```
In[2]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn; seaborn.set()  # set plot styles
```

```
In[3]: plt.hist(inches, 40);
```

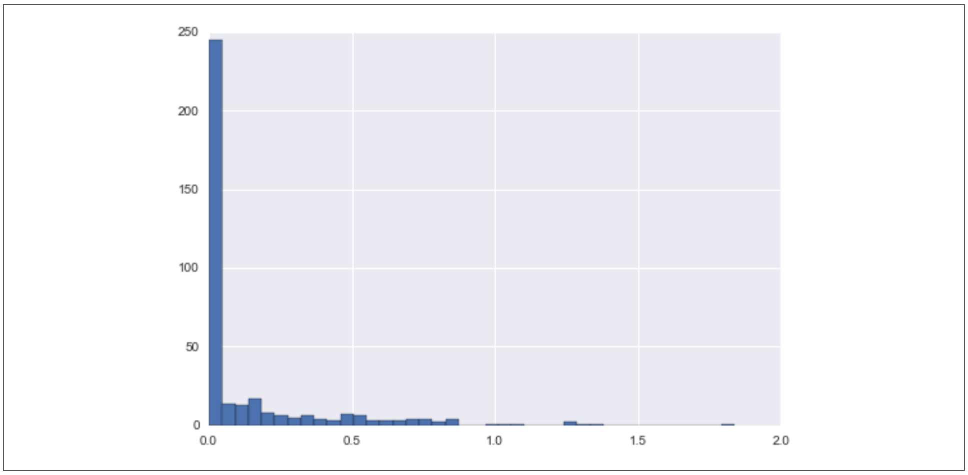

Figure 2-6. Histogram of 2014 rainfall in Seattle

This histogram gives us a general idea of what the data looks like: despite its reputation, the vast majority of days in Seattle saw near zero measured rainfall in 2014. But this doesn't do a good job of conveying some information we'd like to see: for example, how many rainy days were there in the year? What is the average precipitation on those rainy days? How many days were there with more than half an inch of rain?

Digging into the data

One approach to this would be to answer these questions by hand: loop through the data, incrementing a counter each time we see values in some desired range. For reasons discussed throughout this chapter, such an approach is very inefficient, both from the standpoint of time writing code and time computing the result. We saw in "Computation on NumPy Arrays: Universal Functions" on page 50 that NumPy's ufuncs can be used in place of loops to do fast element-wise arithmetic operations on arrays; in the same way, we can use other ufuncs to do element-wise *comparisons* over arrays, and we can then manipulate the results to answer the questions we have. We'll leave the data aside for right now, and discuss some general tools in NumPy to use *masking* to quickly answer these types of questions.

Comparison Operators as ufuncs

In "Computation on NumPy Arrays: Universal Functions" on page 50 we introduced ufuncs, and focused in particular on arithmetic operators. We saw that using +, -, *, /, and others on arrays leads to element-wise operations. NumPy also implements comparison operators such as < (less than) and > (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available:

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

In[5]: x < 3  # less than

Out[5]: array([ True,  True, False, False, False], dtype=bool)

In[6]: x > 3  # greater than

Out[6]: array([False, False, False,  True,  True], dtype=bool)

In[7]: x <= 3  # less than or equal

Out[7]: array([ True,  True,  True, False, False], dtype=bool)

In[8]: x >= 3  # greater than or equal

Out[8]: array([False, False,  True,  True,  True], dtype=bool)

In[9]: x != 3  # not equal

Out[9]: array([ True,  True, False,  True,  True], dtype=bool)

In[10]: x == 3  # equal

Out[10]: array([False, False,  True, False, False], dtype=bool)
```

It is also possible to do an element-by-element comparison of two arrays, and to include compound expressions:

```
In[11]: (2 * x) == (x ** 2)

Out[11]: array([False,  True, False, False, False], dtype=bool)
```

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`. A summary of the comparison operators and their equivalent ufunc is shown here:

Operator	Equivalent ufunc
`==`	`np.equal`
`!=`	`np.not_equal`
`<`	`np.less`
`<=`	`np.less_equal`
`>`	`np.greater`
`>=`	`np.greater_equal`

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```
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.