```
In[12]: rng = np.random.RandomState(0)
        x = rng.randint(10, size=(3, 4))
        x

Out[12]: array([[5, 0, 3, 3],
                [7, 9, 3, 5],
                [2, 4, 7, 6]])

In[13]: x < 6

Out[13]: array([[ True,  True,  True,  True],
                [False, False,  True,  True],
                [ True,  True, False, False]], dtype=bool)
```

In each case, the result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

## Working with Boolean Arrays

Given a Boolean array, there are a host of useful operations you can do. We'll work with `x`, the two-dimensional array we created earlier:

```
In[14]: print(x)

[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
```

### Counting entries

To count the number of `True` entries in a Boolean array, `np.count_nonzero` is useful:

```
In[15]: # how many values less than 6?
        np.count_nonzero(x < 6)

Out[15]: 8
```

We see that there are eight array entries that are less than 6. Another way to get at this information is to use `np.sum`; in this case, `False` is interpreted as `0`, and `True` is interpreted as `1`:

```
In[16]: np.sum(x < 6)

Out[16]: 8
```

The benefit of `sum()` is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

```
In[17]: # how many values less than 6 in each row?
        np.sum(x < 6, axis=1)

Out[17]: array([4, 2, 2])
```

This counts the number of values less than 6 in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) `np.any()` or `np.all()`:

```
In[18]: # are there any values greater than 8?
        np.any(x > 8)

Out[18]: True

In[19]: # are there any values less than zero?
        np.any(x < 0)

Out[19]: False

In[20]: # are all values less than 10?
        np.all(x < 10)

Out[20]: True

In[21]: # are all values equal to 6?
        np.all(x == 6)

Out[21]: False
```

`np.all()` and `np.any()` can be used along particular axes as well. For example:

```
In[22]: # are all values in each row less than 8?
        np.all(x < 8, axis=1)

Out[22]: array([ True, False,  True], dtype=bool)
```

Here all the elements in the first and third rows are less than 8, while this is not the case for the second row.

Finally, a quick warning: as mentioned in "Aggregations: Min, Max, and Everything in Between" on page 58, Python has built-in `sum()`, `any()`, and `all()` functions. These have a different syntax than the NumPy versions, and in particular will fail or produce unintended results when used on multidimensional arrays. Be sure that you are using `np.sum()`, `np.any()`, and `np.all()` for these examples!

### Boolean operators

We've already seen how we might count, say, all days with rain less than four inches, or all days with rain greater than two inches. But what if we want to know about all days with rain less than four inches *and* greater than one inch? This is accomplished through Python's *bitwise logic operators*, `&`, `|`, `^`, and `~`. Like with the standard arithmetic operators, NumPy overloads these as ufuncs that work element-wise on (usually Boolean) arrays.

For example, we can address this sort of compound question as follows:

```
In[23]: np.sum((inches > 0.5) & (inches < 1))

Out[23]: 29
```

So we see that there are 29 days with rainfall between 0.5 and 1.0 inches.

Note that the parentheses here are important—because of operator precedence rules, with parentheses removed this expression would be evaluated as follows, which results in an error:

```
inches > (0.5 & inches) < 1
```

Using the equivalence of *A AND B* and *NOT (A OR B)* (which you may remember if you've taken an introductory logic course), we can compute the same result in a different manner:

```
In[24]: np.sum(~( (inches <= 0.5) | (inches >= 1) ))

Out[24]: 29
```

Combining comparison operators and Boolean operators on arrays can lead to a wide range of efficient logical operations.

The following table summarizes the bitwise Boolean operators and their equivalent ufuncs:

| Operator | Equivalent ufunc |
|---|---|
| `&` | `np.bitwise_and` |
| `\|` | `np.bitwise_or` |
| `^` | `np.bitwise_xor` |
| `~` | `np.bitwise_not` |

Using these tools, we might start to answer the types of questions we have about our weather data. Here are some examples of results we can compute when combining masking with aggregations:

```
In[25]: print("Number days without rain:      ", np.sum(inches == 0))
        print("Number days with rain:         ", np.sum(inches != 0))
        print("Days with more than 0.5 inches:", np.sum(inches > 0.5))
        print("Rainy days with < 0.1 inches  :", np.sum((inches > 0) &
                                                        (inches < 0.2)))

Number days without rain:       215
Number days with rain:          150
Days with more than 0.5 inches: 37
Rainy days with < 0.1 inches  : 75
```

## Boolean Arrays as Masks

In the preceding section, we looked at aggregates computed directly on Boolean arrays. A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Returning to our `x` array from before, suppose we want an array of all values in the array that are less than, say, 5: