CHAPTER 3

Visualizing Data

I believe that visualization is one of the most powerful means of achieving personal goals.

—Harvey Mackay

A fundamental part of the data scientist's toolkit is data visualization. Although it is very easy to create visualizations, it's much harder to produce *good* ones.

There are two primary uses for data visualization:

- To *explore* data
- To *communicate* data

In this chapter, we will concentrate on building the skills that you'll need to start exploring your own data and to produce the visualizations we'll be using throughout the rest of the book. Like most of our chapter topics, data visualization is a rich field of study that deserves its own book. Nonetheless, we'll try to give you a sense of what makes for a good visualization and what doesn't.

matplotlib

A wide variety of tools exists for visualizing data. We will be using the `matplotlib` library, which is widely used (although sort of showing its age). If you are interested in producing elaborate interactive visualizations for the Web, it is likely not the right choice, but for simple bar charts, line charts, and scatterplots, it works pretty well.

In particular, we will be using the `matplotlib.pyplot` module. In its simplest use, `pyplot` maintains an internal state in which you build up a visualization step by step. Once you're done, you can save it (with `savefig()`) or display it (with `show()`).

For example, making simple plots (like Figure 3-1) is pretty simple:

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# add a title
plt.title("Nominal GDP")

# add a label to the y-axis
plt.ylabel("Billions of $")
plt.show()
```

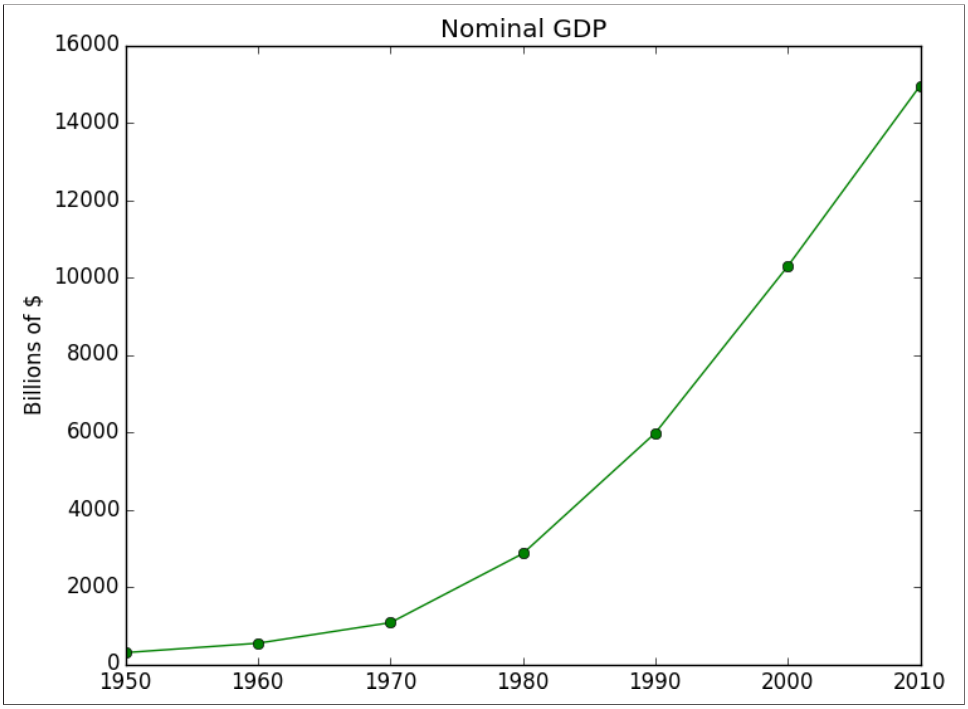

Figure 3-1. A simple line chart

Making plots that look publication-quality good is more complicated and beyond the scope of this chapter. There are many ways you can customize your charts with (for example) axis labels, line styles, and point markers. Rather than attempt a comprehensive treatment of these options, we'll just use (and call attention to) some of them in our examples.

Although we won't be using much of this functionality, matplotlib is capable of producing complicated plots within plots, sophisticated formatting, and interactive visualizations. Check out its documentation if you want to go deeper than we do in this book.

Bar Charts

A bar chart is a good choice when you want to show how some quantity varies among some *discrete* set of items. For instance, Figure 3-2 shows how many Academy Awards were won by each of a variety of movies:

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

# bars are by default width 0.8, so we'll add 0.1 to the left coordinates
# so that each bar is centered
xs = [i + 0.1 for i, _ in enumerate(movies)]

# plot bars with left x-coordinates [xs], heights [num_oscars]
plt.bar(xs, num_oscars)

plt.ylabel("# of Academy Awards")
plt.title("My Favorite Movies")

# label x-axis with movie names at bar centers
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)

plt.show()
```

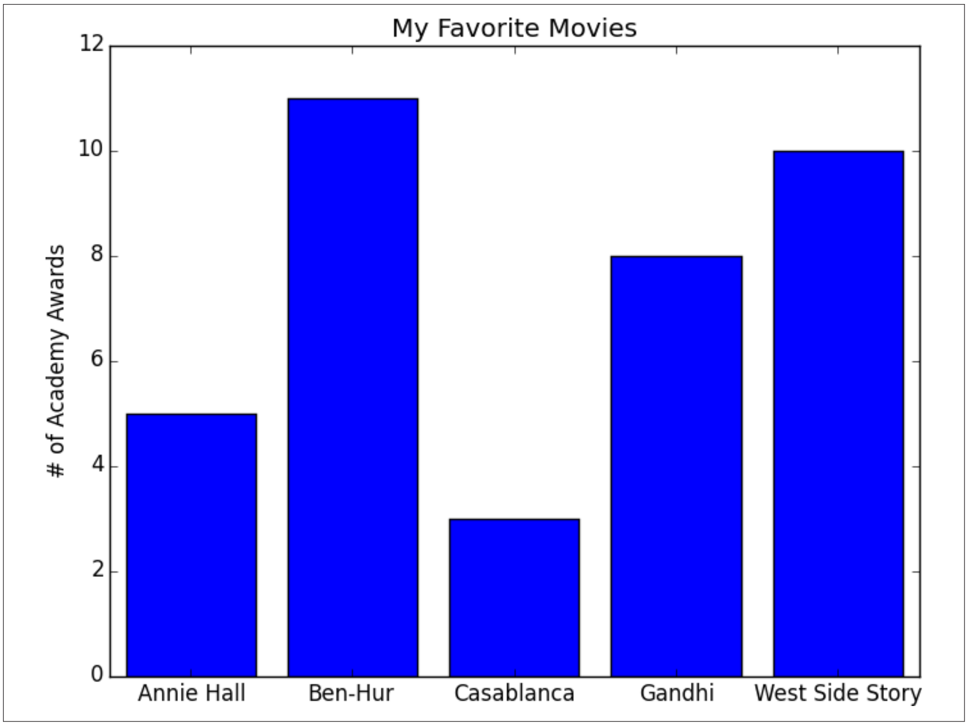

Figure 3-2. A simple bar chart

A bar chart can also be a good choice for plotting histograms of bucketed numeric values, in order to visually explore how the values are *distributed*, as in Figure 3-3:

```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)

plt.bar([x - 4 for x in histogram.keys()], # shift each bar to the left by 4
        histogram.values(),                # give each bar its correct height
        8)                                 # give each bar a width of 8

plt.axis([-5, 105, 0, 5])                  # x-axis from -5 to 105,
                                           # y-axis from 0 to 5

plt.xticks([10 * i for i in range(11)])    # x-axis labels at 0, 10, ..., 100
plt.xlabel("Decile")
plt.ylabel("# of Students")
plt.title("Distribution of Exam 1 Grades")
plt.show()
```

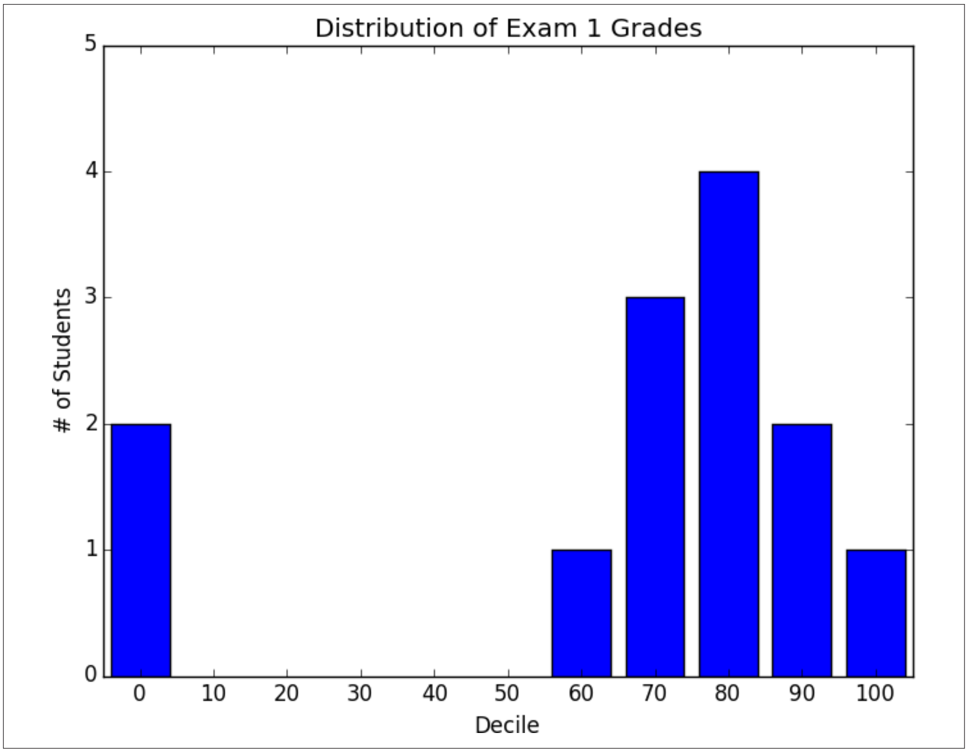

Figure 3-3. Using a bar chart for a histogram

The third argument to `plt.bar` specifies the bar width. Here we chose a width of 8 (which leaves a small gap between bars, since our buckets have width 10). And we shifted the bar left by 4, so that (for example) the "80" bar has its left and right sides at 76 and 84, and (hence) its center at 80.

The call to `plt.axis` indicates that we want the x-axis to range from -5 to 105 (so that the "0" and "100" bars are fully shown), and that the y-axis should range from 0 to 5. And the call to `plt.xticks` puts x-axis labels at 0, 10, 20, …, 100.

Be judicious when using `plt.axis()`. When creating bar charts it is considered especially bad form for your y-axis not to start at 0, since this is an easy way to mislead people (Figure 3-4):

```
mentions = [500, 505]
years = [2013, 2014]

plt.bar([2012.6, 2013.6], mentions, 0.8)
plt.xticks(years)
plt.ylabel("# of times I heard someone say 'data science'")

# if you don't do this, matplotlib will label the x-axis 0, 1
# and then add a +2.013e3 off in the corner (bad matplotlib!)
plt.ticklabel_format(useOffset=False)

# misleading y-axis only shows the part above 500
plt.axis([2012.5,2014.5,499,506])
plt.title("Look at the 'Huge' Increase!")
plt.show()
```

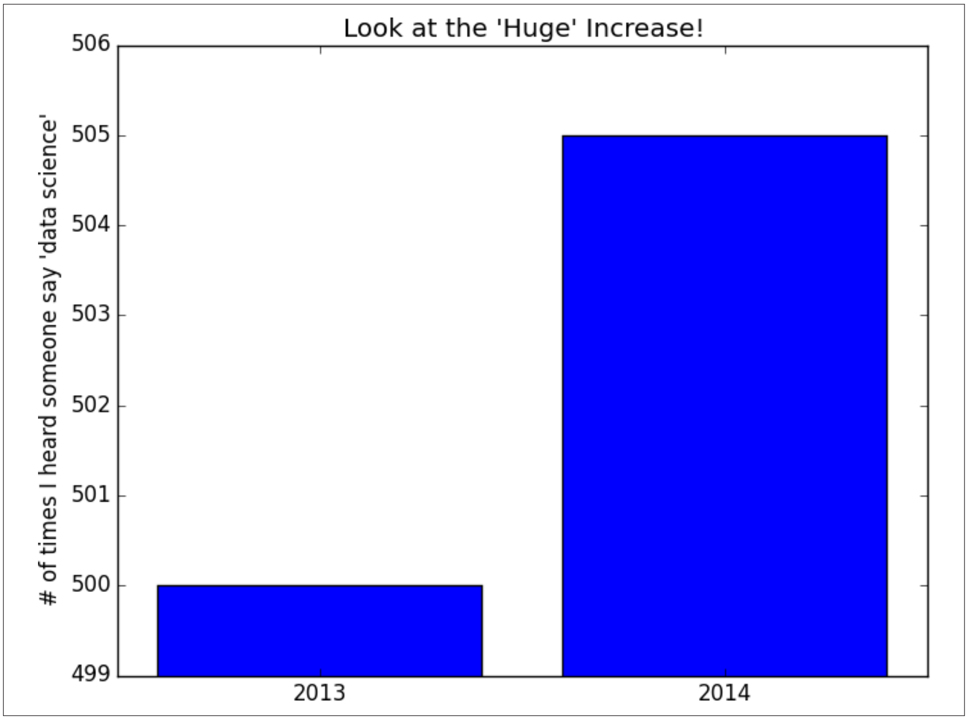

Figure 3-4. A chart with a misleading y-axis

In Figure 3-5, we use more-sensible axes, and it looks far less impressive:

```
plt.axis([2012.5,2014.5,0,550])
plt.title("Not So Huge Anymore")
plt.show()
```

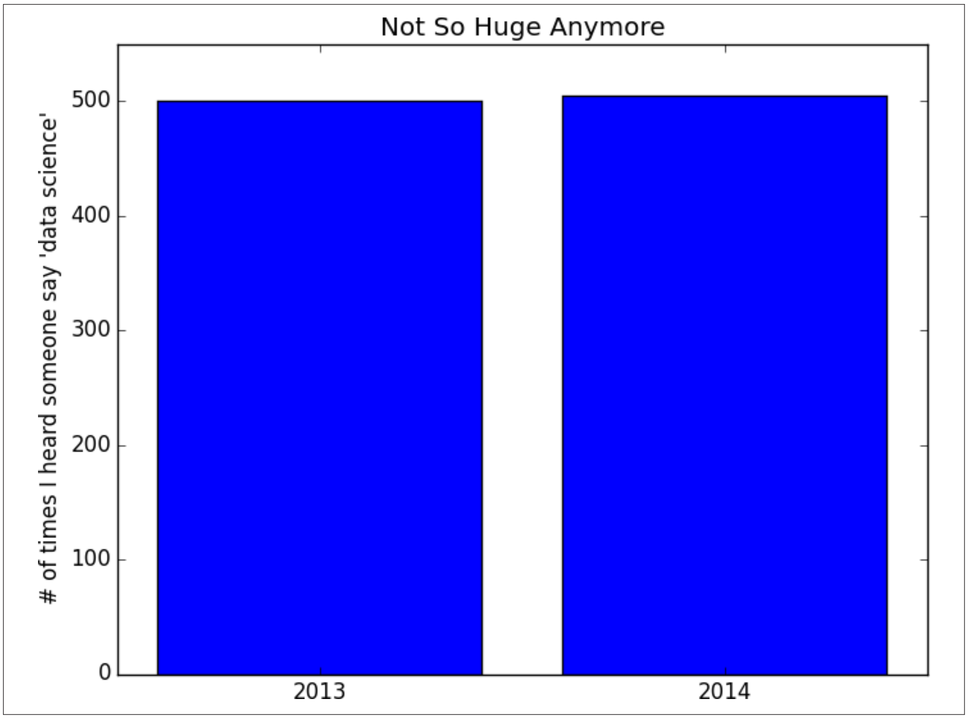

Figure 3-5. The same chart with a nonmisleading y-axis

Line Charts

As we saw already, we can make line charts using `plt.plot()`. These are a good choice for showing *trends*, as illustrated in Figure 3-6:

```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# we can make multiple calls to plt.plot
# to show multiple series on the same chart
plt.plot(xs, variance,     'g-',  label='variance')    # green solid line
plt.plot(xs, bias_squared, 'r-.', label='bias^2')      # red dot-dashed line
plt.plot(xs, total_error,  'b:',  label='total error') # blue dotted line

# because we've assigned labels to each series
# we can get a legend for free
# loc=9 means "top center"
plt.legend(loc=9)
plt.xlabel("model complexity")
plt.title("The Bias-Variance Tradeoff")
plt.show()
```

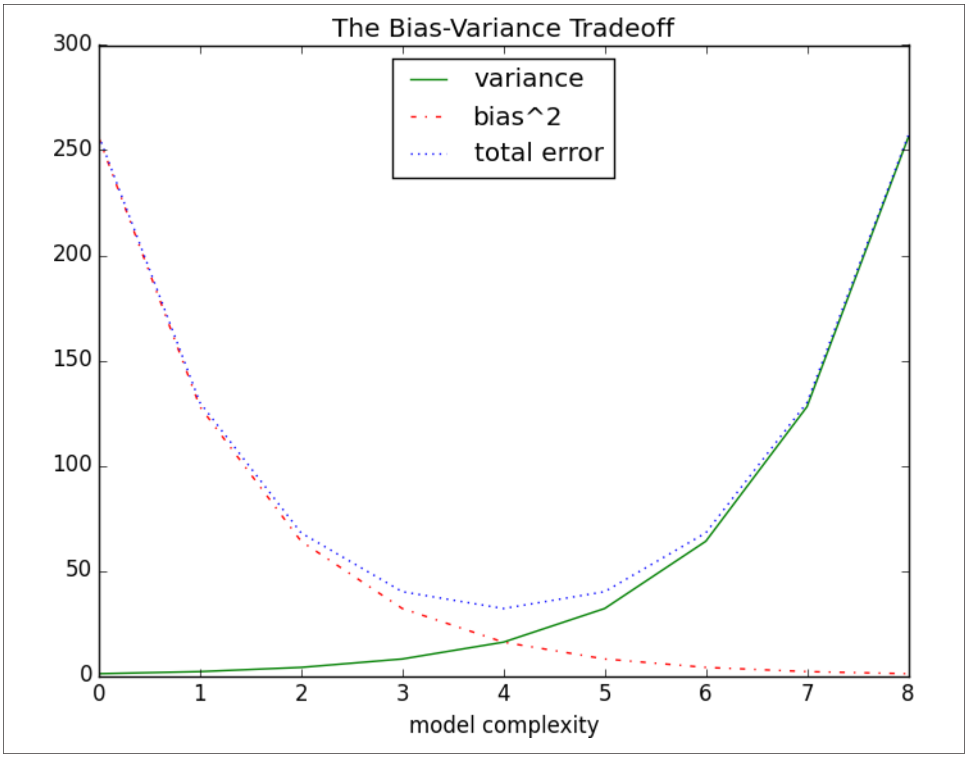

Figure 3-6. Several line charts with a legend

Scatterplots

A scatterplot is the right choice for visualizing the relationship between two paired sets of data. For example, Figure 3-7 illustrates the relationship between the number of friends your users have and the number of minutes they spend on the site every day:

```
friends = [ 70,  65,  72,  63,  71,  64,  60,  64,  67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels =  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

plt.scatter(friends, minutes)

# label each point
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
        xy=(friend_count, minute_count), # put the label with its point
        xytext=(5, -5),                  # but slightly offset
        textcoords='offset points')

plt.title("Daily Minutes vs. Number of Friends")
plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site")
plt.show()
```

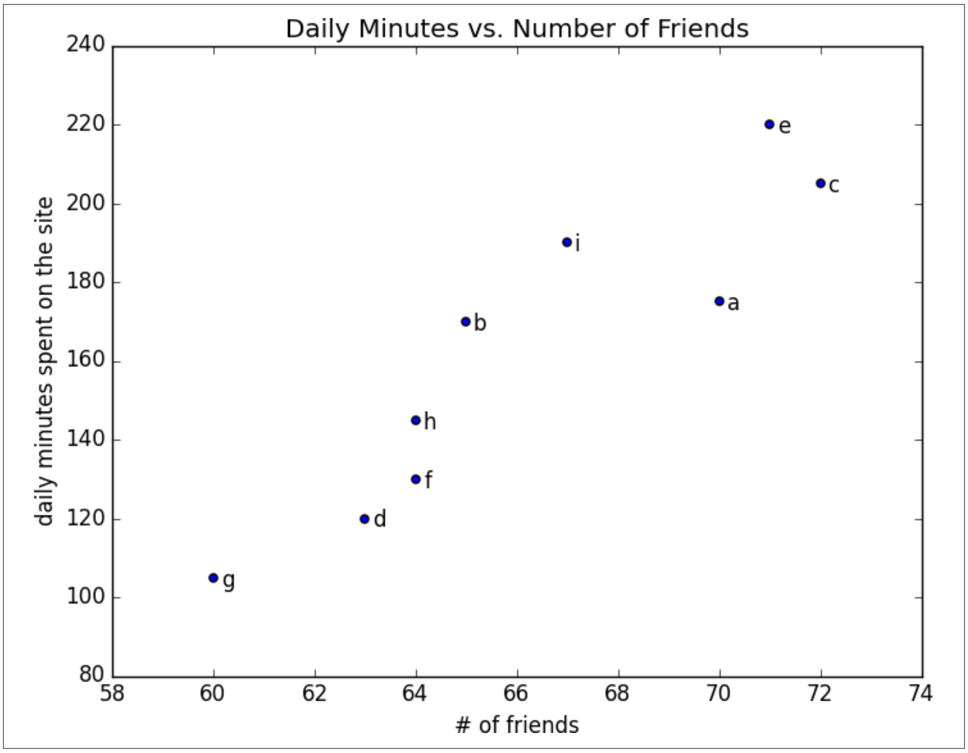

Figure 3-7. A scatterplot of friends and time on the site

If you're scattering comparable variables, you might get a misleading picture if you let `matplotlib` choose the scale, as in Figure 3-8:

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades)
plt.title("Axes Aren't Comparable")
plt.xlabel("test 1 grade")
plt.ylabel("test 2 grade")
plt.show()
```

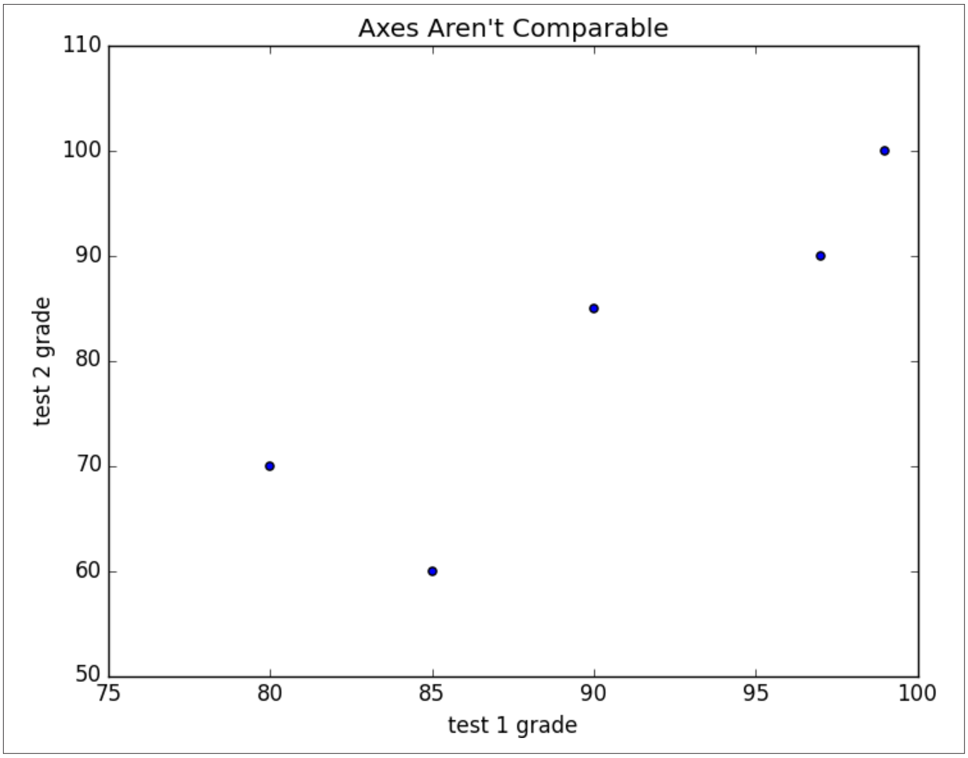

Figure 3-8. A scatterplot with uncomparable axes

If we include a call to `plt.axis("equal")`, the plot (Figure 3-9) more accurately shows that most of the variation occurs on test 2.

That's enough to get you started doing visualization. We'll learn much more about visualization throughout the book.

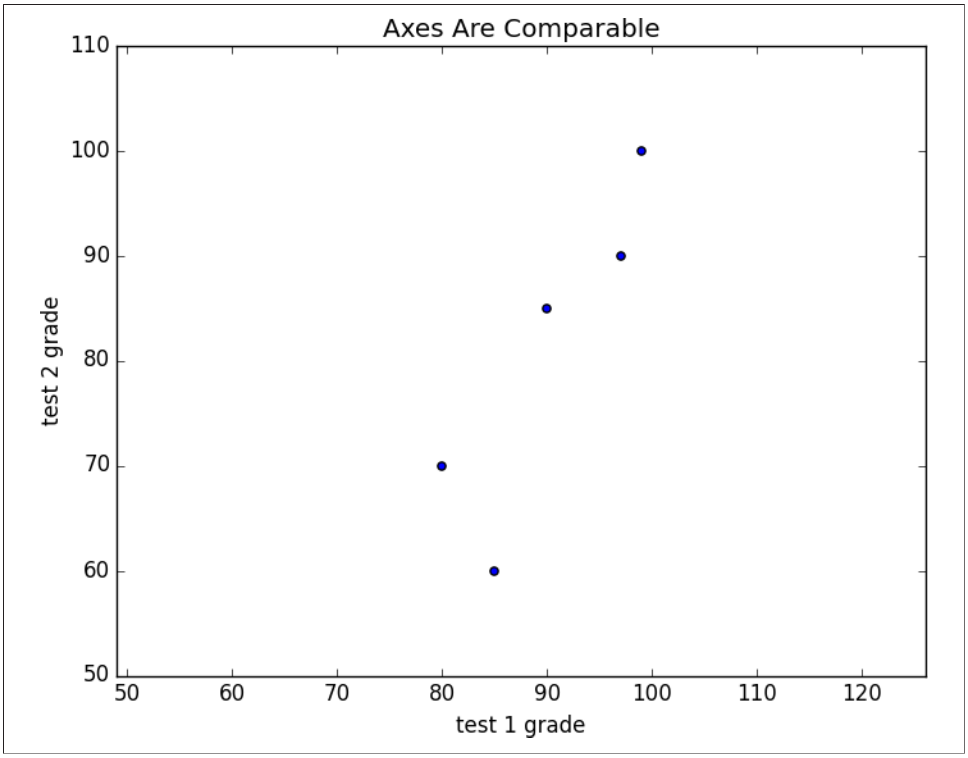

Figure 3-9. The same scatterplot with equal axes

For Further Exploration

- seaborn is built on top of `matplotlib` and allows you to easily produce prettier (and more complex) visualizations.
- D3.js is a JavaScript library for producing sophisticated interactive visualizations for the web. Although it is not in Python, it is both trendy and widely used, and it is well worth your while to be familiar with it.
- Bokeh is a newer library that brings D3-style visualizations into Python.
- ggplot is a Python port of the popular R library `ggplot2`, which is widely used for creating "publication quality" charts and graphics. It's probably most interesting if you're already an avid `ggplot2` user, and possibly a little opaque if you're not.

CHAPTER 4

Linear Algebra

Is there anything more useless or less useful than Algebra?
—Billy Connolly

Linear algebra is the branch of mathematics that deals with *vector spaces*. Although I can't hope to teach you linear algebra in a brief chapter, it underpins a large number of data science concepts and techniques, which means I owe it to you to at least try. What we learn in this chapter we'll use heavily throughout the rest of the book.

Vectors

Abstractly, *vectors* are objects that can be added together (to form new vectors) and that can be multiplied by *scalars* (i.e., numbers), also to form new vectors.

Concretely (for us), vectors are points in some finite-dimensional space. Although you might not think of your data as vectors, they are a good way to represent numeric data.

For example, if you have the heights, weights, and ages of a large number of people, you can treat your data as three-dimensional vectors `(height, weight, age)`. If you're teaching a class with four exams, you can treat student grades as four-dimensional vectors `(exam1, exam2, exam3, exam4)`.

The simplest from-scratch approach is to represent vectors as lists of numbers. A list of three numbers corresponds to a vector in three-dimensional space, and vice versa:

```
height_weight_age = [70,  # inches,
                     170, # pounds,
                     40 ] # years

grades = [95,   # exam1
          80,   # exam2
```

```
              75,    # exam3
              62 ]   # exam4
```

One problem with this approach is that we will want to perform *arithmetic* on vectors. Because Python lists aren't vectors (and hence provide no facilities for vector arithmetic), we'll need to build these arithmetic tools ourselves. So let's start with that.

To begin with, we'll frequently need to add two vectors. Vectors add *componentwise*. This means that if two vectors `v` and `w` are the same length, their sum is just the vector whose first element is `v[0] + w[0]`, whose second element is `v[1] + w[1]`, and so on. (If they're not the same length, then we're not allowed to add them.)

For example, adding the vectors `[1, 2]` and `[2, 1]` results in `[1 + 2, 2 + 1]` or `[3, 3]`, as shown in Figure 4-1.

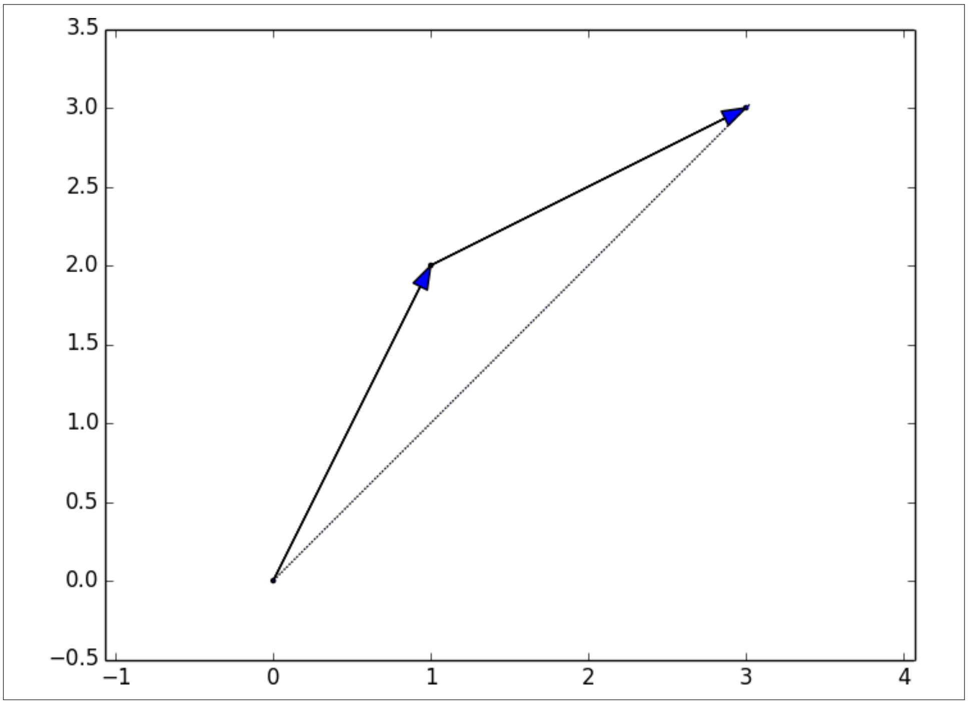

Figure 4-1. Adding two vectors

We can easily implement this by `zip`-ing the vectors together and using a list comprehension to add the corresponding elements:

```
def vector_add(v, w):
    """adds corresponding elements"""
    return [v_i + w_i
            for v_i, w_i in zip(v, w)]
```

Similarly, to subtract two vectors we just subtract corresponding elements:

```
def vector_subtract(v, w):
    """subtracts corresponding elements"""
    return [v_i - w_i
            for v_i, w_i in zip(v, w)]
```

We'll also sometimes want to componentwise sum a list of vectors. That is, create a new vector whose first element is the sum of all the first elements, whose second element is the sum of all the second elements, and so on. The easiest way to do this is by adding one vector at a time:

```
def vector_sum(vectors):
    """sums all corresponding elements"""
    result = vectors[0]                         # start with the first vector
    for vector in vectors[1:]:                  # then loop over the others
        result = vector_add(result, vector)     # and add them to the result
    return result
```

If you think about it, we are just `reduce`-ing the list of vectors using `vector_add`, which means we can rewrite this more briefly using higher-order functions:

```
def vector_sum(vectors):
    return reduce(vector_add, vectors)
```

or even:

```
vector_sum = partial(reduce, vector_add)
```

although this last one is probably more clever than helpful.

We'll also need to be able to multiply a vector by a scalar, which we do simply by multiplying each element of the vector by that number:

```
def scalar_multiply(c, v):
    """c is a number, v is a vector"""
    return [c * v_i for v_i in v]
```

This allows us to compute the componentwise means of a list of (same-sized) vectors:

```
def vector_mean(vectors):
    """compute the vector whose ith element is the mean of the
    ith elements of the input vectors"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))
```

A less obvious tool is the *dot product*. The dot product of two vectors is the sum of their componentwise products:

```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i
               for v_i, w_i in zip(v, w))
```

The dot product measures how far the vector *v* extends in the *w* direction. For example, if `w = [1, 0]` then `dot(v, w)` is just the first component of `v`. Another way of saying this is that it's the length of the vector you'd get if you *projected* *v* onto *w* (Figure 4-2).

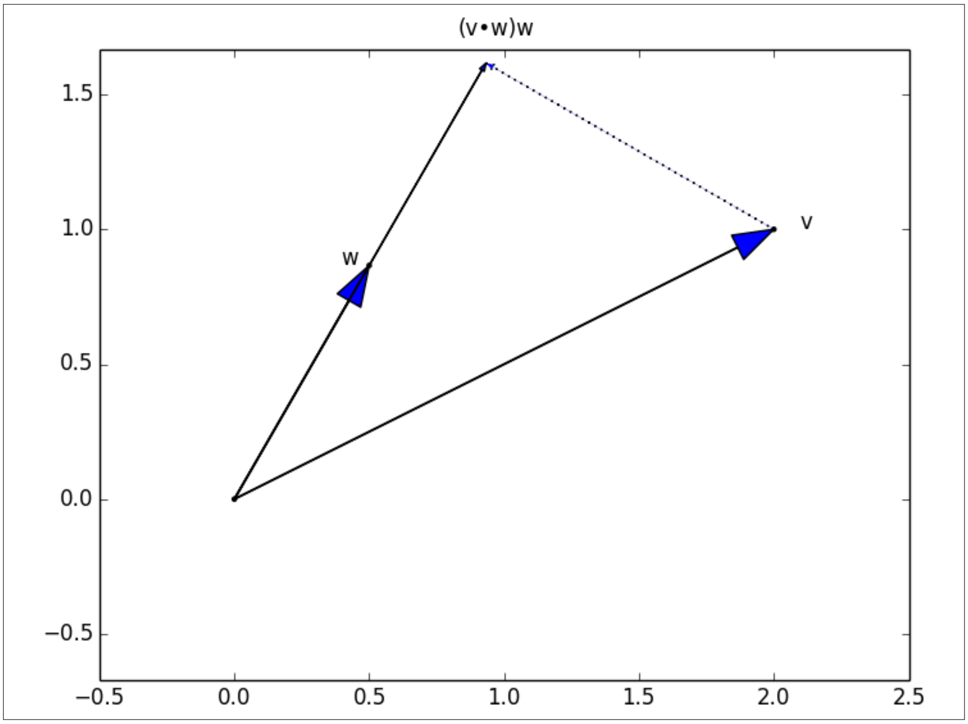

Figure 4-2. The dot product as vector projection

Using this, it's easy to compute a vector's *sum of squares*:

```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)
```

Which we can use to compute its *magnitude* (or length):

```
import math

def magnitude(v):
    return math.sqrt(sum_of_squares(v))   # math.sqrt is square root function
```

We now have all the pieces we need to compute the distance between two vectors, defined as:

$$\sqrt{(v_1 - w_1)^2 + ... + (v_n - w_n)^2}$$

```
def squared_distance(v, w):
    """(v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
    return sum_of_squares(vector_subtract(v, w))

def distance(v, w):
   return math.sqrt(squared_distance(v, w))
```

Which is possibly clearer if we write it as (the equivalent):

```
def distance(v, w):
    return magnitude(vector_subtract(v, w))
```

That should be plenty to get us started. We'll be using these functions heavily throughout the book.

Using lists as vectors is great for exposition but terrible for performance.

In production code, you would want to use the NumPy library, which includes a high-performance array class with all sorts of arithmetic operations included.

Matrices

A *matrix* is a two-dimensional collection of numbers. We will represent matrices as `lists` of `lists`, with each inner list having the same size and representing a *row* of the matrix. If `A` is a matrix, then `A[i][j]` is the element in the *i*th row and the *j*th column. Per mathematical convention, we will typically use capital letters to represent matrices. For example:

```
A = [[1, 2, 3],  # A has 2 rows and 3 columns
     [4, 5, 6]]

B = [[1, 2],     # B has 3 rows and 2 columns
     [3, 4],
     [5, 6]]
```

In mathematics, you would usually name the first row of the matrix "row 1" and the first column "column 1." Because we're representing matrices with Python `lists`, which are zero-indexed, we'll call the first row of a matrix "row 0" and the first column "column 0."

Given this list-of-lists representation, the matrix `A` has `len(A)` rows and `len(A[0])` columns, which we consider its `shape`:

```
def shape(A):
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0   # number of elements in first row
    return num_rows, num_cols
```

If a matrix has n rows and k columns, we will refer to it as a $n \times k$ matrix. We can (and sometimes will) think of each row of a $n \times k$ matrix as a vector of length k, and each column as a vector of length n:

```
def get_row(A, i):
    return A[i]             # A[i] is already the ith row

def get_column(A, j):
    return [A_i[j]          # jth element of row A_i
            for A_i in A]   # for each row A_i
```

We'll also want to be able to create a matrix given its shape and a function for generating its elements. We can do this using a nested list comprehension:

```
def make_matrix(num_rows, num_cols, entry_fn):
    """returns a num_rows x num_cols matrix
    whose (i,j)th entry is entry_fn(i, j)"""
    return [[entry_fn(i, j)             # given i, create a list
             for j in range(num_cols)]  #   [entry_fn(i, 0), ... ]
            for i in range(num_rows)]   # create one list for each i
```

Given this function, you could make a 5 × 5 *identity matrix* (with 1s on the diagonal and 0s elsewhere) with:

```
def is_diagonal(i, j):
    """1's on the 'diagonal', 0's everywhere else"""
    return 1 if i == j else 0

identity_matrix = make_matrix(5, 5, is_diagonal)

# [[1, 0, 0, 0, 0],
#  [0, 1, 0, 0, 0],
#  [0, 0, 1, 0, 0],
#  [0, 0, 0, 1, 0],
#  [0, 0, 0, 0, 1]]
```

Matrices will be important to us for several reasons.

First, we can use a matrix to represent a data set consisting of multiple vectors, simply by considering each vector as a row of the matrix. For example, if you had the heights, weights, and ages of 1,000 people you could put them in a $1,000 \times 3$ matrix:

```
data = [[70, 170, 40],
        [65, 120, 26],
        [77, 250, 19],
        # ....
       ]
```

Second, as we'll see later, we can use an $n \times k$ matrix to represent a linear function that maps k-dimensional vectors to n-dimensional vectors. Several of our techniques and concepts will involve such functions.

Third, matrices can be used to represent binary relationships. In Chapter 1, we represented the edges of a network as a collection of pairs `(i, j)`. An alternative representation would be to create a matrix *A* such that `A[i][j]` is 1 if nodes *i* and *j* are connected and 0 otherwise.

Recall that before we had:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

We could also represent this as:

```
     #     user 0  1  2  3  4  5  6  7  8  9
     #
friendships = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0], # user 0
               [1, 0, 1, 1, 0, 0, 0, 0, 0, 0], # user 1
               [1, 1, 0, 1, 0, 0, 0, 0, 0, 0], # user 2
               [0, 1, 1, 0, 1, 0, 0, 0, 0, 0], # user 3
               [0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # user 4
               [0, 0, 0, 0, 1, 0, 1, 1, 0, 0], # user 5
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # user 6
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # user 7
               [0, 0, 0, 0, 0, 0, 1, 1, 0, 1], # user 8
               [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]] # user 9
```

If there are very few connections, this is a much more inefficient representation, since you end up having to store a lot of zeroes. However, with the matrix representation it is much quicker to check whether two nodes are connected—you just have to do a matrix lookup instead of (potentially) inspecting every edge:

```
friendships[0][2] == 1   # True, 0 and 2 are friends
friendships[0][8] == 1   # False, 0 and 8 are not friends
```

Similarly, to find the connections a node has, you only need to inspect the column (or the row) corresponding to that node:

```
friends_of_five = [i                                          # only need
                   for i, is_friend in enumerate(friendships[5])  # to look at
                   if is_friend]                              # one row
```

Previously we added a list of connections to each node object to speed up this process, but for a large, evolving graph that would probably be too expensive and difficult to maintain.

We'll revisit matrices throughout the book.

For Further Exploration

- Linear algebra is widely used by data scientists (frequently implicitly, and not infrequently by people who don't understand it). It wouldn't be a bad idea to read a textbook. You can find several freely available online:

  - Linear Algebra, from UC Davis
  - Linear Algebra, from Saint Michael's College
  - If you are feeling adventurous, Linear Algebra Done Wrong is a more advanced introduction
- All of the machinery we built here you get for free if you use NumPy. (You get a lot more too.)

CHAPTER 5
Statistics

Facts are stubborn, but statistics are more pliable.
—Mark Twain

Statistics refers to the mathematics and techniques with which we understand data. It is a rich, enormous field, more suited to a shelf (or room) in a library rather than a chapter in a book, and so our discussion will necessarily not be a deep one. Instead, I'll try to teach you just enough to be dangerous, and pique your interest just enough that you'll go off and learn more.

Describing a Single Set of Data

Through a combination of word-of-mouth and luck, DataSciencester has grown to dozens of members, and the VP of Fundraising asks you for some sort of description of how many friends your members have that he can include in his elevator pitches.

Using techniques from Chapter 1, you are easily able to produce this data. But now you are faced with the problem of how to *describe* it.

One obvious description of any data set is simply the data itself:

```
num_friends = [100, 49, 41, 40, 25,
               # ... and lots more
              ]
```

For a small enough data set this might even be the best description. But for a larger data set, this is unwieldy and probably opaque. (Imagine staring at a list of 1 million numbers.) For that reason we use statistics to distill and communicate relevant features of our data.

As a first approach you put the friend counts into a histogram using `Counter` and `plt.bar()` (Figure 5-1):

```
friend_counts = Counter(num_friends)
xs = range(101)                         # largest value is 100
ys = [friend_counts[x] for x in xs]     # height is just # of friends
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Histogram of Friend Counts")
plt.xlabel("# of friends")
plt.ylabel("# of people")
plt.show()
```

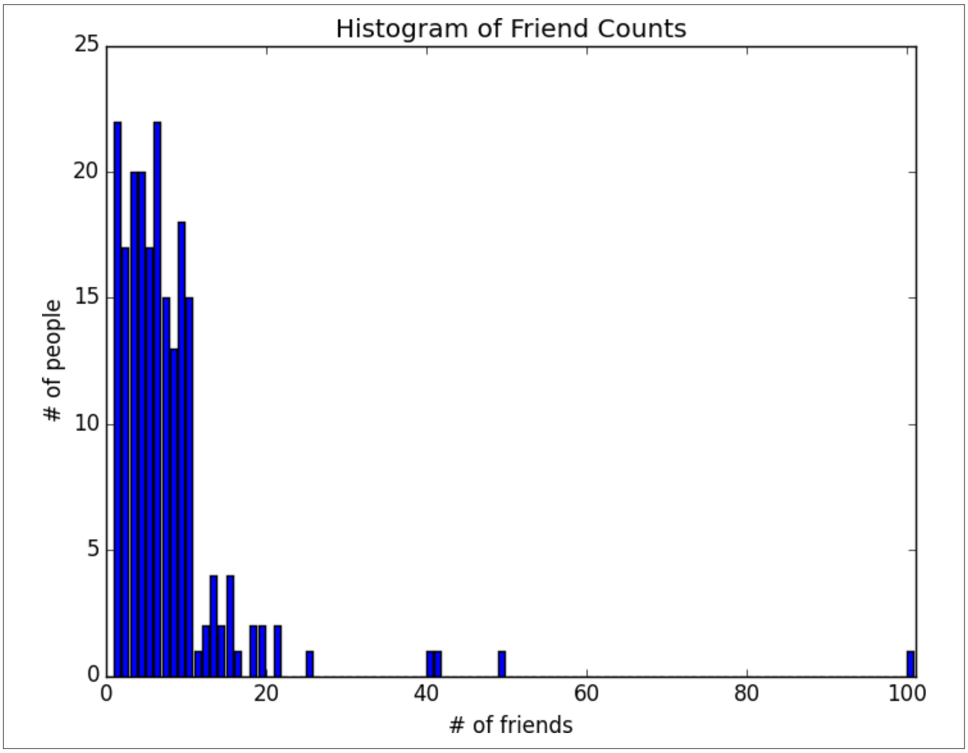

Figure 5-1. A histogram of friend counts

Unfortunately, this chart is still too difficult to slip into conversations. So you start generating some statistics. Probably the simplest statistic is simply the number of data points:

```
num_points = len(num_friends)               # 204
```

You're probably also interested in the largest and smallest values:

```
largest_value = max(num_friends)            # 100
smallest_value = min(num_friends)           # 1
```

which are just special cases of wanting to know the values in specific positions:

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]           # 1
second_smallest_value = sorted_values[1]    # 1
second_largest_value = sorted_values[-2]    # 49
```

But we're only getting started.

Central Tendencies

Usually, we'll want some notion of where our data is centered. Most commonly we'll use the *mean* (or average), which is just the sum of the data divided by its count:

```
# this isn't right if you don't from __future__ import division
def mean(x):
    return sum(x) / len(x)

mean(num_friends)   # 7.333333
```

If you have two data points, the mean is simply the point halfway between them. As you add more points, the mean shifts around, but it always depends on the value of every point.

We'll also sometimes be interested in the *median*, which is the middle-most value (if the number of data points is odd) or the average of the two middle-most values (if the number of data points is even).

For instance, if we have five data points in a sorted vector `x`, the median is `x[5 // 2]` or `x[2]`. If we have six data points, we want the average of `x[2]` (the third point) and `x[3]` (the fourth point).

Notice that—unlike the mean—the median doesn't depend on every value in your data. For example, if you make the largest point larger (or the smallest point smaller), the middle points remain unchanged, which means so does the median.

The `median` function is slightly more complicated than you might expect, mostly because of the "even" case:

```
def median(v):
    """finds the 'middle-most' value of v"""
    n = len(v)
    sorted_v = sorted(v)
    midpoint = n // 2

    if n % 2 == 1:
        # if odd, return the middle value
        return sorted_v[midpoint]
    else:
        # if even, return the average of the middle values
        lo = midpoint - 1
        hi = midpoint
        return (sorted_v[lo] + sorted_v[hi]) / 2
```

```
median(num_friends) # 6.0
```

Clearly, the mean is simpler to compute, and it varies smoothly as our data changes. If we have *n* data points and one of them increases by some small amount *e*, then necessarily the mean will increase by *e* / *n*. (This makes the mean amenable to all sorts of calculus tricks.) Whereas in order to find the median, we have to sort our data. And changing one of our data points by a small amount *e* might increase the median by *e*, by some number less than *e*, or not at all (depending on the rest of the data).

There are, in fact, nonobvious tricks to efficiently compute medians without sorting the data. However, they are beyond the scope of this book, so *we* have to sort the data.

At the same time, the mean is very sensitive to outliers in our data. If our friendliest user had 200 friends (instead of 100), then the mean would rise to 7.82, while the median would stay the same. If outliers are likely to be bad data (or otherwise unrepresentative of whatever phenomenon we're trying to understand), then the mean can sometimes give us a misleading picture. For example, the story is often told that in the mid-1980s, the major at the University of North Carolina with the highest average starting salary was geography, mostly on account of NBA star (and outlier) Michael Jordan.

A generalization of the median is the *quantile*, which represents the value less than which a certain percentile of the data lies. (The median represents the value less than which 50% of the data lies.)

```
def quantile(x, p):
    """returns the pth-percentile value in x"""
    p_index = int(p * len(x))
    return sorted(x)[p_index]

quantile(num_friends, 0.10) # 1
quantile(num_friends, 0.25) # 3
quantile(num_friends, 0.75) # 9
quantile(num_friends, 0.90) # 13
```

Less commonly you might want to look at the *mode*, or most-common value[s]:

```
def mode(x):
    """returns a list, might be more than one mode"""
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.iteritems()
            if count == max_count]

mode(num_friends)       # 1 and 6
```

But most frequently we'll just use the mean.

Dispersion

Dispersion refers to measures of how spread out our data is. Typically they're statistics for which values near zero signify *not spread out at all* and for which large values (whatever that means) signify *very spread out*. For instance, a very simple measure is the *range*, which is just the difference between the largest and smallest elements:

```
# "range" already means something in Python, so we'll use a different name
def data_range(x):
    return max(x) - min(x)

data_range(num_friends) # 99
```

The range is zero precisely when the `max` and `min` are equal, which can only happen if the elements of `x` are all the same, which means the data is as undispersed as possible. Conversely, if the range is large, then the `max` is much larger than the `min` and the data is more spread out.

Like the median, the range doesn't really depend on the whole data set. A data set whose points are all either 0 or 100 has the same range as a data set whose values are 0, 100, and lots of 50s. But it seems like the first data set "should" be more spread out.

A more complex measure of dispersion is the *variance*, which is computed as:

```
def de_mean(x):
    """translate x by subtracting its mean (so the result has mean 0)"""
    x_bar = mean(x)
    return [x_i - x_bar for x_i in x]

def variance(x):
    """assumes x has at least two elements"""
    n = len(x)
    deviations = de_mean(x)
    return sum_of_squares(deviations) / (n - 1)

variance(num_friends) # 81.54
```

This looks like it is almost the average squared deviation from the mean, except that we're dividing by `n-1` instead of `n`. In fact, when we're dealing with a sample from a larger population, `x_bar` is only an *estimate* of the actual mean, which means that on average `(x_i - x_bar) ** 2` is an underestimate of `x_i`'s squared deviation from the mean, which is why we divide by `n-1` instead of `n`. See Wikipedia.

Now, whatever units our data is in (e.g., “friends”), all of our measures of central tendency are in that same unit. The range will similarly be in that same unit. The variance, on the other hand, has units that are the *square* of the original units (e.g., “friends squared”). As it can be hard to make sense of these, we often look instead at the *standard deviation*:

```
def standard_deviation(x):
    return math.sqrt(variance(x))

standard_deviation(num_friends) # 9.03
```

Both the range and the standard deviation have the same outlier problem that we saw earlier for the mean. Using the same example, if our friendliest user had instead 200 friends, the standard deviation would be 14.89, more than 60% higher!

A more robust alternative computes the difference between the 75th percentile value and the 25th percentile value:

```
def interquartile_range(x):
    return quantile(x, 0.75) - quantile(x, 0.25)

interquartile_range(num_friends) # 6
```

which is quite plainly unaffected by a small number of outliers.

Correlation

DataSciencester’s VP of Growth has a theory that the amount of time people spend on the site is related to the number of friends they have on the site (she’s not a VP for nothing), and she’s asked you to verify this.

After digging through traffic logs, you’ve come up with a list `daily_minutes` that shows how many minutes per day each user spends on DataSciencester, and you’ve ordered it so that its elements correspond to the elements of our previous `num_friends` list. We’d like to investigate the relationship between these two metrics.

We’ll first look at *covariance*, the paired analogue of variance. Whereas variance measures how a single variable deviates from its mean, covariance measures how two variables vary in tandem from their means:

```
def covariance(x, y):
    n = len(x)
    return dot(de_mean(x), de_mean(y)) / (n - 1)

covariance(num_friends, daily_minutes) # 22.43
```

Recall that `dot` sums up the products of corresponding pairs of elements. When corresponding elements of `x` and `y` are either both above their means or both below their means, a positive number enters the sum. When one is above its mean and the other below, a negative number enters the sum. Accordingly, a "large" positive covariance means that x tends to be large when y is large and small when y is small. A "large" negative covariance means the opposite—that x tends to be small when y is large and vice versa. A covariance close to zero means that no such relationship exists.

Nonetheless, this number can be hard to interpret, for a couple of reasons:

- Its units are the product of the inputs' units (e.g., friend-minutes-per-day), which can be hard to make sense of. (What's a "friend-minute-per-day"?)
- If each user had twice as many friends (but the same number of minutes), the covariance would be twice as large. But in a sense the variables would be just as interrelated. Said differently, it's hard to say what counts as a "large" covariance.

For this reason, it's more common to look at the *correlation*, which divides out the standard deviations of both variables:

```
def correlation(x, y):
    stdev_x = standard_deviation(x)
    stdev_y = standard_deviation(y)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(x, y) / stdev_x / stdev_y
    else:
        return 0    # if no variation, correlation is zero

correlation(num_friends, daily_minutes) # 0.25
```

The `correlation` is unitless and always lies between -1 (perfect anti-correlation) and 1 (perfect correlation). A number like 0.25 represents a relatively weak positive correlation.

However, one thing we neglected to do was examine our data. Check out Figure 5-2.

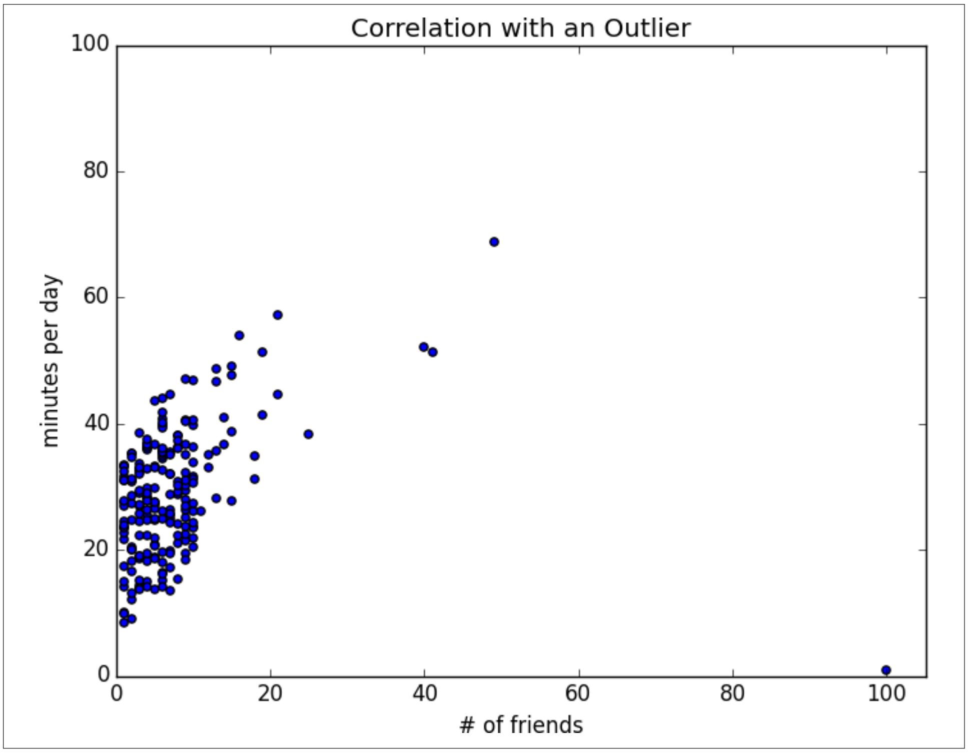

Figure 5-2. Correlation with an outlier

The person with 100 friends (who spends only one minute per day on the site) is a huge outlier, and correlation can be very sensitive to outliers. What happens if we ignore him?

```
outlier = num_friends.index(100)    # index of outlier

num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]

daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if i != outlier]

correlation(num_friends_good, daily_minutes_good) # 0.57
```

Without the outlier, there is a much stronger correlation (Figure 5-3).

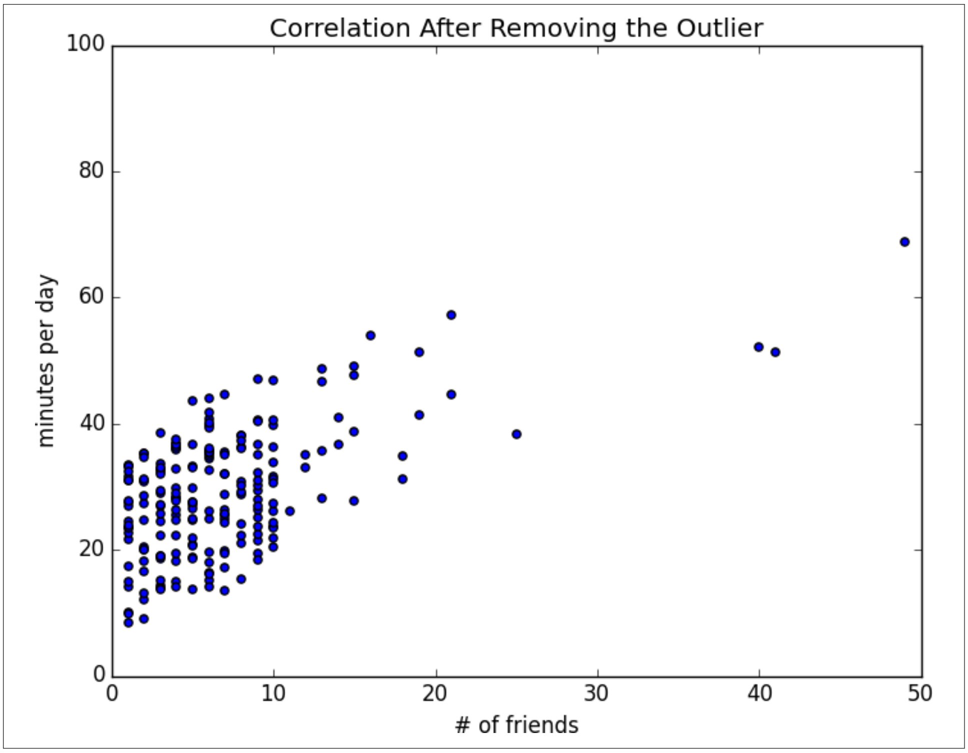

Figure 5-3. Correlation after removing the outlier

You investigate further and discover that the outlier was actually an internal *test* account that no one ever bothered to remove. So you feel pretty justified in excluding it.

Simpson's Paradox

One not uncommon surprise when analyzing data is Simpson's Paradox, in which correlations can be misleading when *confounding* variables are ignored.

For example, imagine that you can identify all of your members as either East Coast data scientists or West Coast data scientists. You decide to examine which coast's data scientists are friendlier:

coast	# of members	avg. # of friends
West Coast	101	8.2
East Coast	103	6.5

It certainly looks like the West Coast data scientists are friendlier than the East Coast data scientists. Your coworkers advance all sorts of theories as to why this might be: maybe it's the sun, or the coffee, or the organic produce, or the laid-back Pacific vibe?

When playing with the data you discover something very strange. If you only look at people with PhDs, the East Coast data scientists have more friends on average. And if you only look at people without PhDs, the East Coast data scientists also have more friends on average!

coast	degree	# of members	avg. # of friends
West Coast	PhD	35	3.1
East Coast	PhD	70	3.2
West Coast	no PhD	66	10.9
East Coast	no PhD	33	13.4

Once you account for the users' degrees, the correlation goes in the opposite direction! Bucketing the data as East Coast/West Coast disguised the fact that the East Coast data scientists skew much more heavily toward PhD types.

This phenomenon crops up in the real world with some regularity. The key issue is that correlation is measuring the relationship between your two variables *all else being equal*. If your data classes are assigned at random, as they might be in a well-designed experiment, "all else being equal" might not be a terrible assumption. But when there is a deeper pattern to class assignments, "all else being equal" can be an awful assumption.

The only real way to avoid this is by *knowing your data* and by doing what you can to make sure you've checked for possible confounding factors. Obviously, this is not always possible. If you didn't have the educational attainment of these 200 data scientists, you might simply conclude that there was something inherently more sociable about the West Coast.

Some Other Correlational Caveats

A correlation of zero indicates that there is no linear relationship between the two variables. However, there may be other sorts of relationships. For example, if:

```
x = [-2, -1, 0, 1, 2]
y = [ 2,  1, 0, 1, 2]
```

then x and y have zero correlation. But they certainly have a relationship—each element of y equals the absolute value of the corresponding element of x. What they don't have is a relationship in which knowing how `x_i` compares to `mean(x)` gives us information about how `y_i` compares to `mean(y)`. That is the sort of relationship that correlation looks for.

In addition, correlation tells you nothing about how large the relationship is. The variables:

```
x = [-2, 1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]
```

are perfectly correlated, but (depending on what you're measuring) it's quite possible that this relationship isn't all that interesting.

Correlation and Causation

You have probably heard at some point that "correlation is not causation," most likely by someone looking at data that posed a challenge to parts of his worldview that he was reluctant to question. Nonetheless, this is an important point—if `x` and `y` are strongly correlated, that might mean that `x` causes `y`, that `y` causes `x`, that each causes the other, that some third factor causes both, or it might mean nothing.

Consider the relationship between `num_friends` and `daily_minutes`. It's possible that having more friends on the site *causes* DataSciencester users to spend more time on the site. This might be the case if each friend posts a certain amount of content each day, which means that the more friends you have, the more time it takes to stay current with their updates.

However, it's also possible that the more time you spend arguing in the DataSciencester forums, the more you encounter and befriend like-minded people. That is, spending more time on the site *causes* users to have more friends.

A third possibility is that the users who are most passionate about data science spend more time on the site (because they find it more interesting) and more actively collect data science friends (because they don't want to associate with anyone else).

One way to feel more confident about causality is by conducting randomized trials. If you can randomly split your users into two groups with similar demographics and give one of the groups a slightly different experience, then you can often feel pretty good that the different experiences are causing the different outcomes.

For instance, if you don't mind being angrily accused of experimenting on your users, you could randomly choose a subset of your users and show them content from only a fraction of their friends. If this subset subsequently spent less time on the site, this would give you some confidence that having more friends *causes* more time on the site.

For Further Exploration

- SciPy, pandas, and StatsModels all come with a wide variety of statistical functions.
- Statistics is *important*. (Or maybe statistics *are* important?) If you want to be a good data scientist it would be a good idea to read a statistics textbook. Many are freely available online. A couple that I like are:
  - *OpenIntro Statistics*
  - *OpenStax Introductory Statistics*

CHAPTER 6

Probability

The laws of probability, so true in general, so fallacious in particular.
—Edward Gibbon

It is hard to do data science without some sort of understanding of *probability* and its mathematics. As with our treatment of statistics in Chapter 5, we'll wave our hands a lot and elide many of the technicalities.

For our purposes you should think of probability as a way of quantifying the uncertainty associated with *events* chosen from a some *universe* of events. Rather than getting technical about what these terms mean, think of rolling a die. The universe consists of all possible outcomes. And any subset of these outcomes is an event; for example, "the die rolls a one" or "the die rolls an even number."

Notationally, we write $P(E)$ to mean "the probability of the event E."

We'll use probability theory to build models. We'll use probability theory to evaluate models. We'll use probability theory all over the place.

One could, were one so inclined, get really deep into the philosophy of what probability theory *means*. (This is best done over beers.) We won't be doing that.

Dependence and Independence

Roughly speaking, we say that two events E and F are *dependent* if knowing something about whether E happens gives us information about whether F happens (and vice versa). Otherwise they are *independent*.

For instance, if we flip a fair coin twice, knowing whether the first flip is Heads gives us no information about whether the second flip is Heads. These events are independent. On the other hand, knowing whether the first flip is Heads certainly gives us information about whether both flips are Tails. (If the first flip is Heads, then definitely it's not the case that both flips are Tails.) These two events are dependent.

Mathematically, we say that two events E and F are independent if the probability that they both happen is the product of the probabilities that each one happens:

$$P(E, F) = P(E)P(F)$$

In the example above, the probability of "first flip Heads" is 1/2, and the probability of "both flips Tails" is 1/4, but the probability of "first flip Heads *and* both flips Tails" is 0.

Conditional Probability

When two events E and F are independent, then by definition we have:

$$P(E, F) = P(E)P(F)$$

If they are not necessarily independent (and if the probability of F is not zero), then we define the probability of E "conditional on F" as:

$$P(E|F) = P(E, F)/P(F)$$

You should think of this as the probability that E happens, given that we know that F happens.

We often rewrite this as:

$$P(E, F) = P(E|F)P(F)$$

When E and F are independent, you can check that this gives:

$$P(E|F) = P(E)$$

which is the mathematical way of expressing that knowing F occurred gives us no additional information about whether E occurred.

One common tricky example involves a family with two (unknown) children.

If we assume that:

1. Each child is equally likely to be a boy or a girl
2. The gender of the second child is independent of the gender of the first child

then the event “no girls” has probability 1/4, the event “one girl, one boy” has probability 1/2, and the event “two girls” has probability 1/4.

Now we can ask what is the probability of the event “both children are girls” (B) conditional on the event “the older child is a girl” (G)? Using the definition of conditional probability:

$$P(B|G) = P(B, G)/P(G) = P(B)/P(G) = 1/2$$

since the event B and G (“both children are girls *and* the older child is a girl”) is just the event B. (Once you know that both children are girls, it’s necessarily true that the older child is a girl.)

Most likely this result accords with your intuition.

We could also ask about the probability of the event “both children are girls” conditional on the event “at least one of the children is a girl” (L). Surprisingly, the answer is different from before!

As before, the event B and L (“both children are girls *and* at least one of the children is a girl”) is just the event B. This means we have:

$$P(B|L) = P(B, L)/P(L) = P(B)/P(L) = 1/3$$

How can this be the case? Well, if all you know is that at least one of the children is a girl, then it is twice as likely that the family has one boy and one girl than that it has both girls.

We can check this by “generating” a lot of families:

```
def random_kid():
    return random.choice(["boy", "girl"])

both_girls = 0
older_girl = 0
either_girl = 0

random.seed(0)
for _ in range(10000):
    younger = random_kid()
    older = random_kid()
    if older == "girl":
        older_girl += 1
    if older == "girl" and younger == "girl":
        both_girls += 1
    if older == "girl" or younger == "girl":
        either_girl += 1
```

```
print "P(both | older):", both_girls / older_girl      # 0.514 ~ 1/2
print "P(both | either): ", both_girls / either_girl   # 0.342 ~ 1/3
```

Bayes's Theorem

One of the data scientist's best friends is Bayes's Theorem, which is a way of "reversing" conditional probabilities. Let's say we need to know the probability of some event E conditional on some other event F occurring. But we only have information about the probability of F conditional on E occurring. Using the definition of conditional probability twice tells us that:

$$P(E|F) = P(E,F)/P(F) = P(F|E)P(E)/P(F)$$

The event F can be split into the two mutually exclusive events "F and E" and "F and not E." If we write $\neg E$ for "not E" (i.e., "E doesn't happen"), then:

$$P(F) = P(F,E) + P(F,\neg E)$$

so that:

$$P(E|F) = P(F|E)P(E)/[P(F|E)P(E) + P(F|\neg E)P(\neg E)]$$

which is how Bayes's Theorem is often stated.

This theorem often gets used to demonstrate why data scientists are smarter than doctors. Imagine a certain disease that affects 1 in every 10,000 people. And imagine that there is a test for this disease that gives the correct result ("diseased" if you have the disease, "nondiseased" if you don't) 99% of the time.

What does a positive test mean? Let's use T for the event "your test is positive" and D for the event "you have the disease." Then Bayes's Theorem says that the probability that you have the disease, conditional on testing positive, is:

$$P(D|T) = P(T|D)P(D)/[P(T|D)P(D) + P(T|\neg D)P(\neg D)]$$

Here we know that $P(T|D)$, the probability that someone with the disease tests positive, is 0.99. $P(D)$, the probability that any given person has the disease, is 1/10,000 = 0.0001. $P(T|\neg D)$, the probability that someone without the disease tests positive, is 0.01. And $P(\neg D)$, the probability that any given person doesn't have the disease, is 0.9999. If you substitute these numbers into Bayes's Theorem you find

$$P(D|T) = 0.98\%$$

That is, less than 1% of the people who test positive actually have the disease.

This assumes that people take the test more or less at random. If only people with certain symptoms take the test we would instead have to condition on the event "positive test *and* symptoms" and the number would likely be a lot higher.

While this is a simple calculation for a data scientist, most doctors will guess that $P(D|T)$ is approximately 2.

A more intuitive way to see this is to imagine a population of 1 million people. You'd expect 100 of them to have the disease, and 99 of those 100 to test positive. On the other hand, you'd expect 999,900 of them not to have the disease, and 9,999 of those to test positive. Which means that you'd expect only 99 out of (99 + 9999) positive testers to actually have the disease.

Random Variables

A *random variable* is a variable whose possible values have an associated probability distribution. A very simple random variable equals 1 if a coin flip turns up heads and 0 if the flip turns up tails. A more complicated one might measure the number of heads observed when flipping a coin 10 times or a value picked from `range(10)` where each number is equally likely.

The associated distribution gives the probabilities that the variable realizes each of its possible values. The coin flip variable equals 0 with probability 0.5 and 1 with probability 0.5. The `range(10)` variable has a distribution that assigns probability 0.1 to each of the numbers from 0 to 9.

We will sometimes talk about the *expected value* of a random variable, which is the average of its values weighted by their probabilities. The coin flip variable has an expected value of 1/2 (= 0 * 1/2 + 1 * 1/2), and the `range(10)` variable has an expected value of 4.5.

Random variables can be *conditioned* on events just as other events can. Going back to the two-child example from "Conditional Probability" on page 70, if X is the random variable representing the number of girls, X equals 0 with probability 1/4, 1 with probability 1/2, and 2 with probability 1/4.

We can define a new random variable Y that gives the number of girls conditional on at least one of the children being a girl. Then Y equals 1 with probability 2/3 and 2 with probability 1/3. And a variable Z that's the number of girls conditional on the older child being a girl equals 1 with probability 1/2 and 2 with probability 1/2.

For the most part, we will be using random variables *implicitly* in what we do without calling special attention to them. But if you look deeply you'll see them.

Continuous Distributions

A coin flip corresponds to a *discrete distribution*—one that associates positive probability with discrete outcomes. Often we'll want to model distributions across a continuum of outcomes. (For our purposes, these outcomes will always be real numbers, although that's not always the case in real life.) For example, the *uniform distribution* puts *equal weight* on all the numbers between 0 and 1.

Because there are infinitely many numbers between 0 and 1, this means that the weight it assigns to individual points must necessarily be zero. For this reason, we represent a continuous distribution with a *probability density function* (pdf) such that the probability of seeing a value in a certain interval equals the integral of the density function over the interval.

If your integral calculus is rusty, a simpler way of understanding this is that if a distribution has density function f, then the probability of seeing a value between x and $x + h$ is approximately $h * f(x)$ if h is small.

The density function for the uniform distribution is just:

```
def uniform_pdf(x):
    return 1 if x >= 0 and x < 1 else 0
```

The probability that a random variable following that distribution is between 0.2 and 0.3 is 1/10, as you'd expect. Python's `random.random()` is a [pseudo]random variable with a uniform density.

We will often be more interested in the *cumulative distribution function* (cdf), which gives the probability that a random variable is less than or equal to a certain value. It's not hard to create the cumulative distribution function for the uniform distribution (Figure 6-1):

```
def uniform_cdf(x):
    "returns the probability that a uniform random variable is <= x"
    if x < 0:   return 0    # uniform random is never less than 0
    elif x < 1: return x    # e.g. P(X <= 0.4) = 0.4
    else:       return 1    # uniform random is always less than 1
```

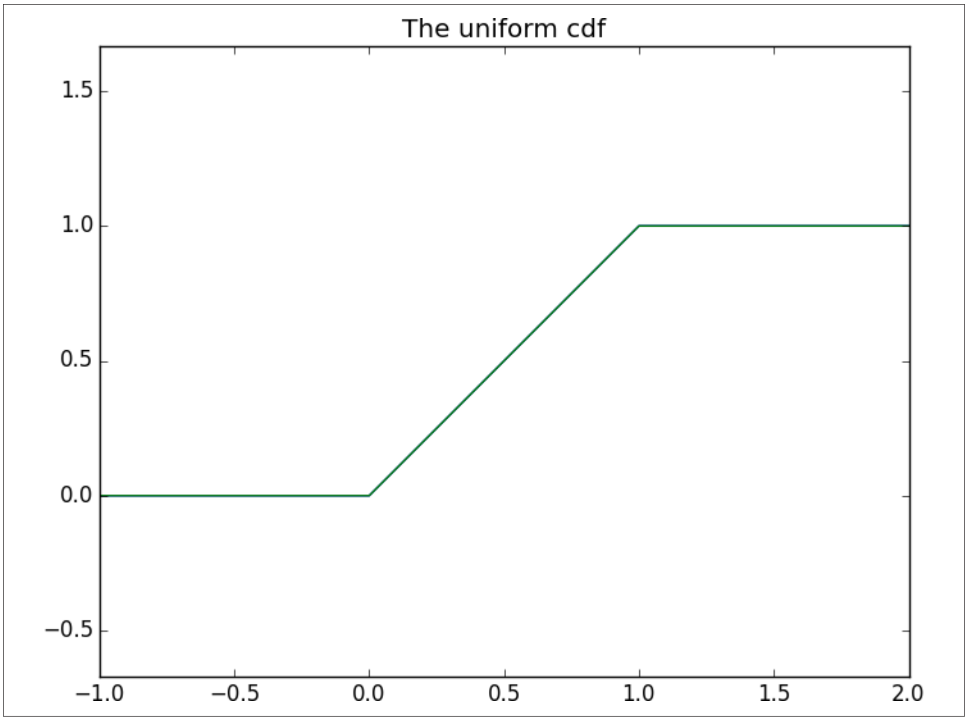

Figure 6-1. The uniform cdf

The Normal Distribution

The normal distribution is the king of distributions. It is the classic bell curve–shaped distribution and is completely determined by two parameters: its mean μ (mu) and its standard deviation σ (sigma). The mean indicates where the bell is centered, and the standard deviation how "wide" it is.

It has the distribution function:

$$f(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

which we can implement as:

```
def normal_pdf(x, mu=0, sigma=1):
    sqrt_two_pi = math.sqrt(2 * math.pi)
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (sqrt_two_pi * sigma))
```

In Figure 6-2, we plot some of these pdfs to see what they look like:

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_pdf(x,mu=-1)   for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend()
plt.title("Various Normal pdfs")
plt.show()
```

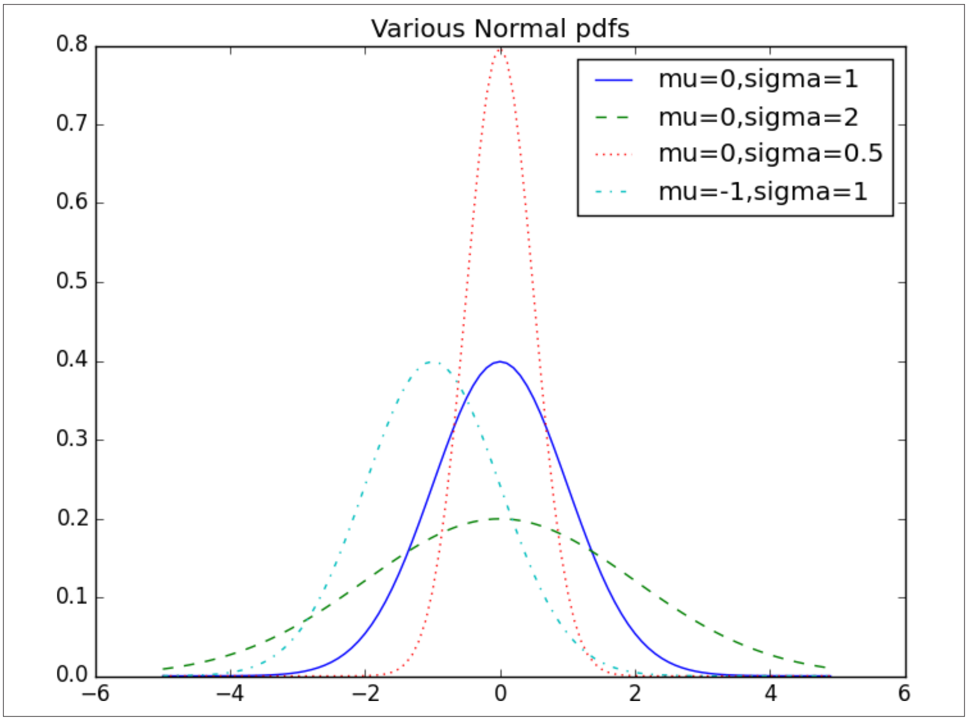

Figure 6-2. Various normal pdfs

When $\mu = 0$ and $\sigma = 1$, it's called the *standard normal distribution*. If Z is a standard normal random variable, then it turns out that:

$$X = \sigma Z + \mu$$

is also normal but with mean μ and standard deviation σ. Conversely, if X is a normal random variable with mean μ and standard deviation σ,

$$Z = (X - \mu)/\sigma$$

is a standard normal variable.

The cumulative distribution function for the normal distribution cannot be written in an "elementary" manner, but we can write it using Python's `math.erf`:

```
def normal_cdf(x, mu=0,sigma=1):
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

Again, in Figure 6-3, we plot a few:

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # bottom right
plt.title("Various Normal cdfs")
plt.show()
```

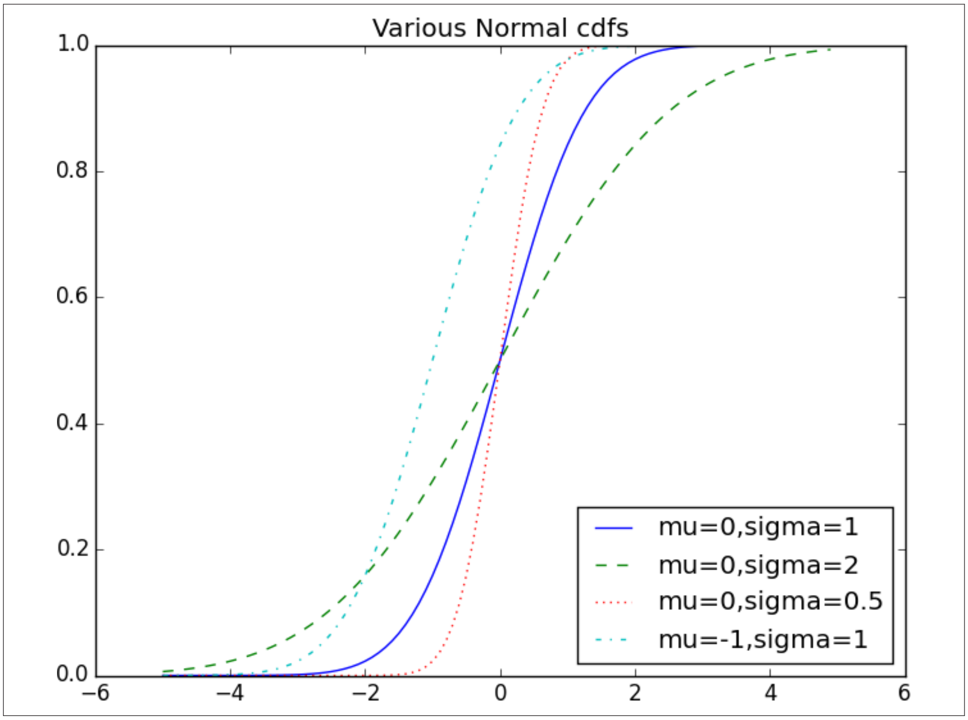

Figure 6-3. Various normal cdfs

Sometimes we'll need to invert `normal_cdf` to find the value corresponding to a specified probability. There's no simple way to compute its inverse, but `normal_cdf` is continuous and strictly increasing, so we can use a binary search:

```
def inverse_normal_cdf(p, mu=0, sigma=1, tolerance=0.00001):
    """find approximate inverse using binary search"""
    # if not standard, compute standard and rescale
    if mu != 0 or sigma != 1:
        return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)

    low_z, low_p = -10.0, 0            # normal_cdf(-10) is (very close to) 0
    hi_z,  hi_p  =  10.0, 1            # normal_cdf(10)  is (very close to) 1
    while hi_z - low_z > tolerance:
        mid_z = (low_z + hi_z) / 2     # consider the midpoint
        mid_p = normal_cdf(mid_z)      # and the cdf's value there
        if mid_p < p:
            # midpoint is still too low, search above it
            low_z, low_p = mid_z, mid_p
        elif mid_p > p:
            # midpoint is still too high, search below it
            hi_z, hi_p = mid_z, mid_p
        else:
            break

    return mid_z
```

The function repeatedly bisects intervals until it narrows in on a Z that's close enough to the desired probability.

The Central Limit Theorem

One reason the normal distribution is so useful is the *central limit theorem*, which says (in essence) that a random variable defined as the average of a large number of independent and identically distributed random variables is itself approximately normally distributed.

In particular, if $x_1, ..., x_n$ are random variables with mean μ and standard deviation σ, and if n is large, then:

$$\frac{1}{n}\left(x_1 + ... + x_n\right)$$

is approximately normally distributed with mean μ and standard deviation $\sigma / \sqrt{n}$. Equivalently (but often more usefully),

$$\frac{\left(x_1 + ... + x_n\right) - \mu n}{\sigma \sqrt{n}}$$

is approximately normally distributed with mean 0 and standard deviation 1.

An easy way to illustrate this is by looking at *binomial* random variables, which have two parameters n and p. A Binomial(n,p) random variable is simply the sum of n

independent Bernoulli(p) random variables, each of which equals 1 with probability p and 0 with probability $1 - p$:

```
def bernoulli_trial(p):
    return 1 if random.random() < p else 0

def binomial(n, p):
    return sum(bernoulli_trial(p) for _ in range(n))
```

The mean of a Bernoulli(p) variable is p, and its standard deviation is $\sqrt{p(1-p)}$. The central limit theorem says that as n gets large, a Binomial(n,p) variable is approximately a normal random variable with mean $\mu = np$ and standard deviation $\sigma = \sqrt{np(1-p)}$. If we plot both, you can easily see the resemblance:

```
def make_hist(p, n, num_points):

    data = [binomial(n, p) for _ in range(num_points)]

    # use a bar chart to show the actual binomial samples
    histogram = Counter(data)
    plt.bar([x - 0.4 for x in histogram.keys()],
            [v / num_points for v in histogram.values()],
            0.8,
            color='0.75')

    mu = p * n
    sigma = math.sqrt(n * p * (1 - p))

    # use a line chart to show the normal approximation
    xs = range(min(data), max(data) + 1)
    ys = [normal_cdf(i + 0.5, mu, sigma) - normal_cdf(i - 0.5, mu, sigma)
          for i in xs]
    plt.plot(xs,ys)
    plt.title("Binomial Distribution vs. Normal Approximation")
    plt.show()
```

For example, when you call `make_hist(0.75, 100, 10000)`, you get the graph in Figure 6-4.

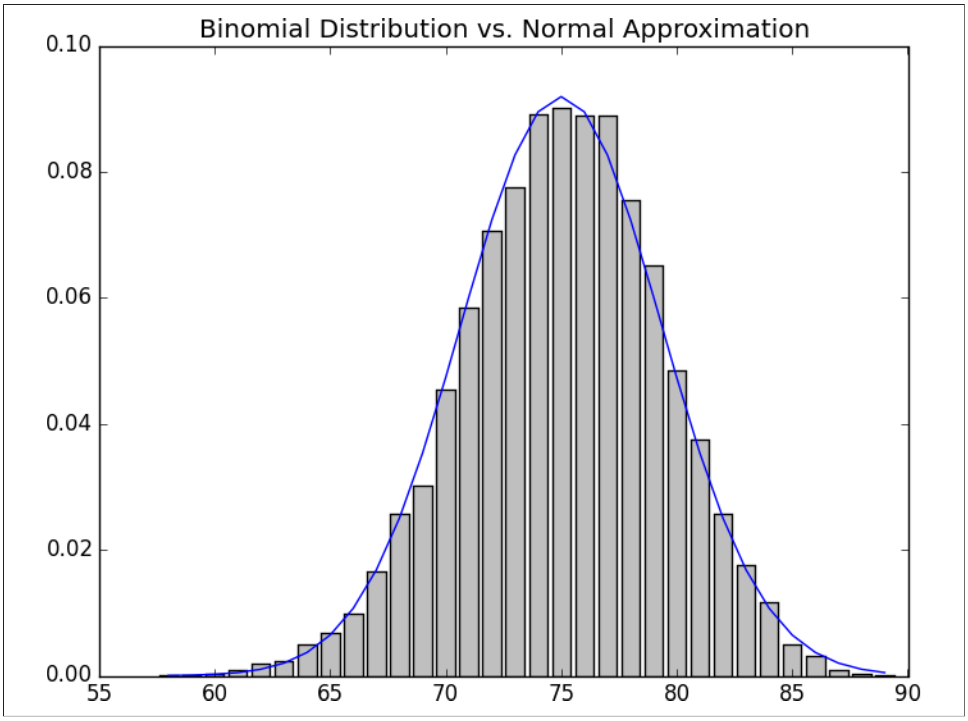

Figure 6-4. The output from `make_hist`

The moral of this approximation is that if you want to know the probability that (say) a fair coin turns up more than 60 heads in 100 flips, you can estimate it as the probability that a Normal(50,5) is greater than 60, which is easier than computing the Binomial(100,0.5) cdf. (Although in most applications you'd probably be using statistical software that would gladly compute whatever probabilities you want.)

For Further Exploration

- scipy.stats contains pdf and cdf functions for most of the popular probability distributions.
- Remember how, at the end of Chapter 5, I said that it would be a good idea to study a statistics textbook? It would also be a good idea to study a probability textbook. The best one I know that's available online is *Introduction to Probability*.

CHAPTER 9

Getting Data

To write it, it took three months; to conceive it, three minutes; to collect the data in it, all my life.

—F. Scott Fitzgerald

In order to be a data scientist you need data. In fact, as a data scientist you will spend an embarrassingly large fraction of your time acquiring, cleaning, and transforming data. In a pinch, you can always type the data in yourself (or if you have minions, make them do it), but usually this is not a good use of your time. In this chapter, we'll look at different ways of getting data into Python and into the right formats.

stdin and stdout

If you run your Python scripts at the command line, you can *pipe* data through them using `sys.stdin` and `sys.stdout`. For example, here is a script that reads in lines of text and spits back out the ones that match a regular expression:

```
# egrep.py
import sys, re

# sys.argv is the list of command-line arguments
# sys.argv[0] is the name of the program itself
# sys.argv[1] will be the regex specified at the command line
regex = sys.argv[1]

# for every line passed into the script
for line in sys.stdin:
    # if it matches the regex, write it to stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

And here's one that counts the lines it receives and then writes out the count:

```
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1

# print goes to sys.stdout
print count
```

You could then use these to count how many lines of a file contain numbers. In Windows, you'd use:

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

whereas in a Unix system you'd use:

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

The | is the pipe character, which means "use the output of the left command as the input of the right command." You can build pretty elaborate data-processing pipelines this way.

If you are using Windows, you can probably leave out the `python` part of this command:

```
type SomeFile.txt | egrep.py "[0-9]" | line_count.py
```

If you are on a Unix system, doing so might require a little more work.

Similarly, here's a script that counts the words in its input and writes out the most common ones:

```
# most_common_words.py
import sys
from collections import Counter

# pass in number of words as first argument
try:
    num_words = int(sys.argv[1])
except:
    print "usage: most_common_words.py num_words"
    sys.exit(1)   # non-zero exit code indicates error

counter = Counter(word.lower()                      # lowercase words
                  for line in sys.stdin             #
                  for word in line.strip().split()  # split on spaces
                  if word)                          # skip empty 'words'

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

after which you could do something like:

```
C:\DataScience>type the_bible.txt | python most_common_words.py 10
64193   the
51380   and
34753   of
13643   to
12799   that
12560   in
10263   he
9840    shall
8987    unto
8836    for
```

If you are a seasoned Unix programmer, you are probably familiar with a wide variety of command-line tools (for example, `egrep`) that are built into your operating system and that are probably preferable to building your own from scratch. Still, it's good to know you can if you need to.

Reading Files

You can also explicitly read from and write to files directly in your code. Python makes working with files pretty simple.

The Basics of Text Files

The first step to working with a text file is to obtain a *file object* using `open`:

```
# 'r' means read-only
file_for_reading = open('reading_file.txt', 'r')

# 'w' is write -- will destroy the file if it already exists!
file_for_writing = open('writing_file.txt', 'w')

# 'a' is append -- for adding to the end of the file
file_for_appending = open('appending_file.txt', 'a')

# don't forget to close your files when you're done
file_for_writing.close()
```

Because it is easy to forget to close your files, you should always use them in a `with` block, at the end of which they will be closed automatically:

```
with open(filename,'r') as f:
    data = function_that_gets_data_from(f)
```

```
# at this point f has already been closed, so don't try to use it
process(data)
```

If you need to read a whole text file, you can just iterate over the lines of the file using `for`:

```
starts_with_hash = 0

with open('input.txt','r') as f:
    for line in file:                 # look at each line in the file
        if re.match("^#",line):       # use a regex to see if it starts with '#'
            starts_with_hash += 1     # if it does, add 1 to the count
```

Every line you get this way ends in a newline character, so you'll often want to `strip()` it before doing anything with it.

For example, imagine you have a file full of email addresses, one per line, and that you need to generate a histogram of the domains. The rules for correctly extracting domains are somewhat subtle (e.g., the Public Suffix List), but a good first approximation is to just take the parts of the email addresses that come after the `@`. (Which gives the wrong answer for email addresses like `joel@mail.datasciencester.com`.)

```
def get_domain(email_address):
    """split on '@' and return the last piece"""
    return email_address.lower().split("@")[-1]

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

Delimited Files

The hypothetical email addresses file we just processed had one address per line. More frequently you'll work with files with lots of data on each line. These files are very often either *comma-separated* or *tab-separated*. Each line has several fields, with a comma (or a tab) indicating where one field ends and the next field starts.

This starts to get complicated when you have fields with commas and tabs and newlines in them (which you inevitably do). For this reason, it's pretty much always a mistake to try to parse them yourself. Instead, you should use Python's `csv` module (or the `pandas` library). For technical reasons that you should feel free to blame on Microsoft, you should always work with `csv` files in *binary* mode by including a *b* after the *r* or *w* (see Stack Overflow).

If your file has no headers (which means you probably want each row as a `list`, and which places the burden on you to know what's in each column), you can use `csv.reader` to iterate over the rows, each of which will be an appropriately split list.

For example, if we had a tab-delimited file of stock prices:

```
6/20/2014	AAPL	90.91
6/20/2014	MSFT	41.68
6/20/2014	FB	64.5
6/19/2014	AAPL	91.86
6/19/2014	MSFT	41.51
6/19/2014	FB	64.34
```

we could process them with:

```
import csv

with open('tab_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

If your file has headers:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

you can either skip the header row (with an initial call to `reader.next()`) or get each row as a `dict` (with the headers as keys) by using `csv.DictReader`:

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.DictReader(f, delimiter=':')
    for row in reader:
        date = row["date"]
        symbol = row["symbol"]
        closing_price = float(row["closing_price"])
        process(date, symbol, closing_price)
```

Even if your file doesn't have headers you can still use `DictReader` by passing it the keys as a `fieldnames` parameter.

You can similarly write out delimited data using `csv.writer`:

```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }

with open('comma_delimited_stock_prices.txt','wb') as f:
    writer = csv.writer(f, delimiter=',')
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

`csv.writer` will do the right thing if your fields themselves have commas in them. Your own hand-rolled writer probably won't. For example, if you attempt:

```
results = [["test1", "success", "Monday"],
           ["test2", "success, kind of", "Tuesday"],
           ["test3", "failure, kind of", "Wednesday"],
           ["test4", "failure, utter", "Thursday"]]

# don't do this!
with open('bad_csv.txt', 'wb') as f:
    for row in results:
        f.write(",".join(map(str, row))) # might have too many commas in it!
        f.write("\n")                    # row might have newlines as well!
```

You will end up with a csv file that looks like:

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

and that no one will ever be able to make sense of.

Scraping the Web

Another way to get data is by scraping it from web pages. Fetching web pages, it turns out, is pretty easy; getting meaningful structured information out of them less so.

HTML and the Parsing Thereof

Pages on the Web are written in HTML, in which text is (ideally) marked up into elements and their attributes:

```
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

In a perfect world, where all web pages are marked up semantically for our benefit, we would be able to extract data using rules like "find the `<p>` element whose `id` is `subject` and return the text it contains." In the actual world, HTML is not generally well-formed, let alone annotated. This means we'll need help making sense of it.

To get data out of HTML, we will use the BeautifulSoup library, which builds a tree out of the various elements on a web page and provides a simple interface for accessing them. As I write this, the latest version is Beautiful Soup 4.3.2 (`pip install beautifulsoup4`), which is what we'll be using. We'll also be using the requests library (pip install requests), which is a much nicer way of making HTTP requests than anything that's built into Python.

Python's built-in HTML parser is not that lenient, which means that it doesn't always cope well with HTML that's not perfectly formed. For that reason, we'll use a different parser, which we need to install:

```
pip install html5lib
```

To use Beautiful Soup, we'll need to pass some HTML into the BeautifulSoup() function. In our examples, this will be the result of a call to requests.get:

```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://www.example.com").text
soup = BeautifulSoup(html, 'html5lib')
```

after which we can get pretty far using a few simple methods.

We'll typically work with Tag objects, which correspond to the tags representing the structure of an HTML page.

For example, to find the first <p> tag (and its contents) you can use:

```
first_paragraph = soup.find('p')        # or just soup.p
```

You can get the text contents of a Tag using its text property:

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

And you can extract a tag's attributes by treating it like a dict:

```
first_paragraph_id = soup.p['id']       # raises KeyError if no 'id'
first_paragraph_id2 = soup.p.get('id')  # returns None if no 'id'
```

You can get multiple tags at once:

```
all_paragraphs = soup.find_all('p')  # or just soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

Frequently you'll want to find tags with a specific class:

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
```

And you can combine these to implement more elaborate logic. For example, if you want to find every <span> element that is contained inside a <div> element, you could do this:

```
# warning, will return the same span multiple times
# if it sits inside multiple divs
# be more clever if that's the case
```

```
spans_inside_divs = [span
                     for div in soup('div')     # for each <div> on the page
                     for span in div('span')]   # find each <span> inside it
```

Just this handful of features will allow us to do quite a lot. If you end up needing to do more-complicated things (or if you're just curious), check the documentation.

Of course, whatever data is important won't typically be labeled as `class="important"`. You'll need to carefully inspect the source HTML, reason through your selection logic, and worry about edge cases to make sure your data is correct. Let's look at an example.

Example: O'Reilly Books About Data

A potential investor in DataSciencester thinks data is just a fad. To prove him wrong, you decide to examine how many data books O'Reilly has published over time. After digging through its website, you find that it has many pages of data books (and videos), reachable through 30-items-at-a-time directory pages with URLs like:

```
http://shop.oreilly.com/category/browse-subjects/data.do?
sortby=publicationDate&page=1
```

Unless you want to be a jerk (and unless you want your scraper to get banned), whenever you want to scrape data from a website you should first check to see if it has some sort of access policy. Looking at:

```
http://oreilly.com/terms/
```

there seems to be nothing prohibiting this project. In order to be good citizens, we should also check for a *robots.txt* file that tells webcrawlers how to behave. The important lines in *http://shop.oreilly.com/robots.txt* are:

```
Crawl-delay: 30
Request-rate: 1/30
```

The first tells us that we should wait 30 seconds between requests, the second that we should request only one page every 30 seconds. So basically they're two different ways of saying the same thing. (There are other lines that indicate directories not to scrape, but they don't include our URL, so we're OK there.)

> There's always the possibility that O'Reilly will at some point revamp its website and break all the logic in this section. I will do what I can to prevent that, of course, but I don't have a ton of influence over there. Although, if every one of you were to convince everyone you know to buy a copy of this book…

To figure out how to extract the data, let's download one of those pages and feed it to Beautiful Soup:

```
# you don't have to split the url like this unless it needs to fit in a book
url = "http://shop.oreilly.com/category/browse-subjects/" + \
      "data.do?sortby=publicationDate&page=1"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
```

If you view the source of the page (in your browser, right-click and select "View source" or "View page source" or whatever option looks the most like that), you'll see that each book (or video) seems to be uniquely contained in a `<td>` table cell element whose `class` is `thumbtext`. Here is (an abridged version of) the relevant HTML for one book:

```
<td class="thumbtext">
  <div class="thumbcontainer">
    <div class="thumbdiv">
      <a href="/product/9781118903407.do">
        <img src="..."/>
      </a>
    </div>
  </div>
  <div class="widthchange">
    <div class="thumbheader">
      <a href="/product/9781118903407.do">Getting a Big Data Job For Dummies</a>
    </div>
    <div class="AuthorName">By Jason Williamson</div>
    <span class="directorydate">        December 2014    </span>
    <div style="clear:both;">
      <div id="146350">
        <span class="pricelabel">
                            Ebook:

                            <span class="price">&nbsp;$29.99</span>
        </span>
      </div>
    </div>
  </div>
</td>
```

A good first step is to find all of the `td thumbtext` tag elements:

```
tds = soup('td', 'thumbtext')
print len(tds)
# 30
```

Next we'd like to filter out the videos. (The would-be investor is only impressed by books.) If we inspect the HTML further, we see that each `td` contains one or more `span` elements whose `class` is `pricelabel`, and whose text looks like `Ebook:` or `Video:` or `Print:`. It appears that the videos contain only one `pricelabel`, whose `text` starts with `Video` (after removing leading spaces). This means we can test for videos with:

```
def is_video(td):
    """it's a video if it has exactly one pricelabel, and if
```

```
        the stripped text inside that pricelabel starts with 'Video'"""
        pricelabels = td('span', 'pricelabel')
        return (len(pricelabels) == 1 and
                pricelabels[0].text.strip().startswith("Video"))

    print len([td for td in tds if not is_video(td)])
    # 21 for me, might be different for you
```

Now we're ready to start pulling data out of the `td` elements. It looks like the book title is the text inside the `<a>` tag inside the `<div class="thumbheader">`:

```
    title = td.find("div", "thumbheader").a.text
```

The author(s) are in the text of the `AuthorName <div>`. They are prefaced by a `By` (which we want to get rid of) and separated by commas (which we want to split out, after which we'll need to get rid of spaces):

```
    author_name = td.find('div', 'AuthorName').text
    authors = [x.strip() for x in re.sub("^By ", "", author_name).split(",")]
```

The ISBN seems to be contained in the link that's in the `thumbheader <div>`:

```
    isbn_link = td.find("div", "thumbheader").a.get("href")

    # re.match captures the part of the regex in parentheses
    isbn = re.match("/product/(.*)\.do", isbn_link).group(1)
```

And the date is just the contents of the `<span class="directorydate">`:

```
    date = td.find("span", "directorydate").text.strip()
```

Let's put this all together into a function:

```
    def book_info(td):
        """given a BeautifulSoup <td> Tag representing a book,
        extract the book's details and return a dict"""

        title = td.find("div", "thumbheader").a.text
        by_author = td.find('div', 'AuthorName').text
        authors = [x.strip() for x in re.sub("^By ", "", by_author).split(",")]
        isbn_link = td.find("div", "thumbheader").a.get("href")
        isbn = re.match("/product/(.*)\.do", isbn_link).groups()[0]
        date = td.find("span", "directorydate").text.strip()

        return {
            "title" : title,
            "authors" : authors,
            "isbn" : isbn,
            "date" : date
        }
```

And now we're ready to scrape:

```
    from bs4 import BeautifulSoup
    import requests
```

```
from time import sleep
base_url = "http://shop.oreilly.com/category/browse-subjects/" + \
           "data.do?sortby=publicationDate&page="

books = []

NUM_PAGES = 31     # at the time of writing, probably more by now

for page_num in range(1, NUM_PAGES + 1):
    print "souping page", page_num, ",", len(books), " found so far"
    url = base_url + str(page_num)
    soup = BeautifulSoup(requests.get(url).text, 'html5lib')

    for td in soup('td', 'thumbtext'):
        if not is_video(td):
            books.append(book_info(td))

    # now be a good citizen and respect the robots.txt!
    sleep(30)
```

Extracting data from HTML like this is more data art than data science. There are countless other find-the-books and find-the-title logics that would have worked just as well.

Now that we've collected the data, we can plot the number of books published each year (Figure 9-1):

```
def get_year(book):
    """book["date"] looks like 'November 2014' so we need to
    split on the space and then take the second piece"""
    return int(book["date"].split()[1])

# 2014 is the last complete year of data (when I ran this)
year_counts = Counter(get_year(book) for book in books
                      if get_year(book) <= 2014)

import matplotlib.pyplot as plt
years = sorted(year_counts)
book_counts = [year_counts[year] for year in years]
plt.plot(years, book_counts)
plt.ylabel("# of data books")
plt.title("Data is Big!")
plt.show()
```

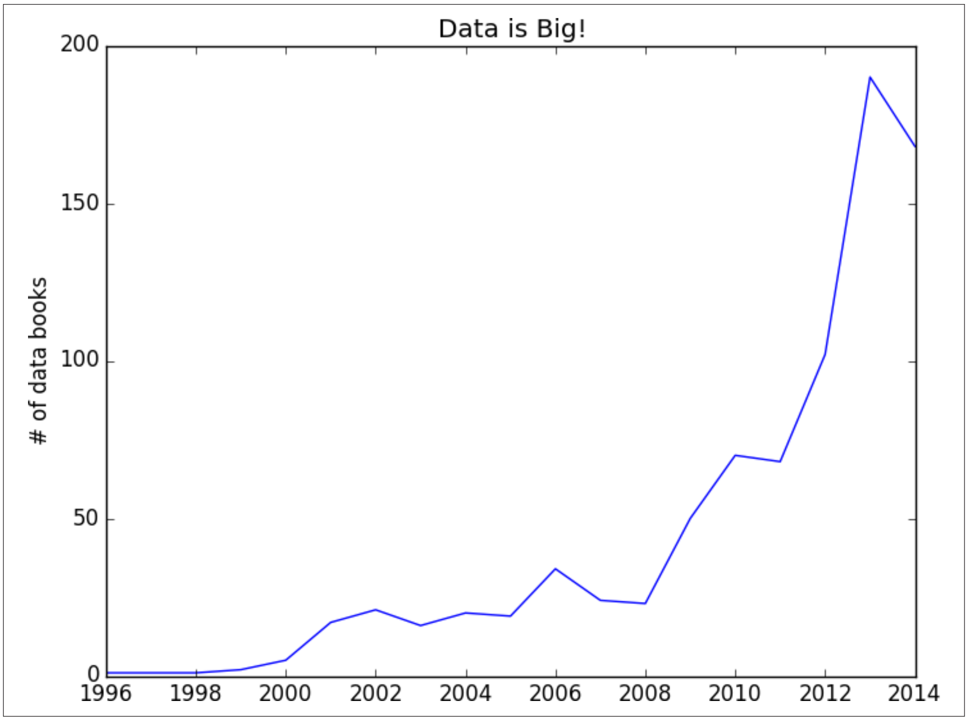

Figure 9-1. Number of data books per year

Unfortunately, the would-be investor looks at the graph and decides that 2013 was "peak data."

Using APIs

Many websites and web services provide application programming interfaces (APIs), which allow you to explicitly request data in a structured format. This saves you the trouble of having to scrape them!

JSON (and XML)

Because HTTP is a protocol for transferring *text*, the data you request through a web API needs to be *serialized* into a string format. Often this serialization uses JavaScript Object Notation (JSON). JavaScript objects look quite similar to Python `dicts`, which makes their string representations easy to interpret:

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2014,
  "topics" : [ "data", "science", "data science"] }
```

We can parse JSON using Python's `json` module. In particular, we will use its `loads` function, which deserializes a string representing a JSON object into a Python object:

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : "Joel Grus",
                  "publicationYear" : 2014,
                  "topics" : [ "data", "science", "data science"] }"""

# parse the JSON to create a Python dict
deserialized = json.loads(serialized)
if "data science" in deserialized["topics"]:
    print deserialized
```

Sometimes an API provider hates you and only provides responses in XML:

```
<Book>
  <Title>Data Science Book</Title>
  <Author>Joel Grus</Author>
  <PublicationYear>2014</PublicationYear>
  <Topics>
    <Topic>data</Topic>
    <Topic>science</Topic>
    <Topic>data science</Topic>
  </Topics>
</Book>
```

You can use `BeautifulSoup` to get data from XML similarly to how we used it to get data from HTML; check its documentation for details.

Using an Unauthenticated API

Most APIs these days require you to first authenticate yourself in order to use them. While we don't begrudge them this policy, it creates a lot of extra boilerplate that muddies up our exposition. Accordingly, we'll first take a look at GitHub's API, with which you can do some simple things unauthenticated:

```
import requests, json
endpoint = "https://api.github.com/users/joelgrus/repos"

repos = json.loads(requests.get(endpoint).text)
```

At this point `repos` is a `list` of Python `dict`s, each representing a public repository in my GitHub account. (Feel free to substitute your username and get your GitHub repository data instead. You do have a GitHub account, right?)

We can use this to figure out which months and days of the week I'm most likely to create a repository. The only issue is that the dates in the response are (Unicode) strings:

```
u'created_at': u'2013-07-05T02:02:28Z'
```

Python doesn't come with a great date parser, so we'll need to install one:

```
pip install python-dateutil
```

from which you'll probably only ever need the `dateutil.parser.parse` function:

```
from dateutil.parser import parse

dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

Similarly, you can get the languages of my last five repositories:

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["created_at"],
                             reverse=True)[:5]

last_5_languages = [repo["language"]
                    for repo in last_5_repositories]
```

Typically we won't be working with APIs at this low "make the requests and parse the responses ourselves" level. One of the benefits of using Python is that someone has already built a library for pretty much any API you're interested in accessing. When they're done well, these libraries can save you a lot of the trouble of figuring out the hairier details of API access. (When they're not done well, or when it turns out they're based on defunct versions of the corresponding APIs, they can cause you enormous headaches.)

Nonetheless, you'll occasionally have to roll your own API-access library (or, more likely, debug why someone else's isn't working), so it's good to know some of the details.

Finding APIs

If you need data from a specific site, look for a developers or API section of the site for details, and try searching the Web for "python __ api" to find a library. There is a Rotten Tomatoes API for Python. There are multiple Python wrappers for the Klout API, for the Yelp API, for the IMDB API, and so on.

If you're looking for lists of APIs that have Python wrappers, two directories are at Python API and Python for Beginners.

If you want a directory of web APIs more broadly (without Python wrappers necessarily), a good resource is Programmable Web, which has a huge directory of categorized APIs.

And if after all that you can't find what you need, there's always scraping, the last refuge of the data scientist.

Example: Using the Twitter APIs

Twitter is a fantastic source of data to work with. You can use it to get real-time news. You can use it to measure reactions to current events. You can use it to find links related to specific topics. You can use it for pretty much anything you can imagine, just as long as you can get access to its data. And you can get access to its data through its API.

To interact with the Twitter APIs we'll be using the Twython library (`pip install twython`). There are quite a few Python Twitter libraries out there, but this is the one that I've had the most success working with. You are encouraged to explore the others as well!

Getting Credentials

In order to use Twitter's APIs, you need to get some credentials (for which you need a Twitter account, which you should have anyway so that you can be part of the lively and friendly Twitter `#datascience` community). Like all instructions that relate to websites that I don't control, these may go obsolete at some point but will hopefully work for a while. (Although they have already changed at least once while I was writing this book, so good luck!)

1. Go to *https://apps.twitter.com/*.
2. If you are not signed in, click Sign in and enter your Twitter username and password.
3. Click Create New App.
4. Give it a name (such as "Data Science") and a description, and put any URL as the website (it doesn't matter which one).
5. Agree to the Terms of Service and click Create.
6. Take note of the consumer key and consumer secret.
7. Click "Create my access token."
8. Take note of the access token and access token secret (you may have to refresh the page).

The consumer key and consumer secret tell Twitter what application is accessing its APIs, while the access token and access token secret tell Twitter *who* is accessing its APIs. If you have ever used your Twitter account to log in to some other site, the "click to authorize" page was generating an access token for that site to use to convince Twitter that it was you (or, at least, acting on your behalf). As we don't need this "let anyone log in" functionality, we can get by with the statically generated access token and access token secret.

The consumer key/secret and access token key/secret should be treated like *passwords*. You shouldn't share them, you shouldn't publish them in your book, and you shouldn't check them into your public GitHub repository. One simple solution is to store them in a *credentials.json* file that doesn't get checked in, and to have your code use `json.loads` to retrieve them.

Using Twython

First we'll look at the Search API, which requires only the consumer key and secret, not the access token or secret:

```
from twython import Twython

twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET)

# search for tweets containing the phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"].encode('utf-8')
    text = status["text"].encode('utf-8')
    print user, ":", text
    print
```

The `.encode("utf-8")` is necessary to deal with the fact that tweets often contain Unicode characters that `print` can't deal with. (If you leave it out, you will very likely get a `UnicodeEncodeError`.)

It is almost certain that at some point in your data science career you will run into some serious Unicode problems, at which point you will need to refer to the Python documentation or else grudgingly start using Python 3, which plays much more nicely with Unicode text.

If you run this, you should get some tweets back like:

```
haithemnyc: Data scientists with the technical savvy &amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization.  So easy and Awesome.
```

This isn't that interesting, largely because the Twitter Search API just shows you whatever handful of recent results it feels like. When you're doing data science, more often you want a lot of tweets. This is where the Streaming API is useful. It allows you to connect to (a sample of) the great Twitter firehose. To use it, you'll need to authenticate using your access tokens.

In order to access the Streaming API with Twython, we need to define a class that inherits from TwythonStreamer and that overrides its on_success method (and possibly its on_error method):

```
from twython import TwythonStreamer

# appending data to a global variable is pretty poor form
# but it makes the example much simpler
tweets = []

class MyStreamer(TwythonStreamer):
    """our own subclass of TwythonStreamer that specifies
    how to interact with the stream"""

    def on_success(self, data):
        """what do we do when twitter sends us data?
        here data will be a Python dict representing a tweet"""

        # only want to collect English-language tweets
        if data['lang'] == 'en':
            tweets.append(data)
            print "received tweet #", len(tweets)

        # stop when we've collected enough
        if len(tweets) >= 1000:
            self.disconnect()

    def on_error(self, status_code, data):
        print status_code, data
        self.disconnect()
```

MyStreamer will connect to the Twitter stream and wait for Twitter to feed it data. Each time it receives some data (here, a Tweet represented as a Python object) it passes it to the on_success method, which appends it to our tweets list if its language is English, and then disconnects the streamer after it's collected 1,000 tweets.

All that's left is to initialize it and start it running:

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)

# starts consuming public statuses that contain the keyword 'data'
stream.statuses.filter(track='data')

# if instead we wanted to start consuming a sample of *all* public statuses
# stream.statuses.sample()
```

This will run until it collects 1,000 tweets (or until it encounters an error) and stop, at which point you can start analyzing those tweets. For instance, you could find the most common hashtags with:

```
top_hashtags = Counter(hashtag['text'].lower()
                       for tweet in tweets
                       for hashtag in tweet["entities"]["hashtags"])

print top_hashtags.most_common(5)
```

Each tweet contains a lot of data. You can either poke around yourself or dig through the Twitter API documentation.

In a non-toy project you probably wouldn't want to rely on an in-memory `list` for storing the tweets. Instead you'd want to save them to a file or a database, so that you'd have them permanently.

For Further Exploration

- pandas is the primary library that data science types use for working with (and, in particular, importing) data.
- Scrapy is a more full-featured library for building more complicated web scrapers that do things like follow unknown links.