CHAPTER 6

Text, Documents, and DNA

> So, here's what I can say: the Library of Congress has more than 3 petabytes of digital collections. What else I can say with all certainty is that by the time you read this, all the numbers—counts and amount of storage—will have changed.
>
> Leslie Johnston, former Chief of Repository Development, Library of Congress
> *Blog post (2012)*

> The roughly 2000 sequencing instruments in labs and hospitals around the world can collectively sequence 15 quadrillion nucleotides per year, which equals about 15 petabytes of compressed genetic data. A petabyte is 2^{50} bytes, or in round numbers, 1000 terabytes. To put this into perspective, if you were to write this data onto standard DVDs, the resulting stack would be more than 2 miles tall. And with sequencing capacity increasing at a rate of around three- to fivefold per year, next year the stack would be around 6 to 10 miles tall. At this rate, within the next five years the stack of DVDs could reach higher than the orbit of the International Space Station.
>
> Michael C. Schatz and Ben Langmead
> *The DNA Data Deluge (2013)*

DATA has become a disruptive force not only in business but also in a broad swath of academic inquiry. Some literary scholars have embraced a new research mode known as "distant reading" in which they seek new insights through computational analyses of entire corpora from growing digital libraries. Similarly, many historians and political scientists are now conducting research in vast digital archives maintained by governments, universities, and nonprofits. In the biological and medical sciences, major advances are being driven by computational analyses of genomic data. The list could go on and on. Despite the variety of application areas however, much of this data shares a common underlying format. In this chapter, we will look at how this textual data are represented in a computer, how to access them from both files

and the web, and how to algorithmically process and analyze them to extract useful information.

6.1 FIRST STEPS

In this section, we will finish the reading level problem that we started in Chapter 1 and, in the process, introduce some first steps in text analyses. You may recall that, in Figure 1.4, we decomposed the reading level problem into three main subproblems, and then decomposed those subproblems further until we arrived at four unique leaves: computing the Flesch-Kincaid formula, computing the number of syllables in a word, and counting the number of words and sentences in a text. We were able to write a function pretty easily for the first of these subproblems and we wrote an algorithm in pseudocode for the second. We will focus in this section on the last two subproblems.

Counting words and sentences are special cases of a problem called ***tokenization***. A ***token*** is defined to be the basic unit of interest in a text, and tokenization is the problem of producing a list of all of the tokens in the text. Usually tokens are words or sentences, but they could also be numbers in a data file or individual characters in a DNA sequence; it depends on the context. Tokenization is also the first step in interpreting or compiling a program. In Python, tokens are names, keywords, literals, operators, delimiters, the newline character, and indentation characters.

Defining what words and sentences are is thornier than it seems. Normally, we can identify words in an English language text because they are separated by spaces or punctuation. But there are always exceptions. For example, what are the rules delimiting words and sentences in the following?

> "It's-a me—Mario!"
>
> It's 8:43 a.m. and I am typing from 140.141.132.1.

To keep things manageable, we will get words by simply splitting the text at runs of one or more whitespace characters (spaces, tabs, and newlines). When we tokenize sentences, we will split at runs of end punctuation marks (`.`, `?`, `!`).

Before text is analyzed, it is often simplified by removing superficial differences between words that should be considered equivalent (e.g., *The* and *the*, *10a.m.* and *10AM*), a process called ***normalization***. Before we perform word tokenization, we will normalize the text by making it all lowercase and removing punctuation. (When we tokenize sentences, we will not want to remove end punctuation marks.) Normalization can also involve spelling correction, removing plurals and other suffixes (called *stemming*), standardizing verb tense (called *lemmatization*), and removing common words (called *stop words*).

The texts that we analyze will be stored as strings. You'll recall that a string is a sequence of characters, and a string constant (also called a *string literal*) is enclosed

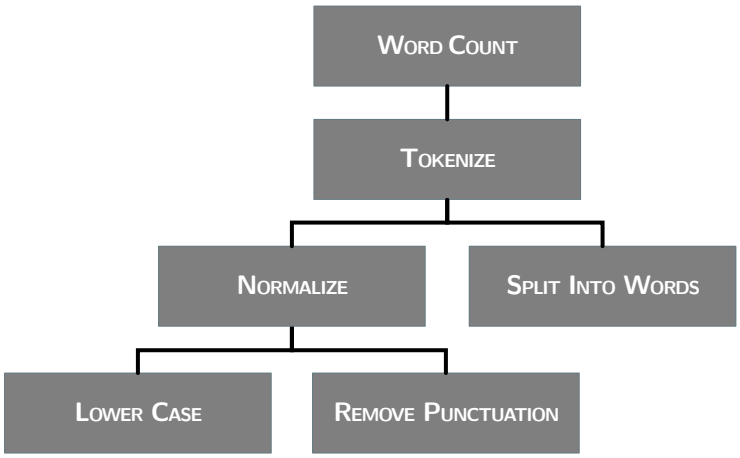


Figure 6.1 Functional decomposition tree for the Word Count problem.

in either single quotes (') or double quotes ("). For example, consider the following string, with spaces (␣) shown explicitly:

```
>>> shortText = "This␣isn't␣long.␣␣But␣it'll␣do.␣\nJust␣a␣few␣sentences..."
```

Reflection 6.1 *Why must this string be enclosed in double quotes rather than single quotes?*

According to our rules, the lists of word and sentence tokens in this text, after normalization, should be:

```
['this', 'isnt', 'long', 'but', 'itll', 'do', 'just', 'a', 'few', 'sentences']
```

and

```
["This isn't long.", "But it'll do.", 'Just a few sentences.']
```

Recall that lists are delimited by square brackets ([ ]) and items are separated by commas. So these lists contain ten and three string items, respectively. Once we have lists of tokens like these, the lengths of the lists will give us the outputs for the word count and sentence count algorithms.

This discussion suggests the functional decomposition of the Word Count problem shown in Figure 6.1. As usual, we will start at the bottom of the decomposition tree and work our way up.

Normalization

Strings, like turtles, are objects. So the string class, called `str`, is another example of an abstract data type. Recall that an abstract data type hides the details of how its data is stored, allowing a programmer to interact with it at a higher level through methods. (As we will discuss in Section 6.3, strings are actually stored as sequences of bytes.) One of many methods available for the `str` class[1] will solve the

[1] For a list, see Appendix A.6.

Tangent 6.1: Natural language processing

Researchers in the field of ***natural language processing*** seek to not only search and organize text, but to develop algorithms that can "understand" and respond to it, in both written and spoken forms. For example, Google Translate (`http://translate.google.com`) performs automatic translation from one language to another in real time. The "virtual assistants" that are becoming more prevalent on commercial websites seek to understand your questions and provide useful answers. Cutting edge systems seek to derive an understanding of immense amounts of unstructured data available on the web and elsewhere to answer open-ended questions. If these problems interest you, you might want to look at `http://www.nltk.org` to learn about the ***Natural Language Toolkit (NLTK)***, a Python module that provides tools for natural language processing. An associated book is available at `http://nltk.org/book` .

subproblem in the leftmost leaf in our decomposition tree. As we did with `Turtle` methods, we preface the name of the method with the name of the object to which we want the method to apply:

```
>>> shortText.lower()
"this isn't long.  but it'll do. \njust a few sentences..."
```

The `lower` method returns a new string in which all characters in a string are made lowercase.

To remove punctuation characters, we could use the `replace` method. The following example removes all periods from `shortText`.

```
>>> shortText.replace('.', '')
"This isn't long  But it'll do \nJust a few sentences"
```

The `replace` method returns a new string in which all instances of its first argument are replaced with its second argument. In this case, we passed in the ***empty string*** `''`, consisting of zero characters, for the second argument, which in effect deletes all instances of the first argument.

Notice that neither of these methods changed the value of `shortText`. Indeed, none of the string methods do because strings are ***immutable***, meaning that they cannot be changed. Instead, string methods always create a new string with the desired changes, leaving the original untouched.

To remove multiple punctuation characters from a text, we could call the `replace` method repeatedly, each time overwriting the previous string:

```
>>> newText = shortText
>>> newText = newText.replace('.', '')
>>> newText = newText.replace("'", '')
>>> newText
'This isnt long  But itll do \nJust a few sentences'
```

In a function to remove all punctuation, we would need to repeatedly call the `replace` method for *every* punctuation character, which is both tedious and inefficient.

Reflection 6.2 *Why is this inefficient? Think about how the* `replace` *method must work and how many times each character in the text must be examined.*

The `replace` method must examine each character in the string, compare it to the first argument, and then replace it with the second argument. So each time we call `replace` we are performing another pass across the characters in the string. Instead, we would like to make only one pass through the string and remove every punctuation character during that one pass.

To do this, we iterate over the characters in a string with a `for` loop, just like we iterated over integers in a `range`. For example, the following `for` loop iterates over the characters in the string `shortText` and prints each one.

```
>>> for character in shortText:
        print(character)
```

In each iteration of this loop, the next character in the string is assigned to the index variable `character`. If we wanted to omit characters from being printed, we would put the call to `print` inside an `if` statement:

```
>>> for character in shortText:
        if character != '.' and character != "'":
            print(character)
```

To adapt this technique to remove punctuation, we need to create a new, modified string in the body of the loop. In general, to create a modified string (remember that we cannot modify the original string), we need to iterate over each character of the original string and, in each iteration, create a new string that is the concatenation of the growing string from the previous iteration and the current character or something else based on the current character. The following function implements the simplest example of this idea, in which every character is concatenated to the end of the new string, creating an exact duplicate of the original.

```
def copy(text):
    """Return a copy of text.

    Parameter:
        text: a string object

    Return value: a copy of text
    """
1   newText = ''
2   for character in text:
3       newText = newText + character
4   return newText
```

This technique is really just another version of an accumulator, called a ***string accumulator***, conceptually similar to the list accumulators that we have been using

for plotting. The trace table below illustrates how this works when `text` is `'abcd'`. Changes in values are highlighted in red.

Trace arguments: `text = 'abcd'`

Step	Line	newText	character	Notes
1	1	`''`	—	newText is initialized to the empty string
2	2	`''`	`'a'`	character ← `'a'`
3	3	`'a'`	`'a'`	newText ← `'' + 'a'`
4	2	`'a'`	`'b'`	character ← `'b'`
5	3	`'ab'`	`'b'`	newText ← `'a' + 'b'`
6	2	`'ab'`	`'c'`	character ← `'c'`
7	3	`'abc'`	`'c'`	newText ← `'ab' + 'c'`
8	2	`'abc'`	`'d'`	character ← `'d'`
9	3	`'abcd'`	`'d'`	newText ← `'abc' + 'd'`

Return value: `'abcd'`

In the first iteration, the first character in `text`, which is `'a'`, is assigned to `character`. Then `newText` is assigned the concatenation of the current value of `newText` and `character`, which is `'' + 'a'`, or `'a'`. In the second iteration, `character` is assigned `'b'`, and `newText` is assigned the concatenation of `newText` and `character`, which is `'a' + 'b'`, or `'ab'`. This continues for two more iterations, resulting in a value of `newText` that is identical to the original `text`.

To apply this technique to remove punctuation from a string, we simply prevent the concatenation from taking place if `character` is a punctuation mark:

```
    for character in text:
        if character != '.' and character != "'":   # and ... etc.
            newText = newText + character
```

Reflection 6.3 *What happens if we replace the* `and` *operator with* `or`*?*

Adding in another test for every remaining punctuation character would be tedious at best, but we can simplify `if` conditions like this using the `in` operator, which evaluates to `True` if one string is contained inside another string. There is also a `not in` operator that has the opposite effect. For example:

```
>>> 'b' in 'abcd'
True
>>> 'bg' in 'abcd'
False
>>> 'b' not in 'abcd'
False
```

To make this even more convenient, there are string literals in the `string` module that contain all of the punctuation and whitespace characters:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.whitespace
' \t\n\r\x0b\x0c'
```

Notice that two of the characters in `string.punctuation`—`\'` and `\\`—are preceded by a backslash. The backslash (\) character, called the ***escape character***, causes the following character to be interpreted literally by the interpreter rather than as a meaningful character in the Python language. Escaping the single quote character allows it to be contained inside a string delimited by single quotes. The backslash character is also escaped because of its special meaning as the escape character! In `string.whitespace`, the first three characters are space, tab, and newline; the others are less common forms of whitespace that will likely not concern us.

Combining these two simplifications, we have the following function:

```
import string

def removePunctuation(text):
    """Remove punctuation from a text.

    Parameter:
        text: a string object

    Return value: a copy of text with punctuation removed
    """

    newText = ''
    for character in text:
        if character not in string.punctuation:
            newText = newText + character
    return newText
```

Now we can use this function and the `lower` method to write `normalize`:

```
def normalize(text):
    """Normalize a text by making it lowercase and removing punctuation.

    Parameter:
        text: a string object

    Return value: a normalized copy of text
    """

    newText = text.lower()
    newText = removePunctuation(newText)
    return newText
```

```
>>> normalize(shortText)
'this isnt long  but itll do \njust a few sentences'
```

Tokenization

The next step, as we work our way up the decomposition tree in Figure 6.1, is to write an algorithm to split a string into words at runs of whitespace characters. There is actually a string method named `split` that can do this for us. When `split` is given a string argument, it returns a list of strings that are separated by that argument. But with no arguments, `split` returns a list of strings that are separated by runs of whitespace:

```
>>> drSeuss = 'one fish two fish red fish blue fish'
>>> drSeuss.split('fish')
['one ', ' two ', ' red ', ' blue ', '']
>>> drSeuss.split()
['one', 'fish', 'two', 'fish', 'red', 'fish', 'blue', 'fish']
```

Although we could use this existing method, we are going to implement the function from scratch instead. There are two reasons for this. First, the general technique will be useful in similar situations that the `split` method cannot handle (e.g., splitting sentences at runs of end punctuation). Second, it will be another good example of how to use string accumulators, and of how to use string and list accumulators together.

The idea in the algorithm is to use a string accumulator to build up a string containing a word, as long as the character we are looking at is not whitespace. When we encounter whitespace, marking the end of the word, we want to append the word to a list of words and then reset the word to be an empty string to capture the next word. In pseudocode, a first draft of this algorithm can be expressed as follows.

Algorithm Split into words – Draft

Input: *text*

```
1  word list ← an empty list
2  word ← an empty string
3  repeat for each character in text:
4      if character is not whitespace:
5          word ← word + character
6      else:
7          append word to the end of word list
8          word ← an empty string
```

Output: *word list*

Notice how, in each iteration of the loop, we are either adding a character to the word or adding a word to the list. The equivalent Python function is very similar:

```
0 def splitIntoWords_Draft(text):
1     wordList = []
2     word = ''
3     for character in text:
4         if character not in string.whitespace:
5             word = word + character
6         else:
7             wordList.append(word)
8             word = ''
9     return wordList
```

Let's test our function by tracing its execution on the simple string `'i am'`.

Trace arguments: `text = 'i am'`

Step	Line	wordList	word	character	Notes
1	1	[ ]	—	—	wordList ← an empty list
2	2	[ ]	''	—	word ← an empty string
3	3	[ ]	''	'i'	character ← 'i'
4	4	[ ]	''	'i'	condition is true; execute line 5
5	5	[ ]	'i'	'i'	word ← '' + 'i'
6	3	[ ]	'i'	' '	character ← ' '
7	4	[ ]	'i'	' '	condition is false; execute line 7
8	7	['i']	'i'	' '	append 'i' to wordList
9	8	['i']	''	' '	word ← an empty string
10	3	['i']	''	'a'	character ← 'a'
11	4	['i']	''	'a'	condition is true; execute line 5
12	5	['i']	'a'	'a'	word ← '' + 'a'
13	3	['i']	'a'	'm'	character ← 'm'
14	4	['i']	'a'	'm'	condition is true; execute line 5
15	5	['i']	'am'	'm'	word ← 'a' + 'm'

Return value: `['i']`

Reflection 6.4 *Why wasn't the last word appended to the list? How do we fix the algorithm so that it is?*

If there had happened to be another space at the end of `text`, this would have prompted the algorithm to append `'am'`. But there wasn't, so it didn't. To fix this, we need to check after the loop if there is a final word remaining to be appended and, if so, append it:

```
if word != '':
    wordList.append(word)
```

There is also a more subtle issue with our algorithm. If there happen to be consecutive whitespace characters in `text`, then lines 7–8 will be executed in consecutive iterations, causing empty strings to be appended to `wordList`. For example, calling the function with

```
splitIntoWords_Draft('i  am   it ')
```

will return the list `['i', '', 'am', '', '', 'it']`. To prevent this, we only want to execute the `else` clause for the first whitespace character in a run of whitespace.

Reflection 6.5 *If the value of* `character` *is whitespace, how can we tell if it is the* first *in a sequence of whitespace characters? (Hint: if it is the first whitespace, what must the previous character* not *be?)*

If the value of `character` is a whitespace character, we know it is the first in a sequence if the previous character was *not* whitespace. So we need to replace the `else` with an `elif` statement that allows lines 7–8 to execute only if the previous character was not whitespace.

These two fixes are reflected in our updated algorithm below.

Algorithm SPLIT INTO WORDS

Input: *text*

```
1   word list ← an empty list
2   word ← an empty string
3   repeat for each character in text:
4       if character is not whitespace:
5           word ← word + character
6       else if the previous character was not also whitespace:
7           append word to the end of word list
8           word ← an empty string
9   if word is not an empty string:
10      append word to the end of word list
```

Output: *word list*

In our Python function, there isn't a way to refer to the "previous character" without explicitly keeping track of it. So we need to save the current value of `character` in a new variable `prevCharacter` at the end of each iteration so it is available when `character` is updated in the next iteration. The final function, with new parts highlighted, looks like this:

```
def splitIntoWords(text):
    """Split a text into words.

    Parameter:
        text: a string object

    Return value: the list of words in the text
    """
```

```
    wordList = []
    prevCharacter = ' '
    word = ''
    for character in text:
        if character not in string.whitespace:
            word = word + character
        elif prevCharacter not in string.whitespace:
            wordList.append(word)
            word = ''
        prevCharacter = character

    if word != '':
        wordList.append(word)

    return wordList
```

Reflection 6.6 *What happens if we do not initialize* `prevCharacter` *before the loop? Why did we initialize* `prevCharacter` *to a space? Does its initial value matter?*

To answer this question, let's consider two possibilities for the value assigned to `character` in the first iteration of the loop. First, suppose `character` is not a whitespace character. Then the `if` condition will be true and the `elif` condition will not be tested, so the initial value of `prevCharacter` does not matter. On the other hand, if the first value assigned to `character` *is* a whitespace character, then the `if` condition will be false and the `elif` condition will be checked. But we want to make sure that the `elif` condition is false so that an empty string is not inappropriately appended to the list of words. Setting `prevCharacter` to a space initially will prevent this from happening.

Now that we have both the `normalize` and `splitIntoWords` functions, we can easily write a tokenization function:

```
def wordTokens(text):
    """Break a text into words with punctuation removed.

    Parameter:
        text: a string object

    Return value: a list of word tokens
    """

    newText = normalize(text)
    tokens = splitIntoWords(newText)

    return tokens
```

And now the `wordCount` function is even easier. The only new thing we need is the `len` function, which returns the length of its argument. When applied to lists, it returns the number of items in the list. When applied to strings, it returns the number of characters in the string:

```
>>> len('i am it')
7
>>> len(['i', 'am', 'it'])
3
```

So the `wordCount` function simply gets a list of words from `wordTokens` and then returns the length of that list.

```
def wordCount(text):
    """Count the number of words in a string.

    Parameter:
        text: a string object

    Return value: the number of words in text
    """

    words = wordTokens(text)
    return len(words)
```

Creating your own module

The five functions that we wrote in this section will be very useful in the future, so let's package them into our own module that we can `import`. A module is just a Python program that is ready to be imported. First, if you haven't already, create a new file that contains the five functions that we wrote in this section and name it `textlib.py`.[2] It should look like this (with the function bodies present, of course):

```
import string

def removePunctuation(text):
    # body omitted

def normalize(text):
    # body omitted

def splitIntoWords(text):
    # body omitted

def wordTokens(text):
    # body omitted

def wordCount(text):
    # body omitted

def main():
    # body omitted

if __name__ == '__main__':
    main()
```

[2] `lib` is short for "library." This is a common naming convention for modules (e.g., `matplotlib`).

When a module is imported, all of the code in the module is executed, so we generally only want a module to contain function definitions, and perhaps some assignments of values to constants, but no function calls. To make a module dual-purpose—to be able to be executed on its own and be imported—we need to be able to differentiate between the two situations and only call `main` if the module is not being imported. This is accomplished by checking the value of `__name__` in the `if` statement before calling `main`. As we saw back in Section 2.6, `__name__` is assigned the value `'__main__'` if the module is executed directly by the Python interpreter. (If our module `textlib.py` is imported instead, then `__name__` will be assigned the value `'textlib'`.) So now if we run our module directly in IDLE, `main` will be executed, but if we import it instead, it won't.

Testing your module

Finally, let's test the module more completely using the techniques from the previous chapter. We will implement our tests in a new file named `test_textlib.py` with the following structure:

```
from textlib import *

def test_removePunctuation():
    # tests of removePunctuation here

    print('Passed all tests of removePunctuation!')

def test_normalize():
    # tests of normalize here

    print('Passed all tests of normalize!')

def test_splitIntoWords():
    # tests of splitIntoWords here

    print('Passed all tests of splitIntoWords!')

def test_wordTokens():
    # tests of wordTokens here

    print('Passed all tests of wordTokens!')

def test_wordCount():
    # tests of wordCount here

    print('Passed all tests of wordCount!')

def test():
    test_removePunctuation()
    test_normalize()
    test_splitIntoWords()
    test_wordTokens()
    test_wordCount()

test()
```

Save this program in the same folder as `textlib.py` so that the `import` statement can find the module.

The first line of the test program imports all of the functions from `textlib.py` into the global namespace of the test program. Recall from Section 2.6 that a normal `import` statement creates a new namespace containing all of the functions from an imported module. Instead, this form of the `import` statement imports functions into the *current* namespace. The advantage is that we do not have to preface every function call with the name of the module. If we wanted to only import some functions, we could replace the `*` with a list of those to import.

Notice that the test program calls `test()` instead of individual unit test functions. Besides being convenient, this technique has the advantage that, when we test new functions, we also re-test previously tested functions. If we make changes to any one function in a program, we want to *both* make sure that this change worked *and* make sure that we have not inadvertently broken something that was working earlier. This idea is called ***regression testing*** because we are making sure that our program has not *regressed* to an earlier error state.

Exercise 6.1.19 below asks you to complete these unit tests. Then exercises 6.1.20–6.1.26 challenge you to write functions for the remaining subproblems in Figure 1.4 and add them to your `textlib.py` module. With this complete module, you will be able to compute the Flesch-Kincaid reading level of any text!

In the next section, we will see how to read an entire text file or web page into a string so that you can use your module to compute the reading levels of actual books. In the next chapter, we will use your module to, among other things, analyze the relative frequencies of all the words and word bigrams in a text.

Exercises

Write a function for each of the following problems. Test each function with both common and boundary case arguments, and document your test cases.

6.1.1* The string method `count` returns the number of occurrences of a string in another string. For example, `shortText.count('is')` would return 2. Write a function

```
vowels(word)
```

that uses the `count` method to return the number of vowels in the string `word`. (Note that `word` may contain upper and lowercase letters.)

6.1.2. Write a function

```
whitespace(text)
```

that uses the `count` method to return the number of whitespace characters (spaces, tabs, and newlines) in the string `text`.

6.1.3* Write a function

```
underscore(sentence)
```

that uses the `replace` string method to return a version of the string `sentence` in which all the spaces have been replaced by the underscore (`_`) character.

6.1.4. Write a function

```
nospaces(sentence)
```

that uses the `replace` string method to return a version of the string `sentence` in which all the spaces have been removed.

6.1.5. Write a function

```
txtHelp(txt)
```

that returns an expanded version of the string `txt`, which may contain texting abbreviations like "brb" and "lol." Your function should expand at least four different texting abbreviations. For example, `txtHelp('imo u r lol brb')` might return the string `'in my opinion you are laugh out loud be right back'`.

6.1.6* Write a function

```
letters(text)
```

that prints the characters of the string `text`, one per line. For example `letters('abc')` should print

```
a
b
c
```

6.1.7* Write a function

```
countCharacter(text, letter)
```

that returns the number of occurrences of the one-character string named `letter` in the string `text`, *without* using the `count` method. (Use a `for` loop and an accumulator instead.)

6.1.8. Write a function

```
vowels(word)
```

that returns the same result as Exercise 6.1.1 *without* using the `count` method. (Use a `for` loop instead.)

6.1.9. Write a function

```
replacePunctuation(text)
```

that uses a `for` loop to return a modified version of the string `text` in which all punctuation characters are replaced by spaces.

6.1.10* Write a function

```
underscore(sentence)
```

that returns the same result as Exercise 6.1.3 *without* using the `replace` method.

6.1.11. Write a function

```
nospaces(sentence)
```

that returns the same result as Exercise 6.1.4 *without* using the `replace` method.

6.1.12. Write a function

```
noVowels(text)
```

that returns a version of the string `text` with all the vowels removed. For example, `noVowels('this is an example.')` should return the string `'ths s n xmpl.'`.

6.1.13. Write a function

```
daffy(word)
```

that returns a string that has Daffy Duck's lisp added to it (Daffy would pronounce the 's' sound as though there was a 'th' after it). For example, `daffy("That's despicable!")` should return the string `"That'sth desthpicable!"`.

6.1.14. Write a function

```
reverse(text)
```

that returns a copy of the string `text` in reverse order.

6.1.15. Create a modified version of the `splitIntoWords` function that just counts the words instead of appending them to a list. Your function should return the word count and should not use a list variable.

6.1.16. Write a function

```
split(text, splitCharacters)
```

that generalizes the `splitIntoWords` function so that it splits `text` at any of the characters, or runs of any of the characters, in the string `splitCharacters`. For example, `split('the best of times', 'sei')` should return the list `['th', ' b', 't of t', 'm']`.

6.1.17* Show how the `wordCount` function can be shortened to a single line by composing functions.

6.1.18. Show how the `wordTokens` function can be shortened to a single line by composing functions.

6.1.19. Test the five functions we developed in this section by completing the `test_textlib.py` program.

The following seven exercises ask you to write the remaining functions in the reading level problem. To guide you, Figure 6.2 shows how data flows between algorithms for the subproblems in Figure 1.4. Consistent with prior diagrams, problem inputs are shown entering on the left and outputs are shown exiting on the right. Values exiting nodes from the bottom are being sent to subproblems a level below as inputs, and the outputs of subproblems are shown returning to the caller. Add each function that you write to your `textlib` *module, and design a unit test for the function in* `test_textlib.py`*.*

6.1.20. Write a function that implements the final Syllable Count algorithm from page 16. Lines 1–3 of the algorithm can together be implemented in a manner very similar to the `splitIntoWords` function. The idea is to only increment the count if a vowel is the first in a run of vowels. Here is a modified algorithm showing this idea.

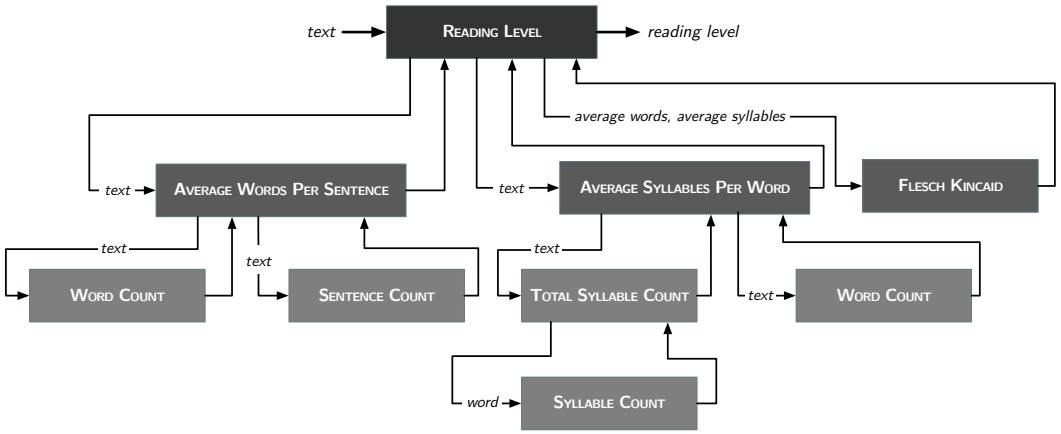


Figure 6.2 Flows of inputs and outputs in the reading level problem from Figure 1.4.

Algorithm Syllable Count (Version 3)

Input: a *word*

1 *count* ← 0
2 repeat for each *letter* in *word*:
3 if *letter* is a vowel and the previous letter is not a vowel, then:
4 *count* ← *count* + 1
5 if *word* ends in e, then:
6 *count* ← *count* − 1

Output: *count*

The last character in the word can be examined with the string method `endswith`:

```
if word.endswith('e'):
    count = count - 1
```

Once you have written the `syllableCount` function, you can get the total number of syllables in a text with the following function.

```
def totalSyllableCount(text):
    wordList = wordTokens(text)
    count = 0
    for word in wordList:   # iterate over each word in wordList
        count = count + syllableCount(word)
    return count
```

Add these two functions to `textlib.py` and test them thoroughly in `test_textlib.py`.

6.1.21. Add the function

```
splitIntoSentences(text)
```

to your `textlib` module. The function should return the number of sentences in the string `text`. This is very similar to the `splitIntoWords` function except

that it splits at runs of end punctuation marks instead of whitespace. The function should also omit all whitespace between sentences. (If it were not for this requirement, you could use the generalized `split` function from Exercise 6.1.16.)

6.1.22. Add the function

```
sentenceTokens(text)
```

to your `textlib` module. The function should make the `text` lowercase and then use your `splitIntoSentences` function from Exercise 6.1.21 to return the list of sentence tokens in the `text`.

6.1.23. Add the function

```
sentenceCount(text)
```

to your `textlib` module. The function should use your `sentenceTokens` function from Exercise 6.1.22 to return the number of sentences in the string `text`.

6.1.24. Add the function

```
averageWords(text)
```

to your `textlib` module. The function should use the `wordCount` function and your `sentenceCount` function from Exercise 6.1.23 to return the average number of words per sentence in the string `text`.

6.1.25. Add the function

```
averageSyllables(text)
```

to your `textlib` module. The function should use your `totalSyllableCount` function from Exercise 6.1.20 and the `wordCount` function to return the average number of syllables per word in the string `text`.

6.1.26. Finally, add the function

```
readingLevel(text)
```

to your `textlib` module. The function should use your `averageWords` and `averageSyllables` functions from Exercises 6.1.24 and 6.1.25, and the `fleschKincaid` function from page 90 to return the Flesch-Kincaid reading level of the string `text`.

6.2 TEXT DOCUMENTS

To apply our text analysis functions to full-size texts, we need to be able to read them from ***files*** stored on a hard drive or flash drive. Like everything else in a computer system, files are stored as sequences of bits. But we interact with files as electronic documents containing information such as text, spreadsheets, or images. These abstractions are mediated by a part of the operating system called the *file system*. The file system organizes files in folders in a hierarchical tree, such as in the simplified view of a macOS file system in Figure 6.3.

The root of the tree is denoted by a forward slash / symbol. Below the root in this figure is a folder named Users where every user of the computer has a *home folder* labeled with his or her name, say george. This home folder contains several subfolders, one of which is Documents. The two subfolders in Documents are named

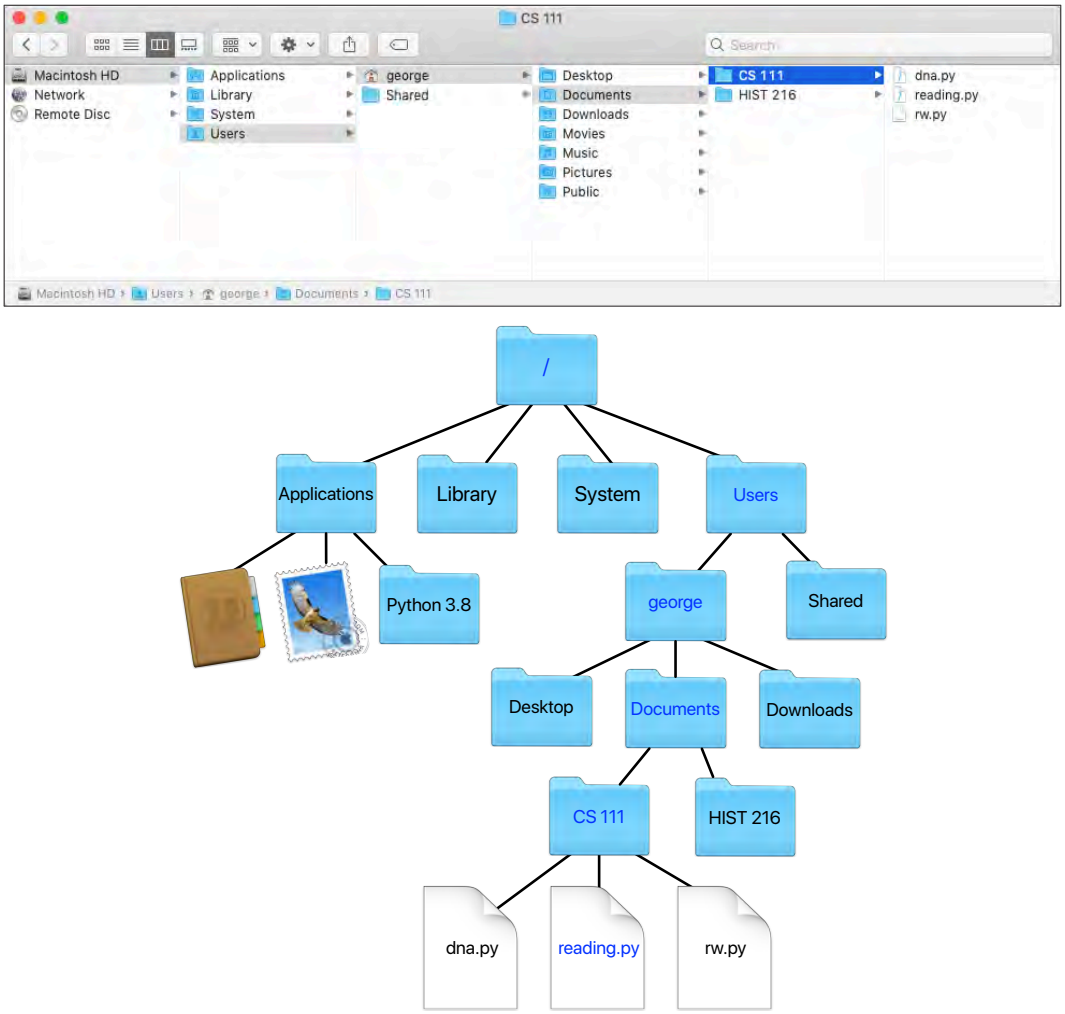


Figure 6.3 A Finder window in macOS and its partial tree representation.

CS 111 and HIST 216. We can represent the location of a file with the *path* one must follow to get there from the root. For example, the path to reading.py, colored blue, is `/Users/george/Documents/CS\ 111/reading.py`. Notice the backslash before the space in `CS\ 111`; this is because spaces usually need to be escaped in pathnames. Any path without the first forward slash is considered to be a *relative path*, relative to the current *working directory* set by the operating system. For example, if the current working directory were `/Users/george/Documents`, then reading.py could be specified with the relative path `CS\ 111/reading.py`.

Reading from text files

In Python, we access files through an abstraction called a ***file object***, which we associate with a file with the `open` function. For example, the following statement

associates the file object named `textFile` with the file named `mobydick.txt` in the current working directory.[3]

```
textFile = open('mobydick.txt', 'r')
```

The second argument to `open` is the *mode* to use when working with the file; `'r'` means that we want to read from the file. If a file with that filename does not exist, a `FileNotFoundError` exception will be raised. If necessary, you can proactively determine in advance whether a file exists and can be read by using a couple of functions from the `os` and `os.path` modules:

```
import os
import os.path

assert os.path.isfile(fileName), fileName + ' does not exist'
assert os.access(fileName, os.R_OK), fileName + ' cannot be read'
```

The value `os.R_OK` is telling the function to check whether Reading the file is OK.

The `read` method of a file object reads the entire contents of a file into a string. For example, the following statement reads the entirety of `mobydick.txt` into a string assigned to the variable `text`.

```
text = textFile.read()
```

When you are finished with a file, it is important to close it. This signals to the operating system that you are done using it, and ensures that any memory allocated to the file is released. To close a file, simply use the file object's `close` method:

```
textFile.close()
```

Let's look at an example that puts all of this together. The following function reads a file with the given file name and returns the number of words in the file using our `wordCount` function from the previous section. (Remember that you will have to save the program containing this function in the same folder as `textlib.py`.)

```
import textlib

def wordCountFile(fileName):
    """Return the number of words in the file with the given name.

    Parameter:
        fileName: the name of a text file

    Return value: the number of words in the file
    """

    textFile = open(fileName, 'r', encoding = 'utf-8')
    text = textFile.read()
    textFile.close()

    return textlib.wordCount(text)
```

[3]Download from the book website or `http://www.gutenberg.org/files/2701/2701-0.txt`.

The optional `encoding` parameter to the `open` function indicates how the bits in the file should be interpreted (we will discuss what UTF-8 is in Section 6.3).

Reflection 6.7 *How many words are there in the file* `mobydick.txt`*?*

Now suppose we want to print a text file, formatted with line numbers to the left of each line. A "line" is defined to be a sequence of characters that end with a newline character. To make this easier, rather than read the whole file in at once, we can read it one line at a time. In the same way that we can iterate over a range of integers or the characters of a string, we can iterate over the lines in a file. When we use a file object as the sequence in a `for` loop, the index variable is assigned a string containing each line in the file, one line per iteration. For example, the following loop prints each line in the file object named `textFile`:

```
for line in textFile:
    print(line)
```

In each iteration of this loop, `line` is assigned the next line in the file, which is then printed in the body of the loop. We can easily extend this idea to a line-numbering function:

```
def lineNumbers(fileName):
    """Print the contents of a file with line numbers added.

    Parameter:
        fileName: the name of a text file

    Return value: None
    """

    textFile = open(fileName, 'r', encoding = 'utf-8')
    count = 1
    for line in textFile:
        print('{0:<5} {1}'.format(count, line.rstrip()))
        count = count + 1
    textFile.close()
```

The `lineNumbers` function combines an accumulator with a `for` loop that reads the text file line by line. After the file is opened, the accumulator variable `count` is initialized to one. Inside the loop, each line is printed using a format string that precedes the line with the current value of `count`. The `rstrip()` method removes whitespace from the right end of the string. (There are also `lstrip()` and `strip()` methods. See Appendix A.6.) At the end of the loop, the accumulator is incremented and the loop repeats.

Reflection 6.8 *What effect does the* `rstrip` *method have? What happens if you replace* `line.rstrip()` *with just* `line`*?*

Reflection 6.9 *How would the output change if* `count` *was incremented before calling* `print` *instead?*

Reflection 6.10 *How many lines are there in the file* `mobydick.txt`*?*

Writing to text files

We can also create new files or write to existing ones. To write text to a new file, we first have to open it using `'w'` ("write") mode:

```
newTextFile = open('newfile.txt', 'w')
```

Opening a file in this way will create a new file named `newfile.txt`, if a file by that name does not exist, or overwrite the file by that name if it does exist. (So be careful!) To append to the end of an existing file, use the `'a'` ("append") mode instead. Once the file is open, we can write text to it using the `write` method:

```
newTextFile.write('Hello.\n')
```

The `write` method does not write a newline character at the end of the string by default, so we have to include one explicitly, as desired.

Remember to always close the new file when you are done.

```
newTextFile.close()
```

Closing a file to which we have written ensures that the changes have actually been written to the drive. To improve efficiency, an operating system does not necessarily write text out to the drive immediately. Instead, it usually waits until a sufficient amount builds up, and then writes it all at once. Therefore, if you forget to close a file and your computer crashes, your program's last writes may not have actually been written. (This is one reason why we sometimes have trouble with corrupted files after a computer crash.)

The following function highlights how to modify the `lineNumbers` function so that it writes the file with line numbers directly to another file instead of printing it.

```
def lineNumbersFile(fileName, newFileName):
    """Write the contents of a file to a new file with line numbers added.

    Parameters:
        fileName:    the name of a text file
        newFileName: the name of the output text file

    Return value: None
    """

    textFile = open(fileName, 'r', encoding = 'utf-8')
    newTextFile = open(newFileName, 'w')
    count = 1
    for line in textFile:
        newTextFile.write('{0:<5} {1}\n'.format(count, line.rstrip()))
        count = count + 1
    textFile.close()
    newTextFile.close()
```

Reading from the web

We can also read text directly from the web using the `urllib.request` module. The function `urlopen` returns a file object abstraction for a web page that is similar to the one returned by `open`. The `urlopen` function takes a web page address, formally known as a *URL*, or *uniform resource locator*, as a parameter. URLs normally begin with the prefix `http://` (http is short for hypertext transfer protocol).

Let's download a book from Project Gutenberg (`www.gutenberg.org`), a vast repository of free classic literature. Mary Shelley's *Frankenstein*, for example, can be downloaded like this:

```
>>> import urllib.request as web
>>> webPage = web.urlopen('http://www.gutenberg.org/files/84/84-0.txt')
>>> rawBytes = webPage.read()
>>> webPage.close()
```

Behind the scenes, the Python interpreter communicates with a web server over the Internet to read this web page. But thanks to the magic of abstraction, we did not have to worry about any of those details. To find the URL for a different book, search for it on Project Gutenberg's web page and then find the link for the "Plain Text UTF-8" version. After you click on the link, copy the URL and use it as the argument to the `urlopen` function.

Because the `urlopen` function does not accept an encoding parameter, the `read` function cannot tell how the text of the web page is encoded. Therefore, `read` returns a `bytes` object instead of a string. A `bytes` object contains a sequence of raw bytes that are not interpreted in any particular way. To convert the `bytes` object to a string before we print it, we can use the `decode` method, as follows:

```
>>> text = rawBytes.decode('utf-8')
```

(Again, we will see what UTF-8 is in Section 6.3.) Now `text` refers to the entire 440 KB text of *Frankenstein* and can be used like any other string.

Reflection 6.11 *Write a short program with these statements that uses your* `wordCount` *function to find the number of words in* Frankenstein. *(You should get about 78,000.)*

It's worth noting that the vast majority of content on the web is not plain text like this. Rather it is written in HTML (short for hypertext markup language), which is the language that web browsers "understand." To see what HTML looks like, try this:

```
>>> webPage = web.urlopen('http://stibitz.denison.edu')
>>> rawBytes = webPage.read()
>>> rawBytes.decode('utf-8')
```

What you see printed is the HTML code for the main web page at this address.

Exercises

6.2.1. If you implemented the `readingLevel` function in Exercise 6.1.26, write a function

```
readingLevelFile(fileName)
```

that returns the Flesch-Kincaid reading level of the file with the given `fileName`.

6.2.2. Modify the `lineNumbers` function so that it only prints a line number on every tenth line (for lines $1, 11, 21, \ldots$).

6.2.3* Write a function

```
lowerCaseFile(fileName)
```

that prints the contents of a file with every character converted to lowercase. Read the file one line at a time in a loop.

6.2.4. Write a function

```
wordCountLines(fileName)
```

that uses the `wordCount` function from your `textlib` module to print the number of words in each line of the file with the given `fileName`.

6.2.5. Write a function

```
paragraphCount(fileName)
```

that returns the number of paragraphs in the file with the given `fileName`. Assume that paragraphs are separated by one or more blank lines.

6.2.6* Write a function

```
plotWordsPerParagraph(fileName)
```

that uses `matplotlib.pyplot` and the `wordCount` function from your `textlib` module to plot the number of words in the paragraphs of the file with the given `fileName`. Assume that paragraphs are separated by one or more blank lines.

6.2.7. Write a function

```
plotIsPerParagraph(fileName)
```

that plots the *fraction* of words that are the word "I" in the paragraphs of `fileName`. You may find the string method `count` helpful. Assume that paragraphs are separated by one or more blank lines. Use `matplotlib.pyplot` and the `wordCount` function from your `textlib` module.

6.2.8. Write a function

```
plotWordsPerChapter(fileName, chapterOneStart)
```

that uses `matplotlib.pyplot` and the `wordCount` function from your `textlib` module to plot the number of words in the chapters of the book with the given `fileName`. Assume that new chapters begin when the word "chapter" is the first non-whitespace word on a line and the previous line is blank. (See the string method `startswith` in Appendix A.6.) The parameter `chapterOneStart` is the line number of the first chapter in the book. Your function should begin by skipping this many lines to get past the table of contents and any other front matter in the text. Remember to include the last chapter (marked by the end of the file) in your plot. You can test your function with the files `mobydick.txt` and `frankenstein.txt` on the book website. The first chapters in these books start on lines 489 and 684, respectively.

6.2.9. Write a function

```
strip(fileName, newFileName)
```

that creates a new version of the file with the given `fileName` in which all whitespace has been removed. The second parameter is the name of the new file.

6.2.10* Write a function

```
writeLineLengths(fileName, outputFile)
```

that writes the lengths of all the lines in `fileName` (not counting newline characters) to a new file named `outputFile`. Each line in the output file should simply be the length of one line.

6.2.11. Write a function

```
writeWordsPerParagraph(fileName, outputFile)
```

that writes the number of words in each paragraph in `fileName` to a new file named `outputFile`. Each line in the output file should contain a paragraph number and the number of words in that paragraph, like this:

```
1: 9
2: 44
3: 6
⋮
```

6.2.12. Write a function that makes a new copy of a file in which every sentence begins on a new line. You may assume that every period, question mark, and exclamation point in the file end a sentence. Before you process each line, use the string method `strip` to remove whitespace (including the newline) from both ends of the line. Then add a space at the end of the line (after the last word). For an extra challenge, also prevent whitespace from appearing at the beginning of each line before a sentence. Your function should take two parameters: the name of the original file and the name of the new file.

6.2.13* Write a function

```
wordCountWeb(url)
```

that reads the text file from the given URL and returns the number of words in the file using the `wordCount` function from your `textlib` module. Test your function on books from Project Gutenberg. Alternatively, you can access mirrored copies of books from the book website.

6.2.14. According to the HTML standard specification, every HTML web page should begin with the declaration

```
<!DOCTYPE html>
```

This declaration is not case sensitive and may contain additional elements such as

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

Write a function

```
isHTML(url)
```

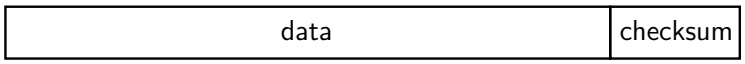
that uses this information to detect whether a web page is an HTML document or not. The function should return `True` if it is HTML and `False` otherwise.

6.3 ENCODING STRINGS

As web pages travel across the Internet to your browser, the transmissions can be corrupted by faulty equipment or electromagnetic interference. Errors can be as small as a single bit being "flipped" or as large as an entire chunk of data being lost. In this section, we will discuss how knowing something about how text is stored at a lower level enables us to detect these kinds of errors. This knowledge will also prove essential for some of the more interesting text analysis we will do in the following sections.

Computing checksums

Network protocols detect errors by sending additional information, called a ***checksum***, along with the data.



In a nutshell, a checksum is used to tell whether what is received is the same as what was sent. When the transmission is received at the destination, a checksum is computed for the data and compared to the transmitted checksum. If they match, all is well. If they don't match, then the receiver asks the sender to retransmit.

The following algorithm represents the simplest way to compute a checksum for a raw sequence of bytes. It just adds the bytes together, each time taking the remainder mod 256 so that the checksum value will fit in one byte. (Recall that one byte is 8 bits, so a byte can store values between 0 and $2^8 - 1 = 255$.)

Algorithm SIMPLE CHECKSUM

Input: a *byte sequence*

```
1  checksum ← 0
2  repeat for each byte in the byte sequence:
3      checksum ← (checksum + byte) mod 256
```

Output: *checksum*

For example, suppose we wanted to send the following byte sequence:

42	207	111	199

The algorithm will compute a checksum like this:

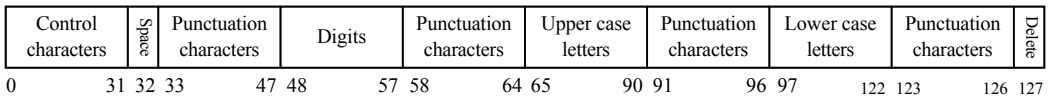


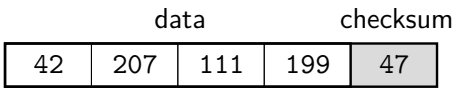
Figure 6.4 A not-to-scale overview of the organization of the ASCII character set (and the Basic Latin segment of Unicode) with decimal code ranges.

Trace input: *byte sequence* = [42, 207, 111, 199]

Step	Line	*byte*	*checksum*	Notes
1	1	—	0	*checksum* ← 0
2	2	42	″	*byte* ← 42
3	3	″	42	*checksum* ← (0 + 42) mod 256 = 42
4	2	207	″	*byte* ← 207
5	3	″	249	*checksum* ← (42 + 207) mod 256 = 249
6	2	111	″	*byte* ← 111
7	3	″	104	*checksum* ← (249 + 111) mod 256 = 104
8	2	199	″	*byte* ← 199
9	3	″	47	*checksum* ← (104 + 199) mod 256 = 47

Return value: 47

The computed checksum is 47, so we append this to the transmitted message:



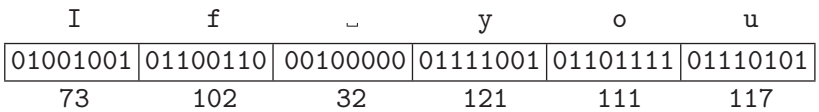
Unicode

To apply this algorithm to text, we need to look more closely at how strings are encoded in binary in a computer's memory. English language text has historically been encoded in a format known as ***ASCII*** (pronounced "ASS-key").[4] ASCII assigns each character a 7-bit binary code. In memory, each ASCII character is stored in one byte, with the leftmost bit of the byte being a 0. So a string is stored as a sequence of bytes. For example, the first six letters of the quote

If you don't like something, change it. If you can't change it, change your attitude.

—Maya Angelou

are encoded in ASCII as



[4]ASCII is an acronym for American Standard Code for Information Interchange.

The values underneath are the decimal equivalents of the binary codes. Figure 6.4 uses these decimal values to illustrate the organization of the ASCII character set. Notice that different types of characters are grouped together. Digits are in the range 48–57, uppercase letters are in 65–90, lowercase letters are in 97–122, etc. Python uses this encoding to define "alphabetical order" when comparing strings.

Reflection 6.12 *Consult Figure 6.4 to explain each of the following results.*

```
>>> '3.14159' < 'pi'
True
>>> 'pi' == 'Pi'
False
>>> 'Zebra' < 'antelope'
True
>>> '314159' < '32'
True
```

The ASCII character set has been largely supplanted, including in Python, by an international standard known as ***Unicode***. Whereas ASCII only provides codes for Latin characters, Unicode encodes over 100,000 different characters from more than 100 languages, using up to 4 bytes per character. A Unicode string can be encoded in one of three ways, but is most commonly encoded using a variable-length system called UTF-8 (that we used in the previous section). Conveniently, UTF-8 is backwards-compatible with ASCII, so each character in the ASCII character set is encoded in the same 1-byte format in UTF-8.

In Python, we can view the Unicode (in decimal) for any character using the `ord` function (short for "ordinal"). For example,

```
>>> ord('I')
73
```

The `chr` function is the inverse of `ord`; given a Unicode value, `chr` returns the corresponding character.

```
>>> chr(73)
'I'
```

Reflection 6.13 *Use* `ord` *on the first characters of the strings in Reflection 6.12 to explain the results of the comparisons.*

The following Python function uses `ord` to apply our SIMPLE CHECKSUM algorithm to strings. Once the one-byte checksum is computed, we convert it to a character using `chr` so that we can concatenate it to the string before sending it.

```
def simpleChecksum(text):
    """Compute a simple one-character checksum for a string.

    Parameter:
        text: a string

    Return value: a character representing the one-byte checksum
    """
```

```
    checksum = 0
    for character in text:
        checksum = (checksum + ord(character)) % 256

    return chr(checksum)
```

Let's use this function to create a checksum for our Maya Angelou quote.

```
>>> quote = "If you don't like something, change it.  If you can't
             change it, change your attitude."
>>> checksum = simpleChecksum(quote)
>>> checksum
'ç'
```

The computed checksum value was 231, which corresponds to a lowercase *c* with a cedilla in Unicode. (The character representation of the checksum doesn't really matter since it is not really part of the text.) To create a message to send across a network, we would next concatenate the string and the checksum.

```
>>> message = quote + checksum
>>> message
"If you don't like something, change it.  If you can't change it,
 change your attitude.ç"
```

This simple checksum algorithm is actually too weak to be used in practice. Most notably, it cannot detect when two bytes are sent out of order.

Reflection 6.14 *Why does the simple checksum algorithm have this problem?*

Fletcher's checksum algorithm fixes this by adding a second checksum that incrementally sums the values of the first checksum.

```
def fletcherChecksum(text):
    """Compute a two character checksum for a string using the
       Fletcher-16 algorithm.

    Parameter:
        text: a string

    Return value: a two-character string representing the checksum
    """

    checksum1 = 0
    checksum2 = 0
    for character in text:
        checksum1 = (checksum1 + ord(character)) % 255
        checksum2 = (checksum2 + checksum1) % 255

    return chr(checksum2) + chr(checksum1)
```

Note that this algorithm also differs in that it mods by 255 instead of 256. The return value is a two-character string created by concatenating the characters corresponding to the two checksum values.

Tangent 6.2: Compressing text files

If a text file is stored in UTF-8 format, then each character is represented by an eight-bit code, requiring one byte of storage per character. For example, the file `mobydick.txt` contains about 1.2 million characters, so it requires about 1.2 MB of disk space. But text files can usually be modified to use far less space, without losing any information. Suppose that a text file contains upper and lowercase letters, plus whitespace and punctuation, for a total of sixty unique characters. Since $2^6 = 64$, we can adopt an alternative encoding scheme in which each of these sixty unique characters is represented by a six-bit code instead. By doing so, the text file will use only $6/8 = 75\%$ of the space.

The ***Huffman coding*** algorithm can do even better by using variable-length codes that are shorter for more frequent characters. As a simple example, suppose a 23,000-character text file contains only five unique characters: A, C, G, N, and T with frequencies of 5, 6, 4, 3, and 5 thousand, respectively. Using the previous fixed-length scheme, we could devise a three-bit code for these characters and use only $3 \cdot 23{,}000 = 69{,}000$ bits instead of the original $8 \cdot 23{,}000 = 184{,}000$ bits. But, by using a prefix code that assigns the more frequent characters A, C, and T to shorter two-bit codes (A = `10`, C = `00`, and T = `11`) and the less frequent characters G and N to three-bit codes (G = `010` and N = `011`), we can store the file in

$$2 \cdot 5{,}000 + 2 \cdot 6{,}000 + 3 \cdot 4{,}000 + 3 \cdot 3{,}000 + 2 \cdot 5{,}000 = 53{,}000$$

bits instead. This is called a *prefix code* because no code is a prefix of another code, which is essential for decoding the file.

An alternative compression technique, used by the ***Lempel-Ziv-Welch algorithm***, replaces repeated strings of characters with fixed-length codes. For example, in the string `CANTNAGATANCANCANNAGANT`, the repeated sequences `CAN` and `NAG` might each be represented with its own code.

Reflection 6.15 *Show how to compute the Fletcher checksums by hand for the string* `'abc'`. *(You should get 39 and 76 for* `checksum1` *and* `checksum2`*, respectively, corresponding to the string* `"L'"`.*)*

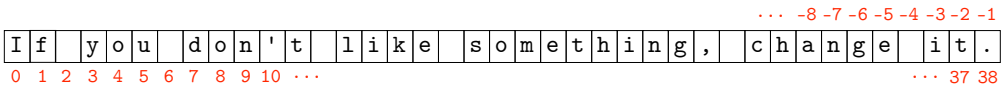
We can now apply the Fletcher checksum algorithm to our quote like this:

```
>>> checksum = fletcherChecksum(quote)
>>> checksum
'\r\x05'
>>> message2 = quote + checksum
>>> message2
"If you don't like something, change it.  If you can't
 change it, change your attitude.\r\x05"
```

(The character `'\r'` is the "carriage return" character and `'\x05'` represents the symbol with ASCII code 5, which is an antiquated non-printable control character that was once used to request a response from computer terminals and teletype machines. But their meanings are irrelevant here.)

Indexing and slicing

To confirm that a message is error-free, the receiver needs to remove the two-character checksum and compare it to a checksum that it computes itself on the remaining text. To implement this, we need to be able to directly access those last few characters. Conceptually, we already know that a string is stored as a sequence of bytes, where each byte represents one character. These bytes are stored in contiguous memory cells. So the first sentence of the Maya Angelou quote, `"If you don't like something, change it."`, can be represented like this:



Each character in the string is identified by an *index* that indicates its position. Indices always start from the left at 0, as shown below the characters above. We can also use *negative indexing*, which starts from the right end of the string, as shown above the characters. We can use these indices to access a character directly by referring to the index in square brackets following the name of the string. For example,

```
>>> shortQuote = "If you don't like something, change it."
>>> shortQuote[0]
'I'
>>> shortQuote[9]
'n'
>>> shortQuote[-30]
'n'
>>> shortQuote[-1]
'.'
```

Notice that each character is itself represented as a single-character string in quotes, and that `quote[9]` and `quote[-30]` refer to the same character. The last character in a string can always be accessed with index `-1`, regardless of the string's length. We can use this to access the checksum character in the first `message`.

```
>>> message[-1]
'ç'
```

To get a string's length, we use the `len` function:

```
>>> len(shortQuote)
39
>>> shortQuote[38]
'.'
>>> shortQuote[39]
IndexError: string index out of range
```

Reflection 6.16 *Why does the last statement above result in an error?*

Notice that `len` returns the number of characters in the string, *not* the index of the last character. The positive index of the last character in a string is always the

length of the string minus one. As shown above, referring to an index that does not exist will give an ***index error*** exception.

To access a substring consisting of multiple characters, like the two-character Fletcher checksum, we use *slicing*. Slice notation uses two indices separated by a colon. The first index is the position of the first character in the slice and the second is the index of the character *just past* the last character in the slice (analogous to how `range` stops just shy of its argument). So we can get the Fletcher checksum from `message2` like this:

```
>>> message2[len(message2)-2:len(message2)]
'\r\x05'
```

or, much more simply,

```
>>> message2[-2:]
'\r\x05'
```

Here we used a negative index for the beginning of the slice and omitted the second index, which means that we want the slice to go to the end of the string. Similarly, if we want a slice from the beginning of a string, we can omit the first index. So `message[:-1]` will give us the "data" portion of the first message and `message2[:-2]` will give us the "data" portion of the second message.

```
>>> message[:-1]
"If you don't like something, change it.  If you can't change it, ..."
>>> message2[:-2]
"If you don't like something, change it.  If you can't change it, ..."
```

The following function uses slicing to verify whether a message with a Fletcher checksum was "received" correctly. If the checksum is correct, it returns the data portion of the message. Otherwise, it returns an empty string.

```
def verifyMessage(message):
    """Verify a message with a Fletcher-16 checksum.

    Parameter:
        message: a string containing data + checksum

    Return value: the data if verified or '' if not
    """

    data = message[:-2]
    checksum = message[-2:]
    if fletcher(data) == checksum:
        return data
    else:
        return ''
```

We will continue to use indexing and slicing in the next sections, as we explore more sophisticated techniques to analyze large texts.

Exercises

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

6.3.1. Suppose you have a string stored in a variable named `word`. Show how you would print

(a)* the string's length

(b)* the first character in the string

(c)* the third character in the string

(d)* the last character in the string

(e) the last three characters in the string

(f) the string consisting of the second, third, and fourth characters

(g) the string consisting of the fifth, fourth, and third to last characters

(h) the string consisting of all but the last character

6.3.2. The following string is a quote by Benjamin Franklin.

```
quote = 'Well done is better than well said.'
```

Use slicing notation to answer each of the following questions.

(a)* What slice of `quote` is equal to `'done'`?

(b)* What slice of `quote` is equal to `'well said.'`?

(c) What slice of `quote` is equal to `'one is bet'`?

(d) What slice of `quote` is equal to `'Well do'`?

6.3.3. What are the values of each of the following expressions? Explain why in each case.

(a)* `'cat' < 'dog'`

(b)* `'cat' < 'catastrophe'`

(c)* `'cat' == 'Cat'`

(d) `'1' > 'one'`

(e) `'8188' < '82'`

(f) `'many' > 'One'`

6.3.4* When we print a numeric value using the `print` function, each digit in the number must be converted to its corresponding character to be displayed. In other words, the value `0` must be converted to the character `'0'`, the value `1` must be converted to the character `'1'`, etc. The Unicode codes for the digit characters are conveniently sequential, so the code for any digit character is equal to `ord('0')` plus the value of the digit. For example, `ord('2')` is the same as `ord('0') + 2` and `chr(ord('0') + 2)` is `'2'`. Write a function

```
digit2String(digit)
```

that generalizes this example to return the string equivalent of the number `digit`. For example, `digit2String(4)` should return `'4'`. If `digit` is not the value of a decimal digit, return `None`.

6.3.5. Suppose we want to convert a letter to an integer representing its position in the alphabet. In other words, we want to convert `'A'` or `'a'` to 1, `'B'` or `'b'` to 2, etc. Like the characters for the digits, the codes for the uppercase and lowercase letters are in consecutive order. Therefore, for an uppercase letter, we can subtract the code for `'A'` from the code for the letter to get the letter's offset relative to `'A'`. Similarly, we can subtract the code for `'a'` from the code for a lowercase letter. For example, `ord('D') - ord('A')` is 3. Write a function

`letter2Position(letter)`

that uses this idea to return the position in the alphabet (1–26) of the upper or lowercase `letter`. If `letter` is not a letter, return `None`.

6.3.6. Write a function

`position2Letter(n)`

that returns the `n`th uppercase letter in the alphabet, using the `chr` and `ord` functions.

6.3.7. Write a function

`string2Digit(digitString)`

that returns the integer value corresponding to the string `digitString`. The parameter will contain a single character `'0'`, `'1'`, ..., `'9'`. Use the `ord` function. For example, `string2Digit('5')` should return the integer value 5.

6.3.8. Any exam score between 60 and 99 can be converted to a letter grade with a single expression using `chr` and `ord`. Demonstrate this by replacing `SOMETHING` in the function below.

```
def letterGrade(score):
    if grade >= 100:
        return 'A'
    if grade > 59:
        return SOMETHING
    return 'F'
```

6.3.9. Write a function

`capitalize(word)`

that uses `ord` and `chr` to return a version of the string `word` with the first letter capitalized. (Note that the word may already be capitalized!)

6.3.10. Write a function

`int2String(number)`

that converts any positive integer value `number` to its string equivalent, *without* using the `str` function. For example, `int2String(1234)` should return the string `'1234'`. (Use the `digit2String` function from Exercise 6.3.4.)

6.3.11. Suppose you work for a state in which all vehicle license plates consist of a string of letters followed by a string of numbers, such as `'ABC 123'`. Write a function

`randomPlate(length)`

that returns a string representing a randomly generated license plate consisting of `length` uppercase letters followed by a space followed by `length` digits. Use the `random.randrange` function.

6.3.12. Write a function

`username(first, last)`

that constructs and returns a username, specified as the last name followed by an underscore and the first initial. For example, `username('martin', 'freeman')` should return the string `'freeman_m'`.

6.3.13. Write a function

`piglatin(word)`

that returns the Pig Latin equivalent of the string `word`. If the first character is a consonant, Pig Latin moves it to the end, and follows it with `'ay'`. If the first character is a vowel, nothing is moved and `'way'` is added to the end. For example, Pig Latin translations of `'python'` and `'asp'` are `'ythonpay'` and `'aspway'`.

6.3.14. Write a function

`pigLatinDict(fileName)`

that prints the Pig Latin equivalent of every word in the dictionary file with the given file name. (See Exercise 6.3.13.) Assume there is exactly one word on each line of the file. Start by testing your function on small files that you create. An actual dictionary file can be found on most Mac OS X and Linux computers at `/usr/share/dict/words`. There is also a dictionary file available on the book website.

6.3.15. Repeat the previous exercise, but have your function write the results to a new file instead, one Pig Latin word per line. Add a second parameter for the name of the new file.

6.3.16* When some people get married, they choose to take the last name of their spouse or hyphenate their last name with the last name of their spouse. Write a function

`marriedName(fullName, spouseLastName, hyphenate)`

that returns the person's new full name with hyphenated last name if `hyphenate` is `True` or the person's new full name with the spouse's last name if `hyphenate` is `False`. The parameter `fullName` is the person's current full name in the form `'Firstname Lastname'` and the parameter `spouseLastName` is the spouse's last name. For example, `marriedName('Jane Doe', 'Deer', True)` should return the string `'Jane Doe-Deer'` and `marriedName('Jane Doe', 'Deer', False)` should return the string `'Jane Deer'`.

6.3.17. *Parity checking* is an even simpler error detection algorithm that is used directly on sequences of bits (often called *bit strings*). A bit string has *even parity* if it has an even number of ones, and *odd parity* otherwise. In an even parity scheme, the sender adds a single bit to the end of the bit string so that the final bit string has an even number of ones. For example, if we wished to send the data `1101011`, we would actually send `11010111` instead so that the bit string has an

even number of ones. If we wished to send `1101001` instead, we would actually send `11010010`. The receiver checks whether the received bit string has even parity; if it does not, the receiver requests a retransmission.

(a) Parity can only detect very simple errors. Give an example of an error that cannot be detected by an even parity scheme.

(b) Propose a solution that would detect the example error you gave above.

(c) In the next two problems, we will pretend that bits are sent as strings (they are not; this would be terribly inefficient). Write a function

```
evenParity(bits)
```

that uses the `count` method to return `True` if the string `bits` has even parity and `False` otherwise. For example, `evenParity('110101')` should return `True` and `evenParity('110001')` should return `False`.

(d) Now write the `evenParity` function without using the `count` method.

(e) Write a function

```
makeEvenParity(bits)
```

that returns a string consisting of `bits` with one additional bit concatenated so that the returned string has even parity. Your function should call your `evenParity` function. For example, `makeEvenParity('110101')` should return `'1101010'` and `makeEvenParity('110001')` should return `'1100011'`.

6.3.18. Julius Caesar is said to have sent secret correspondence using a simple encryption scheme that is now known as the Caesar cipher. In the Caesar cipher, each letter in a text is replaced by the letter some fixed distance, called the *shift*, away. For example, with a shift of 3, A is replaced by D, B is replaced by E, etc. At the end of the alphabet, the encoding wraps around so that X is replaced by A, Y is replaced by B, and Z is replaced by C. Write a function

```
encipher(text, shift)
```

that returns the result of encrypting `text` with a Caesar cypher with the given `shift`. Assume that `text` contains only uppercase letters.

6.3.19. Modify the `encipher` function from the previous problem so that it either encrypts or decrypts `text`, based on the value of an additional Boolean parameter.

6.4 A CONCORDANCE

A *concordance* is an alphabetical listing of all the words in a text, with their contexts. The context is usually one or more lines in which the target word appears. Suppose we want to know where "lash" appears in the text of *Moby Dick*. If we searched, we would find matches on 60 lines, the first 6 of which are:

```
things not properly belonging to the room, there was a hammock lashed
ship was gliding by, like a flash he darted out; gained her side; with
which to manage the barrow--Queequeg puts his chest upon it; lashes it
blow her homeward; seeks all the lashed sea's landlessness again;
sailed with. How he flashed at me!--his eyes like powder-pans! is he
I was so taken all aback with his brow, somehow. It flashed like a
```

Note that we are not necessarily looking for complete words. When we search for the root of a word like "lash," we might also be interested in derivatives like "lashes" and "lashed." But we might also get other words that contain "lash," like "flash" and "flashed." (We will leave finding only complete words as an exercise.)

To make viewing this information easier, we will line up the words and note the line on which each appears in the text:

```
1188 ... properly belonging to the room, there was a hammock lashed
2458                              ship was gliding by, like a flash he ...
2551 ... manage the barrow--Queequeg puts his chest upon it; lashes it
4396                          blow her homeward; seeks all the lashed ...
5103                                    sailed with. How he flashed at ...
5127    I was so taken all aback with his brow, somehow. It flashed like a
```

We will focus here on creating a concordance entry for just one word. Once we have written a function to do this, it will actually be quite easy to create an entire concordance, but the result could be quite large (e.g., *Moby Dick* contains over 20,000 unique words) and we have not yet discussed how we could quickly search through such a large file for a desired entry.

To create a concordance entry, we will iterate over the lines of the text file, and search for the target word in each line. For each line in which the target word is found, we will print the line, prefaced by a line number, lining up the words in a column for easy reading. Here is the algorithm in pseudocode:

Algorithm CONCORDANCE ENTRY

Input: a *text file*, a *target word*

```
1  line number ← 1
2  repeat for each line in the text file:
3      index ← FIND (line, target word)
4      if the target word was found in line, then:
5          print the line number and the line, using index to line up the target words
6      line number ← line number + 1
```

Output: none

Finding a word

The FIND algorithm, which we will write next, will search through a line and return the index of the first occurrence of the target word in that line, if it exists. There is an existing string method to do just this.[5]

```
>>> benFranklin = 'Diligence is the mother of good luck.'
>>> benFranklin.find('good')
27
```

[5]This quote is from *The Way to Wealth* (1758) by Benjamin Franklin.

But we will implement a `find` function from scratch because, as with `splitIntoWords`, the technique involved is important and useful for other problems down the road. The idea is to make a pass across the text, comparing the target string to slices of the text with the same length as the target. When a matching slice is found, we want to return the index in the text where that slice begins. In pseudocode, the algorithm looks like this:

Algorithm FIND

Input: *text*, *target*

```
1  target index ← −1
2  repeat for each slice of text with the same length as target:
3      if slice = target, then:
4          target index ← starting position of the slice in text
5          break out of the loop
```

Output: *target index*

The variable *target index* will store the index of the matching slice when it is found. It is initialized to −1 at the beginning of the algorithm to signify that a match has not been found yet. We chose the value −1 because this is not a value that could possibly be returned by the algorithm if the target is found. Notice that the value of *target index* remains −1 if a matching slice is never found in the loop. If a matching slice is found, then *target index* is assigned to the index of that slice and we exit the loop immediately so that a possible later match does not overwrite this value.

To see how to implement this in Python, let's first consider the simpler problem of searching for a single character in a string. If we iterate over the characters in the string to search for the target character, as we have done in all of our string algorithms to this point, it would look like this:

```
for character in text:
    if character == targetCharacter:  # target character is found
        targetIndex = ???             # get the index where it was found?
```

But when we find the target character, we are left without a satisfactory return value because we do not know the index of `character`!

Instead, we need to iterate over the *indices* of `text` so that, when we find the target character, we know where it is located in the string. In other words, for all values of `index` equal to `0, 1, 2, ...`, we need to test whether `text[index]` is equal to `targetCharacter`. If this condition is true, then we know that `targetCharacter` exists at position `index`!

Reflection 6.17 *How can we get a list of every index in a string to use in a* `for` *loop?*

The list of indices in a string named `text` is `0, 1, 2, ..., len(text) - 1`. This is precisely the list of integers given by `range(len(text))`. So our desired `for` loop looks like the following:

```
for index in range(len(text)):
    if text[index] == targetCharacter: # target character is found
        targetIndex = index            # get the index where it was found
```

Now when we find that `text[index] == targetCharacter`, we know that the desired character is at position `index`.

Reflection 6.18 *Although it isn't always necessary, we can use this kind of loop any time we need to iterate over a string. Use the examples above to show how to accomplish exactly the same thing as the following* `for` *loop by iterating over the indices of* `text` *instead:*

```
for character in text:
    print(character)
```

The following function uses the new loop above to find a target character in a string.

```
1  def findCharacter(text, targetCharacter):
2      """Find the index of first occurrence of a target character in text.

3      Parameters:
4          text:            a string object to search in
5          targetCharacter: a character to search for

6      Return value: index of the first occurrence of targetCharacter in text
7      """

8      targetIndex = -1                          # assume it won't be found
9      for index in range(len(text)):
10         if text[index] == targetCharacter:  # if found, then
11             targetIndex = index             #   remember where
12             break                           #   and exit the loop early
13     return targetIndex
```

The `break` statement on line 12 exits the loop immediately, even if it is not yet done.

Reflection 6.19 *What do you get when you call this function with* `findCharacter('Diligence is the mother of good luck.', 'g')`*? If you remove the* `break` *statement from the loop, what do you get? Why?*

We need to exit the loop when the first occurrence of `targetCharacter` is found because, if we don't and `targetCharacter` occurs again later in `text`, then `targetIndex` will be overwritten, and the index of the *last* occurrence will be returned instead. The following trace table shows this more explicitly, along with how the changing value of `index` affects the value of `text[index]` in the `if` condition.

Trace arguments: `text = 'Diligence is the mother of good luck.', targetCharacter = 'g'`

Step	Line	`targetIndex`	`index`	`text[index]`	Notes
1	8	-1	—	—	`targetCharacter` ← −1
2	9	″	0	`'D'`	`index` ← 0
3	10	″	″	″	`'D' != 'g'`; skip lines 11-12
4	9	″	1	`'i'`	`index` ← 1
5	10	″	″	″	`'i' != 'g'`; skip lines 11-12
6	9	″	2	`'l'`	`index` ← 2
7	10	″	″	″	`'l' != 'g'`; skip lines 11-12
8	9	″	3	`'i'`	`index` ← 3
9	10	″	″	″	`'i' != 'g'`; skip lines 11-12
10	9	″	4	`'g'`	`index` ← 4
11	10	″	″	″	`'g' == 'g'`; do lines 11-12
12	11	4	″	″	`targetIndex` ← `index`
13	12	″	″	″	break from loop; go to line 13
14	13	″	″	″	return 4

Return value: 4

To generalize this function to find a target string of any length, we need to compare the target string to all *slices* with the same length as the target in `text`. For example, suppose we want to search for the target string `'good'` in `text`. We would need to check whether `text[0:4]` is equal to `'good'`, then whether `text[1:5]` is equal to `'good'`, then whether `text[2:6]` is equal to `'good'`, etc. More concisely, for all values of `index` equal to `0, 1, 2, ...`, we need to test whether `text[index:index + 4]` is equal to `'good'`. In general, to find a target string named `target`, we need to test whether `text[index:index + len(target)]` is equal to `target`, as in the following function.

```
1 def find(text, target):
2     """Find the index of the first occurrence of a target string in text.

3     Parameters:
4         text:   a string object to search in
5         target: a string object to search for

6     Return value: the index of the first occurrence of target in text
7     """

8     targetIndex = -1
9     for index in range(len(text) - len(target) + 1):
10        if text[index:index + len(target)] == target:
11            targetIndex = index
12            break
13    return targetIndex
```

Notice how similar this is to `findCharacter` and that, if `len(target)` equals 1, the `find` function does exactly the same thing as `findCharacter`.

Reflection 6.20 *Why is the last index in the* `for` *loop equal to* `len(text) - len(target)` *instead of* `len(text) - 1`*?*

Suppose `text` is `'Diligence is the mother of good luck.'` and `target` is `'good'`. Then `len(text)` is 37 and `len(target)` is 4. If we had the loop iterate until `index` was `len(text) - 1` = 36, then the last three slices to be examined would be the strings corresponding to `text[34:38]`, `text[35:39]`, and `text[36:40]`, which are `'ck.'`, `'k.'`, and `'.'`, respectively. But these strings are too short to possibly be equal to this `target`. In general, we never need to look at a slice that starts after `len(text) - len(target)`, hence we use `range(len(text) - len(target) + 1)`.

Reflection 6.21 *Is what we just said really true? What is returned by a slice that extends beyond the last character (e.g.,* `'good'[2:10]`*)? What is returned by a slice that starts beyond the last character in the string (e.g.,* `'good'[4:8]`*)?*

Let's look more closely at how `find` works when we call it with these arguments. We will omit `targetIndex` from the trace table this time to save space and instead show `text[index:index+len(target)]`.

Trace arguments: `text = 'Diligence is the mother of good luck.', target = 'good'`

Step	Line	`index`	`text[index:index+4]`	Notes
1	8	—	—	initialize `targetIndex` ← −1
2	9	0	`'Dili'`	`index` ← 0
3	10	″	″	`'Dili' != 'good'`; skip lines 11-12
4	9	1	`'ilig'`	`index` ← 1
5	10	″	″	`'ilig' != 'good'`; skip lines 11-12
6	9	2	`'lige'`	`index` ← 2
7	10	″	″	`'lige' != 'good'`; skip lines 11-12
⋮				
54	9	26	`' goo'`	`index` ← 26
55	10	″	″	`' goo' != 'good'`; skip lines 11-12
56	9	27	`'good'`	`index` ← 27
57	10	″	″	`'good' == 'good'`; do lines 11-12
58	11	″	″	`targetIndex` ← `index`
59	12	″	″	break out of the loop; go to line 13
60	13	″	″	return 27

Return value: 27

A concordance entry

We are finally ready to use our `find` function to print a concordance entry. Here is an outline of a function that follows our pseudocode algorithm.

```
def concordanceEntry_Draft(textFile, targetWord):
    """Print all lines in a text file containing the target word.

    Parameters:
        textFile:   a file object
        targetWord: the word to search for

    Return value: None
    """

    lineNumber = 1
    for line in textFile:
        index = find(line.lower(), targetWord)
        # if targetWord is found in line (using the find function):
            print(lineNumber, line)  # not formatted nicely yet
        lineNumber = lineNumber + 1
```

Reflection 6.22 *When we call the* `find` *function to search for* `targetWord`*, how do we know if it was found?*

Because `find` returns `-1` if the target string is not found, any nonnegative value of `index` means that it was found. So the comment above will be replaced with

```
index = find(line.lower(), targetWord)
if index >= 0:
```

To line up the printed lines nicely, we will use a format string to ensure that the line number always takes up the same amount of space and the rightmost ends of the target words line up. We can do this by splitting the line at the end of the `targetWord`, right justifying the first half, and left justifying the second half.

```
align = index + len(targetWord)  # index of the end of targetWord in line
print('{0:<6}{1:>80}{2}'.format(lineNumber, line[:align], line[align:-1]))
```

Incorporating these changes, here is the complete function to print a single concordance entry:

```
def concordanceEntry(textFile, targetWord):
    """ (docstring omitted) """

    lineNumber = 1
    for line in textFile:
        index = find(line.lower(), targetWord)
        if index >= 0:    # targetWord is found in line
            align = index + len(targetWord)
            print('{0:<6}{1:>80}{2}'.format(lineNumber, line[:align],
                                                        line[align:-1]))
        lineNumber = lineNumber + 1
```

There are many more enhancements we can make to this function, some of which we leave as exercises.

A complete concordance

Now we could, if we wanted to, print an entire concordance for a book! The `concordance` function below does just that. Because a full concordance can be very long indeed, we have added a parameter `numEntries` to limit the number of entries to print. There are also a couple of new things in this function that we encourage you to explore more on your own.

```
import textlib
import string

def concordance(fileName, numEntries):
    """Print entries in the concordance for a text file.

    Parameters:
        fileName:   name of the text file
        targetWord: number of entries to print

    Return value: None
    """

    textFile = open(fileName, 'r', encoding = 'utf-8')
    text = textFile.read()
    words = textlib.wordTokens(text) # get all the words in textFile
    vocabulary = list(set(words))    # get the set of unique words
    vocabulary.sort()                # sort the words

    count = 0
    for word in vocabulary:              # iterate over the sorted words
        if word[0] not in string.digits: # omit if starts with a digit
            textFile.seek(0)             # reset file pointer
            print('\n' + word.upper() + '\n')
            concordanceEntry(textFile, word)
            count = count + 1
        if count >= numEntries:          # break when enough entries
            break
    textFile.close()

def main():
    concordance('mobydick.txt', 10)

main()
```

Exercises

6.4.1. For each of the following `for` loops, write an equivalent loop that iterates over the indices of the string `text` instead of the characters.

(a)*
```
for character in text:
    print(character)
```

(b)*
```
newText = ''
for character in text:
    if character != ' ':
        newText = newText + character
```

(c)
```
for character in text[2:10]:
    if character >= 'a' and character <= 'z':
        print(character)
```

(d)
```
for character in text[1:-1]:
    print(text.count(character))
```

6.4.2* Show how to rewrite each of the `for` loops in the previous exercise as `while` loops that increment an index variable in each iteration.

6.4.3. Describe what is wrong with each the following blocks of code, and show how to fix it.

(a)*
```
veggie = 'carrots'
for character in veggie:
    bigVeggie = bigVeggie + character.upper()
```

(b)*
```
while answer != 'q':
    answer = input('Word? ')
    print(len(answer))
```

(c)*
```
veggie = 'peas'
for index in range(veggie):
    if veggie[index] != ' ':
        print(index)
```

(d)
```
veggie = 'sweet potatoes'
for index in len(veggie):
    if veggie[index] != ' ':
        print(veggie[index])
```

(e)
```
for index in range(len(okra)):
    print(okra[len(okra) - index - 1])
```

(f)
```
text = 'I love veggies!'
for character in range(len(text)):
    if character != ' ':
        print(character)  # print a character in text?
```

(g)
```
text = 'Veggies rule!'
for index in text:
    print(text[index])
```

(h)
```
veggies = 'pepperoni'  # not
for index in range(len(veggies)):
    if index == len(veggies) - 1:
        realVeggies = realVeggies + 'cin'
    realVeggies = realVeggies + index
print(realVeggies)
```

6.4.4. Write a function

```
prefixes(word)
```

that prints all of the prefixes of the given word. For example, `prefixes('cart')` should print

```
c
ca
car
cart
```

6.4.5. Write a function

```
replace(text, target, replacement)
```

that returns a new version of `text` in which all occurrences of the substring `target` are replaced with the string `replacement`. For example, `replace('I am cool, very cool.', 'cool', 'very vain'))` should return `'I am very vain, very very vain.'`. Use a `while` loop; do not use the string method `replace`.

6.4.6. An annoying trick used by some simple chatbots is to reply to a statement by reframing it as a question. For some statements, like those starting with "This is" and "There are," this is very easy: just reverse the first two words. For example, "This is a cool sentence." becomes "Is this a cool sentence?" Write a function

```
makeQuestion(sentence)
```

that carries out this transformation on a sentence by finding and swapping the first two words, correcting the capitalization, and ending the sentence with a question mark.

6.4.7* Suppose you develop a secret code that replaces a string with a new string that consists of all the even indexed characters of the original followed by all the odd indexed characters. For example, the string `'computers'` would be encoded as `'cmuesoptr'`. Write a function

```
encode(word)
```

that returns the encoded version of the string named `word`.

6.4.8. Write a function

```
decode(codeword)
```

that reverses the process from the `encode` function in the previous exercise.

6.4.9* Write a function

```
palindrome(text)
```

that returns `True` if `text` is a palindrome and `False` otherwise. Your function should ignore spaces and capitalization. For example,

```
palindrome('Lisa Bonet ate no basil')
```

should return `True`.

6.4.10. Write an interactive program that uses the `find` function from this section to find the first occurrence of any desired word in *Moby Dick*.

6.4.11. The `find` function from this section can also be written like this:

```
def find(text, target):
    for index in range(len(text) - len(target) + 1):
        if text[index:index + len(target)] == target:
            return index
    return -1
```

(a) Explain why this version is also correct.

(b) Why is the following version of the `for` loop incorrect?

```
for index in range(len(text) - len(target) + 1):
    if text[index:index + len(target)] == target:
        return index
    else:
        return -1
```

6.4.12. Write a modified version of the `find` function named `findWord` that only finds an instance of `target` that is a whole word.

6.4.13. Write a modified version of the `find` function named `findAll` that returns a list containing the indices of all matches for the target.

6.4.14. Enhance the `concordanceEntry` function in each of the following ways:

(a) In the line that is printed for each match, display `targetWord` in all caps. For example:

```
... any whale could so SMITE his stout sloop-of-war
...   vessel, so as to SMITE down some of the spars and
```

(b) Use the modified `findWord` function from Exercise 6.4.12 so that each target word is matched only if it is a complete word.

(c) The current version of `concordanceEntry` will only identify the first instance of a word on each line. Modify it so that it will display a new context line for every instance of the target word in every line of the text. For example, "ship" appears twice on line 14673 of *Moby Dick*:

upon the ship, than to rejoice that the ship had so victoriously gained

In this case, the function should print:

```
14673                                upon the SHIP, than...
      upon the ship, than to rejoice that the SHIP had so...
```

The `findAll` function from Exercise 6.4.13 will be useful here.

6.5 WORD FREQUENCY TRENDS

One common way to gain a little insight into the arc of a particular theme in a text is to visualize the relative frequencies of related terms over the course of the text. Figure 6.5 shows an example that visualizes the usage of masculine and feminine

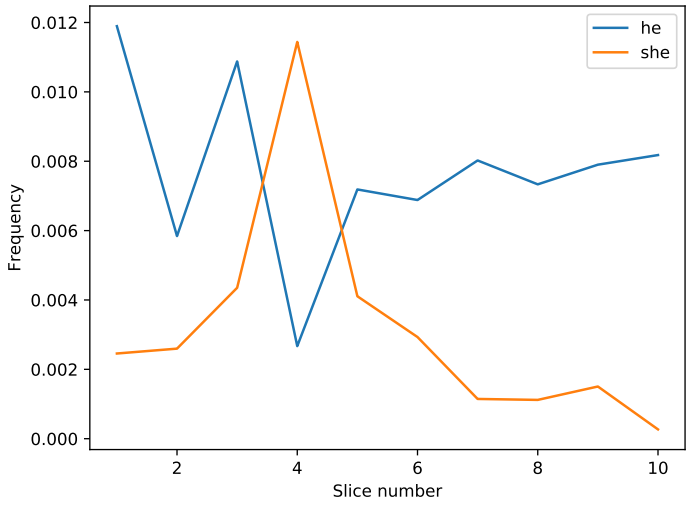
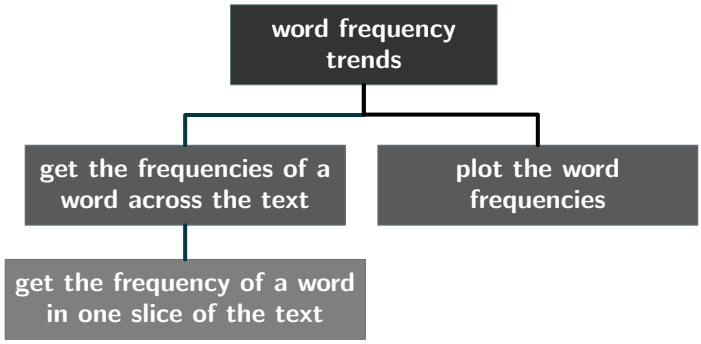


Figure 6.5 A sample plot of word frequencies across ten slices of *Frankenstein*.

pronouns over the course of *Frankenstein*. In this plot, the frequencies of the two words are shown as fractions of the total number of words in each of ten equal-sized slices of the text.

Since writing a program for this problem is a bit more involved, let's explicitly decompose it into subproblems first.



In the first subproblem, we need to compute the frequencies of a word over all of the slices of the text. As a part of this subproblem we will need a function that computes the frequency of a word in just one slice. In the second subproblem, we actually plot the frequencies computed in the first subproblem.

The first subproblem will take three inputs—the text, the desired number of slices, and one of the words—and return a list of the frequencies of the word in the slices of the text. (The main program will need to call this function twice, once for each word.)

To solve this subproblem, we will start at the bottom leaf in the decomposition tree: finding the frequency of a word in a single slice.

Finding the frequency of a word

Computing the frequency of a word in a slice of the text requires three steps. First, we get a list of all of the words in the slice using the `wordTokens` function from our `textlib` module. Second, we iterate over these words and count how many times the target word shows up. Third, we return that count divided by the number of words in the slice. Here is the algorithm in pseudocode.

Algorithm WORD FREQUENCY

Input: a *text* and a *target word*

```
word list ← WORD TOKENS (text)
count ← 0
repeat for each word in word list:
    if word = target word, then:
        count ← count + 1
frequency ← count / length of word list
```

Output: *frequency*

We already know how to perform all of these steps in Python. Iterating over a list of items looks and works just like iterating over a string, except the index variable is assigned consecutive list items instead of characters.

```
import textlib

def wordFrequency(text, targetWord):
    """Get the frequency of the target word as a fraction of all
       words in the text.

    Parameters:
        text:       a string object
        targetWord: a word to count

    Return value: frequency of the target word
    """

    wordList = textlib.wordTokens(text)
    count = 0
    for word in wordList:
        if word == targetWord:
            count = count + 1
    return count / len(wordList)
```

In each iteration of this loop, `word` is a assigned a word in `wordList`, and then `word` is compared to the target word. If they are the same, `count` is incremented. We return the final value of `count` divided by the length of `wordList`, which is the total number of words in `text`. The following trace table illustrates this loop more concretely with a fun input.

Trace arguments: `text = 'one fish two fish', targetWord = 'fish'`

Step	Line	wordList	count	word	Notes
1	10	`['one', 'fish', 'two', 'fish']`	—	—	get words from `wordTokens`
2	11	″	0	—	initialize `count` ← 0
3	12	″	″	`'one'`	`word` ← first item in `wordList`
4	13	″	″	″	`'one' != 'fish'`; skip line 14
5	12	″	″	`'fish'`	`word` ← second item in `wordList`
6	13	″	″	″	`'fish' == 'fish'`; do line 14
7	14	″	1	″	increment `count`
8	12	″	″	`'two'`	`word` ← third item in `wordList`
9	13	″	″	″	`'two' != 'fish'`; skip line 14
10	12	″	″	`'fish'`	`word` ← fourth item in `wordList`
11	13	″	″	″	`'fish' == 'fish'`; do line 14
12	14	″	2	″	increment `count`
13	15	″	″	″	return $2/4 = 0.5$

Return value: `0.5`

Getting the frequencies in slices

Now, to get the frequencies of a word across the entire text, we need to divide the text into slices and call `wordFrequency` with each slice. The algorithm will return a list of these slice frequencies.

Algorithm SLICE FREQUENCIES

Input: a *text*, a *word*, and a *number of slices*

1 *slice length* ← length of *text* / *number of slices*
2 *word frequencies* ← empty list
3 repeat for each *slice* of *slice length* characters in the *text*:
4 *frequency* ← WORD FREQUENCY (*slice*, *word*)
5 append *frequency* to *word frequencies*

Output: *word frequencies*

Implementing this in Python is not quite as straightforward as the pseudocode.

```
def sliceFrequencies(text, word, numSlices):
    """Find the frequency of the word in each slice of the text.

    Parameters:
        text:      a string containing a text
        word:      a word to analyze
        numSlices: the integer number of text slices

    Return values: list of slice frequencies
    """

    sliceLength = (len(text) // numSlices) + 1       # round up
    wordFreqs = [ ]
    for index in range(0, len(text), sliceLength):   # for each slice...
        textSlice = text[index:index + sliceLength]
        frequency = wordFrequency(textSlice, word)
        wordFreqs.append(frequency)
    return wordFreqs
```

To get the slices of the text, we iterate over the indices of `text`, skipping the length of a slice each time. So, in each iteration, the value of `index` is the beginning of a slice. Each slice is located between indices `index` and `index + sliceLength`. This slice of `text` is passed into our `wordFrequency` function along with the `word`, and the returned `frequency` is appended to the list of frequencies.

Plotting the frequencies

Finally, we combine this function with a simple function to plot the frequencies (our second subproblem) to tie it all together.

```
def plotWordFreqs(word1, word2, wordFreqs1, wordFreqs2):
    """Plot 2 lists of word frequencies.

    Parameters:
        word1, word2:           2 words being analyzed
        wordFreqs1, wordFreqs2: lists of 2 words' frequencies

    Return value: None
    """

    numSlices = len(wordFreqs1)
    pyplot.plot(range(1, numSlices + 1), wordFreqs1, label = word1)
    pyplot.plot(range(1, numSlices + 1), wordFreqs2, label = word2)
    pyplot.legend()
    pyplot.xlabel('Slice number')
    pyplot.ylabel('Frequency')
    pyplot.show()
```

```
def wordTrends(fileName, word1, word2, numSlices):
    """Plot frequencies of 2 words across a text.

    Parameters:
        fileName:      name of a text file
        word1, word2:  2 words being analyzed
        numSlices:     the integer number of text slices

    Return value: None
    """

    # open and read the file
    textFile = open(fileName, 'r', encoding = 'utf-8')
    text = textFile.read()
    textFile.close()

    wordFreqs1 = sliceFrequencies(text, word1, numSlices)
    wordFreqs2 = sliceFrequencies(text, word2, numSlices)

    plotWordFreqs(word1, word2, wordFreqs1, wordFreqs2)
```

Exercise 6.5.5 asks you to combine the four functions from this section into a program that you can use to experiment with plotting frequencies of different words in books from Project Gutenberg or the book website.

Exercises

6.5.1* Repeat Exercise 6.1.7 but use a `for` loop that iterates over the indices of the string instead of the characters.

6.5.2* Write a function

```
count(text, target)
```

that returns the number of occurrences of the string `target` in the string `text`. Use a `for` loop; do not use the string method `count`.

6.5.3. Write a function

```
countAll(text, targets)
```

that returns the number of occurrences in the string `text` of any of the strings in the list of strings named `targets`. You can use the `in` operator to determine whether an item is in a list.

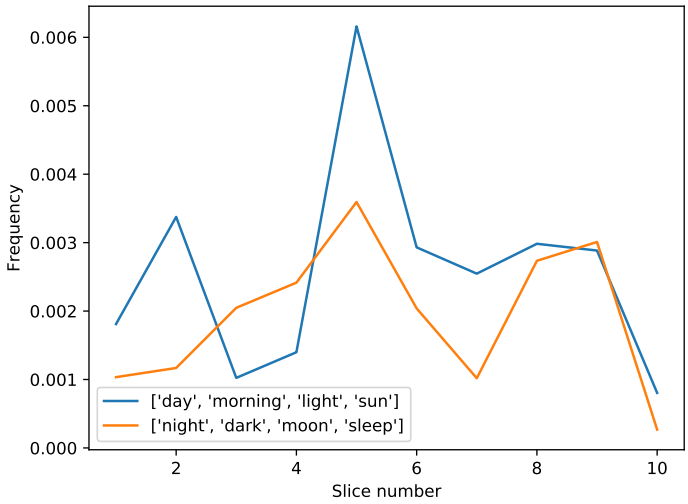
6.5.4. Draw a diagram like the one in Figure 6.2 on page 237 that shows how the inputs and outputs flow among the four functions from this section. Use the functional decomposition tree on page 267 as a starting point.

6.5.5. Combine the four functions from this section into a program that prompts for a filename, two words to analyze, and a number of slices, and then calls `wordTrends`. Experiment with plotting frequencies of different words in books from `http://www.gutenberg.org` or the book website.

6.5.6. Modify the `sliceFrequencies` function so that the slices overlap by a given amount. Substitute the `numSlices` parameter with two parameters:

`sliceLength` and `sliceStep`, the length of each slice and the amount that each slice should shift right in each step, respectively. In the words, the first slice will be from index `0` to `sliceLength`, the second slice will be from `sliceStep` to `sliceStep + sliceLength`, the third slice will be from `2 * sliceStep` to `2 * sliceStep + sliceLength`, etc.

6.5.7. Modify the word frequency trends program you wrote for Exercise 6.5.5 so that it prompts for two *lists* of words instead. Each list might represent a particular theme; an example from *Frankenstein* is shown below. The `wordFrequency` function will need some minor modifications. You can use the `in` operator to determine whether an item is in a list.



6.6 COMPARING TEXTS

There have been many methods developed to measure similarity between texts, most of which are beyond the scope of this book. But one particular method, called a *dot plot* is both accessible and quite powerful. In a dot plot, we associate one text with the x-axis of a plot and another text with the y-axis. We place a dot at position (x,y) if the character or slice of text at index x in the first text is the same as the character or slice at index y in the second text. In this way, a dot plot visually illustrates the similarity between the two texts.

Let's begin by writing an algorithm that only compares individual characters at the same indices in the two texts. Consider the following two sentences:

Text 1: `Peter Piper picked a peck of pickled peppers.`
Text 2: `Peter Pepper picked a peck of pickled capers.`

We will compare the first character in text 1 to the first character in text 2, then the second character in text 1 to the second character in text 2, etc. Although this algorithm, shown below, must iterate over both strings at the same time, and

compare the two strings at each position, it requires only one loop because we always compare the strings at the same index.

```
import matplotlib.pyplot as pyplot

def dotplot1(text1, text2):
    """Display a simplified dot plot comparing two equal-length strings.

    Parameters:
        text1: a string object
        text2: a string object

    Return value: None
    """

    text1 = text1.lower()
    text2 = text2.lower()
    x = []
    y = []
    for index in range(len(text1)):
        if text1[index] == text2[index]:
            x.append(index)
            y.append(index)
    pyplot.scatter(x, y)          # scatter plot
    pyplot.xlim(0, len(text1))    # x axis covers entire text1
    pyplot.ylim(0, len(text2))    # y axis covers entire text2
    pyplot.xlabel(text1)
    pyplot.ylabel(text2)
    pyplot.show()
```

Reflection 6.23 *What is the purpose of the calls to the* `lower` *method?*

Reflection 6.24 *Why must we iterate over the indices of the strings rather than the characters in the strings?*

Every time two characters are found to be equal in the loop, the index of the matching characters is added to both a list of x-coordinates and a list of y-coordinates. These lists are then plotted with the `scatter` function from `matplotlib.pyplot`, which plots points without lines attaching them. Figure 6.6 shows the result for the two strings above.

Reflection 6.25 *Look at Figure 6.6. Which dots correspond to which characters? Why are there only dots on the diagonal?*

We can see that, because this function only recognizes matches at the same index and most of the identical characters in the two sentences do not line up perfectly, this function does not reveal their true degree of similarity. If we were to insert two gaps into the strings, the character-by-character comparison would be quite different:

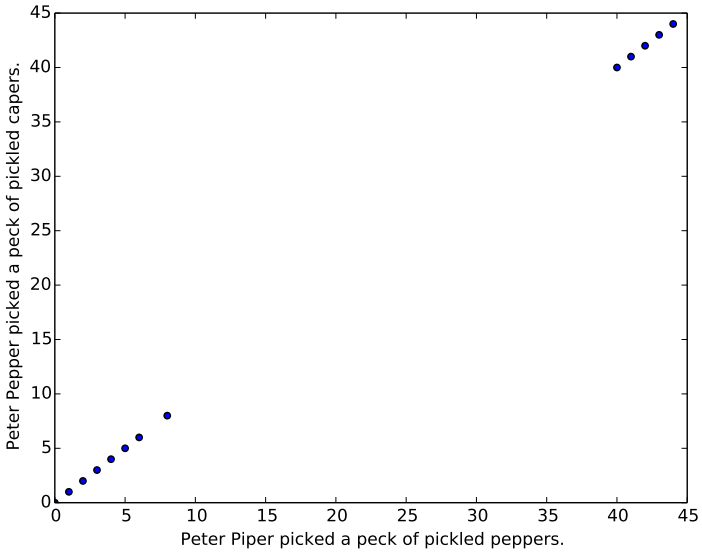


Figure 6.6 Output from the `dotplot1` function.

```
Text 1: Peter Pip er picked a peck of pickled peppers.
Text 2: Peter Pepper picked a peck of pickled ca pers.
```

Dot plots

A real dot plot compares every character in one sequence to every character in the other sequence. This means that we want to compare `text1[0]` to `text2[0]`, then `text1[0]` to `text2[1]`, then `text1[0]` to `text2[2]`, etc., as illustrated below:

```
          0 1 2 3 4 5 6 7 8 9 ...
text1: Peter Piper picked a peck of pickled peppers.
          ...
text2: Peter Pepper picked a peck of pickled capers.
          0 1 2 3 4 5 6 7 8 9 ...
```

After we have compared `text1[0]` to all of the characters in `text2`, we need to repeat this process with `text1[1]`, comparing `text1[1]` to `text2[0]`, then `text1[1]` to `text2[1]`, then `text1[1]` to `text2[2]`, etc., as illustrated below:

```
          0 1 2 3 4 5 6 7 8 9 ...
text1: Peter Piper picked a peck of pickled peppers.
          ...
text2: Peter Pepper picked a peck of pickled capers.
          0 1 2 3 4 5 6 7 8 9 ...
```

In other words, for each value of `index`, we want to compare `text1[index]` to

every character in `text2`, not just to `text2[index]`. To accomplish this, we need to replace the `if` statement in `dotplot1` with another `for` loop:

```
import matplotlib.pyplot as pyplot

def dotplot(text1, text2):
    """Display a dot plot comparing two strings.

    Parameters:
        text1: a string object
        text2: a string object

    Return value: None
    """

    text1 = text1.lower()
    text2 = text2.lower()
    x = []
    y = []
    for index1 in range(len(text1)):
        for index2 in range(len(text2)):
            if text1[index1] == text2[index2]:
                x.append(index1)
                y.append(index2)
    pyplot.scatter(x, y)
    pyplot.xlim(0, len(text1))
    pyplot.ylim(0, len(text2))
    pyplot.xlabel(text1)
    pyplot.ylabel(text2)
    pyplot.show()
```

With this change inside the first `for` loop (we also renamed `index` to `index1`), each character `text1[index1]` is compared to every character in `text2`, indexed by the index variable `index2`, just like the illustrations above. If a match is found, we append `index1` to the `x` list and `index2` to the `y` list because we want to draw a dot at coordinates `(index1, index2)`.

The following trace table shows how this works in more detail, with much smaller inputs. The iterations of the outer `for` loop are separated by thicker black lines and set apart with the curly braces on the left side, annotated with values of `index1`. The iterations of the inner `for` loop are separated by thinner red lines.

	Trace arguments: `text1 = 'spam', text2 = 'pea'`					
	Step	Line	x	y	index1	index2	Notes
	1–4	9–12	`[ ]`	`[ ]`	—	—	initialize variables
0	5	13	″	″	0	—	`index1` ← 0
	6	14	″	″	″	0	`index2` ← 0
	7	15	″	″	″	″	`'s' != 'p'`; skip lines 16–17
	8	14	″	″	″	1	`index2` ← 1
	9	15	″	″	″	″	`'s' != 'e'`; skip lines 16–17
	10	14	″	″	″	2	`index2` ← 2
	11	15	″	″	″	″	`'s' != 'a'`; skip lines 16–17
1	12	13	″	″	1	″	`index1` ← 1
	13	14	″	″	″	0	`index2` ← 0
	14	15	″	″	″	″	`'p' == 'p'`; do lines 16–17
	15–16	16–17	`[1]`	`[0]`	″	″	we want a dot at (1,0)
	17	14	″	″	″	1	`index2` ← 1
	18	15	″	″	″	″	`'p' != 'e'`; skip lines 16–17
	19	14	″	″	″	2	`index2` ← 2
	20	15	″	″	″	″	`'p' != 'a'`; skip lines 16–17
2	21	13	″	″	2	″	`index1` ← 2
	22	14	″	″	″	0	`index2` ← 0
	23	15	″	″	″	″	`'a' != 'p'`; skip lines 16–17
	24	14	″	″	″	1	`index2` ← 1
	25	15	″	″	″	″	`'a' != 'e'`; skip lines 16–17
	26	14	″	″	″	2	`index2` ← 2
	27	15	″	″	″	″	`'a' == 'a'`; do lines 16–17
	28–29	16–17	`[1, 2]`	`[0, 2]`	″	″	we want a dot at (2,2)
3	30	13	″	″	3	″	`index1` ← 3
	31	14	″	″	″	0	`index2` ← 0
	⋮						⋮
	37–42	18–23	`[1, 2]`	`[0, 2]`	3	2	draw the plot

Notice that when `index1` is `0`, the inner `for` loop runs through all the values of `index2`. The inner loop finishes in step 11, also finishing the body of the outer `for` loop. Therefore, in step 12, the outer loop begins its second iteration, with `index1 = 1`. The inner loop then runs through all of its values again, and so on. There are `len(text1)` · `len(text2)` = 4 · 3 = 12 total comparisons because each of the four characters in `'spam'` is compared to each of the three characters in `'pea'`. These kinds of loops are called ***nested loops***. They will become very important in the next few chapters.

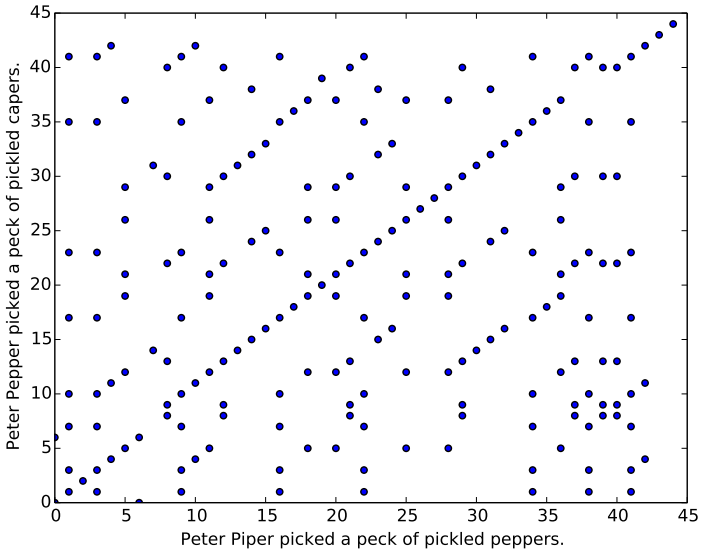


Figure 6.7 Output from the revised `dotplot` function.

Figure 6.7 shows the dot plot from the revised version of the function with the longer strings. Because the two strings share many characters, there are quite a few matches, contributing to a "noisy" plot. But the plot now does pick up the similarity in the strings, illustrated by the dots along the main diagonal.

We can reduce the "noise" in a dot plot by comparing substrings instead of individual characters. In textual analysis applications and computational linguistics, substrings with length n are known as n-grams.[6] When $n = 2$ and $n = 3$, they are also called bigrams and trigrams, respectively. When $n > 1$, there are many more possible substrings, so fewer matches tend to exist. Exercise 6.6.14 asks you to generalize this dot plot function so that it compares n-grams instead of single characters. Figure 6.8 shows the result of this function with $n = 3$.

Dot plots can be helpful in detecting potential plagiarism. Consider the controversy that erupted at the 2016 Republican National Convention, when portions of Melania Trump's speech seemed to closely resemble portions of Michelle Obama's convention speech from eight years prior. The offending portions of these two speeches are compared with a dot plot in Figure 6.9.

Reflection 6.26 *Just by looking at Figure 6.9, would you conclude that portions had been plagiarized? (Think about what a dot plot comparing two random passages would look like.)*

[6] n-grams can also refer to sequences of n words, as we will see in the next chapter.

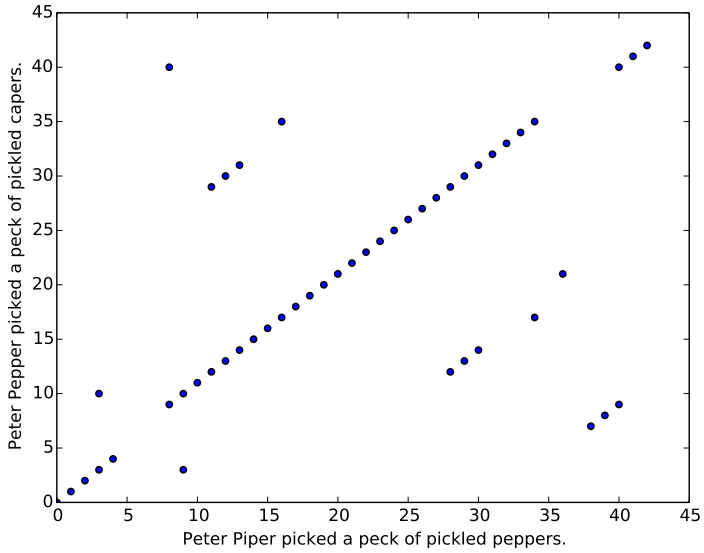


Figure 6.8 Output from the `dotplot` function from Exercise 6.6.14 (trigrams).

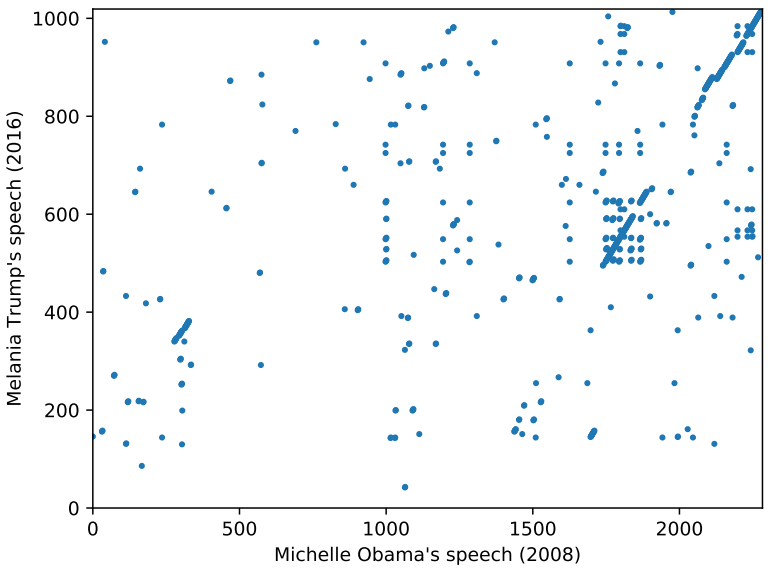


Figure 6.9 A dot plot comparing 6-grams from segments of Michelle Obama's and Melania Trump's convention speeches in 2008 and 2016.

Exercises

6.6.1. What is printed by the following loop? Explain why.

```
text1 = 'tbontb'
text2 = 'oerooe'
for index in range(len(text1)):
    print(text1[index] + text2[index])
```

6.6.2* Consider the following nested loop.

```
text1 = 'abcd'
text2 = 'xyz'
for index1 in range(len(text1)):
    for index2 in range(len(text2)):
        if text1[index1] == text2[index2]:
            print(index1, index2)
```

List the pairs of characters that are compared by the `if` statement, in the order they are compared. How many comparisons are there in total?

6.6.3. Repeat the previous exercise, but with the `for` statements swapped, as shown below.

```
text1 = 'abcd'
text2 = 'xyz'
for index2 in range(len(text2)):
    for index1 in range(len(text1)):
        if text1[index1] == text2[index2]:
            print(index1, index2)
```

6.6.4. What is printed by the following nested loop? Explain why.

```
text = 'imho'
for index1 in range(len(text)):
    for index2 in range(index1, len(text)):
        print(text[index1:index2 + 1])
```

6.6.5. Write a function

```
difference(word1, word2)
```

that returns the first index at which the strings `word1` and `word2` differ. If the words have different lengths, and the shorter word is a prefix of the longer word, the function should return the length of the shorter word. If the two words are the same, the function should return −1. Do this without directly testing whether `word1` and `word2` are equal.

6.6.6* Hamming distance, defined to be the number of bit positions that are different between two bit strings, is used to measure the error that is introduced when data is sent over a network. For example, suppose we sent the bit sequence `011100110001` over a network, but the destination received `011000110101` instead. To measure the transmission error, we can find the Hamming distance between the two sequences by lining them up as follows:

```
Sent:     011100110001
Received: 011000110101
```

Since the bit sequences are different in two positions, the Hamming distance is 2. Write a function

```
hamming(bits1, bits2)
```

that returns the Hamming distance between the two given bit strings. Assume that the two strings have the same length.

6.6.7. Repeat Exercise 6.6.6, but make it work correctly even if the two strings have different lengths. In this case, each "missing" bit at the end of the shorter string counts as one toward the Hamming distance. For example, `hamming('000', '10011')` should return 3.

6.6.8* The following nested `for` loop is intended to print information about characters that are repeated in the string `text`.

```
text = 'two words'
for index1 in range(len(text)):
    for index2 in range(len(text)):
        if text[index1] == text[index2]:
            print(text[index1], index1, index2)
```

There are only two repeated characters in this short string—`'w'` at indices 1 and 4 and `'o'` at indices 2 and 5—but this loop prints more than that. Fix the nested loop so that it correctly prints only two lines. Explain why your solution is correct.

6.6.9. Write a function

```
longestRun(text)
```

that returns the length of the longest run of the same character in the string `text`. For example, `longestRun('aabbbbcccd')` should return 4.

6.6.10. Write a function

```
uniqueCharacters(text)
```

that returns the number of characters that only appear once in the string `text`.

6.6.11* Write a function

```
findRepeats(text, length)
```

that locates all words with the given `length` that are repeated consecutively in the string `text`. For each found repeat, your function should print the word and the index of the first occurrence of the word. Assume that words do not start with a space and that there will be a space between repeats. For example, `findRepeats('the the repeats are are repeating', 3)` should print

```
0 the
16 are
```

6.6.12. Write a function

```
findRepeats(text)
```

that locates words with any length from 2–10 that are repeated consecutively in the string `text`. See the previous problem for assumptions and what to print.

6.6.13. Write a function

```
findDuplicates(text, length)
```

that locates all pairs of identical substrings with the given `length` anywhere in the string `text`. The function should print the substring and a pair of indices representing the first index in each match. For example, `findDuplicates('the the repeats are are repeating', 4)` should print

```
the  0 4
e re 6 22
 rep 7 23
repe 8 24
epea 9 25
peat 10 26
 are 15 19
are  16 20
```

6.6.14. Generalize the `dotplot` function so that it compares n-grams instead of individual characters. The third parameter of the function should be n.

*6.7 TIME COMPLEXITY

This section is available on the book website.

*6.8 COMPUTATIONAL GENOMICS

This section is available on the book website.

6.9 SUMMARY AND FURTHER DISCOVERY

Text is stored as a sequence of bytes, which we can read into one or more strings. The most fundamental string algorithms have one of the following structures:

```
for character in text:
    # process character

for index in range(len(text)):
    # process text[index]
```

In the first case, consecutive characters in the string are assigned to the `for` loop index variable `character`. In the body of the loop, each character can then be examined individually. In the second case, consecutive integers from the list `[0, 1, 2, ..., len(text) - 1]`, which are precisely the indices of the characters in `text`, are assigned to the `for` loop index variable `index`. In this case, the algorithm has more information because, not only can it access the character at `text[index]`, it also knows where that character resides in the string. The first choice tends to be more elegant, but the second choice is necessary when the algorithm needs to know the index of each character, or if it needs to process slices of the string, which can only be accessed with indices.

We called one special case of these loops a ***string accumulator***:

```
newText = ''
for character in text:
    newText = newText + _______
```

Like an integer accumulator and a list accumulator, a string accumulator builds its result cumulatively in each iteration of the loop. Because strings are immutable, a

string accumulator must create a new string in each iteration that is composed of the old string with a new character concatenated.

Algorithms like these that perform one pass over their string parameters and execute a constant number of elementary steps per character are called *linear-time algorithms* because their number of elementary steps is proportional to the length of the input string.

In some cases, we need to compare every character in one string to every character in a second string, so we need a nested loop like the following:

```
for index1 in range(len(text1)):
    for index2 in range(len(text2)):
        # process text1[index1] and text2[index2]
```

If both strings have length n, then a nested loop like this constitutes a *quadratic-time algorithm* with time complexity $\mathcal{O}(n^2)$ (as long as the body of the loop is constant-time) because every one of n characters in the first string is compared to every one of n characters in the second string. We will see more loops like this in later chapters.

Notes for further discovery

The first of the two epigraphs at the beginning of this chapter is from the following blog post by Leslie Johnston, the former Chief of the Repository Development Center at the Library of Congress. She is currently Director of Digital Preservation at The National Archives.

`http://blogs.loc.gov/digitalpreservation/2012/04/a-library-of-congress-worth-of-data-its-all-in-how-you-define-it/`

The second epigraph is from an article titled "The DNA Data Deluge" by Michael C. Schatz and Ben Langmead [55], which can be found at

`http://spectrum.ieee.org/biomedical/devices/the-dna-data-deluge` .

To learn more about how tokenization is performed in the Python interpreter, look here: `https://docs.python.org/3/reference/lexical_analysis.html`.

For the complete Unicode character set, refer to `http://unicode.org`.

The Fletcher checksum algorithm was invented by John Fletcher at the Lawrence Livermore National Laboratory and published in 1982 [18].

A concordance for the works of William Shakespeare can be found at

`http://www.opensourceshakespeare.org/concordance/`.

The Keyword in Context (KWIC) indexing system, also known as a permuted index, is similar to a concordance. In a KWIC index, every word in the title of an article appears in the index in the context in which it appears.

To learn more about text analysis in the digital humanities, we recommend *Macro-*

analysis [26] by Matthew Jockers and *Exploring Big Historical Data* [20] by Shawn Graham, Ian Milligan, and Scott Weingart.

If you are interested in learning more about computational biology, two good places to start are *The Mathematics of Life* [62] by Ian Stewart and *Natural Computing* [60] by Dennis Shasha and Cathy Lazere. The latter book has a wider focus than just computational biology.

*6.10 PROJECTS

This section is available on the book website.