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 tackle 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. is the Python programming language suitable 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 understanding 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/.

1.1 Computing with Language: Texts and Words

We're all very familiar with text, since we read and write it every day. But 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. Before we can do this, we have to get started with the Python interpreter.

Getting Started

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 greater (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 be sure not to type in the `>>>` prompt 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 or floating point division depending on whether you type `1/3` or `1.0/3.0`.

These examples demonstrate how you can work interactively with the interpreter, allowing you to experiment and explore. Now let's try a nonsensical expression to see how the interpreter handles it:

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

Here we have produced a **syntax error**. 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 "standard input").

Searching Text

Now that we can use the Python interpreter, let's see how we can harness its power to process text. The first step is to type a special command at the Python prompt which tells the interpreter to load some texts

The chapter contains many examples and exercises; there is no better way to learn to NLP than to dive in and try these yourself. However, before continuing you need to install NLTK, downloadable for free from http://www.nltk.org/. Once you've done this, install the data required for the book by typing `nltk.download()` at the Python prompt, and download the `book` collection.

for us to explore: `from nltk.book import *` — i.e. load NLTK's `book` module, which contains the examples you'll be working with as you read this chapter. After printing a welcome message, it loads the text of several books, including *Moby Dick*. Type the following, taking care to get spelling and punctuation exactly right:

```
>>> from nltk.book import *
*** Introductory Examples for the NLTK Book ***
Loading: text1, ..., text8 and sent1, ..., sent8
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
>>> text1
<Text: Moby Dick by Herman Melville 1851>
>>> text2
<Text: Sense and Sensibility by Jane Austen 1811>
>>>
```

We can examine the contents of a text in a variety of ways. A concordance view shows us every occurrence of a given word, together with some context. Here we look up the word *monstrous*.

```
>>> text1.concordance("monstrous")
mong the former , one was of a most monstrous size . ... This came towards us , o
ION OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have re
all over with a heathenish array of monstrous clubs and spears . Some were thickl
ed as you gazed , and wondered what monstrous cannibal and savage could ever have
 that has survived the flood ; most monstrous and most mountainous ! That Himmale
 they might scout at Moby Dick as a monstrous fable , or still worse and more det
ath of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall ere lo
ling Scenes . In connexion with the monstrous pictures of whales , I am strongly
>>>
```

Your Turn: Try seaching for other words; you can use the up-arrow key 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 *US Presidential Inaugural Addresses* to see examples of English dating 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 few minutes 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.

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. We can also determine the *location* of a word in the text: how many words in 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 1.1 we see some striking patterns of word usage over the last 220 years (in an artificial text constructed by joining the addresses end-to-end). You can produce this plot as shown below. You might like to try different words, 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"])
>>>
```

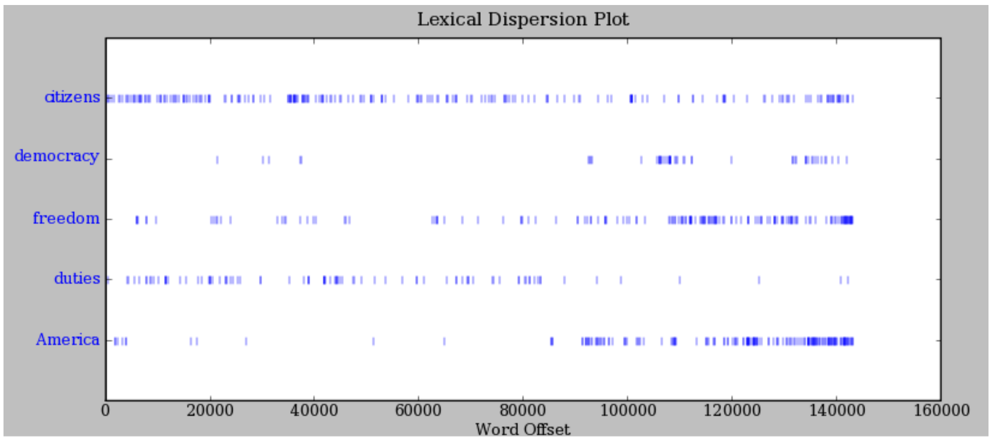


Figure 1.1: Lexical Dispersion Plot for Words in US Presidential Inaugural Addresses

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

A concordance permits us to see words in context, e.g. we saw that *monstrous* appeared in the context *the monstrous pictures*. What other words appear in the same contexts that *monstrous* appears in? We can find out as follows:

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

Observe that we get different results for different books. Melville and Austen use this word quite differently. For Austen *monstrous* has positive connotations, and might even function as an intensifier, like the word *very*. Let's examine the contexts that are shared by *monstrous* and *very*

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

Your Turn: Pick another word and compare its usage in two different texts, using the `similar()` and `common_contexts()` methods.

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 "generate" function:

```
>>> 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 first time you run this, 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 re-uses common words and phrases from the source text and gives us a sense of its style and content.

When text is printed, punctuation has been split off from the previous word. Although this is not correct formatting for English text, we do this to make it clear that punctuation does not belong to the word. This is called "tokenization", and you will learn about it in Chapter 3.

Counting Vocabulary

The most obvious fact about texts that emerges from the previous section 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'll use the text of *Moby Dick* again:

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

That's a quarter of a million words long! But how many distinct words does this text 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 words that it uses, and in Python we can list the vocabulary of `text3` with the command: `set(text3)` (many screens of words will fly past). Now try the following:

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

```
>>> len(text3) / len(set(text3))
16
>>>
```

Here we can see 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 length of the set. There are fewer than 3,000 distinct words in this book. Finally, we can calculate a measure of the lexical richness of the text and learn that each word is used 16 times on average.

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
>>> 100.0 * text4.count('a') / len(text4)
1.4587672822333748
>>>
```

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 might like to repeat such calculations on several texts, but it is tedious to keep retyping it for different texts. Instead, we can come up with our own name for a task, e.g. "score", and associate it with a block of code. Now we only have to type a short name instead of one or more complete lines of Python code, and we can re-use it as often as we like:

```
>>> def score(text):
...     return len(text) / len(set(text))
...
>>> score(text3)
16
>>> score(text5)
4
>>>
```

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. To finish the indented block just enter a blank line.

The keyword `def` is short for "define", and the above code defines a *function*:dt" called "score". We used the function by typing its name, followed by an open parenthesis, the name of the text, then a close parenthesis. This is just what we did for the `len` and `set` functions earlier. These parentheses will show up often: their role is to separate the name of a task — such as `score` — from the data that the task is to be performed on — such as `text3`. Functions are an advanced concept in programming and we only mention them at the outset to give newcomers a sense of the power and creativity of programming. Later we'll see how to use such functions when tabulating data, like 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 functions.

Genre	Token Count	Type Count	Score
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

Table 1.1: Lexical Diversity of Various Genres in the Brown Corpus

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

You've seen some important building blocks of the Python programming language. Let's 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, and we can ask for its length:

```
>>> sent1
['Call', 'me', 'Ishmael', '.']
>>> len(sent1)
4
>>> score(sent1)
1
>>>
```

We can even apply our own "score" function to it. Some more lists have been defined for you, one for the opening sentence of each of our texts, `sent2` … `sent8`. We inspect two of them here; you can see the rest for yourself using the Python interpreter.

```
>>> 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 above in Section 1.1, e.g. `sorted(ex1)`, `len(set(ex1))`, `ex1.count('the')`.

We can also do arithmetic operations with lists in Python. Multiplying a list by a number, e.g. `sent1 * 2`, creates a longer list with multiple copies of the items in the original list. Adding two lists, e.g. `sent4 + sent1`, creates a new list with everything from the first list, followed by everything from the second list:

```
>>> sent1 * 2
['Call', 'me', 'Ishmael', '.', 'Call', 'me', 'Ishmael', '.']
>>> sent4 + sent1
['Fellow', '-', 'Citizens', 'of', 'the', 'Senate', 'and', 'of', 'the',
'House', 'of', 'Representatives', ':', 'Call', 'me', 'Ishmael', '.']
>>>
```

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

Indexing Lists

As we have seen, a text in Python is just a list of words, represented using a particular combination of brackets and quotes. Just as with an ordinary page of text, we can count up the total number of words using `len(text1)`, and count the occurrences of a particular word using `text1.count('heaven')`. And just as we can pick out the first, tenth, or even 14,278th word in a printed text, we can identify the elements of a list by their number, or **index**, by following the name of the text with the index inside brackets. We can also find the index of the first occurrence of any word:

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

Indexes turn out to be a common way to access the words of a text, or — more generally — the elements of a 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',
...         'word11', 'word12', 'word13', 'word14', 'word15',
...         'word16', 'word17', 'word18', 'word19', 'word20']
>>> sent[0]
'word1'
>>> sent[19]
'word20'
>>>
```

Notice that our indexes start from zero: `sent` element zero, written `sent[0]`, is the first word, `'word1'`, while `sent` element 19 is `'word20'`. 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 is initially confusing, but typical of modern programming languages. You'll quickly get the hang of this 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 tell it to go too far, by using an index value that is too large, we get an error:

```
>>> sent[20]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

This time it is not a syntax error, for 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:

```
>>> sent[17:20]
['word18', 'word19', 'word20']
>>> sent[17]
'word18'
>>> sent[18]
'word19'
>>> sent[19]
'word20'
>>>
```

Thus, the slice `17:20` includes `sent` elements 17, 18, and 19. By convention, `m:n` means elements m...n-1. We can omit the first number if the slice begins at the start of the list, and we can omit the second number if the slice goes to the end:

```
>>> sent[:3]
['word1', 'word2', 'word3']
>>> text2[141525:]
['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, e.g. putting `sent[0]` on the left of the equals sign. We can also replace an entire slice with new material:

```
>>> sent[0] = 'First Word'
>>> sent[19] = 'Last Word'
>>> sent[1:19] = ['Second Word', 'Third Word']
>>> sent
['First Word', 'Second Word', 'Third Word', 'Last Word']
>>>
```

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 above. 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 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 copied from the right side to the left. It might help to think of it as a left-arrow. 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:

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

It is good to choose meaningful variable names to help you — and anyone who reads your Python code — to understand 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`. A variable name cannot be any of Python's reserved words, such as `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 can use variables to hold intermediate steps of a computation. This may make the Python 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. You can use underscores anywhere in a name, but not a hyphen, since this gets interpreted 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**):

```
>>> name = 'Monty'
>>> name[0]
'M'
>>> name[:4]
'Mont'
>>> 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.

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.

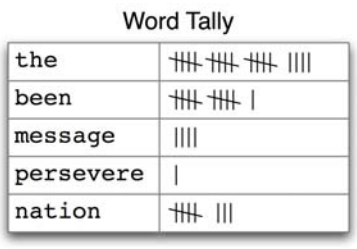
In this section we pick up the question of what makes a text distinct, and use automatic methods to find characteristic words and expressions of a text. As in Section 1.1, you will try new features of the Python language by copying them into the interpreter, and you'll learn about these features systematically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

Before continuing with this section, check your understanding of the previous section by predicting the output of the following code, and using the interpreter to check if you got it right. If you found it difficult 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', '.']
>>> words = set(saying)
>>> words = sorted(words)
>>> words[:2]
>>>
```

Frequency Distributions

How could we automatically identify the words of a text that are most informative about the topic and genre of the text? Let's begin by finding the most frequent words 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 1.2. We would need thousands of counters and it would be a laborious process, so laborious that we would rather assign the task to a machine.


Word Tally

the	卌 卌 卌 ||||
been	卌 卌 |
message	||||
persevere	|
nation	卌 |||

Figure 1.2: Counting Words Appearing in a Text (a frequency distribution)

The table in Figure 1.2 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 the total number of words in the text — 260,819 in the case of *Moby Dick* — 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*.

```
>>> fdist1 = FreqDist(text1)
>>> fdist1
<FreqDist with 260819 samples>
>>> vocabulary1 = fdist1.keys()
>>> vocabulary1[:50]
[',', '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
```

```
>>>
```

Your Turn: Try the above frequency distribution example for yourself, for `text2`. Be careful 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 in the above list 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 English text is taken up with such words? We can generate a cumulative frequency plot for these words, using `fdist1.plot(cumulative=True)`, to produce the graph in Figure 1.3. These 50 words account for nearly half the book!

If the frequent words don't help us, how about the words that occur once only, the so-called **hapaxes**. See them using `fdist1.hapaxes()`. This list contains *lexicographer*, *cetological*, *contraband*, *expostulations*, and about 9,000 others. It seems that there's too many rare words, and without seeing the context we probably can't guess what half of them mean in any case! Neither frequent nor infrequent words help, so 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 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 equivalent Python expression is given in (1b). Notice 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 if `len(w)` is greater than 15; all other words will be ignored. We will discuss this syntax more carefully later.

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*, while those in `text5` reflect its informal

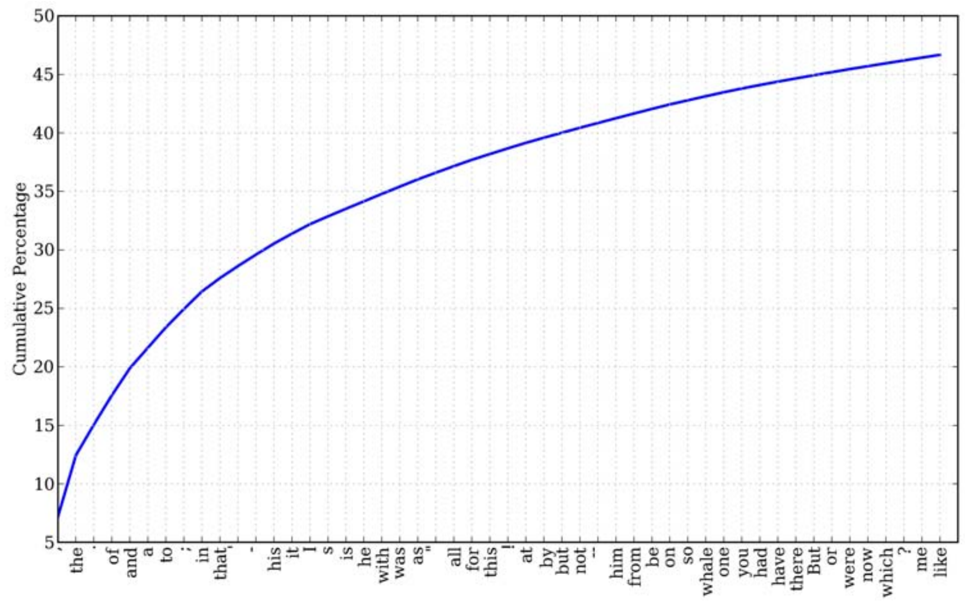


Figure 1.3: Cumulative Frequency Plot for 50 Most Frequent Words in *Moby Dick*

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 like (*antiphilosophists*). Here are all words from the chat corpus that are longer than 7 characters, that occur more than 7 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-occuring content-bearing words of the text. It is a modest but important milestone: a tiny piece of code, processing thousands of words, produces some informative output.

Bigrams and Collocations

Frequency distributions are very powerful. Here we briefly explore a more advanced application that uses word pairs, also known as **bigrams**. We can convert a list of words to a list of bigrams as follows:

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

Your Turn: Try out the above statements in the Python interpreter, and try changing the text, and changing the length condition. Also try changing the variable names, e.g. using `[word for word in vocab if ...]`.

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()
United States; fellow citizens; has been; have been; those who;
Declaration Independence; Old World; Indian tribes; District Columbia;
four years; Chief Magistrate; and the; the world; years ago; Santo
Domingo; Vice President; the people; for the; specie payments; Western
Hemisphere
>>>
```

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]
[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)
>>> fdist
<FreqDist with 260819 samples>
>>> fdist.samples()
[3, 1, 4, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20]
>>>
```

The material being counted up in the frequency distribution consists of the numbers `[1, 4, 4, 2, ...]`, and the result is a distribution containing a quarter of a million items, one per word. There are only twenty distinct items being counted, the numbers 1 through 20. Let's look at the frequency of each sample:

```
>>> 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 50,000 (20%) of of the words of the book. Further analysis of word length might help us understand differences between authors, genres or languages. Table 1.2 summarizes the methods defined in 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`	samples in `fdist1` occur less frequently than in `fdist2`

Table 1.2: Methods Defined for NLTK's Frequency Distributions

Our discussion of frequency distributions has introduced some important Python concepts, and we will look at them systematically in Section 1.4. We've also touched on the topic of normalization, and we'll explore this in depth in Chapter 3.

1.4 Back to Python: Making Decisions and Taking Control

So far, our little programs have had some interesting qualities: (i) the ability to work with language, and (ii) 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 like < and >= for testing the relationship between values. The full set of these **relational operators** are shown in Table 1.3.

Operator	Relationship
<	less than
<=	less than or equal to
==	equal to (note this is two not one = sign)
!=	not equal to
>	greater than
>=	greater than or equal to

Table 1.3: Numerical Comparison Operators

We can use these to select different words from a sentence of news text. Here are some examples — only the operator is changed from one line to the next. They all use `sent7`, the first sentence from `text7` (Wall Street Journal).

```
>>> [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', '.']
>>>
```

Notice the pattern in all of these examples: `[w for w in text if` *condition* `]`. In these cases the condition is always a numerical comparison. However, we can also test various properties of words, using the functions listed in Table 1.4.

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

Table 1.4: Some Word Comparison Operators

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 `and` and `or`: 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 above 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 above notation 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 incorrect result (why?). Don't worry if you don't feel confident with these already. You might like to try some of the exercises at the end of this chapter, or wait til we come back to these again in the next chapter.

Nested Code Blocks

Most programming languages permit us to execute a block of code when a **conditional expression**, or `if` statement, is satisfied. In the following program, we have created a variable called `word` containing the string value `'cat'`. The `if` statement checks whether the conditional expression `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'
...
word length is less than 5
>>>
```

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

If we change the conditional expression to `len(word) >= 5`, to check that the length of `word` is greater than or equal to `5`, then the conditional expression 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. Don't forget 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 performings 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 only print the item 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 'lowercase word'
...     elif token.istitle():
...         print 'titlecase word'
...     else:
...         print 'punctuation'
...
titlecase word
lowercase word
titlecase word
punctuation
>>>
```

As you can see, even with this small amount of Python knowledge, you can start to build multi-line 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, dictionaries, and a programming language. However, we're also interested in exploiting our knowledge of language and computation by building useful language technologies.

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 behaviour 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 analysing unrestricted text become more widespread, the prospect of natural language understanding has re-emerged as a plausible goal.

In this section we describe some language processing components and systems, to give you a sense the interesting challenges that are waiting for you.

Word Sense Disambiguation

In **word sense disambiguation** we want to work out which sense of a word was intended in a given context. Consider the ambiguous words *serve* and *dish*:

(2) a. *serve*: help with food or drink; hold an office; put ball into play

b. *dish*: plate; course of a meal; communications device

Now, in a sentence containing the phrase: *he served the dish*, you can detect that both *serve* and *dish* are being used with their food meanings. Its unlikely that the topic of discussion shifted from sports to crockery in the space of three words — this would force you to invent bizarre images, like a tennis pro taking out her frustrations on a china tea-set laid out beside the court. In other words, we automatically disambiguate words using context, exploiting the simple fact that nearby words have closely related meanings. As another example of this contextual effect, consider the word *by*, which has several meanings, e.g.: *the book by Chesterton* (agentive); *the cup by the stove* (locative); and *submit by Friday* (temporal). Observe in (3c) that the meaning of the italicized word helps us interpret the meaning of *by*.

(3) a. The lost children were found by the *searchers* (agentive)

b. The lost children were found by the *mountain* (locative)

c. The lost children were found by the *afternoon* (temporal)

Pronoun Resolution

A deeper kind of language understanding is to work out who did what to whom — i.e. to detect the subjects and objects of verbs. You learnt to do this in elementary school, but its harder than you might think. In the sentence *the thieves stole the paintings* it is easy to tell who performed the stealing action. Consider three possible following sentences in (4c), and try to determine what was sold, caught, and found (one case is ambiguous).

(4) a. The thieves stole the paintings. They were subsequently *sold*.

b. The thieves stole the paintings. They were subsequently *caught*.

c. The thieves stole the paintings. They were subsequently *found*.

Answering this question involves finding the **antecedent** of the pronoun *they* (the thieves or the paintings). Computational techniques for tackling this problem include **anaphora resolution** — identifying what a pronoun or noun phrase refers to — and **semantic role labeling** — identifying how a noun phrase relates to verb (as agent, patient, instrument, and so on).

Generating Language Output

If we can automatically solve such problems, we will have understood enough of the text to perform some tasks that involve generating language output, such as *question answering* and *machine translation*. In the first case, a machine should be able to answer a user's questions relating to collection of texts:

(5) a. *Text:* ... The thieves stole the paintings. They were subsequently sold. ...

b. *Human:* Who or what was sold?

c. *Machine:* The paintings.

The machine's answer demonstrates that it has correctly worked out that *they* refers to paintings and not to thieves. In the second case, the machine should be able to translate the text into another language, accurately conveying the meaning of the original text. In translating the above text into French, we are forced to choose the gender of the pronoun in the second sentence: *ils* (masculine) if the thieves are sold, and *elles* (feminine) if the paintings are sold. Correct translation actually depends on correct understanding of the pronoun.

(6) a. The thieves stole the paintings. They were subsequently found.

b. Les voleurs ont volé les peintures. Ils ont été trouvés plus tard. (the thieves)

c. Les voleurs ont volé les peintures. Elles ont été trouvées plus tard. (the paintings)

In all of the above examples — working out the sense of a word, the subject of a verb, the antecedent of a pronoun — are steps in establishing the meaning of a sentence, things we would expect a language understanding system to be able to do. We'll come back to some of these topics later in the book.

Spoken Dialog Systems

In the history of artificial intelligence, the chief measure of intelligence has been a linguistic one, namely the *Turing Test*: can a dialogue system, responding to a user's text input, perform so naturally that we cannot distinguish it from a human-generated response? In contrast, today's commercial dialogue systems are very limited, but still perform useful functions in narrowly-defined domains, as we see below:

S: How may I help you?
U: When is Saving Private Ryan playing?
S: For what theater?
U: The Paramount theater.
S: Saving Private Ryan is not playing at the Paramount theater, but
it's playing at the Madison theater at 3:00, 5:30, 8:00, and 10:30.

You could not ask this system to provide driving instructions or details of nearby restaurants unless the required information had already been stored and suitable question-answer pairs had been incorporated into the language processing system.

Observe that the above system seems to understand the user's goals: the user asks when a movie is showing and the system correctly determines from this that the user wants to see the movie. This inference seems so obvious that you probably didn't notice it was made, yet a natural language system needs to be endowed with this capability in order to interact naturally. Without it, when asked *Do you know when Saving Private Ryan is playing*, a system might unhelpfully respond with a cold *Yes*. However, the developers of commercial dialogue systems use contextual assumptions and business logic to ensure that the different ways in which a user might express requests or provide information are handled in a way that makes sense for the particular application. So, if you type *When is* ..., or *I want to know when* ..., or *Can you tell me when* ..., simple rules will always yield screening times. This is enough for the system to provide a useful service.

Dialogue systems give us an opportunity to mention the complete processing pipeline for NLP. Figure 1.4 shows the architecture of a simple dialogue system.

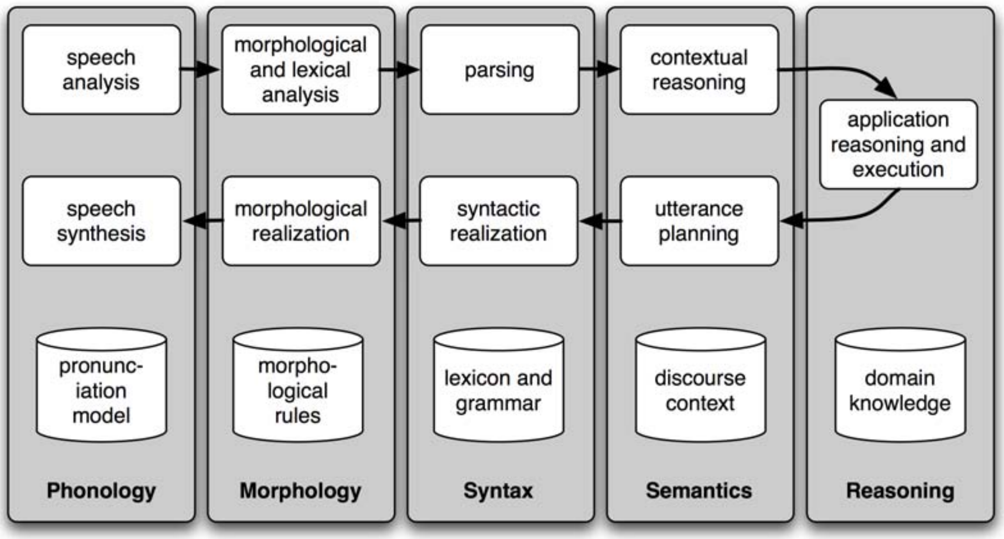


Figure 1.4: Simple Pipeline Architecture for a Spoken Dialogue System

Along the top of the diagram, moving from left to right, is a "pipeline" of some language understanding **components**. These map from speech input via syntactic parsing to some kind of meaning representation. Along the middle, moving from right to left, is the reverse pipeline of components for converting concepts to speech. These components make up the dynamic aspects of the system. At the bottom of the diagram are some representative bodies of static information: the repositories of language-related data that the processing components draw on to do their work.

Your Turn: Try the Eliza chatbot `nltk.chat.eliza.demo()` to see an example of a primitive dialogue system.

Textual Entailment

The challenge of language understanding has been brought into focus in recent years by a public "shared task" called Recognizing Textual Entailment (RTE). The basic scenario is simple. Suppose you want to find find evidence to support the hypothesis: *Sandra Goudie was defeated by Max Purnell*, and that you have another short text that seems to be relevant, for example, *Sandra Goudie was first elected to Parliament in the 2002 elections, narrowly winning the seat of Coromandel by defeating Labour candidate Max Purnell and pushing incumbent Green MP Jeanette Fitzsimons into third place*. Does the text provide enough evidence for you to accept the hypothesis? In this particular case, the answer will be No. You can draw this conclusion easily, but it is very hard to come up with automated methods for making the right decision. The RTE Challenges provide data which allow competitors to develop their systems, but not enough data to brute-force approaches using standard machine learning techniques. Consequently, some linguistic analysis is crucial. In the above example, it is important for the system to note that *Sandra Goudie* names the person being defeated in the hypothesis, not the person doing the defeating in the text. As another illustration of the difficulty of the task, consider the following text/hypothesis pair:

(7) a. David Golinkin is the editor or author of eighteen books, and over 150 responsa, articles, sermons and books

b. Golinkin has written eighteen books

In order to determine whether or not the hypothesis is supported by the text, the system needs the following background knowledge: (i) if someone is an author of a book, then he/she has written that book; (ii) if someone is an editor of a book, then he/she has not written that book; (iii) if someone is editor or author of eighteen books, then one cannot conclude that he/she is author of eighteen books.

Limitations of NLP

Despite the research-led advances in tasks like RTE, natural language systems that have been deployed for real-world applications still cannot perform common-sense reasoning or draw on world knowledge in a general and robust manner. We can wait for these difficult artificial intelligence problems to be solved, but in the meantime it is necessary to live with some severe limitations on the reasoning and knowledge capabilities of natural language systems. Accordingly, right from the beginning, an important goal of NLP research has been to make progress on the holy grail of natural language understanding, using superficial yet powerful counting and symbol manipulation techniques, but *without* recourse to this unrestricted knowledge and reasoning capability.

This is one of the goals of this book, and we hope to equip you with the knowledge and skills to build useful NLP systems, and to contribute to the long-term vision of building intelligent machines.

1.6 Summary

- Texts are represented in Python using lists: `['Monty', 'Python']`. We can use indexing, slicing and the `len()` function on lists.
- We get the vocabulary of a text `t` using `sorted(set(t))`.
- To get the vocabulary, collapsing case distinctions and ignoring punctuation, we can write `set(w.lower() for w in text if w.isalpha())`.
- We operate on each item of a text using `[f(x) for x in text]`.
- We process each word in a text using a `for` statement such as `for w in t:` or `for word in text:`. This must be followed by the colon character and an indented block of code, to be executed each time through the loop.
- We test a condition using an `if` statement: `if len(word) < 5:`. This must be followed by the colon character and an indented block of code, to be executed only if the condition is true.
- A frequency distribution is a collection of items along with their frequency counts (e.g. the words of a text and their frequency of appearance).
- WordNet is a semantically-oriented dictionary of English, consisting of synonym sets — or synsets — and organized into a hierarchical network.

Chapter 2

Accessing Text Corpora and Lexical Resources

Practical work in Natural Language Processing usually involves a variety of established bodies of linguistic data. Such a body of text is called a **corpus** (plural **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 re-use code effectively?

This chapter continues to present programming concepts by example, in the context of a linguistic processing task. We will wait till 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 any large body of text. Many, but not all, 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 we glued them end-to-end and treated them as a single text. In this section we will examine a variety of text corpora and will see how to select individual texts, and how to work with them.

The Gutenberg Corpus

NLTK includes a small selection of texts from the Project Gutenberg http://www.gutenberg.org/ electronic text archive containing some 25,000 free electronic books. We begin by getting the Python interpreter to load the NLTK package, then ask to see `nltk.corpus.gutenberg.files()`, the files in NLTK's corpus of Gutenberg texts:

```
>>> import nltk
>>> nltk.corpus.gutenberg.files()
('austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt',
'blake-poems.txt', 'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt',
'chesterton-thursday.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
```

You cannot carry out concordancing (and other tasks from Section 1.1) using a text defined this way. Instead you have to make the following statement:

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

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

```
>>> from nltk.corpus import gutenberg
>>> gutenberg.files()
('austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt',
'blake-poems.txt', 'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt',
'chesterton-thursday.txt', 'melville-moby_dick.txt', 'milton-paradise.txt',
'shakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt',
'whitman-leaves.txt')
```

Let's write a short program to display other information about each text:

```
>>> for file in gutenberg.files():
...     num_chars = len(gutenberg.raw(file))
...     num_words = len(gutenberg.words(file))
...     num_sents = len(gutenberg.sents(file))
...     num_vocab = len(set(w.lower() for w in gutenberg.words(file)))
...     print num_chars/num_words, num_words/num_sents, num_words/num_vocab, file
...
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 16 12 carroll-alice.txt
4 17 11 chesterton-ball.txt
4 19 11 chesterton-brown.txt
4 16 10 chesterton-thursday.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 has displayed 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 is always 4. Average sentence length and lexical diversity appear to be characteristics of particular authors.

This example also showed how we can access the "raw" text of the book, not split up into words. 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[1038]
['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()`. We access the raw file contents using `raw()`, and get the content sentence by sentence using `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 f in webtext.files():
...     print f, webtext.raw(f)[:70]
...
firefox.txt Cookie Manager: "Don't allow sites that set removed cookies to set fut
grail.txt SCENE 1: [wind] [clop clop clop] KING ARTHUR: Whoa there!  [clop clop
overheard.txt White guy: So, do you have any plans for this evening? Asian girl: Yea
pirates.txt PIRATES OF THE CARRIBEAN: DEAD MAN'S CHEST, by Ted Elliott & Terry Ros
singles.txt 25 SEXY MALE, seeks attrac older single lady, for discreet encounters.
wine.txt Lovely delicate, fragrant Rhone wine. Polished leather and strawberrie
```

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', '.']
```

The 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`).

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*

Table 2.1: Example Document for Each Section of the Brown Corpus

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(files=['cg22'])
['Does', 'our', 'society', 'have', 'a', 'runaway', ',', ...]
>>> brown.sents(categories=['news', 'editorial', 'reviews'])
[['The', 'Fulton', 'County'...], ['The', 'jury', 'further'...], ...]
```

We can use the Brown Corpus to study 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:

```
>>> 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 above method 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. To save re-typing, we can put the above code into a function, and use the function several times over. (We discuss functions in more detail in Section 2.3.) However, there is an even better way, using NLTK's support for conditional frequency distributions (Section 2.2), as follows:

```
>>> cfd = nltk.ConditionalFreqDist((g,w)
...           for g in brown.categories()
...           for w in brown.words(categories=g))
>>> 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*, suggesting a focus on the future, while the most frequent modal in the romance genre is *could*, suggesting a focus on possibilities.

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" (for training and testing algorithms that automatically detect the topic of a document, as we will explore further in Chapter 5).

```
>>> from nltk.corpus import reuters
>>> reuters.files()
('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 name or a list of names.

```
>>> reuters.categories('training/9865')
['barley', 'corn', 'grain', 'wheat']
>>> reuters.categories(['training/9865', 'training/9880'])
['barley', 'corn', 'grain', 'money-fx', 'wheat']
>>> reuters.files('barley')
['test/15618', 'test/15649', 'test/15676', 'test/15728', 'test/15871', ...]
>>> reuters.files(['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 upper case.

```
>>> 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', ...]
```

Many other English text corpora are provided with NLTK. For a list see Appendix 1.1. For more examples of how to access NLTK corpora, please consult the Corpus HOWTO at http://www.nltk.org/howto.

US Presidential Inaugural Addresses

In section 1.1, we looked at the US Presidential Inaugural Addresses corpus, but treated it as a single text. The graph in Figure fig-inaugural, used word offset as one of the axes, but this is difficult to interpret. 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.files()
('1789-Washington.txt', '1793-Washington.txt', '1797-Adams.txt', ...)
>>> [file[:4] for file in inaugural.files()]
['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 file name, we extracted the first four characters, using `file[:4]`.

Let's look at how the words *America* and *citizen* are used over time. The following code will count similar words, such as plurals of these words, or the word *Citizens* as it would appear at the start of a sentence (how?). The result is shown in Figure 2.1.

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

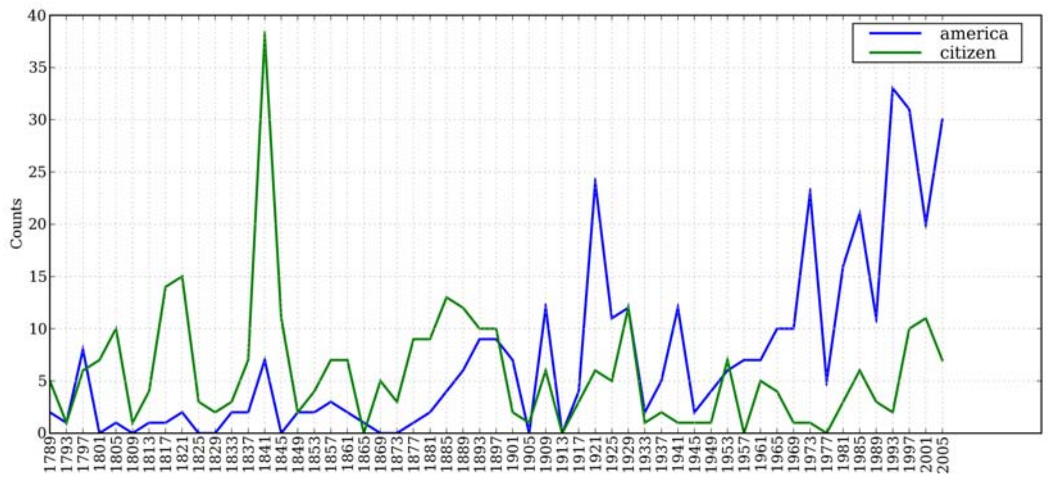


Figure 2.1: Conditional Frequency Distribution for Two Words in the Inaugural Address Corpus

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 (for a listing, see `http://www.nltk.org/data`). We will discuss these these in later chapters.

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 Appendix 1).

```
>>> 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.udhr.files()
('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', ...]
>>> 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', ...]
```

The last of these corpora, `udhr`, contains the Universal Declaration of Human Rights in over 300 languages. (Note that the names of the files in this corpus include information about character encoding, and for now we will stick with texts in ISO Latin-1, or ASCII)

Let's use a conditional frequency distribution to examine the differences in word lengths, for a selection of languages included in this corpus. The output is shown in Figure 2.2 (run the program yourself to see a color plot).

```
>>> 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)
```

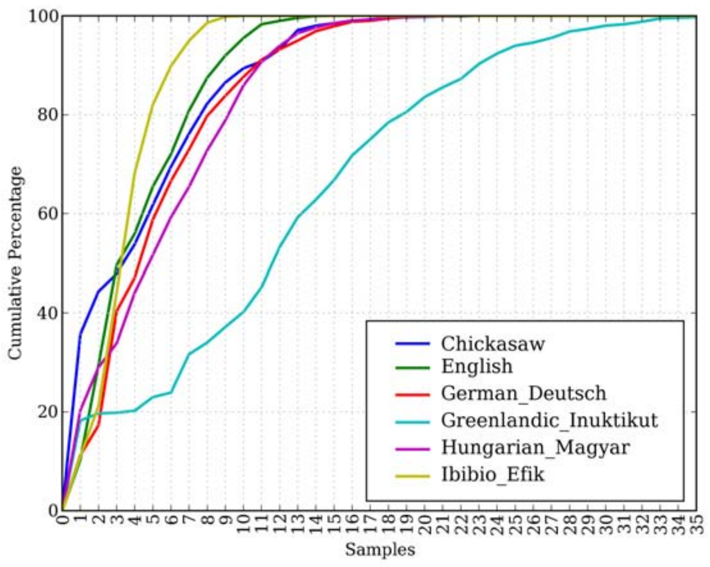


Figure 2.2: Cumulative Word Length Distributions for Several Languages

Unfortunately, for many languages, substantial corpora are not yet available. Often there is no government or industrial support for developing language resources, and individual efforts are piecemeal and hard to discover or re-use. Some languages have no established writing system, or are endangered. A good place to check is the search service of the *Open Language Archives Community*, at `http://www.language-archives.org/`. This service indexes the catalogs of dozens of language resource archives and publishers.

Text Corpus Structure

The corpora we have seen exemplify a variety of common corpus structures, 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, since a text can be relevant to more than one topic. Occasionally, text collections have temporal structure, news collections being the most common.

NLTK's corpus readers support efficient access to a variety of corpora, and can easily be extended to work with new corpora [REF]. Table 2.2 lists the basic methods provided by the corpus readers.

Example	**Description**
`files()`	the files of the corpus

Example	Description
`categories()`	the categories of the corpus
`abspath(file)`	the location of the given file on disk
`words()`	the words of the whole corpus
`words(files=[f1,f2,f3])`	the words of the specified files
`words(categories=[c1,c2])`	the words of the specified categories
`sents()`	the sentences of the specified categories
`sents(files=[f1,f2,f3])`	the sentences of the specified files
`sents(categories=[c1,c2])`	the sentences of the specified categories

Table 2.2: Basic Methods Defined in NLTK's Corpus Package

Loading your own Corpus

If you have a collection of text files that you would like to access using the above methods, you can easily load them with the help of NLTK's `PlaintextCorpusReader` as follows:

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

The second parameter of the `PlaintextCorpusReader` can be a list of file pathnames, like `['a.txt', 'test/b.txt']`, or a pattern that matches all file pathnames, like `'[abc]/.*\.txt'` (see Section 3.3 for information about regular expressions, and Chapter 11 for more information about NLTK's corpus readers.

2.2 Conditional Frequency Distributions

We introduced frequency distributions in Chapter 1, and 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. 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 to enable study of 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.

Your Turn: Pick a language of interest in `udhr.files()`, 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()`.

The most complete inventory of the world's languages is *Ethnologue*, `http://www.ethnologue.com/`.

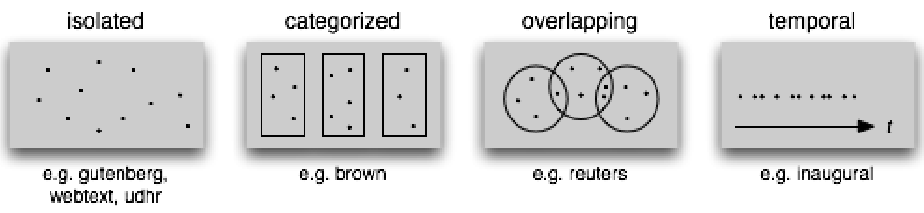


Figure 2.3: Common Structures for Text Corpora (one point per text)

For more information about NLTK's Corpus Package, type `help(nltk.corpus.reader)` at the Python prompt, or see the Corpus HOWTO at http://www.nltk.org/howto. You will probably have other text sources, stored in files on your computer or accessible via the web. We'll discuss how to work with these in Chapter 3.

Conditions and Events

As we saw in Chapter 1, a frequency distribution counts observable events, such as the appearance of words in a text. A conditional frequency distribution needs to pair each such event with a condition. So instead of processing a text (a sequence of words), we have to process a sequence of pairs:

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

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

[TUPLES]

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.

```
>>> cfd = nltk.ConditionalFreqDist((g,w)
...                                for g in brown.categories()
...                                for w in brown.words(categories=g))
```

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

```
>>> genre_word = [(g,w) for g in ['news', 'romance'] for w in brown.words(categories=g)]
>>> len(genre_word)
170576
```

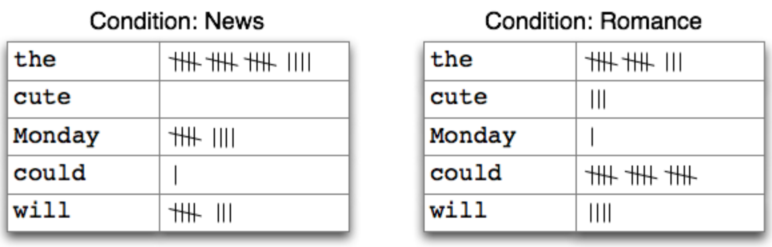

Condition: News
the
cute
Monday
could
will
Condition: Romance
the
cute
Monday
could
will

Figure 2.4: Counting Words Appearing in a Text Collection (a conditional frequency distribution)

So pairs at the beginning of the list `genre_word` will be of the form (`’news’`, *word*) while those at the end will be of the form (`’romance’`, *word*). (Recall that `[-4:]` gives us a slice consisting of the last four items of a sequence.)

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

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, and verify it has two conditions:

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

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

```
>>> cfd[’news’]
<FreqDist with 100554 samples>
>>> cfd[’romance’]
<FreqDist with 70022 samples>
>>> 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
```

Apart from combining two or more frequency distributions, and being easy to initialize, a `ConditionalFreqDist` provides some useful methods for tabulation and plotting. We can optionally specify which conditions to display with a `conditions=` parameter. When we omit it, we get all the conditions.

Your Turn: 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`.

Other Conditions

The plot in Figure 2.2 is based on a conditional frequency distribution where the condition is the name of the language and the counts being plotted are derived from word lengths. It exploits the fact that the filename for each language is the language name followed by"'-Latin1'" (the character encoding).

```
>>> cfd = nltk.ConditionalFreqDist((lang, len(word))
...           for lang in languages
...           for word in udhr.words(lang + '-Latin1'))
```

The plot in Figure 2.1 is based on a conditional frequency distribution where the condition is either of two words *america* or *citizen*, and the counts being plotted are the number of times the word occurs in a particular speech. It expoits the fact that the filename for each speech, e.g. `1865-Lincoln.txt` contains the year as the first four characters.

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

This code will generate the tuple `('america', '1865')` for every instance of a word whose lower-cased form starts with "america" — such as "Americans" — in the file `1865-Lincoln.txt`.

Generating Random Text with Bigrams

We can use a conditional frequency distribution to create a table of bigrams (word pairs). (We introducted bigrams 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 Figure 2.5, 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.

Summary

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

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

Figure 2.5: Generating Random Text in the Style of Genesis

Example	Description
`cfdist = ConditionalFreqDist(pairs)`	create a conditional frequency distribution
`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.plot()`	graphical plot of the conditional frequency distribution
`cfdist1 < cfdist2`	samples in `cfdist1` occur less frequently than in `cfdist2`

Table 2.3: Methods Defined for NLTK's Conditional Frequency Distributions

2.3 More Python: Reusing Code

By this time you've probably retyped a lot of code. 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 interative interpreter performs your instructions as soon as you type them. Often, it is better to compose a multi-line 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:

```
msg = 'Monty Python'
```

Save this program in a file called `test.py`, then go to the `Run` menu, and select the command `Run Module`. The result in the main IDLE window should look like this:

```
>>> ================================ RESTART ================================
>>>
>>>
```

Now, where is the output showing the value of `msg`? The answer is that the program in `test.py` will show a value only if you explicitly tell it to, using the `print` statement. So add another line to `test.py` so that it looks as follows:

```
msg = 'Monty Python'
print msg
```

Select `Run Module` again, and this time you should get output that looks like this:

```
>>> ================================ RESTART ================================
>>>
```

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, and consulting the interactive help facility. Once you're ready, you can paste the code (minus any `>>>` 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`.

Our inline code examples will continue to 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 above.

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. It usually has some inputs, also 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:

```
>>> def score(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 the above example, all the work of the function is done in the `return` statement. Here's an equivalent definition which does the same work using multiple lines of code. We'll change the parameter name to remind you that this is an arbitrary choice:

```
>>> def score(my_text_data):
...     word_count = len(my_text_data)
...     vocab_size = len(set(my_text_data))
...     richness_score = word_count / vocab_size
...     return richness_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. Notice also that defining a function like this produces no output. Functions do nothing until they are "called" (or "invoked").

Let's return to our earlier scenario, and actually define a simple plural function. The function `plural()` in Figure 2.6 takes a singular noun and generates a plural form (one which is not always correct).

(There is much more to be said about functions, but we will hold off until Section 6.3.)

Modules

Over time you will find that you create a variety of useful little text processing functions, and you end up copy-pasting them from old programs to new ones. Which file contains the latest version of the

```
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'
    return word + 's'

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

Figure 2.6: Example of a Python function

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 any copying and pasting.

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 has an error, and we'll need to fix it. This time, we won't produce another version, but instead we'll fix the existing one. Thus, at every stage, there is only one version of our plural function, and no confusion about which one we should use.

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**.

[Work in somewhere: In general, we use `import` statements when we want to get access to Python code that doesn't already come as part of core Python. This code will exist somewhere as one or more files. Each such file corresponds to a Python **module** — this is a way of grouping together code and data that we regard as reusable. When you write down some Python statements in a file, you are in effect creating a new Python module. And you can make your code depend on another module by using the `import` statement.]

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 folder / directory.)

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 a defined a text `my_text`, then `vocab = sorted(set(my_text))` builds the vocabulary of `my_text`, while `word_freq = FreqDist(my_text)` counts the frequency of each word in the text. Both of `vocab` and `word_freq` are simple lexical resources. Similarly, a concordance (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 2.7.

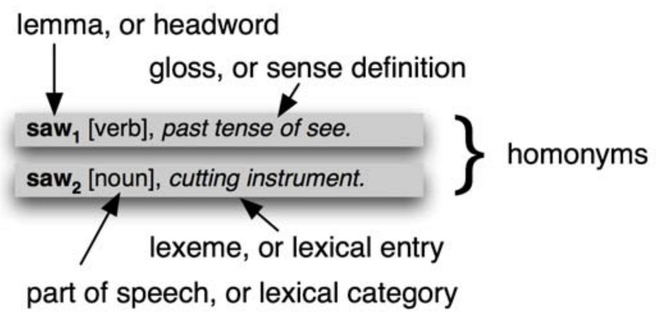

lemma, or headword
gloss, or sense definition
saw₁ [verb], past tense of see.
saw₂ [noun], cutting instrument.
homonyms
lexeme, or lexical entry
part of speech, or lexical category

Figure 2.7: Lexicon Terminology

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 spell checkers. We can use it to find unusual or mis-spelt words in a text corpus, as shown in Figure 2.8.

There is also a corpus of **stopwords**, that is, high-frequency words like *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 fail 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 1.0 * len(content) / len(text)
...
```

```
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', ...]
```

Figure 2.8: Using a Lexical Resource to Filter a 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.

A wordlist is useful for solving word puzzles, such as the one in Figure 2.9. Our program iterates through every word and, for each one, checks whether it meets the conditions. The obligatory letter and length constraint are easy to check (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 latter appear twice (here, the letter *v*). We use the `FreqDist` comparison method 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
...                      and obligatory in w
...                      and nltk.FreqDist(w) <= puzzle_letters]
['glover', 'gorlin', 'govern', 'grovel', 'ignore', 'involver', 'lienor',
'linger', 'longer', 'lovering', 'noiler', 'overling', 'region', 'renvoi',
'revolving', 'ringle', 'roving', 'violer', 'virole']
```

Your Turn: Can you think of an English word that contains *gnt*? Write Python code to find any such words in the wordlist.

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 which appear in both files, i.e. names that are ambiguous for gender:

```
>>> names = nltk.corpus.names
>>> names.files()
('female.txt', 'male.txt')
>>> male_names = names.words('male.txt')
```

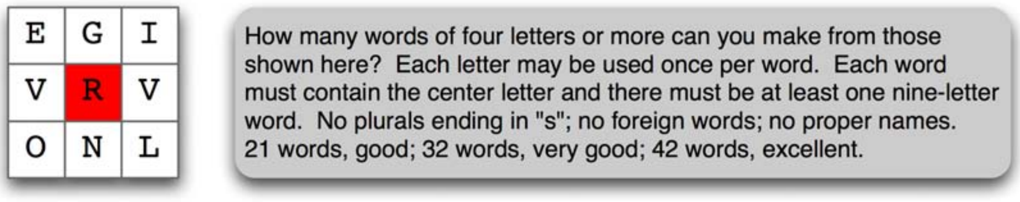

E G I
V R V
O N L
How many words of four letters or more can you make from those shown here? Each letter may be used once per word. Each word must contain the center letter and there must be at least one nine-letter word. No plurals ending in "s"; no foreign words; no proper names.
21 words, good; 32 words, very good; 42 words, excellent.

Figure 2.9: A Word Puzzle Known as "Target"

```
>>> 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 2.10, produced by the following code:

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

A Pronouncing Dictionary

As we have seen, the entries in a wordlist lack internal structure — they are just words. 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 US 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 US 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*

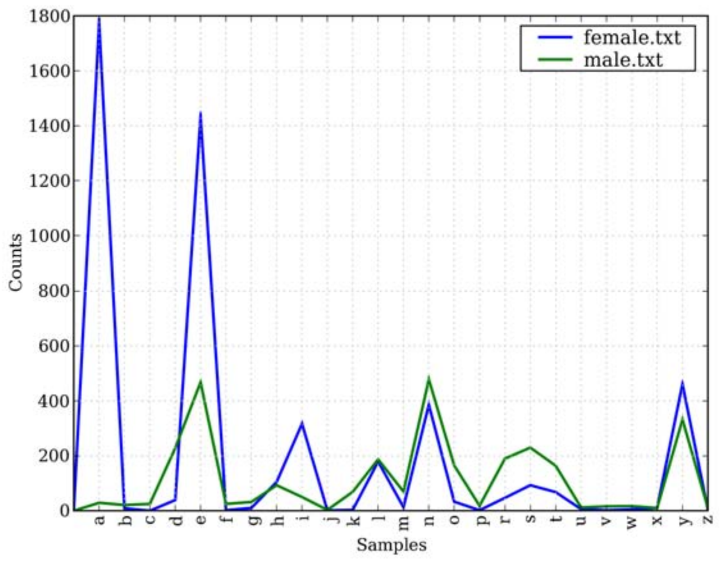


Figure 2.10: Frequency of Final Letter of Female vs Male Names

variable names. 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:
...     if len(pron) == 3:
...         ph1, ph2, ph3 = pron
...         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 above program scans the lexicon looking for entries whose pronunciation consists of three phones (`len(pron) == 3`). If the condition is true, we assign the contents of `pron` to three new variables `ph1`, `ph2` and `ph3`. Notice the unusual form of the statement which does that work: `ph1, ph2, ph3 = pron`.

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 spelt in several ways: *nics*, *niks*, *nix*, 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 [int(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', ...]
```

Note that this example has a user-defined function inside the condition of a list comprehension.

We can use a conditional frequency distribution to help us find minimally-constrasting sets of words. Here we find all the *p*-words, and group them according to their final sound.

```
>>> p3 = [(pron[0]+'-'+pron[2], word) for (word, pron) in entries
...                                   if len(pron) == 3 and pron[0] == 'P']
>>> cfd = nltk.ConditionalFreqDist(p3)
>>> for template in cfd.conditions():
...     if len(cfd[template]) > 10:
...         words = cfd[template].keys()
...         print template, ' '.join(words)
...
P-CH perch puche poche peach petsche poach pietsch putsch pautsch piche petsch pooch pitsch piech pi
P-K pik peek pic pique paque polk perc poke perk pac pock poch purk pak paik puck pake paek peake pe
P-L pil poehl pille pehl pol pall pohl pahl paul perl pale paille perle poll pyle pail peele pearl p
P-N paine payne pon pain pin pawn pinn pun pine paign pen pyne pane penn peine poon pan penh
P-P pap paap pipp paup pape pup pep poop pop pipe paape popp pip peep pope papp poppe
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 purse pease perse poss pus pi
P-T pot puett pit pete putt pat purt pet peart pott pett pait pert pote patt peat pate put piet peet
P-Z pays p.s pao's pais paws p.'s pas pez paz pei's pose poise peas paiz pause p's perz purrs pies p
```

Rather than iterating over the whole dictionary, we can also access it by looking up particular words. (This uses Python's dictionary data structure, which we will study in Section 4.3.)

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

We look up a dictionary by specifying its name, followed by a **key** (such as the word *fire*) inside square brackets: `prondict['fire']`. If we try to look up a non-existent key, we get a `KeyError`, as we did when indexing a list with an integer that was too large. The word *blog* is missing from the pronouncing dictionary, so we tweak our version by assigning a value for this key (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.files()
('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.

```
>>> fr2en = swadesh.entries(['fr', 'en'])
>>> fr2en
[('je', 'I'), ('tu, vous', 'you (singular), thou'), ('il', 'he'), ('nous', 'we'), ...]
>>> 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, 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 will return to Python's dictionary data type `dict()` in Section 4.3.) We can compare words in various Germanic and Romance languages:

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

Resources for Lexical Semantics

Tasks such as word sense disambiguation (Section 1.5) and semantic interpretation more generally (Chapter 10) depend on lexical resources containing rich semantic information. WordNet is the most widely used lexical semantic resource and is discussed in Section 2.5. Other important resources are VerbNet and PropBank, both supported in NLTK. We will discuss these further in chapter [REF].

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, like `('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), `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) by *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**. Let's 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"):

```
>>> wn.synset('car.n.01').lemma_names
['car', 'auto', 'automobile', 'machine', 'motorcar']
```

Each word of a synset can have several meanings, e.g. *car* can also signify a train carriage, a gondola, or an elevator car. However, we are only interested in the single meaning that is common to all words of the above synset. Synsets also come with a prose definition and some example sentences:

```
>>> wn.synset('car.n.01').definition
'a motor vehicle with four wheels; usually propelled by an internal combustion engine'
>>> wn.synset('car.n.01').examples
['he needs a car to get to work']
```

Although these help humans understand the intended meaning of a synset, the *words* of the synset are often more useful for our programs. To eliminate ambiguity, we will identify these words as `car.n.01.automobile`, `car.n.01.motorcar`, and so on. This pairing of a synset with a word is called a lemma, and here's how to access them:

```
>>> wn.synset('car.n.01').lemmas
[Lemma('car.n.01.car'), Lemma('car.n.01.auto'), Lemma('car.n.01.automobile'),
Lemma('car.n.01.machine'), Lemma('car.n.01.motorcar')]
>>> wn.lemma('car.n.01.automobile')
Lemma('car.n.01.automobile')
>>> wn.lemma('car.n.01.automobile').synset
Synset('car.n.01')
>>> wn.lemma('car.n.01.automobile').name
'automobile'
```

Unlike the words *automobile* and *motorcar*, the word *car* itself is ambiguous, having five synsets:

```
>>> wn.synsets('car')
[Synset('car.n.01'), Synset('car.n.02'), Synset('car.n.03'), Synset('car.n.04'),
Synset('cable_car.n.01')]
>>> for synset in wn.synsets('car'):
...     print synset.lemma_names
...
['car', 'auto', 'automobile', 'machine', 'motorcar']
['car', 'railcar', 'railway_car', 'railroad_car']
['car', 'gondola']
['car', 'elevator_car']
['cable_car', 'car']
```

For convenience, we can access all the lemmas involving the word *car* as follows:

```
>>> wn.lemmas('car')
[Lemma('car.n.01.car'), Lemma('car.n.02.car'), Lemma('car.n.03.car'),
Lemma('car.n.04.car'), Lemma('cable_car.n.01.car')]
```

Observe that there is a one-to-one correspondence between the synsets of car and the lemmas of car.

Your Turn: Write down all the senses of the word *dish* that you can think of. Now, explore this word with the help of WordNet, using the same operations we used above.

The WordNet Hierarchy

WordNet synsets correspond to abstract concepts, and they don't always have corresponding words in English. These concepts are linked together in a hierarchy. Some concepts are very general, such as *Entity*, *State*, *Event* — these are called **unique beginners** or root synsets. Others, such as *gas guzzler* and *hatchback*, are much more specific. A small portion of a concept hierarchy is illustrated in Figure 2.11. The edges between nodes indicate the hypernym/hyponym relation...

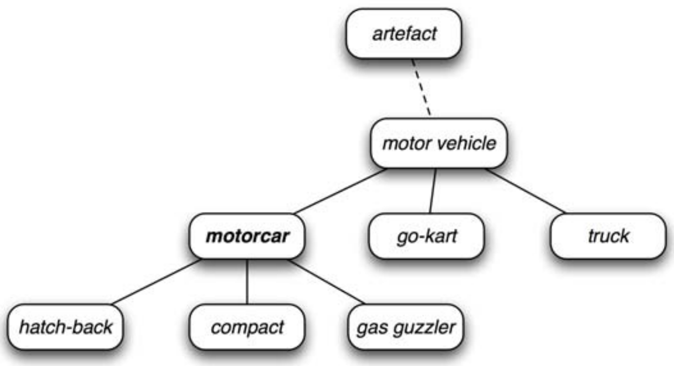


Figure 2.11: Fragment of WordNet Concept Hierarchy

WordNet makes it easy to navigate between concepts. For example, given a concept like *motorcar*, we can look at the concepts that are more specific; the (immediate) **hyponyms**.

```
>>> motorcar = wn.synset('car.n.01')
>>> types_of_motorcar = motorcar.hyponyms()
>>> types_of_motorcar[26]
Synset('ambulance.n.01')
>>> sorted([lemma.name for synset in types_of_motorcar for lemma in synset.lemmas])
['Model_T', 'S.U.V.', 'SUV', 'Stanley_Steamer', 'ambulance', 'beach_waggon',
'beach_wagon', 'bus', 'cab', 'compact', 'compact_car', 'convertible',
'coupe', 'cruiser', 'electric', 'electric_automobile', 'electric_car',
'estate_car', 'gas_guzzler', 'hack', 'hardtop', 'hatchback', 'heap',
'horseless_carriage', 'hot-rod', 'hot_rod', 'jalopy', 'jeep', 'landrover',
'limo', 'limousine', 'loaner', 'minicar', 'minivan', 'pace_car', 'patrol_car',
'phaeton', 'police_car', 'police_cruiser', 'prowl_car', 'race_car', 'racer',
'racing_car', 'roadster', 'runabout', 'saloon', 'secondhand_car', 'sedan',
'sport_car', 'sport_utility', 'sport_utility_vehicle', 'sports_car', 'squad_car',
'station_waggon', 'station_wagon', 'stock_car', 'subcompact', 'subcompact_car',
'taxi', 'taxicab', 'tourer', 'touring_car', 'two-seater', 'used-car', 'waggon', 'wagon']
```

We can also navigate up the hierarchy by visiting hypernyms. Some words have multiple paths, because they can be classified in more than one way. There are two paths between `car.n.01` and `entity.n.01` because `wheeled_vehicle.n.01` can be classified either as a vehicle or as a container.

```
>>> motorcar.hypernyms()
[Synset('motor_vehicle.n.01')]
>>> [synset.name for synset in motorcar.hypernym_paths()[1]]
['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02',
'artifact.n.01', 'instrumentality.n.03', 'conveyance.n.03', 'vehicle.n.01',
'wheeled_vehicle.n.01', 'self-propelled_vehicle.n.01', 'motor_vehicle.n.01',
'car.n.01']
```

We can get the most general hypernyms (or root hypernyms) of a synset as follows:

```
>>> motorcar.root_hypernyms()
[Synset('entity.n.01')]
```

NLTK includes a convenient web-browser interface to WordNet `nltk.wordnet.browser()`

More Lexical Relations

Hypernyms and hyponyms are called lexical "relations" because they relate one synset to another. These two relations navigate up and down the "is-a" hierarchy. Another important way to navigate the WordNet network is from items to their components (**meronyms**) or to the things they are contained in (*holonyms*). For example, the parts of a *tree* are its *trunk*, *crown*, and so on; the `part_meronyms()`. The *substance* a tree is made of include *heartwood* and *sapwood*; the `substance_meronyms()`. A collection of trees forms a *forest*; the `member_holonyms()`:

```
>>> wn.synset('tree.n.01').part_meronyms()
[Synset('burl.n.02'), Synset('crown.n.07'), Synset('stump.n.01'),
Synset('trunk.n.01'), Synset('limb.n.02')]
>>> wn.synset('tree.n.01').substance_meronyms()
[Synset('heartwood.n.01'), Synset('sapwood.n.01')]
>>> wn.synset('tree.n.01').member_holonyms()
[Synset('forest.n.01')]
```

To see just how intricate things can get, consider the word *mint*, which has several closely-related senses. We can see that `mint.n.04` is part of `mint.n.02` and the substance from which `mint.n.05` is made.

```
>>> for synset in wn.synsets('mint', wn.NOUN):
...     print synset.name + ':', synset.definition
...
batch.n.02: (often followed by `of') a large number or amount or extent
mint.n.02: any north temperate plant of the genus Mentha with aromatic leaves and small mauve flower
mint.n.03: any member of the mint family of plants
mint.n.04: the leaves of a mint plant used fresh or candied
mint.n.05: a candy that is flavored with a mint oil
mint.n.06: a plant where money is coined by authority of the government
>>> wn.synset('mint.n.04').part_holonyms()
[Synset('mint.n.02')]
>>> wn.synset('mint.n.04').substance_holonyms()
[Synset('mint.n.05')]
```

There are also relationships between verbs. For example, the act of *walking* involves the act of *stepping*, so *walking* **entails** *stepping*. Some verbs have multiple entailments:

```
>>> wn.synset('walk.v.01').entailments()
[Synset('step.v.01')]
>>> wn.synset('eat.v.01').entailments()
[Synset('swallow.v.01'), Synset('chew.v.01')]
>>> wn.synset('tease.v.03').entailments()
[Synset('arouse.v.07'), Synset('disappoint.v.01')]
```

Some lexical relationships hold between lemmas, e.g. antonymy:

```
>>> wn.lemma('supply.n.02.supply').antonyms()
[Lemma('demand.n.02.demand')]
>>> wn.lemma('rush.v.01.rush').antonyms()
[Lemma('linger.v.04.linger')]
>>> wn.lemma('horizontal.a.01.horizontal').antonyms()
[Lemma('vertical.a.01.vertical'), Lemma('inclined.a.02.inclined')]
>>> wn.lemma('staccato.r.01.staccato').antonyms()
[Lemma('legato.r.01.legato')]
```

Semantic Similarity

We have seen that synsets are linked by a complex network of lexical relations. Given a particular synset, we can traverse the WordNet network to find synsets with related meanings. Knowing which words are semantically related is useful for indexing a collection of texts, so that a search for a general term like *vehicle* will match documents containing specific terms like *limousine*.

Recall that each synset has one or more hypernym paths that link it to a root hypernym such as `entity.n.01`. Two synsets linked to the same root may have several hypernyms in common. If two synsets share a very specific hypernym — one that is low down in the hypernym hierarchy — they must be closely related.

```
>>> orca = wn.synset('orca.n.01')
>>> minke = wn.synset('minke_whale.n.01')
>>> tortoise = wn.synset('tortoise.n.01')
>>> novel = wn.synset('novel.n.01')
>>> orca.lowest_common_hypernyms(minke)
[Synset('whale.n.02')]
```

```
>>> orca.lowest_common_hypernyms(tortoise)
[Synset('vertebrate.n.01')]
>>> orca.lowest_common_hypernyms(novel)
[Synset('entity.n.01')]
```

Of course we know that *whale* is very specific, *vertebrate* is more general, and *entity* is completely general. We can quantify this concept of generality by looking up the depth of each synset:

```
>>> wn.synset('whale.n.02').min_depth()
13
>>> wn.synset('vertebrate.n.01').min_depth()
8
>>> wn.synset('entity.n.01').min_depth()
0
```

The WordNet package includes a variety of sophisticated measures that incorporate this basic insight. For example, `path_similarity` assigns a score in the range `0`–`1`, based on the shortest path that connects the concepts in the hypernym hierarchy (`-1` is returned in those cases where a path cannot be found). Comparing a synset with itself will return `1`.

```
>>> orca.path_similarity(minke)
0.14285714285714285
>>> orca.path_similarity(tortoise)
0.071428571428571425
>>> orca.path_similarity(novel)
0.041666666666666664
```

This is a convenient interface, and gives us the same relative ordering as before. Several other similarity measures are available (see `help(wn)`).

NLTK also includes VerbNet, a hierarhical verb lexicon linked to WordNet. It can be accessed with `nltk.corpus.verbnet`.

2.6 Summary

- A text corpus is a large, structured collection of texts. NLTK comes with many corpora, e.g. the Brown Corpus, `nltk.corpus.brown`.
- Some text corpora are categorized, e.g. by genre or topic; sometimes the categories of a corpus overlap each other.
- To find out about some variable `v` that you have created, type `help(v)` to read the help entry for this kind of object.
- Some functions are not available by default, but must be accessed using Python's `import` statement.
- Exploratory data analysis, a technique for learning about a specific linguistic pattern, consists of four steps: search, categorization, counting, and hypothesis refinement.

2.7 Further Reading (NOTES)

Natural Language Processing

Several websites have useful information about NLP, including conferences, resources, and special-interest groups, e.g. `www.lt-world.org`, `www.aclweb.org`, `www.elsnet.org`. The website of the *Association for Computational Linguistics*, at `www.aclweb.org`, contains an overview of computational linguistics, including copies of introductory chapters from recent textbooks. Wikipedia has entries for NLP and its subfields (but don't confuse natural language processing with the other NLP: neuro-linguistic programming.) The new, second edition of *Speech and Language Processing*, is a more advanced textbook that builds on the material presented here. Three books provide comprehensive surveys of the field: [Cole, 1997], [Dale, Moisl, & Somers, 2000], [Mitkov, 2002]. Several NLP systems have online interfaces that you might like to experiment with, e.g.:

- WordNet: `http://wordnet.princeton.edu/`
- Translation: `http://world.altavista.com/`
- ChatterBots: `http://www.loebner.net/Prizef/loebner-prize.html`
- Question Answering: `http://www.answerbus.com/`
- Summarization: `http://newsblaster.cs.columbia.edu/`

Python

[Rossum & Drake, 2006] is a Python tutorial by Guido van Rossum, the inventor of Python and Fred Drake, the official editor of the Python documentation. It is available online at `http://docs.python.org/tut/tut.html`. A more detailed but still introductory text is [Lutz & Ascher, 2003], which covers the essential features of Python, and also provides an overview of the standard libraries. A more advanced text, [Rossum & Drake, 2006] is the official reference for the Python language itself, and describes the syntax of Python and its built-in datatypes in depth. It is also available online at `http://docs.python.org/ref/ref.html`. [Beazley, 2006] is a succinct reference book; although not suitable as an introduction to Python, it is an excellent resource for intermediate and advanced programmers. Finally, it is always worth checking the official *Python Documentation* at http://docs.python.org/.

Two freely available online texts are the following:

- Josh Cogliati, *Non-Programmer's Tutorial for Python*, `http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python/Contents`
- Jeffrey Elkner, Allen B. Downey and Chris Meyers, *How to Think Like a Computer Scientist: Learning with Python* (Second Edition), `http://openbookproject.net/thinkCSpy/`

Learn more about functions in Python by reading Chapter 4 of [Lutz & Ascher, 2003].

Archives of the CORPORA mailing list.

[Woods, Fletcher, & Hughes, 1986]

LDC, ELRA

The online API documentation at http://www.nltk.org/ contains extensive reference material for all NLTK modules.

Although WordNet was originally developed for research in psycholinguistics, it is widely used in NLP and Information Retrieval. WordNets are being developed for many other languages, as documented at `http://www.globalwordnet.org/`.

For a detailed comparison of wordnet similarity measures, see [Budanitsky & Hirst, 2006].

2.8 Exercises

1. ☼ How many words are there in `text2`? How many distinct words are there?

2. ☼ Compare the lexical diversity scores for humor and romance fiction in Table 1.1. Which genre is more lexically diverse?

3. ☼ Produce a dispersion plot of the four main protagonists in *Sense and Sensibility*: Elinor, Marianne, Edward, Willoughby. What can you observe about the different roles played by the males and females in this novel? Can you identify the couples?

4. ☼ According to Strunk and White's *Elements of Style*, the word *however*, used at the start of a sentence, means "in whatever way" or "to whatever extent", and not "nevertheless". They give this example of correct usage: *However you advise him, he will probably do as he thinks best.* (http://www.bartleby.com/141/strunk3.html) Use the concordance tool to study actual usage of this word in the various texts we have been considering.

5. ☼ Create a variable `phrase` containing a list of words. Experiment with the operations described in this chapter, including addition, multiplication, indexing, slicing, and sorting.

6. ☼ The first sentence of `text3` is provided to you in the variable `sent3`. The index of *the* in `sent3` is 1, because `sent3[1]` gives us `'the'`. What are the indexes of the two other occurrences of this word in `sent3`?

7. ☼ Using the Python interactive interpreter, experiment with the examples in this section. Think of a short phrase and represent it as a list of strings, e.g. ['Monty', 'Python']. Try the various operations for indexing, slicing and sorting the elements of your list.

8. ☼ Investigate the holonym / meronym relations for some nouns. Note that there are three kinds (member, part, substance), so access is more specific, e.g., `wordnet.MEMBER_MERONYM`, `wordnet.SUBSTANCE_HOLONYM`.

9. ☼ The polysemy of a word is the number of senses it has. Using WordNet, we can determine that the noun *dog* has 7 senses with: `len(nltk.wordnet.N['dog'])`. Compute the average polysemy of nouns, verbs, adjectives and adverbs according to WordNet.

10. ☼ Using the Python interpreter in interactive mode, experiment with the dictionary examples in this chapter. Create a dictionary `d`, and add some entries. What happens if you try to access a non-existent entry, e.g. `d['xyz']`?

11. ☼ Try deleting an element from a dictionary, using the syntax `del d['abc']`. Check that the item was deleted.

12. ☼ Create a dictionary `e`, to represent a single lexical entry for some word of your choice. Define keys like `headword`, `part-of-speech`, `sense`, and `example`, and assign them suitable values.

13. ☼ Try the examples in this section, then try the following.

    a) Create a variable called `msg` and put a message of your own in this variable. Remember that strings need to be quoted, so you will need to type something like: `msg = "I like NLP!"`

    b) Now print the contents of this variable in two ways, first by simply typing the variable name and pressing enter, then by using the `print` statement.

    c) Try various arithmetic expressions using this string, e.g. `msg + msg`, and `5 * msg`.

    d) Define a new string `hello`, and then try `hello + msg`. Change the `hello` string so that it ends with a space character, and then try `hello + msg` again.

14. ☼ Consider the following two expressions which have the same result. Which one will typically be more relevant in NLP? Why?

    a) `"Monty Python"[6:12]`

    b) `["Monty", "Python"][1]`

15. ☼ Define a string `s = 'colorless'`. Write a Python statement that changes this to “colourless” using only the slice and concatenation operations.

16. ☼ Try the slice examples from this section using the interactive interpreter. Then try some more of your own. Guess what the result will be before executing the command.

17. ☼ We can use the slice notation to remove morphological endings on words. For example, `'dogs'[:-1]` removes the last character of `dogs`, leaving `dog`. Use slice notation to remove the affixes from these words (we’ve inserted a hyphen to indicate the affix boundary, but omit this from your strings): `dish-es`, `run-ning`, `nation-ality`, `un-do`, `pre-heat`.

18. ☼ We saw how we can generate an `IndexError` by indexing beyond the end of a string. Is it possible to construct an index that goes too far to the left, before the start of the string?

19. ☼ We can also specify a “step” size for the slice. The following returns every second character within the slice: `msg[6:11:2]`. It also works in the reverse direction: `msg[10:5:-2]` Try these for yourself, then experiment with different step values.

20. ☼ What happens if you ask the interpreter to evaluate `msg[::-1]`? Explain why this is a reasonable result.

21. ☼ Define a conditional frequency distribution over the Names corpus that allows you to see which initial letters are more frequent for males vs females (cf. Figure 2.10).

22. ☼ Use the corpus module to read `austen-persuasion.txt`. How many word tokens does this book have? How many word types?

23. ☼ Use the Brown corpus reader `nltk.corpus.brown.words()` or the Web text corpus reader `nltk.corpus.webtext.words()` to access some sample text in two different genres.

24. ☼ Read in the texts of the *State of the Union* addresses, using the `state_union` corpus reader. Count occurrences of `men`, `women`, and `people` in each document. What has happened to the usage of these words over time?

25. ◑ Consider the following Python expression: `len(set(text4))`. State the purpose of this expression. Describe the two steps involved in performing this computation.

26. ◑ Pick a pair of texts and study the differences between them, in terms of vocabulary, vocabulary richness, genre, etc. Can you find pairs of words which have quite different meanings across the two texts, such as *monstrous* in *Moby Dick* and in *Sense and Sensibility*?

27. ◑ Use `text9.index(??)` to find the index of the word *sunset*. By a process of trial and error, find the slice for the complete sentence that contains this word.

28. ◑ Using list addition, and the `set` and `sorted` operations, compute the vocabulary of the sentences `sent1` ... `sent8`.

29. ◑ What is the difference between `sorted(set(w.lower() for w in text1))` and `sorted(w.lower() for w in set(text1))`? Which one will gives a larger value? Will this be the case for other texts?

30. ◑ Write the slice expression to produces the last two words of `text2`.

31. ◑ Read the BBC News article: *UK's Vicky Pollards 'left behind'* `http://news.bbc.co.uk/1/hi/education/6173441.stm`. The article gives the following statistic about teen language: "the top 20 words used, including yeah, no, but and like, account for around a third of all words." How many word types account for a third of all word tokens, for a variety of text sources? What do you conclude about this statistic? Read more about this on *LanguageLog*, at `http://itre.cis.upenn.edu/~myl/languagelog/archives/003993.html`.

32. ◑ Assign a new value to `sent`, namely the sentence `['she', 'sells', 'sea', 'shells', 'by', 'the', 'sea', 'shore']`, then write code to perform the following tasks:

    a) Print all words beginning with `'sh'`:

    b) Print all words longer than 4 characters.

33. ◑ What does the following Python code do? `sum(len(w) for w in text1)` Can you use it to work out the average word length of a text?

34. ◑ What is the difference between the following two tests: `w.isupper()`, `not w.islower()`?

35. ◑ Investigate the table of modal distributions and look for other patterns. Try to explain them in terms of your own impressionistic understanding of the different genres. Can you find other closed classes of words that exhibit significant differences across different genres?

36. ◑ The CMU Pronouncing Dictionary contains multiple pronunciations for certain words. How many distinct words does it contain? What fraction of words in this dictionary have more than one possible pronunciation?

37. ◑ What is the branching factor of the noun hypernym hierarchy? (For all noun synsets that have hyponyms, how many do they have on average?)

38. ◑ Define a function `supergloss(s)` that takes a synset `s` as its argument and returns a string consisting of the concatenation of the glosses of `s`, all hypernyms of `s`, and all hyponyms of `s`.

39. ☺ Review the mappings in Table 4.4. Discuss any other examples of mappings you can think of. What type of information do they map from and to?

40. ◑ Write a program to find all words that occur at least three times in the Brown Corpus.

41. ◑ Write a program to generate a table of token/type ratios, as we saw in Table 1.1. Include the full set of Brown Corpus genres (`nltk.corpus.brown.categories()`). Which genre has the lowest diversity (greatest number of tokens per type)? Is this what you would have expected?

42. ◑ Modify the text generation program in Figure 2.5 further, to do the following tasks:

    a) Store the *n* most likely words in a list `lwords` then randomly choose a word from the list using `random.choice()`.
    b) Select a particular genre, such as a section of the Brown Corpus, or a genesis translation, one of the Gutenberg texts, or one of the Web texts. Train the model on this corpus and get it to generate random text. You may have to experiment with different start words. How intelligible is the text? Discuss the strengths and weaknesses of this method of generating random text.
    c) Now train your system using two distinct genres and experiment with generating text in the hybrid genre. Discuss your observations.

43. ◑ Write a program to print the most frequent bigrams (pairs of adjacent words) of a text, omitting non-content words, in order of decreasing frequency.

44. ◑ Write a program to create a table of word frequencies by genre, like the one given above for modals. Choose your own words and try to find words whose presence (or absence) is typical of a genre. Discuss your findings.

45. ◑ Write a function that finds the 50 most frequently occurring words of a text that are not stopwords.

46. ◑ Write a function `tf()` that takes a word and the name of a section of the Brown Corpus as arguments, and computes the text frequency of the word in that section of the corpus.

47. ◑ Write a program to guess the number of syllables contained in a text, making use of the CMU Pronouncing Dictionary.

48. ◑ Define a function `hedge(text)` which processes a text and produces a new version with the word `'like'` between every third word.

49. ★ **Zipf's Law**: Let *f(w)* be the frequency of a word *w* in free text. Suppose that all the words of a text are ranked according to their frequency, with the most frequent word first. Zipf's law states that the frequency of a word type is inversely proportional to its rank (i.e. $f.r = k$, for some constant k). For example, the 50th most common word type should occur three times as frequently as the 150th most common word type.

    a) Write a function to process a large text and plot word frequency against word rank using `pylab.plot`. Do you confirm Zipf's law? (Hint: it helps to use a logarithmic scale). What is going on at the extreme ends of the plotted line?
    b) Generate random text, e.g. using `random.choice("abcdefg ")`, taking care to include the space character. You will need to `import random` first. Use the string concatenation operator to accumulate characters into a (very) long string. Then tokenize this string, and generate the Zipf plot as before, and compare the two plots. What do you make of Zipf's Law in the light of this?

50. ★ Modify the `generate_model()` function in Figure 2.5 to use Python's `random.choose()` method to randomly pick the next word from the available set of words.

51. ★ Define a function `find_language()` that takes a string as its argument, and returns a list of languages that have that string as a word. Use the `udhr` corpus and limit your searches to files in the Latin-1 encoding.

52. ★ Use one of the predefined similarity measures to score the similarity of each of the following pairs of words. Rank the pairs in order of decreasing similarity. How close is your ranking to the order given here? (Note that this order was established experimentally by [Miller & Charles, 1998].)

:: car-automobile, gem-jewel, journey-voyage, boy-lad, coast-shore, asylum-madhouse, magician-wizard, midday-noon, furnace-stove, food-fruit, bird-cock, bird-crane, tool-implement, brother-monk, lad-brother, crane-implement, journey-car, monk-oracle, cemetery-woodland, food-rooster, coast-hill, forest-graveyard, shore-woodland, monk-slave, coast-forest, lad-wizard, chord-smile, glass-magician, rooster-voyage, noon-string.

Chapter 3

Processing Raw Text

The most important source of texts is undoubtedly the Web. Its convenient to have existing text collections to explore, such as the corpora we saw in the previous chapters. However, you probably have your own text sources in mind, and need to learn how to access them.

The goal of this chapter is to answer the following questions:

1. How can we write programs to access text from local files and from the web, in order to get hold of an unlimited range of language material?
2. How can we split documents up into individual words and punctuation symbols, so we can do the same kinds of analysis we did with text corpora in earlier chapters?
3. What features of the Python programming language are needed to do this?

In order to address these questions, we will be covering key concepts in NLP, including tokenization and stemming. Along the way you will consolidate your Python knowledge and learn about strings, files, and regular expressions. Since so much text on the web is in HTML format, we will also see how to dispense with markup.

From this chapter onwards, our program samples will assume you begin your interactive session or your program with the following import statment: `import nltk, re, pprint`

3.1 Accessing Text from the Web and from Disk

Electronic Books

A small sample of texts from Project Gutenberg appears in the NLTK corpus collection. However, you may be interested in analyzing other texts from Project Gutenberg. You can browse the catalog of 25,000 free online books at `http://www.gutenberg.org/catalog/`, and obtain a URL to an ASCII text file. Although 90% of the texts in Project Gutenberg are in English, it includes material in over 50 other languages, including Catalan, Chinese, Dutch, Finnish, French, German, Italian, Portuguese and Spanish (with more than 100 texts each).

Text number 2554 is an English translation of *Crime and Punishment*, and we can access it as follows:

```
>>> from urllib import urlopen
>>> url = "http://www.gutenberg.org/files/2554/2554.txt"
>>> raw = urlopen(url).read()
>>> type(raw)
<type 'str'>
>>> len(raw)
1176831
>>> raw[:75]
'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r\n'
```

The `read()` process will take a few seconds as it downloads this large book. If you're using an internet proxy which is not correctly detected by Python, you may need to specify the proxy manually as follows:

```
>>> proxies = {'http': 'http://www.someproxy.com:3128'}
>>> raw = urllib.urlopen(url, proxies=proxies).read()
```

The variable `raw` contains a string with 1,176,831 characters. This is the raw content of the book, including many details we are not interested in such as whitespace, line breaks and blank lines. Instead, we want to break it up into words and punctuation, as we saw in Chapter 1. This step is called **tokenization**, and it produces our familiar structure, a list of words and punctuation. From now on we will call these **tokens**.

```
>>> text = nltk.wordpunct_tokenize(raw)
>>> type(text)
<class 'nltk.text.Text'>
>>> len(text)
255809
>>> text[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime', 'and', 'Punishment', ',', 'by']
```

If we now take the further step of creating an NLTK text from this list, we can carry out all of the other linguistic processing we saw in Chapter 1, along with the regular list operations like slicing:

```
>>> text = nltk.Text(tokens)
>>> type(text)
<type 'nltk.text.Text'>
>>> text[1020:1060]
['CHAPTER', 'I', 'On', 'an', 'exceptionally', 'hot', 'evening', 'early', 'in',
'July', 'a', 'young', 'man', 'came', 'out', 'of', 'the', 'garret', 'in',
'which', 'he', 'lodged', 'in', 'S', '.', 'Place', 'and', 'walked', 'slowly',
',', 'as', 'though', 'in', 'hesitation', ',', 'towards', 'K', '.', 'bridge', '.']
>>> text.collocations()
Katerina Ivanovna; Pulcheria Alexandrovna; Avdotya Romanovna; Pyotr
Petrovitch; Project Gutenberg; Marfa Petrovna; Rodion Romanovitch;
Sofya Semyonovna; Nikodim Fomitch; did not; Hay Market; Andrey
Semyonovitch; old woman; Literary Archive; Dmitri Prokofitch; great
deal; United States; Praskovya Pavlovna; Porfiry Petrovitch; ear rings
```

Notice that *Project Gutenberg* appears as a collocation. This is because each text downloaded from Project Gutenberg contains a header with the name of the text, the author, the names of people who scanned and corrected the text, a license, and so on. Sometimes this information appears in a footer at the end of the file. We cannot reliably detect where the content begins and ends, and so have to resort to manual inspection of the file, to discover unique strings that mark the beginning and the end, before trimming `raw` to be just the content and nothing else:

```
>>> raw.find("PART I")
```

```
5303
>>> raw.rfind("End of Project Gutenberg's Crime")
1157681
>>> raw = raw[5303:1157681]
```

The `find()` and `rfind()` ("reverse find") functions help us get the right index values. Now the raw text begins with "PART I", and goes up to (but not including) the phrase that marks the end of the content.

This was our first brush with reality: texts found on the web may contain unwanted material, and there may not be an automatic way to remove it. But with a small amount of extra work we can extract the material we need.

Dealing with HTML

Much of the text on the web is in the form of HTML documents. You can use a web browser to save a page as text to a local file, then access this as described in the section on files below. However, if you're going to do this a lot, its easiest to get Python to do the work directly. The first step is the same as before, using `urlopen`. For fun we'll pick a BBC News story called *Blondes to die out in 200 years*, an urban legend reported as established scientific fact:

```
>>> url = "http://news.bbc.co.uk/2/hi/health/2284783.stm"
>>> html = urlopen(url).read()
>>> html[:60]
'<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN'
```

You can type `print html` to see the HTML content in all its glory, including meta tags, an image map, JavaScript, forms, and tables.

Getting text out of HTML is a sufficiently common task that NLTK provides a helper function `nltk.clean_html()`, which takes an HTML string and returns raw text. We can then tokenize this to get our familiar text structure:

```
>>> raw = nltk.clean_html(html)
>>> tokens = nltk.wordpunct_tokenize(raw)
>>> tokens
['BBC', 'NEWS', '|', 'Health', '|', 'Blondes', "'", 'to', 'die', 'out', ...]
```

This still contains unwanted material concerning site navigation and related stories. With some trial and error you can find the start and end indexes of the content and select the tokens of interest, and initialize a text as before.

```
>>> tokens = tokens[96:399]
>>> text = nltk.Text(tokens)
>>> text.concordance('gene')
 they say too few people now carry the gene for blondes to last beyond the next tw
t blonde hair is caused by a recessive gene . In order for a child to have blonde
to have blonde hair , it must have the gene on both sides of the family in the gra
there is a disadvantage of having that gene or by chance . They don ' t disappear
ondes would disappear is if having the gene was a disadvantage and I do not think
```

For more sophisticated processing of HTML, use the *Beautiful Soup* package, available from `http://www.crummy.com/software/BeautifulSoup/`

Processing Search Engine Results

The web can be thought of as a huge corpus of unannotated text. Web search engines provide an efficient means of searching this large quantity of text for relevant linguistic examples. The main advantage or search engines is size: since you are searching such a large set of documents, you are more likely to find any linguistic pattern you are interested in. Furthermore, you can make use of very specific patterns, which would only match one or two examples on a smaller example, but which might match tens of thousands of examples when run on the web. A second advantage of web search engines is that they are very easy to use. Thus, they provide a very convenient tool for quickly checking a theory, to see if it is reasonable.

[Accessing a search engine programmatically: search results; counts; Python code to produce the contents of Table 3.1; mention Google API and xref to discussion of this in Chapter 11.]

Google hits	***adore***	***love***	***like***	***prefer***
absolutely	289,000	905,000	16,200	644
definitely	1,460	51,000	158,000	62,600
ratio	198:1	18:1	1:10	1:97

Table 3.1: *Absolutely* vs *Definitely* (Liberman 2005, LanguageLog.org)

Unfortunately, search engines have some significant shortcomings. First, the allowable range of search patterns is severely restricted. Unlike local corpora, where you write programs to search for arbitrarily complex patterns, search engines generally only allow you to search for individual words or strings of words, sometimes with wildcards. Second, search engines give inconsistent results, and can give widely different figures when used at different times or in different locations. When content has been duplicated across multiple sites, search results may be boosted. Finally, the markup the result returned by a search engine may change unpredictably, breaking any pattern-based method of locating particular content.

Your Turn: Search the web for `"the of"` (inside quotes). Based on the large count, can we conclude that *the of* is a frequent collocation in English?

Processing RSS Feeds

The blogosphere is an important source of text, in both formal and informal registers. With the help of a third-party Python library called the *Universal Feed Parser*, freely downloadable from `http://feedparser.org/`, we can easily access the content of a blog, as shown below:

```
>>> import feedparser
>>> llog = feedparser.parse("http://languagelog.ldc.upenn.edu/nll/?feed=atom")
>>> llog['feed']['title']
u'Language Log'
>>> len(llog.entries)
15
>>> post = llog.entries[2]
>>> post.title
u"He's My BF"
```

```
>>> content = post.content[0].value
>>> content[:70]
u'<p>Today I was chatting with three of our visiting graduate students f'
>>> nltk.wordpunct_tokenize(nltk.html_clean(content))
>>> nltk.wordpunct_tokenize(nltk.clean_html(llog.entries[2].content[0].value))
[u'Today', u'I', u'was', u'chatting', u'with', u'three', u'of', u'our', u'visiting',
u'graduate', u'students', u'from', u'the', u'PRC', u'.', u'Thinking', u'that', u'I',
u'was', u'being', u'au', u'courant', u',', u'I', u'mentioned', u'the', u'expression',
u'DUI4XIANG4', u'\u5c0d\u8c61', u'("', u'boy', u'/', u'girl', u'friend', u'"', ...]
```

Reading Local Files

Your Turn: Create a file called `document.txt` using a text editor, and type in a few lines of text, and save it as plain text. If you are using IDLE, select the *New Window* command in the *File* menu, typing the required text into this window, and then saving the file as `doc.txt` inside the directory that IDLE offers in the pop-up dialogue box. Next, in the Python interpreter, open the file using `f = open('doc.txt')`, then inspect its contents using `print f.read()`.

Various things might have gone wrong when you tried this. If the interpreter couldn't find your file, you would have seen an error like this:

```
>>> f = open('document.txt')
Traceback (most recent call last):
File "<pyshell#7>", line 1, in -toplevel-
f = open('document.txt')
IOError: [Errno 2] No such file or directory: 'document.txt'
```

To check that the file that you are trying to open is really in the right directory, use IDLE's *Open* command in the *File* menu; this will display a list of all the files in the directory where IDLE is running. An alternative is to examine the current directory from within Python:

```
>>> import os
>>> os.listdir('.')
```

Another possible problem you might have encountered when accessing a text file is the newline conventions, which are different for different operating systems. The built-in `open()` function has a second parameter for controlling how the file is opened: `open('document.txt', 'rU')` — `'r'` means to open the file for reading (the default), and `'U'` stands for "Universal", which lets us ignore the different conventions used for marking newlines.

Assuming that you can open the file, there are several methods for reading it. The `read()` method creates a string with the contents of the entire file:

```
>>> f.read()
'Time flies like an arrow.\nFruit flies like a banana.\n'
```

Recall that the `'\n'` characters are **newlines**; this is equivalent to pressing *Enter* on a keyboard and starting a new line.

We can also read a file one line at a time using a `for` loop:

```
>>> f = open('document.txt', 'rU')
>>> for line in f:
...     print line.strip()
Time flies like an arrow.
Fruit flies like a banana.
```

Here we use the `strip()` function to remove the newline character at the end of the input line.

NLTK's corpus files can also be accessed using these methods. We simply have to use `nltk.data.find()` to get the filename for any corpus item. Then we can open it in the usual way:

```
>>> file = nltk.data.find('corpora/gutenberg/melville-moby_dick.txt')
>>> raw = open(file, 'rU').read()
```

Extracting Text from PDF, MSWord and other Binary Formats

ASCII text and HTML text are human readable formats. Text often comes in binary formats — like PDF and MSWord — that can only be opened using specialized software. Third-party libraries such as `pypdf` and `pywin32` can be used to access these formats. Extracting text from multi-column documents can be particularly challenging. For once-off conversion of a few documents, it is simpler to open the document with a suitable application, then save it as text to your local drive, and access it as described below. If the document is already on the web, you can enter its URL in Google's search box. The search result often includes a link to an HTML version of the document, which you can save as text.

Getting User Input

Another source of text is a user interacting with our program. We can prompt the user to type a line of input using the Python function `raw_input()`. We can save that to a variable and manipulate it just as we have done for other strings.

```
>>> s = raw_input("Enter some text: ")
Enter some text: On an exceptionally hot evening early in July
>>> print "You typed", len(nltk.wordpunct_tokenize(s)), "words."
You typed 8 words.
```

Summary

Figure 3.1 summarizes what we have covered in this section, including the process of building a vocabulary that we saw in Chapter 1. (One step, normalization, will be discussed in section 3.5).

There's a lot going on in this pipeline. To understand it properly, it helps to be clear about the type of each variable that it mentions. We find out the type of any Python object `x` using `type(x)`, e.g. `type(1)` is `<int>` since `1` is an integer.

When we load the contents of a URL or file, and when we strip out HTML markup, we are dealing with strings, Python's `<str>` data type (We will learn more about strings in section 3.2):

```
>>> raw = open('document.txt').read()
>>> type(raw)
<type 'str'>
```

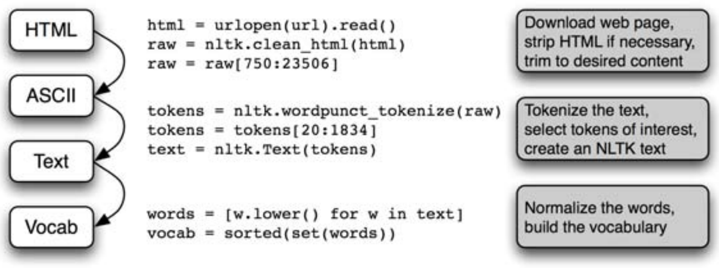

HTML
ASCII
Text
Vocab
html = urlopen(url).read()
raw = nltk.clean_html(html)
raw = raw[750:23506]
tokens = nltk.wordpunct_tokenize(raw)
tokens = tokens[20:1834]
text = nltk.Text(tokens)
words = [w.lower() for w in text]
vocab = sorted(set(words))
Download web page, strip HTML if necessary, trim to desired content
Tokenize the text, select tokens of interest, create an NLTK text
Normalize the words, build the vocabulary

Figure 3.1: The Processing Pipeline

When we tokenize a string we produce a list (of words), and this is Python's `<list>` type. Normalizing and sorting lists produces other lists:

```
>>> tokens = nltk.wordpunct_tokenize(raw)
>>> type(tokens)
<type 'list'>
>>> words = [w.lower() for w in tokens]
>>> type(words)
<type 'list'>
>>> vocab = sorted(set(words))
>>> type(vocab)
<type 'list'>
```

The type of an object determines what operations you can perform on it. So, for example, we can append to a list but not to a string:

```
>>> vocab.append('blog')
>>> raw.append('blog')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```

Similarly, we can concatenate strings with strings, and lists with lists, but we cannot concatenate strings with lists:

```
>>> query = 'Who knows?'
>>> beatles = ['john', 'paul', 'george', 'ringo']
>>> query + beatles
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'list' objects
```

You may also have noticed that our analogy between operations on strings and numbers works for multiplication and addition, but not subtraction or division:

```
>>> 'very' - 'y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>> 'very' / 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

These error messages are another example of Python telling us that we have got our data types in a muddle. In the first case, we are told that the operation of substraction (i.e., −) cannot apply to objects of type `str` (strings), while in the second, we are told that division cannot take `str` and `int` as its two operands.

3.2 Strings: Text Processing at the Lowest Level

It's time to study a fundamental data type that we've been studiously avoiding so far. In earlier chapters we focussed on a text as a list of words. We didn't look too closely at words and how they are handled in the programming language. By using NLTK's corpus interface we were able to ignore the files that these texts had come from. The contents of a word, and of a file, are represented by programming languages as a fundamental data type known as a **string**. In this section we explore strings in detail, and show the connection between strings, words, texts and files.

Basic Operations with Strings

```
>>> monty = 'Monty Python'
>>> monty
'Monty Python'
>>> circus = 'Monty Python's Flying Circus'
  File "<stdin>", line 1
    circus = 'Monty Python's Flying Circus'
                           ^
SyntaxError: invalid syntax
>>> circus = "Monty Python's Flying Circus"
>>> circus
"Monty Python's Flying Circus"
```

The + operation can be used with strings, and is known as **concatenation**. It produces a new string that is a copy of the two original strings pasted together end-to-end. Notice that concatenation doesn't do anything clever like insert a space between the words. The Python interpreter has no way of knowing that you want a space; it does *exactly* what it is told. Given the example of +, you might be able guess what multiplication will do:

```
>>> 'very' + 'very' + 'very'
'veryveryvery'
>>> 'very' * 3
'veryveryvery'
```

Caution!
Be careful to distinguish between the string `' '`, which is a single whitespace character, and `''`, which is the empty string.

Printing Strings

So far, when we have wanted to look at the contents of a variable or see the result of a calculation, we have just typed the variable name into the interpreter. We can also see the contents of a variable using the `print` statement:

```
>>> print monty
Monty Python
```

Notice that there are no quotation marks this time. When we inspect a variable by typing its name in the interpreter, the interpreter prints the Python representation of its value. Since it's a string, the result is quoted. However, when we tell the interpreter to print the contents of the variable, we don't see quotation characters since there are none inside the string.

The `print` statement allows us to display more than one item on a line in various ways, as shown below:

```
>>> grail = 'Holy Grail'
>>> print monty + grail
Monty PythonHoly Grail
>>> print monty, grail
Monty Python Holy Grail
>>> print monty, "and the", grail
Monty Python and the Holy Grail
```

Accessing Individual Characters

As we saw in Section 1.2 for lists, strings are indexed, starting from zero. When we index a string, we get one of its characters (or letters):

```
>>> monty[0]
'M'
>>> monty[3]
't'
>>> monty[5]
' '
```

As with lists, if we try to access an index that is outside of the string we get an error:

```
>>> monty[20]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Again as with lists, we can use negative indexes for strings, where `-1` is the index of the last character. Using positive and negative indexes, we have two ways to refer to any position in a string. In this case, when the string had a length of 12, indexes `5` and `-7` both refer to the same character (a space), and: `5 = len(monty) - 7`.

```
>>> monty[-1]
'n'
>>> monty[-7]
' '
```

We can write `for` loops to iterate over the characters in strings. This `print` statement ends with a trailing comma, which is how we tell Python not to print a newline at the end.

```
>>> sent = 'colorless green ideas sleep furiously'
>>> for char in sent:
...     print char,
...
c o l o r l e s s   g r e e n   i d e a s   s l e e p   f u r i o u s l y
```

We can count individual characters as well. We should ignore the case distinction by normalizing everything to lowercase, and filter out non-alphabetic characters:

```
>>> from nltk.corpus import gutenberg
>>> raw = gutenberg.raw('melville-moby_dick.txt')
>>> fdist = nltk.FreqDist(ch.lower() for ch in raw if ch.isalpha())
>>> fdist.keys()
['e', 't', 'a', 'o', 'n', 'i', 's', 'h', 'r', 'l', 'd', 'u', 'm', 'c', 'w',
'f', 'g', 'p', 'b', 'y', 'v', 'k', 'q', 'j', 'x', 'z']
```

This gives us the letters of the alphabet, with the most frequently occurring letters listed first (this is quite complicated and we'll explain it more carefully below). You might like to visualize the distribution using `fdist.plot()`. The relative character frequencies of a text can be used in automatically identifying the language of the text.

Accessing Substrings

A substring is any continuous section of a string that we want to pull out for further processing. We can easily access substrings using the same slice notation we used for lists. For example, the following code accesses the substring starting at index `6`, up to (but not including) index `10`:

```
>>> monty[6:10]
'Pyth'
```

Here we see the characters are `'P'`, `'y'`, `'t'`, and `'h'` which correspond to `monty[6]` ... `monty[9]` but not `monty[10]`. This is because a slice *starts* at the first index but finishes *one before* the end index.

We can also slice with negative indices — the same basic rule of starting from the start index and stopping one before the end index applies; here we stop before the space character.

```
>>> monty[0:-7]
'Monty'
```

As with list slices, if we omit the first value, the substring begins at the start of the string. If we omit the second value, the substring continues to the end of the string:

```
>>> monty[:5]
'Monty'
>>> monty[6:]
'Python'
```

We can also find the position of a substring within a string, using `find()`:

```
>>> monty.find('Python')
6
```

Your Turn: Make up a sentence and assign it to a variable, e.g. `sent = 'my sentence ...'`. Now write slice expressions to pull out individual words. (This is obviously not a convenient way to process the words of a text!)

Analyzing Strings

- character frequency plot, e.g get text in some language using `language_x = nltk.corpus.udhr.raw(x)`, then construct its frequency distribution `fdist = FreqDist(language_x)`, then view the distribution with `fdist.keys()` and `fdist.plot()`.
- functions involving strings, e.g. determining past tense
- built-ins, `find()`, `rfind()`, `index()`, `rindex()`
- revisit string tests like `endswith()` from chapter 1

The Difference between Lists and Strings

Strings and lists are both kind of **sequence**. We can pull them apart by indexing and slicing them, and we can join them together by concatenating them. However, we cannot join strings and lists:

```
>>> query = 'Who knows?'
>>> beatles = ['John', 'Paul', 'George', 'Ringo']
>>> query[2]
'o'
>>> beatles[2]
'George'
>>> query[:2]
'Wh'
>>> beatles[:2]
['John', 'Paul']
>>> query + " I don't"
"Who knows? I don't"
>>> beatles + 'Brian'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
>>> beatles + ['Brian']
['John', 'Paul', 'George', 'Ringo', 'Brian']
```

When we open a file for reading into a Python program, we get a string corresponding to the contents of the whole file. If we to use a `for` loop to process the elements of this string, all we can pick out are the individual characters — we don't get to choose the granularity. By contrast, the elements of a list can be as big or small as we like: for example, they could be paragraphs, sentence, phrases, words, characters. So lists have the advantage that we can be flexible about the elements they contain, and correspondingly flexible about any downstream processing. So one of the first things we are likely to do in a piece of NLP code is tokenize a string into a list of strings (Section 3.6). Conversely, when we want to write our results to a file, or to a terminal, we will usually format them as a string (Section 3.8).

Lists and strings do not have exactly the same functionality. Lists have the added power that you can change their elements:

```
>>> beatles[0] = "John Lennon"
>>> del beatles[-1]
>>> beatles
['John Lennon', 'Paul', 'George']
```

On the other hand if we try to do that with a *string* — changing the 0th character in `query` to `'F'` — we get:

```
>>> query[0] = 'F'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

This is because strings are **immutable** — you can't change a string once you have created it. However, lists are **mutable**, and their contents can be modified at any time. As a result, lists support operations that modify the original value rather than producing a new value.

3.3 Regular Expressions for Detecting Word Patterns

Many linguistic processing tasks involve pattern matching. For example, we can find words ending with *ed* using `endswith('ed')`. We saw a variety of such "word tests" in Figure 1.4. Regular expressions give us a more powerful and flexible method for describing the character patterns we are interested in.

There are many other published introductions to regular expressions, organized around the syntax of regular expressions and applied to searching text files. Instead of doing this again, we focus on the use of regular expressions at different stages of linguistic processing. As usual, we'll adopt a problem-based approach and present new features only as they are needed to solve practical problems. In our discussion we will mark regular expressions using chevrons like this: «`patt`».

To use regular expressions in Python we need to import the `re` library using: `import re`. We also need a list of words to search; we'll use the words corpus again (Section 2.4). We will preprocess it to remove any proper names.

```
>>> import re
>>> wordlist = [w for w in nltk.corpus.words.words() if w.islower()]
```

Using Basic Meta-Characters

Let's find words ending with *ed* using the regular expression «`ed$`». We will use the `re.search(p, s)` function to check whether the pattern `p` can be found somewhere inside the string `s`. We need to specify the characters of interest, and use the dollar sign which has a special behavior in the context of regular expressions in that it matches the end of the word:

```
>>> [w for w in wordlist if re.search('ed$', w)]
['abaissed', 'abandoned', 'abased', 'abashed', 'abatised', 'abed', 'aborted', ...]
```

The "`.`" **wildcard** symbol matches any single character. Suppose we have room in a crossword puzzle for an 8-letter word with *j* as its third letter and *t* as its sixth letter. In place of each blank cell we use a period:

```
>>> [w for w in wordlist if re.search('^..j..t..$', w)]
['abjectly', 'adjuster', 'dejected', 'dejectly', 'injector', 'majestic', ...]
```

Finally, the "`?`" symbol specifies that the previous character is optional. Thus «`e-?mail`» will match both *email* and *e-mail*. We could count the total number of occurrences of this word (in either spelling) using `len(w for w in text if re.search('e-?mail', w))`.

Your Turn: The caret symbol ^ matches the start of the word, just like the $ matches the end of the word. What results to we get with the above example if we leave out both of these, and search for «..j..t..»?

Ranges and Closures

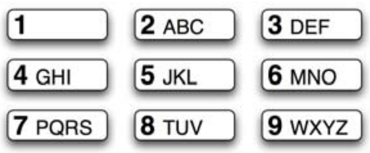


Figure 3.2: T9: Text on 9 Keys

The **T9** system is used for entering text on mobile phones. Two or more words that are entered using the same sequence of keystrokes are known as **textonyms**. For example, both *hole* and *golf* are entered using 4653. What other words could be produced with the same sequence? Here we use the regular expression «^[ghi][mno][jlk][def]$»:

```
>>> [w for w in wordlist if re.search('^[ghi][mno][jlk][def]$', w)]
['gold', 'golf', 'hold', 'hole']
```

The first part of the expression, «^[ghi]», matches the start of a word followed by *g*, *h*, or *i*. The next part of the expression, «[mno]», constrains the second character to be *m*, *n*, or *o*. The third and fourth characters are also constrained. Only six words satisfy all these constraints. Note that the order of characters inside the square brackets is not significant, so we could have written «^[hig][nom][ljk][fed]$» and matched the same words.

Your Turn: Look for some "finger-twisters", by searching for words that only use part of the number-pad. For example «^[g-o]+$» will match words that only use keys 4, 5, 6 in the center row, and «^[a-fj-o]+$» will match words that use keys 2, 3, 5, 6 in the top-right corner. What do "-" and "+" mean?

Let's explore the "+" symbol a bit further. Notice that it can be applied to individual letters, or to bracketed sets of letters:

```
>>> chat_words = sorted(set(w for w in nltk.corpus.nps_chat.words()))
>>> [w for w in chat_words if re.search('^m+i+n+e+$', w)]
['miiiiiiiiiiiiinnnnnnnnnnneeeeeeeeee', 'miiiiiinnnnnnnnnneeeeeeee', 'mine',
'mmmmmmmmiiiiiiiiinnnnnnnnneeeeeeee']
>>> [w for w in chat_words if re.search('^[ha]+$', w)]
['a', 'aaaaaaaaaaaaaaaaa', 'aaahhhh', 'ah', 'ahah', 'ahahah', 'ahh',
'ahhahahaha', 'ahhh', 'ahhhh', 'ahhhhhh', 'ahhhhhhhhhhhhhh', 'h', 'ha', 'haaa',
'hah', 'haha', 'hahaaa', 'hahah', 'hahaha', 'hahahaa', 'hahahah', 'hahahaha', ...]
```

It should be clear that "+" simply means "one or more instances of the preceding item", which could be an individual character like m, a set like [fed] or a range like [d-f]. Now let's replace "+" with "*" which means "zero or more instances of the preceding item". The regular expression «^m*i*n*e*$» will match everything that we found using «^m+i+n+e+$», but also words where some of the letters don't appear at all, e.g. *me*, *min*, and *mmmmm*. Note that the "+" and "*" symbols are sometimes referred to as **Kleene closures**, or simply **closures**.

The "^" operator has another function when it appears inside square brackets. For example «`[^aeiouAEIOU]`» matches any character other than a vowel. We can search the Chat corpus for words that are made up entirely of non-vowel characters using «`^[^aeiouAEIOU]+$`» to find items like these: `:):):)`, `grrr`, `cyb3r` and `zzzzzzzz`. Notice this includes non-alphabetic characters.

Your Turn: Study the following examples and work out what the `\`, `{}` and `|` notations mean:

```
>>> wsj = sorted(set(nltk.corpus.treebank.words()))
>>> [w for w in wsj if re.search('^[0-9]+\.[0-9]+$', w)]
['0.0085', '0.05', '0.1', '0.16', '0.2', '0.25', '0.28', '0.3', '0.4', '0.5',
'0.50', '0.54', '0.56', '0.60', '0.7', '0.82', '0.84', '0.9', '0.95', '0.99',
'1.01', '1.1', '1.125', '1.14', '1.1650', '1.17', '1.18', '1.19', '1.2', ...]
>>> [w for w in wsj if re.search('^[A-Z]+\$$', w)]
['C$', 'US$']
>>> [w for w in wsj if re.search('^[0-9]{4}$', w)]
['1614', '1637', '1787', '1901', '1903', '1917', '1925', '1929', '1933', ...]
>>> [w for w in wsj if re.search('^[0-9]+-[a-z]{3,5}$', w)]
['10-day', '10-lap', '10-year', '100-share', '12-point', '12-year', ...]
>>> [w for w in wsj if re.search('^[a-z]{5,}-[a-z]{2,3}-[a-z]{,6}$', w)]
['black-and-white', 'bread-and-butter', 'father-in-law', 'machine-gun-toting',
'savings-and-loan']
>>> [w for w in wsj if re.search('(ed|ing)$', w)]
['62%-owned', 'Absorbed', 'According', 'Adopting', 'Advanced', 'Advancing', ...]
```

You probably worked out that a backslash means that the following character is deprived of its special powers and must literally match a specific character in the word. Thus, while `'.'` is special, `'\.'` only matches a period. The brace characters are used to specify the number of repeats of the previous item.

The meta-characters we have seen are summarized in Table 3.2.

Operator	**Behavior**
`.`	Wildcard, matches any character
`^abc`	Matches some pattern *abc* at the start of a string
`abc$`	Matches some pattern *abc* at the end of a string
`[abc]`	Matches a set of characters
`[A-Z0-9]`	Matches a range of characters
`ed|ing|s`	Matches one of the specified strings (disjunction)
`*`	Zero or more of previous item, e.g. `a*`, `[a-z]*` (also known as *Kleene Closure*)
`+`	One or more of previous item, e.g. `a+`, `[a-z]+`
`?`	Zero or one of the previous item (i.e. optional), e.g. `a?`, `[a-z]?`
`{n}`	Exactly *n* repeats where n is a non-negative integer
`{m,n}`	At least *m* and no more than *n* repeats (*m*, *n* optional)
`(ab|c)+`	Parentheses that indicate the scope of the operators

Table 3.2: Basic Regular Expression Meta-Characters, Including Wildcards, Ranges and Closures

3.4 Useful Applications of Regular Expressions

The above examples all involved searching for words *w* that match some regular expression *regexp* using `re.search(regexp, w)`. Apart from checking if a regular expression matches a word, we can use regular expressions to extract material from words, or to modify words in specific ways.

Extracting Word Pieces

The *re.findall()*‘ (“find all”) method finds all (non-overlapping) matches of the given regular expression. Let’s find all the vowels in a word, then count them:

```
>>> word = 'supercalifragulisticexpialidocious'
>>> re.findall('[aeiou]', word)
['u', 'e', 'a', 'i', 'a', 'u', 'i', 'i', 'e', 'i', 'a', 'i', 'o', 'i', 'o', 'u']
>>> len(re.findall('[aeiou]', word))
16
```

Let’s look for all sequences of two or more vowels in some text, and determine their relative frequency:

```
>>> wsj = sorted(set(nltk.corpus.treebank.words()))
>>> fd = nltk.FreqDist(vs for word in wsj
...                       for vs in re.findall('[aeiou]{2,}', word))
>>> fd.items()
[('io', 549), ('ea', 476), ('ie', 331), ('ou', 329), ('ai', 261), ('ia', 253),
('ee', 217), ('oo', 174), ('ua', 109), ('au', 106), ('ue', 105), ('ui', 95),
('ei', 86), ('oi', 65), ('oa', 59), ('eo', 39), ('iou', 27), ('eu', 18), ...]
```

Your Turn: In the W3C Date Time Format, dates are represented like this: 2009-12-31. Replace the `?` in the following Python code with a regular expression, in order to convert the string `'2009-12-31'` to a list of integers `[2009, 12, 31]`.

```
[int(n) for n in re.findall(?, '2009-12-31')]
```

Doing More with Word Pieces

Once we can use `re.findall()` to extract material from words, there’s interesting things to do with the pieces, like glue them back together or plot them.

It is sometimes noted that English text is highly redundant, and it is still easy to read when word-internal vowels are left out. For example, *declaration* becomes *dclrtn*, and *inalienable* becomes *inlnble*, retaining any initial or final vowel sequences. This regular expression matches initial vowel sequences, final vowel sequences, and all consonants; everything else is ignored. We use `re.findall()` to extract all the matching pieces, and `''.join()` to join them together (see Section 3.8 for more about the join operation).

```
>>> regexp = '^[AEIOUaeiou]+|[AEIOUaeiou]+$|[^AEIOUaeiou]'
>>> def compress(word):
...     pieces = re.findall(regexp, word)
...     return ''.join(pieces)
...
>>> english_udhr = nltk.corpus.udhr.words('English-Latin1')
>>> print nltk.tokenwrap(compress(w) for w in english_udhr[:75])
```

```
Unvrsl Dclrtn of Hmn Rghts Prmble Whrs rcgntn of the inhrnt dgnty and
of the eql and inlnble rghts of all mmbrs of the hmn fmly is the fndtn
of frdm , jstce and pce in the wrld , Whrs dsrgrd and cntmpt fr hmn
rghts hve rsltd in brbrs acts whch hve outrgd the cnscnce of mnknd ,
and the advnt of a wrld in whch hmn bngs shll enjy frdm of spch and
```

Next, let's combine regular expressions with conditional frequency distributions. Here we will extract all consonant-vowel sequences from the words of Rotokas, such as *ka* and *si*. Since each of these is a pair, it can be used to initialize a conditional frequency distribution. We then tabulate the frequency of each pair:

```
>>> rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words for cv in re.findall('[ptksvr][aeiou]', w)]
>>> cfd = nltk.ConditionalFreqDist(cvs)
>>> cfd.tabulate()
     a    e    i    o    u
k  418  148   94  420  173
p   83   31  105   34   51
r  187   63   84   89   79
s    0    0  100    2    1
t   47    8    0  148   37
v   93   27  105   48   49
```

Examining the rows for *s* and *t*, we see they are in partial "complementary distribution", which is evidence that they are not distinct phonemes in the language. Thus, we could conceivably drop *s* from the Rotokas alphabet and simply have a pronunciation rule that the letter *t* is pronounced *s* when followed by *i*. (Note that single entry having *su*, namely *kasuari* 'cassowary' is a loanword).

If we want to be able to inspect the words behind the numbers in the above table, it would be helpful to have an index, allowing us to quickly find the list of words that contains a given consonant-vowel pair, e.g. `cv_index['su']` should give us all words containing *su*. Here's how we can do this:

```
>>> cv_word_pairs = [(cv, w) for w in rotokas_words
...                          for cv in re.findall('[ptksvr][aeiou]', w)]
>>> cv_index = nltk.Index(cv_word_pairs)
>>> cv_index['su']
['kasuari']
>>> cv_index['po']
['kaapo', 'kaapopato', 'kaipori', 'kaiporipie', 'kaiporivira', 'kapo', 'kapoa',
'kapokao', 'kapokapo', 'kapokapo', 'kapokapoa', 'kapokapoa', 'kapokapora', ...]
```

This program processes each word `w` in turn, and for each one, finds every substring that matches the regular expression «`[ptksvr][aeiou]`». In the case of the word *kasuari*, it finds *ka*, *su* and *ri*. Therefore, the `cv_word_pairs` list will contain `('ka', 'kasuari')`, `('su', 'kasuari')` and `('ri', 'kasuari')`. One further step, using `nltk.Index()`, converts this into a useful index.

Finding Word Stems

When we use a web search engine, we usually don't mind (or even notice) if the words in the document differ from our search terms in having different endings. A query for *laptops* finds documents containing *laptop* and vice versa. Indeed, *laptop* and *laptops* are just two forms of the *same* word. For some language processing tasks we want to ignore word endings, and just deal with word stems.

There are various ways we can pull out the stem of a word. Here's a simple-minded approach which just strips off anything that looks like a suffix:

```
>>> def stem(word):
...     for suffix in ['ing', 'ly', 'ed', 'ious', 'ies', 'ive', 'es', 's', 'ment']:
...         if word.endswith(suffix):
...             return word[:-len(suffix)]
...     return word
```

Although we will ultimately use NLTK's built-in stemmers, its interesting to see how we can use regular expressions for this task. Our first step is to build up a disjunction of all the suffixes. We need to enclose it in parentheses in order to limit the scope of the disjunction.

```
>>> re.findall('^.*(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
['ing']
```

Here, `re.findall()` just gave us the suffix even though the regular expression matched the entire word. This is because the parentheses have a second function, to select substrings to be extracted. If we want to use the parentheses for scoping the disjunction but not for selecting output, we have to add `?:` (just one of many arcane subtleties of regular expressions). Here's the revised version.

```
>>> re.findall('^.*(?:ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
['processing']
```

However, we'd actually like to split the word into stem and suffix. Instead, we should just parenthesize both parts of the regular expression:

```
>>> re.findall('^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
```

This looks promising, but still has a problem. Let's look at a different word, *processes*

```
>>> re.findall('^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
```

The regular expression incorrectly found an *-s* suffix instead of an *-es* suffix. This demonstrates another subtlety: the star operator is "greedy" and the `.*` part of the expression tries to consume as much of the input as possible. If we use the "non-greedy" version of the star operator, written `*?`, we get what we want:

```
>>> re.findall('^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('process', 'es')]
```

This works even when we allow empty suffix, by making the content of the second parentheses optional:

```
>>> re.findall('^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$', 'language')
[('language', '')]
```

This approach still has many problems (can you spot them?) but we will move on to define a stemming function and apply it to a whole text:

```
>>> def stem(word):
...     regexp = '^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$'
...     stem, suffix = re.findall(regexp, word)[0]
...     return stem
```

```
...
>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords
... is no basis for a system of government.  Supreme executive power derives from
... a mandate from the masses, not from some farcical aquatic ceremony."""
>>> tokens = nltk.wordpunct_tokenize(raw)
>>> [stem(t) for t in tokens]
['DENNIS', ':', 'Listen', ',', 'strange', 'women', 'ly', 'in', 'pond',
'distribut', 'sword', 'i', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',
'.', 'Supreme', 'execut', 'power', 'deriv', 'from', 'a', 'mandate', 'from',
'the', 'mass', ',', 'not', 'from', 'some', 'farcical', 'aquatic', 'ceremony', '.']
```

Notice that our regular expression removed the *s* from *ponds* but also from *is* and *basis*. It produced some non-words like *distribut* and *deriv*, but these are acceptable stems.

Searching Tokenized Text

You can use a special kind of regular expression for searching across multiple words in a text (where a text is a list of tokens).

```
>>> from nltk.corpus import gutenberg, nps_chat
>>> moby = nltk.Text(gutenberg.words('melville-moby_dick.txt'))
>>> moby.findall("<a>(<.*>)<man>")
monied; nervous; dangerous; white; white; white; pious; queer; good;
mature; white; Cape; great; wise; wise; butterless; white; fiendish;
pale; furious; better; certain; complete; dismasted; younger; brave;
brave; brave; brave
>>> chat = nltk.Text(nps_chat.words())
>>> chat.search("<.*><.*><bro>")
you rule bro; telling you bro; u twizted bro
>>> chat.search("<l.*>{3,}")
lol lol lol; lmao lol lol; lol lol lol; la la la la la; la la la; la
la la; lovely lol lol love; lol lol lol.; la la la; la la la
```

Your Turn: Consolidate your understanding of regular expression patterns and substitutions using `nltk.re_show(p, s)` which annotates the string `s` to show every place where pattern `p` was matched, and `nltk.draw.finding_nemo()` which provides a graphical interface for exploring regular expressions.

It is easy to build search patterns when the linguistic phenomenon we're studying is tied to particular words. In some cases, a little creativity will go a long way. For instance, searching a large text corpus for expressions of the form *x and other ys* allows us to discover examples of instances and their corresponding types:

```
>>> hobbies_learned = nltk.Text(brown.words(categories=['hobbies', 'learned']))
>>> hobbies_learned.findall("<\w*> <and> <other> <\w*s>")
speed and other activities; water and other liquids; tomb and other
landmarks; Statues and other monuments; pearls and other jewels;
charts and other items; roads and other features; figures and other
objects; military and other areas; demands and other factors;
abstracts and other compilations; iron and other metals
```

With enough text, this approach would give us a useful store of information about the taxonomy of objects, without the need for any manual labor. However, our search results will usually contain false positives, i.e. cases that we would want to exclude. For example, the result: *demands and other factors* suggests that *demand* is an instance of the type *factor*, but this sentence is actually about wage demands.

Nevertheless, we could construct our own corpus of instances and types by manually correcting the output of such searches.

This combination of automatic and manual processing is the most common way for new corpora to be constructed. We will return to this in Chapter 11.

Searching corpora also suffers from the problem of false negatives, i.e. omitting cases that we would want to include. It is risky to conclude that some linguistic phenomenon doesn't exist in a corpus just because we couldn't find any instances of a search pattern. Perhaps we just didn't think carefully enough about suitable patterns.

Your Turn: Look for instances of the pattern *as x as y* to discover information about entities and their properties.

3.5 Normalizing Text

In earlier program examples we have often converted text to lowercase before doing anything with its words, e.g. `set(w.lower() for w in text)`. By using `lower()`, we have **normalized** the text to lowercase so that the distinction between *The* and *the* is ignored. Often we want to go further than this, and strip off any affixes, a task known as stemming. A further step is to make sure that the resulting form is a known word in a dictionary, a task known as lemmatization. We discuss each of these in turn.

Stemmers

NLTK includes several off-the-shelf stemmers, and if you ever need a stemmer you should use one of these in preference to crafting your own using regular expressions, since these handle a wide range of irregular cases. The Porter Stemmer strips affixes and knows about some special cases, e.g. that *lie* not *ly* is the stem of *lying*.

```
>>> porter = nltk.PorterStemmer()
>>> lancaster = nltk.LancasterStemmer()
>>> [porter.stem(t) for t in tokens]
['DENNI', ':', 'Listen', ',', 'strang', 'women', 'lie', 'in', 'pond',
'distribut', 'sword', 'is', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',
'.', 'Suprem', 'execut', 'power', 'deriv', 'from', 'a', 'mandat', 'from',
'the', 'mass', ',', 'not', 'from', 'some', 'farcic', 'aquat', 'ceremoni', '.']
>>> [lancaster.stem(t) for t in tokens]
['den', ':', 'list', ',', 'strange', 'wom', 'lying', 'in', 'pond', 'distribut',
'sword', 'is', 'no', 'bas', 'for', 'a', 'system', 'of', 'govern', '.', 'suprem',
'execut', 'pow', 'der', 'from', 'a', 'mand', 'from', 'the', 'mass', ',', 'not',
'from', 'som', 'farc', 'aqu', 'ceremony', '.']
```

Stemming is not a well-defined process, and we typically pick the stemmer that best suits the application we have in mind. The Porter Stemmer is a good choice if you are indexing some texts and want to support search using alternative forms of words (illustrated in Figure 3.3, which uses *object oriented* programming techniques that will be covered in Chapter REF, and string formatting techniques to be covered in section 3.8).

```
class IndexedText(object):

    def __init__(self, stemmer, text):
        self._text = text
        self._stemmer = stemmer
        self._index = nltk.Index((self._stem(word), i)
                                 for (i, word) in enumerate(text))

    def concordance(self, word, width=40):
        key = self._stem(word)
        wc = width/4                # words of context
        for i in self._index[key]:
            lcontext = ' '.join(self._text[i-wc:i])
            rcontext = ' '.join(self._text[i:i+wc])
            ldisplay = '%*s'  % (width, lcontext[-width:])
            rdisplay = '%-*s' % (width, rcontext[:width])
            print ldisplay, rdisplay

    def _stem(self, word):
        return self._stemmer.stem(word).lower()

>>> porter = nltk.PorterStemmer()
>>> grail = nltk.corpus.webtext.words('grail.txt')
>>> text = IndexedText(porter, grail)
>>> text.concordance('lie')
r king ! DENNIS : Listen , strange women lying in ponds distributing swords is no
 beat a very brave retreat . ROBIN : All lies ! MINSTREL : [ singing ] Bravest of
       Nay . Nay . Come . Come . You may lie here . Oh , but you are wounded !
doctors immediately ! No , no , please ! Lie down . [ clap clap ] PIGLET : Well
ere is much danger , for beyond the cave lies the Gorge of Eternal Peril , which
   you . Oh ... TIM : To the north there lies a cave -- the cave of Caerbannog --
h it and lived ! Bones of full fifty men lie strewn about its lair . So , brave k
not stop our fight ' til each one of you lies dead , and the Holy Grail returns t
```

Figure 3.3: Indexing a Text Using a Stemmer

Lemmatization

The WordNet lemmatizer only removes affixes if the resulting word is in its dictionary (and this additional checking process makes it slower). It doesn't handle *lying*, but it converts *women* to *woman*.

```
>>> wnl = nltk.WordNetLemmatizer()
>>> [wnl.lemmatize(t) for t in tokens]
['DENNIS', ':', 'Listen', ',', 'strange', 'woman', 'lying', 'in', 'pond',
'distributing', 'sword', 'is', 'no', 'basis', 'for', 'a', 'system', 'of',
'government', '.', 'Supreme', 'executive', 'power', 'derives', 'from', 'a',
'mandate', 'from', 'the', 'mass', ',', 'not', 'from', 'some', 'farcical',
'aquatic', 'ceremony', '.']
```

The WordNet lemmatizer is a good choice if you want to compile the vocabulary of some texts and want a list of valid lexical items.

Non-Standard Words

[Discuss the practice of mapping words such as numbers, abbreviations, dates to a special vocabulary, based on Sproat et al 2001; new NLTK support planned...]

3.6 Regular Expressions for Tokenizing Text

Tokenization is the task of cutting a string into identifiable linguistic units that constitute a piece of language data. Although it is a fundamental task, we have been able to delay it til now because many corpora are already tokenized, and because NLTK includes some tokenizers. Now that you are familiar with regular expressions, you can learn how to use them to tokenize text, and to have much more control over the process.

Simple Approaches to Tokenization

The very simplest method for tokenizing text is to split on whitespace. Consider the following text from *Alice's Adventures in Wonderland*:

```
>>> raw = """'When I'M a Duchess,' she said to herself, (not in a very hopeful tone
... though), 'I won't have any pepper in my kitchen AT ALL. Soup does very
... well without--Maybe it's always pepper that makes people hot-tempered,'..."""
```

We could split this raw text on whitespace using `raw.split()`. To do the same using a regular expression, we need to match any number of spaces, tabs, or newlines.

```
>>> re.split(r'[ \t\n]+', raw)
["'When", "I'M", 'a', "Duchess,'", 'she', 'said', 'to', 'herself,', '(not', 'in', 'a',
'very', 'hopeful', 'tone', 'though),', "'I", "won't", 'have', 'any', 'pepper', 'in',
'my', 'kitchen', 'AT', 'ALL.', 'Soup', 'does', 'very', 'well', 'without--Maybe',
"it's", 'always', 'pepper', 'that', 'makes', 'people', "hot-tempered,'..."]
```

The regular expression «`[ \t\n]+`» matches one or more space, tab (`\t`) or newline (`\n`). Other whitespace characters, such as carriage-return and form-feed should really be included too. Instead, When using regular expressions that contain the backslash character, you should prefix the string with the letter `r` (meaning "raw"), which instructs the Python interpreter to treat them as literal backslashes.

we will can use a built-in `re` abbreviation, `\s`, which means any whitespace character. The above statement can be rewritten as `re.split(r'\s+', raw)`.

Splitting on whitespace gives us tokens like `'(not'` and `'herself,'`. An alternative is to use the fact that Python provides us with a character class `\w` for word characters [define] and also the complement of this class `\W`. So, we can split on anything *other* than a word character:

```
>>> re.split(r'\W+', raw)
['', 'When', 'I', 'M', 'a', 'Duchess', 'she', 'said', 'to', 'herself', 'not', 'in',
'a', 'very', 'hopeful', 'tone', 'though', 'I', 'won', 't', 'have', 'any', 'pepper', 'in',
'my', 'kitchen', 'AT', 'ALL', 'Soup', 'does', 'very', 'well', 'without', 'Maybe',
'it', 's', 'always', 'pepper', 'that', 'makes', 'people', 'hot', 'tempered', '']
```

Observe that this gives us empty strings [explain why]. We get the same result using `re.findall(r'\w+', raw)`, using a pattern that matches the words instead of the spaces.

```
>>> re.findall(r'\w+|\S\w*', raw)
["'When", 'I', "'M", 'a', 'Duchess', ',', "'", 'she', 'said', 'to', 'herself', ',',
'(not', 'in', 'a', 'very', 'hopeful', 'tone', 'though', ')', ',', "'I", 'won', "'t",
'have', 'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL', '.', 'Soup', 'does',
'very', 'well', 'without', '-', '-Maybe', 'it', "'s", 'always', 'pepper', 'that',
'makes', 'people', 'hot', '-tempered', ',', "'", '.', '.', '.']
```

The regular expression «`\w+|\S\w*`» will first try to match any sequence of word characters. If no match is found, it will try to match any *non*-whitespace character (`\S` is the complement of `\s`) followed by further word characters. This means that punctuation is grouped with any following letters (e.g. *'s*) but that sequences of two or more punctuation characters are separated. Let's generalize the `\w+` in the above expression to permit word-internal hyphens and apostrophes: «`\w+([-']\w+)*`». This expression means `\w+` followed by zero or more instances of `[-']\w+`; it would match *hot-tempered* and *it's*. (We need to include `?:` in this expression for reasons discussed earlier.) We'll also add a pattern to match quote characters so these are kept separate from the text they enclose.

```
>>> print re.findall(r"\w+(?:[-']\w+)*|'|[-.(]+|\S\w*", raw)
["'", 'When', "I'M", 'a', 'Duchess', ',', "'", 'she', 'said', 'to', 'herself', ',',
'(', 'not', 'in', 'a', 'very', 'hopeful', 'tone', 'though', ')', ',', "'", 'I', "won't",
'have', 'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL', '.', 'Soup', 'does',
'very', 'well', 'without', '--', 'Maybe', "it's", 'always', 'pepper', 'that',
'makes', 'people', 'hot-tempered', ',', "'", '...']
```

The above expression also included «`[-.)]+`» which causes the double hyphen, ellipsis, and open bracket to be tokenized separately.

Table 3.3 lists the regular expression character class symbols we have seen in this section.

Symbol	Function
`\b`	Word boundary (zero width)
`\d`	Any decimal digit (equivalent to `[0-9]`)
`\D`	Any non-digit character (equivalent to `[^0-9]`)

Symbol	Function
`\s`	Any whitespace character (equivalent to `[ \t\n\r\f\v]`
`\S`	Any non-whitespace character (equivalent to `[^ \t\n\r\f\v]`)
`\w`	Any alphanumeric character (equivalent to `[a-zA-Z0-9_]`)
`\W`	Any non-alphanumeric character (equivalent to `[^a-zA-Z0-9_]`)
`\t`	The tab character
`\n`	The newline character

Table 3.3: Regular Expression Symbols

NLTK's Regular Expression Tokenizer

The function `nltk.regexp_tokenize()` is like `re.findall`, except it is more efficient and it avoids the need for special treatment of parentheses. For readability we break up the regular expression over several lines and add a comment about each line. The special `(?x)` "verbose flag" tells Python to strip out the embedded whitespace and comments.

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

The `regexp_tokenize()` function has an optional `gaps` parameter. When set to `True`, the regular expression is applied to the gaps between tokens (cf `re.split()`).

We can evaluate a tokenizer by comparing the resulting tokens with a wordlist, and reporting any tokens that don't appear in the wordlist, using `set(tokens).difference(wordlist)`. You'll probably want to lowercase all the tokens first.

Dealing with Contractions

A final issue for tokenization is the presence of contractions, such as *didn't*. If we are analyzing the meaning of a sentence, it would probably be more useful to normalize this form to two separate forms: *did* and *n't* (or *not*). [MORE]

3.7 Sentence Segmentation

[Explain how sentence segmentation followed by word tokenization can give different results to word tokenization on its own.]

Manipulating texts at the level of individual words often presupposes the ability to divide a text into individual sentences. As we have seen, some corpora already provide access at the sentence level. In the following example, we compute the average number of words per sentence in the Brown Corpus:

```
>>> len(nltk.corpus.brown.words()) / len(nltk.corpus.brown.sents())
20
```

In other cases, the text is only available as a stream of characters. Before tokenizing the text into words, we need to segment it into sentences. NLTK facilitates this by including the Punkt sentence segmenter [Tibor & Jan, 2006], along with supporting data for English. Here is an example of its use in segmenting the text of a novel:

```
>>> sent_tokenizer=nltk.data.load('tokenizers/punkt/english.pickle')
>>> text = nltk.corpus.gutenberg.raw('chesterton-thursday.txt')
>>> sents = sent_tokenizer.tokenize(text)
>>> pprint.pprint(sents[171:181])
['"Nonsense!',
 '" said Gregory, who was very rational when anyone else\nattempted paradox.',
 '"Why do all the clerks and navvies in the\nrailway trains look so sad and tired, so very sad and t
 'I will\ntell you.',
 'It is because they know that the train is going right.',
 'It\nis because they know that whatever place they have taken a ticket\nfor that place they will re
 'It is because after they have\npassed Sloane Square they know that the next station must be\nVicto
 'Oh, their wild rapture!',
 'oh,\ntheir eyes like stars and their souls again in Eden, if the next\nstation were unaccountably
 '"\n\n"It is you who are unpoetical," replied the poet Syme.']
```

Notice that this example is really a single sentence, reporting the speech of Mr Lucian Gregory. However, the quoted speech contains several sentences, and these have been split into individual strings. This is reasonable behavior for most applications.

3.8 Formatting: From Lists to Strings

Often we write a program to report a single data item, such as a particular element in a corpus that meets some complicated criterion, or a single summary statistic such as a word-count or the performance of a tagger. More often, we write a program to produce a structured result, such as a tabulation of numbers or linguistic forms, or a reformatting of the original data. When the results to be presented are linguistic, textual output is usually the most natural choice. However, when the results are numerical, it may be preferable to produce graphical output. In this section you will learn about a variety of ways to present program output.

Converting Between Strings and Lists (notes)

We specify the string to be used as the "glue", followed by a period, followed by the `join()` function.

```
>>> silly = ['We', 'called', 'him', 'Tortoise', 'because', 'he', 'taught', 'us', '.']
>>> ' '.join(silly)
'We called him Tortoise because he taught us .'
>>> ';'.join(silly)
'We;called;him;Tortoise;because;he;taught;us;.'
```

So `' '.join(silly)` means: take all the items in `silly` and concatenate them as one big string, using `' '` as a spacer between the items. (Many people find the notation for `join()` rather unintuitive.)

Notice that `join()` only works on a list of strings (what we have been calling a text).

Formatting Output

The output of a program is usually structured to make the information easily digestible by a reader. Instead of running some code and then manually inspecting the contents of a variable, we would like the code to tabulate some output. There are many ways we might want to format the output of a program. For instance, we might want to place the length value in parentheses *after* the word, and print all the output on a single line:

```
>>> saying = ['After', 'all', 'is', 'said', 'and', 'done', ',',
...           'more', 'is', 'said', 'than', 'done', '.']
>>> for word in saying:
...     print word, '(' + str(len(word)) + '),',
After (5), all (3), is (2), said (4), and (3), done (4), , (1), more (4), is (2), said (4), than (4), done (4)
```

However, this approach has some problems. First, the `print` statement intermingles variables and punctuation, making it a little difficult to read. Second, the output has spaces around every item that was printed. Third, we have to convert the length of the word to a string so that we can surround it with parentheses. A cleaner way to produce structured output uses Python's **string formatting expressions**. Before diving into clever formatting tricks, however, let's look at a really simple example. We are going to use a special symbol, `%s`, as a placeholder in strings. Once we have a string containing this placeholder, we follow it with a single `%` and then a value `v`. Python then returns a new string where `v` has been slotted in to replace `%s`:

```
>>> "I want a %s right now" % "coffee"
'I want a coffee right now'
```

In fact, we can have a number of placeholders, but following the `%` operator we need to specify a tuple with exactly the same number of values.

```
>>> "%s wants a %s %s" % ("Lee", "sandwich", "for lunch")
'Lee wants a sandwich for lunch'
>>>
```

We can also provide the values for the placeholders indirectly. Here's an example using a `for` loop:

```
>>> menu = ['sandwich', 'spam fritter', 'pancake']
>>> for snack in menu:
...     "Lee wants a %s right now" % snack
...
'Lee wants a sandwich right now'
'Lee wants a spam fritter right now'
'Lee wants a pancake right now'
>>>
```

We oversimplified things when we said that placeholders were of the form `%s`; in fact, this is a complex object, called a **conversion specifier**. This has to start with the `%` character, and ends with conversion character such as `s` or `d`. The `%s` specifier tells Python that the corresponding variable is a string (or should be converted into a string), while the `%d` specifier indicates that the corresponding variable should be converted into a decimal representation. The string containing conversion specifiers is called a **format string**.

Picking up on the `print` example that we opened this section with, here's how we can use two different kinds of conversion specifier:

```
>>> for word in saying:
...     print "%s (%d)," % (word, len(word)),
After (5), all (3), is (2), said (4), and (3), done (4), , (1), more (4), is (2), said (4), than (4)
```

To summarize, string formatting is accomplished with a three-part object having the syntax: *`format`* `%` *`values`*. The *`format`* section is a string containing format specifiers such as `%s` and `%d` that Python will replace with the supplied values. The *`values`* section of a formatting string is a parenthesized list containing exactly as many items as there are format specifiers in the *`format`* section. In the case that there is just one item, the parentheses can be left out.

In the above example, we used a trailing comma to suppress the printing of a newline. Suppose, on the other hand, that we want to introduce some additional newlines in our output. We can accomplish this by inserting the "special" character `\n` into the `print` string:

```
>>> for i, word in enumerate(saying[:6]):
...    print "Word = %s\nIndex = %s" % (word, i)
...
Word = After
Index = 0
Word = all
Index = 1
Word = is
Index = 2
Word = said
Index = 3
Word = and
Index = 4
Word = done
Index = 5
```

Strings and Formats

We have seen that there are two ways to display the contents of an object:

```
>>> word = 'cat'
>>> sentence = """hello
... world"""
>>> print word
cat
>>> print sentence
hello
world
>>> word
'cat'
>>> sentence
'hello\nworld'
```

The `print` command yields Python's attempt to produce the most human-readable form of an object. The second method — naming the variable at a prompt — shows us a string that can be used to recreate this object. It is important to keep in mind that both of these are just strings, displayed for the benefit of you, the user. They do not give us any clue as to the actual internal representation of the object.

There are many other useful ways to display an object as a string of characters. This may be for the benefit of a human reader, or because we want to **export** our data to a particular file format for use in an external program.

Formatted output typically contains a combination of variables and pre-specified strings, e.g. given a dictionary `wordcount` consisting of words and their frequencies we could do:

```
>>> wordcount = {'cat':3, 'dog':4, 'snake':1}
>>> for word in sorted(wordcount):
...     print word, '->', wordcount[word], ';',
cat -> 3 ; dog -> 4 ; snake -> 1 ;
```

Apart from the problem of unwanted whitespace, print statements that contain alternating variables and constants can be difficult to read and maintain. A better solution is to use formatting strings:

```
>>> for word in sorted(wordcount):
...    print '%s->%d;' % (word, wordcount[word]),
cat->3; dog->4; snake->1;
```

Lining Things Up

So far our formatting strings have contained specifications of fixed width, such as `%6s`, a string that is padded to width 6 and right-justified. We can include a minus sign to make it left-justified. In case we don't know in advance how wide a displayed value should be, the width value can be replaced with a star in the formatting string, then specified using a variable:

```
>>> '%6s' % 'dog'
'   dog'
>>> '%-6s' % 'dog'
'dog   '
>>> width = 6
>>> '%-*s' % (width, 'dog')
'dog   '
```

Other control characters are used for decimal integers and floating point numbers. Since the percent character `%` has a special interpretation in formatting strings, we have to precede it with another `%` to get it in the output:

```
>>> "accuracy for %d words: %2.4f%%" % (9375, 100.0 * 3205/9375)
'accuracy for 9375 words: 34.1867%'
```

An important use of formatting strings is for tabulating data. Recall that in section 2.1 we saw data being tabulated from a conditional frequency distribution. Let's perform the tabulation ourselves, exercising full control of headings and column widths. Note the clear separation between the language processing work, and the tabulation of results.

Recall from the listing in Figure 3.3 that we used a formatting string `"%*s"`. This allows us to specify the width of a field using a variable.

```
>>> '%*s' % (15, "Monty Python")
'   Monty Python'
```

We could use this to automatically customise the width of a column to be the smallest value required to fit all the words, using `width = min(len(w) for w in words)`. Remember that the comma at the end of print statements adds an extra space, and this is sufficient to prevent the column headings from running into each other.

```
def tabulate(cfdist, words, categories):
    print '%-16s' % 'Category',
    for word in words:                                  # column headings
        print '%6s' % word,
    print
    for category in categories:
        print '%-16s' % category,                       # row heading
        for word in words:                              # for each word
            print '%6d' % cfdist[category][word],       # print table cell
        print                                           # end the row

>>> from nltk.corpus import brown
>>> cfd = nltk.ConditionalFreqDist((g,w)
...                                  for g in brown.categories()
...                                  for w in brown.words(categories=g))
>>> genres = ['news', 'religion', 'hobbies', 'science_fiction', 'romance', 'humor']
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> tabulate(cfd, modals, genres)
Category            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
```

Figure 3.4: Frequency of Modals in Different Sections of the Brown Corpus

Writing Results to a File

We have seen how to read text from files (Section 3.1). It is often useful to write output to files as well. The following code opens a file `output.txt` for writing, and saves the program output to the file.

```
>>> file = open('output.txt', 'w')
>>> words = set(nltk.corpus.genesis.words('english-kjv.txt'))
>>> for word in sorted(words):
...     file.write(word + "\n")
```

When we write non-text data to a file we must convert it to a string first. We can do this conversion using formatting strings, as we saw above. We can also do it using Python's backquote notation, which converts any object into a string. Let's write the total number of words to our file, before closing it.

```
>>> len(words)
2789
>>> `len(words)`
'2789'
>>> file.write(`len(words)` + "\n")
>>> file.close()
```

3.9 Conclusion

In this chapter we saw that we can do a variety of interesting language processing tasks that focus solely on words. Tokenization turns out to be far more difficult than expected. No single solution works well across-the-board, and we must decide what counts as a token depending on the application domain. We also looked at normalization (including lemmatization) and saw how it collapses distinctions between tokens. In the next chapter we will look at word classes and automatic tagging.

3.10 Summary

- In this book we view a text as a list of words. A "raw text" is a potentially long string containing words and whitespace formatting, and is how we typically store and visualize a text.
- A string is specified in Python using single or double quotes: `'Monty Python'`, `"Monty Python"`.
- The characters of a string are accessed using indexes, counting from zero: `'Monty Python'[1]` gives the value `o`. The length of a string is found using `len()`.
- Substrings are accessed using slice notation: `'Monty Python'[1:5]` gives the value `onty`. If the start index is omitted, the substring begins at the start of the string; if the end index is omitted, the slice continues to the end of the string.
- Strings can be split into lists: `'Monty Python'.split()` gives `['Monty', 'Python']`. Lists can be joined into strings: `'/'.join(['Monty', 'Python'])` gives `'Monty/Python'`.
- we can read text from a file `f` using `text = open(f).read()`
- we can read text from a URL `u` using `text = urlopen(u).read()`
- texts found on the web may contain unwanted material (such as headers, footers, markup), that need to be removed before we do any linguistic processing.
- a word token is an individual occurrence of a word in a particular context
- a word type is the vocabulary item, independent of any particular use of that item
- tokenization is the segmentation of a text into basic units — or tokens — such as words and punctuation.
- tokenization based on whitespace is inadequate for many applications because it bundles punctuation together with words
- lemmatization is a process that maps the various forms of a word (such as *appeared*, *appears*) to the canonical or citation form of the word, also known as the lexeme or lemma (e.g. APPEAR).
- Regular expressions are a powerful and flexible method of specifying patterns. Once we have imported the `re` module, we can use `re.findall()` to find all substrings in a string that match a pattern, and we can use `re.sub()` to replace substrings of one sort with another.
- If a regular expression string includes a backslash, you should tell Python not to preprocess the string, by using a raw string with an `r` prefix: `r'regexp'`.
- Normalization of words collapses distinctions, and is useful when indexing texts.

3.11 Further Reading (NOTES)

To learn about Unicode, see 1.

Sources discussing the unreliability of google hits.

as x as y: http://acl.ldc.upenn.edu/P/P07/P07-1008.pdf

Richard Sproat, Alan Black, Stanley Chen, Shankar Kumar, Mari Ostendorf, and Christopher Richards. "Normalization of non-standard words." Computer Speech and Language, 15(3), 287-333, 2001

A.M. Kuchling. *Regular Expression HOWTO*, http://www.amk.ca/python/howto/regex/

For more examples of processing words with NLTK, please see the tokenization, stemming and corpus HOWTOs at http://www.nltk.org/howto. Chapters 2 and 3 of [Jurafsky & Martin, 2008] contain more advanced material on regular expressions and morphology.

For languages with a non-Roman script, tokenizing text is even more challenging. For example, in Chinese text there is no visual representation of word boundaries. The three-character string: 爱国人 (ai4 "love" (verb), guo3 "country", ren2 "person") could be tokenized as 爱国 / 人, "country-loving person" or as 爱 / 国人, "love country-person." The problem of tokenizing Chinese text is a major focus of SIGHAN, the ACL Special Interest Group on Chinese Language Processing `http://sighan.org/`.

Regular Expressions

There are many references for regular expressions, both practical and theoretical. [Friedl, 2002] is a comprehensive and detailed manual in using regular expressions, covering their syntax in most major programming languages, including Python.

For an introductory tutorial to using regular expressions in Python with the `re` module, see A. M. Kuchling, *Regular Expression HOWTO*, http://www.amk.ca/python/howto/regex/.

Chapter 3 of [Mertz, 2003] provides a more extended tutorial on Python's facilities for text processing with regular expressions.

http://www.regular-expressions.info/ is a useful online resource, providing a tutorial and references to tools and other sources of information.

Unicode Regular Expressions: http://www.unicode.org/reports/tr18/

Regex Library: http://regexlib.com/

3.12 Exercises

1. ☼ Describe the class of strings matched by the following regular expressions.

   a) `[a-zA-Z]+`

   b) `[A-Z][a-z]*`

   c) `p[aeiou]{,2}t`

   d) `\d+(\.\d+)?`

   e) `([^aeiou][aeiou][^aeiou])*`

f) `\w+|[^\w\s]+`

Test your answers using `re_show()`.

2. ☼ Write regular expressions to match the following classes of strings:

    a) A single determiner (assume that *a*, *an*, and *the* are the only determiners).
    b) An arithmetic expression using integers, addition, and multiplication, such as `2*3+8`.

3. ☼ Write a utility function that takes a URL as its argument, and returns the contents of the URL, with all HTML markup removed. Use `urllib.urlopen` to access the contents of the URL, e.g. `raw_contents = urllib.urlopen('http://nltk.org/').read()`.

4. ☼ Save some text into a file `corpus.txt`. Define a function `load(f)` that reads from the file named in its sole argument, and returns a string containing the text of the file.

    a) Use `nltk.regexp_tokenize()` to create a tokenizer that tokenizes the various kinds of punctuation in this text. Use a single regular expression, with inline comments using the `re.VERBOSE` flag.
    b) Use `nltk.regexp_tokenize()` to create a tokenizer that tokenizes the following kinds of expression: monetary amounts; dates; names of people and companies.

5. ☼ Rewrite the following loop as a list comprehension:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> result = []
>>> for word in sent:
...     word_len = (word, len(word))
...     result.append(word_len)
>>> result
[('The', 3), ('dog', 3), ('gave', 4), ('John', 4), ('the', 3), ('newspaper', 9)]
```

6. ☼ Split `sent` on some other character, such as `'s'`.

7. ☼ We pointed out that when `phrase` is a list, `phrase.reverse()` returns a modified version of `phrase` rather than a new list. On the other hand, we can use the slice trick mentioned in the exercises for the previous section, `[::-1]` to create a *new* reversed list without changing `phrase`. Show how you can confirm this difference in behavior.

8. ☼ We have seen how to represent a sentence as a list of words, where each word is a sequence of characters. What does `phrase1[2][2]` do? Why? Experiment with other index values.

9. ☼ Write a `for` loop to print out the characters of a string, one per line.

10. ☼ What is the difference between calling `split` on a string with no argument or with `' '` as the argument, e.g. `sent.split()` versus `sent.split(' ')`? What happens when the string being split contains tab characters, consecutive space characters, or a sequence of tabs and spaces? (In IDLE you will need to use `'\t'` to enter a tab character.)

11. ☼ Create a variable `words` containing a list of words. Experiment with `words.sort()` and `sorted(words)`. What is the difference?

12. ☼ Earlier, we asked you to use a text editor to create a file called `test.py`, containing the single line `msg = 'Monty Python'`. If you haven't already done this (or can't find the file), go ahead and do it now. Next, start up a new session with the Python interpreter, and enter the expression `msg` at the prompt. You will get an error from the interpreter. Now, try the following (note that you have to leave off the `.py` part of the filename):

    ```
    >>> from test import msg
    >>> msg
    ```

    This time, Python should return with a value. You can also try `import test`, in which case Python should be able to evaluate the expression `test.msg` at the prompt.

13. ◑ Read in some text from a corpus, tokenize it, and print the list of all *wh*-word types that occur. (*wh*-words in English are used in questions, relative clauses and exclamations: *who*, *which*, *what*, and so on.) Print them in order. Are any words duplicated in this list, because of the presence of case distinctions or punctuation?

14. ◑ Write code to access a favorite webpage and extract some text from it. For example, access a weather site and extract the forecast top temperature for your town or city today.

15. ◑ Write a function `unknown()` that takes a URL as its argument, and returns a list of unknown words that occur on that webpage. In order to do this, extract all substrings consisting of lowercase letters (using `re.findall()`) and remove any items from this set that occur in the words corpus (`nltk.corpus.words`). Try to categorize these words manually and discuss your findings.

16. ◑ Examine the results of processing the URL `http://news.bbc.co.uk/` using the regular expressions suggested above. You will see that there is still a fair amount of non-textual data there, particularly Javascript commands. You may also find that sentence breaks have not been properly preserved. Define further regular expressions that improve the extraction of text from this web page.

17. ◑ Define a function `ghits()` that takes a word as its argument and builds a Google query string of the form `http://www.google.com/search?q=word`. Strip the HTML markup and normalize whitespace. Search for a substring of the form `Results 1 - 10 of about`, followed by some number *n*, and extract *n*. Convert this to an integer and return it.

18. ◑ The above example of extracting (name, domain) pairs from text does not work when there is more than one email address on a line, because the + operator is "greedy" and consumes too much of the input.

    a) Experiment with input text containing more than one email address per line, such as that shown below. What happens?

    b) Using `re.findall()`, write another regular expression to extract email addresses, replacing the period character with a range or negated range, such as `[a-z]+` or `[^ >]+`.

c) Now try to match email addresses by changing the regular expression `.+` to its "non-greedy" counterpart, `.+?`

```
>>> s = """
... austen-emma.txt:hart@vmd.cso.uiuc.edu  (internet)  hart@uiucvmd (bitnet)
... austen-emma.txt:Internet (72600.2026@compuserve.com); TEL: (212-254-5093)
... austen-persuasion.txt:Editing by Martin Ward (Martin.Ward@uk.ac.durham)
... blake-songs.txt:Prepared by David Price, email ccx074@coventry.ac.uk
... """
```

19. ◑ Are you able to write a regular expression to tokenize text in such a way that the word *don't* is tokenized into *do* and *n't*? Explain why this regular expression won't work: «`n't|\w+`».

20. ◑ Write code to convert text into *hAck3r* again, this time using regular expressions and substitution, where `e` → `3`, `i` → `1`, `o` → `0`, `l` → `|`, `s` → `5`, `.` → `5w33t!`, `ate` → `8`. Normalize the text to lowercase before converting it. Add more substitutions of your own. Now try to map `s` to two different values: `$` for word-initial `s`, and `5` for word-internal `s`.

21. ◑ *Pig Latin* is a simple transliteration of English. Each word of the text is converted as follows: move any consonant (or consonant cluster) that appears at the start of the word to the end, then append *ay*, e.g. *string* → *ingstray*, *idle* → *idleay*. `http://en.wikipedia.org/wiki/Pig_Latin`

    a) Write a function to convert a word to Pig Latin.

    b) Write code that converts text, instead of individual words.

    c) Extend it further to preserve capitalization, to keep `qu` together (i.e. so that `quiet` becomes `ietquay`), and to detect when `y` is used as a consonant (e.g. `yellow`) vs a vowel (e.g. `style`).

22. ◑ Download some text from a language that has vowel harmony (e.g. Hungarian), extract the vowel sequences of words, and create a vowel bigram table.

23. ◑ Consider the numeric expressions in the following sentence from the MedLine corpus: *The corresponding free cortisol fractions in these sera were 4.53 +/- 0.15% and 8.16 +/- 0.23%, respectively.* Should we say that the numeric expression *4.53 +/- 0.15%* is three words? Or should we say that it's a single compound word? Or should we say that it is actually *nine* words, since it's read "four point five three, plus or minus fifteen percent"? Or should we say that it's not a "real" word at all, since it wouldn't appear in any dictionary? Discuss these different possibilities. Can you think of application domains that motivate at least two of these answers?

24. ◑ Readability measures are used to score the reading difficulty of a text, for the purposes of selecting texts of appropriate difficulty for language learners. Let us define μ_w to be the average number of letters per word, and μ_s to be the average number of words per sentence, in a given text. The Automated Readability Index (ARI) of the text is defined to be: ``4.71 * `` |mu|\ :subscript:`w` ``+ 0.5 * `` |mu|\ :subscript:`s` ``- 21.43``. Compute the ARI score for various sections of the Brown Corpus, including section `f` (popular lore) and `j` (learned). Make use of the fact that `nltk.corpus.brown.words()` produces a sequence of words, while `nltk.corpus.brown.sents()` produces a sequence of sentences.

25. ◑ Use the Porter Stemmer to normalize some tokenized text, calling the stemmer on each word. Do the same thing with the Lancaster Stemmer and see if you observe any differences.

26. ◑ Process the list `saying` using a `for` loop, and store the result in a new list `lengths`. Hint: begin by assigning the empty list to `lengths`, using `lengths = []`. Then each time through the loop, use `append()` to add another length value to the list.

27. ◑ Define a variable `silly` to contain the string: `'newly formed bland ideas are inexpressible in an infuriating way'`. (This happens to be the legitimate interpretation that bilingual English-Spanish speakers can assign to Chomsky's famous nonsense phrase, *colorless green ideas sleep furiously* according to Wikipedia). Now write code to perform the following tasks:

    a) Split `silly` into a list of strings, one per word, using Python's `split()` operation, and save this to a variable called `bland`.
    b) Extract the second letter of each word in `silly` and join them into a string, to get `'eoldrnnnna'`.
    c) Combine the words in `bland` back into a single string, using `join()`. Make sure the words in the resulting string are separated with whitespace.
    d) Print the words of `silly` in alphabetical order, one per line.

28. ◑ The `index()` function can be used to look up items in sequences. For example, `'inexpressible'.index('e')` tells us the index of the first position of the letter `e`.

    a) What happens when you look up a substring, e.g. `'inexpressible'.index('re')`?
    b) Define a variable `words` containing a list of words. Now use `words.index()` to look up the position of an individual word.
    c) Define a variable `silly` as in the exercise above. Use the `index()` function in combination with list slicing to build a list `phrase` consisting of all the words up to (but not including) `in` in `silly`.

29. ◑ Write code to abbreviate text by removing all the vowels. Define `sentence` to hold any string you like, then initialize a new string `result` to hold the empty string `''`. Now write a `for` loop to process the string, one character at a time, and append any non-vowel characters to the result string.

30. ◑ Write code to convert nationality adjectives like *Canadian* and *Australian* to their corresponding nouns *Canada* and *Australia*. (see `http://en.wikipedia.org/wiki/List_of_adjectival_forms_of_place_names`)

31. ◑ Read the LanguageLog post on phrases of the form *as best as p can* and *as best p can*, where *p* is a pronoun. Investigate this phenomenon with the help of a corpus and the `findall()` method for searching tokenized text described in Section 3.4. `http://itre.cis.upenn.edu/~myl/languagelog/archives/002733.html`

32. ★ An interesting challenge for tokenization is words that have been split across a line-break. E.g. if *long-term* is split, then we have the string `long-\nterm`.

a) Write a regular expression that identifies words that are hyphenated at a line-break. The expression will need to include the `\n` character.

b) Use `re.sub()` to remove the `\n` character from these words.

33. ★ Read the Wikipedia entry on *Soundex*. Implement this algorithm in Python.

34. ★ Define a function `percent(word, text)` that calculates how often a given word occurs in a text, and expresses the result as a percentage.

35. ★ Obtain raw texts from two or more genres and compute their respective reading difficulty scores as in the previous exercise. E.g. compare ABC Rural News and ABC Science News (`nltk.corpus.abc`). Use Punkt to perform sentence segmentation.

36. ★ Rewrite the following nested loop as a nested list comprehension:

```
>>> words = ['attribution', 'confabulation', 'elocution',
...          'sequoia', 'tenacious', 'unidirectional']
>>> vsequences = set()
>>> for word in words:
...     vowels = []
...     for char in word:
...         if char in 'aeiou':
...             vowels.append(char)
...     vsequences.add(''.join(vowels))
>>> sorted(vsequences)
['aiuio', 'eaiou', 'eouio', 'euoia', 'oauaio', 'uiieioa']
```

37. ★ Write a program that processes a text and discovers cases where a word has been used with a novel sense. For each word, compute the wordnet similarity between all synsets of the word and all synsets of the words in its context. (Note that this is a crude approach; doing it well is an open research problem.)