

# NumPy基础：数组和矢量计算

NumPy（Numerical Python的简称）是高性能科学计算和数据分析的基础包。它是本书所介绍的几乎所有高级工具的构建基础。其部分功能如下：

- `ndarray`，一个具有矢量算术运算和复杂广播能力的快速且节省空间的多维数组。
- 用于对整组数据进行快速运算的标准数学函数（无需编写循环）。
- 用于读写磁盘数据的工具以及用于操作内存映射文件的工具。
- 线性代数、随机数生成以及傅里叶变换功能。
- 用于集成由C、C++、Fortran等语言编写的代码的工具。

最后一点也是从生态系统角度来看最重要的一点。由于NumPy提供了一个简单易用的C API，因此很容易将数据传递给由低级语言编写的外部库，外部库也能以NumPy数组的形式将数据返回给Python。这个功能使Python成为一种包装C/C++/Fortran历史代码库的选择，并使被包装库拥有一个动态的、易用的接口。

NumPy本身并没有提供多么高级的数据分析功能，理解NumPy数组以及面向数组的计算将有助于你更加高效地使用诸如pandas之类的工具。如果你是Python新手，而且只是想用pandas随便处理一下数据就行，那就跳过本章吧，没关系的。更多NumPy高级功能（比如广播），请参见第12章。

对于大部分数据分析应用而言，我最关注的功能主要集中在：

- 用于数据整理和清理、子集构造和过滤、转换等快速的矢量化数组运算。
- 常用的数组算法，如排序、唯一化、集合运算等。



- 高效的描述统计和数据聚合/摘要运算。
- 用于异构数据集的合并/连接运算的数据对齐和关系型数据运算。
- 将条件逻辑表述为数组表达式（而不是带有if-elif-else分支的循环）。
- 数据的分组运算（聚合、转换、函数应用等）。第5章将对此进行详细讲解。

虽然NumPy提供了这些功能的计算基础，但你可能还是想将pandas作为数据分析工作的基础（尤其是对于结构化或表格化数据），因为它提供了能使大部分常见数据任务变得非常简洁的丰富高级接口。pandas还提供了一些NumPy所没有的更加领域特定的功能，如时间序列处理等。

---

**注意：**在本章以及本书中，我将依照标准的NumPy约定，即总是使用`import numpy as np`。当然，你也可以为了不写`np.`而直接在代码中使用`from numpy import *`，但我得提醒你最好还是不要养成这样的坏习惯。

---

## NumPy的ndarray：一种多维数组对象

NumPy最重要的一个特点就是其N维数组对象（即`ndarray`），该对象是一个快速而灵活的大数据集容器。你可以利用这种数组对整块数据执行一些数学运算，其语法跟标量元素之间的运算一样：

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

In [9]: data * 10
Out[9]:
array([[ 9.5256, -2.4601, -8.8565],
       [ 5.6385,  2.3794,  9.104 ]])

In [10]: data + data
Out[10]:
array([[ 1.9051, -0.492 , -1.7713],
       [ 1.1277,  0.4759,  1.8208]])
```

`ndarray`是一个通用的同构数据多维容器，也就是说，其中的所有元素必须是相同类型的。每个数组都有一个`shape`（一个表示各维度大小的元组）和一个`dtype`（一个用于说明数组数据类型的对象）：

```
In [11]: data.shape
Out[11]: (2, 3)
In [12]: data.dtype
Out[12]: dtype('float64')
```

本章将会介绍NumPy数组的基本用法，这对于本书后面各章的理解基本够用。虽然大多数数据分析工作不需要深入理解NumPy，但是精通面向数组的编程和思维方式是成为Python科学计算牛人的一大关键步骤。



---

注意：当你在本书中看到“数组”、“NumPy数组”、“ndarray”时，基本上都指的是同一样东西，即 ndarray 对象。

---

## 创建 ndarray

创建数组最简单的办法就是使用 `array` 函数。它接受一切序列型的对象（包括其他数组），然后产生一个新的含有传入数据的 NumPy 数组。以一个列表的转换为例：

```
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.])
```

嵌套序列（比如由一组等长列表组成的列表）将会被转换为一个多维数组：

```
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)
```

除非显式说明（稍后将会详细介绍），`np.array` 会尝试为新建的这个数组推断出一个较为合适的数据类型。数据类型保存在一个特殊的 `dtype` 对象中。比如说，在上面的两个例子中，我们有：

```
In [21]: arr1.dtype  
Out[21]: dtype('float64')  
In [22]: arr2.dtype  
Out[22]: dtype('int64')
```

除 `np.array` 之外，还有一些函数也可以新建数组。比如，`zeros` 和 `ones` 分别可以创建指定长度或形状的全 0 或全 1 数组。`empty` 可以创建一个没有任何具体值的数组。要用这些方法创建多维数组，只需传入一个表示形状的元组即可：

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

---

**警告：**认为`np.empty`会返回全0数组的想法是不安全的。很多情况下（如前所示），它返回的都是一些未初始化的垃圾值。

---

`arange`是Python内置函数`range`的数组版：

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

表4-1列出了一些数组创建函数。由于NumPy关注的是数值计算，因此，如果没有特别指定，数据类型基本都是`float64`（浮点数）。

表4-1：数组创建函数

函数	说明
<code>array</code>	将输入数据（列表、元组、数组或其他序列类型）转换为 <code>ndarray</code> 。要么推断出 <code>dtype</code> ，要么显式指定 <code>dtype</code> 。默认直接复制输入数据
<code>asarray</code>	将输入转换为 <code>ndarray</code> ，如果输入本身就是一个 <code>ndarray</code> 就不进行复制
<code>arange</code>	类似于内置的 <code>range</code> ，但返回的是一个 <code>ndarray</code> 而不是列表
<code>ones</code> 、 <code>ones_like</code>	根据指定的形状和 <code>dtype</code> 创建一个全1数组。 <code>ones_like</code> 以另一个数组为参数，并根据其形状和 <code>dtype</code> 创建一个全1数组
<code>zeros</code> 、 <code>zeros_like</code>	类似于 <code>ones</code> 和 <code>ones_like</code> ，只不过产生的是全0数组而已
<code>empty</code> 、 <code>empty_like</code>	创建新数组，只分配内存空间但不填充任何值
<code>eye</code> 、 <code>identity</code>	创建一个正方的 $N \times N$ 单位矩阵（对角线为1，其余为0）



## ndarray的数据类型

`dtype`（数据类型）是一个特殊的对象，它含有`ndarray`将一块内存解释为特定数据类型所需的信息：

```
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
```

```
Out[29]: dtype('float64')
```

```
In [30]: arr2.dtype
```

```
Out[30]: dtype('int32')
```

`dtype`是NumPy如此强大和灵活的原因之一。多数情况下，它们直接映射到相应的机器表示，这使得“读写磁盘上的二进制数据流”以及“集成低级语言代码（如C、Fortran）”等工作变得更加简单。数值型`dtype`的命名方式相同：一个类型名（如`float`或`int`），后面跟一个用于表示各元素位长的数字。标准的双精度浮点值（即Python中的`float`对象）需要占用8字节（即64位）。因此，该类型在NumPy中就记作`float64`。表4-2列出了NumPy所支持的全部数据类型。

---

注意：记不住这些NumPy的`dtype`也没关系，新手更是如此。通常只需要知道你所处理的数据的大致类型是浮点数、复数、整数、布尔值、字符串，还是普通的Python对象即可。当你需要控制数据在内存和磁盘中的存储方式时（尤其是对大数据集），那就得了解如何控制存储类型。

---

表4-2：NumPy的数据类型

类型	类型代码	说明
<code>int8</code> 、 <code>uint8</code>	<code>i1</code> 、 <code>u1</code>	有符号和无符号的8位（1个字节）整型
<code>int16</code> 、 <code>uint16</code>	<code>i2</code> 、 <code>u2</code>	有符号和无符号的16位（2个字节）整型
<code>int32</code> 、 <code>uint32</code>	<code>i4</code> 、 <code>u4</code>	有符号和无符号的32位（4个字节）整型
<code>int64</code> 、 <code>uint64</code>	<code>i8</code> 、 <code>u8</code>	有符号和无符号的64位（8个字节）整型
<code>float16</code>	<code>f2</code>	半精度浮点数
<code>float32</code>	<code>f4</code> 或 <code>f</code>	标准的单精度浮点数。与C的 <code>float</code> 兼容
<code>float64</code>	<code>f8</code> 或 <code>d</code>	标准的双精度浮点数。与C的 <code>double</code> 和Python的 <code>float</code> 对象兼容
<code>float128</code>	<code>f16</code> 或 <code>g</code>	扩展精度浮点数
<code>complex64</code> 、 <code>complex128</code> 、 <code>complex256</code>	<code>c8</code> 、 <code>c16</code> 、 <code>c32</code>	分别用两个32位、64位或128位浮点数表示的复数
<code>bool</code>	?	存储 <code>True</code> 和 <code>False</code> 值的布尔类型



表4-2：NumPy的数据类型（续）

类型	类型代码	说明
object	O	Python对象类型
string_	S	固定长度的字符串类型（每个字符1个字节）。例如，要创建一个长度为10的字符串，应使用S10
unicode_	U	固定长度的unicode类型（字节数由平台决定）。跟字符串的定义方式一样（如U10）

你可以通过ndarray的astype方法显式地转换其dtype：

```
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')
```

在本例中，整数被转换成了浮点数。如果将浮点数转换成整数，则小数部分将会被截断：

```
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)
```

如果某字符串数组表示的全是数字，也可以用astype将其转换为数值形式：

```
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. ])
```

如果转换过程因为某种原因而失败了（比如某个不能被转换为float64的字符串），就会引发一个TypeError。看到了吧，我比较懒，写的是float而不是np.float64；NumPy很聪明，它会将Python类型映射到等价的dtype上。

数组的dtype还有另外一个用法：

```
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.])
```

你还可以用简洁的类型代码来表示dtype:

```
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)
```

---

注意：调用astype无论如何都会创建出一个新的数组（原始数据的一份拷贝），即使新dtype跟老dtype相同也是如此。

---

警告：注意，浮点数（比如float64和float32）只能表示近似的分数值。在复杂计算中，由于可能会积累一些浮点错误，因此比较操作只能在一定小数位以内有效。

---

## 数组和标量之间的运算

数组很重要，因为它使你不用编写循环即可对数据执行批量运算。这通常就叫做矢量化（vectorization）。大小相等的数组之间的任何算术运算都会将运算应用到元素级：

```
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.]])
```

同样，数组与标量的算术运算也会将那个标量值传播到各个元素：

```
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]])
```

不同大小的数组之间的运算叫做广播（broadcasting），我们将在第12章中对其进行详细讨论。本书的内容不需要对广播机制有多深的理解。

---



## 基本的索引和切片

NumPy数组的索引是一个内容丰富的主题，因为选取数据子集或单个元素的方式有很多。一维数组很简单。从表面上看，它们跟Python列表的功能差不多：

```
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])
```

如上所示，当你将一个标量值赋值给一个切片时（如`arr[5:8] = 12`），该值会自动传播（也就说后面将会讲到的“广播”）到整个选区。跟列表最重要的区别在于，数组切片是原始数组的视图。这意味着数据不会被复制，视图上的任何修改都会直接反映到源数组上：

```
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])
```

如果你刚开始接触NumPy，可能会对此感到惊讶（尤其是当你曾经用过其他热衷于复制数组数据的编程语言）。由于NumPy的设计目的是处理大数据，所以你可以想象一下，假如NumPy坚持要将数据复制来复制去的话会产生何等的性能和内存问题。

---

**警告：**如果你想要得到的是ndarray切片的一份副本而非视图，就需要显式地进行复制操作，例如`arr[5:8].copy()`。

---

对于高维度数组，能做的事情更多。在一个二维数组中，各索引位置上的元素不再是标量而是一维数组：



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

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

因此，可以对各个元素进行递归访问，但这样需要做的事情有点多。你可以传入一个以逗号隔开的索引列表来选取单个元素。也就是说，下面两种方式是等价的：

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

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

图4-1说明了二维数组的索引方式。

		axis 1			
		0	1	2	
axis 0		0	0,0	0,1	0,2
		1	1,0	1,1	1,2
		2	2,0	2,1	2,2

图4-1：NumPy数组中的元素索引

在多维数组中，如果省略了后面的索引，则返回对象会是一个维度低一点的ndarray（它含有高一级维度上的所有数据<sup>译注1</sup>）。因此，在 $2 \times 2 \times 3$ 数组arr3d中：

```
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]])
```

译注1：括号外面的“维度”是一维、二维、三维、四维之类的意思，而括号里面的应该理解为“轴”。也就是说，这里指的是“返回的低维数组含有原始高维数组某条轴上的所有数据”。



`arr3d[0]`是一个 $2 \times 3$ 数组：

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

标量值和数组都可以被赋值给`arr3d[0]`：

```
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]]])
```

以此类推，`arr3d[1, 0]`可以访问索引以(1, 0)开头的那些值（以一维数组的形式返回）：

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

注意，在上面所有这些选取数组子集的例子中，返回的数组都是视图。

## 切片索引

`ndarray`的切片语法跟Python列表这样的一维对象差不多：

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

高维度对象的花样更多，你可以在一个或多个轴上进行切片，也可以跟整数索引混合使用。对于上面那个二维数组`arr2d`，其切片方式稍显不同：

<pre>In [76]: arr2d Out[76]: array([[1, 2, 3],        [4, 5, 6],        [7, 8, 9]])</pre>	<pre>In [77]: arr2d[:2] Out[77]: array([[1, 2, 3],        [4, 5, 6]])</pre>
---	---



可以看出，它是沿着第0轴（即第一个轴）切片的。也就是说，切片是沿着一个轴向选取元素的。你可以一次传入多个切片，就像传入多个索引那样：

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

像这样进行切片时，只能得到相同维数的数组视图。通过将整数索引和切片混合，可以得到低维度的切片：

```
In [79]: arr2d[1, :2]           In [80]: arr2d[2, :1]  
Out[79]: array([4, 5])         Out[80]: array([7])
```

图4-2对此进行了说明。注意，“只有冒号”表示选取整个轴，因此你可以像下面这样只对高维轴进行切片：

```
In [81]: arr2d[:, :1]  
Out[81]:  
array([[1],  
      [4],  
      [7]])
```

自然，对切片表达式的赋值操作也会被扩散到整个选区：

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

## 布尔型索引

来看这样一个例子，假设我们有一个用于存储数据的数组以及一个存储姓名的数组（含有重复项）。在这里，我将使用numpy.random中的randn函数生成一些正态分布的随机数据：

```
In [83]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])  
In [84]: data = 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],  
      [ 0.4412,  0.148 , -0.8448,  0.144 ]],  
      dtype='float64')
```



```
[ 0.5994,  0.8174, -0.9297, -1.2564]])
```

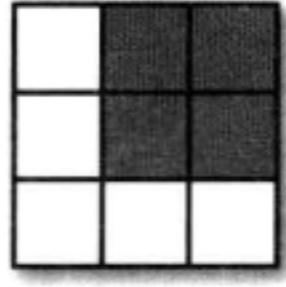
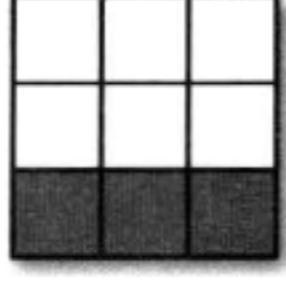
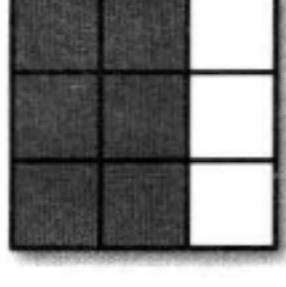
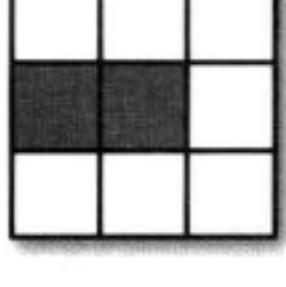
Expression	Shape
	<code>arr[::2, ::1]</code> (2, 2)
	<code>arr[2]</code> (3,) <code>arr[2, ::]</code> (3,) <code>arr[2:, ::]</code> (1, 3)
	<code>arr[:, ::2]</code> (3, 2)
	<code>arr[1, ::2]</code> (2,) <code>arr[1:2, ::2]</code> (1, 2)

图4-2：二维数组切片

假设每个名字都对应data数组中的一行，而我们想要选出对应于名字“Bob”的所有行。跟算术运算一样，数组的比较运算（如`==`）也是矢量化的。因此，对names和字符串“Bob”的比较运算将会产生一个布尔型数组：

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

这个布尔型数组可用于数组索引：

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

布尔型数组的长度必须跟被索引的轴长度一致。此外，还可以将布尔型数组跟切片、整数（或整数序列，稍后将对此进行详细讲解）混合使用：

```
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])
```

要选择除“Bob”以外的其他值，既可以使用不等于符号（!=），也可以通过负号（-）对条件进行否定：

```
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]])
```

选取这三个名字中的两个需要组合应用多个布尔条件，使用&（和）、|（或）之类的布尔算术运算符即可：

```
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]])
```

通过布尔型索引选取数组中的数据，将总是创建数据的副本，即使返回一模一样的数组也是如此。

---

**警告：** Python关键字and和or在布尔型数组中无效。

---

通过布尔型数组设置值是一种经常用到的手段。为了将data中的所有负值都设置为0，我们只需：

```
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.      ,  0.8174,  0.9297, -1.2564],  
      [ 0.      ,  0.8799,  0.1698, -0.1002],  
      [ 0.      ,  1.1503,  1.7289,  0.3103],  
      [ 0.      ,  2.1452, -0.2349,  0.047 ]])
```



```
[ 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.      ]])
```

通过一维布尔数组设置整行或列的值也很简单：

```
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.      ]])
```

## 花式索引

花式索引（Fancy indexing）是一个NumPy术语，它指的是利用整数数组进行索引。假设我们有一个 $8 \times 4$ 数组：

```
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.]])
```

为了以特定顺序选取行子集，只需传入一个用于指定顺序的整数列表或ndarray即可：

```
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.]])
```

这段代码确实达到我们的要求了！使用负数索引将会从末尾开始选取行：



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

一次传入多个索引数组会有一点特别。它返回的是一个一维数组，其中的元素对应各个索引元组：

```
# 有关reshape的知识将在第12章中讲解  
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])
```

我们来看看具体是怎么一回事。最终选出的是元素(1, 0)、(5, 3)、(7, 1)和(2, 2)。这个花式索引的行为可能会跟某些用户的预期不一样（包括我在内），选取矩阵的行列子集应该是矩形区域的形式才对。下面是得到该结果的一个办法：

```
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]])
```

另外一个办法是使用np.ix\_函数，它可以将两个一维整数数组转换为一个用于选取方形区域的索引器：

```
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]])
```

记住，花式索引跟切片不一样，它总是将数据复制到新数组中。



## 数组转置和轴对换

转置（transpose）是重塑的一种特殊形式，它返回的是源数据的视图（不会进行任何复制操作）。数组不仅有transpose方法，还有一个特殊的T属性：

```
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]])
```

在进行矩阵计算时，经常需要用到该操作，比如利用np.dot计算矩阵内积 $X^T X$ ：

```
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]])
```

对于高维数组，transpose需要得到一个由轴编号组成的元组才能对这些轴进行转置（比较费脑子）：

```
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]]])
```

简单的转置可以使用.T，它其实就是进行轴对换而已。ndarray还有一个swapaxes方法，它需要接受一对轴编号：



```
In [118]: arr
Out[118]:
array([[[ 0,  1,  2,  3],
       [ 4,  5,  6,  7]],

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

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

swapaxes也是返回源数据的视图（不会进行任何复制操作）。

## 通用函数：快速的元素级数组函数

通用函数（即ufunc）是一种对ndarray中的数据执行元素级运算的函数。你可以将其看做简单函数（接受一个或多个标量值，并产生一个或多个标量值）的矢量化包装器。

许多ufunc都是简单的元素级变体，如sqrt和exp：

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

In [121]: np.sqrt(arr)
Out[121]:
array([ 0.        ,  1.        ,  1.41421356,  1.73205081,
       2.        ,  2.23606798,  2.44948974,
       2.64575131,  2.82842712,  3.        ])

In [122]: np.exp(arr)
Out[122]:
array([ 1.        ,          2.71828183,          7.38905619,
       20.08553692,         54.5982118, 148.41320412,
      403.42881394, 1096.63324329, 2980.95804645,
     8103.08390271])
```

这些都是一元（unary）ufunc。另外一些（如add或maximum）接受2个数组（因此也叫二元（binary）ufunc），并返回一个结果数组：

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

In [124]: y = 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) # 元素级最大值  
Out[127]:  
array([ 0.267 ,  0.0974,  0.2002,  0.6117,  0.4655,  0.9222,  0.446 , -0.7147])
```

虽然并不常见，但有些ufunc的确可以返回多个数组。`modf`就是一个例子，它是Python内置函数`divmod`的矢量化版本，用于浮点数数组的小数和整数部分。

```
In [128]: arr = 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.]))
```

表4-3和表4-4分别列出了一些一元和二元ufunc。

表4-3：一元ufunc

函数	说明
abs、fabs	计算整数、浮点数或复数的绝对值。对于非复数值，可以使用更快的fabs
sqrt	计算各元素的平方根。相当于 <code>arr ** 0.5</code>
square	计算各元素的平方。相当于 <code>arr ** 2</code>
exp	计算各元素的指数 $e^x$
log、log10、log2、log1p	分别为自然对数（底数为e）、底数为10的log、底数为2的log、 $\log(1 + x)$
sign	计算各元素的正负号：1（正数）、0（零）、-1（负数）
ceil	计算各元素的ceiling值，即大于等于该值的最小整数
floor	计算各元素的floor值，即小于等于该值的最大整数
rint	将各元素值四舍五入到最接近的整数，保留 <code>dtype</code>
modf	将数组的小数和整数部分以两个独立数组的形式返回
isnan	返回一个表示“哪些值是NaN（这不是一个数字）”的布尔型数组
isfinite、isinf	分别返回一个表示“哪些元素是有穷的（非inf，非NaN）”或“哪些元素是无穷的”的布尔型数组
cos、cosh、sin、sinh、tan、tanh	普通型和双曲型三角函数



表4-3：一元ufunc（续）

函数	说明
arccos、arccosh、arcsin、arcsinh、arctan、arctanh	反三角函数
logical_not	计算各元素not x的真值。相当于-not

表4-4：二元ufunc

函数	说明
add	将数组中对应的元素相加
subtract	从第一个数组中减去第二个数组中的元素
multiply	数组元素相乘
divide、floor_divide	除法或向下圆整除法（丢弃余数）
power	对第一个数组中的元素A，根据第二个数组中的相应元素B，计算 $A^B$
maximum、fmax	元素级的最大值计算。fmax将忽略NaN
minimum、fmin	元素级的最小值计算。fmin将忽略NaN
mod	元素级的求模计算（除法的余数）
copysign	将第二个数组中的值的符号复制给第一个数组中的值
greater、greater_equal、less、less_equal、equal、not_equal	执行元素级的比较运算，最终产生布尔型数组。相当于中缀运算符>、>=、<、<=、==、!=
logical_and、logical_or、logical_xor	执行元素级的真值逻辑运算。相当于中缀运算符&、 、^

## 利用数组进行数据处理

NumPy数组使你可以将许多种数据处理任务表述为简洁的数组表达式（否则需要编写循环）。用数组表达式代替循环的做法，通常被称为矢量化。一般来说，矢量化数组运算要比等价的纯Python方式快上一两个数量级（甚至更多），尤其是各种数值计算。在后面内容中（见第12章）我将介绍广播，这是一种针对矢量化计算的强大手段。

假设我们想要在一组值（网格型）上计算函数 $\sqrt{x^2 + y^2}$ 。`np.meshgrid`函数接受两个一维数组，并产生两个二维矩阵（对应于两个数组中所有的(x, y)对）：

```
In [130]: points = np.arange(-5, 5, 0.01) # 1000个间隔相等的点
```



```
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]])
```

现在，对该函数的求值运算就好办了，把这两个数组当做两个浮点数那样编写表达式即可：

```
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>
```

函数值（一个二维数组）的图形化结果如图4-3所示。这张图我是用matplotlib的imshow函数创建的。

## 将条件逻辑表述为数组运算

numpy.where函数是三元表达式`x if condition else y`的矢量化版本。假设我们有一个布尔数组和两个值数组：

```
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])
```



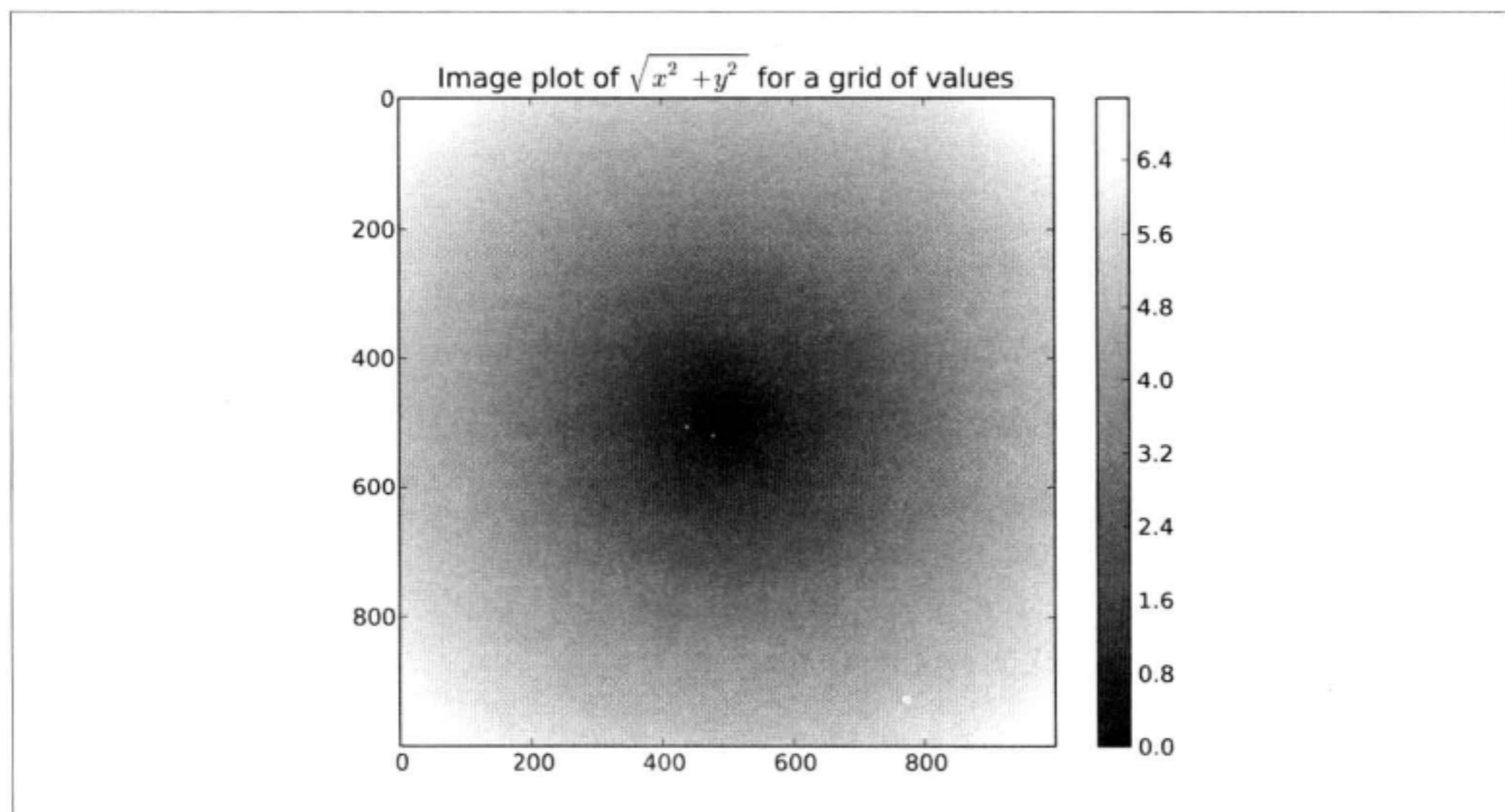


图4-3：根据网格对函数求值的结果

假设我们想要根据cond中的值选取xarr和yarr的值：当cond中的值为True时，选取xarr的值，否则从yarr中选取。列表推导式的写法应该如下所示：

```
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, 1.399999999999999, 2.5]
```

这有几个问题。第一，它对大数组的处理速度不是很快（因为所有工作都是由纯Python完成的）。第二，无法用于多维数组。若使用np.where，则可以将该功能写得非常简洁：

```
In [145]: result = np.where(cond, xarr, yarr)
In [146]: result
Out[146]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

np.where的第二个和第三个参数不必是数组，它们都可以是标量值。在数据分析工作中，where通常用于根据另一个数组而产生一个新的数组。假设有一个由随机数据组成的矩阵，你希望将所有正值替换为2，将所有负值替换为-2。若利用np.where，则会非常简单：

```
In [147]: arr = 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) # 只将正值设置为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]])
```

传递给where的数组大小可以不相等，甚至可以是标量值。

只要稍微动动脑子，你就能用where表述出更复杂的逻辑。想象一下这样一个例子，我有两个布尔型数组cond1和cond2，希望根据4种不同的布尔值组合实现不同的赋值操作：

```
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)
```

虽然不是很明显，但这个for循环确实可以被改写成一个嵌套的where表达式：

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

在这个特殊的例子中，我们还可以利用“布尔值在计算过程中可以被当做0或1处理”这个事实，所以还能将其写成下面这样的算术运算（虽然看上去有点神秘）：

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

## 数学和统计方法

可以通过数组上的一组数学函数对整个数组或某个轴向的数据进行统计计算。sum、



`mean`以及标准差`std`等聚合计算（aggregation，通常叫做约简（reduction））既可以当做数组的实例方法调用，也可以当做顶级NumPy函数使用：

```
In [151]: arr = np.random.randn(5, 4) # 正态分布的数据
```

```
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
```

`mean`和`sum`这类的函数可以接受一个`axis`参数（用于计算该轴向上的统计值），最终结果是一个少一维的数组：

```
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])
```

其他如`cumsum`和`cumprod`之类的方法则不聚合，而是产生一个由中间结果组成的数组：

```
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]])
```

表4-5列出了全部的基本数组统计方法。后续章节中有很多例子都会用到这些方法。

表4-5：基本数组统计方法

方法	说明
<code>sum</code>	对数组中全部或某轴向的元素求和。零长度的数组的 <code>sum</code> 为0
<code>mean</code>	算术平均数。零长度的数组的 <code>mean</code> 为NaN
<code>std</code> 、 <code>var</code>	分别为标准差和方差，自由度可调（默认为n）
<code>min</code> 、 <code>max</code>	最大值和最小值
<code>argmin</code> 、 <code>argmax</code>	分别为最大和最小元素的索引



表4-5：基本数组统计方法（续）

方法	说明
cumsum	所有元素的累计和
cumprod	所有元素的累计积

## 用于布尔型数组的方法

在上面这些方法中，布尔值会被强制转换为1 (`True`) 和0 (`False`)。因此，`sum`经常被用来对布尔型数组中的`True`值计数：

```
In [160]: arr = randn(100)  
In [161]: (arr > 0).sum() # 正值的数量  
Out[161]: 44
```

另外还有两个方法`any`和`all`，它们对布尔型数组非常有用。`any`用于测试数组中是否存在一个或多个`True`，而`all`则检查数组中所有值是否都是`True`：

```
In [162]: bools = np.array([False, False, True, False])  
In [163]: bools.any()  
Out[163]: True  
  
In [164]: bools.all()  
Out[164]: False
```

这两个方法也能用于非布尔型数组，所有非0元素将会被当做`True`。

## 排序

跟Python内置的列表类型一样，NumPy数组也可以通过`sort`方法就地排序：

```
In [165]: arr = 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])
```

多维数组可以在任何一个轴向上进行排序，只需将轴编号传给`sort`即可：

```
In [169]: arr = 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]])
```

顶级方法`np.sort`返回的是数组的已排序副本，而就地排序则会修改数组本身。计算数组分位数最简单的办法是对其进行排序，然后选取特定位置的值：

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

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

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

更多关于NumPy排序方法以及诸如间接排序之类的高级技术，请参阅第12章。在pandas中还可以找到一些其他跟排序有关的数据操作（比如根据一列或多列对表格型数据进行排序）。

## 唯一化以及其他集合逻辑

NumPy提供了一些针对一维ndarray的基本集合运算。最常用的可能要数`np.unique`了，它用于找出数组中的唯一值并返回已排序的结果：

```
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])
```

拿跟`np.unique`等价的纯Python代码来对比一下：



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

另一个函数`np.in1d`用于测试一个数组中的值在另一个数组中的成员资格，返回一个布尔型数组：

```
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)
```

NumPy中的集合函数请参见表4-6。

表4-6：数组的集合运算

方法	说明
<code>unique(x)</code>	计算x中的唯一元素，并返回有序结果
<code>intersect1d(x, y)</code>	计算x和y中的公共元素，并返回有序结果
<code>union1d(x, y)</code>	计算x和y的并集，并返回有序结果
<code>in1d(x, y)</code>	得到一个表示“x的元素是否包含于y”的布尔型数组
<code>setdiff1d(x, y)</code>	集合的差，即元素在x中且不在y中
<code>setxor1d(x, y)</code>	集合的对称差，即存在于一个数组中但不同时存在于两个数组中的元素 <sup>译注2</sup>

## 用于数组的文件输入输出

NumPy能够读写磁盘上的文本数据或二进制数据。后面的章节将会告诉你一些pandas中用于将表格型数据读取到内存的工具。

## 将数组以二进制格式保存到磁盘

`np.save`和`np.load`是读写磁盘数组数据的两个主要函数。默认情况下，数组是以未压缩的原始二进制格式保存在扩展名为`.npy`的文件中的。

```
In [183]: arr = np.arange(10)
In [184]: np.save('some_array', arr)
```

如果文件路径末尾没有扩展名`.npy`，则该扩展名会被自动加上。然后就可以通过`np.load`读取磁盘上的数组：

译注2：简单点说，就是“异或”。



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

通过`np.savez`可以将多个数组保存到一个压缩文件中，将数组以关键字参数的形式传入即可：

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

加载`.npz`文件时，你会得到一个类似字典的对象，该对象会对各个数组进行延迟加载：

```
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])
```

## 存取文本文件

从文件中加载文本是一个非常标准的任务。Python中的文件读写函数的格式很容易将新手搞晕，所以我将主要介绍pandas中的`read_csv`和`read_table`函数。有时，我们需要用`np.loadtxt`或更为专门化的`np.genfromtxt`将数据加载到普通的NumPy数组中。

这些函数都有许多选项可供使用：指定各种分隔符、针对特定列的转换器函数、需要跳过的行数等。以一个简单的逗号分隔文件（CSV）为例：

```
In [191]: !cat array_ex.txt译注3
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
```

该文件可以被加载到一个二维数组中，如下所示：

```
In [192]: arr = np.loadtxt('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`执行的是相反的操作：将数组写到以某种分隔符隔开的文本文件中。

---

译注3：这是Linux的，Windows得用`type`。



`genfromtxt`跟`loadtxt`差不多，只不过它面向的是结构化数组和缺失数据处理。更多有关结构化数组的知识，请参阅第12章。

---

注意：更多有关文件读写（尤其是表格型数据）的知识，请参阅本书后面有关pandas和DataFrame对象的章节。

---

## 线性代数

线性代数（如矩阵乘法、矩阵分解、行列式以及其他方阵数学等）是任何数组库的重要组成部分。不像某些语言（如MATLAB），通过`*`对两个二维数组相乘得到的是一个元素级的积，而不是一个矩阵点积。因此，NumPy提供了一个用于矩阵乘法的`dot`函数（既是一个数组方法也是numpy命名空间中的一个函数）：

```
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
```

```
Out[196]:
```

```
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

```
In [197]: y
```

```
Out[197]:
```

```
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])
```

```
In [198]: x.dot(y) # 相当于np.dot(x, y)
```

```
Out[198]:
```

```
array([[ 28.,  64.],
       [ 67., 181.]])
```

一个二维数组跟一个大小合适的一维数组的矩阵点积运算之后将会得到一个一维数组：

```
In [199]: np.dot(x, np.ones(3))
```

```
Out[199]: array([ 6., 15.])
```

`numpy.linalg`中有一组标准的矩阵分解运算以及诸如求逆和行列式之类的东西。它们跟MATLAB和R等语言所使用的是相同的行业标准级Fortran库，如BLAS、LAPACK、Intel MKL（可能有，取决于你的NumPy版本）等：

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

```
In [202]: X = 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]])
```

表4-7中列出了一些最常用的线性代数函数。

---

注意： Python科学计算社区盼望着有朝一日能实现矩阵乘法的中缀运算符，以便能用一种更漂亮的语法代替np.dot。不过目前就只能先这样了。

---

表4-7：常用的numpy.linalg函数

函数	说明
diag	以一维数组的形式返回方阵的对角线（或非对角线）元素，或将一维数组转换为方阵（非对角线元素为0）
dot	矩阵乘法
trace	计算对角线元素的和
det	计算矩阵行列式
eig	计算方阵的本征值和本征向量
inv	计算方阵的逆
pinv	计算矩阵的Moore-Penrose伪逆
qr	计算QR分解
svd	计算奇异值分解（SVD）
solve	解线性方程组 $Ax = b$ ，其中A为一个方阵
lstsq	计算 $Ax = b$ 的最小二乘解



# 随机数生成

`numpy.random`模块对Python内置的`random`进行了补充，增加了一些用于高效生成多种概率分布的样本值的函数。例如，你可以用`normal`来得到一个标准正态分布的 $4 \times 4$ 样本数组：

```
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]])
```

而Python内置的`random`模块则只能一次生成一个样本值。从下面的测试结果中可以看出，如果需要产生大量样本值，`numpy.random`快了不止一个数量级：

```
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
```

表4-8列出了`numpy.random`中的部分函数。在下一节中，我将给出一些利用这些函数一次性生成大量样本值的范例。

表4-8：部分`numpy.random`函数

函数	说明
<code>seed</code>	确定随机数生成器的种子
<code>permutation</code>	返回一个序列的随机排列或返回一个随机排列的范围
<code>shuffle</code>	对一个序列就地随机排列
<code>rand</code>	产生均匀分布的样本值
<code>randint</code>	从给定的上下限范围内随机选取整数
<code>randn</code>	产生正态分布（平均值为0，标准差为1）的样本值，类似于MATLAB接口
<code>binomial</code>	产生二项分布的样本值
<code>normal</code>	产生正态（高斯）分布的样本值
<code>beta</code>	产生Beta分布的样本值



表4-8：部分numpy.random函数（续）

函数	说明
chisquare	产生卡方分布的样本值
gamma	产生Gamma分布的样本值
uniform	产生在[0, 1)中均匀分布的样本值

## 范例：随机漫步

我们通过模拟随机漫步来说明如何运用数组运算。先来看一个简单的随机漫步的例子：从0开始，步长1和-1出现的概率相等。我们通过内置的random模块以纯Python的方式实现1000步的随机漫步：

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

图4-4是根据前100个随机漫步值生成的折线图。

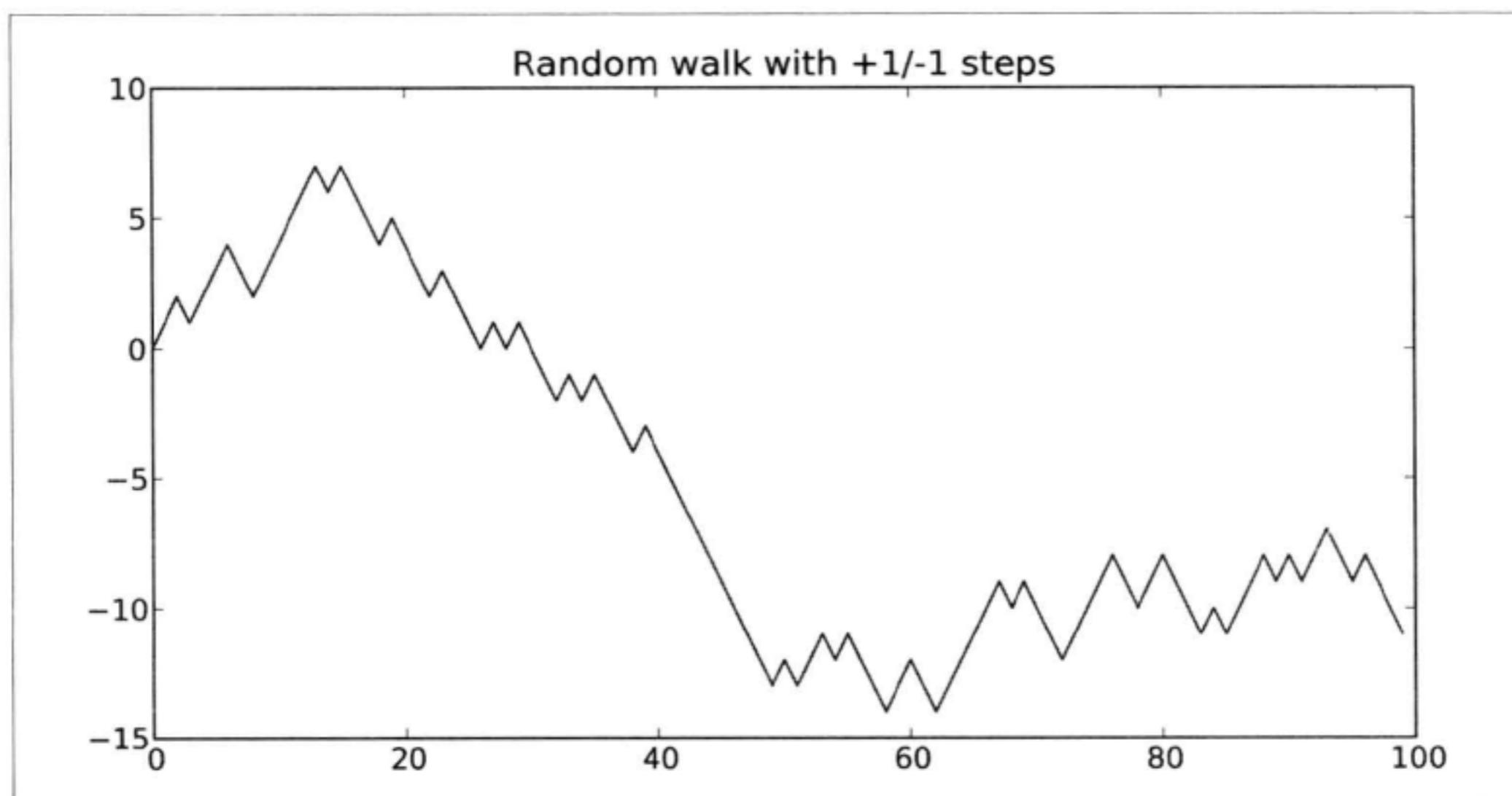


图4-4：简单的随机漫步

不难看出，这其实就是随机漫步中各步的累计和，可以用一个数组运算来实现。因此，



我用np.random模块一次性随机产生1000个“掷硬币”结果（即两个数中任选一个），将其分别设置为1或-1，然后计算累计和：

```
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()
```

有了这些数据之后，我们就可以做一些统计工作了，比如求取最大值和最小值：

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

现在来看一个复杂点的统计任务——首次穿越时间，即随机漫步过程中第一次到达某个特定值的时间。假设我们想要知道本次随机漫步需要多久才能距离初始0点至少10步远（任一方向均可）。`np.abs(walk) >= 10`可以得到一个布尔型数组，它表示的是距离是否达到或超过10，而我们想要知道的是第一个10或-10的索引。可以用`argmax`来解决这个问题，它返回的是该布尔型数组第一个最大值的索引（True就是最大值）：

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

注意，这里使用`argmax`并不是很高效，因为它无论如何都会对数组进行完全扫描。在本例中，只要发现了一个True，那我们就知道它是个最大值了。

## 一次模拟多个随机漫步

如果你希望模拟多个随机漫步过程（比如5000个），只需对上面的代码做一点点修改即可生成所有的随机漫步过程。只要给numpy.random的函数传入一个二元元组就可以产生一个二维数组，然后我们就可以一次性计算5000个随机漫步过程（一行一个）的累计和了：

```
In [222]: nwalks = 5000  
In [223]: nsteps = 1000  
In [224]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0或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]])
```

现在，我们来计算所有随机漫步过程的最大值和最小值：

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

得到这些数据之后，我们来计算30或-30的最小穿越时间。这里得要稍微动一下脑筋，因为不是5000个过程都到达了30。我们可以用any方法来对此进行检查：

```
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() # 到达30或-30的数量
Out[232]: 3410
```

然后我们利用这个布尔型数组选出那些穿越了30（绝对值）的随机漫步（行），并调用argmax在轴1上获取穿越时间：

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

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

请尝试用其他分布方式得到漫步数据。只需使用不同的随机数生成函数即可，如normal用于生成指定均值和标准差的正态分布数据：

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

