

# Advanced NumPy

## ndarray Object Internals

The NumPy ndarray provides a means to interpret a block of homogeneous data (either contiguous or strided, more on this later) as a multidimensional array object. As you've seen, the data type, or *dtype*, determines how the data is interpreted as being floating point, integer, boolean, or any of the other types we've been looking at.

Part of what makes ndarray powerful is that every array object is a *strided* view on a block of data. You might wonder, for example, how the array view `arr[:, :2, ::-1]` does not copy any data. Simply put, the ndarray is more than just a chunk of memory and a dtype; it also has striding information which enables the array to move through memory with varying step sizes. More precisely, the ndarray internally consists of the following:

- A *pointer to data*, that is a block of system memory
- The *data type* or dtype
- A tuple indicating the array's *shape*; For example, a 10 by 5 array would have shape (10, 5)

```
In [8]: np.ones((10, 5)).shape
Out[8]: (10, 5)
```

- A tuple of *strides*, integers indicating the number of bytes to “step” in order to advance one element along a dimension; For example, a typical (C order, more on this later) 3 x 4 x 5 array of `float64` (8-byte) values has strides (160, 40, 8)

```
In [9]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[9]: (160, 40, 8)
```

While it is rare that a typical NumPy user would be interested in the array strides, they are the critical ingredient in constructing copyless array views. Strides can even be negative which enables an array to move *backward* through memory, which would be the case in a slice like `obj[::-1]` or `obj[:, ::-1]`.

See [Figure 12-1](#) for a simple mockup the ndarray innards.

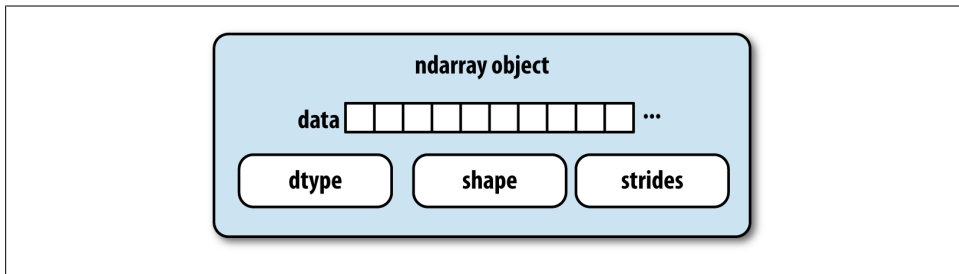


Figure 12-1. The NumPy ndarray object

## NumPy dtype Hierarchy

You may occasionally have code which needs to check whether an array contains integers, floating point numbers, strings, or Python objects. Because there are many types of floating point numbers (`float16` through `float128`), checking that the dtype is among a list of types would be very verbose. Fortunately, the dtypes have superclasses such as `np.integer` and `np.floating` which can be used in conjunction with the `np.issubdtype` function:

```
In [10]: ints = np.ones(10, dtype=np.uint16)

In [11]: floats = np.ones(10, dtype=np.float32)

In [12]: np.issubdtype(ints.dtype, np.integer)
Out[12]: True

In [13]: np.issubdtype(floats.dtype, np.floating)
Out[13]: True
```

You can see all of the parent classes of a specific dtype by calling the type's `mro` method:

```
In [14]: np.float64.mro()
Out[14]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

Most NumPy users will never have to know about this, but it occasionally comes in handy. See [Figure 12-2](#) for a graph of the dtype hierarchy and parent-subclass relationships <sup>1</sup>.

1. Some of the dtypes have trailing underscores in their names. These are there to avoid variable name conflicts between the NumPy-specific types and the Python built-in ones.

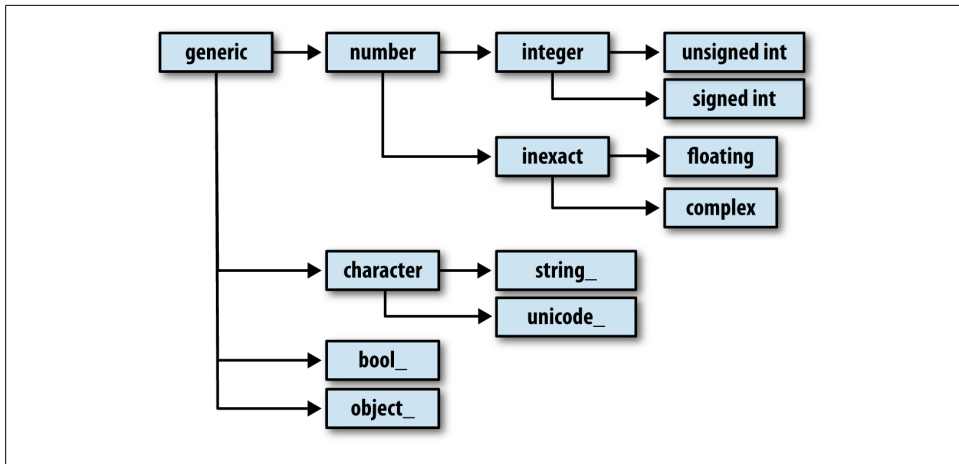


Figure 12-2. The NumPy dtype class hierarchy

## Advanced Array Manipulation

There are many ways to work with arrays beyond fancy indexing, slicing, and boolean subsetting. While much of the heavy lifting for data analysis applications is handled by higher level functions in pandas, you may at some point need to write a data algorithm that is not found in one of the existing libraries.

### Reshaping Arrays

Given what we know about NumPy arrays, it should come as little surprise that you can convert an array from one shape to another without copying any data. To do this, pass a tuple indicating the new shape to the `reshape` array instance method. For example, suppose we had a one-dimensional array of values that we wished to rearrange into a matrix:

```

In [15]: arr = np.arange(8)

In [16]: arr
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [17]: arr.reshape((4, 2))
Out[17]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
  
```

A multidimensional array can also be reshaped:

```

In [18]: arr.reshape((4, 2)).reshape((2, 4))
Out[18]:
  
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

One of the passed shape dimensions can be -1, in which case the value used for that dimension will be inferred from the data:

```
In [19]: arr = np.arange(15)
In [20]: arr.reshape((5, -1))
Out[20]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Since an array's `shape` attribute is a tuple, it can be passed to `reshape`, too:

```
In [21]: other_arr = np.ones((3, 5))

In [22]: other_arr.shape
Out[22]: (3, 5)

In [23]: arr.reshape(other_arr.shape)
Out[23]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

The opposite operation of `reshape` from one-dimensional to a higher dimension is typically known as *flattening* or *raveling*:

```
In [24]: arr = np.arange(15).reshape((5, 3))
In [25]: arr
Out[25]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])

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

`ravel` does not produce a copy of the underlying data if it does not have to (more on this below). The `flatten` method behaves like `ravel` except it always returns a copy of the data:

```
In [27]: arr.flatten()
Out[27]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

The data can be reshaped or raveled in different orders. This is a slightly nuanced topic for new NumPy users and is therefore the next subtopic.

## C versus Fortran Order

Contrary to some other scientific computing environments like R and MATLAB, NumPy gives you much more control and flexibility over the layout of your data in

memory. By default, NumPy arrays are created in *row major* order. Spatially this means that if you have a two-dimensional array of data, the items in each row of the array are stored in adjacent memory locations. The alternative to row major ordering is *column major* order, which means that (you guessed it) values within each column of data are stored in adjacent memory locations.

For historical reasons, row and column major order are also known as C and Fortran order, respectively. In FORTRAN 77, the language of our forebears, matrices were all column major.

Functions like `reshape` and `ravel`, accept an `order` argument indicating the order to use the data in the array. This can be 'C' or 'F' in most cases (there are also less commonly-used options 'A' and 'K'; see the NumPy documentation). These are illustrated in [Figure 12-3](#).

```
In [28]: arr = np.arange(12).reshape((3, 4))

In [29]: arr
Out[29]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [30]: arr.ravel()
Out[30]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [31]: arr.ravel('F')
Out[31]: array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

Reshaping arrays with more than two dimensions can be a bit mind-bending. The key difference between C and Fortran order is the order in which the dimensions are walked:

- *C / row major order*: traverse higher dimensions *first* (e.g. axis 1 before advancing on axis 0).
- *Fortran / column major order*: traverse higher dimensions *last* (e.g. axis 0 before advancing on axis 1).

## Concatenating and Splitting Arrays

`numpy.concatenate` takes a sequence (tuple, list, etc.) of arrays and joins them together in order along the input axis.

```
In [32]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [33]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [34]: np.concatenate([arr1, arr2], axis=0)
Out[34]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
[ 7,  8,  9],
 [10, 11, 12]])

In [35]: np.concatenate([arr1, arr2], axis=1)
Out[35]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

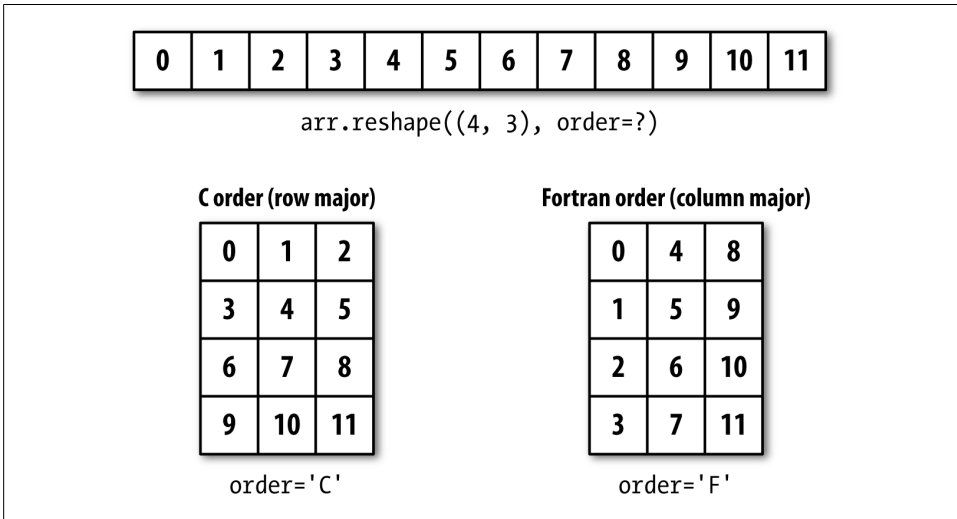


Figure 12-3. Reshaping in C (row major) or Fortran (column major) order

There are some convenience functions, like `vstack` and `hstack`, for common kinds of concatenation. The above operations could have been expressed as:

```
In [36]: np.vstack((arr1, arr2))
Out[36]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [37]: np.hstack((arr1, arr2))
Out[37]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

`split`, on the other hand, slices apart an array into multiple arrays along an axis:

```
In [38]: from numpy.random import randn

In [39]: arr = randn(5, 2)
In [40]: arr
Out[40]:
array([[ 0.1689,  0.3287],
       [ 0.4703,  0.8989],
       [ 0.1535,  0.0243],
       [-0.2832,  1.1536],
       [ 0.2707,  0.8075]])

In [41]: first, second, third = np.split(arr, [1, 3])

In [42]: first
```

```
Out[42]: array([[ 0.1689,  0.3287]])

In [43]: second
Out[43]: array([[ 0.4703,  0.8989],
                [ 0.1535,  0.0243]])

In [44]: third
Out[44]: array([[ -0.2832,  1.1536],
                [ 0.2707,  0.8075]])
```

See [Table 12-1](#) for a list of all relevant concatenation and splitting functions, some of which are provided only as a convenience of the very general purpose `concatenate`.

*Table 12-1. Array concatenation functions*

Function	Description
<code>concatenate</code>	Most general function, concatenates collection of arrays along one axis
<code>vstack, row_stack</code>	Stack arrays row-wise (along axis 0)
<code>hstack</code>	Stack arrays column-wise (along axis 1)
<code>column_stack</code>	Like <code>hstack</code> , but converts 1D arrays to 2D column vectors first
<code>dstack</code>	Stack arrays “depth”-wise (along axis 2)
<code>split</code>	Split array at passed locations along a particular axis
<code>hsplit / vsplit / dsplit</code>	Convenience functions for splitting on axis 0, 1, and 2, respectively.

## Stacking helpers: `r_` and `c_`

There are two special objects in the NumPy namespace, `r_` and `c_`, that make stacking arrays more concise:

```
In [45]: arr = np.arange(6)

In [46]: arr1 = arr.reshape((3, 2))

In [47]: arr2 = randn(3, 2)

In [48]: np.r_[arr1, arr2]
Out[48]: array([[ 0.    ,  1.    ],
                [ 2.    ,  3.    ],
                [ 4.    ,  5.    ],
                [ 0.7258, -1.5325],
                [-0.4696, -0.2127],
                [-0.1072,  1.2871]])

In [49]: np.c_[np.r_[arr1, arr2], arr]
Out[49]: array([[ 0.    ,  1.    ,  0.    ],
                [ 2.    ,  3.    ,  1.    ],
                [ 4.    ,  5.    ,  2.    ],
                [ 0.7258, -1.5325,  3.    ],
                [-0.4696, -0.2127,  4.    ],
                [-0.1072,  1.2871,  5.    ]])
```

These additionally can translate slices to arrays:

```
In [50]: np.c_[1:6, -10:-5]
Out[50]: array([[ 1, -10],
                [ 2, -9],
                [ 3, -8],
                [ 4, -7],
                [ 5, -6]])
```

See the docstring for more on what you can do with `c_` and `r_`.

## Repeating Elements: Tile and Repeat



The need to replicate or repeat arrays is less common with NumPy than it is with other popular array programming languages like MATLAB. The main reason for this is that *broadcasting* fulfills this need better, which is the subject of the next section.

The two main tools for repeating or replicating arrays to produce larger arrays are the `repeat` and `tile` functions. `repeat` replicates each element in an array some number of times, producing a larger array:

```
In [51]: arr = np.arange(3)

In [52]: arr.repeat(3)
Out[52]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

By default, if you pass an integer, each element will be repeated that number of times. If you pass an array of integers, each element can be repeated a different number of times:

```
In [53]: arr.repeat([2, 3, 4])
Out[53]: array([0, 0, 1, 1, 1, 2, 2, 2, 2, 2])
```

Multidimensional arrays can have their elements repeated along a particular axis.

```
In [54]: arr = randn(2, 2)

In [55]: arr
Out[55]:
array([[ 0.7157, -0.6387],
       [ 0.3626,  0.849 ]])

In [56]: arr.repeat(2, axis=0)
Out[56]:
array([[ 0.7157, -0.6387],
       [ 0.7157, -0.6387],
       [ 0.3626,  0.849 ],
       [ 0.3626,  0.849 ]])
```

Note that if no axis is passed, the array will be flattened first, which is likely not what you want. Similarly you can pass an array of integers when repeating a multidimensional array to repeat a given slice a different number of times:

```
In [57]: arr.repeat([2, 3], axis=0)
Out[57]:
array([[ 0.7157, -0.6387],
       [ 0.7157, -0.6387],
       [ 0.3626,  0.849 ],
       [ 0.3626,  0.849 ],
       [ 0.3626,  0.849 ]])

In [58]: arr.repeat([2, 3], axis=1)
Out[58]:
array([[ 0.7157,  0.7157, -0.6387, -0.6387, -0.6387],
       [ 0.3626,  0.3626,  0.849 ,  0.849 ,  0.849 ]])
```



`tile`, on the other hand, is a shortcut for stacking copies of an array along an axis. You can visually think about it as like “laying down tiles”:

```
In [59]: arr
Out[59]:
array([[ 0.7157, -0.6387],
       [ 0.3626,  0.849 ]])

In [60]: np.tile(arr, 2)
Out[60]:
array([[ 0.7157, -0.6387,  0.7157, -0.6387],
       [ 0.3626,  0.849 ,  0.3626,  0.849 ]])
```

The second argument is the number of tiles; with a scalar, the tiling is made row-by-row, rather than column by column: The second argument to `tile` can be a tuple indicating the layout of the “tiling”:

```
In [61]: arr
Out[61]:
array([[ 0.7157, -0.6387],
       [ 0.3626,  0.849 ]])

In [62]: np.tile(arr, (2, 1))
Out[62]:
array([[ 0.7157, -0.6387],
       [ 0.3626,  0.849 ],
       [ 0.7157, -0.6387],
       [ 0.3626,  0.849 ]])

In [63]: np.tile(arr, (3, 2))
Out[63]:
array([[ 0.7157, -0.6387,  0.7157, -0.6387],
       [ 0.3626,  0.849 ,  0.3626,  0.849 ],
       [ 0.7157, -0.6387,  0.7157, -0.6387],
       [ 0.3626,  0.849 ,  0.3626,  0.849 ],
       [ 0.7157, -0.6387,  0.7157, -0.6387],
       [ 0.3626,  0.849 ,  0.3626,  0.849 ]])
```

## Fancy Indexing Equivalents: Take and Put

As you may recall from [Chapter 4](#), one way to get and set subsets of arrays is by *fancy* indexing using integer arrays:

```
In [64]: arr = np.arange(10) * 100

In [65]: inds = [7, 1, 2, 6]
Out[66]: arr[inds]
Out[66]: array([700, 100, 200, 600])
```

There are alternate ndarray methods that are useful in the special case of only making a selection on a single axis:

```
In [67]: arr.take(inds)
Out[67]: array([700, 100, 200, 600])

In [68]: arr.put(inds, 42)

In [69]: arr
Out[69]: array([  0,  42,  42, 300, 400, 500,  42,  42, 800, 900])

In [70]: arr.put(inds, [40, 41, 42, 43])
```

```
In [71]: arr
Out[71]: array([ 0, 41, 42, 300, 400, 500, 43, 40, 800, 900])
```

To use `take` along other axes, you can pass the `axis` keyword:

```
In [72]: inds = [2, 0, 2, 1]

In [73]: arr = randn(2, 4)

In [74]: arr
Out[74]:
array([[ -0.8237,  2.6047, -0.4578, -1.    ],
       [ 2.3198, -1.0792,  0.518 ,  0.2527]])

In [75]: arr.take(inds, axis=1)
Out[75]:
array([[ -0.4578, -0.8237, -0.4578,  2.6047],
       [ 0.518 ,  2.3198,  0.518 , -1.0792]])
```

`put` does not accept an `axis` argument but rather indexes into the flattened (one-dimensional, C order) version of the array (this could be changed in principle). Thus, when you need to set elements using an index array on other axes, you will want to use fancy indexing.



As of this writing, the `take` and `put` functions in general have better performance than their fancy indexing equivalents by a significant margin. I regard this as a “bug” and something to be fixed in NumPy, but it’s something worth keeping in mind if you’re selecting subsets of large arrays using integer arrays:

```
In [76]: arr = randn(1000, 50)

# Random sample of 500 rows
In [77]: inds = np.random.permutation(1000)[:500]

In [78]: %timeit arr[inds]
1000 loops, best of 3: 356 us per loop

In [79]: %timeit arr.take(inds, axis=0)
10000 loops, best of 3: 34 us per loop
```

## Broadcasting

*Broadcasting* describes how arithmetic works between arrays of different shapes. It is a very powerful feature, but one that can be easily misunderstood, even by experienced users. The simplest example of broadcasting occurs when combining a scalar value with an array:

```
In [80]: arr = np.arange(5)

In [81]: arr
Out[81]: array([0, 1, 2, 3, 4])

In [82]: arr * 4
Out[82]: array([ 0,  4,  8, 12, 16])
```

Here we say that the scalar value 4 has been *broadcast* to all of the other elements in the multiplication operation.

For example, we can demean each column of an array by subtracting the column means. In this case, it is very simple:

```
In [83]: arr = randn(4, 3)

In [84]: arr.mean(0)
Out[84]: array([ 0.1321,  0.552 ,  0.8571])

In [85]: demeaned = arr - arr.mean(0)

In [86]: demeaned
Out[86]: array([[ 0.1718, -0.1972, -1.3669],
                [-0.1292,  1.6529, -0.3429],
                [-0.2891, -0.0435,  1.2322],
                [ 0.2465, -1.4122,  0.4776]])

In [87]: demeaned.mean(0)
Out[87]: array([ 0., -0., -0.])
```

See Figure 12-4 for an illustration of this operation. Demeaning the rows as a broadcast operation requires a bit more care. Fortunately, broadcasting potentially lower dimensional values across any dimension of an array (like subtracting the row means from each column of a two-dimensional array) is possible as long as you follow the rules. This brings us to:

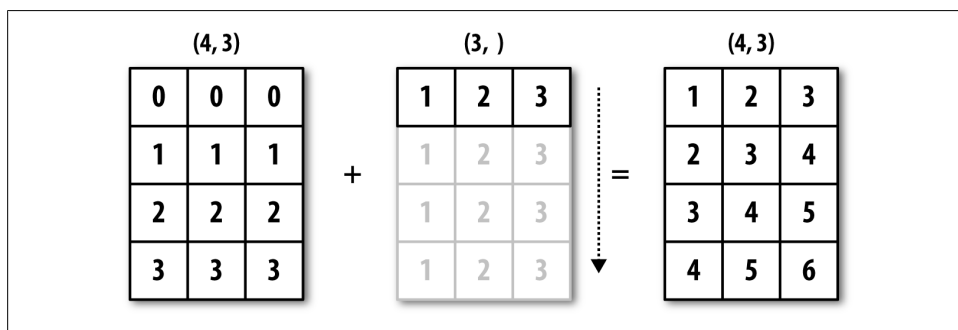


Figure 12-4. Broadcasting over axis 0 with a 1D array

## The Broadcasting Rule

Two arrays are compatible for broadcasting if for each *trailing dimension* (that is, starting from the end), the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing and / or length 1 dimensions.

Even as an experienced NumPy user, I often must stop to draw pictures and think about the broadcasting rule. Consider the last example and suppose we wished instead to subtract the mean value from each row. Since `arr.mean(0)` has length 3, it is compatible

for broadcasting across axis 0 because the trailing dimension in `arr` is 3 and therefore matches. According to the rules, to subtract over axis 1 (that is, subtract the row mean from each row), the smaller array must have shape `(4, 1)`:

```
In [88]: arr
Out[88]:
array([[ 0.3039,  0.3548, -0.5097],
       [ 0.0029,  2.2049,  0.5142],
       [-0.1571,  0.5085,  2.0893],
       [ 0.3786, -0.8602,  1.3347]])

In [89]: row_means = arr.mean(1)
In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.0496],
       [ 0.9073],
       [ 0.8136],
       [ 0.2844]])

In [91]: demeaned = arr - row_means.reshape((4, 1))

In [92]: demeaned.mean(1)
Out[92]: array([ 0.,  0.,  0.,  0.])
```

Has your head exploded yet? See [Figure 12-5](#) for an illustration of this operation.

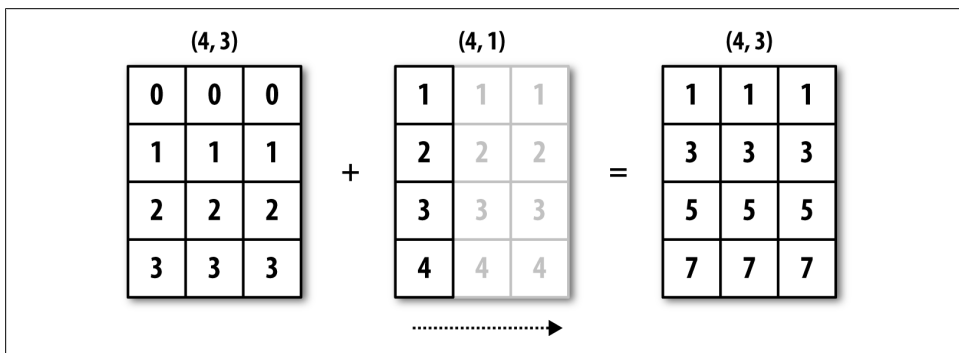


Figure 12-5. Broadcasting over axis 1 of a 2D array

See [Figure 12-6](#) for another illustration, this time adding a two-dimensional array to a three-dimensional one across axis 0.

## Broadcasting Over Other Axes

Broadcasting with higher dimensional arrays can seem even more mind-bending, but it is really a matter of following the rules. If you don't, you'll get an error like this:

```
In [93]: arr - arr.mean(1)
-----
ValueError                                Traceback (most recent call last)
<ipython-input-93-7b87b85a20b2> in <module>()
```

```
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes (4,3) (4)
```

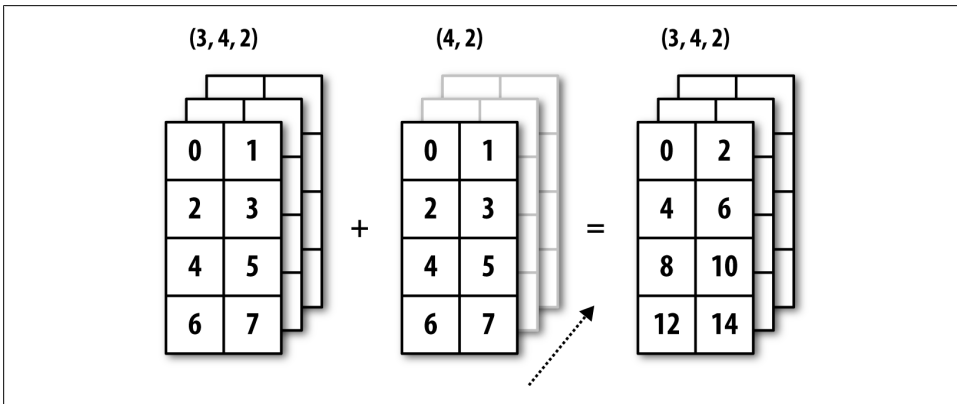


Figure 12-6. Broadcasting over axis 0 of a 3D array

It's quite common to want to perform an arithmetic operation with a lower dimensional array across axes other than axis 0. According to the broadcasting rule, the “broadcast dimensions” must be 1 in the smaller array. In the example of row demeaning above this meant reshaping the row means to be shape (4, 1) instead of (4,):

```
In [94]: arr - arr.mean(1).reshape((4, 1))
Out[94]:
array([[ 0.2542,  0.3051, -0.5594],
       [-0.9044,  1.2976, -0.3931],
       [-0.9707, -0.3051,  1.2757],
       [ 0.0942, -1.1446,  1.0503]])
```

In the three-dimensional case, broadcasting over any of the three dimensions is only a matter of reshaping the data to be shape-compatible. See Figure 12-7 for a nice visualization of the shapes required to broadcast over each axis of a three-dimensional array.

A very common problem, therefore, is needing to add a new axis with length 1 specifically for broadcasting purposes, especially in generic algorithms. Using `reshape` is one option, but inserting an axis requires constructing a tuple indicating the new shape. This can often be a tedious exercise. Thus, NumPy arrays offer a special syntax for inserting new axes by indexing. We use the special `np.newaxis` attribute along with “full” slices to insert the new axis:

```
In [95]: arr = np.zeros((4, 4))

In [96]: arr_3d = arr[:, np.newaxis, :]
In [97]: arr_3d.shape
Out[97]: (4, 1, 4)

In [98]: arr_1d = np.random.normal(size=3)

In [99]: arr_1d[:, np.newaxis]
Out[99]:
In [100]: arr_1d[np.newaxis, :]
Out[100]: array([[ -0.3899,  0.396 , -0.1852]])
```

```
array([[ -0.3899],
       [ 0.396  ],
       [-0.1852]])
```

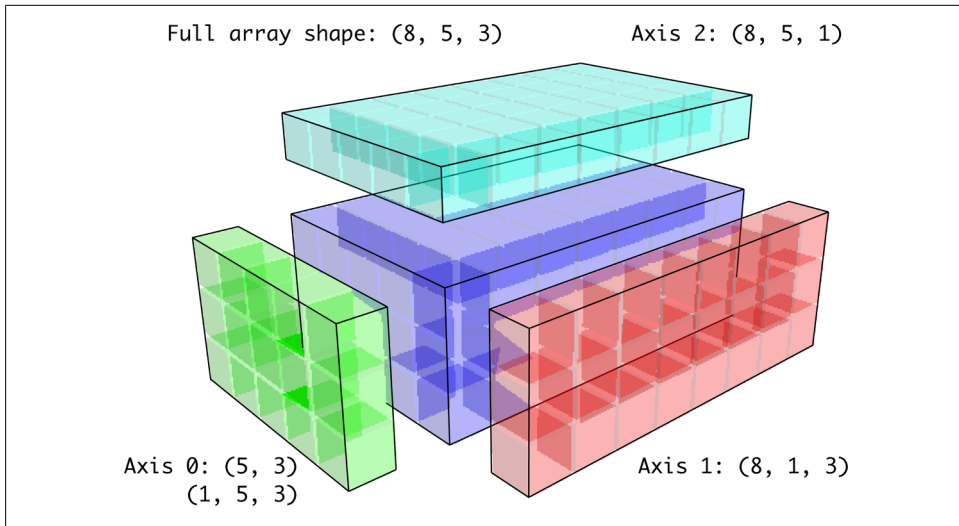


Figure 12-7. Compatible 2D array shapes for broadcasting over a 3D array

Thus, if we had a three-dimensional array and wanted to demean axis 2, say, we would only need to write:

```
In [101]: arr = randn(3, 4, 5)

In [102]: depth_means = arr.mean(2)

In [103]: depth_means
Out[103]:
array([[ 0.1097,  0.3118, -0.5473,  0.2663],
       [ 0.1747,  0.1379,  0.1146, -0.4224],
       [ 0.0217,  0.3686, -0.0468,  1.3026]])

In [104]: demeaned = arr - depth_means[:, :, np.newaxis]

In [105]: demeaned.mean(2)
Out[105]:
array([[ 0.,  0., -0.,  0.],
       [ 0., -0., -0.,  0.],
       [-0., -0.,  0.,  0.]])
```

If you're completely confused by this, don't worry. With practice you will get the hang of it!

Some readers might wonder if there's a way to generalize demeaning over an axis without sacrificing performance. There is, in fact, but it requires some indexing gymnastics:

```
def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # This generalized things like[:, :, np.newaxis] to N dimensions
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]
```

## Setting Array Values by Broadcasting

The same broadcasting rule governing arithmetic operations also applies to setting values via array indexing. In the simplest case, we can do things like:

```
In [106]: arr = np.zeros((4, 3))

In [107]: arr[:] = 5          In [108]: arr
Out[108]:
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

However, if we had a one-dimensional array of values we wanted to set into the columns of the array, we can do that as long as the shape is compatible:

```
In [109]: col = np.array([1.28, -0.42, 0.44, 1.6])

In [110]: arr[:, np.newaxis] = col[:, np.newaxis]
In [111]: arr
Out[111]:
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])

In [112]: arr[:2] = [[-1.37], [0.509]]
In [113]: arr
Out[113]:
array([[ -1.37, -1.37, -1.37 ],
       [ 0.509,  0.509,  0.509],
       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6 ,  1.6 ,  1.6 ]])
```

## Advanced ufunc Usage

While many NumPy users will only make use of the fast element-wise operations provided by the universal functions, there are a number of additional features that occasionally can help you write more concise code without loops.

## ufunc Instance Methods

Each of NumPy's binary ufuncs has special methods for performing certain kinds of special vectorized operations. These are summarized in [Table 12-2](#), but I'll give a few concrete examples to illustrate how they work.

`reduce` takes a single array and aggregates its values, optionally along an axis, by performing a sequence of binary operations. For example, an alternate way to sum elements in an array is to use `np.add.reduce`:

```
In [114]: arr = np.arange(10)
```

```
In [115]: np.add.reduce(arr)
Out[115]: 45
```

```
In [116]: arr.sum()
Out[116]: 45
```

The starting value (0 for `add`) depends on the ufunc. If an axis is passed, the reduction is performed along that axis. This allows you to answer certain kinds of questions in a concise way. As a less trivial example, we can use `np.logical_and` to check whether the values in each row of an array are sorted:

```
In [118]: arr = randn(5, 5)
```

```
In [119]: arr[:, :2].sort(1) # sort a few rows
```

```
In [120]: arr[:, :-1] < arr[:, 1:]
Out[120]:
array([[ True,  True,  True,  True],
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
```

```
In [121]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
Out[121]: array([ True, False,  True, False,  True], dtype=bool)
```

Of course, `logical_and.reduce` is equivalent to the `all` method.

`accumulate` is related to `reduce` like `cumsum` is related to `sum`. It produces an array of the same size with the intermediate “accumulated” values:

```
In [122]: arr = np.arange(15).reshape((3, 5))
```

```
In [123]: np.add.accumulate(arr, axis=1)
Out[123]:
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

`outer` performs a pairwise cross-product between two arrays:

```
In [124]: arr = np.arange(3).repeat([1, 2, 2])
```



```

In [125]: arr
Out[125]: array([0, 1, 1, 2, 2])

In [126]: np.multiply.outer(arr, np.arange(5))
Out[126]:
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])

```

The output of `outer` will have a dimension that is the sum of the dimensions of the inputs:

```

In [127]: result = np.subtract.outer(randn(3, 4), randn(5))

In [128]: result.shape
Out[128]: (3, 4, 5)

```

The last method, `reduceat`, performs a “local reduce”, in essence an array `groupby` operation in which slices of the array are aggregated together. While it’s less flexible than the `GroupBy` capabilities in `pandas`, it can be very fast and powerful in the right circumstances. It accepts a sequence of “bin edges” which indicate how to split and aggregate the values:

```

In [129]: arr = np.arange(10)

In [130]: np.add.reduceat(arr, [0, 5, 8])
Out[130]: array([10, 18, 17])

```

The results are the reductions (here, sums) performed over `arr[0:5]`, `arr[5:8]`, and `arr[8:]`. Like the other methods, you can pass an `axis` argument:

```

In [131]: arr = np.multiply.outer(np.arange(4), np.arange(5))

In [132]: arr
Out[132]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12]])

In [133]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[133]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])

```

Table 12-2. *ufunc methods*

Method	Description
<code>reduce(x)</code>	Aggregate values by successive applications of the operation
<code>accumulate(x)</code>	Aggregate values, preserving all partial aggregates
<code>reduceat(x, bins)</code>	“Local” reduce or “group by”. Reduce contiguous slices of data to produce aggregated array.
<code>outer(x, y)</code>	Apply operation to all pairs of elements in <code>x</code> and <code>y</code> . Result array has shape <code>x.shape + y.shape</code>

## Custom ufuncs

There are a couple facilities for creating your own functions with ufunc-like semantics. `numpy.frompyfunc` accepts a Python function along with a specification for the number of inputs and outputs. For example, a simple function that adds element-wise would be specified as:

```
In [134]: def add_elements(x, y):
.....:     return x + y

In [135]: add_them = np.frompyfunc(add_elements, 2, 1)

In [136]: add_them(np.arange(8), np.arange(8))
Out[136]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

Functions created using `frompyfunc` always return arrays of Python objects which isn't very convenient. Fortunately, there is an alternate, but slightly less featureful function `numpy.vectorize` that is a bit more intelligent about type inference:

```
In [137]: add_them = np.vectorize(add_elements, otypes=[np.float64])

In [138]: add_them(np.arange(8), np.arange(8))
Out[138]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.]
```

These functions provide a way to create ufunc-like functions, but they are very slow because they require a Python function call to compute each element, which is a lot slower than NumPy's C-based ufunc loops:

```
In [139]: arr = randn(10000)

In [140]: %timeit add_them(arr, arr)
100 loops, best of 3: 2.12 ms per loop

In [141]: %timeit np.add(arr, arr)
100000 loops, best of 3: 11.6 us per loop
```

There are a number of projects under way in the scientific Python community to make it easier to define new ufuncs whose performance is closer to that of the built-in ones.

## Structured and Record Arrays

You may have noticed up until now that `ndarray` is a *homogeneous* data container; that is, it represents a block of memory in which each element takes up the same number of bytes, determined by the `dtype`. On the surface, this would appear to not allow you to represent heterogeneous or tabular-like data. A *structured* array is an `ndarray` in which each element can be thought of as representing a *struct* in C (hence the “structured” name) or a row in a SQL table with multiple named fields:

```
In [142]: dtype = [('x', np.float64), ('y', np.int32)]

In [143]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)
```

```
In [144]: sarr
Out[144]:
array([(1.5, 6), (3.141592653589793, -2)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

There are several ways to specify a structured dtype (see the online NumPy documentation). One typical way is as a list of tuples with (`field_name`, `field_data_type`). Now, the elements of the array are tuple-like objects whose elements can be accessed like a dictionary:

```
In [145]: sarr[0]
Out[145]: (1.5, 6)

In [146]: sarr[0]['y']
Out[146]: 6
```

The field names are stored in the `dtype.names` attribute. On accessing a field on the structured array, a strided view on the data is returned thus copying nothing:

```
In [147]: sarr['x']
Out[147]: array([ 1.5 ,  3.1416])
```

## Nested dtypes and Multidimensional Fields

When specifying a structured dtype, you can additionally pass a shape (as an int or tuple):

```
In [148]: dtype = [('x', np.int64, 3), ('y', np.int32)]

In [149]: arr = np.zeros(4, dtype=dtype)

In [150]: arr
Out[150]:
array([(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0)],
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

In this case, the `x` field now refers to an array of length three for each record:

```
In [151]: arr[0]['x']
Out[151]: array([0, 0, 0])
```

Conveniently, accessing `arr['x']` then returns a two-dimensional array instead of a one-dimensional array as in prior examples:

```
In [152]: arr['x']
Out[152]:
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

This enables you to express more complicated, nested structures as a single block of memory in an array. Though, since dtypes can be arbitrarily complex, why not nested dtypes? Here is a simple example:

```

In [153]: dtype = [('x', [(('a', 'f8'), ('b', 'f4'))], ('y', np.int32)]

In [154]: data = np.array([(1, 2), 5], [(3, 4), 6], dtype=dtype)

In [155]: data['x']
Out[155]:
array([(1.0, 2.0), (3.0, 4.0)],
      dtype=[('a', '<f8'), ('b', '<f4')])

In [156]: data['y']
Out[156]: array([5, 6], dtype=int32)

In [157]: data['x']['a']
Out[157]: array([ 1.,  3.])

```

As you can see, variable-shape fields and nested records is a very rich feature that can be the right tool in certain circumstances. A DataFrame from pandas, by contrast, does not support this feature directly, though it is similar to hierarchical indexing.

## Why Use Structured Arrays?

Compared with, say, a DataFrame from pandas, NumPy structured arrays are a comparatively low-level tool. They provide a means to interpreting a block of memory as a tabular structure with arbitrarily complex nested columns. Since each element in the array is represented in memory as a fixed number of bytes, structured arrays provide a very fast and efficient way of writing data to and from disk (including memory maps, more on this later), transporting it over the network, and other such use.

As another common use for structured arrays, writing data files as fixed length record byte streams is a common way to serialize data in C and C++ code, which is commonly found in legacy systems in industry. As long as the format of the file is known (the size of each record and the order, byte size, and data type of each element), the data can be read into memory using `np.fromfile`. Specialized uses like this are beyond the scope of this book, but it's worth knowing that such things are possible.

## Structured Array Manipulations: `numpy.lib.recfunctions`

While there is not as much functionality available for structured arrays as for DataFrames, the NumPy module `numpy.lib.recfunctions` has some helpful tools for adding and dropping fields or doing basic join-like operations. The thing to remember with these tools is that it is typically necessary to create a new array to make any modifications to the dtype (like adding or dropping a column). These functions are left to the interested reader to explore as I do not use them anywhere in this book.

## More About Sorting

Like Python's built-in list, the ndarray `sort` instance method is an *in-place* sort, meaning that the array contents are rearranged without producing a new array:

```
In [158]: arr = randn(6)

In [159]: arr.sort()

In [160]: arr
Out[160]: array([-1.082 ,  0.3759,  0.8014,  1.1397,  1.2888,  1.8413])
```

When sorting arrays in-place, remember that if the array is a view on a different ndarray, the original array will be modified:

```
In [161]: arr = randn(3, 5)

In [162]: arr
Out[162]:
array([[ -0.3318, -1.4711,  0.8705, -0.0847, -1.1329],
       [ -1.0111, -0.3436,  2.1714,  0.1234, -0.0189],
       [  0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])

In [163]: arr[:, 0].sort() # Sort first column values in-place

In [164]: arr
Out[164]:
array([[ -1.0111, -1.4711,  0.8705, -0.0847, -1.1329],
       [ -0.3318, -0.3436,  2.1714,  0.1234, -0.0189],
       [  0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])
```

On the other hand, `numpy.sort` creates a new, sorted copy of an array. Otherwise it accepts the same arguments (such as `kind`, more on this below) as `ndarray.sort`:

```
In [165]: arr = randn(5)

In [166]: arr
Out[166]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])

In [167]: np.sort(arr)
Out[167]: array([-2.0051, -1.1181, -1.0614, -0.2415,  0.7379])

In [168]: arr
Out[168]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])
```

All of these sort methods take an `axis` argument for sorting the sections of data along the passed axis independently:

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

In [170]: arr
Out[170]:
array([[ 0.5955, -0.2682,  1.3389, -0.1872,  0.9111],
       [-0.3215,  1.0054, -0.5168,  1.1925, -0.1989],
       [ 0.3969, -1.7638,  0.6071, -0.2222, -0.2171]])
```

```
In [171]: arr.sort(axis=1)

In [172]: arr
Out[172]:
array([[ -0.2682, -0.1872,  0.5955,  0.9111,  1.3389],
       [ -0.5168, -0.3215, -0.1989,  1.0054,  1.1925],
       [ -1.7638, -0.2222, -0.2171,  0.3969,  0.6071]])
```

You may notice that none of the sort methods have an option to sort in descending order. This is not actually a big deal because array slicing produces views, thus not producing a copy or requiring any computational work. Many Python users are familiar with the “trick” that for a list `values`, `values[::-1]` returns a list in reverse order. The same is true for `ndarrays`:

```
In [173]: arr[:, ::-1]
Out[173]:
array([[ 1.3389,  0.9111,  0.5955, -0.1872, -0.2682],
       [ 1.1925,  1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071,  0.3969, -0.2171, -0.2222, -1.7638]])
```

## Indirect Sorts: `argsort` and `lexsort`

In data analysis it’s very common to need to reorder data sets by one or more keys. For example, a table of data about some students might need to be sorted by last name then by first name. This is an example of an *indirect* sort, and if you’ve read the pandas-related chapters you have already seen many higher-level examples. Given a key or keys (an array or values or multiple arrays of values), you wish to obtain an array of integer *indices* (I refer to them colloquially as *indexers*) that tells you how to reorder the data to be in sorted order. The two main methods for this are `argsort` and `numpy.lexsort`. As a trivial example:

```
In [174]: values = np.array([5, 0, 1, 3, 2])

In [175]: indexer = values.argsort()

In [176]: indexer
Out[176]: array([1, 2, 4, 3, 0])

In [177]: values[indexer]
Out[177]: array([0, 1, 2, 3, 5])
```

As a less trivial example, this code reorders a 2D array by its first row:

```
In [178]: arr = randn(3, 5)

In [179]: arr[0] = values

In [180]: arr
Out[180]:
array([[ 5.      ,  0.      ,  1.      ,  3.      ,  2.      ],
       [-0.3636, -0.1378,  2.1777, -0.4728,  0.8356],
       [-0.2089,  0.2316,  0.728 , -1.3918,  1.9956]])
```

```
In [181]: arr[:, arr[0].argsort()]
Out[181]:
array([[ 0.    ,  1.    ,  2.    ,  3.    ,  5.    ],
       [-0.1378,  2.1777,  0.8356, -0.4728, -0.3636],
       [ 0.2316,  0.728 ,  1.9956, -1.3918, -0.2089]])
```

`lexsort` is similar to `argsort`, but it performs an indirect *lexicographical* sort on multiple key arrays. Suppose we wanted to sort some data identified by first and last names:

```
In [182]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])
In [183]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'])
In [184]: sorter = np.lexsort((first_name, last_name))
In [185]: zip(last_name[sorter], first_name[sorter])
Out[185]:
[('Arnold', 'Jane'),
 ('Arnold', 'Steve'),
 ('Jones', 'Bill'),
 ('Jones', 'Bob'),
 ('Walters', 'Barbara')]
```

`lexsort` can be a bit confusing the first time you use it because the order in which the keys are used to order the data starts with the *last* array passed. As you can see, `last_name` was used before `first_name`.



pandas methods like `Series`'s and `DataFrame`'s `sort_index` methods and the `Series` `order` method are implemented with variants of these functions (which also must take into account missing values)

## Alternate Sort Algorithms

A *stable* sorting algorithm preserves the relative position of equal elements. This can be especially important in indirect sorts where the relative ordering is meaningful:

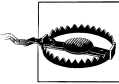
```
In [186]: values = np.array(['2:first', '2:second', '1:first', '1:second', '1:third'])
In [187]: key = np.array([2, 2, 1, 1, 1])
In [188]: indexer = key.argsort(kind='mergesort')
In [189]: indexer
Out[189]: array([2, 3, 4, 0, 1])
In [190]: values.take(indexer)
Out[190]:
array(['1:first', '1:second', '1:third', '2:first', '2:second'],
      dtype='<S8')
```

The only stable sort available is *mergesort* which has guaranteed  $O(n \log n)$  performance (for complexity buffs), but its performance is on average worse than the default

quicksort method. See [Table 12-3](#) for a summary of available methods and their relative performance (and performance guarantees). This is not something that most users will ever have to think about but useful to know that it's there.

Table 12-3. Array sorting methods

Kind	Speed	Stable	Work space	Worst-case
'quicksort'	1	No	0	$O(n^2)$
'mergesort'	2	Yes	$n/2$	$O(n \log n)$
'heapsort'	3	No	0	$O(n \log n)$



At the time of this writing, sort algorithms other than quicksort are not available on arrays of Python objects (`dtype=object`). This means occasionally that algorithms requiring stable sorting will require work-arounds when dealing with Python objects.

## numpy.searchsorted: Finding elements in a Sorted Array

`searchsorted` is an array method that performs a binary search on a sorted array, returning the location in the array where the value would need to be inserted to maintain sortedness:

```
In [191]: arr = np.array([0, 1, 7, 12, 15])  
  
In [192]: arr.searchsorted(9)  
Out[192]: 3
```

As you might expect, you can also pass an array of values to get an array of indices back:

```
In [193]: arr.searchsorted([0, 8, 11, 16])  
Out[193]: array([0, 3, 3, 5])
```

You might have noticed that `searchsorted` returned 0 for the 0 element. This is because the default behavior is to return the index at the left side of a group of equal values:

```
In [194]: arr = np.array([0, 0, 0, 1, 1, 1, 1])  
  
In [195]: arr.searchsorted([0, 1])  
Out[195]: array([0, 3])  
  
In [196]: arr.searchsorted([0, 1], side='right')  
Out[196]: array([3, 7])
```

As another application of `searchsorted`, suppose we had an array of values between 0 and 10,000) and a separate array of “bucket edges” that we wanted to use to bin the data:

```
In [197]: data = np.floor(np.random.uniform(0, 10000, size=50))  
  
In [198]: bins = np.array([0, 100, 1000, 5000, 10000])  
  
In [199]: data
```



```
Out[199]:
array([ 8304.,  4181.,  9352.,  4907.,  3250.,  8546.,  2673.,  6152.,
        2774.,  5130.,  9553.,  4997.,  1794.,  9688.,   426.,  1612.,
         651.,  8653.,  1695.,  4764.,  1052.,  4836.,  8020.,  3479.,
        1513.,  5872.,  8992.,  7656.,  4764.,  5383.,  2319.,  4280.,
        4150.,  8601.,  3946.,  9904.,  7286.,  9969.,  6032.,  4574.,
        8480.,  4298.,  2708.,  7358.,  6439.,  7916.,  3899.,  9182.,
         871.,  7973.]])
```

To then get a labeling of which interval each data point belongs to (where 1 would mean the bucket `[0, 100)`), we can simply use `searchsorted`:

```
In [200]: labels = bins.searchsorted(data)

In [201]: labels
Out[201]:
array([4, 3, 4, 3, 3, 4, 3, 4, 3, 4, 4, 3, 3, 4, 2, 3, 2, 4, 3, 3, 3, 3, 4,
       3, 3, 4, 4, 4, 3, 4, 3, 3, 3, 4, 3, 4, 4, 4, 4, 3, 4, 3, 3, 4, 4, 4,
       3, 4, 2, 4])
```

This, combined with pandas's `groupby`, can be used to easily bin data:

```
In [202]: Series(data).groupby(labels).mean()
Out[202]:
2    649.333333
3   3411.521739
4    7935.041667
```

Note that NumPy actually has a function `digitize` that computes this bin labeling:

```
In [203]: np.digitize(data, bins)
Out[203]:
array([4, 3, 4, 3, 3, 4, 3, 4, 3, 4, 4, 3, 3, 4, 2, 3, 2, 4, 3, 3, 3, 3, 4,
       3, 3, 4, 4, 4, 3, 4, 3, 3, 3, 4, 3, 4, 4, 4, 4, 3, 4, 3, 3, 4, 4, 4,
       3, 4, 2, 4])
```

## NumPy Matrix Class

Compared with other languages for matrix operations and linear algebra, like MATLAB, Julia, and GAUSS, NumPy's linear algebra syntax can often be quite verbose. One reason is that matrix multiplication requires using `numpy.dot`. Also NumPy's indexing semantics are different, which makes porting code to Python less straightforward at times. Selecting a single row (e.g. `X[1, :]`) or column (e.g. `X[:, 1]`) from a 2D array yields a 1D array compared with a 2D array as in, say, MATLAB.

```
In [204]: X = np.array([[ 8.82768214,  3.82222409, -1.14276475,  2.04411587],
.....:                  [ 3.82222409,  6.75272284,  0.83909108,  2.08293758],
.....:                  [-1.14276475,  0.83909108,  5.01690521,  0.79573241],
.....:                  [ 2.04411587,  2.08293758,  0.79573241,  6.24095859]])

In [205]: X[:, 0] # one-dimensional
Out[205]: array([ 8.8277,  3.8222, -1.1428,  2.0441])

In [206]: y = X[:, :1] # two-dimensional by slicing
```

```
In [207]: X
Out[207]:
array([[ 8.8277,  3.8222, -1.1428,  2.0441],
       [ 3.8222,  6.7527,  0.8391,  2.0829],
       [-1.1428,  0.8391,  5.0169,  0.7957],
       [ 2.0441,  2.0829,  0.7957,  6.241 ]])
```

```
In [208]: y
Out[208]:
array([[ 8.8277],
       [ 3.8222],
       [-1.1428],
       [ 2.0441]])
```

In this case, the product  $y^T X y$  would be expressed like so:

```
In [209]: np.dot(y.T, np.dot(X, y))
Out[209]: array([[ 1195.468]])
```

To aid in writing code with a lot of matrix operations, NumPy has a `matrix` class which has modified indexing behavior to make it more MATLAB-like: single rows and columns come back two-dimensional and multiplication with `*` is matrix multiplication. The above operation with `numpy.matrix` would look like:

```
In [210]: Xm = np.matrix(X)
```

```
In [211]: ym = Xm[:, 0]
```

```
In [212]: Xm
Out[212]:
matrix([[ 8.8277,  3.8222, -1.1428,  2.0441],
        [ 3.8222,  6.7527,  0.8391,  2.0829],
        [-1.1428,  0.8391,  5.0169,  0.7957],
        [ 2.0441,  2.0829,  0.7957,  6.241 ]])
```

```
In [213]: ym
Out[213]:
matrix([[ 8.8277],
        [ 3.8222],
        [-1.1428],
        [ 2.0441]])
```

```
In [214]: ym.T * Xm * ym
Out[214]: matrix([[ 1195.468]])
```

`matrix` also has a special attribute `I` which returns the matrix inverse:

```
In [215]: Xm.I * X
Out[215]:
matrix([[ 1., -0., -0., -0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  1.]])
```

I do not recommend using `numpy.matrix` as a replacement for regular `ndarrays` because they are generally more seldom used. In individual functions with lots of linear algebra, it may be helpful to convert the function argument to `matrix` type, then cast back to regular arrays with `np.asarray` (which does not copy any data) before returning them.

## Advanced Array Input and Output

In [Chapter 4](#), I introduced you to `np.save` and `np.load` for storing arrays in binary format on disk. There are a number of additional options to consider for more sophisticated use. In particular, memory maps have the additional benefit of enabling you to work with data sets that do not fit into RAM.

### Memory-mapped Files

A *memory-mapped* file is a method for treating potentially very large binary data on disk as an in-memory array. NumPy implements a `mmap` object that is `ndarray`-like, enabling small segments of a large file to be read and written without reading the whole array into memory. Additionally, a `mmap` has the same methods as an in-memory array and thus can be substituted into many algorithms where an `ndarray` would be expected.

To create a new `mmap`, use the function `np.mmap` and pass a file path, `dtype`, `shape`, and file mode:

```
In [216]: mmap = np.mmap('mymmap', dtype='float64', mode='w+', shape=(10000, 10000))
```

```
In [217]: mmap
```

```
Out[217]:
```

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

Slicing a `mmap` returns views on the data on disk:

```
In [218]: section = mmap[:5]
```

If you assign data to these, it will be buffered in memory (like a Python file object), but can be written to disk by calling `flush`:

```
In [219]: section[:] = np.random.randn(5, 10000)
```

```
In [220]: mmap.flush()
```

```
In [221]: mmap
```

```
Out[221]:
```

```
mmap([[-0.1614, -0.1768,  0.422 , ..., -0.2195, -0.1256, -0.4012],
       [ 0.4898, -2.2219, -0.7684, ..., -2.3517, -1.0782,  1.3208],
       [-0.6875,  1.6901, -0.7444, ..., -1.4218, -0.0509,  1.2224],
```

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

```
In [222]: del mmap
```

Whenever a memory map falls out of scope and is garbage-collected, any changes will be flushed to disk also. When *opening an existing memory map*, you still have to specify the dtype and shape as the file is just a block of binary data with no metadata on disk:

```
In [223]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))
```

```
In [224]: mmap
```

```
Out[224]:
```

```
memmap([[ -0.1614, -0.1768,  0.422 , ..., -0.2195, -0.1256, -0.4012],
 [ 0.4898, -2.2219, -0.7684, ..., -2.3517, -1.0782,  1.3208],
 [-0.6875,  1.6901, -0.7444, ..., -1.4218, -0.0509,  1.2224],
 ...,
 [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
 [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
 [ 0. , 0. , 0. , ..., 0. , 0. , 0. ]])
```

Since a memory map is just an on-disk ndarray, there are no issues using a structured dtype as described above.

## HDF5 and Other Array Storage Options

PyTables and h5py are two Python projects providing NumPy-friendly interfaces for storing array data in the efficient and compressible HDF5 format (HDF stands for *hierarchical data format*). You can safely store hundreds of gigabytes or even terabytes of data in HDF5 format. The use of these libraries is unfortunately outside the scope of the book.

PyTables provides a rich facility for working with structured arrays with advanced querying features and the ability to add column indexes to accelerate queries. This is very similar to the table indexing capabilities provided by relational databases.

## Performance Tips

Getting good performance out of code utilizing NumPy is often straightforward, as array operations typically replace otherwise comparatively extremely slow pure Python loops. Here is a brief list of some of the things to keep in mind:

- Convert Python loops and conditional logic to array operations and boolean array operations
- Use broadcasting whenever possible
- Avoid copying data using array views (slicing)
- Utilize ufuncs and ufunc methods

If you can't get the performance you require after exhausting the capabilities provided by NumPy alone, writing code in C, Fortran, or especially Cython (see a bit more on this below) may be in order. I personally use Cython (<http://cython.org>) heavily in my own work as an easy way to get C-like performance with minimal development.

## The Importance of Contiguous Memory

While the full extent of this topic is a bit outside the scope of this book, in some applications the memory layout of an array can significantly affect the speed of computations. This is based partly on performance differences having to do with the cache hierarchy of the CPU; operations accessing contiguous blocks of memory (for example, summing the rows of a C order array) will generally be the fastest because the memory subsystem will buffer the appropriate blocks of memory into the ultrafast L1 or L2 CPU cache. Also, certain code paths inside NumPy's C codebase have been optimized for the contiguous case in which generic strided memory access can be avoided.

To say that an array's memory layout is *contiguous* means that the elements are stored in memory in the order that they appear in the array with respect to Fortran (column major) or C (row major) ordering. By default, NumPy arrays are created as *C-contiguous* or just simply contiguous. A column major array, such as the transpose of a C-contiguous array, is thus said to be Fortran-contiguous. These properties can be explicitly checked via the `flags` attribute on the `ndarray`:

```
In [227]: arr_c = np.ones((1000, 1000), order='C')
```

```
In [228]: arr_f = np.ones((1000, 1000), order='F')
```

In [229]: arr_c.flags	In [230]: arr_f.flags
Out[229]:	Out[230]:
C_CONTIGUOUS : True	C_CONTIGUOUS : False
F_CONTIGUOUS : False	F_CONTIGUOUS : True
OWNDATA : True	OWNDATA : True
WRITEABLE : True	WRITEABLE : True
ALIGNED : True	ALIGNED : True
UPDATEIFCOPY : False	UPDATEIFCOPY : False

```
In [231]: arr_f.flags.f_contiguous
```

```
Out[231]: True
```

In this example, summing the rows of these arrays should, in theory, be faster for `arr_c` than `arr_f` since the rows are contiguous in memory. Here I check for sure using `%timeit` in IPython:

```
In [232]: %timeit arr_c.sum(1)
1000 loops, best of 3: 1.33 ms per loop
```

```
In [233]: %timeit arr_f.sum(1)
100 loops, best of 3: 8.75 ms per loop
```

When looking to squeeze more performance out of NumPy, this is often a place to invest some effort. If you have an array that does not have the desired memory order, you can use `copy` and pass either 'C' or 'F':

```
In [234]: arr_f.copy('C').flags
Out[234]:
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

When constructing a view on an array, keep in mind that the result is not guaranteed to be contiguous:

```
In [235]: arr_c[:50].flags.contiguous
Out[235]: True

In [236]: arr_c[:, :50].flags
Out[236]:
C_CONTIGUOUS : False
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

## Other Speed Options: Cython, f2py, C

In recent years, the Cython project (<http://cython.org>) has become the tool of choice for many scientific Python programmers for implementing fast code that may need to interact with C or C++ libraries, but without having to write pure C code. You can think of Cython as Python with static types and the ability to interleave functions implemented in C into Python-like code. For example, a simple Cython function to sum the elements of a one-dimensional array might look like:

```
from numpy cimport ndarray, float64_t

def sum_elements(ndarray[float64_t] arr):
    cdef Py_ssize_t i, n = len(arr)
    cdef float64_t result = 0

    for i in range(n):
        result += arr[i]

    return result
```

Cython takes this code, translates it to C, then compiles the generated C code to create a Python extension. Cython is an attractive option for performance computing because the code is only slightly more time-consuming to write than pure Python code and it integrates closely with NumPy. A common workflow is to get an algorithm working in Python, then translate it to Cython by adding type declarations and a handful of other tweaks. For more, see the project documentation.

Some other options for writing high performance code with NumPy include f2py, a wrapper generator for Fortran 77 and 90 code, and writing pure C extensions.

