Structured Data: NumPy's Structured Arrays

格式化数据：NumPy里的格式化数组

> While often our data can be well represented by a homogeneous array of values, sometimes this is not the case. This section demonstrates the use of NumPy's *structured arrays* and *record arrays*, which provide efficient storage for compound, heterogeneous data. While the patterns shown here are useful for simple operations, scenarios like this often lend themselves to the use of Pandas `DataFrame`s, which we'll explore in Chapter 3.

虽然我们的数据很多情况下都能表示成同种类的数组，但是某些情况下，这是不适用的。本小节展示了如何使用NumPy的*结构化数组*和*记录数组*，它们能够提供对于复合的，不同种类的数组的有效存储形式。本小节的内容，包括场景和操作，通常都会在Pandas的`DataFrame`中使用，有关内容我们会在第三章中详细讨论。

In [1]:
```
import numpy as np
```

> Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

考虑一下，我们有一些关于人的不同种类的数据（例如姓名，年龄和体重），现在我们想要将它们保存到python程序中。当然它们可以被保存到三个独立的数组之中：

In [2]:
```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

> But this is a bit clumsy. There's nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data. NumPy can handle this through structured arrays, which are arrays with compound data types.

显然这种做法有些笨拙。没有任何额外的信息让我们知道这三个数组是关联的；如果我们可以使用一个结构保存所有这些数据的话，会更加的自然。NumPy使用结构化数组来处理这种情况，结构化数组可以用来存储复合的数据类型。

> Recall that previously we created a simple array using an expression like this:

回忆前面我们创建一个简单数组的方法：

In [3]:
```
x = np.zeros(4, dtype=int)
```

> We can similarly create a structured array using a compound data type specification:

我们也可以类似的创建一个复合类型的数组，只需要指定相应的dtype数据类型即可：

In [4]:
```
# 使用复杂的dtype参数来创建结构化数组
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                          'formats':('U10', 'i4', 'f8')})
print(data.dtype)
```

```
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

> Here `'U10'` translates to "Unicode string of maximum length 10," `'i4'` translates to "4-byte (i.e., 32 bit) integer," and `'f8'` translates to "8-byte (i.e., 64 bit) float." We'll discuss other options for these type codes in the following section.

这里的 `U10` 代表着"unicode编码的字符串，最大长度10"， `i4` 代表着"4字节（32比特）整数"， `f8` 代表着"8字节（64比特）浮点数"。本节后面我们会介绍其他的类型选项。

> Now that we've created an empty container array, we can fill the array with our lists of values:

现在我们已经创建了一个空的结构化数组，我们可以使用上面的数据列表将数据填充到数组中：

In [5]:
```
data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
```

```
[('Alice', 25, 55. ) ('Bob', 45, 85.5) ('Cathy', 37, 68. )
 ('Doug', 19, 61.5)]
```

> As we had hoped, the data is now arranged together in one convenient block of memory.

正如我们希望那样，数组的数据现在被存储在一整块的内存空间中。

> The handy thing with structured arrays is that you can now refer to values either by index or by name:

使用结构化数组的方便的地方是你可以使用字段的名称而不是序号来访问元素值了：

In [6]:
```
# 获得所有的名字
data['name']
```

Out[6]:
```
array(['Alice', 'Bob', 'Cathy', 'Doug'], dtype='<U10')
```

In [7]:
```
# 获得第一行
data[0]
```

Out[7]:
```
('Alice', 25, 55.)
```

In [8]:
```
# 获得最后一行的名字
data[-1]['name']
```

Out[8]:
```
'Doug'
```

> Using Boolean masking, this even allows you to do some more sophisticated operations such as filtering on age:

使用布尔遮盖，我们能写出更加复杂但易懂的过滤条件，比如年龄的过滤：

In [9]:
```
# 获得所有年龄小于30的人的姓名
data[data['age'] < 30]['name']
```

Out[9]:
```
array(['Alice', 'Doug'], dtype='<U10')
```

> Note that if you'd like to do any operations that are any more complicated than these, you should probably consider the Pandas package, covered in the next chapter. As we'll see, Pandas provides a `DataFrame` object, which is a structure built on NumPy arrays that offers a variety of useful data manipulation functionality similar to what we've shown here, as well as much, much more.

请注意，如果你想要完成的工作比上面的需求还要复杂的话，你应该考虑使用Pandas包，下一章的主要内容。我们将会看到，Pandas提供了 `DataFrame` 对象，它是一个在NumPy数组的基础上构建的结构，提供了很多有用的数据操作功能，包括上面结构化数组的功能。

Creating Structured Arrays

创建结构化数组

> Structured array data types can be specified in a number of ways. Earlier, we saw the dictionary method:

结构化数组的数据类型可以采用集中方式指定。前面我们介绍了字典的方式：

In [10]:
```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':('U10', 'i4', 'f8')})
```

Out[10]:
```
dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

> For clarity, numerical types can be specified using Python types or NumPy `dtype`s instead:

需要说明的是，数字类型也可以通过Python类型或NumPy数据类型来指定：

In [11]:
```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':((np.str_, 10), int, np.float32)})
```

Out[11]:
```
dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

> A compound type can also be specified as a list of tuples:

一个复合类型也可以使用一个元祖的列表来指定：

In [12]:
```
np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
```

Out[12]:
```
dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

> If the names of the types do not matter to you, you can specify the types alone in a comma-separated string:

如果类型的名称并不重要，你可以省略它们，你甚至可以在一个以逗号分隔的字符串中指定所有类型：

In [13]:
```
np.dtype('S10,i4,f8')
```

Out[13]:
```
dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

> The shortened string format codes may seem confusing, but they are built on simple principles. The first (optional) character is `<` or `>`, which means "little endian" or "big endian," respectively, and specifies the ordering convention for significant bits. The next character specifies the type of data: characters, bytes, ints, floating points, and so on (see the table below). The last character or characters represents the size of the object in bytes.

类型的字符串形式的缩写初看起来很困惑，但实际上它们都是依据简单原则得到的。第一个（可选的）字符是 `<` 或 `>`，代表这类型是 *小尾* 还是 *大尾*，用来指定存储的字节序。下一个字符指定数据类型：字符，字节，整数，浮点数或其他（见下表）。最后一个字符代表类型的长度。

字符	说明	举例
`'b'`	字节	`np.dtype('b')`
`'i'`	带符号整数	`np.dtype('i4') == np.int32`
`'u'`	无符号整数	`np.dtype('u1') == np.uint8`
`'f'`	浮点数	`np.dtype('f8') == np.int64`
`'c'`	复数	`np.dtype('c16') == np.complex128`
`'S', 'a'`	字符串	`np.dtype('S5')`
`'U'`	Unicode字符串	`np.dtype('U') == np.str_`
`'V'`	原始数据	`np.dtype('V') == np.void`

More Advanced Compound Types

高级复合类型

> It is possible to define even more advanced compound types. For example, you can create a type where each element contains an array or matrix of values. Here, we'll create a data type with a `mat` component consisting of a 3×3 floating-point matrix:

除此之外，还可以定义更加复杂的复合类型。例如，你可以创建一个类型，其中的每一个元素都是一个数组或矩阵。下面，创建一个数据类型内含一个 `mat` 对象，里一个3 x 3的浮点数矩阵：

In [14]:
```
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][0])
```

```
(0, [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

> Now each element in the `X` array consists of an `id` and a 3×3 matrix. Why would you use this rather than a simple multidimensional array, or perhaps a Python dictionary? The reason is that this NumPy `dtype` directly maps onto a C structure definition, so the buffer containing the array content can be accessed directly within an appropriately written C program. If you find yourself writing a Python interface to a legacy C or Fortran library that manipulates structured data, you'll probably find structured arrays quite useful!

`X` 数组中的每个元素都有一个 `id` 和一个 3 x 3的矩阵。为什么需要这样做，为什么不用一个多维数组或者甚至是Python的字典呢？原因是NumPy的 `dtype` 数据类型直接对应这一个C语言的结构体定义，因此存储这个数组的内容可以直接被C语言的程序访问到。如果你在写访问底层C语言或Fortran语言的Python接口的话，你会发现这种结构化数组很有用。

RecordArrays: Structured Arrays with a Twist

记录数组：面向对象的结构化数组

> NumPy also provides the `np.recarray` class, which is almost identical to the structured arrays just described, but with one additional feature: fields can be accessed as attributes rather than as dictionary keys. Recall that we previously accessed the ages by writing:

NumPy还提供了 `np.recarray` 对象，看起来基本和前面介绍的结构化数组相同，但是有一个额外的特性：字段不是使用字典关键字来访问，而是使用属性进行访问。前面我们使用关键字来访问数组的年龄字段：

In [15]:
```
data['age']
```

Out[15]:
```
array([25, 45, 37, 19], dtype=int32)
```

> If we view our data as a record array instead, we can access this with slightly fewer keystrokes:

如果我们使用记录数组来展示数据化，我们可以使用对象属性方式访问年龄字段，少打几个字：

In [16]:
```
data_rec = data.view(np.recarray)
data_rec.age
```

Out[16]:
```
array([25, 45, 37, 19], dtype=int32)
```

> The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax. We can see this here:

这样做的缺点是，当按照对象属性来访问数组数据时，会有额外的性能损耗。下面的例子可以看到：

In [17]:
```
%timeit data['age']
%timeit data_rec['age']
%timeit data_rec.age
```

```
95.2 ns ± 3.1 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
2.42 µs ± 187 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
3.13 µs ± 206 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

> Whether the more convenient notation is worth the additional overhead will depend on your own application.

是使用更方便简洁的写法还是使用更高性能的写法，取决于你应用的需求。

On to Pandas

进入Pandas

> This section on structured and record arrays is purposely at the end of this chapter, because it leads so well into the next package we will cover: Pandas. Structured arrays like the ones discussed here are good to know about for certain situations, especially in case you're using NumPy arrays to map onto binary data formats in C, Fortran, or another language. For day-to-day use of structured data, the Pandas package is a much better choice, and we'll dive into a full discussion of it in the chapter that follows.

本小节介绍的结构化和记录数组是本章的结束内容。因为它将带我们进入下一章的主要内容：Pandas。本节介绍的结构化数组在某些情况下是有用的，特别是当你使用NumPy数组来获取C、Fortran或其他语言存储的二进制数据时。但是对于日常的结构化数据应用来说，Pandas包是一个好得多的选择，我们在下一章会以一整章的篇幅来详细介绍它。