```
In[19]: x[i] -= 10
        print(x)

[ 0 89 89  3 89  5  6  7 89  9]
```

Notice, though, that repeated indices with these operations can cause some potentially unexpected results. Consider the following:

```
In[20]: x = np.zeros(10)
        x[[0, 0]] = [4, 6]
        print(x)

[ 6.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Where did the 4 go? The result of this operation is to first assign `x[0] = 4`, followed by `x[0] = 6`. The result, of course, is that `x[0]` contains the value 6.

Fair enough, but consider this operation:

```
In[21]: i = [2, 3, 3, 4, 4, 4]
        x[i] += 1
        x

Out[21]: array([ 6.,  0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.])
```

You might expect that `x[3]` would contain the value 2, and `x[4]` would contain the value 3, as this is how many times each index is repeated. Why is this not the case? Conceptually, this is because `x[i] += 1` is meant as a shorthand of `x[i] = x[i] + 1`. `x[i] + 1` is evaluated, and then the result is assigned to the indices in `x`. With this in mind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather nonintuitive results.

So what if you want the other behavior where the operation is repeated? For this, you can use the `at()` method of ufuncs (available since NumPy 1.8), and do the following:

```
In[22]: x = np.zeros(10)
        np.add.at(x, i, 1)
        print(x)

[ 0.  0.  1.  2.  3.  0.  0.  0.  0.  0.]
```

The `at()` method does an in-place application of the given operator at the specified indices (here, `i`) with the specified value (here, 1). Another method that is similar in spirit is the `reduceat()` method of ufuncs, which you can read about in the NumPy documentation.

Example: Binning Data

You can use these ideas to efficiently bin data to create a histogram by hand. For example, imagine we have 1,000 values and would like to quickly find where they fall within an array of bins. We could compute it using `ufunc.at` like this:

```
In[23]: np.random.seed(42)
        x = np.random.randn(100)

        # compute a histogram by hand
        bins = np.linspace(-5, 5, 20)
        counts = np.zeros_like(bins)

        # find the appropriate bin for each x
        i = np.searchsorted(bins, x)

        # add 1 to each of these bins
        np.add.at(counts, i, 1)
```

The counts now reflect the number of points within each bin—in other words, a histogram (Figure 2-9):

```
In[24]: # plot the results
        plt.plot(bins, counts, linestyle='steps');
```

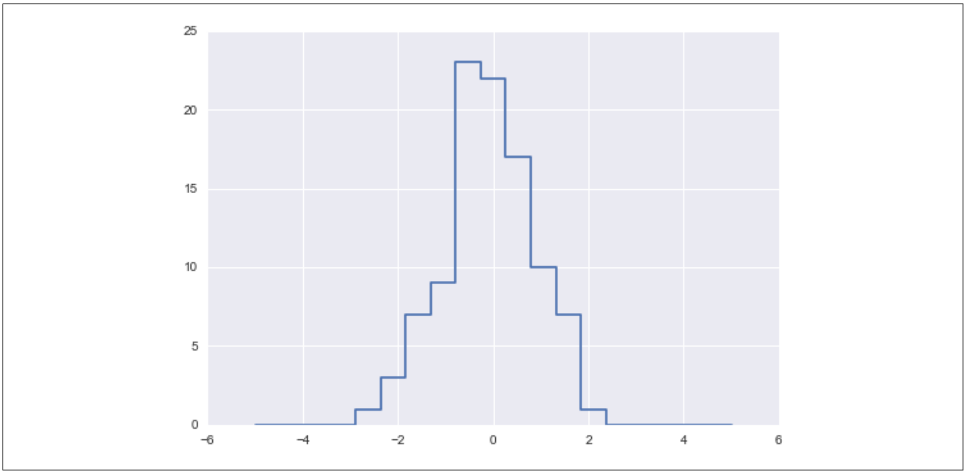

Figure 2-9. A histogram computed by hand

Of course, it would be silly to have to do this each time you want to plot a histogram. This is why Matplotlib provides the `plt.hist()` routine, which does the same in a single line:

```
plt.hist(x, bins, histtype='step');
```

This function will create a nearly identical plot to the one seen here. To compute the binning, Matplotlib uses the `np.histogram` function, which does a very similar computation to what we did before. Let's compare the two here:

```
In[25]: print("NumPy routine:")
        %timeit counts, edges = np.histogram(x, bins)
```

```
print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy routine:
10000 loops, best of 3: 97.6 µs per loop
Custom routine:
10000 loops, best of 3: 19.5 µs per loop
```

Our own one-line algorithm is several times faster than the optimized algorithm in NumPy! How can this be? If you dig into the `np.histogram` source code (you can do this in IPython by typing **`np.histogram??`**), you'll see that it's quite a bit more involved than the simple search-and-count that we've done; this is because NumPy's algorithm is more flexible, and particularly is designed for better performance when the number of data points becomes large:

```
In[26]: x = np.random.randn(1000000)
        print("NumPy routine:")
        %timeit counts, edges = np.histogram(x, bins)

        print("Custom routine:")
        %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy routine:
10 loops, best of 3: 68.7 ms per loop
Custom routine:
10 loops, best of 3: 135 ms per loop
```

What this comparison shows is that algorithmic efficiency is almost never a simple question. An algorithm efficient for large datasets will not always be the best choice for small datasets, and vice versa (see "Big-O Notation" on page 92). But the advantage of coding this algorithm yourself is that with an understanding of these basic methods, you could use these building blocks to extend this to do some very interesting custom behaviors. The key to efficiently using Python in data-intensive applications is knowing about general convenience routines like `np.histogram` and when they're appropriate, but also knowing how to make use of lower-level functionality when you need more pointed behavior.

Sorting Arrays

Up to this point we have been concerned mainly with tools to access and operate on array data with NumPy. This section covers algorithms related to sorting values in NumPy arrays. These algorithms are a favorite topic in introductory computer science courses: if you've ever taken one, you probably have had dreams (or, depending on your temperament, nightmares) about *insertion sorts*, *selection sorts*, *merge sorts*, *quick sorts*, *bubble sorts*, and many, many more. All are means of accomplishing a similar task: sorting the values in a list or array.