Chapter 2

Words: The Building Blocks of Language

2.1 Introduction

Language can be divided up into pieces of varying sizes, ranging from morphemes to paragraphs. In this chapter we will focus on words, the most fundamental level for NLP. Just what are words, and how should we represent them in a machine? These questions may seem trivial, but we'll see that there are some important issues involved in defining and representing words. Once we've tackled them, we're in a good position to do further processing, such as find related words and analyze the style of a text (this chapter), to categorize words (Chapter 3), to group them into phrases (Chapter 6 and Part II), and to do a variety of data-intensive language processing tasks (Chapter 4).

In the following sections, we will explore the division of text into words; the distinction between types and tokens; sources of text data including files, the web, and linguistic corpora; accessing these sources using Python and NLTK; stemming and normalization; the WordNet lexical database; and a variety of useful programming tasks involving words.

> **Note**
>
> From this chapter onwards, our program samples will assume you begin your interactive session or your program with: `import nltk, re, pprint`

2.2 Tokens, Types and Texts

In Chapter chap-programming, we showed how a string could be split into a list of words. Once we have derived a list, the `len()` function will count the number of words it contains:

```
>>> sentence = "This is the time -- and this is the record of the time."
>>> words = sentence.split()
>>> len(words)
13
```

This process of segmenting a string of characters into words is known as **tokenization**. Tokenization is a prelude to pretty much everything else we might want to do in NLP, since it tells our processing software what our basic units are. We will discuss tokenization in more detail shortly.

We also pointed out that we could compile a list of the unique vocabulary items in a string by using `set()` to eliminate duplicates:

```
>>> len(set(words))
10
```

So if we ask how many words there are in `sentence`, we get different answers depending on whether we count duplicates. Clearly we are using different senses of "word" here. To help distinguish between them, let's introduce two terms: **token** and **type**. A word token is an individual occurrence of a word in a concrete context; it exists in time and space. A word **type** is a more abstract; it's what we're talking about when we say that the three occurrences of `the` in `sentence` are "the same word."

Something similar to a type-token distinction is reflected in the following snippet of Python:

```
>>> words[2]
'the'
>>> words[2] == words[8]
True
>>> words[2] is words[8]
False
>>> words[2] is words[2]
True
```

The operator == tests whether two expressions are equal, and in this case, it is testing for string-identity. This is the notion of identity that was assumed by our use of `set()` above. By contrast, the `is` operator tests whether two objects are stored in the same location of memory, and is therefore analogous to token-identity. When we used `split()` to turn a string into a list of words, our tokenization method was to say that any strings that are delimited by whitespace count as a word token. But this simple approach doesn't always give the desired results. Also, testing string-identity isn't a very useful criterion for assigning tokens to types. We therefore need to address two questions in more detail: *Tokenization:* Which substrings of the original text should be treated as word tokens? *Type definition:* How do we decide whether two tokens have the same type?

To see the problems with our first stab at defining tokens and types in `sentence`, let's look at the actual tokens we found:

```
>>> set(words)
set(['and', 'this', 'record', 'This', 'of', 'is', '--', 'time.', 'time', 'the'])
```

Observe that `'time'` and `'time.'` are incorrectly treated as distinct types since the trailing period has been bundled with the rest of the word. Although `'--'` is some kind of token, it's not a *word* token. Additionally, `'This'` and `'this'` are incorrectly distinguished from each other, because of a difference in capitalization that should be ignored.

If we turn to languages other than English, tokenizing text is even more challenging. In Chinese text there is no visual representation of word boundaries. Consider the following three-character string: 爱国人 (in pinyin plus tones: ai4 "love" (verb), guo3 "country", ren2 "person"). This could either be segmented as [爱国]人, "country-loving person" or as 爱[国人], "love country-person."

The terms *token* and *type* can also be applied to other linguistic entities. For example, a **sentence token** is an individual occurrence of a sentence; but a **sentence type** is an abstract sentence, without context. If I say the same sentence twice, I have uttered two sentence tokens but only used one sentence type. When the kind of token or type is obvious from context, we will simply use the terms token and type.

To summarize, we cannot just say that two word tokens have the same type if they are the same string of characters. We need to consider a variety of factors in determining what counts as the same word, and we need to be careful in how we identify tokens in the first place.

Up till now, we have relied on getting our source texts by defining a string in a fragment of Python code. However, this is impractical for all but the simplest of texts, and makes it hard to present realistic examples. So how do we get larger chunks of text into our programs? In the rest of this section, we will see how to extract text from files, from the web, and from the corpora distributed with NLTK.

2.2.1 Extracting Text from Files

It is easy to access local files in Python. As an exercise, create a file called `corpus.txt` using a text editor, and enter the following text:

```
Hello World!
This is a test file.
```

Be sure to save the file as plain text. You also need to make sure that you have saved the file in the same directory or folder in which you are running the Python interactive interpreter.

> **Note**
>
> If you are using IDLE, you can easily create this file by selecting the *New Window* command in the *File* menu, typing the required text into this window, and then saving the file as `corpus.txt` in the first directory that IDLE offers in the pop-up dialogue box.

The next step is to **open** a file using the built-in function `open()` which takes two arguments, the name of the file, here `corpus.txt`, and the mode to open the file with (`'r'` means to open the file for reading, and `'U'` stands for "Universal", which lets us ignore the different conventions used for marking newlines).

```
>>> f = open('corpus.txt', 'rU')
```

> **Note**
>
> If the interpreter cannot find your file, it will give an error like this:
>
> ```
> >>> f = open('corpus.txt', 'rU')
> Traceback (most recent call last):
>     File "<pyshell#7>", line 1, in -toplevel-
>     f = open('corpus.txt', 'rU')
> IOError: [Errno 2] No such file or directory: 'corpus.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('.')
```

There are several methods for reading the file. The following uses the method `read()` on the file object `f`; this reads the entire contents of a file into a string.

```
>>> f.read()
'Hello World!\nThis is a test file.\n'
```

Recall that the `'\n'` characters are **newlines**; this is equivalent to pressing *Enter* on a keyboard and starting a new line. Note that we can open and read a file in one step:

```
>>> text = open('corpus.txt', 'rU').read()
```

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

```
>>> f = open('corpus.txt', 'rU')
>>> for line in f:
...     print line[:-1]
Hello world!
This is a test file.
```

Here we use the slice `[:-1]` to remove the newline character at the end of the input line.

2.2.2 Extracting Text from the Web

Opening a web page is not much different to opening a file, except that we use `urlopen()`:

```
>>> from urllib import urlopen
>>> page = urlopen("http://news.bbc.co.uk/").read()
>>> print page[:60]
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN
```

Web pages are usually in HTML format. To extract the text, we need to strip out the HTML markup, i.e. remove all material enclosed in angle brackets. Let's digress briefly to consider how to carry out this task using regular expressions. Our first attempt might look as follows:

```
>>> line = '<title>BBC NEWS | News Front Page</title>'
>>> new = re.sub(r'<.*>', '', line)
```

So the regular expression `'<.*>'` is intended to match a pair of left and right angle brackets, with a string of any characters intervening. However, look at what the result is:

```
>>> new
''
```

What has happened here? The problem is twofold. First, the wildcard `'.'` matches any character other than `'\n'`, so it will match `'>'` and `'<'`. Second, the `'*'` operator is "greedy", in the sense that it matches as many characters as it can. In the above example, `'.*'` will return not the shortest match, namely `'title'`, but the longest match, `'title>BBC NEWS | News Front Page</title'`. To get the *shortest* match we have to use the `'*?'` operator. We will also normalize whitespace, replacing any sequence of spaces, tabs or newlines (`'\s+'`) with a single space character.

```
>>> page = re.sub('<.*?>', '', page)
>>> page = re.sub('\s+', ' ', page)
>>> print page[:60]
 BBC NEWS | News Front Page News Sport Weather World Service
```

> **Note**
>
> Note that your output for the above code may differ from ours, because the BBC home page may have been changed since this example was created.

You will probably find it useful to borrow the structure of the above code snippet for future tasks involving regular expressions: each time through a series of substitutions, the result of operating on `page` gets assigned as the new value of `page`. This approach allows us to decompose the transformations we need into a series of simple regular expression substitutions, each of which can be tested and debugged on its own.

> **Note**
>
> 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 text.

2.2.3 Extracting Text from NLTK Corpora

NLTK is distributed with several corpora and corpus samples and many are supported by the `corpus` package. Here we use a selection of texts from the Project Gutenberg electronic text archive, and list the files it contains:

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

We can count the number of tokens for each text in our Gutenberg sample as follows:

```
>>> for book in nltk.corpus.gutenberg.files():
...     print book + ':', len(nltk.corpus.gutenberg.words(book))
austen-emma.txt: 192432
austen-persuasion.txt: 98191
austen-sense.txt: 141586
bible-kjv.txt: 1010735
blake-poems.txt: 8360
blake-songs.txt: 6849
chesterton-ball.txt: 97396
chesterton-brown.txt: 89090
chesterton-thursday.txt: 69443
milton-paradise.txt: 97400
shakespeare-caesar.txt: 26687
shakespeare-hamlet.txt: 38212
shakespeare-macbeth.txt: 23992
whitman-leaves.txt: 154898
```

> **Note**
>
> It is possible to use the methods described in section 2.2.1 along with `nltk.data.find()` method to access and read the corpus files directly. The method described in this section is superior since it takes care of tokenization and conveniently skips over the Gutenberg file header.

But note that this has several disadvantages. The ones that come to mind immediately are: (i) The corpus reader automatically strips out the Gutenberg header; this version doesn't. (ii) The corpus reader uses a somewhat smarter method to break lines into words; this version just splits on whitespace. (iii)

Using the corpus reader, you can also access the documents by sentence or paragraph; doing that by hand, you'd need to do some extra work.

The Brown Corpus was the first million-word, part-of-speech tagged electronic corpus of English, created in 1961 at Brown University. Each of the sections a through r represents a different genre, as shown in Table 2.1.

Sec	Genre	Sec	Genre	Sec	Genre
a	Press: Reportage	b	Press: Editorial	c	Press: Reviews
d	Religion	e	Skill and Hobbies	f	Popular Lore
g	Belles-Lettres	h	Government	j	Learned
k	Fiction: General	k	Fiction: General	l	Fiction: Mystery
m	Fiction: Science	n	Fiction: Adventure	p	Fiction: Romance
r	Humor				

Table 2.1: Sections 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 a section of the corpus to read:

```
>>> nltk.corpus.brown.categories()
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'r']
>>> nltk.corpus.brown.words(categories='a')
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
>>> nltk.corpus.brown.sents(categories='a')
[['The', 'Fulton', 'County'...], ['The', 'jury', 'further'...], ...]
```

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.

```
>>> print nltk.corpus.nps_chat.words()
['now', 'im', 'left', 'with', 'this', 'gay', 'name', ...]
>>> 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.words('Javanese-Latin1')[11:]
['Saben', 'umat', 'manungsa', 'lair', 'kanthi', 'hak', ...]
>>> 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\x
```

Before concluding this section, we return to the original topic of distinguishing tokens and types. Now that we can access substantial quantities of text, we will give a preview of the interesting computations we will be learning how to do (without yet explaining all the details). Listing 2.1 computes vocabulary growth curves for US Presidents, shown in Figure 2.1 (a color figure in the online version). These curves show the number of word types seen after *n* word tokens have been read.

Note

Listing 2.1 uses the PyLab package which supports sophisticated plotting functions with a MATLAB-style interface. For more information about this package please see http://matplotlib.sourceforge.net/. The listing also uses the yield statement, which will be explained in Chapter 5.

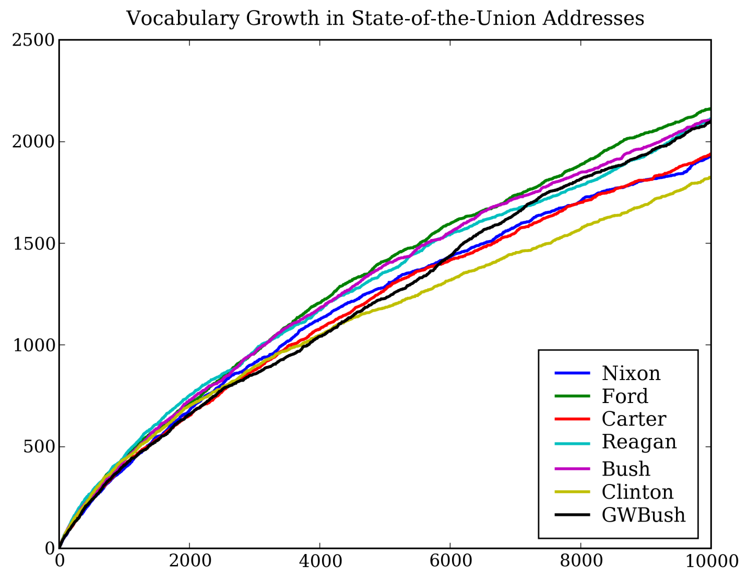

Figure 2.1: Vocabulary Growth in State-of-the-Union Addresses

2.2.4 Exercises

1. ☼ Create a small text file, and write a program to read it and print it with a line number at the start of each line. (Make sure you don't introduce an extra blank line between each line.)

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

3. ☼ 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.

4. ☼ Use the Brown corpus reader `nltk.corpus.brown.sents()` to find sentence-initial examples of the word *however*. Check whether these conform to Strunk and White's prohibition against sentence-initial *however* used to mean "although".

5. ☼ 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?

6. ◑ Write code to read a file and print the lines in reverse order, so that the last line is listed first.

7. ◑ 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?

Listing 2.1 Vocabulary Growth in State-of-the-Union Addresses

```
def vocab_growth(texts):
    vocabulary = set()
    for text in texts:
        for word in text:
            vocabulary.add(word)
            yield len(vocabulary)

def speeches():
    presidents = []
    texts = nltk.defaultdict(list)
    for speech in nltk.corpus.state_union.files():
        president = speech.split('-')[1]
        if president not in texts:
            presidents.append(president)
        texts[president].append(nltk.corpus.state_union.words(speech))
    return [(president, texts[president]) for president in presidents]

>>> import pylab
>>> for president, texts in speeches()[-7:]:
...     growth = list(vocab_growth(texts))[:10000]
...     pylab.plot(growth, label=president, linewidth=2)
>>> pylab.title('Vocabulary Growth in State-of-the-Union Addresses')
>>> pylab.legend(loc='lower right')
>>> pylab.show()
```

8. ◑ 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.

9. ◑ 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.

10. ◑ 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.

11. ◑ Take a copy of the `http://news.bbc.co.uk/` over three different days, say at two-day intervals. This should give you three different files, `bbc1.txt`, `bbc2.txt` and `bbc3.txt`, each corresponding to a different snapshot of world events. Collect the 100 most frequent word tokens for each file. What can you tell from the changes in frequency?

12. ◑ 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.

13. ◑ Try running the various chatbots included with NLTK, using `nltk.chat.demo()`. How *intelligent* are these programs? Take a look at the program code and see if you can discover how it works. You can find the code online at: `http://nltk.org/nltk/chat/`.

14. ★ 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.

2.3 Tokenization and Normalization

Tokenization, as we saw, is the task of extracting a sequence of elementary tokens that constitute a piece of language data. In our first attempt to carry out this task, we started off with a string of characters, and used the `split()` method to break the string at whitespace characters. Recall that "whitespace" covers not only inter-word space, but also tabs and newlines. We pointed out that tokenization based solely on whitespace is too simplistic for most applications. In this section we will take a more sophisticated approach, using regular expressions to specify which character sequences should be treated as words. We will also look at ways to normalize tokens.

2.3.1 Tokenization with Regular Expressions

The function `nltk.tokenize.regexp_tokenize()` takes a text string and a regular expression, and returns the list of substrings that match the regular expression. To define a tokenizer that includes punctuation as separate tokens, we could do the following:

```
>>> text = '''Hello.  Isn't this fun?'''
>>> pattern = r'\w+|[^\w\s]+'
>>> nltk.tokenize.regexp_tokenize(text, pattern)
['Hello', '.', 'Isn', "'", 't', 'this', 'fun', '?']
```

The regular expression in this example will match a sequence consisting of one or more word characters `\w+`. It will also match a sequence consisting of one or more punctuation characters (or non-word, non-space characters `[^\w\s]+`). This is another negated range expression; it matches one or more characters that are not word characters (i.e., not a match for `\w`) and not a whitespace character (i.e., not a match for `\s`). We use the disjunction operator `|` to combine these into a single complex expression `\w+|[^\w\s]+`.

There are a number of ways we could improve on this regular expression. For example, it currently breaks *$22.50* into four tokens; we might want it to treat this as a single token. Similarly, *U.S.A.* should count as a single token. We can deal with these by adding further cases to the regular expression. For readability we will break it up and insert comments, and insert the special `(?x)` "verbose flag" so that Python knows to strip out the embedded whitespace and comments.

```
>>> text = 'That poster costs $22.40.'
>>> pattern = r'''(?x)
...      \w+                # sequences of 'word' characters
...    | \$?\d+(\.\d+)?     # currency amounts, e.g. $12.50
...    | ([A-Z]\.)+         # abbreviations, e.g. U.S.A.
...    | [^\w\s]+           # sequences of punctuation
... '''
>>> nltk.tokenize.regexp_tokenize(text, pattern)
['That', 'poster', 'costs', '$22.40', '.']
```

It is sometimes more convenient to write a regular expression matching the material that appears *between* tokens, such as whitespace and punctuation. The `nltk.tokenize.regexp_tokenize()` function permits an optional boolean parameter `gaps`; when set to `True` the pattern is matched against the gaps. For example, we could define a whitespace tokenizer as follows:

```
>>> nltk.tokenize.regexp_tokenize(text, pattern=r'\s+', gaps=True)
['That', 'poster', 'costs', '$22.40.']
```

It is more convenient to call NLTK's whitespace tokenizer directly, as `nltk.WhitespaceTokenizer(text)`. (However, in this case is generally better to use Python's `split()` method, defined on strings: `text.split()`.)

2.3.2 Lemmatization and Normalization

Earlier we talked about counting word tokens, and completely ignored the rest of the sentence in which these tokens appeared. Thus, for an example like *I saw the saw*, we would have treated both *saw* tokens as instances of the same type. However, one is a form of the verb *see*, and the other is the name of a cutting instrument. How do we know that these two forms of *saw* are unrelated? One answer is that as speakers of English, we know that these would appear as different entries in a dictionary. Another, more

empiricist, answer is that if we looked at a large enough number of texts, it would become clear that the two forms have very different distributions. For example, only the noun *saw* will occur immediately after determiners such as *the*. Distinct words that have the same written form are called **homographs**. We can distinguish homographs with the help of context; often the previous word suffices. We will explore this idea of context briefly, before addressing the main topic of this section.

As a first approximation to discovering the distribution of a word, we can look at all the bigrams it occurs in. A **bigram** is simply a pair of words. For example, in the sentence *She sells sea shells by the sea shore*, the bigrams are *She sells*, *sells sea*, *sea shells*, *shells by*, *by the*, *the sea*, *sea shore*. Let's consider all bigrams from the Brown Corpus that have the word *often* as first element. Here is a small selection, ordered by their counts:

```
often ,           16
often a           10
often in          8
often than        7
often the         7
often been        6
often do          5
often called      4
often appear      3
often were        3
often appeared    2
often are         2
often did         2
often is          2
often appears     1
often call        1
```

In the topmost entry, we see that *often* is frequently followed by a comma. This suggests that *often* is common at the end of phrases. We also see that *often* precedes verbs, presumably as an adverbial modifier. We might conclude that when *saw* appears in the context *often saw*, then *saw* is being used as a verb.

You will also see that this list includes different grammatical forms of the same verb. We can form separate groups consisting of *appear* ~ *appears* ~ *appeared*; *call* ~ *called*; *do* ~ *did*; and *been* ~ *were* ~ *are* ~ *is*. It is common in linguistics to say that two forms such as *appear* and *appeared* belong to a more abstract notion of a word called a **lexeme**; by contrast, *appeared* and *called* belong to different lexemes. You can think of a lexeme as corresponding to an entry in a dictionary, and a **lemma** as the headword for that entry. By convention, small capitals are used when referring to a lexeme or lemma: APPEAR.

Although *appeared* and *called* belong to different lexemes, they do have something in common: they are both past tense forms. This is signaled by the segment *-ed*, which we call a morphological **suffix**. We also say that such morphologically complex forms are **inflected**. If we strip off the suffix, we get something called the **stem**, namely *appear* and *call* respectively. While *appeared*, *appears* and *appearing* are all morphologically inflected, *appear* lacks any morphological inflection and is therefore termed the **base** form. In English, the base form is conventionally used as the **lemma** for a word.

Our notion of context would be more compact if we could group different forms of the various verbs into their lemmas; then we could study which verb lexemes are typically modified by a particular adverb. **Lemmatization** — the process of mapping words to their lemmas — would yield the following picture of the distribution of *often*. Here, the counts for *often appear* (3), *often appeared* (2) and *often appears* (1) are combined into a single line.

```
often ,            16
often a            10
often be           13
often in           8
often than         7
often the          7
often do           7
often appear       6
often call         5
```

Lemmatization is a rather sophisticated process that uses rules for the regular word patterns, and table look-up for the irregular patterns. Within NLTK, we can use off-the-shelf stemmers, such as the **Porter Stemmer**, the **Lancaster Stemmer**, and the stemmer that comes with WordNet, e.g.:

```
>>> stemmer = nltk.PorterStemmer()
>>> verbs = ['appears', 'appear', 'appeared', 'calling', 'called']
>>> stems = []
>>> for verb in verbs:
...     stemmed_verb = stemmer.stem(verb)
...     stems.append(stemmed_verb)
>>> sorted(set(stems))
['appear', 'call']
```

Stemmers for other languages are added to NLTK as they are contributed, e.g. the RSLP Portuguese Stemmer, `nltk.RSLPStemmer()`.

Lemmatization and stemming are special cases of **normalization**. They identify a canonical representative for a set of related word forms. Normalization collapses distinctions. Exactly how we normalize words depends on the application. Often, we convert everything into lower case so that we can ignore the written distinction between sentence-initial words and the rest of the words in the sentence. The Python string method `lower()` will accomplish this for us:

```
>>> str = 'This is the time'
>>> str.lower()
'this is the time'
```

A final issue for normalization 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*).

2.3.3 Transforming Lists

Lemmatization and normalization involve applying the same operation to each word token in a text. **List comprehensions** are a convenient Python construct for doing this. Here we lowercase each word:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> [word.lower() for word in sent]
['the', 'dog', 'gave', 'john', 'the', 'newspaper']
```

A list comprehension usually has the form `[item.foo() for item in sequence]`, or `[foo(item) for item in sequence]`. It creates a list but applying an operation to every item in the supplied sequence. Here we rewrite the loop for identifying verb stems that we saw in the previous section:

```
>>> [stemmer.stem(verb) for verb in verbs]
['appear', 'appear', 'appear', 'call', 'call']
```

Now we can eliminate repeats using `set()`, by passing the list comprehension as an argument. We can actually leave out the square brackets, as will be explained further in Chapter 9.

```
>>> set(stemmer.stem(verb) for verb in verbs)
set(['call', 'appear'])
```

This syntax might be reminiscent of the notation used for building sets, e.g. $\{(x,y) \mid x^2 + y^2 = 1\}$. (We will return to sets later in Section 9). Just as this set definition incorporates a constraint, list comprehensions can constrain the items they include. In the next example we remove some non-content words from a list of words:

```
>>> def is_lexical(word):
...     return word.lower() not in ('a', 'an', 'the', 'that', 'to')
>>> [word for word in sent if is_lexical(word)]
['dog', 'gave', 'John', 'newspaper']
```

Now we can combine the two ideas (constraints and normalization), to pull out the content words and normalize them.

```
>>> [word.lower() for word in sent if is_lexical(word)]
['dog', 'gave', 'john', 'newspaper']
```

List comprehensions can build nested structures too. For example, the following code builds a list of tuples, where each tuple consists of a word and its stem.

```
>>> sent = nltk.corpus.brown.sents(categories='a')[0]
>>> [(x, stemmer.stem(x).lower()) for x in sent]
[('The', 'the'), ('Fulton', 'fulton'), ('County', 'counti'),
('Grand', 'grand'), ('Jury', 'juri'), ('said', 'said'), ('Friday', 'friday'),
('an', 'an'), ('investigation', 'investig'), ('of', 'of'),
("Atlanta's", "atlanta'"), ('recent', 'recent'), ('primary', 'primari'),
('election', 'elect'), ('produced', 'produc'), ('``', '``'), ('no', 'no'),
('evidence', 'evid'), ("''", "''"), ('that', 'that'), ('any', 'ani'),
('irregularities', 'irregular'), ('took', 'took'), ('place', 'place'), ('.', '.')]
```

2.3.4 Sentence Segmentation

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 doing word tokenization, we need to do sentence segmentation. 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(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,
 '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 p
 'It is because after they have\npassed Sloane Square they know that the next stati
 'Oh, their wild rapture!',
 'oh,\ntheir eyes like stars and their souls again in Eden, if the next\nstation we
 '"\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.

2.3.5 Exercises

1. ☼ **Regular expression tokenizers:** 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.tokenize.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.tokenize.regexp_tokenize()` to create a tokenizer that tokenizes the following kinds of expression: monetary amounts; dates; names of people and companies.

2. ☼ 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)
```

3. ◑ 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.

4. ◑ 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?

5. ◑ 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.

6. ★ 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.

7. ★ 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']
```

2.4 Counting Words: Several Interesting Applications

Now that we can count words (tokens or types), we can write programs to perform a variety of useful tasks, to study stylistic differences in language use, differences between languages, and even to generate random text.

Before getting started, we need to see how to get Python to count the number of occurrences of *each* word in a document.

```
>>> counts = nltk.defaultdict(int)                    ①
>>> sec_a = nltk.corpus.brown.words(categories='a')
>>> for token in sec_a:
...     counts[token] += 1                            ②
>>> for token in sorted(counts)[:5]:                  ③
...     print counts[token], token
38 !
5 $1
2 $1,000
1 $1,000,000,000
3 $1,500
```

In line ① we initialize the dictionary. Then for each word in each sentence we increment a counter (line ②). To view the contents of the dictionary, we can iterate over its keys and print each entry (here just for the first 5 entries, line ③).

2.4.1 Frequency Distributions

This style of output and our `counts` object are just different forms of the same abstract structure — a collection of items and their frequencies — known as a **frequency distribution**. Since we will often need to count things, NLTK provides a `FreqDist()` class. We can write the same code more conveniently as follows:

```
>>> fd = nltk.FreqDist(sec_a)
>>> for token in sorted(fd)[:5]:
...     print fd[token], token
38 !
5 $1
2 $1,000
1 $1,000,000,000
3 $1,500
```

Some of the methods defined on NLTK frequency distributions are shown in Table 2.2.

Name	Sample	Description
Count	`fd['the']`	number of times a given sample occurred
Frequency	`fd.freq('the')`	frequency of a given sample
N	`fd.N()`	number of samples
Samples	`list(fd)`	list of distinct samples recorded (also `fd.keys()`)
Max	`fd.max()`	sample with the greatest number of outcomes

Table 2.2: Frequency Distribution Module

This output isn't very interesting. Perhaps it would be more informative to list the most frequent word tokens first. Now a `FreqDist` object is just a kind of dictionary, so we can easily get its key-value pairs and sort them by decreasing values, as follows:

```
>>> from operator import itemgetter
>>> sorted_word_counts = sorted(fd.items(), key=itemgetter(1), reverse=True)   ①
>>> [token for (token, freq) in sorted_word_counts[:20]]
['the', ',', '.', 'of', 'and', 'to', 'a', 'in', 'for', 'The', 'that',
'``', 'is', 'was', "''", 'on', 'at', 'with', 'be', 'by']
```

Note the arguments of the `sorted()` function (line ①): `itemgetter(1)` returns a function that can be called on any sequence object to return the item at position `1`; `reverse=True` performs the sort in reverse order. Together, these ensure that the word with the highest frequency is listed first. This reversed sort by frequency is such a common requirement that it is built into the `FreqDist` object. Listing 2.2 demonstrates this, and also prints rank and cumulative frequency.

Unfortunately the output in Listing 2.2 is surprisingly dull. A mere handful of tokens account for a third of the text. They just represent the plumbing of English text, and are completely uninformative! How can we find words that are more indicative of a text? As we will see in the exercises for this section, we can modify the program to discard the non-content words. In the next section we see another approach.

Listing 2.2 Words and Cumulative Frequencies, in Order of Decreasing Frequency

```
def print_freq(tokens, num=50):
    fd = nltk.FreqDist(tokens)
    cumulative = 0.0
    rank = 0
    for word in fd.sorted()[:num]:
        rank += 1
        cumulative += fd[word] * 100.0 / fd.N()
        print "%3d %3.2d%% %s" % (rank, cumulative, word)

>>> print_freq(nltk.corpus.brown.words(categories='a'), 20)
  1  05% the
  2  10% ,
  3  14% .
  4  17% of
  5  19% and
  6  21% to
  7  23% a
  8  25% in
  9  26% for
 10  27% The
 11  28% that
 12  28% ``
 13  29% is
 14  30% was
 15  31% ''
 16  31% on
 17  32% at
 18  32% with
 19  33% be
 20  33% by
```

2.4.2 Stylistics

Stylistics is a broad term covering literary genres and varieties of language use. Here we will look at a document collection that is categorized by genre, and try to learn something about the patterns of word usage. For example, Table 2.3 was constructed by counting the number of times various modal words appear in different sections of the corpus:

Genre	can	could	may	might	must	will
skill and hobbies	273	59	130	22	83	259
humor	17	33	8	8	9	13
fiction: science	16	49	4	12	8	16
press: reportage	94	86	66	36	50	387
fiction: romance	79	195	11	51	46	43
religion	84	59	79	12	54	64

Table 2.3: Use of Modals in Brown Corpus, by Genre

Observe that the most frequent modal in the reportage 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.

We can also measure the lexical diversity of a genre, by calculating the ratio of word types and word tokens, as shown in Table 2.4. Genres with lower diversity have a higher number of tokens per type, thus we see that humorous prose is almost twice as lexically diverse as romance prose.

Genre	Token Count	Type Count	Ratio
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 2.4: Lexical Diversity of Various Genres in the Brown Corpus

We can carry out a variety of interesting explorations simply by counting words. In fact, the field of **Corpus Linguistics** focuses heavily on creating and interpreting such tables of word counts.

2.4.3 Aside: Defining Functions

It often happens that part of a program needs to be used several times over. For example, suppose we were writing a program that needed to be able to form the plural of a singular noun, and that this needed to be done at various places during the program. 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 a programming construct that can be called with one or more inputs and which returns an output. We define a function using the keyword `def` followed by the function name and any input parameters,

followed by a colon; this in turn is followed by the body of the function. We use the keyword `return` to indicate the value that is produced as output by the function. The best way to convey this is with an example. Our function `plural()` in Listing 2.3 takes a singular noun and generates a plural form (one which is not always correct).

Listing 2.3 Example of a Python function

```
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'
```

(There is much more to be said about ways of defining functions, but we will defer this until Section 5.4.)

2.4.4 Lexical Dispersion

Word tokens vary in their distribution throughout a text. We can visualize word distributions to get an overall sense of topics and topic shifts. For example, consider the pattern of mention of the main characters in Jane Austen's *Sense and Sensibility*: Elinor, Marianne, Edward and Willoughby. The following plot contains four rows, one for each name, in the order just given. Each row contains a series of lines, drawn to indicate the position of each token.

Figure 2.2: Lexical Dispersion Plot for the Main Characters in *Sense and Sensibility*

As you can see, *Elinor* and *Marianne* appear rather uniformly throughout the text, while *Edward* and *Willoughby* tend to appear separately. Here is the code that generated the above plot.

```
>>> names = ['Elinor', 'Marianne', 'Edward', 'Willoughby']
>>> text = nltk.corpus.gutenberg.words('austen-sense.txt')
>>> nltk.draw.dispersion_plot(text, names)
```

2.4.5 Comparing Word Lengths in Different Languages

We can use a frequency distribution to examine the distribution of word lengths in a corpus. For each word, we find its length, and increment the count for words of this length.

```
>>> def print_length_dist(text):
...     fd = nltk.FreqDist(len(token) for token in text if re.match(r'\w+$', token))
...     for i in range(1,15):
...         print "%2d" % int(100*fd.freq(i)),
...     print
```

Now we can call `print_length_dist` on a text to print the distribution of word lengths. We see that the most frequent word length for the English sample is 3 characters, while the most frequent length for the Finnish sample is 5-6 characters.

```
>>> print_length_dist(nltk.corpus.genesis.words('english-kjv.txt'))
 2 15 30 23 12  6  4  2  1  0  0  0  0  0
>>> print_length_dist(nltk.corpus.genesis.words('finnish.txt'))
 0 12  6 10 17 17 11  9  5  3  2  1  0  0
```

This is an intriguing area for exploration, and so in Listing 2.4 we look at it on a larger scale using the Universal Declaration of Human Rights corpus, which has text samples from over 300 languages. (Note that the names of the files in this corpus include information about character encoding; here we will use texts in ISO Latin-1.) The output is shown in Figure 2.3 (a color figure in the online version).

Listing 2.4 Cumulative Word Length Distributions for Several Languages

```
import pylab

def cld(lang):
    text = nltk.corpus.udhr.words(lang)
    fd = nltk.FreqDist(len(token) for token in text)
    ld = [100*fd.freq(i) for i in range(36)]
    return [sum(ld[0:i+1]) for i in range(len(ld))]

>>> langs = ['Chickasaw-Latin1', 'English-Latin1',
...          'German_Deutsch-Latin1', 'Greenlandic_Inuktikut-Latin1',
...          'Hungarian_Magyar-Latin1', 'Ibibio_Efik-Latin1']
>>> dists = [pylab.plot(cld(l), label=l[:-7], linewidth=2) for l in langs]
>>> pylab.title('Cumulative Word Length Distributions for Several Languages')
>>> pylab.legend(loc='lower right')
>>> pylab.show()
```

2.4.6 Generating Random Text with Style

We have used frequency distributions to count the number of occurrences of each word in a text. Here we will generalize this idea to look at the distribution of words in a given context. A **conditional frequency distribution** is a collection of frequency distributions, each one for a different condition. Here the condition will be the preceding word.

In Listing 2.5, we've defined a function `train_model()` that uses `ConditionalFreqDist()` to count words as they appear relative to the context defined by the preceding word (stored in `prev`). It scans the corpus, incrementing the appropriate counter, and updating the value of `prev`. The function `generate_model()` contains a simple loop to generate text: we set an initial context, pick the most likely token in that context as our next word (using `max()`), and then use that word as our new context. 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.

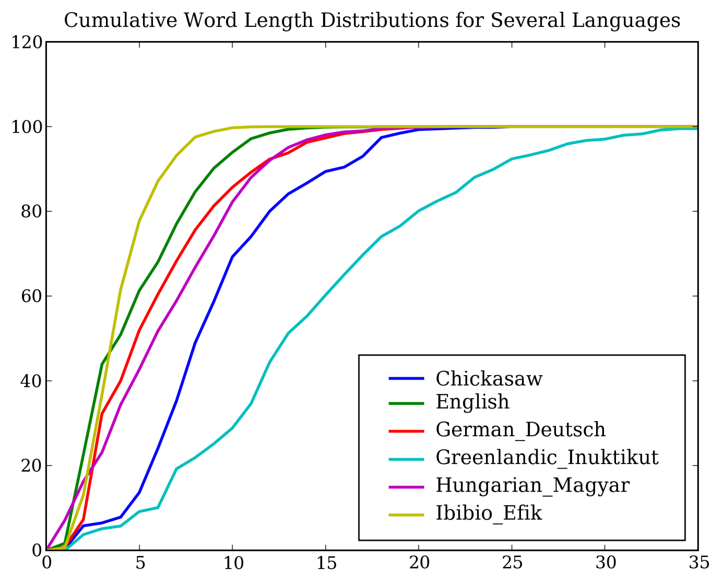

Figure 2.3: Cumulative Word Length Distributions for Several Languages

Listing 2.5 Generating Random Text in the Style of Genesis

```
def train_model(text):
    cfdist = nltk.ConditionalFreqDist()
    prev = None
    for word in text:
        cfdist[prev].inc(word)
        prev = word
    return cfdist

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

>>> model = train_model(nltk.corpus.genesis.words('english-kjv.txt'))
>>> model['living']
<FreqDist with 16 samples>
>>> list(model['living'])
['substance', ',', '.', 'thing', 'soul', 'creature']
>>> generate_model(model, 'living')
living creature that he said , and the land of the land of the land
```

2.4.7 Collocations

Collocations are pairs of content words that occur together more often than one would expect if the words of a document were scattered randomly. We can find collocations by counting how many times a pair of words w_1, w_2 occurs together, compared to the overall counts of these words (this program uses a heuristic related to the **mutual information** measure, `http://www.collocations.de/`) In Listing 2.6 we try this for the files in the webtext corpus.

2.4.8 Exercises

1. ☺ Compare the lexical dispersion plot with Google Trends, which shows the frequency with which a term has been referenced in news reports or been used in search terms over time.

2. ☼ Pick a text, and explore the dispersion of particular words. What does this tell you about the words, or the text?

3. ☼ The program in Listing 2.2 used a dictionary of word counts. Modify the code that creates these word counts so that it ignores non-content words. You can easily get a list of words to ignore with:

```
>>> ignored_words = nltk.corpus.stopwords.words('english')
```

4. ☼ Modify the `generate_model()` function in Listing 2.5 to use Python's `random.choice()` method to randomly pick the next word from the available set of words.

5. ☼ **The demise of teen language:** 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." Use the program in Listing 2.2 to find out 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`.

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

7. ◑ Write a program to generate a table of token/type ratios, as we saw in Table 2.4. 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?

8. ◑ Modify the text generation program in Listing 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.

Listing 2.6 A Simple Program to Find Collocations

```
def collocations(words):
    from operator import itemgetter

    # Count the words and bigrams
    wfd = nltk.FreqDist(words)
    pfd = nltk.FreqDist(tuple(words[i:i+2]) for i in range(len(words)-1))

    #
    scored = [((w1,w2), score(w1, w2, wfd, pfd)) for w1, w2 in pfd]
    scored.sort(key=itemgetter(1), reverse=True)
    return map(itemgetter(0), scored)

def score(word1, word2, wfd, pfd, power=3):
    freq1 = wfd[word1]
    freq2 = wfd[word2]
    freq12 = pfd[(word1, word2)]
    return freq12 ** power / float(freq1 * freq2)

>>> for file in nltk.corpus.webtext.files():
...     words = [word.lower() for word in nltk.corpus.webtext.words(file) if len(word) > 2]
...     print file, [w1+' '+w2 for w1, w2 in collocations(words)[:15]]
overheard ['new york', 'teen boy', 'teen girl', 'you know', 'middle aged',
'flight attendant', 'puerto rican', 'last night', 'little boy', 'taco bell',
'statue liberty', 'bus driver', 'ice cream', 'don know', 'high school']
pirates ['jack sparrow', 'will turner', 'elizabeth swann', 'davy jones',
'flying dutchman', 'lord cutler', 'cutler beckett', 'black pearl', 'tia dalma',
'heh heh', 'edinburgh trader', 'port royal', 'bamboo pole', 'east india', 'jar dirt']
singles ['non smoker', 'would like', 'dining out', 'like meet', 'age open',
'sense humour', 'looking for', 'social drinker', 'down earth', 'long term',
'quiet nights', 'easy going', 'medium build', 'nights home', 'weekends away']
wine ['high toned', 'top ***', 'not rated', 'few years', 'medium weight',
'year two', 'cigar box', 'cote rotie', 'mixed feelings', 'demi sec',
'from half', 'brown sugar', 'bare ****', 'tightly wound', 'sous bois']
```

c) Now train your system using two distinct genres and experiment with generating text in the hybrid genre. Discuss your observations.

9. ◑ 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.

10. ◑ 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.

11. ◑ **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?

12. ◑ **Exploring text genres:** 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?

13. ◑ 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.

14. ★ **Authorship identification:** Reproduce some of the results of [Zhao & Zobel, 2007].

15. ★ **Gender-specific lexical choice:** Reproduce some of the results of `http://www.clintoneast.com/articles/words.php`

2.5 WordNet: An English Lexical Database

WordNet is a semantically-oriented dictionary of English, similar to a traditional thesaurus but with a richer structure. WordNet groups words into synonym sets, or **synsets**, each with its own definition and with links to other synsets. WordNet 3.0 data is distributed with NLTK, and includes 117,659 synsets.

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/`.

2.5.1 Senses and Synonyms

Consider the following sentence:

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

If we replace *motorcar* in (1) by *automobile*, the meaning of the sentence stays pretty much the same:

(2) 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**.

In order to look up the senses of a word, we need to pick a part of speech for the word. WordNet contains four dictionaries: `N` (nouns), `V` (verbs), `ADJ` (adjectives), and `ADV` (adverbs). To simplify our discussion, we will focus on the `N` dictionary here. Let's look up *motorcar* in the `N` dictionary.

```
>>> from nltk import wordnet
>>> car = wordnet.N['motorcar']
>>> car
motorcar (noun)
```

The variable `car` is now bound to a `Word` object. Words will often have more than sense, where each sense is represented by a synset. However, *motorcar* only has one sense in WordNet, as we can discover using `len()`. We can then find the synset (a set of lemmas), the words it contains, and a gloss.

```
>>> len(car)
1
>>> car[0]
{noun: car, auto, automobile, machine, motorcar}
>>> list(car[0])
['car', 'auto', 'automobile', 'machine', 'motorcar']
>>> car[0].gloss
'a motor vehicle with four wheels; usually propelled by an
internal combustion engine;
"he needs a car to get to work"'
```

The `wordnet` module also defines `Synsets`. Let's look at a word which is **polysemous**; that is, which has multiple synsets:

```
>>> poly = wordnet.N['pupil']
>>> for synset in poly:
...     print synset
{noun: student, pupil, educatee}
{noun: pupil}
{noun: schoolchild, school-age_child, pupil}
>>> poly[1].gloss
'the contractile aperture in the center of the iris of the eye;
resembles a large black dot'
```

2.5.2 The WordNet Hierarchy

WordNet synsets correspond to abstract concepts, which may or may not have corresponding words in English. These concepts are linked together in a hierarchy. Some are very general, such as *Entity*, *State*, *Event* — these are called **unique beginners**. Others, such as *gas guzzler* and *hatchback*, are much more specific. A small portion of a concept hierarchy is illustrated in Figure 2.4. The edges between nodes indicate the hypernym/hyponym relation; the dotted line at the top is intended to indicate that *artifact* is a non-immediate hypernym of *motorcar*.

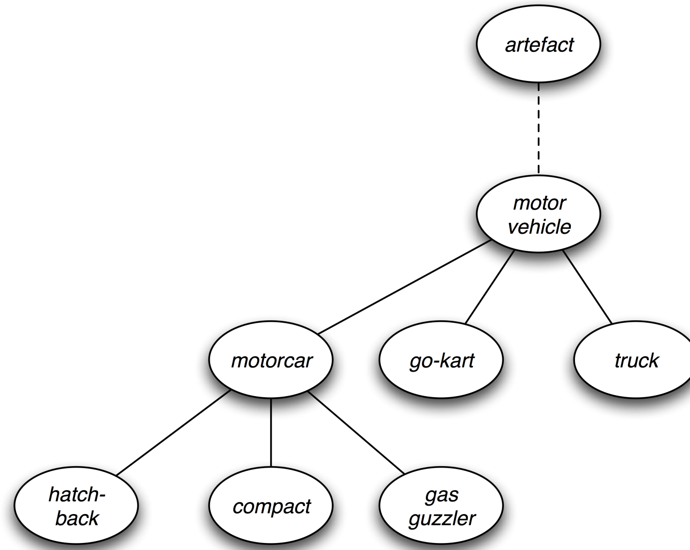

Figure 2.4: Fragment of WordNet Concept Hierarchy

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**. Here is one way to carry out this navigation:

```
>>> for concept in car[0][wordnet.HYPONYM][:10]:
...         print concept
{noun: ambulance}
{noun: beach_wagon, station_wagon, wagon, estate_car, beach_waggon, station_waggon,
{noun: bus, jalopy, heap}
{noun: cab, hack, taxi, taxicab}
{noun: compact, compact_car}
{noun: convertible}
{noun: coupe}
{noun: cruiser, police_cruiser, patrol_car, police_car, prowl_car, squad_car}
{noun: electric, electric_automobile, electric_car}
{noun: gas_guzzler}
```

We can also move up the hierarchy, by looking at broader concepts than *motorcar*, e.g. the immediate **hypernym** of a concept:

```
>>> car[0][wordnet.HYPERNYM]
[{noun: motor_vehicle, automotive_vehicle}]
```

We can also look for the hypernyms of hypernyms. In fact, from any synset we can trace (multiple) paths back to a unique beginner. Synsets have a method for doing this, called `tree()`, which produces a nested list structure.

```
>>> pprint.pprint(wordnet.N['car'][0].tree(wordnet.HYPERNYM))
[{noun: car, auto, automobile, machine, motorcar},
 [{noun: motor_vehicle, automotive_vehicle},
  [{noun: self-propelled_vehicle},
   [{noun: wheeled_vehicle},
    [{noun: vehicle},
     [{noun: conveyance, transport},
      [{noun: instrumentality, instrumentation},
       [{noun: artifact, artefact},
        [{noun: whole, unit},
         [{noun: object, physical_object},
          [{noun: physical_entity}, [{noun: entity}]]]]]]]]],
    [{noun: container},
     [{noun: instrumentality, instrumentation},
      [{noun: artifact, artefact},
       [{noun: whole, unit},
        [{noun: object, physical_object},
         [{noun: physical_entity}, [{noun: entity}]]]]]]]]]]
```

A related method `closure()` produces a flat version of this structure, with repeats eliminated. Both of these functions take an optional `depth` argument that permits us to limit the number of steps to take. (This is important when using unbounded relations like `SIMILAR`.) Table 2.5 lists the most important lexical relations supported by WordNet; see `dir(wordnet)` for a full list.

Hypernym	more general	*animal* is a hypernym of *dog*
Hyponym	more specific	*dog* is a hyponym of *animal*
Meronym	part of	*door* is a meronym of *house*
Holonym	has part	*house* is a holonym of *door*
Synonym	similar meaning	*car* is a synonym of *automobile*
Antonym	opposite meaning	*like* is an antonym of *dislike*
Entailment	necessary action	*step* is an entailment of *walk*

Table 2.5: Major WordNet Lexical Relations

Recall that we can iterate over the words of a synset, with `for word in synset`. We can also test if a word is in a dictionary, e.g. `if word in wordnet.V`. As our last task, let's put these together to find "animal words" that are used as verbs. Since there are a lot of these, we will cut this off at depth 4. Can you think of the animal and verb sense of each word?

```
>>> animals = wordnet.N['animal'][0].closure(wordnet.HYPONYM, depth=4)
>>> [word for synset in animals for word in synset if word in wordnet.V]
['pet', 'stunt', 'prey', 'quarry', 'game', 'mate', 'head', 'dog',
 'stray', 'dam', 'sire', 'steer', 'orphan', 'spat', 'sponge',
 'worm', 'grub', 'pooch', 'toy', 'queen', 'baby', 'pup', 'whelp',
 'cub', 'kit', 'kitten', 'foal', 'lamb', 'fawn', 'bird', 'grouse',
 'hound', 'bulldog', 'stud', 'hog', 'baby', 'fish', 'cock', 'parrot',
 'frog', 'beetle', 'bug', 'bug', 'queen', 'leech', 'snail', 'slug',
```

```
'clam', 'cockle', 'oyster', 'scallop', 'scollop', 'escallop', 'quail']
```

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

2.5.3 WordNet Similarity

We would expect that the semantic similarity of two concepts would correlate with the length of the path between them in WordNet. The `wordnet` package includes a variety of 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). A score of 1 represents identity, i.e., comparing a sense with itself will return 1.

```
>>> wordnet.N['poodle'][0].path_similarity(wordnet.N['dalmatian'][1])
0.33333333333333331
>>> wordnet.N['dog'][0].path_similarity(wordnet.N['cat'][0])
0.20000000000000001
>>> wordnet.V['run'][0].path_similarity(wordnet.V['walk'][0])
0.25
>>> wordnet.V['run'][0].path_similarity(wordnet.V['think'][0])
-1
```

Several other similarity measures are provided in `wordnet`: Leacock-Chodorow, Wu-Palmer, Resnik, Jiang-Conrath, and Lin. For a detailed comparison of various measures, see [Budanitsky & Hirst, 2006].

2.5.4 Exercises

1. ☼ Familiarize yourself with the WordNet interface, by reading the documentation available via `help(wordnet)`. Try out the text-based browser, `wordnet.browse()`.
2. ☼ 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`.

1. ☼ 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.
2. ◑ What is the branching factor of the noun hypernym hierarchy? (For all noun synsets that have hyponyms, how many do they have on average?)
3. ◑ 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`.
4. ◑ Write a program to score the similarity of two nouns as the depth of their first common hypernym.

5. ★ 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.

1. ★ 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.)

2.6 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.

2.7 Summary

- 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()`
- NLTK comes with many corpora, e.g. the Brown Corpus, `corpus.brown`.
- 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).
- 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.

2.8 Further Reading

For a more extended treatment of regular expressions, see 1. To learn about Unicode, see 1.

For more examples of processing words with NLTK, please see the guides at `http://nltk.org/doc/guides/tokenize.html`, `http://nltk.org/doc/guides/stem.html`, and `http://nltk.org/doc/guides/wordnet.html`. A guide on accessing NLTK corpora is available at: `http://nltk.org/doc/guides/corpus.html`. Chapters 2 and 3 of [Jurafsky & Martin, 2008] contain more advanced material on regular expressions and morphology.

About this document...

This chapter is a draft from *Natural Language Processing* [http://nltk.org/book.html], by Steven Bird, Ewan Klein and Edward Loper, Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [http://nltk.org/], Version 0.9.5, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [http://creativecommons.org/licenses/by-nc-nd/3.0/us/].

This document is