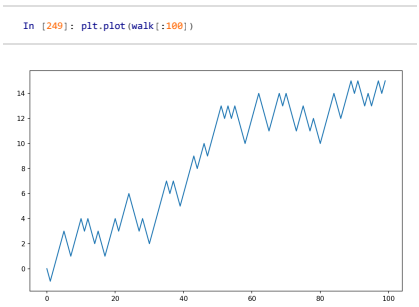


Figura 4-4. Una caminata aleatoria simple

Puede hacer la observación que `walk`es simplemente la suma acumulativa de los pasos aleatorios y podría evaluarse como una expresión de matriz. Por lo tanto, uso el `np.random`módulo para dibujar 1,000 lanzamientos de monedas a la vez, establecerlos en 1 y -1, y calcular la suma acumulativa:

```
In [251]: nsteps = 1000
In [252]: draws = np.random.randint(0, 2, size=nsteps)
In [253]: steps = np.where(draws > 0, 1, -1)
In [254]: walk = steps.cumsum()
```

De esto podemos comenzar a extraer estadísticas como el valor mínimo y máximo a lo largo de la trayectoria de la caminata:

```
In [255]: walk.min()
Out[255]: -3

In [256]: walk.max()
Out[256]: 31
```

Una estadística más complicada es el *primer tiempo de cruce* , el paso en el que la caminata aleatoria alcanza un valor particular. Aquí podríamos querer saber cuánto tiempo tomó la caminata aleatoria para obtener al menos 10 pasos del origen 0 en cualquiera de los dosdirección. `np.abs(walk) >= 10`nos da una matriz booleana que indica dónde la caminata ha alcanzado o excedido 10, pero queremos el índice de los *primeros* 10 o -10. Resulta que podemos calcular esto using `argmax`, que devuelve el primer índice del valor máximo en la matriz booleana (`True`es el valor máximo):



```
In [257]: (np.abs(walk) >= 10).argmax()
Out[257]: 37
```

Tenga en cuenta que usar `argmax` aquí no siempre es eficiente porque siempre realiza un escaneo completo de la matriz. En este caso especial, una vez que `True` se observa un, sabemos que es el valor máximo.

Simulando muchos paseos aleatorios a la vez

Si su objetivo era simular muchas caminatas aleatorias, digamos 5,000 de ellas, puede generar todas las caminatas aleatorias con modificaciones menores al código anterior. Si se pasa una tupla de 2, las `numpy.random` funciones generarán una matriz bidimensional de sorteos, y podemos calcular la suma acumulativa a través de las filas para calcular las 5,000 caminatas aleatorias en una sola toma:

```
In [258]: nwalks = 5000

In [259]: nsteps = 1000

In [260]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 o 1

In [261]: steps = np.where(draws > 0, 1, -1)

In [262]: walks = steps.cumsum(1)

In [263]: walks
Out[263]:
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]])
```

Ahora, podemos calcular los valores máximos y mínimos obtenidos en todas las caminatas:

```
In [264]: walks.max()
Out[264]: 138

In [265]: walks.min()
Out[265]: -133
```

Fuera de estas caminatas, calculemos el tiempo mínimo de cruce a 30 o -30. Esto es un poco complicado porque no todos los 5,000 alcanzan 30. Podemos verificar esto usando el `any` método:

```
In [266]: hits30 = (np.abs(walks) >= 30).any(1)

In [267]: hits30
Out[267]: array([False,  True, False, ..., False,  True, False], dtype=bool)

In [268]: hits30.sum() # Number that hit 30 or -30
Out[268]: 3410
```

Podemos usar esta matriz booleana para seleccionar las filas `walks` que realmente cruzan el nivel absoluto de 30 y llamar a `argmax` a través del eje 1 para obtener los tiempos de cruce:

```
In [269]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [270]: crossing_times.mean()
Out[270]: 498.88973607038122
```

Siéntase libre de experimentar con otras distribuciones para los pasos que no sean lanzamientos de monedas del mismo tamaño. Solo necesita usar una función de generación de números aleatorios diferente, como `normal` para generar pasos distribuidos normalmente con alguna media y estándar desviación:

```
In [271]: steps = np.random.normal(loc=0, scale=0.25,
.....:                             size=(nwalks, nsteps))
```

4.8 Conclusión

Si bien gran parte del resto del libro se enfocará en desarrollar habilidades de discusión de datos con `pandas`, continuaremos trabajando en un estilo similar basado en matrices. En el [Apéndice A](#), profundizaremos en las características de `NumPy` para ayudarlo a desarrollar aún más sus habilidades de computación en matriz.

