CHAPTER 7

Data Analysis

"Data! Data! Data!" he cried impatiently. "I can't make bricks without clay."

Sherlock Holmes
The Adventure of the Copper Beeches (1892)

In this chapter, we will focus on analyzing and manipulating numerical data: earthquake measurements, SAT scores, isotope ratios, unemployment rates, meteorite locations, consumer demand, river flow, and more. Data sets such as these have become a (if not, *the*) vital component of many scientific, nonprofit, and commercial ventures. Many of these sectors now employ experts in ***data science*** who use advanced techniques to transform data into valuable information to guide the organization.

To solve these problems, we need an abstract data type (ADT) in which to store a collection of data. The simplest and most intuitive such abstraction is a *list*, which is simply a sequence of items. In previous chapters, we discovered how to generate Python `list` objects with the `range` function and how to accumulate lists of coordinates to visualize in plots. To solve the problems in this chapter, we will also grow and shrink lists, and modify and rearrange their contents, without having to worry about where or how they are stored in memory. Later in the chapter, we will develop algorithms to compute the frequencies of all the words and bigrams in a book, perform linear regression analyses, and cluster data into groups of similar items.

7.1 SUMMARY STATISTICS

Suppose we are running a small business, and we need to get some basic descriptive statistics about last week's daily sales. We can store these sales numbers in a list like this:

```
>>> sales = [32, 42, 11, 15, 58, 44, 16]
```

You'll recall that a list is represented as a sequence of items, separated by commas, and enclosed in square brackets (`[ ]`). Lists can contain any kind of data we want, even items with different types. For example, these are all valid lists:

```
>>> unemployment = [0.082, 0.092, 0.091, 0.063, 0.068, 0.052]
>>> votes = ['yea', 'yea', 'nay', 'yea', 'nay']
>>> points = [[2, 1], [12, 3], [6, 5], [3, 14]]
>>> crazyTown = [15, 'gtaaca', [1, 2, 3], max(4.1, 1.4), 'cookies']
```

Mean and variance

Let's start by computing the mean (or average) daily sales for the week. To find the mean of a list of numbers, we need to first find their sum by iterating over the list. As we saw a few times before, iterating over the values in a list is essentially identical to iterating over the characters in a string.

```
total = 0
for item in sales:
    total = total + item
return total / len(sales)
```

In each iteration of the `for` loop, `item` is assigned the next value in the list named `sales`, and then added to the running sum. After the loop, we divide the sum by the length of the list, which is retrieved with the same `len` function we used on strings.

Reflection 7.1 *Does this work when the list is empty?*

If `sales` is the empty list (`[ ]`), then the value of `len(sales)` is zero, resulting in a "division by zero" error in the `return` statement. We have several options to deal with this. First, we could just let the error happen. Second, we could use an `assert` statement to print an error message and abort. Third, we could catch the error with a `try/except` statement. Fourth, we could detect this error before it happens with an `if` statement and return something that indicates that an error occurred. We adopt the last option in the following function by returning `None` and indicating this possibility in the docstring.

```
1 def mean(data):
2     """Compute the mean of a non-empty list of numbers.

3     Parameter:
4         data: a list of numbers

5     Return value: the mean of numbers in data or None if data is empty
6     """

7     if len(data) == 0:
8         return None

9     total = 0
10    for item in data:
11        total = total + item
12    return total / len(data)
```

This `for` loop is yet another example of an accumulator. To illustrate what is happening, let's trace through the function with one week of sales data.

Trace arguments: `data = [32, 42, 11, 15, 58, 44, 16]`

Step	Line	total	item	Notes
1	7	—	—	`len(data)` is `7 > 0`; skip line 8
2	9	0	—	`total` ← `0`
3	10	″	32	`item` ← `data[0]`
4	11	32	″	`total` ← `total + item = 0 + 32`
5	10	″	42	`item` ← `data[1]`
6	11	74	″	`total` ← `total + item = 32 + 42`
⋮				
15	10	202	16	`item` ← `data[6]`
16	11	218	″	`total` ← `total + item = 202 + 16`
17	12	″	″	`return 218 / 7`

Return value: `31.142857142857142`

Reflection 7.2 *Fill in the missing steps above to show how the function arrives at a* `total` *of* `218`.

The mean of a data set does not adequately describe it if there is a lot of variability in the data, i.e., if there is no "typical" value. In these cases, we need to accompany the mean with the ***variance***, which is measure of how much the data varies from the mean. Computing the variance is left as Exercise 7.1.10.

Minimum and maximum

Now let's think about how to find the minimum and maximum sales in the list. Of course, it is easy for us to just look at a short list like the one above and pick out the minimum and maximum. But a computer does not have this ability. Therefore, as you think about these problems, it may be better to think about a very long list instead, one in which the minimum and maximum are not so obvious.

Reflection 7.3 *Think about how you would write an algorithm to find the minimum value in a long list. (Similar to a running sum, keep track of the current minimum.)*

As the hint suggests, we want to maintain the current minimum while we iterate over the list with a `for` loop. When we examine each item, we need to test whether it is smaller than the current minimum. If it is, we assign the current item to be the new minimum. The following function implements this algorithm.

```
def min(data):
    """Compute the minimum value in a non-empty list of numbers.

    Parameter:
        data: a list of numbers

    Return value: the minimum value in data or None if data is empty
    """

    if len(data) == 0:
        return None

    minimum = data[0]
    for item in data[1:]:
        if item < minimum:
            minimum = item
    return minimum
```

Since lists are sequences like strings, they can also be indexed and sliced. But now indices refer to list elements instead of characters and slices are sublists instead of substrings. We use indexing before the loop to initialize `minimum` to be the first value in the list. Then we iterate over the slice of remaining values in the list. In each iteration, we compare the current value of `item` to `minimum` and, if `item` is smaller, update `minimum` to be the value of `item`. At the end of the loop, `minimum` has been assigned the smallest value in the list.

Reflection 7.4 *If the list* `[32, 42, 11, 15, 58, 44, 16]` *is assigned to* `data`*, then what are the values of* `data[0]` *and* `data[1:]`*?*

Let's look at a small example of how this function works when we call it with the list containing only the first four numbers from the list above: `[32, 42, 11, 15]`. The function begins by assigning the value 32 to `minimum`. The first value of `item` is 42. Since 42 is not less than 32, `minimum` remains unchanged. In the next iteration of the loop, the third value in the list, 11, is assigned to `item`. In this case, since 11

is less than 32, the value of `minimum` is updated to 11. Finally, in the last iteration of the loop, `item` is assigned the value 15. Since 15 is greater than 11, `minimum` is unchanged. At the end, the function returns the final value of `minimum`, which is 11. A function to compute the maximum is very similar, so we leave it as an exercise.

Reflection 7.5 *What would happen if we iterated over* `data` *instead of* `data[1:]`*? Would the function still work?*

If we iterated over the entire list instead, the first comparison would be useless (because `item` and `minimum` would be the same) so it would be a little less efficient, but the function would still work fine.

Now what if we also wanted to know on which day of the week the minimum sales occurred? To answer this question, assuming we know how indices correspond to days of the week, we need to find the index of the minimum value in the list. As we learned in Chapter 6, we need to iterate over the indices in situations like this:

```
def minDay(data):
    """Find the index of the minimum value in a non-empty list.

    Parameter:
        data: a list of numbers

    Return value: index of the minimum value in data or -1 if data == [ ]
    """

    if len(data) == 0:
        return -1

    minIndex = 0
    for index in range(1, len(data)):
        if data[index] < data[minIndex]:
            minIndex = index
    return minIndex
```

This function performs almost exactly the same algorithm as our `min` function, but now each value in the list is identified by `data[index]` instead of `item`, and we remember the index of current minimum in the loop instead of its value.

Reflection 7.6 *How can we modify the* `minDay` *function to return a day of the week instead of an index, assuming the sales data starts on a Sunday?*

One option would be to replace `return minIndex` with `if/elif/else` statements, like the following:

```
if minIndex == 0:
    return 'Sunday'
elif minIndex == 1:
    return 'Monday'
 ⋮
else:
    return 'Saturday'
```

But a more clever solution is to create a list of the days of the week that are in the same order as the sales data. Then we can simply use the value of `minIndex` as an index into this list to return the correct string.

```
days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
        'Friday', 'Saturday']
return days[minIndex]
```

There are many other descriptive statistics that we can use to summarize the contents of a list. The following exercises challenge you to implement some of them.

Exercises

When an exercise asks you to write a function, test it with both common and boundary case arguments, and document your test cases. Also use a trace table to show the execution of the function with at least one of the test cases.

7.1.1. Suppose a list is assigned to the variable name `data`. Show how you would

(a)* print the length of `data`

(b)* print the third element in `data`

(c)* print the last three elements in `data`

(d) print the last element in `data`

(e) print the first four elements in `data`

(f) print the list consisting of the second, third, and fourth elements in `data`

7.1.2. In the `mean` function, we returned `None` if `data` was empty. Show how to modify the following `main` function so that it properly tests for this possibility and prints an appropriate message.

```
def main():
    someData = getInputFromSomewhere()
    average = mean(someData)
    print('The mean value is ' + str(average) + '.')
```

7.1.3. Write a function

```
sumList(data)
```

that returns the sum of all of the numbers in the list `data`. For example, `sumList([1, 2, 3])` should return 6.

7.1.4* Write a function

```
sumOdds(data)
```

that returns the sum of only the odd integers in the list `data`. For example, `sumOdds([1, 2, 3])` should return 4.

7.1.5. Write a function

```
countOdds(data)
```

that returns the number of odd integers in the list `data`. For example, `countOdds([1, 2, 3])` should return 2.

7.1.6. Write a function

```
multiples5(data)
```

that returns the number of multiples of 5 in a list of integers. For example, `multiples5([5, 7, 2, 10])` should return 2.

7.1.7. Write a function

```
countNames(words)
```

that returns the number of capitalized names in the list of strings named `words`. For example,

```
countNames(['Fili', 'Oin', 'Thorin', 'and', 'Bilbo', 'are',
            'characters', 'in', 'a', 'book', 'by', 'Tolkien'])
```

should return 5.

7.1.8. The percentile associated with a particular value in a data set is the number of values that are less than or equal to it, divided by the total number of values, times 100. Write a function

```
percentile(data, value)
```

that returns the percentile of `value` in the list named `data`.

7.1.9. Write a function

```
meanSquares(data)
```

that returns the mean of the squares of the numbers in a list named `data`.

7.1.10. Write a function

```
variance(data)
```

that returns the variance of a list of numbers named `data`. The variance is defined to be the mean of the squares of the numbers in the list minus the square of the mean of the numbers in the list. In your implementation, call your function from Exercise 7.1.9 and the `mean` function from this section.

7.1.11. Write a function

```
max(data)
```

that returns the maximum value in the list of numbers named `data`. Do not use the built-in `max` function.

7.1.12* Write a function

```
shortest(words)
```

that returns the shortest string in a list of strings named `words`. In case of ties, return the first shortest string. For example,

```
shortest(['spider', 'ant', 'beetle', 'bug'])
```

should return the string `'ant'`.

7.1.13. Write a function

```
span(data)
```

that returns the difference between the largest and smallest numbers in the list named `data`. Do not use the built-in `min` and `max` functions. (But you may use your own functions.) For example, `span([9, 4, 1, 7, 7])` should return 8.

7.1.14. Write a function

```
maxIndex(data)
```

that returns the *index* of the maximum item in the list of numbers named `data`. Do not use any built-in functions.

7.1.15. Write a function

```
secondLargest(data)
```

that returns the second largest number in `data`. Do not use the built-in `max` function. (But you may use your `maxIndex` function from Exercise 7.1.14.)

7.1.16* Write a function

```
linearSearch(data, target)
```

that returns `True` if `target` is in the list named `data`, and `False` otherwise. Do not use the `in` operator to test whether an item is in the list. For example,

```
linearSearch(['Tris', 'Tobias', 'Caleb'], 'Tris')
```

should return `True`, but

```
linearSearch(['Tris', 'Tobias', 'Caleb'], 'Peter')
```

should return `False`.

7.1.17* Write a function

```
linearSearch(data, target)
```

that returns the index of `target` if it is found in `data`, and `-1` otherwise. Do not use the `in` operator or the `index` method to test whether items are in the list. For example,

```
linearSearch(['Tris', 'Tobias', 'Caleb'], 'Tris')
```

should return `0`, but

```
linearSearch(['Tris', 'Tobias', 'Caleb'], 'Peter')
```

should return `-1`.

7.1.18* Write a function

```
intersect(data1, data2)
```

that returns `True` if the two lists named `data1` and `data2` have any common elements, and `False` otherwise. (You may use your `linearSearch` function from Exercise 7.1.16.) For example,

```
intersect(['Katniss', 'Peeta', 'Gale'], ['Foxface', 'Marvel'])
```

should return `False`, but

```
intersect(['Katniss', 'Peeta', 'Gale'], ['Gale', 'Haymitch'])
```

should return `True`.

7.1.19. Write a function

```
differ(data1, data2)
```

that returns the first index at which the two lists `data1` and `data2` differ. If the two lists are the same, your function should return `-1`. You may assume that the lists have the same length. For example,

```
differ(['CS', 'rules', '!'], ['CS', 'totally', 'rules!'])
```

should return the index 1.

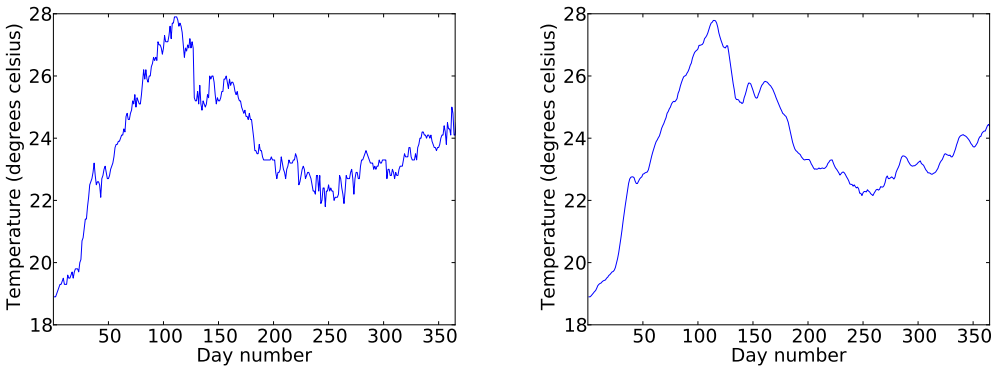


Figure 7.1 Plots of (a) daily temperatures and (b) the same temperatures smoothed over a five-day window.

7.1.20. The Luhn algorithm is the standard algorithm used to validate credit card numbers and protect against accidental errors. Read about the algorithm online, and then write a function

```
validateLuhn(number)
```

that returns `True` if the number if valid and `False` otherwise. The `number` parameter will be a list of digits. For example, to determine if the credit card number 4563 9601 2200 1999 is valid, one would call the function with the parameter `[4, 5, 6, 3, 9, 6, 0, 1, 2, 2, 0, 0, 1, 9, 9, 9]`. (Hint: use a `for` loop that iterates in reverse over the indices of the list.)

7.2 WRANGLING DATA

Suppose, as part of an ongoing climate study, we are tracking daily surface seawater temperatures recorded by a drifting buoy in the Atlantic Ocean.[1] Our list of daily temperature readings (in degrees Celsius) starts like this:

```
[18.9, 18.9, 19.0, 19.2, 19.3, 19.3, 19.2, 19.1, 19.4, 19.3, ...]
```

Often, when we are dealing with data sets like this, anomalies can arise due to errors in the sensing equipment, human fallibility, or corruption in the network used to send results to a lab or another collection point. We can mask these erroneous measurements by "smoothing" the data, replacing each value with the mean of the values in a "window" of values containing it. This technique is also useful for extracting general patterns in data by eliminating distracting "bumpy" areas. For example, Figure 7.1 shows a year's worth of raw temperature data from an actual ocean buoy, next to the same data smoothed over a five day window. Smoothing data like this is sometimes also called computing *moving averages*, as in "the 5-day moving average price of the stock is $27.13."

[1] For example, see `http://www.coriolis.eu.org`.

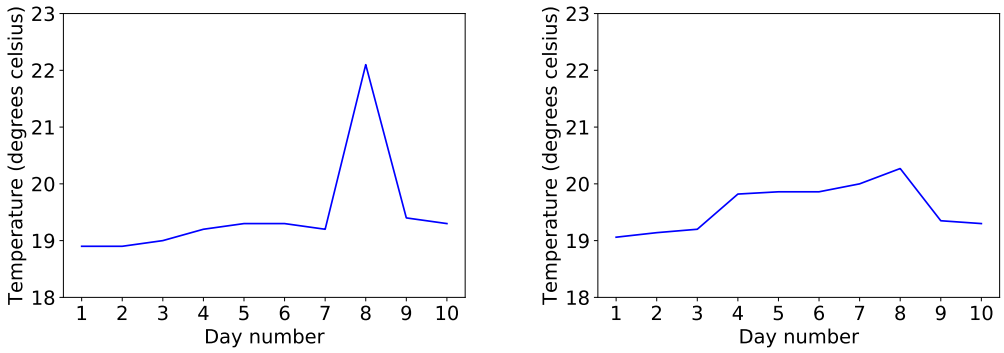


Figure 7.2 Plots of (a) ten daily temperatures and (b) the same temperatures smoothed over a five-day window.

Smoothing data

Let's design an algorithm for this problem. We will begin by looking at a small example consisting of the ten temperature readings above, with an anomalous reading inserted, marked in red:

```
[18.9, 18.9, 19.0, 19.2, 19.3, 19.3, 19.2, 22.1, 19.4, 19.3]
```

The plot of this data in Figure 7.2(a) illustrates this erroneous "bump."

Now let's smooth the data by averaging over windows of size five. For each value in the original list, its window will include itself and the four values that come after it. (The last four values do not have four values after them, so their windows will be smaller.) Our algorithm will need to compute the mean of each of these windows, and then add each of these means to a new smoothed list. The first window looks like this:

```
[18.9, 18.9, 19.0, 19.2, 19.3, 19.3, 19.2, 22.1, 19.4, 19.3]
 └──────────────────────┘
 mean = 95.3 / 5 = 19.06
```

To find the mean temperature for the window, we sum the five values and divide by 5. The result, 19.06, will represent this window in the smoothed list. The remaining windows are computed in the same way:

```
[18.9, 18.9, 19.0, 19.2, 19.3, 19.3, 19.2, 22.1, 19.4, 19.3]
       └──────────────────────┘
       mean = 95.7 / 5 = 19.14

                  ⋮

[18.9, 18.9, 19.0, 19.2, 19.3, 19.3, 19.2, 22.1, 19.4, 19.3]
                               └──────────────────────┘
                               mean = 99.3 / 5 = 19.86
```

```
[18.9, 18.9, 19.0, 19.2, 19.3, 19.3, [19.2, 22.1, 19.4, 19.3]]
                                      mean = 80.0 / 4 = 20.00
[18.9, 18.9, 19.0, 19.2, 19.3, 19.3, 19.2, [22.1, 19.4, 19.3]]
                                           mean = 60.8 / 3 = 20.27
[18.9, 18.9, 19.0, 19.2, 19.3, 19.3, 19.2, 22.1, [19.4, 19.3]]
                                                 mean = 38.7 / 2 = 19.35
[18.9, 18.9, 19.0, 19.2, 19.3, 19.3, 19.2, 22.1, 19.4, [19.3]]
                                                       mean = 19.3 / 1 = 19.3
```

In the end, our list of smoothed temperature readings is:

```
[19.06, 19.14, 19.20, 19.82, 19.86, 19.86, 20.0, 20.27, 19.35, 19.3]
```

We can see from the plot of this smoothed list in Figure 7.2(b) that the "bump" has indeed been smoothed (although, due to the small window size, it still causes the window means to increase more than we would probably like).

Suppose that our list of values is named `data` and our desired window size is assigned to variable named `width`. Then we can realize this algorithm with a ***list accumulator*** just like the ones we have been using to plot data since Chapter 4.

```
smoothedData = [ ]                          # initialize list
for index in range(len(data)):
    window = data[index:index + width]
    smoothedData.append(mean(window))       # append the next mean
```

Before the loop, the list `smoothedData` is initialized to be an empty list. In each iteration, we call our `mean` function from the previous section on one window, and append the result to the list of smoothed data. After the loop, we have "acccumulated" the means for all of the windows.

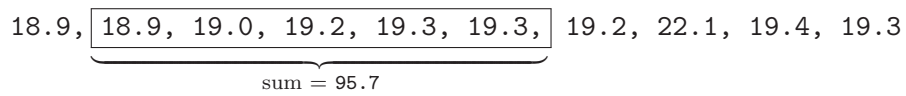
A more efficient algorithm

Reflection 7.7 *Can you think of a way to solve the smoothing problem with fewer arithmetic operations?*

We can design a more efficient algorithm by exploiting the simple observation that, while finding the sums of neighboring windows in the `mean` function, we unnecessarily performed some addition operations multiple times. For example, we added the fourth and fifth temperature readings four different times, once in each of the first four windows. We can eliminate this extra work by taking advantage of the relationship between the sums of two contiguous windows. For example, consider the first window:

```
[ 18.9, 18.9, 19.0, 19.2, 19.3] 19.3, 19.2, 22.1, 19.4, 19.3
 sum = 95.3
```

The sum for the second window must be almost the same as the first window, since they have four numbers in common. The only difference in the second window is that it loses the first `18.9` and gains `19.3`. So once we have the sum for the first window (95.3), we can get the sum of the second window with only two additional arithmetic operations: 95.3 – 18.9 + 19.3 = 95.7.



We can apply this process to every subsequent window as well, as demonstrated by the following improved algorithm.

```
1 def smooth(data, width):
2     """Return a new list of data, smoothed over windows of the given width.

3     Parameters:
4         data:  a list of numbers
5         width: the width of each window

6     Return value: a list of smoothed data values
7     """

8     smoothedData = []
9     total = 0                                 # get sum for the first window
10    for index in range(width):
11        total = total + data[index]

12    for index in range(len(data)):
13        width = min(width, len(data) - index) # adjust width near the end
14        smoothedData.append(total / width)    # append the window mean
15        total = total - data[index]           # subtract leftmost value
16        if index + width < len(data):           # if possible,
17            total = total + data[index + width] #   add rightmost value

18    return smoothedData
```

At the beginning of the function, in lines 8–11, we initialize a new list and get the sum of the values in the first window. Then, in the `for` loop, in lines 13–14, the mean for the window is appended to the new list. Before the mean is computed, if necessary, the `width` of the window is reduced to the number of remaining values in `data`. In lines 15–17, the sum is adjusted for the next iteration by subtracting the leftmost value in the current window and, if possible, adding the next value after the current window.

Reflection 7.8 *To make sure you understand the algorithm, use a trace table to execute the* `smooth` *function on the list above with a window size of 5. You should get the same smoothed list that we derived previously.*

To see how much better this algorithm is than our original, let's compare the number

of arithmetic operations that are performed by each algorithm, since this is the vast majority of the work taking place. In the first algorithm, if each window has size 5, then we perform five addition operations and one division operation each time we call the `mean` function, for a total of six arithmetic operations per full-size window. Therefore, the total number of arithmetic operations is at most six times the number of windows. In general, if the window width is denoted w, the algorithm performs at most w additions and one division per window, for a total of $n(w+1)$ arithmetic operations.

The new algorithm is performing w additions in the first `for` loop. In each iteration of the second `for` loop, it is performing a division and at most two additions, for a total of at most $3n$ arithmetic operations. In total then, the new algorithm performs at most $3n + w$ arithmetic operations. Therefore, our old algorithm requires

$$\frac{n(w+1)}{3n+w}$$

times as many operations as the new one. It may be hard to tell from this ratio, but our new algorithm is doing about $w/3$ times less work. To see this more concretely, suppose our list contains ten years (about 3,652 days) worth of temperature readings, so the speedup ratio is

$$\frac{3652\,(w+1)}{3 \cdot 3652 + w}.$$

The following table shows the value of this fraction for increasing window sizes w.

w	Speedup
5	2.0
10	3.7
20	7.0
100	33.4

When w is small, our new algorithm does not make much difference, but the speedup becomes quite pronounced when w gets larger. In real applications of smoothing on extremely large data sets containing billions or trillions of items, such as statistics on DNA sequences, window sizes can be as high as w = 100,000. So our refined algorithm can have a marked impact!

Modifying lists in place

We can modify existing lists with `append` because, unlike strings, lists are ***mutable***. In other words, the items in a list can be changed directly. This means that we could smooth the data in a list by overwriting each individual item in the list itself instead of creating a new list. This is often referred to as modifying the list *in place*.

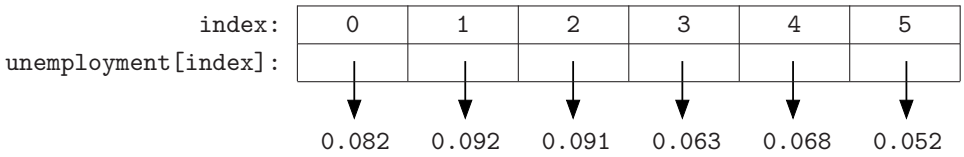
Before we tackle that problem, let's look at a simpler problem to illustrate the mechanics involved. Suppose we have a list of monthly unemployment rates like those below.

```
>>> unemployment = [0.082, 0.092, 0.091, 0.063, 0.068, 0.052]
```

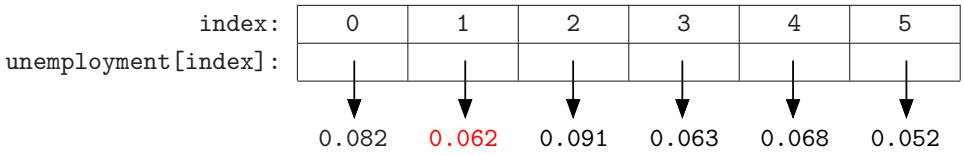
If we need to change the second value in the list, we can do so like this:

```
>>> unemployment[1] = 0.062
>>> unemployment
[0.082, 0.062, 0.091, 0.063, 0.068, 0.052]
```

We can change individual elements in a list because each of the elements is an independent reference to a value, like any other variable name. We can visualize the original `unemployment` list like this:



The value `0.082` is assigned to the name `unemployment[0]`, the value `0.092` is assigned to the name `unemployment[1]`, etc. When we assigned a new value to `unemployment[1]`, we were simply assigning a new value to the name `unemployment[1]`, like any other assignment statement:



Suppose we wanted to adjust all of the unemployment rates in this list by subtracting one percent from each of them. We can do this with a `for` loop that iterates over the indices of the list.

```
>>> for index in range(len(unemployment)):
        unemployment[index] = unemployment[index] - 0.01
>>> unemployment
[0.072, 0.052, 0.081, 0.053, 0.058, 0.042]
```

This `for` loop is equivalent to the following six assignment statements:

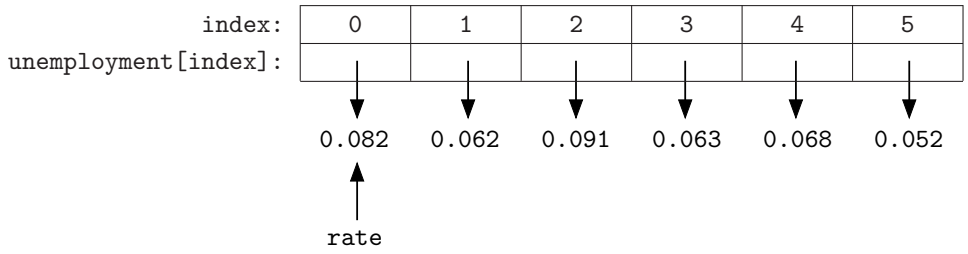
```
unemployment[0] = unemployment[0] - 0.01
unemployment[1] = unemployment[1] - 0.01
unemployment[2] = unemployment[2] - 0.01
unemployment[3] = unemployment[3] - 0.01
unemployment[4] = unemployment[4] - 0.01
unemployment[5] = unemployment[5] - 0.01
```

Reflection 7.9 *Is it possible to modify a list in place by iterating over the values in the list instead? In other words, does the following* `for` *loop accomplish the same thing? (Try it.) Why or why not?*

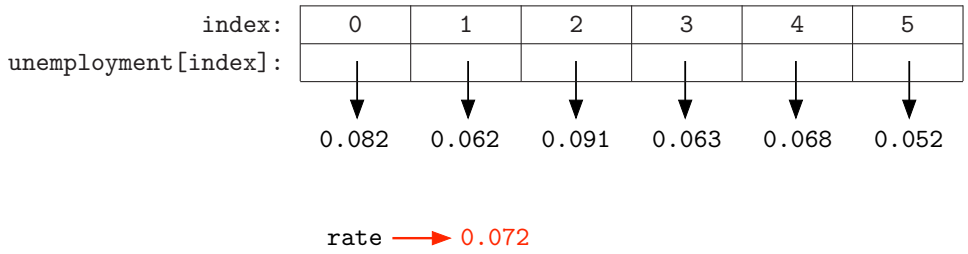
```
for rate in unemployment:
    rate = rate - 0.01
```

This loop does *not* modify the list because `rate`, which is being modified, is not

actually a name in the list. Instead, it is being assigned the same value as an item in the list. So, although the value assigned to `rate` is being modified, the list itself is not. As illustrated below, at the beginning of the first iteration, `0.082` is assigned to `rate`.



Then, when the modified value `rate - 0.01` is assigned to `rate`, this only affects `rate`, not the original list, as illustrated below.



Now let's put the correct loop above in a function named `adjust` that takes a list of unemployment rates as a parameter.

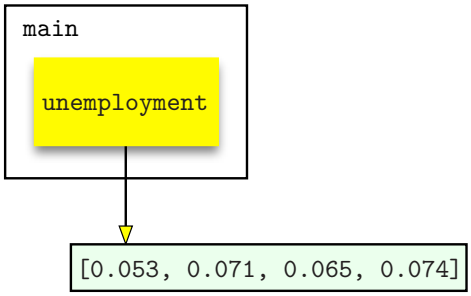
```
def adjust(rates):
    """Subtract one percent (0.01) from each rate in a list.

    Parameter:
        rates: a list of numbers representing rates (percentages)

    Return value: None
    """

    for index in range(len(rates)):
        rates[index] = rates[index] - 0.01
```

In the following `main` function, the list named `unemployment` is passed into the `adjust` function for the parameter `rates`.

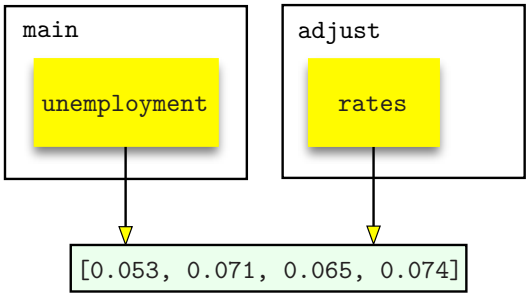
```
def main():
    unemployment = [0.053, 0.071, 0.065, 0.074]
    adjust(unemployment)
    print(unemployment)

main()
```

Inside the `adjust` function, every value in `rates` is decremented by 0.01. What effect, if any, does this have on the list assigned to `unemployment`? To find out, we need to look carefully at what happens when the function is called.

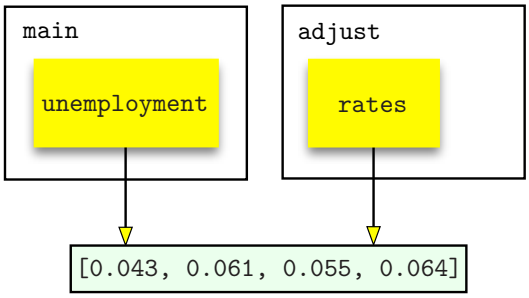
Right after the assignment statement in the `main` function, the situation looks like the following, with the variable named `unemployment` in the `main` namespace assigned the list `[0.053, 0.071, 0.065, 0.074]`.



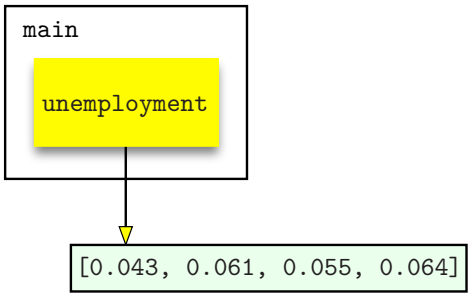
Now recall from Section 2.5 that, when an argument is passed to a function, it is *assigned* to its associated parameter. Therefore, immediately after the `adjust` function is called from `main`, the parameter `rates` is assigned the same list as `unemployment`:



After `adjust` executes, 0.01 has been subtracted from each value in `rates`, as the following picture illustrates.



Notice that, since the same list is assigned to `unemployment`, these changes will also be reflected in the value of `unemployment` back in the `main` function. In other words, after the `adjust` function returns, the picture looks like this:



So when `unemployment` is printed at the end of `main`, the adjusted list `[0.043, 0.061, 0.055, 0.064]` will be displayed.

Reflection 7.10 *Why does the argument's value change in this case when it did not in the parameter passing examples in Section 2.5? What is different?*

The difference here is that lists are mutable. When you pass a mutable type as an argument, any changes to the associated formal parameter inside the function will be reflected in the value of the argument. Therefore, when we pass a list as an argument to a function, the values in the list can be changed inside the function.

What if we did *not* want to change `unemployment` when we passed it to the `adjust` function, and instead return a *new* adjusted list? One alternative, illustrated in the function below, would be to make a copy of `rates`, using the list method `copy`, and then modify this copy instead.

```
def adjust(rates):
    """ (docstring omitted) """

    ratesCopy = rates.copy()
    for index in range(len(ratesCopy)):
        ratesCopy[index] = ratesCopy[index] - 0.01
    return ratesCopy
```

The `copy` method creates an independent copy of the list in memory, and returns a reference to this new list so that it can be assigned to a variable name (in this case, `ratesCopy`). There are other solutions to this problem as well, which we leave as exercises.

With this knowledge in hand, we can return to the problem of smoothing data in place. The following modified function does this by replacing each value of `data[index]` with the mean of the window starting at position `index`.

```
1 def smoothInPlace(data, width):
2     """Smooth data in place over windows with the given width.

3     Parameters:
4         data:  a list of numbers
5         width: the width of each window

6     Return value: None
7     """
```

```
8     total = 0                           # get the sum for the first window
9     for index in range(width):
10        total = total + data[index]

11    for index in range(len(data)):
12        width = min(width, len(data) - index) # adjust width near the end
13        mean = total / width                  # compute the window mean
14        total = total - data[index]           # subtract leftmost value
15        if index + width < len(data):           # if possible,
16            total = total + data[index + width] #   add rightmost value
17        data[index] = mean                    # replace with window mean
```

The first things to notice are that the function has no `smoothedData` list and it does not return anything because smoothing will be reflected directly in `data`. The only two other changes are highlighted. In the first highlighted line, the mean value is computed for the window, but not yet changed in the list. The mean is assigned to `data[index]` in the second highlighted line, at the bottom of the loop. The reason for the separation between these two steps is subtle but important.

Reflection 7.11 *Why can we not change the first highlighted line to* `data[index] = total / width`*?*

If we assigned the window mean to `data[index]` on line 13, then we would be overwriting the original value of `data[index]` that we need in line 14! The ordering of events here is crucial: computing the mean of the current window in line 13 must come before `total` is modified for the next window in lines 14–16, and subtracting `data[index]` from `total` on line 14 must come before updating `data[index]` on line 17.

List operators and methods

In addition to `append`, there are several more ways to create and modify lists. These will be especially helpful in the projects at the end of this chapter.

List operators

First, the two operators that we used to create new strings can also be used to create new lists. The repetition operator `*` creates a new list that is built from repeats of the contents of a smaller list. For example:

```
>>> empty = [0] * 5
>>> empty
[0, 0, 0, 0, 0]
>>> ['up', 'down'] * 4
['up', 'down', 'up', 'down', 'up', 'down', 'up', 'down']
```

The concatenation operator `+` creates a new list that is the result of "sticking together" two lists. For example:

```
>>> unemployment = [0.082, 0.092, 0.091, 0.063, 0.068, 0.052]
```

```
>>> unemployment = unemployment + [0.087, 0.101]
>>> unemployment
[0.082, 0.092, 0.091, 0.063, 0.068, 0.052, 0.087, 0.101]
```

Notice that the concatenation operator combines two lists to create a new list, whereas the `append` method *inserts* a new element into the end of an existing list. In other words,

```
unemployment = unemployment + [0.087, 0.101]
```

accomplishes the same thing as the two statements

```
unemployment.append(0.087)
unemployment.append(0.101)
```

However, using concatenation actually creates a new list that is then assigned to `unemployment`, whereas using `append` modifies an existing list. So using `append` is usually more efficient than concatenation if you are just adding to the end of an existing list.

Sorting a list

The `sort` method sorts the items in a list in increasing order. For example, suppose we have a list of SAT scores that we would like to sort:

```
>>> scores = [620, 710, 520, 550, 640, 730, 600]
>>> scores.sort()
>>> scores
[520, 550, 600, 620, 640, 710, 730]
```

It is worth emphasizing that none of these list methods return new lists; instead they modify the lists in place. In other words, the following is a mistake:

```
>>> scores = [620, 710, 520, 550, 640, 730, 600]
>>> newScores = scores.sort()   # MISTAKE
```

Reflection 7.12 *What is the value of* `newScores` *after we execute the statements above?*

Printing the value of `newScores` reveals that it refers to the value `None` because `sort` does not return anything. However, `scores` was modified as we expected:

```
>>> newScores
>>> scores
[520, 550, 600, 620, 640, 710, 730]
```

The `sort` method will sort any list that contains comparable items, including strings. For example, suppose we have a list of names that we want to be in alphabetical order:

```
>>> names = ['Eric', 'Michael', 'Connie', 'Graham']
>>> names.sort()
>>> names
['Connie', 'Eric', 'Graham', 'Michael']
```

Reflection 7.13 *What happens if you try to sort a list containing items that cannot be compared to each other? For example, try sorting the list* `[3, 'one', 4, 'two']`.

Inserting and deleting items

The `insert` method inserts an item into a list at a particular index. For example, suppose we want to insert new names into the sorted list above to maintain alphabetical order:

```
>>> names.insert(3, 'John')
>>> names
['Connie', 'Eric', 'Graham', 'John', 'Michael']
>>> names.insert(0, 'Carol')
>>> names
['Carol', 'Connie', 'Eric', 'Graham', 'John', 'Michael']
```

The first parameter of the `insert` method is the index where the inserted item will reside *after* the insertion.

The `pop` method is the inverse of `insert`; `pop` deletes the list item at a given index and returns the deleted value. For example,

```
>>> inMemoriam = names.pop(3)
>>> names
['Carol', 'Connie', 'Eric', 'John', 'Michael']
>>> inMemoriam
'Graham'
```

If the argument to `pop` is omitted, `pop` deletes and returns the last item in the list.

Using `pop` in a loop can be tricky. To see why, try this loop, which is meant to delete all of the items in `names` that start with a *C*.

```
>>> for index in range(len(names)):
        name = names[index]
        if name[0] == 'C':
            names.pop(index)

'Carol'
IndexError: list index out of range
```

Reflection 7.14 *Where and why does the* `IndexError` *occur in this loop? Were both* `'Carol'` *and* `'Connie'` *deleted as intended?*

The error occurs because, after each call to `pop` is executed, the `names` list becomes shorter. But the loop is still going to iterate through the length of the original list because `len(names)` was evaluated before the loop started. So, in the latter iterations, the value of `index` will be beyond the end of the modified list, triggering an `IndexError` exception. There is also a more subtle issue. The condition in the `if` statement is first true when the value of `index` is 0, causing `'Carol'` to be popped. After this happens, the list looks like this: `['Connie', 'Eric', 'John', 'Michael']`. In the next iteration of the loop, `index` will be 1, so `names[index]` will be `'Eric'`. But `'Connie'`, which is now at index 0, was skipped! To avoid this mistake, the loop needs to not increment `index` when an item is popped.

To fix these errors, we need to use a `while` loop. In a `while` loop, we can make sure

that the value of `len(names)` will be re-evaluated before each iteration and that `index` is only incremented when an item is not popped:

```
>>> index = 0
>>> while index < len(names):
        name = names[index]
        if name[0] == 'C':
            names.pop(index)
        else:
            index = index + 1
```

Reflection 7.15 *Reset* `names` *to be* `['Carol', 'Connie', 'Eric', 'John', 'Michael']` *and try this new loop. How does it fix the problems in the* `for` *loop above?*

The `remove` method also deletes an item from a list, but takes the *value* of an item as its parameter rather than its index. If there are multiple items in the list with the given value, the `remove` method only deletes the first one. For example,

```
>>> names.remove('John')
>>> names
['Carol', 'Connie', 'Eric', 'Michael']
```

Reflection 7.16 *What happens if you try to remove* `'Graham'` *from* `names`*?*

As you saw, the `remove` method raises a `ValueError` exception if its argument is not found. It is usually a good idea to catch this exception rather than have your program abort. If it doesn't matter whether the argument is found, you can catch the exception but do nothing:

```
try:
    names.remove('Graham')
except ValueError:
    pass
```

*List comprehensions

The list accumulator pattern is so common that there is a shorthand for it in Python called a ***list comprehension***. A list comprehension allows us to build up a list in a single statement. For example, suppose we wanted to create a list of the first 15 even numbers. Using a `for` loop, we can construct the desired list with:

```
evens = [ ]
for i in range(15):
    evens.append(2 * i)
```

An equivalent list comprehension looks like this:

```
evens = [2 * i for i in range(15)]
```

The first part of the list comprehension is an expression representing the items we want in the list. This is the same as the expression that would be passed to the `append` method if we constructed the list the "long way" with a `for` loop. This expression is followed by a `for` loop clause that specifies the values of an index variable for which the expression should be evaluated. The `for` loop clause is also

Tangent 7.1: NumPy arrays

NumPy is a Python module that provides a different list-like class named `array`. (Because the `numpy` module is required by the `matplotlib` module, you should already have it installed.) Unlike a list, a NumPy `array` is treated as a mathematical ***vector***. There are several different ways to create a new array. We will only illustrate two:

```
>>> import numpy
>>> a = numpy.array([1, 2, 3, 4, 5])
>>> print(a)
[1 2 3 4 5]
>>> b = numpy.zeros(5)
>>> print(b)
[ 0.  0.  0.  0.  0.]
```

In the first case, `a` was assigned an `array` created from a list of numbers. In the second, `b` was assigned an `array` consisting of 5 zeros. One advantage of an `array` over a `list` is that arithmetic operations and functions are applied to each of an `array` object's elements individually. For example:

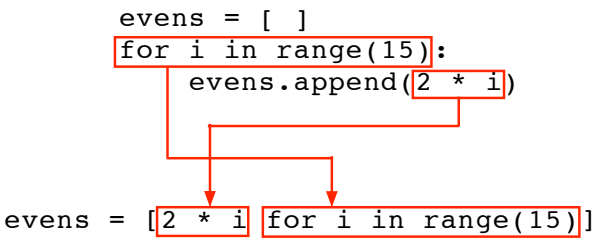
```
>>> print(a * 3)
[ 3  6  9 12 15]
>>> c = numpy.array([3, 4, 5, 6, 7])
>>> print(a + c)
[ 4  6  8 10 12]
```

There are also many functions and methods available to `array` objects. For example:

```
>>> print(c.sum())
25
>>> print(numpy.sqrt(c))
[ 1.73205081  2.          2.23606798  2.44948974  2.64575131]
```

An `array` object can also have more than one dimension, as we will discuss in Chapter 8. If you are interested in learning more about NumPy, visit `http://www.numpy.org`.

identical to the `for` loop that we would use to construct the list the "long way." This correspondence is illustrated below:



List comprehensions can also incorporate `if` statements. For example, suppose we wanted a list of the first 15 even numbers that are not divisible by 6. A `for` loop

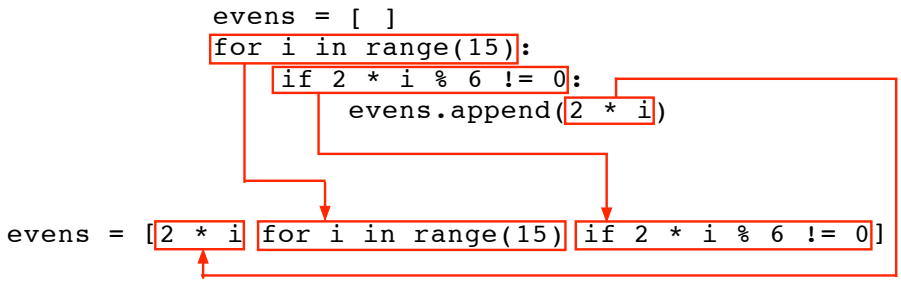
to create this list would look just like the previous example, with an additional `if` statement that checks that `2 * i` is not divisible by 6 before appending it:

```
evens = [ ]
for i in range(15):
    if 2 * i % 6 != 0:
        evens.append(2 * i)
```

This can be reproduced with a list comprehension that looks like this:

```
evens = [2 * i for i in range(15) if 2 * i % 6 != 0]
```

The corresponding parts of this loop and list comprehension are illustrated below:



In general, the initial expression in a list comprehension can be followed by any sequence of `for` and `if` clauses that specify the values for which the expression should be evaluated.

Exercises

When an exercise asks you to write a function, test it with both common and boundary case arguments, and document your test cases.

7.2.1* Show how to add the string `'grapes'` to the end of the following list using both concatenation and the `append` method.

```
fruit = ['apples', 'pears', 'kiwi']
```

7.2.2* Write a function

```
squares(n)
```

that returns a list containing the squares of the integers 1 through `n`. Use a `for` loop.

7.2.3. Write a function

```
getCodons(dna)
```

that returns a list containing the codons in the string `dna`. Your algorithm should use a `for` loop. (This exercise assumes that you have read Section 6.8.)

7.2.4* Write a function

```
square(data)
```

that takes a list of numbers named `data` and squares each number in `data` in place. The function should not return anything. For example, if the list `[4, 2, 5]` is assigned to a variable named `numbers` then, after calling `square(numbers)`, `numbers` should have the value `[16, 4, 25]`.

7.2.5* Write a function

```
swap(data, i, j)
```

that swaps the positions of the items with indices `i` and `j` in the list named `data`.

7.2.6. Write a function

```
reverse(data)
```

that reverses the list `data` in place. Your function should not return anything. (Hint: use the `swap` function you wrote above.)

7.2.7. Suppose you are given a list of `'yea'` and `'nay'` votes. Write a function

```
winner(votes)
```

that returns the majority vote. For example, `winner(['yea', 'nay', 'yea'])` should return `'yea'`. If there is a tie, return `'tie'`.

7.2.8* Write a function

```
delete(data, index)
```

that returns a new list that contains the same elements as the list `data` except for the one at the given `index`. If the value of `index` is negative or exceeds the length of `data`, return a copy of the original list. Do not use the `pop` method. For example, `delete([3, 1, 5, 9], 2)` should return the list `[3, 1, 9]`.

7.2.9. Write a function

```
removeAll(data, value)
```

that returns a new list that contains the same elements as the list `data` except for those that equal `value`. Do not use the builit-in `remove` method. Note that, unlike the built-in `remove` method, your function should remove *all* items equal to `value`. For example, `remove([3, 1, 5, 3, 9], 3)` should return the list `[1, 5, 9]`.

7.2.10. Write a function

```
centeredMean(data)
```

that returns the average of the numbers in `data` with the largest and smallest numbers removed. You may assume that there are at least three numbers in the list. For example, `centeredMean([2, 10, 3, 5])` should return `4`.

7.2.11. On page 301, we showed one way to write the `adjust` function so that it returned an adjusted list rather than modifying the original list. Give another way to accomplish the same thing.

7.2.12. Write a function

```
shuffle(data)
```

that shuffles the items in the list named `data` in place, without using the `shuffle` function from the `random` module. Instead, use the `swap` function you wrote in Exercise 7.2.5 to swap 100 pairs of randomly chosen items. For each swap, choose a random index for the first item and then choose a greater random index for the second item.

7.2.13. Write a function

```
median(data)
```

that returns the median number in a list of numbers named `data`.

7.2.14. Consider the following alphabetized grocery list:

```
groceries = ['cookies', 'gum', 'ham', 'ice cream', 'soap']
```

Show a sequence of calls to list methods that insert each of the following into their correct alphabetical positions, so that the final list containing nine items is alphabetized:

(a)* `'jelly beans'`

(b)* `'donuts'`

(c) `'bananas'`

(d) `'watermelon'`

Next, show a sequence of calls to the `pop` method that delete all of the following items from the final list above.

(e)* `'soap'`

(f)* `'watermelon'`

(g) `'bananas'`

(h) `'ham'`

7.2.15* *Stop words* are common words that are often ignored when a text is analyzed.

(a) Write a function

```
getStopWords(fileName)
```

that reads stop words from the file with the given `fileName` and returns them in a list. The file will contain one word per line; some stop word files are available on the book website. Remove all punctuation from the words using the `removePunctuation` function from Section 6.1.

(b) Write a function

```
removeStopWords(wordList, stopWordList)
```

that uses `pop` to delete all of the stop words in `stopWordList` from the list of words named `wordList`.

7.2.16. Write a function

```
removeNumbers(wordList)
```

that uses `pop` to delete all of the words that begin with a digit from the list of words named `wordList`.

7.2.17. The string method `join` takes a list of strings as an argument and creates a new string that is the concatenation of the strings in the list, separated by the string object on which the method is operating. For example, `', '.join('cookies', 'gum', 'ham')` returns the string `'cookies, gum, ham'` and `'/'.join('either', 'or')` returns the string `'either/or'`.

Rewrite your `makeQuestion` function from Exercise 6.4.6 by using `textlib.wordTokens` from Section 6.1 to split the `sentence` into a list words, and then using `join` to create the question from this list.

7.2.18. Given n people in a room, what is the probability that at least one pair of people shares a birthday? To answer this question, first write a function

```
sameBirthday(numPeople)
```

that creates a list of `numPeople` random birthdays and returns `True` if two birthdays are the same, and `False` otherwise. Use the numbers 0 to 364 to represent 365 different birthdays. Next, write a function

```
birthdayProblem(numPeople, trials)
```

that performs a Monte Carlo simulation with the given number of `trials` to approximate the probability that, in a room with `numPeople` people, two people share a birthday.

7.2.19. Write a function

```
birthdayProblem2(trials)
```

that uses your `birthdayProblem` function from the previous problem to return the smallest number of people for which the probability of a pair sharing a birthday is at least 0.5.

7.2.20* Rewrite the `squares` function from Exercise 7.2.2 using a list comprehension.

7.2.21. Rewrite the `remove` function from Exercise 7.2.9 using a list comprehension.

7.2.22. Rewrite the `getCodons` function from Exercise 7.2.3 using a list comprehension. (This exercise assumes that you have read Section 6.8.)

7.3 TALLYING FREQUENCIES

In Section 6.5, we designed an algorithm to track the frequency of specific words across a text. In this section, we will use a new kind of abstract data type to record the frequencies of *every* word in a text. Word frequencies are the most basic building blocks of algorithms used to discover patterns and themes in raw texts.

Word frequencies

To find the number of times that each word appears in a text, we can iterate over the list of words, keeping track of how many times we see each word. We can imagine using a simple table for this purpose. To illustrate, suppose we have the following very short text and associated list of words, obtained by calling the `wordTokens` function from Section 6.1:

```
>>> import textlib
>>> drSeuss = 'one fish two fish red fish blue fish'
>>> wordList = textlib.wordTokens(drSeuss)
>>> wordList
['one', 'fish', 'two', 'fish', 'red', 'fish', 'blue', 'fish']
```

Upon seeing the first word in the list, `'one'`, we create an entry in the table for it and add a tally mark.

Word:	`'one'`
Frequency:	|

The second word in the list is `'fish'`, so we create another entry and tally mark.

Word:	`'one'`	`'fish'`
Frequency:	|	|

The third word is `'two'`, so we create a third entry and tally mark.

Word:	`'one'`	`'fish'`	`'two'`
Frequency:	|	|	|

The fourth word is `'fish'` again, so we add a tally mark to that entry.

Word:	`'one'`	`'fish'`	`'two'`
Frequency:	|	||	|

Continuing in this way with the rest of the list, we get the following final table.

Word:	`'one'`	`'fish'`	`'two'`	`'red'`	`'blue'`
Frequency:	|	||||	|	|	|

Or, equivalently:

Word:	`'one'`	`'fish'`	`'two'`	`'red'`	`'blue'`
Frequency:	1	4	1	1	1

Dictionaries

Notice how the frequency table resembles the picture of a list on page 298, except that the indices are replaced by words. In other words, the frequency table looks like a generalized list in which the indices are replaced by values that we choose. This kind of abstract data type is called a ***dictionary***. In a dictionary, each index is replaced with a unique ***key***. Unlike a list, in which the indices are implicit, a dictionary in Python (called a `dict` object) must define the correspondence between a key and its value explicitly with a `key:value` pair. To differentiate it from a list, a dictionary is enclosed in curly braces (`{ }`). For example, the frequency table above would be represented in Python like this:

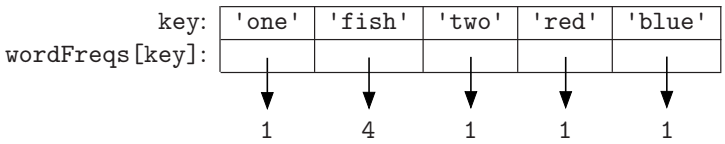
```
>>> wordFreqs = {'one': 1, 'fish': 4, 'two': 1, 'red': 1, 'blue': 1}
```

The first pair in `wordFreqs` has key `'one'` and value `1`, the second pair has key `'fish'` and value `4`, etc.

Each entry in a dictionary object can be referenced using the familiar indexing notation, but using a key in the square brackets instead of an index. For example:

```
>>> wordFreqs['blue']
1
>>> wordFreqs['fish']
4
```

The model of a dictionary in memory is similar to a list:



Each entry in a dictionary is a reference to a value in the same way that each entry in a list is a reference to a value. So, as with a list, we can change values in a dictionary. For example, we can increment the value associated with the key `'two'`:

```
>>> wordFreqs['two'] = wordFreqs['two'] + 1
>>> wordFreqs
{'one': 1, 'fish': 4, 'two': 2, 'red': 1, 'blue': 1}
```

To insert a new pair into the dictionary, we assign a value to the new key just like we would assign a value to an item in a list:

```
>>> wordFreqs['whale'] = 5
>>> wordFreqs
{'one': 1, 'fish': 4, 'two': 2, 'red': 1, 'blue': 1, 'whale': 5}
```

Now let's use a dictionary to implement the algorithm that we developed above to find the frequencies of words. To begin, we will create an empty dictionary named `wordFreqs` in which to record our tally marks:

```
wordFreqs = { }
```

Each entry in this dictionary will have its key equal to a unique word and its value equal to the word's frequency count. To tally the frequencies, we need to iterate over the words in `wordList`. As in our tallying algorithm, if there is already an entry in `wordFreqs` with a key equal to the word, we will increment the word's associated value; otherwise, we will create a new entry with `wordFreqs[word] = 1`. To differentiate between the two cases, we can use the `in` operator: `word in wordFreqs` evaluates to `True` if there is a key equal to `word` in the dictionary named `wordFreqs`.

The following function puts all of this together to create a word frequency dictionary for all of the words in a given text.

```
1  import textlib

2  def wordFrequencies(text):
3      """ (docstring omitted) """

4      wordList = textlib.wordTokens(text)  # get the list of words in text
5      wordFreqs = { }
6      for word in wordList:
7          if word in wordFreqs:    # if word is already a key in wordFreqs,
8              wordFreqs[word] = wordFreqs[word] + 1   # count the word
9          else:                    # otherwise,
10             wordFreqs[word] = 1  #   create a new entry word:1

11     return wordFreqs
```

Carefully study the following trace table for the function, which assumes that `wordList` and `wordFreqs` have already been assigned in lines 4–5.

Trace arguments: `wordList = ['one', 'fish', 'two', 'fish', 'red', 'fish', 'blue', 'fish']`

Step	Line	wordFreqs	word	Notes
3	6	{ }	'one'	word ← wordList[0]
4	7	″	″	'one' not a key; skip to line 10
5	10	{'one': 1}	″	insert 'one': 1
6	6	″	'fish'	word ← wordList[1]
7	7	″	″	'fish' not a key; skip to line 10
8	10	{'one': 1, 'fish': 1}	″	insert 'fish': 1
9	6	″	'two'	word ← wordList[2]
10	7	″	″	'two' not a key; skip to line 10
11	10	{'one': 1, 'fish': 1, 'two': 1}	″	insert 'two': 1
12	6	″	'fish'	word ← wordList[3]
13	7	″	″	'fish' is a key; do line 8
14	8	{'one': 1, 'fish': 2, 'two': 1}	″	increment wordFreqs['fish']
⋮				
24	6	{'one': 1, 'fish': 3, 'two': 1, 'red': 1, 'blue': 1}	'fish'	word ← wordList[7]
25	7	″	″	'fish' is a key; do line 8
26	8	{'one': 1, 'fish': 4, 'two': 1, 'red': 1, 'blue': 1}	″	increment wordFreqs['fish']
27	11	″	″	return wordFreqs

Return value: `{'one': 1, 'fish': 4, 'two': 1, 'red': 1, 'blue': 1}`

If we are to apply this function to a larger text, we will need a nice way to display the results. The following function prints an alphabetized table of words with their frequencies.

```
def printFrequencies(frequencies, number):
    """Print an alphabetized table of keys and frequencies.

    Parameters:
        frequencies: a dictionary containing key:count pairs
        number:      the number of entries to print

    Return value: None
    """
```

```
    keyList = list(frequencies.keys())   # get a list of the keys
    keyList.sort()                       # and sort them

    print('{0:<20} {1}'.format('Key', 'Frequency'))
    print('{0:<20} {1}'.format('---', '---------'))
    for key in keyList[:number]:         # iterate over the sorted list
        print('{0:<20} {1:>5}'.format(str(key), frequencies[key]))
```

We have designed the function to refer to generic keys, rather than words, to make it more general. To print the keys in alphabetical order, we first get a list of the keys in the dictionary using the `keys` method. Since `keys` returns an object from the specialized `dict_keys` class, we need to convert it to a list before we sort it. Then we iterate over the desired `number` of words in this sorted list instead of the `frequencies` dictionary.

We can find the frequencies of the first five words (alphabetically) in *Frankenstein* with the following program:

```
def main():
    textFile = open('frankenstein.txt', 'r')
    text = textFile.read()
    textFile.close()

    freqs = wordFrequencies(text)
    printFrequencies(freqs, 5)

main()
```

The result is the following table.

```
Key                  Frequency
---                  ---------
1                            4
10                           2
11                           2
11th                         2
12                           2
```

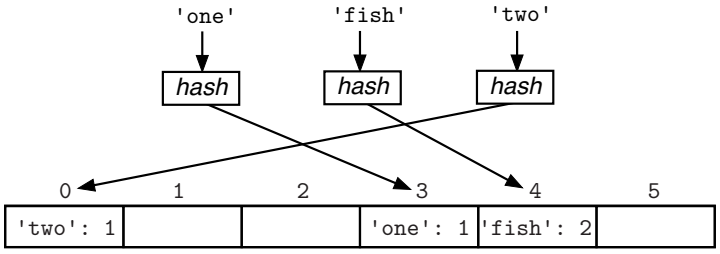
Reflection 7.17 *Why are the first "words" in the table numbers?*

If you remove the numbers from `keyList` before printing, using the `removeNumbers` function from Exercise 7.2.16, you will see this instead:

```
Key                  Frequency
---                  ---------
a                         1384
abandon                      2
abandoned                    3
abbey                        1
abhor                        5
```

Tangent 7.2: Hash tables

A Python dictionary can be implemented with a structure called a *hash table*. A hash table contains a fixed number of indexed *slots* in which the `key:value` pairs are stored. The slot assigned to a particular `key:value` pair is determined by a *hash function* that "translates" the key to a slot index. The picture below shows how some items in the `wordFreqs` dictionary might be placed in an underlying hash table.



In this illustration, the hash function associates the key `'one'` with slot `3`, `'fish'` with slot `4`, and `'two'` with slot `0`.

The underlying hash table allows us to access a value in a dictionary (e.g., `wordFreqs['fish']`) or test for inclusion (e.g., `key in wordFreqs`) in a constant amount of time because each operation only involves a hash computation and then a direct access (like indexing in a string or a list). In contrast, if the pairs were stored in a list, then the list would need to be searched (in linear time) to perform these operations.

Unfortunately, this constant-time access could be foiled if a key is mapped to an occupied slot, an event called a *collision*. Collisions can be resolved by using adjacent slots, using a second hash function, or associating a list of items with each slot. A good hash function tries to prevent collisions by assigning slots in a seemingly random manner, so that keys are evenly distributed in the table and similar keys are not mapped to the same slot. Because hash functions tend to be so good, we can still consider an average dictionary access to be a constant-time operation, or one elementary step, even with collisions.

Finding the most frequent word

To find the highest frequency word(s) in a text, we first need to find the maximum frequency. To do this, we first extract a list of the values from the frequency dictionary using the `values` method.

```
frequencyValues = list(frequencies.values())
```

Because the `values` method returns an object from another special class called `dict_values`, we convert the result to a list. Then we can find the maximum frequency in that list:

```
maxFrequency = max(frequencyValues)
```

To find the word(s) (there could be more than one) with the highest frequencies, we need to iterate over all the keys in the frequency dictionary, checking if the frequency of each is equal to `maxFrequency`. To iterate over the keys in a dictionary, we simply

iterate over the dictionary itself with a familiar `for` loop. The following function uses a loop like this to append every key with the maximum frequency to a list.

```
def mostFrequent(frequencies):
    """Find the key(s) with the highest frequency.

    Parameter:
        frequencies: a dictionary containing key:count pairs

    Return value: a list of the most frequent keys in frequencies
    """

    frequencyValues = list(frequencies.values())
    maxFrequency = max(frequencyValues)

    mostFrequentKeys = [ ]
    for key in frequencies:
        if frequencies[key] == maxFrequency:
            mostFrequentKeys.append(key)

    return mostFrequentKeys
```

Reflection 7.18 *Use this function to find the most frequent word in Frankenstein.*

Not surprisingly, the word in *Frankenstein* with the highest frequency is *the*. Because results like this are not very interesting, common words, called *stop words*, are often removed from word lists. (You may have already written a function to do this in Exercise 7.2.15.)

Let's go one step further and write a function to print a table of words, sorted by frequency, from highest to lowest. We did something similar in the `printFrequencies` function by sorting the keys and then iterating over this sorted list. But this function is a little trickier because, once we have a sorted list of dictionary values, there is no way to recover the associated keys. In a dictionary, we can get the value of a pair given the key, but not vice versa.

But here is a trick we can use. We will create a list of (value, key) pairs, and then sort this list. When the `sort` method is given a list of tuples, it sorts the list by the first item in the tuple, so this will give us a list sorted by frequency, and each frequency will be accompanied by its key. Then we can iterate over this sorted list. The following function demonstrates this technique.

```
def printMostFrequent(frequencies, number):
    """Print a table of the highest frequency keys.

    Parameters:
        frequencies: a dictionary containing key:count pairs
        number:      the number of entries to print

    Return value: None
    """
```

```
sortedValues = []            # create list of (value, key) tuples
for key in frequencies:
    sortedValues.append((frequencies[key], key))
sortedValues.sort(reverse = True)     # sort in descending order

print('{0:<20} {1}'.format('Key', 'Frequency'))
print('{0:<20} {1}'.format('---', '---------'))
for pair in sortedValues[:number]:    # iterate over the sorted list
    key = pair[1]
    print('{0:<20} {1:>5}'.format(str(key), frequencies[key]))
```

Reflection 7.19 *Use this function to find the five most frequent words in Frankenstein.*

The five most frequent words are again not terribly surprising:

```
Key                  Frequency
---                  ---------
the                   4187
and                   2970
i                     2842
of                    2639
to                    2092
```

But after removing stop words, they are more meaningful. (Producing this list is left as an exercise.)

```
Key                  Frequency
---                  ---------
man                     131
life                    114
father                  112
eyes                    104
time                     98
```

Bigram frequencies

In the field of natural language processing (NLP), word frequencies by themselves are known as a "bag of words" language model. More insight into a text can be had by computing the frequencies of pairs of adjacent words, called *bigrams*. For example, in the `drSeuss` text, there are seven bigrams:

```
[('one', 'fish'), ('fish', 'two'), ('two', 'fish'), ('fish', 'red'),
 ('red', 'fish'), ('fish', 'blue'), ('blue', 'fish')]
```

Bigram frequencies, and n-gram frequencies more generally, are one of the basic measures used in speech recognition algorithms; knowing which words are most likely to follow other words can dramatically reduce the number of possibilities the algorithm must consider. More generally, distributions of bigram frequencies serve as the basis of language models in many NLP classification algorithms. Bigram frequencies can also serve as a simple "digital signature" of an author's writing style.

Tangent 7.3: Sentiment analysis

Sentiment analysis is the problem of determining whether some text is expressing a positive or negative opinion (or possibly something in between). It is frequently applied to movie and product reviews, as well as social media posts, by those wishing to study how a product or topic is being viewed by some constituency.

Word and bigram probabilities are the building blocks of *machine learning* algorithms for this and other classification problems in the field of natural language processing (NLP). The probability that a word appears in a text is simply the word's frequency divided by the total number of words in the text.

A machine learning algorithm first goes through a training phase in which probabilistic models of the classes are built from large corpora that have been previously classified by human beings. The algorithm classifies a new text by computing the probability that the text is in each class, and outputting the class with the highest probability.

A naive Bayes classifier is the simplest machine learning algorithm used for sentiment analysis. It is "naive" because it assumes that word probabilities are independent of both their positions in the text and their relationships to other words in the text. The algorithm is trained by computing two sets of probabilities. First, it computes the probability that each word appears in a set of pre-classified documents with positive sentiment. Then it computes the probability that each word appears in a different set of pre-classified documents with negative sentiment. To classify an unknown text, the algorithm computes the probability that the text belongs to each class, defined to be the product of the prior probability of the class (usually 1/2) and the probabilities that every word in the new text appears in that class. The output is the class giving the highest probability.

Computing bigram frequencies is similar to computing word frequencies, with two main differences. First, as we iterate over the list of words, since we are interested in consecutive pairs of words, we need to also keep track of the previous word. (This is similar to keeping track of the previous character in the `splitIntoWords` function from Section 6.1.)

```
prevWord = wordList[0]
for index in range(1, len(wordList)):
    word = wordList[index]

    # count the bigram (prevWord, word)

    prevWord = word
```

Before the loop, we initialize `prevWord` to be the first word in the list of words, and then start the `for` loop at the second word in the list. In each iteration of the loop, the bigram we are counting is the pair `(prevWord, word)`. At the bottom of the loop, we update `prevWord` in preparation for the next iteration.

The second difference is that a bigram is a pair of strings. We could store bigrams as two-item lists, but dictionaries only allow immutable objects as keys. Instead,

we will use ***tuples***, which are like lists, except they are enclosed in parentheses and are immutable. In general, tuples are used in place of lists when the objects being represented have a fixed length, and individual components are not likely to change. For example, tuples are often used to represent the (red, green, blue) components of colors and the (x,y) coordinates of points. Tuples are also more memory efficient because extra memory is set aside in a list for a few future appends.

With the exception of these two differences, highlighted below, the following `bigramFrequencies` function is very similar to `wordFrequencies`.

```
import textlib

def bigramFrequencies(text):
    """Find the frequencies of all bigrams in a text.

    Parameter:
        text: a string object

    Return value: a dictionary containing the bigram frequencies
    """

    wordList = textlib.wordTokens(text)
    bigramFreqs = { }
    prevWord = wordList[0]
    for index in range(1, len(wordList)):
        word = wordList[index]
        bigram = (prevWord, word)
        if bigram not in bigramFreqs:
            bigramFreqs[bigram] = 1
        else:
            bigramFreqs[bigram] = bigramFreqs[bigram] + 1
        prevWord = word

    return bigramFreqs
```

Reflection 7.20 *Use this function and* `printMostFrequent` *to find the five most frequent bigrams in Frankenstein.*

The five most frequent bigrams in *Frankenstein* are:

```
Key                 Frequency
---                 ---------
('of', 'the')         526
('of', 'my')          272
('in', 'the')         262
('i', 'was')          226
('i', 'had')          219
```

The following exercises ask you to apply and modify these functions, and explore some additional uses for dictionaries. In Chapter 11, we will also use dictionaries to model social networks and other types of highly connected phenomena, and show

how computer algorithms applied to these networks can help diffuse pandemics, market new products, and make infrastructures more resilient.

Exercises

7.3.1. Write a function

```
wordFrequenciesFile(fileName)
```

that uses the `wordFrequencies` function to return a dictionary containing the frequencies of the words in the file with the given `fileName`.

7.3.2* Show how to find the five words in Jane Austen's *Sense and Sensibility* (available on the book website) with the highest frequencies. What are they? Similarly, what are the five bigrams with the highest frequencies?

7.3.3* Use the functions from Exercise 7.2.15 to modify the `wordFrequencies` function so that it does not include stop words. With your changes, what are the five most frequent words in *Sense and Sensibility*?

7.3.4. Write a new version of the `printMostFrequent` function that prints a desired number of most frequent words, but does not include stop words. Use the `getStopWords` function from Exercise 7.2.15.

7.3.5* Write a function

```
firstLetterCounts(wordList)
```

that takes as a parameter a list of strings named `words` and returns a dictionary with lowercase letters as keys and the number of words in `words` that begin with that letter (lower or uppercase) as values. For example, if the list is `['ant', 'bee', 'armadillo', 'dog', 'cat']`, then your function should return the dictionary `{'a': 2, 'b': 1, 'c': 1, 'd': 1}`.

7.3.6. Similar to the Exercise 7.3.5, write a function

```
firstLetterWords(wordList)
```

that takes as a parameter a list of strings named `words` and returns a dictionary with lowercase letters as keys. But now associate with each key the *list of the words* in `words` that begin with that letter. For example, if the list is `['ant', 'bee', 'armadillo', 'dog', 'cat']`, then your function should return the following dictionary:

```
{'a': ['ant', 'armadillo'], 'b': ['bee'], 'c': ['cat'],
 'd': ['dog']}
```

7.3.7. The *probability mass function* (PMF) of a data set gives the probability of each value in the set. A dictionary representing the PMF is a frequency dictionary with each frequency value divided by the total number of values in the original data set. For example, the (rounded) probabilities for the values in the dictionary `{18.9: 2, 19.0: 1, 19.2: 1, 19.3: 2}` are

```
{18.9: 0.286, 19.0: 0.143, 19.2: 0.143, 19.3: 0.286}
```

Write a function

```
pmf(frequency)
```

that returns a dictionary containing the PMF of the frequency dictionary passed as a parameter.

7.3.8. The probability that a word appears in a text can be found by dividing the frequency of the word by the total number of words in the text. These probabilities form the basis of many machine learning algorithms like those described in Tangent 7.3. Write a function

```
wordProbabilities(text)
```

that returns a dictionary containing the probabilities of all words in `text`. Your function should call the `wordFrequencies` function.

7.3.9. Write a function

```
histogram(data, numBins)
```

that displays a histogram of the values in the list `data` using `numBins` bins. The bins correspond to equal sized intervals between the minimum and maximum values in `data`. Each bin will count the number of values that fall in its corresponding interval. For example, if `data = [4, 7, 2, 8, 3]` and `numBins = 3`, then the bins will correspond to the intervals $[2, 4)$, $[4, 6)$, and $[6, 8]$, and the counts in these bins will be 2, 1, and 3. The histogram can be displayed using the `bar` function from `matplotlib.pyplot`:

```
pyplot.bar(range(1, numBins + 1), binCounts, align = 'center')
pyplot.xticks(range(1, numBins + 1))  # label the bins
```

Test your function with both small and large lists of randomly generated values in a variety of ranges. Also test it with a list of sums of random values such as

```
data = []
for count in range(10000):
    data.append(random.randrange(20, 80) + random.randrange(50))
```

What do you notice?

7.3.10* Write a function

```
bonus(salaries)
```

that takes as a parameter a dictionary named `salaries`, with names as keys and salaries as values, and increases the salary of everyone in the dictionary by 5%.

7.3.11. Write a function

```
updateAges(names, ages)
```

that takes as parameters a list of `names` of people whose birthday is today and a dictionary named `ages`, with names as keys and ages as values, and increments the age of each person in the dictionary whose birthday is today.

7.3.12. Write a function

```
seniorList(students, year)
```

that takes as a parameter a dictionary named `students`, with names as keys and class years as values, and returns a list of names of students who are graduating in `year`.

7.3.13* Dictionaries are also well-suited for handling translations. For example, the following dictionary associates a meaning with each of three texting abbreviations.

```
translations = {'lol': 'laugh out loud', 'u': 'you', 'r': 'are'}
```

With this, we can find the meaning of `lol` with

```
translations['lol']
```

Write a function

```
txtTranslate(txtWord)
```

that uses a dictionary to return the English meaning of the texting abbreviation `txtWord`. Incorporate translations for at least ten texting abbreviations. If the abbreviation is not in the dictionary, your function should return a suitable string message instead. For example, `txtTranslate('lol')` should return `'laugh out loud'`.

7.3.14. Write a function

```
createDictionary()
```

that creates a dictionary, inserts several English words as keys and the Pig Latin (or any other language) translations as values, and then returns the completed dictionary.

Next write a function

```
translate()
```

that calls your `createDictionary` function to create a dictionary, and then repeatedly asks for a word to translate. For each entered word, it should print the translation using the dictionary. If a word does not exist in the dictionary, the function should say so. The function should end when the word `quit` is typed.

7.3.15. Write a function

```
login(passwords)
```

that takes as a parameter a dictionary named `passwords`, with usernames as keys and passwords as values, and repeatedly prompts for a username and password until a valid pair is entered. Your function should continue to prompt even if an invalid username is entered.

7.3.16. Write a function

```
union(dict1, dict2)
```

that returns a new dictionary that contains all of the entries of the two dictionaries `dict1` and `dict2`. If the dictionaries share a key, use the value in the first dictionary. For example, `union({'pies': 3, 'cakes': 5}, {'cakes': 4, 'tarts': 2})` should return the dictionary `{'pies': 3, 'cakes': 5, 'tarts': 2}`.

7.3.17. The *Mohs hardness scale* rates the hardness of a rock or mineral on a 10-point scale, where 1 is very soft (like talc) and 10 is very hard (like diamond). Suppose we have a list such as

```
rocks = [('talc', 1), ('lead', 1.5), ('copper', 3),
         ('nickel', 4), ('silicon', 6.5), ('emerald', 7.5),
         ('boron', 9.5), ('diamond', 10)]
```

where the first element of each tuple is the name of a rock or mineral, and the second element is its hardness. Write a function

```
hardness(rocks)
```

that returns a dictionary organizing the rocks and minerals in such a list into four categories: soft (1–3), medium (3.1–5), hard (5.1–8), and very hard (8.1–10). For example, given the list above, the function would return the dictionary

```
{'soft': ['talc', 'lead', 'copper'],
 'medium': ['nickel'],
 'hard': ['silicon', 'emerald'],
 'very hard': ['boron', 'diamond']}
```

7.3.18. Write a function

```
getBigramProbabilities(bigramFreqs)
```

that returns a new dictionary containing, for each bigram in `bigramFreqs`, the probability that the second word in the bigram follows the first word in the bigram. This is defined as the bigram's frequency divided by the sum of the frequencies of all bigrams with the same first word.

7.3.19. Write a function

```
printNextWords(bigramFreqs, word)
```

that prints a table containing all of the words that come after `word` according to the dictionary `bigramFreqs`. Each word in the table should be accompanied by its frequency and the probability that it follows `word`, defined as its frequency divided by the sum of the frequencies of all bigrams with `word` as their first word. For example, in *Sense and Sensibility*, passing `'twisted'` in for `word` should print

```
Next Word       Frequency Probability
---------       --------- -----------
in                      1   0.250
tree                    1   0.250
blasted                 1   0.250
his                     1   0.250
```

7.3.20* Write a function

```
wordPredictor(bigramFreqs)
```

that mimics the predictive text function of phones using the given dictionary of bigram frequencies. The function should prompt for a first word, then suggest the top three highest frequency words that follow that word. The chosen word will serve as the next word in a loop. The loop should continue until either no next word is found or a word is entered that is not among the choices offered. For example, a possible session based on bigram frequencies from *Sense and Sensibility* might look like the following.

```
First word: she
Choose among ['was', 'had', 'could']: could
she could
Choose among ['not', 'be', 'have']: be
she could be
Choose among ['a', 'the', 'in']: the
she could be the
Choose among ['same', 'world', 'house']: world
she could be the world
Choose among ['to', 'and', 'of']: quit
```

7.3.21. Bigram frequencies can be used to create fun, computer-generated "poetry." Write a function

```
writer(bigramFreqs, firstWord, length)
```

that returns a string containing a sequence of words in which the next word is randomly chosen from among the most likely words to follow the previous word, based on the given bigram frequencies. When choosing the next word, if there are n bigrams starting with the previous word, the list of candidates should be created from the $n/2$ with the highest frequencies. The second and third parameters are the starting word and the desired number of words in the "poem." For example, one particular 15-word output, based on the bigram frequencies in *Sense and Sensibility*, is

```
she read and unaffected sincerity that what say it by him such
behaviour in many particulars
```

This exercise is very similar to the previous exercise, but with no human intervention.

7.3.22. (This exercise assumes that you have read Section 6.8.) Rewrite the `complement` function on page O6.8-5 using a dictionary. (Do not use any `if` statements.)

7.3.23. (This exercise assumes that you have read Section 6.8.) Suppose we have a set of homologous DNA sequences with the same length. A *profile* for these sequences contains the frequency of each base in each position. For example, suppose we have the following five sequences, lined up in a table:

```
   G  G  T  T  C
   G  A  T  T  A
   G  C  A  T  A
   C  A  A  T  C
   G  C  A  T  A
  ---------------
A: 0  2  3  0  3
C: 1  2  0  0  2
G: 4  1  0  0  0
T: 0  0  2  5  0
  ---------------
   G  A  A  T  A  <-- consensus sequence
```

Their profile is shown below the sequences. The first column of the profile indicates that there is one sequence with a `C` in its first position and four sequences with a `G` in their first position. The second column of the profile shows that there are two sequences with `A` in their second position, two sequences with `C` in their second position, and one sequence with `G` in its second position. And so on. The *consensus sequence* for a set of sequences has in each position the most common base in the profile. The consensus for this list of sequences is shown below the profile.

A profile can be implemented as a list of 4-element dictionaries, one for each column. A consensus sequence can then be constructed by finding the base with the maximum frequency in each position. In this exercise, you will build up a function to find a consensus sequence in four parts.

(a) Write a function

```
profile1(sequences, index)
```

that returns a dictionary containing the frequency of each base in position `index` in the list of DNA sequences named `sequences`. For example, if we pass in `['GGTTC', 'GATTA', 'GCATA', 'CAATC', 'GCATA']` for `sequences` (the sequences from the example above) and 2 for `index`, then the function should return the dictionary `{'A': 3, 'C': 0, 'G': 0, 'T': 2}`, equivalent to the third column in the profile above.

(b) Write a function

```
profile(sequences)
```

that returns a list of dictionaries representing the profile for the list of DNA sequences named `sequences`. For example, given the list of sequences above, the function should return the list

```
[{'A': 0, 'C': 1, 'G': 4, 'T': 0},
 {'A': 2, 'C': 2, 'G': 1, 'T': 0},
 {'A': 3, 'C': 0, 'G': 0, 'T': 2},
 {'A': 0, 'C': 0, 'G': 0, 'T': 5},
 {'A': 3, 'C': 2, 'G': 0, 'T': 0}]
```

Your `profile` function should call your `profile1` function in a loop.

(c) Write a function

```
maxBase(freqs)
```

that returns the base with the maximum frequency in a dictionary of base frequencies named `freqs`. For example,

```
maxBase({'A': 0, 'C': 1, 'G': 4, 'T': 0})
```

should return `'G'`.

(d) Write a function

```
consensus(sequences)
```

that returns the consensus sequence for the list of DNA sequences named `sequences`. Your `consensus` function should call your `profile` function and also call your `maxBase` function in a loop.

7.4 READING TABULAR DATA

The earthquake locations that we plotted back in Section 2.2 came from the U.S. Geological Survey (USGS), which maintains up-to-the-minute data about earthquakes happening around the world.[2] In this section, we will read this data directly from the USGS and plot it with `matplotlib.pyplot`, incorporating other characteristics such as the magnitudes and depths of the earthquakes, to visualize our planet's major tectonic plates and how they interact.

[2] `http://earthquake.usgs.gov/earthquakes/feed/v1.0/csv.php`

Earthquakes

USGS earthquake data is available in many formats, the simplest of which is a tabular format called CSV, short for "comma-separated values." The first few rows and columns of a USGS CSV file look like the following.

```
time,latitude,longitude,depth,mag,...
2020-04-19T16:34:38.090Z,19.361166,-155.0753326,6.08,1.84,...
2020-04-19T16:34:03.110Z,35.7088333,-117.5816667,10.66,1.02,...
2020-04-19T16:32:18.020Z,61.3918,-150.1225,30.5,1.4,...
2020-04-19T16:27:34.030Z,35.768,-117.603,3.41,1,...
2020-04-19T16:24:50.590Z,38.0505,-118.6257,7.2,1.1,...
2020-04-19T16:16:42.700Z,38.1872,-118.7235,0.1,1.4,...
```

CSV files contain one row of text per line, with columns separated by commas. The first row is a header row containing the names of the fifteen columns in the file, only the first five of which are shown here. Each additional row consists of data from one earthquake. If you were to view these first five columns in a spreadsheet program, it would like something like this:

time	latitude	longitude	depth	mag
2020-04-19T16:34:38.090Z	19.361166	-155.0753326	6.08	1.84
2020-04-19T16:34:03.110Z	35.7088333	-117.5816667	10.66	1.02
2020-04-19T16:32:18.020Z	61.3918	-150.1225	30.5	1.4
2020-04-19T16:27:34.030Z	35.768	-117.603	3.41	1
2020-04-19T16:24:50.590Z	38.0505	-118.6257	7.2	1.1
2020-04-19T16:16:42.700Z	38.1872	-118.7235	0.1	1.4

The first earthquake in this file was detected at 16:34 UTC (Coordinated Universal Time) on 2020-04-19 at 19.361166 degrees latitude and −155.0753326 degrees longitude. The earthquake occurred at a depth of 6.08 km and had magnitude 1.84.

A CSV file containing data about all of the earthquakes in the past 30 days is available on the web at the URL

```
http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_month.csv
```

(If you have trouble with this file, you can try smaller files by replacing `all_month` with `2.5_month` or `4.5_month`. The numbers indicate the minimum magnitude of the earthquakes included in the file.)

Reflection 7.21 *Enter the URL above into a web browser to see the data file for yourself. About how many earthquakes were recorded in the past 30 days?*

We can read the contents of this CSV file in Python using the same techniques you learned in Section 6.2. We can either download and save the file manually (and then read the file from our program), or we can download it directly in our program using the `urllib.request` module. We use the latter method in the following function.

```
1 import urllib.request as web
2 import matplotlib.pyplot as pyplot

3 def plotQuakes():
4     """Plot the locations of all earthquakes in the past 30 days.

5     Parameters: None

6     Return value: None
7     """

8     url = 'http://earthquake.usgs.gov/...' # see above for full URL
9     quakeFile = web.urlopen(url)
10    header = quakeFile.readline()          # get past the header row

11    longitudes = []
12    latitudes = []
13    for rawLine in quakeFile:
14        line = rawLine.decode('utf-8')    # interpret the line as text
15        row = line.split(',')             # split columns at commas
16        latitudes.append(float(row[1]))   # append latitude
17        longitudes.append(float(row[2]))  # append longitude
18    quakeFile.close()

19    pyplot.scatter(longitudes, latitudes, 10)  # 10 = area of each point
20    pyplot.show()
```

To begin our function, in lines 8–10, we open the URL, and read the header row containing the column names. We do not actually use the header row; we just need to get past it to get to the data.

To visualize fault boundaries with `matplotlib.pyplot`, we need all the longitude (x) values in one list and all the latitude (y) values in another list. These lists are initialized before the loop in lines 11–12. To maintain an association between the latitude and longitude of a particular earthquake, we need these lists to be ***parallel lists***, in the sense that the longitude and latitude at any particular index belong to the same earthquake.

The `for` loop in lines 13–17 iterates over the lines in the file. Remember that data from the web is read as a raw `bytes` object that needs to be converted to text before we can use it. To extract the necessary information from each line, we use the `split` method from the string class. Recall that `split` splits a string at every instance of a given character, and returns the list of strings that result. In this case, in line 15, we want to split `line` at every comma. The resulting list named `row` will have the time of the earthquake at index 0, the latitude at index 1, the longitude at index 2, etc. Note that each of these values is a string, so we need to convert the latitude and longitude to numbers using `float`. After converting each value, we append it to its respective list in lines 16–17.

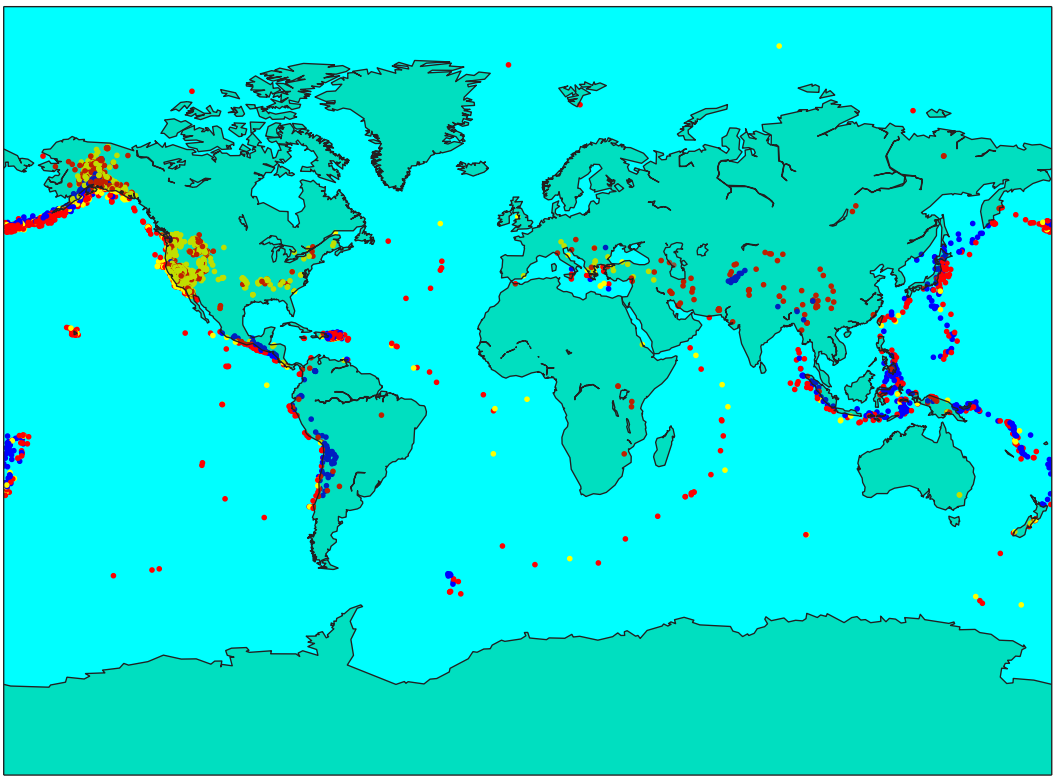


Figure 7.3 Plotted earthquake locations with colors representing depths (yellow are shallower, red are medium depth, and blue are deeper).

The plotted earthquakes are shown in Figure 7.3 over a map background. (Your plot will not show the map, but if you would like to add it, look into the `Basemap` class from the `mpl_toolkits.basemap` module.) The colors of the points in this plot reflect the earthquakes' depths. To incorporate these colors into our function, we will also need to extract the depths of the earthquakes in a third list.

Reflection 7.22 *What do we need to add to our function to also get a list of earthquake depths?*

Between lines 12 and 13, we will need to initialize a third parallel list named `depths` and then, after line 17, append the value in column 3 to this list with

```
depths.append(float(row[3]))
```

We can color each point according to its depth by passing the `scatter` function a list of colors, one for each point. Shallow (less than 10 km deep) earthquakes will be yellow, medium depth (between 10 and 50 km) earthquakes will be red, and deep earthquakes (greater than 50 km deep) will be blue. To create this list, we will iterate over the final `depths` list and, for each depth, append the appropriate color string to another list named `colors`:

```
colors = []
for depth in depths:
    if depth < 10:
        colors.append('yellow')
    elif depth < 50:
        colors.append('red')
    else:
        colors.append('blue')
```

To assign colors to points, we pass this `colors` list into `scatter` as a *keyword argument*:

```
pyplot.scatter(longitudes, latitudes, 10, color = colors)
```

We also saw keyword arguments briefly in Section 4.1. The name `color` is the name of a parameter of the `scatter` function for which we are passing the argument `colors`. (We will only use keyword arguments with `matplotlib.pyplot` functions, although we could also use them in our functions if we wished to do so.)

Reflection 7.23 *Make these modifications to the* `plotQuakes` *function, and look at the resulting figure. Try to identify the different tectonic plates. Can you infer anything about the way neighboring plates interact from the depth colors?*

Looking at Figure 7.3, the ring of red and yellow dots around Africa encloses the African plate, and the dense line of blue and red dots northeast of Australia delineates the boundary between the Pacific and Australian plates. The depth information gives geologists information about the types of the boundaries and the directions in which the plates are moving. For example, the shallow earthquakes on the west coast of North America mark a transform boundary in which the plates are sliding past from each other, while the deeper earthquakes in the Aleutian islands in Alaska mark a subduction zone at a convergent boundary where the Pacific plate to the south is diving underneath the North American plate to the north.

Exercises

7.4.1* Modify `plotQuakes` so that it also reads earthquake magnitudes into a list, and draws larger circles for higher magnitude earthquakes. The sizes of the points can be changed by passing a list of sizes, similar to the list of colors, as the third argument to the `scatter` function. The size of each point should be the square of the magnitude of the corresponding earthquake.

7.4.2. Modify the `firstLetterCounts` function from Exercise 7.3.5 so that it takes a file name as a parameter and uses the words from this file instead. Test your function using the SCRABBLE dictionary on the book website.[3]

7.4.3* Modify the `login` function from Exercise 7.3.15 so that it takes a file name as a parameter and creates a username/password dictionary with the usernames and passwords in that file before it starts prompting for a username and password. Assume that the file contains one username and password per line, separated by a space. There is an example file on the book website.

[3]SCRABBLE is a registered trademark of Hasbro Inc.

7.4.4. Write a function

```
plotPopulation()
```

that plots the world population over time from the tab-separated data file on the book website named `worldpopulation.txt`. To read a tab-separated file, split each line with `line.split('\t')`. These figures are U.S. Census Bureau midyear population estimates from 1950–2050. Your function should create two plots. The first shows the years on the x-axis and the populations on the y-axis. The second shows the years on the x-axis and the annual growth rate on the y-axis. The growth rate is the difference between this year's population and last year's population, divided by last year's population. Be sure to label your axes in both plots with the `xlabel` and `ylabel` functions.

What is the overall trend in world population growth? Do you have any hypotheses regarding the most significant spikes and dips?

7.4.5. Write a function

```
plotMeteorites()
```

that plots the location (longitude and latitude) of every known meteorite that has fallen to earth, using a tab-separated data file from the book website named `meteoritessize.txt`. Split each line of a tab-separated file with `line.split('\t')`. There are large areas where no meteorites have apparently fallen. Is this accurate? Why do you think no meteorites show up in these areas?

7.4.6* Write a function

```
plotTemps()
```

that reads a CSV data file from the book website named `madison_temp.csv` to plot several years of monthly minimum temperature readings from Madison, Wisconsin. The temperature readings in the file are integers in tenths of a degree Celsius and each date is expressed in a `YYYYMMDD` format. Rather than putting every date in a list for the x-axis, just make a list of the years that are represented in the file. Then plot the data and put a year label at each January tick with

```
pyplot.plot(range(len(minTemps)), minTemps)
pyplot.xticks(range(0, len(minTemps), 12), years)
```

The first argument to the `xticks` function says to only put a tick at every twelfth x value, and the second argument supplies the list of years to use to label those ticks. It will be helpful to know that the data file starts with a January 1 reading.

7.4.7. Write a function

```
plotZebras()
```

that plots the migration of seven Burchell's zebra in northern Botswana. (The very interesting story behind this data can be found at `https://www.movebank.org/node/11921`.) The function should read a CSV data file from the book website named `zebra.csv`. Each line in the data file is a record of the location of an individual zebra at a particular time. Each individual has a unique identifier. There are over 50,000 records in the file. For each record, your function should extract the individual identifier (column index 9), longitude (column index 3), and latitude (column index 4). Store the data in two dictionaries, each with

identifiers as keys. In one dictionary, the value associated with each identifier should be a list of the longitudes of all tracking events for the zebra with that identifier. Similarly, the second dictionary should contain the corresponding latitude values. Plot the locations of the seven zebras in seven different colors to visualize their migration patterns.

How can you determine from the data which direction the zebras are migrating?

7.4.8. On the book website, there is a tab-separated data file named `education.txt` that contains information about the maximum educational attainment of U.S. citizens, as of 2013. Each non-header row contains the number of people in a particular category (in thousands) that have attained each of fifteen different educational levels. Look at the file in a text editor (*not* a spreadsheet program) to view its contents. Write a function

```
plotEducation()
```

that reads this data and then plots separately (but in one figure) the educational attainment of all males, females, and both sexes together over 18 years of age. The x-axis should be the fifteen different educational attainment levels and the y axis should be the *percentage* of each group that has attained that level. Notice that you will only need to extract three lines of the data file, skipping over the rest. To label the ticks on the x-axis, use the following:

```
pyplot.xticks(range(15), titles[2:], rotation = 270)
pyplot.subplots_adjust(bottom = 0.45)
```

The first statement labels the x ticks with the educational attainment categories, rotated 270 degrees. The second statement reserves 45% of the vertical space for these x tick labels. Can you draw any conclusions from the plot about relative numbers of men and women who pursue various educational degrees?

7.4.9. The ocean temperature data that we used in Section 7.2 is based on data acquired from *Coriolis*[4], a French organization that monitors the characteristics of the oceans from satellites, ships, and floating monitoring stations. These data are used to better understand and predict climate change. A CSV file containing data from one floating station in the eastern Mediterranean Sea can be downloaded on the book website as `float.csv`. Each row in the file contains one measurement of temperature and practical salinity of the seawater at a particular depth. The depth is recorded in terms of decibars of water pressure. This file contains the records of 45 profiles, where each profile consists of 39–164 measurements at varying depths. Each profile is designated by a date and time. Look at the data file carefully to understand its format.

(a) Write a function

```
getData()
```

that reads the data from the file into a dictionary in which each key is a date (a string) representing a profile. The value corresponding to each date will be a list of (pressure, salinity, temperature) tuples representing the measurements made in that profile. To simplify matters, use the first ten characters of the values in the `DATE` column as your keys. Your function should return the data dictionary for use by the functions that follow.

[4] `http://www.coriolis.eu.org`

(b) Write a function

```
plotDateTemp(dataDict)
```

that plots the surface seawater temperature recorded for each date in the dictionary named `dataDict`. You may assume that the maximum recorded temperature represents the surface temperature. Label your axes appropriately. Since there are too many dates to display on the x-axis neatly, you can display every tenth date with `pyplot.xticks(range(0, len(dates), 10), dates[::10])`.

(c) Write a function

```
plotTempPressure(dataDict)
```

that plots temperature (x-axis) vs. water pressure (y-axis) readings for November 17, 2019 (`'2019/11/17'`). Sort the list of triples for that day (by pressure) to ensure a clean plot. You can reverse the y axis so that the pressure reading decreases to mimic ocean depth with `pyplot.ylim(maxPressure, 0)`, where `maxPressure` is the maximum pressure reading on that day.

(d) Write a function

```
plotSalinityTemp(dataDict)
```

that plots salinity (x-axis) vs. temperature (y-axis) readings from all 45 profiles on a single plot. To differentiate the different curves, you can use a colormap that gradually changes color as the days progress. Doing this is much more arcane than usual, so here is an outline of the function that includes the colormap implementation:

```
def plotSalinityTemp(dataDict):
    numberDays = len(dataDict)    # set up the color map
    colorMap = cm.get_cmap('plasma', numberDays)
    normalizer = pyplot.Normalize(0, numberDays)
    pyplot.colorbar(cm.ScalarMappable(norm = normalizer,
                                      cmap = colorMap),
                                      label = 'Profile Number')
    count = 0
    for date in dataDict:
        readings = dataDict[date]
        readings.sort()

        # get the lists of salinity and temperature
        # measurements for this date

        pyplot.plot(salinities, temperatures,
                    linewidth = 0.25,
                    color = colorMap(count / numberDays))
        count = count + 1

    pyplot.xlabel('Practical Salinity')
    pyplot.ylabel('Temperature (degrees Celsius)')
    pyplot.show()
```

*7.5 DESIGNING EFFICIENT ALGORITHMS

This section is available on the book website.

*7.6 LINEAR REGRESSION

This section is available on the book website.

*7.7 DATA CLUSTERING

This section is available on the book website.

7.8 SUMMARY AND FURTHER DISCOVERY

It is often said that those who know how to manipulate and extract meaning from data will be the decision makers of the future.

In this chapter, we developed algorithms to summarize the contents of a list with various descriptive statistics, modify the contents of lists, and use dictionaries to describe the frequency of values in a list. The beauty of these techniques is that they can be used with a wide variety of data types and applications. But before any of them can be used on real data, the data must be read from its source and wrangled into a usable form. To this end, we also discussed basic methods for reading and formatting tabular data both from local files and from the web.

In later sections, we went beyond simply describing data sets to *data mining* techniques that can make predictions from them. *Linear regression* seeks a linear pattern in data and then uses this pattern to predict missing data points. The *k-means clustering* algorithm partitions data into clusters of like items to elicit hidden relationships.

Algorithms that manipulate lists can quickly become much more complicated than what we have seen previously, and therefore paying attention to their time complexity is important. To illustrate, we worked through a sequence of increasingly more elegant and more efficient algorithms for removing duplicates from a list. In the end, we saw that the additional time taken to think through a problem carefully and reduce its time complexity can pay dividends.

Notes for further discovery

Sherlock Holmes was an early (fictional) data scientist, always insisting on fact-based theories (and not vice versa). The chapter epigraph is from Sir Arthur Conan Doyle's short story, *The Adventure of the Copper Beeches* [14].

A good resource for current data-related news is the "data journalism" website *FiveThirtyEight* at `http://fivethirtyeight.com` . If you are interested in learning more about the emerging field of data science, one resource is *Doing Data Science* by Rachel Schutt and Cathy O'Neil [58].

Tangent 7.4: Privacy in the age of big data

Companies collect (and buy) a lot of data about their customers, including demographics (age, address, marital status), education, financial and credit history, buying habits, and web browsing behavior. They then mine this data, using techniques like clustering, to learn more about customers so they can target them with advertising that is more likely to lead to sales. But when does this practice lead to unacceptable breaches of privacy? For example, a recent article explained how a major retailer is able to figure out when a woman is pregnant before her family does.

When companies store this data online, it also becomes vulnerable to unauthorized access by hackers. In recent years, there have been several high-profile incidents of retail, government, and financial data breaches. As our medical records also begin to migrate online, more people are taking notice of the risks involved in storing "big data."

So as you continue to work with data, remember to always balance the reward with the inherent risk. Just because we *can* do something doesn't mean that we *should* do it.

There really are drifting buoys (called profiling floats) in the world's oceans that are constantly taking temperature readings to monitor climate change. For example, see

`http://www.coriolis.eu.org` .

The article referenced in Tangent 7.4 is from *The New York Times* [15]. The non-profit Electronic Frontier Foundation (EFF), founded in 1990, works at the forefront of issues of digital privacy and free speech. To learn more about contemporary privacy issues, visit its website at `http://www.eff.org`. For more about ethical issues in computing in general, we recommend *Computer Ethics* by Deborah Johnson and Keith Miller [27].

*7.9 PROJECTS

This section is available on the book website.