CHAPTER 1

Language Processing and Python

It is easy to get our hands on millions of words of text. What can we do with it, assuming we can write some simple programs? In this chapter, we'll address the following questions:

1. What can we achieve by combining simple programming techniques with large quantities of text?
2. How can we automatically extract key words and phrases that sum up the style and content of a text?
3. What tools and techniques does the Python programming language provide for such work?
4. What are some of the interesting challenges of natural language processing?

This chapter is divided into sections that skip between two quite different styles. In the "computing with language" sections, we will take on some linguistically motivated programming tasks without necessarily explaining how they work. In the "closer look at Python" sections we will systematically review key programming concepts. We'll flag the two styles in the section titles, but later chapters will mix both styles without being so up-front about it. We hope this style of introduction gives you an authentic taste of what will come later, while covering a range of elementary concepts in linguistics and computer science. If you have basic familiarity with both areas, you can skip to Section 1.5; we will repeat any important points in later chapters, and if you miss anything you can easily consult the online reference material at *http://www.nltk.org/*. If the material is completely new to you, this chapter will raise more questions than it answers, questions that are addressed in the rest of this book.

1.1 Computing with Language: Texts and Words

We're all very familiar with text, since we read and write it every day. Here we will treat text as *raw data* for the programs we write, programs that manipulate and analyze it in a variety of interesting ways. But before we can do this, we have to get started with the Python interpreter.

Getting Started with Python

One of the friendly things about Python is that it allows you to type directly into the interactive **interpreter**—the program that will be running your Python programs. You can access the Python interpreter using a simple graphical interface called the Interactive DeveLopment Environment (IDLE). On a Mac you can find this under Applications→MacPython, and on Windows under All Programs→Python. Under Unix you can run Python from the shell by typing `idle` (if this is not installed, try typing `python`). The interpreter will print a blurb about your Python version; simply check that you are running Python 2.4 or 2.5 (here it is 2.5.1):

```
Python 2.5.1 (r251:54863, Apr 15 2008, 22:57:26)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you are unable to run the Python interpreter, you probably don't have Python installed correctly. Please visit *http://python.org/* for detailed instructions.

The `>>>` prompt indicates that the Python interpreter is now waiting for input. When copying examples from this book, don't type the "`>>>`" yourself. Now, let's begin by using Python as a calculator:

```
>>> 1 + 5 * 2 - 3
8
>>>
```

Once the interpreter has finished calculating the answer and displaying it, the prompt reappears. This means the Python interpreter is waiting for another instruction.

Your Turn: Enter a few more expressions of your own. You can use asterisk (*) for multiplication and slash (/) for division, and parentheses for bracketing expressions. Note that division doesn't always behave as you might expect—it does integer division (with rounding of fractions downwards) when you type `1/3` and "floating-point" (or decimal) division when you type `1.0/3.0`. In order to get the expected behavior of division (standard in Python 3.0), you need to type: `from __future__ import division`.

The preceding examples demonstrate how you can work interactively with the Python interpreter, experimenting with various expressions in the language to see what they do. Now let's try a non-sensical expression to see how the interpreter handles it:

```
>>> 1 +
  File "<stdin>", line 1
    1 +
      ^
SyntaxError: invalid syntax
>>>
```

This produced a **syntax error**. In Python, it doesn't make sense to end an instruction with a plus sign. The Python interpreter indicates the line where the problem occurred (line 1 of `<stdin>`, which stands for "standard input").

Now that we can use the Python interpreter, we're ready to start working with language data.

Getting Started with NLTK

Before going further you should install NLTK, downloadable for free from *http://www.nltk.org/*. Follow the instructions there to download the version required for your platform.

Once you've installed NLTK, start up the Python interpreter as before, and install the data required for the book by typing the following two commands at the Python prompt, then selecting the `book` collection as shown in Figure 1-1.

```
>>> import nltk
>>> nltk.download()
```

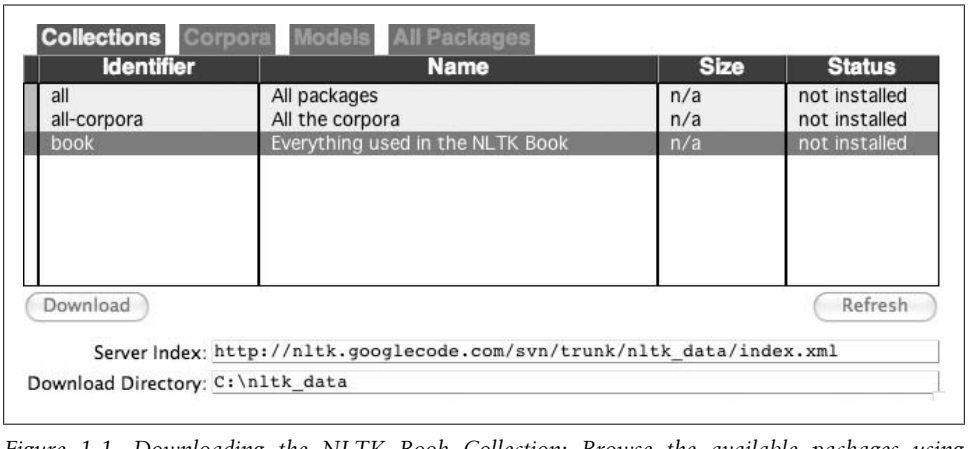


Figure 1-1. Downloading the NLTK Book Collection: Browse the available packages using `nltk.download()`. *The* ***Collections*** *tab on the downloader shows how the packages are grouped into sets, and you should select the line labeled* ***book*** *to obtain all data required for the examples and exercises in this book. It consists of about 30 compressed files requiring about 100Mb disk space. The full collection of data (i.e.,* ***all*** *in the downloader) is about five times this size (at the time of writing) and continues to expand.*

Once the data is downloaded to your machine, you can load some of it using the Python interpreter. The first step is to type a special command at the Python prompt, which

tells the interpreter to load some texts for us to explore: `from nltk.book import *`. This says "from NLTK's `book` module, load all items." The `book` module contains all the data you will need as you read this chapter. After printing a welcome message, it loads the text of several books (this will take a few seconds). Here's the command again, together with the output that you will see. Take care to get spelling and punctuation right, and remember that you don't type the `>>>`.

```
>>> from nltk.book import *
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
>>>
```

Any time we want to find out about these texts, we just have to enter their names at the Python prompt:

```
>>> text1
<Text: Moby Dick by Herman Melville 1851>
>>> text2
<Text: Sense and Sensibility by Jane Austen 1811>
>>>
```

Now that we can use the Python interpreter, and have some data to work with, we're ready to get started.

Searching Text

There are many ways to examine the context of a text apart from simply reading it. A concordance view shows us every occurrence of a given word, together with some context. Here we look up the word *monstrous* in *Moby Dick* by entering `text1` followed by a period, then the term `concordance`, and then placing `"monstrous"` in parentheses:

```
>>> text1.concordance("monstrous")
Building index...
Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came towards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r
ll over with a heathenish array of monstrous clubs and spears . Some were thick
d as you gazed , and wondered what monstrous cannibal and savage could ever hav
that has survived the flood ; most monstrous and most mountainous ! That Himmal
they might scout at Moby Dick as a monstrous fable , or still worse and more de
th of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall ere l
ing Scenes . In connexion with the monstrous pictures of whales , I am strongly
ere to enter upon those still more monstrous stories of them which are to be fo
```

```
ght have been rummaged out of this monstrous cabinet there is no telling . But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up dead u
>>>
```

Your Turn: Try searching for other words; to save re-typing, you might be able to use up-arrow, Ctrl-up-arrow, or Alt-p to access the previous command and modify the word being searched. You can also try searches on some of the other texts we have included. For example, search *Sense and Sensibility* for the word *affection*, using `text2.concordance("affection")`. Search the book of Genesis to find out how long some people lived, using: `text3.concordance("lived")`. You could look at `text4`, the *Inaugural Address Corpus*, to see examples of English going back to 1789, and search for words like *nation*, *terror*, *god* to see how these words have been used differently over time. We've also included `text5`, the *NPS Chat Corpus*: search this for unconventional words like *im*, *ur*, *lol*. (Note that this corpus is uncensored!)

Once you've spent a little while examining these texts, we hope you have a new sense of the richness and diversity of language. In the next chapter you will learn how to access a broader range of text, including text in languages other than English.

A concordance permits us to see words in context. For example, we saw that *monstrous* occurred in contexts such as *the ___ pictures* and *the ___ size*. What other words appear in a similar range of contexts? We can find out by appending the term `similar` to the name of the text in question, then inserting the relevant word in parentheses:

```
>>> text1.similar("monstrous")
Building word-context index...
subtly impalpable pitiable curious imperial perilous trustworthy
abundant untoward singular lamentable few maddens horrible loving lazy
mystifying christian exasperate puzzled
>>> text2.similar("monstrous")
Building word-context index...
very exceedingly so heartily a great good amazingly as sweet
remarkably extremely vast
>>>
```

Observe that we get different results for different texts. Austen uses this word quite differently from Melville; for her, *monstrous* has positive connotations, and sometimes functions as an intensifier like the word *very*.

The term `common_contexts` allows us to examine just the contexts that are shared by two or more words, such as *monstrous* and *very*. We have to enclose these words by square brackets as well as parentheses, and separate them with a comma:

```
>>> text2.common_contexts(["monstrous", "very"])
be_glad am_glad a_pretty is_pretty a_lucky
>>>
```

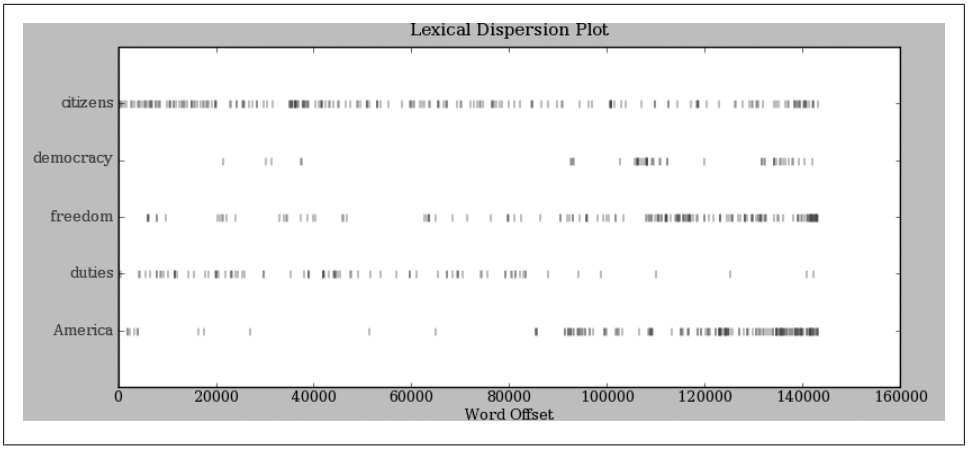


Figure 1-2. Lexical dispersion plot for words in U.S. Presidential Inaugural Addresses: This can be used to investigate changes in language use over time.

Your Turn: Pick another pair of words and compare their usage in two different texts, using the `similar()` and `common_contexts()` functions.

It is one thing to automatically detect that a particular word occurs in a text, and to display some words that appear in the same context. However, we can also determine the *location* of a word in the text: how many words from the beginning it appears. This positional information can be displayed using a **dispersion plot**. Each stripe represents an instance of a word, and each row represents the entire text. In Figure 1-2 we see some striking patterns of word usage over the last 220 years (in an artificial text constructed by joining the texts of the Inaugural Address Corpus end-to-end). You can produce this plot as shown below. You might like to try more words (e.g., *liberty*, *constitution*) and different texts. Can you predict the dispersion of a word before you view it? As before, take care to get the quotes, commas, brackets, and parentheses exactly right.

```
>>> text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])
>>>
```

Important: You need to have Python's NumPy and Matplotlib packages installed in order to produce the graphical plots used in this book. Please see *http://www.nltk.org/* for installation instructions.

Now, just for fun, let's try generating some random text in the various styles we have just seen. To do this, we type the name of the text followed by the term `generate`. (We need to include the parentheses, but there's nothing that goes between them.)

```
>>> text3.generate()
In the beginning of his brother is a hairy man , whose top may reach
unto heaven ; and ye shall sow the land of Egypt there was no bread in
all that he was taken out of the month , upon the earth . So shall thy
wages be ? And they made their father ; and Isaac was old , and kissed
him : and Laban with his cattle in the midst of the hands of Esau thy
first born , and Phichol the chief butler unto his son Isaac , she
>>>
```

Note that the first time you run this command, it is slow because it gathers statistics about word sequences. Each time you run it, you will get different output text. Now try generating random text in the style of an inaugural address or an Internet chat room. Although the text is random, it reuses common words and phrases from the source text and gives us a sense of its style and content. (What is lacking in this randomly generated text?)

When `generate` produces its output, punctuation is split off from the preceding word. While this is not correct formatting for English text, we do it to make clear that words and punctuation are independent of one another. You will learn more about this in Chapter 3.

Counting Vocabulary

The most obvious fact about texts that emerges from the preceding examples is that they differ in the vocabulary they use. In this section, we will see how to use the computer to count the words in a text in a variety of useful ways. As before, you will jump right in and experiment with the Python interpreter, even though you may not have studied Python systematically yet. Test your understanding by modifying the examples, and trying the exercises at the end of the chapter.

Let's begin by finding out the length of a text from start to finish, in terms of the words and punctuation symbols that appear. We use the term `len` to get the length of something, which we'll apply here to the book of Genesis:

```
>>> len(text3)
44764
>>>
```

So Genesis has 44,764 words and punctuation symbols, or "tokens." A **token** is the technical name for a sequence of characters—such as `hairy`, `his`, or `:)`—that we want to treat as a group. When we count the number of tokens in a text, say, the phrase *to be or not to be*, we are counting occurrences of these sequences. Thus, in our example phrase there are two occurrences of *to*, two of *be*, and one each of *or* and *not*. But there are only four distinct vocabulary items in this phrase. How many distinct words does the book of Genesis contain? To work this out in Python, we have to pose the question slightly differently. The vocabulary of a text is just the *set* of tokens that it uses, since in a set, all duplicates are collapsed together. In Python we can obtain the vocabulary

items of `text3` with the command: `set(text3)`. When you do this, many screens of words will fly past. Now try the following:

```
>>> sorted(set(text3)) ❶
['!', "'", '(', ')', ',', ',)', '.', '.)', ':', ';', ';)', '?', '?)',
'A', 'Abel', 'Abelmizraim', 'Abidah', 'Abide', 'Abimael', 'Abimelech',
'Abr', 'Abrah', 'Abraham', 'Abram', 'Accad', 'Achbor', 'Adah', ...]
>>> len(set(text3)) ❷
2789
>>>
```

By wrapping `sorted()` around the Python expression `set(text3)` ❶, we obtain a sorted list of vocabulary items, beginning with various punctuation symbols and continuing with words starting with *A*. All capitalized words precede lowercase words. We discover the size of the vocabulary indirectly, by asking for the number of items in the set, and again we can use `len` to obtain this number ❷. Although it has 44,764 tokens, this book has only 2,789 distinct words, or "word types." A **word type** is the form or spelling of the word independently of its specific occurrences in a text—that is, the word considered as a unique item of vocabulary. Our count of 2,789 items will include punctuation symbols, so we will generally call these unique items **types** instead of word types.

Now, let's calculate a measure of the lexical richness of the text. The next example shows us that each word is used 16 times on average (we need to make sure Python uses floating-point division):

```
>>> from __future__ import division
>>> len(text3) / len(set(text3))
16.050197203298673
>>>
```

Next, let's focus on particular words. We can count how often a word occurs in a text, and compute what percentage of the text is taken up by a specific word:

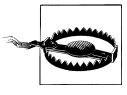
```
>>> text3.count("smote")
5
>>> 100 * text4.count('a') / len(text4)
1.4643016433938312
>>>
```

Your Turn: How many times does the word *lol* appear in `text5`? How much is this as a percentage of the total number of words in this text?

You may want to repeat such calculations on several texts, but it is tedious to keep retyping the formula. Instead, you can come up with your own name for a task, like "lexical_diversity" or "percentage", and associate it with a block of code. Now you only have to type a short name instead of one or more complete lines of Python code, and you can reuse it as often as you like. The block of code that does a task for us is

called a **function**, and we define a short name for our function with the keyword `def`. The next example shows how to define two new functions, `lexical_diversity()` and `percentage()`:

```
>>> def lexical_diversity(text): ❶
...     return len(text) / len(set(text)) ❷
...
>>> def percentage(count, total): ❸
...     return 100 * count / total
...
```



Caution!

The Python interpreter changes the prompt from `>>>` to `...` after encountering the colon at the end of the first line. The `...` prompt indicates that Python expects an **indented code block** to appear next. It is up to you to do the indentation, by typing four spaces or hitting the Tab key. To finish the indented block, just enter a blank line.

In the definition of `lexical diversity()` ❶, we specify a **parameter** labeled `text`. This parameter is a "placeholder" for the actual text whose lexical diversity we want to compute, and reoccurs in the block of code that will run when the function is used, in line ❷. Similarly, `percentage()` is defined to take two parameters, labeled `count` and `total` ❸.

Once Python knows that `lexical_diversity()` and `percentage()` are the names for specific blocks of code, we can go ahead and use these functions:

```
>>> lexical_diversity(text3)
16.050197203298673
>>> lexical_diversity(text5)
7.4200461589185629
>>> percentage(4, 5)
80.0
>>> percentage(text4.count('a'), len(text4))
1.4643016433938312
>>>
```

To recap, we use or **call** a function such as `lexical_diversity()` by typing its name, followed by an open parenthesis, the name of the text, and then a close parenthesis. These parentheses will show up often; their role is to separate the name of a task—such as `lexical_diversity()`—from the data that the task is to be performed on—such as `text3`. The data value that we place in the parentheses when we call a function is an **argument** to the function.

You have already encountered several functions in this chapter, such as `len()`, `set()`, and `sorted()`. By convention, we will always add an empty pair of parentheses after a function name, as in `len()`, just to make clear that what we are talking about is a function rather than some other kind of Python expression. Functions are an important concept in programming, and we only mention them at the outset to give newcomers

a sense of the power and creativity of programming. Don't worry if you find it a bit confusing right now.

Later we'll see how to use functions when tabulating data, as in Table 1-1. Each row of the table will involve the same computation but with different data, and we'll do this repetitive work using a function.

Table 1-1. Lexical diversity of various genres in the Brown Corpus

Genre	Tokens	Types	Lexical diversity
skill and hobbies	82345	11935	6.9
humor	21695	5017	4.3
fiction: science	14470	3233	4.5
press: reportage	100554	14394	7.0
fiction: romance	70022	8452	8.3
religion	39399	6373	6.2

1.2 A Closer Look at Python: Texts as Lists of Words

You've seen some important elements of the Python programming language. Let's take a few moments to review them systematically.

Lists

What is a text? At one level, it is a sequence of symbols on a page such as this one. At another level, it is a sequence of chapters, made up of a sequence of sections, where each section is a sequence of paragraphs, and so on. However, for our purposes, we will think of a text as nothing more than a sequence of words and punctuation. Here's how we represent text in Python, in this case the opening sentence of *Moby Dick*:

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>>
```

After the prompt we've given a name we made up, `sent1`, followed by the equals sign, and then some quoted words, separated with commas, and surrounded with brackets. This bracketed material is known as a **list** in Python: it is how we store a text. We can inspect it by typing the name ❶. We can ask for its length ❷. We can even apply our own `lexical_diversity()` function to it ❸.

```
>>> sent1 ❶
['Call', 'me', 'Ishmael', '.']
>>> len(sent1) ❷
4
>>> lexical_diversity(sent1) ❸
1.0
>>>
```

Some more lists have been defined for you, one for the opening sentence of each of our texts, `sent2` ... `sent9`. We inspect two of them here; you can see the rest for yourself using the Python interpreter (if you get an error saying that `sent2` is not defined, you need to first type `from nltk.book import *`).

```
>>> sent2
['The', 'family', 'of', 'Dashwood', 'had', 'long',
'been', 'settled', 'in', 'Sussex', '.']
>>> sent3
['In', 'the', 'beginning', 'God', 'created', 'the',
'heaven', 'and', 'the', 'earth', '.']
>>>
```

Your Turn: Make up a few sentences of your own, by typing a name, equals sign, and a list of words, like this: `ex1 = ['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']`. Repeat some of the other Python operations we saw earlier in Section 1.1, e.g., `sorted(ex1)`, `len(set(ex1))`, `ex1.count('the')`.

A pleasant surprise is that we can use Python's addition operator on lists. Adding two lists ❶ creates a new list with everything from the first list, followed by everything from the second list:

```
>>> ['Monty', 'Python'] + ['and', 'the', 'Holy', 'Grail'] ❶
['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']
```

This special use of the addition operation is called **concatenation**; it combines the lists together into a single list. We can concatenate sentences to build up a text.

We don't have to literally type the lists either; we can use short names that refer to predefined lists.

```
>>> sent4 + sent1
['Fellow', '-', 'Citizens', 'of', 'the', 'Senate', 'and', 'of', 'the',
'House', 'of', 'Representatives', ':', 'Call', 'me', 'Ishmael', '.']
>>>
```

What if we want to add a single item to a list? This is known as **appending**. When we `append()` to a list, the list itself is updated as a result of the operation.

```
>>> sent1.append("Some")
>>> sent1
['Call', 'me', 'Ishmael', '.', 'Some']
>>>
```

Indexing Lists

As we have seen, a text in Python is a list of words, represented using a combination of brackets and quotes. Just as with an ordinary page of text, we can count up the total number of words in `text1` with `len(text1)`, and count the occurrences in a text of a particular word—say, *heaven*—using `text1.count('heaven')`.

With some patience, we can pick out the 1st, 173rd, or even 14,278th word in a printed text. Analogously, we can identify the elements of a Python list by their order of occurrence in the list. The number that represents this position is the item's **index**. We instruct Python to show us the item that occurs at an index such as `173` in a text by writing the name of the text followed by the index inside square brackets:

```
>>> text4[173]
'awaken'
>>>
```

We can do the converse; given a word, find the index of when it first occurs:

```
>>> text4.index('awaken')
173
>>>
```

Indexes are a common way to access the words of a text, or, more generally, the elements of any list. Python permits us to access sublists as well, extracting manageable pieces of language from large texts, a technique known as **slicing**.

```
>>> text5[16715:16735]
['U86', 'thats', 'why', 'something', 'like', 'gamefly', 'is', 'so', 'good',
'because', 'you', 'can', 'actually', 'play', 'a', 'full', 'game', 'without',
'buying', 'it']
>>> text6[1600:1625]
['We', "'", 're', 'an', 'anarcho', '-', 'syndicalist', 'commune', '.', 'We',
'take', 'it', 'in', 'turns', 'to', 'act', 'as', 'a', 'sort', 'of', 'executive',
'officer', 'for', 'the', 'week']
>>>
```

Indexes have some subtleties, and we'll explore these with the help of an artificial sentence:

```
>>> sent = ['word1', 'word2', 'word3', 'word4', 'word5',
...         'word6', 'word7', 'word8', 'word9', 'word10']
>>> sent[0]
'word1'
>>> sent[9]
'word10'
>>>
```

Notice that our indexes start from zero: `sent` element zero, written `sent[0]`, is the first word, `'word1'`, whereas `sent` element 9 is `'word10'`. The reason is simple: the moment Python accesses the content of a list from the computer's memory, it is already at the first element; we have to tell it how many elements forward to go. Thus, zero steps forward leaves it at the first element.

This practice of counting from zero is initially confusing, but typical of modern programming languages. You'll quickly get the hang of it if you've mastered the system of counting centuries where 19XY is a year in the 20th century, or if you live in a country where the floors of a building are numbered from 1, and so walking up *n-1* flights of stairs takes you to level *n*.

Now, if we accidentally use an index that is too large, we get an error:

```
>>> sent[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

This time it is not a syntax error, because the program fragment is syntactically correct. Instead, it is a **runtime error**, and it produces a `Traceback` message that shows the context of the error, followed by the name of the error, `IndexError`, and a brief explanation.

Let's take a closer look at slicing, using our artificial sentence again. Here we verify that the slice `5:8` includes `sent` elements at indexes 5, 6, and 7:

```
>>> sent[5:8]
['word6', 'word7', 'word8']
>>> sent[5]
'word6'
>>> sent[6]
'word7'
>>> sent[7]
'word8'
>>>
```

By convention, `m:n` means elements *m...n-1*. As the next example shows, we can omit the first number if the slice begins at the start of the list ❶, and we can omit the second number if the slice goes to the end ❷:

```
>>> sent[:3] ❶
['word1', 'word2', 'word3']
>>> text2[141525:] ❷
['among', 'the', 'merits', 'and', 'the', 'happiness', 'of', 'Elinor', 'and', 'Marianne',
',', 'let', 'it', 'not', 'be', 'ranked', 'as', 'the', 'least', 'considerable', ',',
'that', 'though', 'sisters', ',', 'and', 'living', 'almost', 'within', 'sight', 'of',
'each', 'other', ',', 'they', 'could', 'live', 'without', 'disagreement', 'between',
'themselves', ',', 'or', 'producing', 'coolness', 'between', 'their', 'husbands', '.',
'THE', 'END']
>>>
```

We can modify an element of a list by assigning to one of its index values. In the next example, we put `sent[0]` on the left of the equals sign ❶. We can also replace an entire slice with new material ❷. A consequence of this last change is that the list only has four elements, and accessing a later value generates an error ❸.

```
>>> sent[0] = 'First' ❶
>>> sent[9] = 'Last'
>>> len(sent)
10
>>> sent[1:9] = ['Second', 'Third'] ❷
>>> sent
['First', 'Second', 'Third', 'Last']
>>> sent[9] ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

Your Turn: Take a few minutes to define a sentence of your own and modify individual words and groups of words (slices) using the same methods used earlier. Check your understanding by trying the exercises on lists at the end of this chapter.

Variables

From the start of Section 1.1, you have had access to texts called `text1`, `text2`, and so on. It saved a lot of typing to be able to refer to a 250,000-word book with a short name like this! In general, we can make up names for anything we care to calculate. We did this ourselves in the previous sections, e.g., defining a **variable** `sent1`, as follows:

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>>
```

Such lines have the form: *variable = expression*. Python will evaluate the expression, and save its result to the variable. This process is called **assignment**. It does not generate any output; you have to type the variable on a line of its own to inspect its contents. The equals sign is slightly misleading, since information is moving from the right side to the left. It might help to think of it as a left-arrow. The name of the variable can be anything you like, e.g., `my_sent`, `sentence`, `xyzzy`. It must start with a letter, and can include numbers and underscores. Here are some examples of variables and assignments:

```
>>> my_sent = ['Bravely', 'bold', 'Sir', 'Robin', ',', 'rode',
... 'forth', 'from', 'Camelot', '.']
>>> noun_phrase = my_sent[1:4]
>>> noun_phrase
['bold', 'Sir', 'Robin']
>>> wOrDs = sorted(noun_phrase)
>>> wOrDs
['Robin', 'Sir', 'bold']
>>>
```

Remember that capitalized words appear before lowercase words in sorted lists.

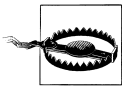
Notice in the previous example that we split the definition of `my_sent` over two lines. Python expressions can be split across multiple lines, so long as this happens within any kind of brackets. Python uses the `...` prompt to indicate that more input is expected. It doesn't matter how much indentation is used in these continuation lines, but some indentation usually makes them easier to read.

It is good to choose meaningful variable names to remind you—and to help anyone else who reads your Python code—what your code is meant to do. Python does not try to make sense of the names; it blindly follows your instructions, and does not object if you do something confusing, such as `one = 'two'` or `two = 3`. The only restriction is that a variable name cannot be any of Python's reserved words, such as `def`, `if`, `not`, and `import`. If you use a reserved word, Python will produce a syntax error:

```
>>> not = 'Camelot'
File "<stdin>", line 1
    not = 'Camelot'
        ^
SyntaxError: invalid syntax
>>>
```

We will often use variables to hold intermediate steps of a computation, especially when this makes the code easier to follow. Thus `len(set(text1))` could also be written:

```
>>> vocab = set(text1)
>>> vocab_size = len(vocab)
>>> vocab_size
19317
>>>
```



Caution!

Take care with your choice of names (or **identifiers**) for Python variables. First, you should start the name with a letter, optionally followed by digits (`0` to `9`) or letters. Thus, `abc23` is fine, but `23abc` will cause a syntax error. Names are case-sensitive, which means that `myVar` and `myvar` are distinct variables. Variable names cannot contain whitespace, but you can separate words using an underscore, e.g., `my_var`. Be careful not to insert a hyphen instead of an underscore: `my-var` is wrong, since Python interprets the `-` as a minus sign.

Strings

Some of the methods we used to access the elements of a list also work with individual words, or **strings**. For example, we can assign a string to a variable ❶, index a string ❷, and slice a string ❸.

```
>>> name = 'Monty' ❶
>>> name[0] ❷
'M'
>>> name[:4] ❸
'Mont'
>>>
```

We can also perform multiplication and addition with strings:

```
>>> name * 2
'MontyMonty'
>>> name + '!'
'Monty!'
>>>
```

We can join the words of a list to make a single string, or split a string into a list, as follows:

```
>>> ' '.join(['Monty', 'Python'])
'Monty Python'
>>> 'Monty Python'.split()
['Monty', 'Python']
>>>
```

We will come back to the topic of strings in Chapter 3. For the time being, we have two important building blocks—lists and strings—and are ready to get back to some language analysis.

1.3 Computing with Language: Simple Statistics

Let's return to our exploration of the ways we can bring our computational resources to bear on large quantities of text. We began this discussion in Section 1.1, and saw how to search for words in context, how to compile the vocabulary of a text, how to generate random text in the same style, and so on.

In this section, we pick up the question of what makes a text distinct, and use automatic methods to find characteristic words and expressions of a text. As in Section 1.1, you can try new features of the Python language by copying them into the interpreter, and you'll learn about these features systematically in the following section.

Before continuing further, you might like to check your understanding of the last section by predicting the output of the following code. You can use the interpreter to check whether you got it right. If you're not sure how to do this task, it would be a good idea to review the previous section before continuing further.

```
>>> saying = ['After', 'all', 'is', 'said', 'and', 'done',
...           'more', 'is', 'said', 'than', 'done']
>>> tokens = set(saying)
>>> tokens = sorted(tokens)
>>> tokens[-2:]
what output do you expect here?
>>>
```

Frequency Distributions

How can we automatically identify the words of a text that are most informative about the topic and genre of the text? Imagine how you might go about finding the 50 most frequent words of a book. One method would be to keep a tally for each vocabulary item, like that shown in Figure 1-3. The tally would need thousands of rows, and it would be an exceedingly laborious process—so laborious that we would rather assign the task to a machine.

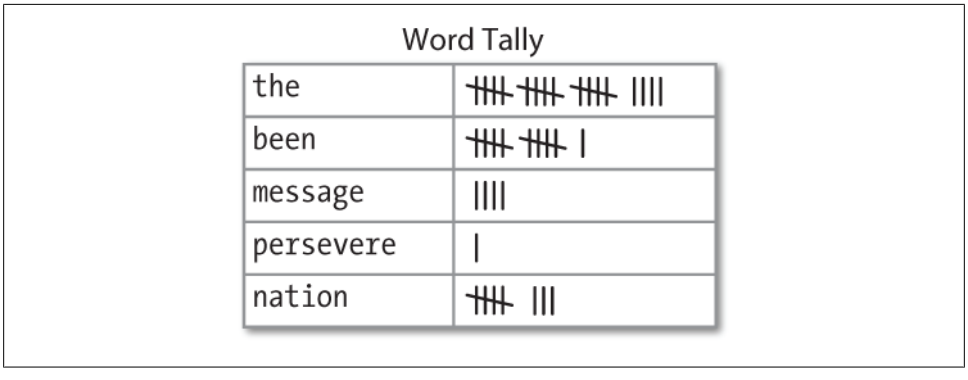


Figure 1-3. Counting words appearing in a text (a frequency distribution).

The table in Figure 1-3 is known as a **frequency distribution** , and it tells us the frequency of each vocabulary item in the text. (In general, it could count any kind of observable event.) It is a "distribution" since it tells us how the total number of word tokens in the text are distributed across the vocabulary items. Since we often need frequency distributions in language processing, NLTK provides built-in support for them. Let's use a `FreqDist` to find the 50 most frequent words of *Moby Dick*. Try to work out what is going on here, then read the explanation that follows.

```
>>> fdist1 = FreqDist(text1) ❶
>>> fdist1 ❷
<FreqDist with 260819 outcomes>
>>> vocabulary1 = fdist1.keys() ❸
>>> vocabulary1[:50] ❹
[',', 'the', '.', 'of', 'and', 'a', 'to', ';', 'in', 'that', "'", '-',
'his', 'it', 'I', 's', 'is', 'he', 'with', 'was', 'as', '"', 'all', 'for',
'this', '!', 'at', 'by', 'but', 'not', '--', 'him', 'from', 'be', 'on',
'so', 'whale', 'one', 'you', 'had', 'have', 'there', 'But', 'or', 'were',
'now', 'which', '?', 'me', 'like']
>>> fdist1['whale']
906
>>>
```

When we first invoke `FreqDist`, we pass the name of the text as an argument ❶. We can inspect the total number of words ("outcomes") that have been counted up ❷—260,819 in the case of *Moby Dick*. The expression `keys()` gives us a list of all the distinct types in the text ❸, and we can look at the first 50 of these by slicing the list ❹.

Your Turn: Try the preceding frequency distribution example for yourself, for `text2`. Be careful to use the correct parentheses and uppercase letters. If you get an error message `NameError: name 'FreqDist' is not defined`, you need to start your work with `from nltk.book import *`.

Do any words produced in the last example help us grasp the topic or genre of this text? Only one word, *whale*, is slightly informative! It occurs over 900 times. The rest of the words tell us nothing about the text; they're just English "plumbing." What proportion of the text is taken up with such words? We can generate a cumulative frequency plot for these words, using `fdist1.plot(50, cumulative=True)`, to produce the graph in Figure 1-4. These 50 words account for nearly half the book!

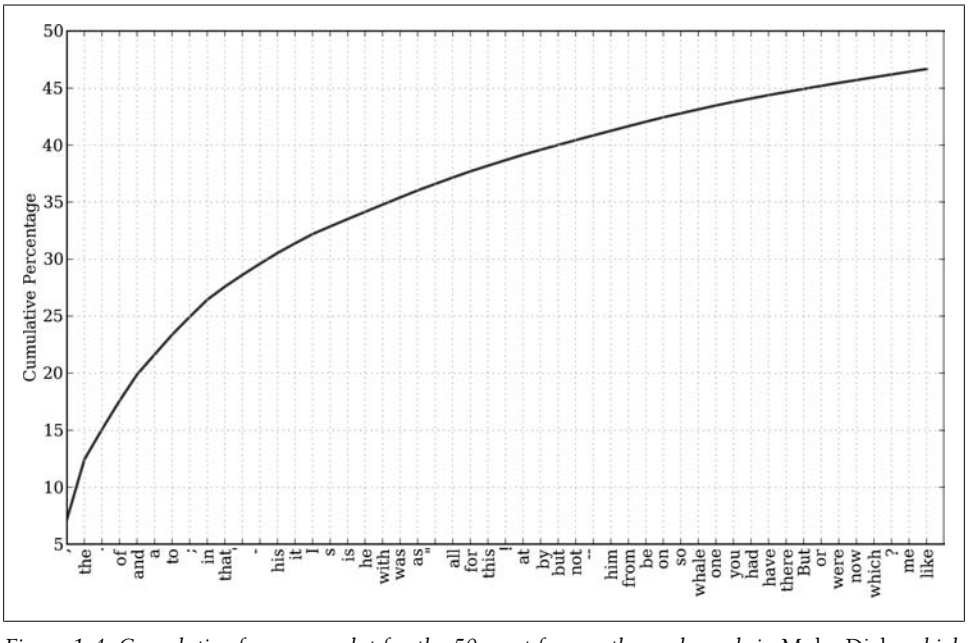


Figure 1-4. Cumulative frequency plot for the 50 most frequently used words in Moby Dick, *which account for nearly half of the tokens.*

If the frequent words don't help us, how about the words that occur once only, the so-called **hapaxes**? View them by typing `fdist1.hapaxes()`. This list contains *lexicographer*, *cetological*, *contraband*, *expostulations*, and about 9,000 others. It seems that there are too many rare words, and without seeing the context we probably can't guess what half of the hapaxes mean in any case! Since neither frequent nor infrequent words help, we need to try something else.

Fine-Grained Selection of Words

Next, let's look at the *long* words of a text; perhaps these will be more characteristic and informative. For this we adapt some notation from set theory. We would like to find the words from the vocabulary of the text that are more than 15 characters long. Let's call this property P, so that $P(w)$ is true if and only if w is more than 15 characters long. Now we can express the words of interest using mathematical set notation as shown in (1a). This means "the set of all w such that w is an element of V (the vocabulary) and w has property P."

(1) a. $\{w \mid w \in V \ \& \ P(w)\}$

b. `[w for w in V if p(w)]`

The corresponding Python expression is given in (1b). (Note that it produces a list, not a set, which means that duplicates are possible.) Observe how similar the two notations are. Let's go one more step and write executable Python code:

```
>>> V = set(text1)
>>> long_words = [w for w in V if len(w) > 15]
>>> sorted(long_words)
['CIRCUMNAVIGATION', 'Physiognomically', 'apprehensiveness', 'cannibalistically',
'characteristically', 'circumnavigating', 'circumnavigation', 'circumnavigations',
'comprehensiveness', 'hermaphroditical', 'indiscriminately', 'indispensableness',
'irresistibleness', 'physiognomically', 'preternaturalness', 'responsibilities',
'simultaneousness', 'subterraneousness', 'supernaturalness', 'superstitiousness',
'uncomfortableness', 'uncompromisedness', 'undiscriminating', 'uninterpenetratingly']
>>>
```

For each word `w` in the vocabulary `V`, we check whether `len(w)` is greater than 15; all other words will be ignored. We will discuss this syntax more carefully later.

Your Turn: Try out the previous statements in the Python interpreter, and experiment with changing the text and changing the length condition. Does it make an difference to your results if you change the variable names, e.g., using `[word for word in vocab if ...]`?

Let's return to our task of finding words that characterize a text. Notice that the long words in `text4` reflect its national focus—*constitutionally*, *transcontinental*—whereas those in `text5` reflect its informal content: *boooooooooooglyyyyyy* and *yuuuuuuuuuuuummmmmmmmmmmm*. Have we succeeded in automatically extracting words that typify a text? Well, these very long words are often hapaxes (i.e., unique) and perhaps it would be better to find *frequently occurring* long words. This seems promising since it eliminates frequent short words (e.g., *the*) and infrequent long words (e.g., *antiphilosophists*). Here are all words from the chat corpus that are longer than seven characters, that occur more than seven times:

```
>>> fdist5 = FreqDist(text5)
>>> sorted([w for w in set(text5) if len(w) > 7 and fdist5[w] > 7])
['#14-19teens', '#talkcity_adults', '((((((((((', '........', 'Question',
'actually', 'anything', 'computer', 'cute.-ass', 'everyone', 'football',
'innocent', 'listening', 'remember', 'seriously', 'something', 'together',
'tomorrow', 'watching']
>>>
```

Notice how we have used two conditions: `len(w) > 7` ensures that the words are longer than seven letters, and `fdist5[w] > 7` ensures that these words occur more than seven times. At last we have managed to automatically identify the frequently occurring content-bearing words of the text. It is a modest but important milestone: a tiny piece of code, processing tens of thousands of words, produces some informative output.

Collocations and Bigrams

A **collocation** is a sequence of words that occur together unusually often. Thus *red wine* is a collocation, whereas *the wine* is not. A characteristic of collocations is that they are resistant to substitution with words that have similar senses; for example, *maroon wine* sounds very odd.

To get a handle on collocations, we start off by extracting from a text a list of word pairs, also known as **bigrams**. This is easily accomplished with the function `bigrams()`:

```
>>> bigrams(['more', 'is', 'said', 'than', 'done'])
[('more', 'is'), ('is', 'said'), ('said', 'than'), ('than', 'done')]
>>>
```

Here we see that the pair of words *than-done* is a bigram, and we write it in Python as `('than', 'done')`. Now, collocations are essentially just frequent bigrams, except that we want to pay more attention to the cases that involve rare words. In particular, we want to find bigrams that occur more often than we would expect based on the frequency of individual words. The `collocations()` function does this for us (we will see how it works later):

```
>>> text4.collocations()
Building collocations list
United States; fellow citizens; years ago; Federal Government; General
Government; American people; Vice President; Almighty God; Fellow
citizens; Chief Magistrate; Chief Justice; God bless; Indian tribes;
public debt; foreign nations; political parties; State governments;
```

```
National Government; United Nations; public money
>>> text8.collocations()
Building collocations list
medium build; social drinker; quiet nights; long term; age open;
financially secure; fun times; similar interests; Age open; poss
rship; single mum; permanent relationship; slim build; seeks lady;
Late 30s; Photo pls; Vibrant personality; European background; ASIAN
LADY; country drives
>>>
```

The collocations that emerge are very specific to the genre of the texts. In order to find *red wine* as a collocation, we would need to process a much larger body of text.

Counting Other Things

Counting words is useful, but we can count other things too. For example, we can look at the distribution of word lengths in a text, by creating a `FreqDist` out of a long list of numbers, where each number is the length of the corresponding word in the text:

```
>>> [len(w) for w in text1] ❶
[1, 4, 4, 2, 6, 8, 4, 1, 9, 1, 1, 8, 2, 1, 4, 11, 5, 2, 1, 7, 6, 1, 3, 4, 5, 2, ...]
>>> fdist = FreqDist([len(w) for w in text1])  ❷
>>> fdist  ❸
<FreqDist with 260819 outcomes>
>>> fdist.keys()
[3, 1, 4, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20]
>>>
```

We start by deriving a list of the lengths of words in `text1` ❶, and the `FreqDist` then counts the number of times each of these occurs ❷. The result ❸ is a distribution containing a quarter of a million items, each of which is a number corresponding to a word token in the text. But there are only 20 distinct items being counted, the numbers 1 through 20, because there are only 20 different word lengths. I.e., there are words consisting of just 1 character, 2 characters, ..., 20 characters, but none with 21 or more characters. One might wonder how frequent the different lengths of words are (e.g., how many words of length 4 appear in the text, are there more words of length 5 than length 4, etc.). We can do this as follows:

```
>>> fdist.items()
[(3, 50223), (1, 47933), (4, 42345), (2, 38513), (5, 26597), (6, 17111), (7, 14399),
(8, 9966), (9, 6428), (10, 3528), (11, 1873), (12, 1053), (13, 567), (14, 177),
(15, 70), (16, 22), (17, 12), (18, 1), (20, 1)]
>>> fdist.max()
3
>>> fdist[3]
50223
>>> fdist.freq(3)
0.19255882431878046
>>>
```

From this we see that the most frequent word length is 3, and that words of length 3 account for roughly 50,000 (or 20%) of the words making up the book. Although we will not pursue it here, further analysis of word length might help us understand

differences between authors, genres, or languages. Table 1-2 summarizes the functions defined in frequency distributions.

Table 1-2. Functions defined for NLTK's frequency distributions

Example	Description
`fdist = FreqDist(samples)`	Create a frequency distribution containing the given samples
`fdist.inc(sample)`	Increment the count for this sample
`fdist['monstrous']`	Count of the number of times a given sample occurred
`fdist.freq('monstrous')`	Frequency of a given sample
`fdist.N()`	Total number of samples
`fdist.keys()`	The samples sorted in order of decreasing frequency
`for sample in fdist:`	Iterate over the samples, in order of decreasing frequency
`fdist.max()`	Sample with the greatest count
`fdist.tabulate()`	Tabulate the frequency distribution
`fdist.plot()`	Graphical plot of the frequency distribution
`fdist.plot(cumulative=True)`	Cumulative plot of the frequency distribution
`fdist1 < fdist2`	Test if samples in `fdist1` occur less frequently than in `fdist2`

Our discussion of frequency distributions has introduced some important Python concepts, and we will look at them systematically in Section 1.4.

1.4 Back to Python: Making Decisions and Taking Control

So far, our little programs have had some interesting qualities: the ability to work with language, and the potential to save human effort through automation. A key feature of programming is the ability of machines to make decisions on our behalf, executing instructions when certain conditions are met, or repeatedly looping through text data until some condition is satisfied. This feature is known as **control**, and is the focus of this section.

Conditionals

Python supports a wide range of operators, such as `<` and `>=`, for testing the relationship between values. The full set of these **relational operators** are shown in Table 1-3.

Table 1-3. Numerical comparison operators

Operator	Relationship
`<`	Less than
`<=`	Less than or equal to
`==`	Equal to (note this is two "="signs, not one)

Operator	Relationship
!=	Not equal to
>	Greater than
>=	Greater than or equal to

We can use these to select different words from a sentence of news text. Here are some examples—notice only the operator is changed from one line to the next. They all use `sent7`, the first sentence from `text7` (*Wall Street Journal*). As before, if you get an error saying that `sent7` is undefined, you need to first type: `from nltk.book import *`.

```
>>> sent7
['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the',
'board', 'as', 'a', 'nonexecutive', 'director', 'Nov.', '29', '.']
>>> [w for w in sent7 if len(w) < 4]
[',', '61', 'old', ',', 'the', 'as', 'a', '29', '.']
>>> [w for w in sent7 if len(w) <= 4]
[',', '61', 'old', ',', 'will', 'join', 'the', 'as', 'a', 'Nov.', '29', '.']
>>> [w for w in sent7 if len(w) == 4]
['will', 'join', 'Nov.']
>>> [w for w in sent7 if len(w) != 4]
['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'the', 'board',
'as', 'a', 'nonexecutive', 'director', '29', '.']
>>>
```

There is a common pattern to all of these examples: `[w for w in text if condition]`, where *`condition`* is a Python "test" that yields either true or false. In the cases shown in the previous code example, the condition is always a numerical comparison. However, we can also test various properties of words, using the functions listed in Table 1-4.

Table 1-4. Some word comparison operators

Function	Meaning
`s.startswith(t)`	Test if s starts with t
`s.endswith(t)`	Test if s ends with t
`t in s`	Test if t is contained inside s
`s.islower()`	Test if all cased characters in s are lowercase
`s.isupper()`	Test if all cased characters in s are uppercase
`s.isalpha()`	Test if all characters in s are alphabetic
`s.isalnum()`	Test if all characters in s are alphanumeric
`s.isdigit()`	Test if all characters in s are digits
`s.istitle()`	Test if s is titlecased (all words in s have initial capitals)

Here are some examples of these operators being used to select words from our texts: words ending with *-ableness*; words containing *gnt*; words having an initial capital; and words consisting entirely of digits.

```
>>> sorted([w for w in set(text1) if w.endswith('ableness')])
['comfortableness', 'honourableness', 'immutableness', 'indispensableness', ...]
>>> sorted([term for term in set(text4) if 'gnt' in term])
['Sovereignty', 'sovereignties', 'sovereignty']
>>> sorted([item for item in set(text6) if item.istitle()])
['A', 'Aaaaaaaaah', 'Aaaaaaaah', 'Aaaaaah', 'Aaaah', 'Aaaaugh', 'Aaagh', ...]
>>> sorted([item for item in set(sent7) if item.isdigit()])
['29', '61']
>>>
```

We can also create more complex conditions. If c is a condition, then `not` c is also a condition. If we have two conditions c_1 and c_2, then we can combine them to form a new condition using conjunction and disjunction: c_1 `and` c_2, c_1 `or` c_2.

Your Turn: Run the following examples and try to explain what is going on in each one. Next, try to make up some conditions of your own.

```
>>> sorted([w for w in set(text7) if '-' in w and 'index' in w])
>>> sorted([wd for wd in set(text3) if wd.istitle() and len(wd) > 10])
>>> sorted([w for w in set(sent7) if not w.islower()])
>>> sorted([t for t in set(text2) if 'cie' in t or 'cei' in t])
```

Operating on Every Element

In Section 1.3, we saw some examples of counting items other than words. Let's take a closer look at the notation we used:

```
>>> [len(w) for w in text1]
[1, 4, 4, 2, 6, 8, 4, 1, 9, 1, 1, 8, 2, 1, 4, 11, 5, 2, 1, 7, 6, 1, 3, 4, 5, 2, ...]
>>> [w.upper() for w in text1]
['[', 'MOBY', 'DICK', 'BY', 'HERMAN', 'MELVILLE', '1851', ']', 'ETYMOLOGY', '.', ...]
>>>
```

These expressions have the form `[f(w) for ...]` or `[w.f() for ...]`, where `f` is a function that operates on a word to compute its length, or to convert it to uppercase. For now, you don't need to understand the difference between the notations `f(w)` and `w.f()`. Instead, simply learn this Python idiom which performs the same operation on every element of a list. In the preceding examples, it goes through each word in `text1`, assigning each one in turn to the variable `w` and performing the specified operation on the variable.

The notation just described is called a "list comprehension." This is our first example of a Python idiom, a fixed notation that we use habitually without bothering to analyze each time. Mastering such idioms is an important part of becoming a fluent Python programmer.

Let's return to the question of vocabulary size, and apply the same idiom here:

```
>>> len(text1)
260819
```

```
>>> len(set(text1))
19317
>>> len(set([word.lower() for word in text1]))
17231
>>>
```

Now that we are not double-counting words like *This* and *this*, which differ only in capitalization, we've wiped 2,000 off the vocabulary count! We can go a step further and eliminate numbers and punctuation from the vocabulary count by filtering out any non-alphabetic items:

```
>>> len(set([word.lower() for word in text1 if word.isalpha()]))
16948
>>>
```

This example is slightly complicated: it lowercases all the purely alphabetic items. Perhaps it would have been simpler just to count the lowercase-only items, but this gives the wrong answer (why?).

Don't worry if you don't feel confident with list comprehensions yet, since you'll see many more examples along with explanations in the following chapters.

Nested Code Blocks

Most programming languages permit us to execute a block of code when a **conditional expression**, or `if` statement, is satisfied. We already saw examples of conditional tests in code like `[w for w in sent7 if len(w) < 4]`. In the following program, we have created a variable called `word` containing the string value `'cat'`. The `if` statement checks whether the test `len(word) < 5` is true. It is, so the body of the `if` statement is invoked and the `print` statement is executed, displaying a message to the user. Remember to indent the `print` statement by typing four spaces.

```
>>> word = 'cat'
>>> if len(word) < 5:
...     print 'word length is less than 5'
...   ❶
word length is less than 5
>>>
```

When we use the Python interpreter we have to add an extra blank line ❶ in order for it to detect that the nested block is complete.

If we change the conditional test to `len(word) >= 5`, to check that the length of `word` is greater than or equal to `5`, then the test will no longer be true. This time, the body of the `if` statement will not be executed, and no message is shown to the user:

```
>>> if len(word) >= 5:
...   print 'word length is greater than or equal to 5'
...
>>>
```

An `if` statement is known as a **control structure** because it controls whether the code in the indented block will be run. Another control structure is the `for` loop. Try the following, and remember to include the colon and the four spaces:

```
>>> for word in ['Call', 'me', 'Ishmael', '.']:
...     print word
...
Call
me
Ishmael
.
>>>
```

This is called a loop because Python executes the code in circular fashion. It starts by performing the assignment `word = 'Call'`, effectively using the `word` variable to name the first item of the list. Then, it displays the value of `word` to the user. Next, it goes back to the `for` statement, and performs the assignment `word = 'me'` before displaying this new value to the user, and so on. It continues in this fashion until every item of the list has been processed.

Looping with Conditions

Now we can combine the `if` and `for` statements. We will loop over every item of the list, and print the item only if it ends with the letter *l*. We'll pick another name for the variable to demonstrate that Python doesn't try to make sense of variable names.

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>> for xyzzy in sent1:
...     if xyzzy.endswith('l'):
...         print xyzzy
...
Call
Ishmael
>>>
```

You will notice that `if` and `for` statements have a colon at the end of the line, before the indentation begins. In fact, all Python control structures end with a colon. The colon indicates that the current statement relates to the indented block that follows.

We can also specify an action to be taken if the condition of the `if` statement is not met. Here we see the `elif` (else if) statement, and the `else` statement. Notice that these also have colons before the indented code.

```
>>> for token in sent1:
...     if token.islower():
...         print token, 'is a lowercase word'
...     elif token.istitle():
...         print token, 'is a titlecase word'
...     else:
...         print token, 'is punctuation'
...
Call is a titlecase word
me is a lowercase word
```

```
Ishmael is a titlecase word
. is punctuation
>>>
```

As you can see, even with this small amount of Python knowledge, you can start to build multiline Python programs. It's important to develop such programs in pieces, testing that each piece does what you expect before combining them into a program. This is why the Python interactive interpreter is so invaluable, and why you should get comfortable using it.

Finally, let's combine the idioms we've been exploring. First, we create a list of *cie* and *cei* words, then we loop over each item and print it. Notice the comma at the end of the print statement, which tells Python to produce its output on a single line.

```
>>> tricky = sorted([w for w in set(text2) if 'cie' in w or 'cei' in w])
>>> for word in tricky:
...     print word,
ancient ceiling conceit conceited conceive conscience
conscientious conscientiously deceitful deceive ...
>>>
```

1.5 Automatic Natural Language Understanding

We have been exploring language bottom-up, with the help of texts and the Python programming language. However, we're also interested in exploiting our knowledge of language and computation by building useful language technologies. We'll take the opportunity now to step back from the nitty-gritty of code in order to paint a bigger picture of natural language processing.

At a purely practical level, we all need help to navigate the universe of information locked up in text on the Web. Search engines have been crucial to the growth and popularity of the Web, but have some shortcomings. It takes skill, knowledge, and some luck, to extract answers to such questions as: *What tourist sites can I visit between Philadelphia and Pittsburgh on a limited budget? What do experts say about digital SLR cameras? What predictions about the steel market were made by credible commentators in the past week?* Getting a computer to answer them automatically involves a range of language processing tasks, including information extraction, inference, and summarization, and would need to be carried out on a scale and with a level of robustness that is still beyond our current capabilities.

On a more philosophical level, a long-standing challenge within artificial intelligence has been to build intelligent machines, and a major part of intelligent behavior is understanding language. For many years this goal has been seen as too difficult. However, as NLP technologies become more mature, and robust methods for analyzing unrestricted text become more widespread, the prospect of natural language understanding has re-emerged as a plausible goal.

CHAPTER 2

Accessing Text Corpora and Lexical Resources

Practical work in Natural Language Processing typically uses large bodies of linguistic data, or **corpora**. The goal of this chapter is to answer the following questions:

1. What are some useful text corpora and lexical resources, and how can we access them with Python?
2. Which Python constructs are most helpful for this work?
3. How do we avoid repeating ourselves when writing Python code?

This chapter continues to present programming concepts by example, in the context of a linguistic processing task. We will wait until later before exploring each Python construct systematically. Don't worry if you see an example that contains something unfamiliar; simply try it out and see what it does, and—if you're game—modify it by substituting some part of the code with a different text or word. This way you will associate a task with a programming idiom, and learn the hows and whys later.

2.1 Accessing Text Corpora

As just mentioned, a text corpus is a large body of text. Many corpora are designed to contain a careful balance of material in one or more genres. We examined some small text collections in Chapter 1, such as the speeches known as the US Presidential Inaugural Addresses. This particular corpus actually contains dozens of individual texts—one per address—but for convenience we glued them end-to-end and treated them as a single text. Chapter 1 also used various predefined texts that we accessed by typing `from book import *`. However, since we want to be able to work with other texts, this section examines a variety of text corpora. We'll see how to select individual texts, and how to work with them.

Gutenberg Corpus

NLTK includes a small selection of texts from the Project Gutenberg electronic text archive, which contains some 25,000 free electronic books, hosted at *http://www.gutenberg.org/*. We begin by getting the Python interpreter to load the NLTK package, then ask to see `nltk.corpus.gutenberg.fileids()`, the file identifiers in this corpus:

```
>>> import nltk
>>> nltk.corpus.gutenberg.fileids()
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt',
'blake-poems.txt', 'bryant-stories.txt', 'burgess-busterbrown.txt',
'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt',
'chesterton-thursday.txt', 'edgeworth-parents.txt', 'melville-moby_dick.txt',
'milton-paradise.txt', 'shakespeare-caesar.txt', 'shakespeare-hamlet.txt',
'shakespeare-macbeth.txt', 'whitman-leaves.txt']
```

Let's pick out the first of these texts—*Emma* by Jane Austen—and give it a short name, `emma`, then find out how many words it contains:

```
>>> emma = nltk.corpus.gutenberg.words('austen-emma.txt')
>>> len(emma)
192427
```

In Section 1.1, we showed how you could carry out concordancing of a text such as `text1` with the command `text1.concordance()`. However, this assumes that you are using one of the nine texts obtained as a result of doing `from nltk.book import *`. Now that you have started examining data from `nltk.corpus`, as in the previous example, you have to employ the following pair of statements to perform concordancing and other tasks from Section 1.1:

```
>>> emma = nltk.Text(nltk.corpus.gutenberg.words('austen-emma.txt'))
>>> emma.concordance("surprize")
```

When we defined `emma`, we invoked the `words()` function of the `gutenberg` object in NLTK's `corpus` package. But since it is cumbersome to type such long names all the time, Python provides another version of the `import` statement, as follows:

```
>>> from nltk.corpus import gutenberg
>>> gutenberg.fileids()
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', ...]
>>> emma = gutenberg.words('austen-emma.txt')
```

Let's write a short program to display other information about each text, by looping over all the values of `fileid` corresponding to the `gutenberg` file identifiers listed earlier and then computing statistics for each text. For a compact output display, we will make sure that the numbers are all integers, using `int()`.

```
>>> for fileid in gutenberg.fileids():
...     num_chars = len(gutenberg.raw(fileid)) ❶
...     num_words = len(gutenberg.words(fileid))
...     num_sents = len(gutenberg.sents(fileid))
```

```
...     num_vocab = len(set([w.lower() for w in gutenberg.words(fileid)]))
...     print int(num_chars/num_words), int(num_words/num_sents), int(num_words/num_vocab),
        fileid
...
4 21 26 austen-emma.txt
4 23 16 austen-persuasion.txt
4 24 22 austen-sense.txt
4 33 79 bible-kjv.txt
4 18 5 blake-poems.txt
4 17 14 bryant-stories.txt
4 17 12 burgess-busterbrown.txt
4 16 12 carroll-alice.txt
4 17 11 chesterton-ball.txt
4 19 11 chesterton-brown.txt
4 16 10 chesterton-thursday.txt
4 18 24 edgeworth-parents.txt
4 24 15 melville-moby_dick.txt
4 52 10 milton-paradise.txt
4 12 8 shakespeare-caesar.txt
4 13 7 shakespeare-hamlet.txt
4 13 6 shakespeare-macbeth.txt
4 35 12 whitman-leaves.txt
```

This program displays three statistics for each text: average word length, average sentence length, and the number of times each vocabulary item appears in the text on average (our lexical diversity score). Observe that average word length appears to be a general property of English, since it has a recurrent value of *4*. (In fact, the average word length is really *3*, not *4*, since the `num_chars` variable counts space characters.) By contrast average sentence length and lexical diversity appear to be characteristics of particular authors.

The previous example also showed how we can access the "raw" text of the book ❶, not split up into tokens. The `raw()` function gives us the contents of the file without any linguistic processing. So, for example, `len(gutenberg.raw('blake-poems.txt'))` tells us how many *letters* occur in the text, including the spaces between words. The `sents()` function divides the text up into its sentences, where each sentence is a list of words:

```
>>> macbeth_sentences = gutenberg.sents('shakespeare-macbeth.txt')
>>> macbeth_sentences
[['[', 'The', 'Tragedie', 'of', 'Macbeth', 'by', 'William', 'Shakespeare',
'1603', ']'], ['Actus', 'Primus', '.'], ...]
>>> macbeth_sentences[1037]
['Double', ',', 'double', ',', 'toile', 'and', 'trouble', ';',
'Fire', 'burne', ',', 'and', 'Cauldron', 'bubble']
>>> longest_len = max([len(s) for s in macbeth_sentences])
>>> [s for s in macbeth_sentences if len(s) == longest_len]
[['Doubtfull', 'it', 'stood', ',', 'As', 'two', 'spent', 'Swimmers', ',', 'that',
'doe', 'cling', 'together', ',', 'And', 'choake', 'their', 'Art', ':', 'The',
'mercilesse', 'Macdonwald', ...], ...]
```

Most NLTK corpus readers include a variety of access methods apart from `words()`, `raw()`, and `sents()`. Richer linguistic content is available from some corpora, such as part-of-speech tags, dialogue tags, syntactic trees, and so forth; we will see these in later chapters.

Web and Chat Text

Although Project Gutenberg contains thousands of books, it represents established literature. It is important to consider less formal language as well. NLTK's small collection of web text includes content from a Firefox discussion forum, conversations overheard in New York, the movie script of *Pirates of the Carribean*, personal advertisements, and wine reviews:

```
>>> from nltk.corpus import webtext
>>> for fileid in webtext.fileids():
...     print fileid, webtext.raw(fileid)[:65], '...'
...
firefox.txt Cookie Manager: "Don't allow sites that set removed cookies to se...
grail.txt SCENE 1: [wind] [clop clop clop] KING ARTHUR: Whoa there!  [clop...
overheard.txt White guy: So, do you have any plans for this evening? Asian girl...
pirates.txt PIRATES OF THE CARRIBEAN: DEAD MAN'S CHEST, by Ted Elliott & Terr...
singles.txt 25 SEXY MALE, seeks attrac older single lady, for discreet encoun...
wine.txt Lovely delicate, fragrant Rhone wine. Polished leather and strawb...
```

There is also a corpus of instant messaging chat sessions, originally collected by the Naval Postgraduate School for research on automatic detection of Internet predators. The corpus contains over 10,000 posts, anonymized by replacing usernames with generic names of the form "UserNNN", and manually edited to remove any other identifying information. The corpus is organized into 15 files, where each file contains several hundred posts collected on a given date, for an age-specific chatroom (teens, 20s, 30s, 40s, plus a generic adults chatroom). The filename contains the date, chatroom, and number of posts; e.g., `10-19-20s_706posts.xml` contains 706 posts gathered from the 20s chat room on 10/19/2006.

```
>>> from nltk.corpus import nps_chat
>>> chatroom = nps_chat.posts('10-19-20s_706posts.xml')
>>> chatroom[123]
['i', 'do', "n't", 'want', 'hot', 'pics', 'of', 'a', 'female', ',',
'I', 'can', 'look', 'in', 'a', 'mirror', '.']
```

Brown Corpus

The Brown Corpus was the first million-word electronic corpus of English, created in 1961 at Brown University. This corpus contains text from 500 sources, and the sources have been categorized by genre, such as *news*, *editorial*, and so on. Table 2-1 gives an example of each genre (for a complete list, see *http://icame.uib.no/brown/bcm-los.html*).

Table 2-1. Example document for each section of the Brown Corpus

ID	File	Genre	Description
A16	*ca16*	news	Chicago Tribune: *Society Reportage*
B02	*cb02*	editorial	Christian Science Monitor: *Editorials*
C17	*cc17*	reviews	Time Magazine: *Reviews*
D12	*cd12*	religion	Underwood: *Probing the Ethics of Realtors*
E36	*ce36*	hobbies	Norling: *Renting a Car in Europe*
F25	*cf25*	lore	Boroff: *Jewish Teenage Culture*
G22	*cg22*	belles_lettres	Reiner: *Coping with Runaway Technology*
H15	*ch15*	government	US Office of Civil and Defence Mobilization: *The Family Fallout Shelter*
J17	*cj19*	learned	Mosteller: *Probability with Statistical Applications*
K04	*ck04*	fiction	W.E.B. Du Bois: *Worlds of Color*
L13	*cl13*	mystery	Hitchens: *Footsteps in the Night*
M01	*cm01*	science_fiction	Heinlein: *Stranger in a Strange Land*
N14	*cn15*	adventure	Field: *Rattlesnake Ridge*
P12	*cp12*	romance	Callaghan: *A Passion in Rome*
R06	*cr06*	humor	Thurber: *The Future, If Any, of Comedy*

We can access the corpus as a list of words or a list of sentences (where each sentence is itself just a list of words). We can optionally specify particular categories or files to read:

```
>>> from nltk.corpus import brown
>>> brown.categories()
['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies',
'humor', 'learned', 'lore', 'mystery', 'news', 'religion', 'reviews', 'romance',
'science_fiction']
>>> brown.words(categories='news')
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
>>> brown.words(fileids=['cg22'])
['Does', 'our', 'society', 'have', 'a', 'runaway', ',', ...]
>>> brown.sents(categories=['news', 'editorial', 'reviews'])
[['The', 'Fulton', 'County'...], ['The', 'jury', 'further'...], ...]
```

The Brown Corpus is a convenient resource for studying systematic differences between genres, a kind of linguistic inquiry known as **stylistics**. Let's compare genres in their usage of modal verbs. The first step is to produce the counts for a particular genre. Remember to `import nltk` before doing the following:

```
>>> from nltk.corpus import brown
>>> news_text = brown.words(categories='news')
>>> fdist = nltk.FreqDist([w.lower() for w in news_text])
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> for m in modals:
...     print m + ':', fdist[m],
```

```
...
can: 94 could: 87 may: 93 might: 38 must: 53 will: 389
```

Your Turn: Choose a different section of the Brown Corpus, and adapt the preceding example to count a selection of *wh* words, such as *what*, *when*, *where*, *who* and *why*.

Next, we need to obtain counts for each genre of interest. We'll use NLTK's support for conditional frequency distributions. These are presented systematically in Section 2.2, where we also unpick the following code line by line. For the moment, you can ignore the details and just concentrate on the output.

```
>>> cfd = nltk.ConditionalFreqDist(
...           (genre, word)
...           for genre in brown.categories()
...           for word in brown.words(categories=genre))
>>> genres = ['news', 'religion', 'hobbies', 'science_fiction', 'romance', 'humor']
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> cfd.tabulate(conditions=genres, samples=modals)
                  can could  may might must will
            news   93   86   66   38   50  389
       religion   82   59   78   12   54   71
        hobbies  268   58  131   22   83  264
science_fiction   16   49    4   12    8   16
        romance   74  193   11   51   45   43
          humor   16   30    8    8    9   13
```

Observe that the most frequent modal in the news genre is *will*, while the most frequent modal in the romance genre is *could*. Would you have predicted this? The idea that word counts might distinguish genres will be taken up again in Chapter 6.

Reuters Corpus

The Reuters Corpus contains 10,788 news documents totaling 1.3 million words. The documents have been classified into 90 topics, and grouped into two sets, called "training" and "test"; thus, the text with fileid `'test/14826'` is a document drawn from the test set. This split is for training and testing algorithms that automatically detect the topic of a document, as we will see in Chapter 6.

```
>>> from nltk.corpus import reuters
>>> reuters.fileids()
['test/14826', 'test/14828', 'test/14829', 'test/14832', ...]
>>> reuters.categories()
['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa',
'coconut', 'coconut-oil', 'coffee', 'copper', 'copra-cake', 'corn',
'cotton', 'cotton-oil', 'cpi', 'cpu', 'crude', 'dfl', 'dlr', ...]
```

Unlike the Brown Corpus, categories in the Reuters Corpus overlap with each other, simply because a news story often covers multiple topics. We can ask for the topics

covered by one or more documents, or for the documents included in one or more categories. For convenience, the corpus methods accept a single fileid or a list of fileids.

```
>>> reuters.categories('training/9865')
['barley', 'corn', 'grain', 'wheat']
>>> reuters.categories(['training/9865', 'training/9880'])
['barley', 'corn', 'grain', 'money-fx', 'wheat']
>>> reuters.fileids('barley')
['test/15618', 'test/15649', 'test/15676', 'test/15728', 'test/15871', ...]
>>> reuters.fileids(['barley', 'corn'])
['test/14832', 'test/14858', 'test/15033', 'test/15043', 'test/15106',
'test/15287', 'test/15341', 'test/15618', 'test/15618', 'test/15648', ...]
```

Similarly, we can specify the words or sentences we want in terms of files or categories. The first handful of words in each of these texts are the titles, which by convention are stored as uppercase.

```
>>> reuters.words('training/9865')[:14]
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', 'BIDS',
'DETAILED', 'French', 'operators', 'have', 'requested', 'licences', 'to', 'export']
>>> reuters.words(['training/9865', 'training/9880'])
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', ...]
>>> reuters.words(categories='barley')
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', ...]
>>> reuters.words(categories=['barley', 'corn'])
['THAI', 'TRADE', 'DEFICIT', 'WIDENS', 'IN', 'FIRST', ...]
```

Inaugural Address Corpus

In Section 1.1, we looked at the Inaugural Address Corpus, but treated it as a single text. The graph in Figure 1-2 used "word offset" as one of the axes; this is the numerical index of the word in the corpus, counting from the first word of the first address. However, the corpus is actually a collection of 55 texts, one for each presidential address. An interesting property of this collection is its time dimension:

```
>>> from nltk.corpus import inaugural
>>> inaugural.fileids()
['1789-Washington.txt', '1793-Washington.txt', '1797-Adams.txt', ...]
>>> [fileid[:4] for fileid in inaugural.fileids()]
['1789', '1793', '1797', '1801', '1805', '1809', '1813', '1817', '1821', ...]
```

Notice that the year of each text appears in its filename. To get the year out of the filename, we extracted the first four characters, using `fileid[:4]`.

Let's look at how the words *America* and *citizen* are used over time. The following code converts the words in the Inaugural corpus to lowercase using `w.lower()` ❶, then checks whether they start with either of the "targets" `america` or `citizen` using `startswith()` ❶. Thus it will count words such as *American's* and *Citizens*. We'll learn about conditional frequency distributions in Section 2.2; for now, just consider the output, shown in Figure 2-1.

```
>>> cfd = nltk.ConditionalFreqDist(
...           (target, file[:4])
...           for fileid in inaugural.fileids()
...           for w in inaugural.words(fileid)
...           for target in ['america', 'citizen']
...           if w.lower().startswith(target)) ❶
>>> cfd.plot()
```

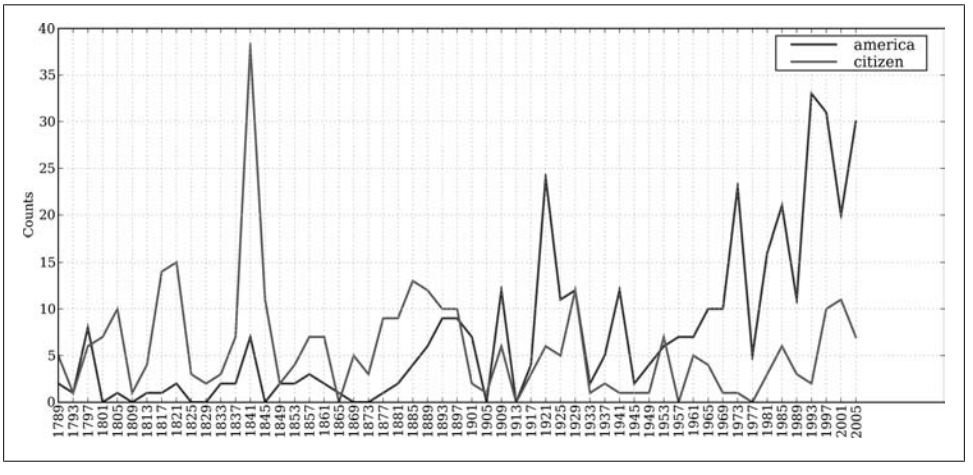


Figure 2-1. Plot of a conditional frequency distribution: All words in the Inaugural Address Corpus that begin with america *or* citizen *are counted; separate counts are kept for each address; these are plotted so that trends in usage over time can be observed; counts are not normalized for document length.*

Annotated Text Corpora

Many text corpora contain linguistic annotations, representing part-of-speech tags, named entities, syntactic structures, semantic roles, and so forth. NLTK provides convenient ways to access several of these corpora, and has data packages containing corpora and corpus samples, freely downloadable for use in teaching and research. Table 2-2 lists some of the corpora. For information about downloading them, see *http://www.nltk.org/data*. For more examples of how to access NLTK corpora, please consult the Corpus HOWTO at *http://www.nltk.org/howto*.

Table 2-2. Some of the corpora and corpus samples distributed with NLTK

Corpus	Compiler	Contents
Brown Corpus	Francis, Kucera	15 genres, 1.15M words, tagged, categorized
CESS Treebanks	CLiC-UB	1M words, tagged and parsed (Catalan, Spanish)
Chat-80 Data Files	Pereira & Warren	World Geographic Database
CMU Pronouncing Dictionary	CMU	127k entries
CoNLL 2000 Chunking Data	CoNLL	270k words, tagged and chunked

Corpus	Compiler	Contents
CoNLL 2002 Named Entity	CoNLL	700k words, POS and named entity tagged (Dutch, Spanish)
CoNLL 2007 Dependency Parsed Treebanks (selections)	CoNLL	150k words, dependency parsed (Basque, Catalan)
Dependency Treebank	Narad	Dependency parsed version of Penn Treebank sample
Floresta Treebank	Diana Santos et al.	9k sentences, tagged and parsed (Portuguese)
Gazetteer Lists	Various	Lists of cities and countries
Genesis Corpus	Misc web sources	6 texts, 200k words, 6 languages
Gutenberg (selections)	Hart, Newby, et al.	18 texts, 2M words
Inaugural Address Corpus	CSpan	U.S. Presidential Inaugural Addresses (1789–present)
Indian POS Tagged Corpus	Kumaran et al.	60k words, tagged (Bangla, Hindi, Marathi, Telugu)
MacMorpho Corpus	NILC, USP, Brazil	1M words, tagged (Brazilian Portuguese)
Movie Reviews	Pang, Lee	2k movie reviews with sentiment polarity classification
Names Corpus	Kantrowitz, Ross	8k male and female names
NIST 1999 Info Extr (selections)	Garofolo	63k words, newswire and named entity SGML markup
NPS Chat Corpus	Forsyth, Martell	10k IM chat posts, POS and dialogue-act tagged
Penn Treebank (selections)	LDC	40k words, tagged and parsed
PP Attachment Corpus	Ratnaparkhi	28k prepositional phrases, tagged as noun or verb modifiers
Proposition Bank	Palmer	113k propositions, 3,300 verb frames
Question Classification	Li, Roth	6k questions, categorized
Reuters Corpus	Reuters	1.3M words, 10k news documents, categorized
Roget's Thesaurus	Project Gutenberg	200k words, formatted text
RTE Textual Entailment	Dagan et al.	8k sentence pairs, categorized
SEMCOR	Rus, Mihalcea	880k words, POS and sense tagged
Senseval 2 Corpus	Pedersen	600k words, POS and sense tagged
Shakespeare texts (selections)	Bosak	8 books in XML format
State of the Union Corpus	CSpan	485k words, formatted text
Stopwords Corpus	Porter et al.	2,400 stopwords for 11 languages
Swadesh Corpus	Wiktionary	Comparative wordlists in 24 languages
Switchboard Corpus (selections)	LDC	36 phone calls, transcribed, parsed
TIMIT Corpus (selections)	NIST/LDC	Audio files and transcripts for 16 speakers
Univ Decl of Human Rights	United Nations	480k words, 300+ languages
VerbNet 2.1	Palmer et al.	5k verbs, hierarchically organized, linked to WordNet
Wordlist Corpus	OpenOffice.org et al.	960k words and 20k affixes for 8 languages
WordNet 3.0 (English)	Miller, Fellbaum	145k synonym sets

Corpora in Other Languages

NLTK comes with corpora for many languages, though in some cases you will need to learn how to manipulate character encodings in Python before using these corpora (see Section 3.3).

```
>>> nltk.corpus.cess_esp.words()
['El', 'grupo', 'estatal', 'Electricit\xe9_de_France', ...]
>>> nltk.corpus.floresta.words()
['Um', 'revivalismo', 'refrescante', 'O', '7_e_Meio', ...]
>>> nltk.corpus.indian.words('hindi.pos')
['\xe0\xa4\xaa\xe0\xa5\x82\xe0\xa4\xb0\xe0\xa5\x8d\xe0\xa4\xa3',
'\xe0\xa4\xaa\xe0\xa5\x8d\xe0\xa4\xb0\xe0\xa4\xa4\xe0\xa4\xbf\xe0\xa4\xac\xe0\xa4
\x82\xe0\xa4\xa7', ...]
>>> nltk.corpus.udhr.fileids()
['Abkhaz-Cyrillic+Abkh', 'Abkhaz-UTF8', 'Achehnese-Latin1', 'Achuar-Shiwiar-Latin1',
'Adja-UTF8', 'Afaan_Oromo_Oromiffa-Latin1', 'Afrikaans-Latin1', 'Aguaruna-Latin1',
'Akuapem_Twi-UTF8', 'Albanian_Shqip-Latin1', 'Amahuaca', 'Amahuaca-Latin1', ...]
>>> nltk.corpus.udhr.words('Javanese-Latin1')[11:]
[u'Saben', u'umat', u'manungsa', u'lair', u'kanthi', ...]
```

The last of these corpora, `udhr`, contains the Universal Declaration of Human Rights in over 300 languages. The fileids for this corpus include information about the character encoding used in the file, such as `UTF8` or `Latin1`. Let's use a conditional frequency distribution to examine the differences in word lengths for a selection of languages included in the `udhr` corpus. The output is shown in Figure 2-2 (run the program yourself to see a color plot). Note that `True` and `False` are Python's built-in Boolean values.

```
>>> from nltk.corpus import udhr
>>> languages = ['Chickasaw', 'English', 'German_Deutsch',
...     'Greenlandic_Inuktikut', 'Hungarian_Magyar', 'Ibibio_Efik']
>>> cfd = nltk.ConditionalFreqDist(
...           (lang, len(word))
...           for lang in languages
...           for word in udhr.words(lang + '-Latin1'))
>>> cfd.plot(cumulative=True)
```

Your Turn: Pick a language of interest in `udhr.fileids()`, and define a variable `raw_text = udhr.raw(`*`Language-Latin1`*`)`. Now plot a frequency distribution of the letters of the text using

```
nltk.FreqDist(raw_text).plot().
```

Unfortunately, for many languages, substantial corpora are not yet available. Often there is insufficient government or industrial support for developing language resources, and individual efforts are piecemeal and hard to discover or reuse. Some languages have no established writing system, or are endangered. (See Section 2.7 for suggestions on how to locate language resources.)

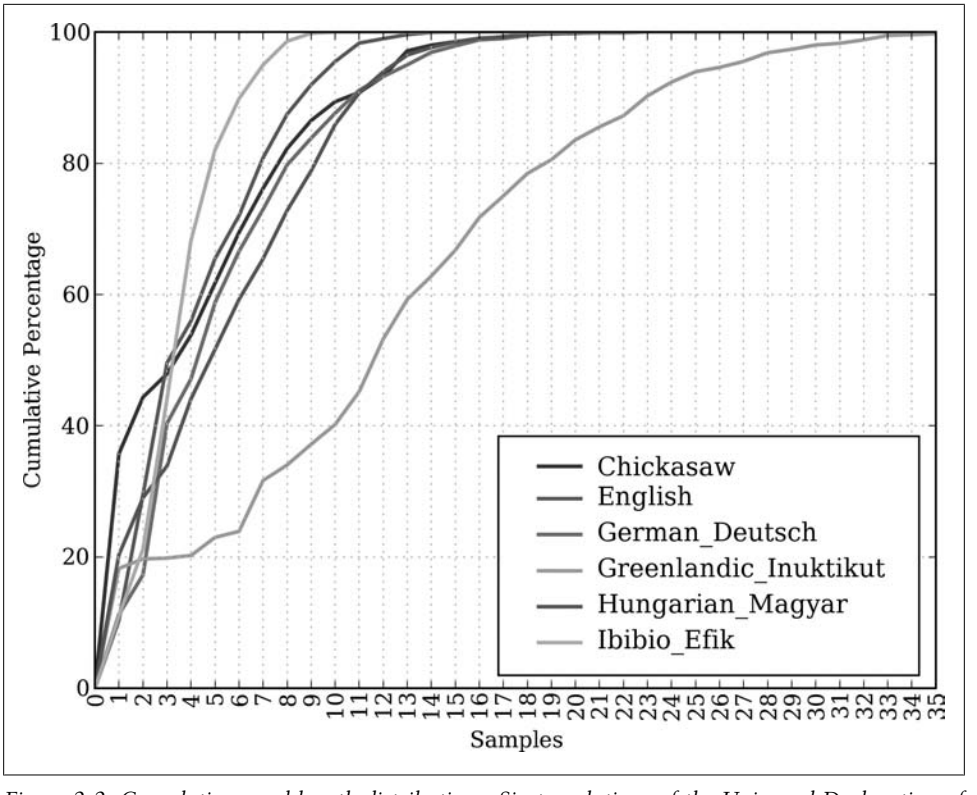


Figure 2-2. Cumulative word length distributions: Six translations of the Universal Declaration of Human Rights are processed; this graph shows that words having five or fewer letters account for about 80% of Ibibio text, 60% of German text, and 25% of Inuktitut text.

Text Corpus Structure

We have seen a variety of corpus structures so far; these are summarized in Figure 2-3. The simplest kind lacks any structure: it is just a collection of texts. Often, texts are grouped into categories that might correspond to genre, source, author, language, etc. Sometimes these categories overlap, notably in the case of topical categories, as a text can be relevant to more than one topic. Occasionally, text collections have temporal structure, news collections being the most common example.

NLTK's corpus readers support efficient access to a variety of corpora, and can be used to work with new corpora. Table 2-3 lists functionality provided by the corpus readers.

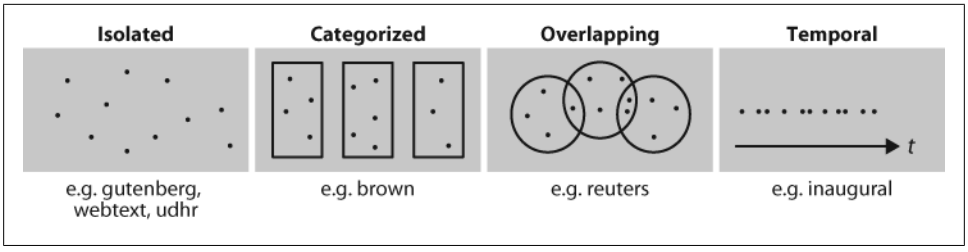


Figure 2-3. Common structures for text corpora: The simplest kind of corpus is a collection of isolated texts with no particular organization; some corpora are structured into categories, such as genre (Brown Corpus); some categorizations overlap, such as topic categories (Reuters Corpus); other corpora represent language use over time (Inaugural Address Corpus).

Table 2-3. Basic corpus functionality defined in NLTK: More documentation can be found using help(nltk.corpus.reader) and by reading the online Corpus HOWTO at http://www.nltk.org/howto.

Example	Description
`fileids()`	The files of the corpus
`fileids([categories])`	The files of the corpus corresponding to these categories
`categories()`	The categories of the corpus
`categories([fileids])`	The categories of the corpus corresponding to these files
`raw()`	The raw content of the corpus
`raw(fileids=[f1,f2,f3])`	The raw content of the specified files
`raw(categories=[c1,c2])`	The raw content of the specified categories
`words()`	The words of the whole corpus
`words(fileids=[f1,f2,f3])`	The words of the specified fileids
`words(categories=[c1,c2])`	The words of the specified categories
`sents()`	The sentences of the specified categories
`sents(fileids=[f1,f2,f3])`	The sentences of the specified fileids
`sents(categories=[c1,c2])`	The sentences of the specified categories
`abspath(fileid)`	The location of the given file on disk
`encoding(fileid)`	The encoding of the file (if known)
`open(fileid)`	Open a stream for reading the given corpus file
`root()`	The path to the root of locally installed corpus
`readme()`	The contents of the README file of the corpus

We illustrate the difference between some of the corpus access methods here:

```
>>> raw = gutenberg.raw("burgess-busterbrown.txt")
>>> raw[1:20]
'The Adventures of B'
>>> words = gutenberg.words("burgess-busterbrown.txt")
>>> words[1:20]
```

```
['The', 'Adventures', 'of', 'Buster', 'Bear', 'by', 'Thornton', 'W', '.',
'Burgess', '1920', ']', 'I', 'BUSTER', 'BEAR', 'GOES', 'FISHING', 'Buster',
'Bear']
>>> sents = gutenberg.sents("burgess-busterbrown.txt")
>>> sents[1:20]
[['I'], ['BUSTER', 'BEAR', 'GOES', 'FISHING'], ['Buster', 'Bear', 'yawned', 'as',
'he', 'lay', 'on', 'his', 'comfortable', 'bed', 'of', 'leaves', 'and', 'watched',
'the', 'first', 'early', 'morning', 'sunbeams', 'creeping', 'through', ...], ...]
```

Loading Your Own Corpus

If you have a your own collection of text files that you would like to access using the methods discussed earlier, you can easily load them with the help of NLTK's `PlaintextCorpusReader`. Check the location of your files on your file system; in the following example, we have taken this to be the directory */usr/share/dict*. Whatever the location, set this to be the value of `corpus_root` ❶. The second parameter of the `PlaintextCorpusReader` initializer ❷ can be a list of fileids, like `['a.txt', 'test/b.txt']`, or a pattern that matches all fileids, like `'[abc]/.*\.txt'` (see Section 3.4 for information about regular expressions).

```
>>> from nltk.corpus import PlaintextCorpusReader
>>> corpus_root = '/usr/share/dict' ❶
>>> wordlists = PlaintextCorpusReader(corpus_root, '.*') ❷
>>> wordlists.fileids()
['README', 'connectives', 'propernames', 'web2', 'web2a', 'words']
>>> wordlists.words('connectives')
['the', 'of', 'and', 'to', 'a', 'in', 'that', 'is', ...]
```

As another example, suppose you have your own local copy of Penn Treebank (release 3), in `C:\corpora`. We can use the `BracketParseCorpusReader` to access this corpus. We specify the `corpus_root` to be the location of the parsed *Wall Street Journal* component of the corpus ❶, and give a `file_pattern` that matches the files contained within its subfolders ❷ (using forward slashes).

```
>>> from nltk.corpus import BracketParseCorpusReader
>>> corpus_root = r"C:\corpora\penntreebank\parsed\mrg\wsj" ❶
>>> file_pattern = r".*/wsj_.*\.mrg" ❷
>>> ptb = BracketParseCorpusReader(corpus_root, file_pattern)
>>> ptb.fileids()
['00/wsj_0001.mrg', '00/wsj_0002.mrg', '00/wsj_0003.mrg', '00/wsj_0004.mrg', ...]
>>> len(ptb.sents())
49208
>>> ptb.sents(fileids='20/wsj_2013.mrg')[19]
['The', '55-year-old', 'Mr.', 'Noriega', 'is', "n't", 'as', 'smooth', 'as', 'the',
'shah', 'of', 'Iran', ',', 'as', 'well-born', 'as', 'Nicaragua', "'s", 'Anastasio',
'Somoza', ',', 'as', 'imperial', 'as', 'Ferdinand', 'Marcos', 'of', 'the', 'Philippines',
'or', 'as', 'bloody', 'as', 'Haiti', "'s", 'Baby', Doc', 'Duvalier', '.']
```

2.2 Conditional Frequency Distributions

We introduced frequency distributions in Section 1.3. We saw that given some list `mylist` of words or other items, `FreqDist(mylist)` would compute the number of occurrences of each item in the list. Here we will generalize this idea.

When the texts of a corpus are divided into several categories (by genre, topic, author, etc.), we can maintain separate frequency distributions for each category. This will allow us to study systematic differences between the categories. In the previous section, we achieved this using NLTK's `ConditionalFreqDist` data type. A **conditional frequency distribution** is a collection of frequency distributions, each one for a different "condition." The condition will often be the category of the text. Figure 2-4 depicts a fragment of a conditional frequency distribution having just two conditions, one for news text and one for romance text.

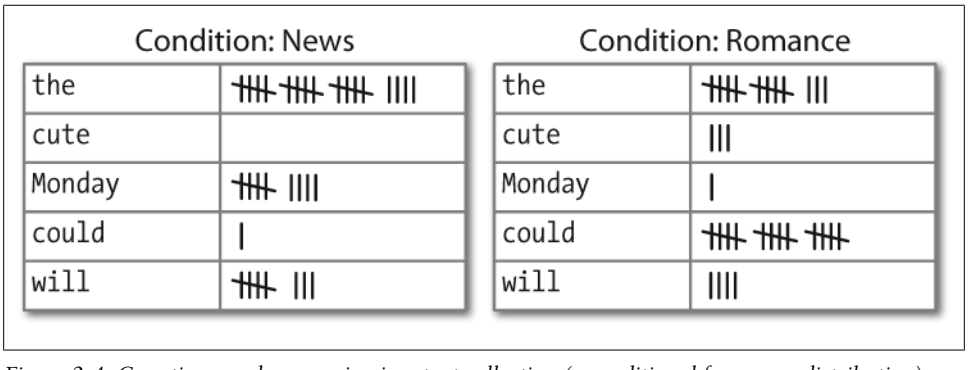


Figure 2-4. Counting words appearing in a text collection (a conditional frequency distribution).

Conditions and Events

A frequency distribution counts observable events, such as the appearance of words in a text. A conditional frequency distribution needs to pair each event with a condition. So instead of processing a sequence of words ❶, we have to process a sequence of pairs ❷:

```
>>> text = ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...] ❶
>>> pairs = [('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ...] ❷
```

Each pair has the form (*condition*, *event*). If we were processing the entire Brown Corpus by genre, there would be 15 conditions (one per genre) and 1,161,192 events (one per word).

Counting Words by Genre

In Section 2.1, we saw a conditional frequency distribution where the condition was the section of the Brown Corpus, and for each condition we counted words. Whereas `FreqDist()` takes a simple list as input, `ConditionalFreqDist()` takes a list of pairs.

```
>>> from nltk.corpus import brown
>>> cfd = nltk.ConditionalFreqDist(
...           (genre, word)
...           for genre in brown.categories()
...           for word in brown.words(categories=genre))
```

Let's break this down, and look at just two genres, news and romance. For each genre ❷, we loop over every word in the genre ❸, producing pairs consisting of the genre and the word ❶:

```
>>> genre_word = [(genre, word) ❶
...               for genre in ['news', 'romance'] ❷
...               for word in brown.words(categories=genre)] ❸
>>> len(genre_word)
170576
```

So, as we can see in the following code, pairs at the beginning of the list `genre_word` will be of the form (`'news'`, *word*) ❶, whereas those at the end will be of the form (`'roman ce'`, *word*) ❷.

```
>>> genre_word[:4]
[('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ('news', 'Grand')] ❶
>>> genre_word[-4:]
[('romance', 'afraid'), ('romance', 'not'), ('romance', "''"), ('romance', '.')] ❷
```

We can now use this list of pairs to create a `ConditionalFreqDist`, and save it in a variable `cfd`. As usual, we can type the name of the variable to inspect it ❶, and verify it has two conditions ❷:

```
>>> cfd = nltk.ConditionalFreqDist(genre_word)
>>> cfd ❶
<ConditionalFreqDist with 2 conditions>
>>> cfd.conditions()
['news', 'romance'] ❷
```

Let's access the two conditions, and satisfy ourselves that each is just a frequency distribution:

```
>>> cfd['news']
<FreqDist with 100554 outcomes>
>>> cfd['romance']
<FreqDist with 70022 outcomes>
>>> list(cfd['romance'])
[',', '.', 'the', 'and', 'to', 'a', 'of', '``', "''", 'was', 'I', 'in', 'he', 'had',
'?', 'her', 'that', 'it', 'his', 'she', 'with', 'you', 'for', 'at', 'He', 'on', 'him',
'said', '!', '--', 'be', 'as', ';', 'have', 'but', 'not', 'would', 'She', 'The', ...]
>>> cfd['romance']['could']
193
```

Plotting and Tabulating Distributions

Apart from combining two or more frequency distributions, and being easy to initialize, a `ConditionalFreqDist` provides some useful methods for tabulation and plotting.

The plot in Figure 2-1 was based on a conditional frequency distribution reproduced in the following code. The condition is either of the words *america* or *citizen* ❷, and the counts being plotted are the number of times the word occurred in a particular speech. It exploits the fact that the filename for each speech—for example, *1865-Lincoln.txt*—contains the year as the first four characters ❶. This code generates the pair `('america', '1865')` for every instance of a word whose lowercased form starts with *america*—such as *Americans*—in the file *1865-Lincoln.txt*.

```
>>> from nltk.corpus import inaugural
>>> cfd = nltk.ConditionalFreqDist(
...           (target, fileid[:4]) ❶
...           for fileid in inaugural.fileids()
...           for w in inaugural.words(fileid)
...           for target in ['america', 'citizen'] ❷
...           if w.lower().startswith(target))
```

The plot in Figure 2-2 was also based on a conditional frequency distribution, reproduced in the following code. This time, the condition is the name of the language, and the counts being plotted are derived from word lengths ❶. It exploits the fact that the filename for each language is the language name followed by `'-Latin1'` (the character encoding).

```
>>> from nltk.corpus import udhr
>>> languages = ['Chickasaw', 'English', 'German_Deutsch',
...     'Greenlandic_Inuktikut', 'Hungarian_Magyar', 'Ibibio_Efik']
>>> cfd = nltk.ConditionalFreqDist(
...           (lang, len(word)) ❶
...           for lang in languages
...           for word in udhr.words(lang + '-Latin1'))
```

In the `plot()` and `tabulate()` methods, we can optionally specify which conditions to display with a `conditions=` parameter. When we omit it, we get all the conditions. Similarly, we can limit the samples to display with a `samples=` parameter. This makes it possible to load a large quantity of data into a conditional frequency distribution, and then to explore it by plotting or tabulating selected conditions and samples. It also gives us full control over the order of conditions and samples in any displays. For example, we can tabulate the cumulative frequency data just for two languages, and for words less than 10 characters long, as shown next. We interpret the last cell on the top row to mean that 1,638 words of the English text have nine or fewer letters.

```
>>> cfd.tabulate(conditions=['English', 'German_Deutsch'],
...              samples=range(10), cumulative=True)
                  0    1    2    3    4    5    6    7    8    9
       English    0  185  525  883  997 1166 1283 1440 1558 1638
German_Deutsch    0  171  263  614  717  894 1013 1110 1213 1275
```

Your Turn: Working with the news and romance genres from the Brown Corpus, find out which days of the week are most newsworthy, and which are most romantic. Define a variable called `days` containing a list of days of the week, i.e., `['Monday', ...]`. Now tabulate the counts for these words using `cfd.tabulate(samples=days)`. Now try the same thing using `plot` in place of `tabulate`. You may control the output order of days with the help of an extra parameter: `conditions=['Monday', ...]`.

You may have noticed that the multiline expressions we have been using with conditional frequency distributions look like list comprehensions, but without the brackets. In general, when we use a list comprehension as a parameter to a function, like `set([w.lower for w in t])`, we are permitted to omit the square brackets and just write `set(w.lower() for w in t)`. (See the discussion of "generator expressions" in Section 4.2 for more about this.)

Generating Random Text with Bigrams

We can use a conditional frequency distribution to create a table of bigrams (word pairs, introduced in Section 1.3). The `bigrams()` function takes a list of words and builds a list of consecutive word pairs:

```
>>> sent = ['In', 'the', 'beginning', 'God', 'created', 'the', 'heaven',
...   'and', 'the', 'earth', '.']
>>> nltk.bigrams(sent)
[('In', 'the'), ('the', 'beginning'), ('beginning', 'God'), ('God', 'created'),
('created', 'the'), ('the', 'heaven'), ('heaven', 'and'), ('and', 'the'),
('the', 'earth'), ('earth', '.')]
```

In Example 2-1, we treat each word as a condition, and for each one we effectively create a frequency distribution over the following words. The function `generate_model()` contains a simple loop to generate text. When we call the function, we choose a word (such as `'living'`) as our initial context. Then, once inside the loop, we print the current value of the variable `word`, and reset `word` to be the most likely token in that context (using `max()`); next time through the loop, we use that word as our new context. As you can see by inspecting the output, this simple approach to text generation tends to get stuck in loops. Another method would be to randomly choose the next word from among the available words.

Example 2-1. Generating random text: This program obtains all bigrams from the text of the book of Genesis, then constructs a conditional frequency distribution to record which words are most likely to follow a given word; e.g., after the word living, *the most likely word is* creature; *the* `generate_model()` *function uses this data, and a seed word, to generate random text.*

```
def generate_model(cfdist, word, num=15):
    for i in range(num):
        print word,
        word = cfdist[word].max()
```

```
text = nltk.corpus.genesis.words('english-kjv.txt')
bigrams = nltk.bigrams(text)
cfd = nltk.ConditionalFreqDist(bigrams) ❶
```

```
>>> print cfd['living']
<FreqDist: 'creature': 7, 'thing': 4, 'substance': 2, ',': 1, '.': 1, 'soul': 1>
>>> generate_model(cfd, 'living')
living creature that he said , and the land of the land of the land
```

Conditional frequency distributions are a useful data structure for many NLP tasks. Their commonly used methods are summarized in Table 2-4.

Table 2-4. NLTK's conditional frequency distributions: Commonly used methods and idioms for defining, accessing, and visualizing a conditional frequency distribution of counters

Example	Description
`cfdist = ConditionalFreqDist(pairs)`	Create a conditional frequency distribution from a list of pairs
`cfdist.conditions()`	Alphabetically sorted list of conditions
`cfdist[condition]`	The frequency distribution for this condition
`cfdist[condition][sample]`	Frequency for the given sample for this condition
`cfdist.tabulate()`	Tabulate the conditional frequency distribution
`cfdist.tabulate(samples, conditions)`	Tabulation limited to the specified samples and conditions
`cfdist.plot()`	Graphical plot of the conditional frequency distribution
`cfdist.plot(samples, conditions)`	Graphical plot limited to the specified samples and conditions
`cfdist1 < cfdist2`	Test if samples in `cfdist1` occur less frequently than in `cfdist2`

2.3 More Python: Reusing Code

By this time you've probably typed and retyped a lot of code in the Python interactive interpreter. If you mess up when retyping a complex example, you have to enter it again. Using the arrow keys to access and modify previous commands is helpful but only goes so far. In this section, we see two important ways to reuse code: text editors and Python functions.

Creating Programs with a Text Editor

The Python interactive interpreter performs your instructions as soon as you type them. Often, it is better to compose a multiline program using a text editor, then ask Python to run the whole program at once. Using IDLE, you can do this by going to the File menu and opening a new window. Try this now, and enter the following one-line program:

```
print 'Monty Python'
```

Save this program in a file called *monty.py*, then go to the Run menu and select the command Run Module. (We'll learn what modules are shortly.) The result in the main IDLE window should look like this:

```
>>> ================================ RESTART ================================
>>>
Monty Python
>>>
```

You can also type `from monty import *` and it will do the same thing.

From now on, you have a choice of using the interactive interpreter or a text editor to create your programs. It is often convenient to test your ideas using the interpreter, revising a line of code until it does what you expect. Once you're ready, you can paste the code (minus any `>>>` or `...` prompts) into the text editor, continue to expand it, and finally save the program in a file so that you don't have to type it in again later. Give the file a short but descriptive name, using all lowercase letters and separating words with underscore, and using the *.py* filename extension, e.g., *monty_python.py*.

Important: Our inline code examples include the `>>>` and `...` prompts as if we are interacting directly with the interpreter. As they get more complicated, you should instead type them into the editor, without the prompts, and run them from the editor as shown earlier. When we provide longer programs in this book, we will leave out the prompts to remind you to type them into a file rather than using the interpreter. You can see this already in Example 2-1. Note that the example still includes a couple of lines with the Python prompt; this is the interactive part of the task where you inspect some data and invoke a function. Remember that all code samples like Example 2-1 are downloadable from *http://www.nltk.org/*.

Functions

Suppose that you work on analyzing text that involves different forms of the same word, and that part of your program needs to work out the plural form of a given singular noun. Suppose it needs to do this work in two places, once when it is processing some texts and again when it is processing user input.

Rather than repeating the same code several times over, it is more efficient and reliable to localize this work inside a **function**. A function is just a named block of code that performs some well-defined task, as we saw in Section 1.1. A function is usually defined to take some inputs, using special variables known as **parameters**, and it may produce a result, also known as a **return value**. We define a function using the keyword `def` followed by the function name and any input parameters, followed by the body of the function. Here's the function we saw in Section 1.1 (including the `import` statement that makes division behave as expected):

```
>>> from __future__ import division
>>> def lexical_diversity(text):
...     return len(text) / len(set(text))
```

We use the keyword `return` to indicate the value that is produced as output by the function. In this example, all the work of the function is done in the `return` statement. Here's an equivalent definition that does the same work using multiple lines of code. We'll change the parameter name from `text` to `my_text_data` to remind you that this is an arbitrary choice:

```
>>> def lexical_diversity(my_text_data):
...     word_count = len(my_text_data)
...     vocab_size = len(set(my_text_data))
...     diversity_score = word_count / vocab_size
...     return diversity_score
```

Notice that we've created some new variables inside the body of the function. These are **local variables** and are not accessible outside the function. So now we have defined a function with the name `lexical_diversity`. But just defining it won't produce any output! Functions do nothing until they are "called" (or "invoked").

Let's return to our earlier scenario, and actually define a simple function to work out English plurals. The function `plural()` in Example 2-2 takes a singular noun and generates a plural form, though it is not always correct. (We'll discuss functions at greater length in Section 4.4.)

Example 2-2. A Python function: This function tries to work out the plural form of any English noun; the keyword def (define) is followed by the function name, then a parameter inside parentheses, and a colon; the body of the function is the indented block of code; it tries to recognize patterns within the word and process the word accordingly; e.g., if the word ends with y, delete the y and add ies.

```
def plural(word):
    if word.endswith('y'):
        return word[:-1] + 'ies'
    elif word[-1] in 'sx' or word[-2:] in ['sh', 'ch']:
        return word + 'es'
    elif word.endswith('an'):
        return word[:-2] + 'en'
    else:
        return word + 's'
```

```
>>> plural('fairy')
'fairies'
>>> plural('woman')
'women'
```

The `endswith()` function is always associated with a string object (e.g., `word` in Example 2-2). To call such functions, we give the name of the object, a period, and then the name of the function. These functions are usually known as **methods**.

Modules

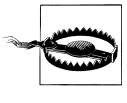
Over time you will find that you create a variety of useful little text-processing functions, and you end up copying them from old programs to new ones. Which file contains the latest version of the function you want to use? It makes life a lot easier if you can collect your work into a single place, and access previously defined functions without making copies.

To do this, save your function(s) in a file called (say) *textproc.py*. Now, you can access your work simply by importing it from the file:

```
>>> from textproc import plural
>>> plural('wish')
wishes
>>> plural('fan')
fen
```

Our plural function obviously has an error, since the plural of *fan* is *fans*. Instead of typing in a new version of the function, we can simply edit the existing one. Thus, at every stage, there is only one version of our plural function, and no confusion about which one is being used.

A collection of variable and function definitions in a file is called a Python **module**. A collection of related modules is called a **package**. NLTK's code for processing the Brown Corpus is an example of a module, and its collection of code for processing all the different corpora is an example of a package. NLTK itself is a set of packages, sometimes called a **library**.



Caution!

If you are creating a file to contain some of your Python code, do *not* name your file *nltk.py*: it may get imported in place of the "real" NLTK package. When it imports modules, Python first looks in the current directory (folder).

2.4 Lexical Resources

A lexicon, or lexical resource, is a collection of words and/or phrases along with associated information, such as part-of-speech and sense definitions. Lexical resources are secondary to texts, and are usually created and enriched with the help of texts. For example, if we have defined a text `my_text`, then `vocab = sorted(set(my_text))` builds the vocabulary of `my_text`, whereas `word_freq = FreqDist(my_text)` counts the frequency of each word in the text. Both `vocab` and `word_freq` are simple lexical resources. Similarly, a concordance like the one we saw in Section 1.1 gives us information about word usage that might help in the preparation of a dictionary. Standard terminology for lexicons is illustrated in Figure 2-5. A **lexical entry** consists of a **headword** (also known as a **lemma**) along with additional information, such as the part-of-speech and

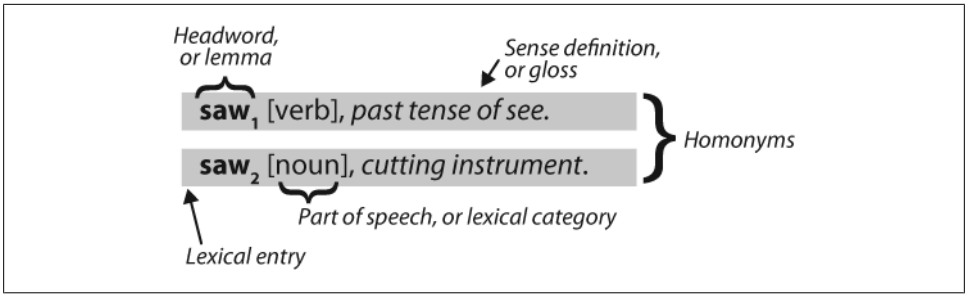


Figure 2-5. Lexicon terminology: Lexical entries for two lemmas having the same spelling (homonyms), providing part-of-speech and gloss information.

the sense definition. Two distinct words having the same spelling are called **homonyms**.

The simplest kind of lexicon is nothing more than a sorted list of words. Sophisticated lexicons include complex structure within and across the individual entries. In this section, we'll look at some lexical resources included with NLTK.

Wordlist Corpora

NLTK includes some corpora that are nothing more than wordlists. The Words Corpus is the */usr/dict/words* file from Unix, used by some spellcheckers. We can use it to find unusual or misspelled words in a text corpus, as shown in Example 2-3.

Example 2-3. Filtering a text: This program computes the vocabulary of a text, then removes all items that occur in an existing wordlist, leaving just the uncommon or misspelled words.

```
def unusual_words(text):
    text_vocab = set(w.lower() for w in text if w.isalpha())
    english_vocab = set(w.lower() for w in nltk.corpus.words.words())
    unusual = text_vocab.difference(english_vocab)
    return sorted(unusual)
```

```
>>> unusual_words(nltk.corpus.gutenberg.words('austen-sense.txt'))
['abbeyland', 'abhorrence', 'abominably', 'abridgement', 'accordant', 'accustomary',
'adieus', 'affability', 'affectedly', 'aggrandizement', 'alighted', 'allenham',
'amiably', 'annamaria', 'annuities', 'apologising', 'arbour', 'archness', ...]
>>> unusual_words(nltk.corpus.nps_chat.words())
['aaaaaaaaaaaaaaaaa', 'aaahhhh', 'abou', 'abourted', 'abs', 'ack', 'acros',
'actualy', 'adduser', 'addy', 'adoted', 'adreniline', 'ae', 'afe', 'affari', 'afk',
'agaibn', 'agurlwithbigguns', 'ahah', 'ahahah', 'ahahh', 'ahahha', 'ahem', 'ahh', ...]
```

There is also a corpus of **stopwords**, that is, high-frequency words such as *the*, *to*, and *also* that we sometimes want to filter out of a document before further processing. Stopwords usually have little lexical content, and their presence in a text fails to distinguish it from other texts.

```
>>> from nltk.corpus import stopwords
>>> stopwords.words('english')
['a', "a's", 'able', 'about', 'above', 'according', 'accordingly', 'across',
```

```
'actually', 'after', 'afterwards', 'again', 'against', "ain't", 'all', 'allow',
'allows', 'almost', 'alone', 'along', 'already', 'also', 'although', 'always', ...]
```

Let's define a function to compute what fraction of words in a text are *not* in the stopwords list:

```
>>> def content_fraction(text):
...     stopwords = nltk.corpus.stopwords.words('english')
...     content = [w for w in text if w.lower() not in stopwords]
...     return len(content) / len(text)
...
>>> content_fraction(nltk.corpus.reuters.words())
0.65997695393285261
```

Thus, with the help of stopwords, we filter out a third of the words of the text. Notice that we've combined two different kinds of corpus here, using a lexical resource to filter the content of a text corpus.

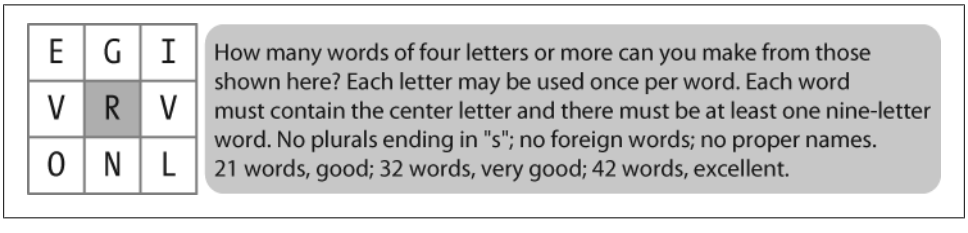


Figure 2-6. A word puzzle: A grid of randomly chosen letters with rules for creating words out of the letters; this puzzle is known as "Target."

A wordlist is useful for solving word puzzles, such as the one in Figure 2-6. Our program iterates through every word and, for each one, checks whether it meets the conditions. It is easy to check obligatory letter ❷ and length ❶ constraints (and we'll only look for words with six or more letters here). It is trickier to check that candidate solutions only use combinations of the supplied letters, especially since some of the supplied letters appear twice (here, the letter *v*). The `FreqDist` comparison method ❸ permits us to check that the frequency of each *letter* in the candidate word is less than or equal to the frequency of the corresponding letter in the puzzle.

```
>>> puzzle_letters = nltk.FreqDist('egivrvonl')
>>> obligatory = 'r'
>>> wordlist = nltk.corpus.words.words()
>>> [w for w in wordlist if len(w) >= 6 ❶
...                      and obligatory in w ❷
...                      and nltk.FreqDist(w) <= puzzle_letters] ❸
['glover', 'gorlin', 'govern', 'grovel', 'ignore', 'involver', 'lienor',
'linger', 'longer', 'lovering', 'noiler', 'overling', 'region', 'renvoi',
'revolving', 'ringle', 'roving', 'violer', 'virole']
```

One more wordlist corpus is the Names Corpus, containing 8,000 first names categorized by gender. The male and female names are stored in separate files. Let's find names that appear in both files, i.e., names that are ambiguous for gender:

```
>>> names = nltk.corpus.names
>>> names.fileids()
['female.txt', 'male.txt']
>>> male_names = names.words('male.txt')
>>> female_names = names.words('female.txt')
>>> [w for w in male_names if w in female_names]
['Abbey', 'Abbie', 'Abby', 'Addie', 'Adrian', 'Adrien', 'Ajay', 'Alex', 'Alexis',
'Alfie', 'Ali', 'Alix', 'Allie', 'Allyn', 'Andie', 'Andrea', 'Andy', 'Angel',
'Angie', 'Ariel', 'Ashley', 'Aubrey', 'Augustine', 'Austin', 'Averil', ...]
```

It is well known that names ending in the letter *a* are almost always female. We can see this and some other patterns in the graph in Figure 2-7, produced by the following code. Remember that `name[-1]` is the last letter of `name`.

```
>>> cfd = nltk.ConditionalFreqDist(
...           (fileid, name[-1])
...           for fileid in names.fileids()
...           for name in names.words(fileid))
>>> cfd.plot()
```

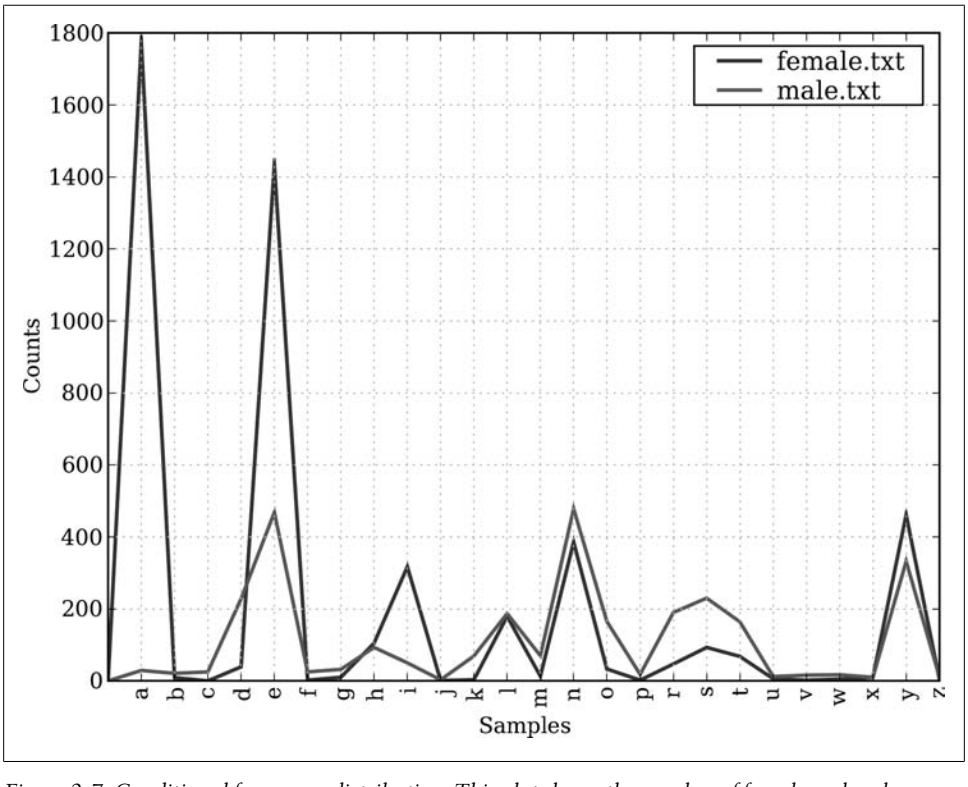


Figure 2-7. Conditional frequency distribution: This plot shows the number of female and male names ending with each letter of the alphabet; most names ending with a, e, *or* i *are female; names ending in* h *and* l *are equally likely to be male or female; names ending in* k, o, r, s, *and* t *are likely to be male.*

A Pronouncing Dictionary

A slightly richer kind of lexical resource is a table (or spreadsheet), containing a word plus some properties in each row. NLTK includes the CMU Pronouncing Dictionary for U.S. English, which was designed for use by speech synthesizers.

```
>>> entries = nltk.corpus.cmudict.entries()
>>> len(entries)
127012
>>> for entry in entries[39943:39951]:
...     print entry
...
('fir', ['F', 'ER1'])
('fire', ['F', 'AY1', 'ER0'])
('fire', ['F', 'AY1', 'R'])
('firearm', ['F', 'AY1', 'ER0', 'AA2', 'R', 'M'])
('firearm', ['F', 'AY1', 'R', 'AA2', 'R', 'M'])
('firearms', ['F', 'AY1', 'ER0', 'AA2', 'R', 'M', 'Z'])
('firearms', ['F', 'AY1', 'R', 'AA2', 'R', 'M', 'Z'])
('fireball', ['F', 'AY1', 'ER0', 'B', 'AO2', 'L'])
```

For each word, this lexicon provides a list of phonetic codes—distinct labels for each contrastive sound—known as *phones*. Observe that *fire* has two pronunciations (in U.S. English): the one-syllable `F AY1 R`, and the two-syllable `F AY1 ER0`. The symbols in the CMU Pronouncing Dictionary are from the *Arpabet*, described in more detail at *http://en.wikipedia.org/wiki/Arpabet*.

Each entry consists of two parts, and we can process these individually using a more complex version of the `for` statement. Instead of writing `for entry in entries:`, we replace `entry` with *two* variable names, `word, pron` ❶. Now, each time through the loop, `word` is assigned the first part of the entry, and `pron` is assigned the second part of the entry:

```
>>> for word, pron in entries: ❶
...     if len(pron) == 3: ❷
...         ph1, ph2, ph3 = pron ❸
...         if ph1 == 'P' and ph3 == 'T':
...             print word, ph2,
...
pait EY1 pat AE1 pate EY1 patt AE1 peart ER1 peat IY1 peet IY1 peete IY1 pert ER1
pet EH1 pete IY1 pett EH1 piet IY1 piette IY1 pit IH1 pitt IH1 pot AA1 pote OW1
pott AA1 pout AW1 puett UW1 purt ER1 put UH1 putt AH1
```

The program just shown scans the lexicon looking for entries whose pronunciation consists of three phones ❷. If the condition is true, it assigns the contents of `pron` to three new variables: `ph1`, `ph2`, and `ph3`. Notice the unusual form of the statement that does that work ❸.

Here's another example of the same `for` statement, this time used inside a list comprehension. This program finds all words whose pronunciation ends with a syllable sounding like *nicks*. You could use this method to find rhyming words.

```
>>> syllable = ['N', 'IH0', 'K', 'S']
>>> [word for word, pron in entries if pron[-4:] == syllable]
["atlantic's", 'audiotronics', 'avionics', 'beatniks', 'calisthenics', 'centronics',
'chetniks', "clinic's", 'clinics', 'conics', 'cynics', 'diasonics', "dominic's",
'ebonics', 'electronics', "electronics'", 'endotronics', "endotronics'", 'enix', ...]
```

Notice that the one pronunciation is spelled in several ways: *nics*, *niks*, *nix*, and even *ntic's* with a silent *t*, for the word *atlantic's*. Let's look for some other mismatches between pronunciation and writing. Can you summarize the purpose of the following examples and explain how they work?

```
>>> [w for w, pron in entries if pron[-1] == 'M' and w[-1] == 'n']
['autumn', 'column', 'condemn', 'damn', 'goddamn', 'hymn', 'solemn']
>>> sorted(set(w[:2] for w, pron in entries if pron[0] == 'N' and w[0] != 'n'))
['gn', 'kn', 'mn', 'pn']
```

The phones contain digits to represent primary stress (`1`), secondary stress (`2`), and no stress (`0`). As our final example, we define a function to extract the stress digits and then scan our lexicon to find words having a particular stress pattern.

```
>>> def stress(pron):
...     return [char for phone in pron for char in phone if char.isdigit()]
>>> [w for w, pron in entries if stress(pron) == ['0', '1', '0', '2', '0']]
['abbreviated', 'abbreviating', 'accelerated', 'accelerating', 'accelerator',
'accentuated', 'accentuating', 'accommodated', 'accommodating', 'accommodative',
'accumulated', 'accumulating', 'accumulative', 'accumulator', 'accumulators', ...]
>>> [w for w, pron in entries if stress(pron) == ['0', '2', '0', '1', '0']]
['abbreviation', 'abbreviations', 'abomination', 'abortifacient', 'abortifacients',
'academicians', 'accommodation', 'accommodations', 'accreditation', 'accreditations',
'accumulation', 'accumulations', 'acetylcholine', 'acetylcholine', 'adjudication', ...]
```

A subtlety of this program is that our user-defined function `stress()` is invoked inside the condition of a list comprehension. There is also a doubly nested `for` loop. There's a lot going on here, and you might want to return to this once you've had more experience using list comprehensions.

We can use a conditional frequency distribution to help us find minimally contrasting sets of words. Here we find all the *p* words consisting of three sounds ❷, and group them according to their first and last sounds ❶.

```
>>> p3 = [(pron[0]+'-'+pron[2], word)                    ❶
...       for (word, pron) in entries
...       if pron[0] == 'P' and len(pron) == 3]          ❷
>>> cfd = nltk.ConditionalFreqDist(p3)
>>> for template in cfd.conditions():
...     if len(cfd[template]) > 10:
...         words = cfd[template].keys()
...         wordlist = ' '.join(words)
...         print template, wordlist[:70] + "..."
...
P-CH perch puche poche peach petsche poach pietsch putsch pautsch piche pet...
```

```
P-K pik peek pic pique paque polk perc poke perk pac pock poch purk pak pa...
P-L pil poehl pille pehl pol pall pohl pahl paul perl pale paille perle po...
P-N paine payne pon pain pin pawn pinn pun pine paign pen pyne pane penn p...
P-P pap paap pipp paup pape pup pep poop pop pipe paape popp pip peep pope...
P-R paar poor par poore pear pare pour peer pore parr por pair porr pier...
P-S pearse piece posts pasts peace perce pos pers pace puss pesce pass pur...
P-T pot puett pit pete putt pat purt pet peart pott pett pait pert pote pa...
P-Z pays p.s pao's pais paws p.'s pas pez paz pei's pose poise peas paiz p...
```

Rather than iterating over the whole dictionary, we can also access it by looking up particular words. We will use Python's dictionary data structure, which we will study systematically in Section 5.3. We look up a dictionary by specifying its name, followed by a **key** (such as the word `'fire'`) inside square brackets ❶.

```
>>> prondict = nltk.corpus.cmudict.dict()
>>> prondict['fire'] ❶
[['F', 'AY1', 'ER0'], ['F', 'AY1', 'R']]
>>> prondict['blog'] ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'blog'
>>> prondict['blog'] = [['B', 'L', 'AA1', 'G']] ❸
>>> prondict['blog']
[['B', 'L', 'AA1', 'G']]
```

If we try to look up a non-existent key ❷, we get a `KeyError`. This is similar to what happens when we index a list with an integer that is too large, producing an `IndexError`. The word *blog* is missing from the pronouncing dictionary, so we tweak our version by assigning a value for this key ❸ (this has no effect on the NLTK corpus; next time we access it, *blog* will still be absent).

We can use any lexical resource to process a text, e.g., to filter out words having some lexical property (like nouns), or mapping every word of the text. For example, the following text-to-speech function looks up each word of the text in the pronunciation dictionary:

```
>>> text = ['natural', 'language', 'processing']
>>> [ph for w in text for ph in prondict[w][0]]
['N', 'AE1', 'CH', 'ER0', 'AH0', 'L', 'L', 'AE1', 'NG', 'G', 'W', 'AH0', 'JH',
'P', 'R', 'AA1', 'S', 'EH0', 'S', 'IH0', 'NG']
```

Comparative Wordlists

Another example of a tabular lexicon is the **comparative wordlist**. NLTK includes so-called **Swadesh wordlists**, lists of about 200 common words in several languages. The languages are identified using an ISO 639 two-letter code.

```
>>> from nltk.corpus import swadesh
>>> swadesh.fileids()
['be', 'bg', 'bs', 'ca', 'cs', 'cu', 'de', 'en', 'es', 'fr', 'hr', 'it', 'la', 'mk',
'nl', 'pl', 'pt', 'ro', 'ru', 'sk', 'sl', 'sr', 'sw', 'uk']
>>> swadesh.words('en')
['I', 'you (singular), thou', 'he', 'we', 'you (plural)', 'they', 'this', 'that',
```

```
'here', 'there', 'who', 'what', 'where', 'when', 'how', 'not', 'all', 'many', 'some',
'few', 'other', 'one', 'two', 'three', 'four', 'five', 'big', 'long', 'wide', ...]
```

We can access cognate words from multiple languages using the `entries()` method, specifying a list of languages. With one further step we can convert this into a simple dictionary (we'll learn about `dict()` in Section 5.3).

```
>>> fr2en = swadesh.entries(['fr', 'en'])
>>> fr2en
[('je', 'I'), ('tu, vous', 'you (singular), thou'), ('il', 'he'), ...]
>>> translate = dict(fr2en)
>>> translate['chien']
'dog'
>>> translate['jeter']
'throw'
```

We can make our simple translator more useful by adding other source languages. Let's get the German-English and Spanish-English pairs, convert each to a dictionary using `dict()`, then *update* our original `translate` dictionary with these additional mappings:

```
>>> de2en = swadesh.entries(['de', 'en'])    # German-English
>>> es2en = swadesh.entries(['es', 'en'])    # Spanish-English
>>> translate.update(dict(de2en))
>>> translate.update(dict(es2en))
>>> translate['Hund']
'dog'
>>> translate['perro']
'dog'
```

We can compare words in various Germanic and Romance languages:

```
>>> languages = ['en', 'de', 'nl', 'es', 'fr', 'pt', 'la']
>>> for i in [139, 140, 141, 142]:
...     print swadesh.entries(languages)[i]
...
('say', 'sagen', 'zeggen', 'decir', 'dire', 'dizer', 'dicere')
('sing', 'singen', 'zingen', 'cantar', 'chanter', 'cantar', 'canere')
('play', 'spielen', 'spelen', 'jugar', 'jouer', 'jogar, brincar', 'ludere')
('float', 'schweben', 'zweven', 'flotar', 'flotter', 'flutuar, boiar', 'fluctuare')
```

Shoebox and Toolbox Lexicons

Perhaps the single most popular tool used by linguists for managing data is *Toolbox*, previously known as *Shoebox* since it replaces the field linguist's traditional shoebox full of file cards. Toolbox is freely downloadable from *http://www.sil.org/computing/toolbox/*.

A Toolbox file consists of a collection of entries, where each entry is made up of one or more fields. Most fields are optional or repeatable, which means that this kind of lexical resource cannot be treated as a table or spreadsheet.

Here is a dictionary for the Rotokas language. We see just the first entry, for the word *kaa*, meaning "to gag":

```
>>> from nltk.corpus import toolbox
>>> toolbox.entries('rotokas.dic')
[('kaa', [('ps', 'V'), ('pt', 'A'), ('ge', 'gag'), ('tkp', 'nek i pas'),
('dcsv', 'true'), ('vx', '1'), ('sc', '???'), ('dt', '29/Oct/2005'),
('ex', 'Apoka ira kaaroi aioa-ia reoreopaoro.'),
('xp', 'Kaikai i pas long nek bilong Apoka bikos em i kaikai na toktok.'),
('xe', 'Apoka is gagging from food while talking.')]), ...]
```

Entries consist of a series of attribute-value pairs, such as `('ps', 'V')` to indicate that the part-of-speech is `'V'` (verb), and `('ge', 'gag')` to indicate that the gloss-into-English is `'gag'`. The last three pairs contain an example sentence in Rotokas and its translations into Tok Pisin and English.

The loose structure of Toolbox files makes it hard for us to do much more with them at this stage. XML provides a powerful way to process this kind of corpus, and we will return to this topic in Chapter 11.

The Rotokas language is spoken on the island of Bougainville, Papua New Guinea. This lexicon was contributed to NLTK by Stuart Robinson. Rotokas is notable for having an inventory of just 12 phonemes (contrastive sounds); see *http://en.wikipedia.org/wiki/Rotokas_language*

2.5 WordNet

WordNet is a semantically oriented dictionary of English, similar to a traditional thesaurus but with a richer structure. NLTK includes the English WordNet, with 155,287 words and 117,659 synonym sets. We'll begin by looking at synonyms and how they are accessed in WordNet.

Senses and Synonyms

Consider the sentence in (1a). If we replace the word *motorcar* in (1a) with *automobile*, to get (1b), the meaning of the sentence stays pretty much the same:

(1) a. Benz is credited with the invention of the motorcar.
b. Benz is credited with the invention of the automobile.

Since everything else in the sentence has remained unchanged, we can conclude that the words *motorcar* and *automobile* have the same meaning, i.e., they are **synonyms**. We can explore these words with the help of WordNet:

```
>>> from nltk.corpus import wordnet as wn
>>> wn.synsets('motorcar')
[Synset('car.n.01')]
```

Thus, *motorcar* has just one possible meaning and it is identified as `car.n.01`, the first noun sense of *car*. The entity `car.n.01` is called a **synset**, or "synonym set," a collection of synonymous words (or "lemmas"):