Chapter 3

Words: The Building Blocks of Language

3.1 Introduction

Language can be divided up into pieces of varying sizes, ranging from morphemes to paragraphs. In this chapter we will focus on words, a very important level for much work in NLP. Just what are words, and how should we represent them in a machine? These may seem like trivial questions, but it turns out that there are some important issues involved in defining and representing words.

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 normalisation; Wordnet; and a variety of useful programming tasks involving words

3.2 Tokens, Types and Texts

In Chapter 1, 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 for us:

```
>>> 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 two different answers, depending on whether we count duplicates or not. 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.

In effect, when we used `split()` above to turn a string into a list of words, our tokenization method was to say that any strings which are delimited by whitespace count as a word token. But this simple approach doesn't always lead to the results we want. Moreover, string-identity doesn't always give us a 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 more closely at what is contained in `set(words)`:

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

One point to note is that `'time'` and `'time.'` come out as distinct tokens, and of necessity, distinct types, since the trailing period has been bundled up with the rest of the word into a single token. We might also argue that although `'--'` is some kind of token, it isn't really a *word* token. Third, we would probably want to say that `'This'` and `'this'` are not distinct types, since capitalization should be ignored.

If we turn to languages other than English, segmenting words can be even more of a challenge. For example, in Chinese orthography, characters correspond to monosyllabic morphemes. Many morphemes are words in their own right, but words contain more than one morpheme. However, there is no visual representation of word boundaries in Chinese text. For example, 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, although the type/token distinction is a useful one, we cannot just say that two word tokens have the same type if they are the same string of characters — we need to take into consideration

a number of other factors in determining what counts as the same word. Moreover, we also need to be more 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 an impractical approach 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.

3.2.1 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 in 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('foo.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('.')
```

To read the contents of the file we can use lots of different methods. The following uses the read 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'
```

You will recall that the strange `'\n'` character on the end of the string is a **newline** character; this is equivalent to pressing *Enter* on a keyboard and starting a new line. .. There is also a `'\t'` character for representing tab. 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.

3.2.2 Extracting text from the Web

To read in a web page, 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 plain text, we can strip out the HTML markup, that is 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>'
>>> import re
>>> 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 two-fold. First, as already noted, the wildcard `'.'` matches any character other than `'\n'`, so in particular it will match `'>'` and `'<'`. Second, the `'*'` operator is 'greedy', in the sense that it matches as many characters as it can. In the example we just looked at, therefore, `'.*'` will return not the shortest match, namely `'title'`, but the longest match, `'title >BBC NEWS | News Front Page</title'`.

In order to get the results we want, we need to think about the task in a slightly different way. Our assumption is that after we have encountered a `'<'`, any character can occur within the tag except a `'>'`; once we find the latter, we know the tag is closed. Now, we have already seen how to match everything but α, for some character α; we use a negated range expression. In this case, the expression we need is `'[^<]'`: match everything except `'<'`. This range expression is then quantified with the `'*'` operator. In our revised example below, we use the improved regular expression, and we also normalise whitespace, replacing any sequence of one or more spaces, tabs or newlines (these are all matched by `'\s+'`) with a single space character.

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

You will probably find it useful to borrow the structure of this 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.

3.2.3 Extracting text from NLTK Corpora

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

```
>>> from nltk_lite.corpora import gutenberg
>>> gutenberg.items
['austen-emma', 'austen-persuasion', 'austen-sense', 'bible-kjv',
'blake-poems', 'blake-songs', 'chesterton-ball', 'chesterton-brown',
'chesterton-thursday', 'milton-paradise', 'shakespeare-caesar',
'shakespeare-hamlet', 'shakespeare-macbeth', 'whitman-leaves']
```

Next we iterate over the text content to find the number of word tokens:

```
>>> count = 0
>>> for word in gutenberg.raw('whitman-leaves'):
...     count += 1
>>> print count
154873
```

NLTK also includes the Brown Corpus, 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.

```
>>> from nltk_lite.corpora import brown
>>> brown.items
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'r']
```

We can extract individual sentences (as lists of words) from the corpus using the `extract()` function. This is called below with `0` as an argument, indicating that we want the first sentence of the corpus to be returned; `1` will return the second sentence, and so on. `brown.raw()` is an iterator which gives us the words without their part-of-speech tags.

```
>>> from nltk_lite.corpora import extract
>>> print extract(0, brown.raw())
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an',
'investigation', 'of', "Atlanta's", 'recent', 'primary', 'election',
'produced', '``', 'no', 'evidence', "''", 'that', 'any', 'irregularities',
'took', 'place', '.']
```

3.2.4 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.

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

3. ☼ Use the Brown corpus reader `brown.raw()` to access some sample text in two different genres.

4. ☼ 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?

5. ◑ Write a program to generate a table of token/type ratios, as we saw above. Include the full set of Brown Corpus genres. Use the dictionary `brown.item_name` to find out the genre of each section of the corpus. Which genre has the lowest diversity (greatest number of tokens per type)? Is this what you would have expected?

6. ◑ 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 questions 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?

7. ◑ 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 which improve the extraction of text from this web page.

8. ◑ 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?

9. ◑ Define a function `ghits()`, which 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.

10. ◑ Try running the various chatbots. 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.sourceforge.net/lite/nltk_lite/chat/`.

3.3 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 interword 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 expression to specify which character sequences should be treated as words. We will also consider important ways to normalize tokens.

3.3.1 Tokenization with Regular Expressions

The function `tokenize.regexp()` 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:

```
>>> from nltk_lite import tokenize
>>> text = '''Hello.  Isn't this fun?'''
>>> pattern = r'\w+|[^\w\s]+'
>>> list(tokenize.regexp(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 which 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 might want to improve this regular expression. For example, it currently breaks *$22.50* into four tokens; but we might want it to treat this as a single token. Similarly, we would want to treat *U.S.A.* as a single token. We can deal with these by adding further clauses to the tokenizer's regular expression. For readability we break it up and insert comments, and use the `re.VERBOSE` flag, so that Python knows to strip out the embedded whitespace and comments.

```
>>> import re
>>> text = 'That poster costs $22.40.'
>>> pattern = re.compile(r'''
...     \w+                # sequences of 'word' characters
...   | \$?\d+(\.\d+)?     # currency amounts, e.g. $12.50
...   | [\A\.]+            # abbreviations, e.g. U.S.A.
...   | [^\w\s]+           # sequences of punctuation
... ''', re.VERBOSE)
>>> list(tokenize.regexp(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 `tokenize.regexp()` function permits an optional boolean parameter `gaps`; when set to `True` the pattern is matched against the gaps. For example, here is how `tokenize.whitespace()` is defined:

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

Of course, we can also invoke the `whitespace()` method directly:

```
>>> text = 'That poster costs $22.40.'
>>> list(tokenize.whitespace(text))
['That', 'poster', 'costs', '$22.40.']
```

3.3.2 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 which 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.

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*.

As a first approximation to discovering the distribution of a word, we can look at all the bigrams it occurs in. Let's consider all bigrams from the Brown Corpus which 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 infer from this that if we come across *saw* in the context *often* __, 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 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 contast, *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 signalled 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 grammatical forms into their lemmas — would yield the following picture of the distribution of *often*.

```
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 which requires a mixture of rules for regular inflections and table look-up for irregular morphological patterns. Within NLTK, a simpler approach is offered by the **Porter Stemmer** and the **Lancaster Stemmer**, which strip inflectional suffixes from words, collapsing the different forms of APPEAR and CALL. Given the simple nature of the stemming algorithms, you may not be surprised to learn that this stemmer does not attempt to identify *were* as a form of the lexeme BE.

```
>>> from nltk_lite import stem
>>> stemmer = stem.Porter()
>>> verbs = ['appears', 'appear', 'appeared', 'calling', 'called']
>>> stems = []
>>> for verb in verbs:
...     stemmed_verb = stemmer.stem(verb)
...     if stemmed_verb not in stems:
...         stems.append(stemmed_verb)
>>> stems
['appear', 'call']
```

Lemmatization and stemming can be regarded as special cases of **normalization**. They identify a canonical representative for a group of related word forms. By its nature, normalization collapses distinctions. An example is case normalization, where all variants are mapped into a single format. What counts as the normalized form will vary according to context. Often, we convert everything into lower case, so that words which were capitalized by virtue of being sentence-initial are treated the same as those which occur elsewhere in the sentence. The Python string method `lower()` will accomplish this for us:

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

We need to be careful, however; case normalization will also collapse the *New* of *New York* with the *new* of *my new car*.

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 *not*.

3.3.3 Aside: List Comprehensions

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']
```

Here we rewrite the loop for identifying verb stems, from the previous section:

```
>>> [stemmer.stem(verb) for verb in verbs]
['appear', 'appear', 'appear', 'call', 'call']
>>> 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. $\{(x,y) \mid x^2 + y^2 = 1\}$. Just as this set definition incorporates a constraint, list comprehensions can constrain the items they include. In the next example we remove all determiners 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, 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 length.

```
>>> sent = extract(0, brown.raw())
>>> [(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'), ('.', '.')]
```

3.3.4 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 `tokenize.regexp()` to create a tokenizer which tokenizes the various kinds of punctuation in this text. Use a single regular expression, with inline comments using the `re.VERBOSE` flag.

b) Use `tokenize.regexp()` to create a tokenizer which 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.

4. ◑ Readability measures are used to score the reading difficulty of a text, for the purposes of selecting texts of appropriate difficulty for language learners. For example, the Automated Readability Index (ARI) of a text is defined to be: `4.71 *` μ_w `+ 0.5 *` μ_s `- 21.43`, where μ_w is the mean word length (in letters), and where μ_s is the mean sentence length (in words). With the help of your word and sentence tokenizers, compute the ARI scores for a collection of texts.

5. ★ 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(vowels)
>>> sorted(vsequences)
['aiuio', 'eaiou', 'eouio', 'euoia', 'oauaio', 'uiieioa']
```

1. ★ **Sentence tokenizers:** Develop a sentence tokenizer. Test it on the Brown Corpus, which has been grouped into sentences.

3.4 Lexical Resources (INCOMPLETE)

[This section will contain a discussion of lexical resources, focusing on Wordnet, but also including the `cmudict` and `timit` corpus readers.]

3.4.1 Pronunciation Dictionary

Here we access the pronunciation of words...

```
>>> from nltk_lite.corpora import cmudict
>>> from string import join
>>> for word, num, pron in cmudict.raw():
...     if pron[-4:] == ('N', 'IH0', 'K', 'S'):
...         print word.lower(),
atlantic's audiotronics avionics beatniks calisthenics centronics
chetniks clinic's clinics conics cynics diasonics dominic's
ebonics electronics electronics' endotronics endotronics' enix
environics ethnics eugenics fibronics flextronics harmonics
hispanics histrionics identics ionics kibbutzniks lasersonics
lumonics mannix mechanics mechanics' microelectronics minix minnix
mnemonics mnemonics molonicks mullenix mullenix mullinix mulnix
munich's nucleonics onyx panic's panics penix pennix personics
phenix philharmonic's phoenix phonics photronics pinnix
plantronics pyrotechnics refuseniks resnick's respironics sconnix
siliconix skolniks sonics sputniks technics tectonics tektronix
telectronics telephonics tonics unix vinick's vinnick's vitronics
```

3.4.2 WordNet Semantic Network

> **Note**
>
> Before using Wordnet it must be installed on your machine, along with NLTK version 0.8 Please see the instructions on the NLTK website. Help on the `wordnet` interface is available using `help(wordnet)`.

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. More technically, we say that they are **synonyms**.

Wordnet is a semantically-oriented dictionary which will allow us to find the set of synonyms — or **synset** — for any word. However, 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_lite.wordnet import *
>>> car = 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 by checking the length of `car`. 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}
>>> [word for word in 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 = N['pupil']
>>> for synset in poly:
...     print synset
{noun: student, pupil, educatee}
{noun: schoolchild, school-age child, pupil}
{noun: pupil}
>>> poly[2].gloss
'the contractile aperture in the center of the iris of the eye;
resembles a large black dot'
```

We can think of synsets as being concrete manifestations of the concepts that are **lexicalized** by words in a particular language. In principle, we can postulate concepts that have no lexical realization in English. Wordnet builds on a tradition which claims that concepts are linked together in a hierarchy. Some concepts are very general, such as *Entity*, *State*, *Event* — these are called **unique beginners** in Wordnet. Others, such as *gas guzzler* and *hatchback*, are much more specific. A small portion of a concept hierarchy is illustrated in Figure 3.1. The edges between nodes indicate the hypernym/hyponym relation; the dotted line at the top is intended to indicate that *artefact* is non-immediate hypernym of *motorcar*.

Wordnet has been designed to make it easy to navigate between concepts. For example, given a concept like *motorcar*, we can look at the concepts which are more specific; these are usually called **hyponyms**. Here is one way to carry out this navigation:

```
>>> for concept in car[0][HYPONYM][:10]:
...         print concept
{noun: ambulance}
{noun: beach wagon, station wagon, wagon, estate car, beach waggon, station waggon, 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*. The `wordnet` package offers a shortcut for finding the immediate **hypernym** of a concept:

```
>>> car[0][HYPERNYM]
[{noun: motor vehicle, automotive vehicle}]
```

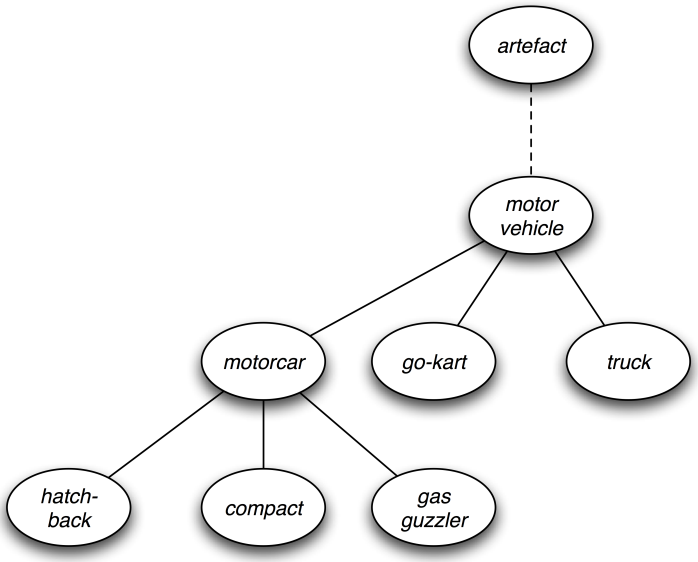

Figure 3.1: Fragment of Wordnet Concept Hierarchy

Of course, we can also look for the hypernyms of hypernyms; from any synset we can trace (multiple) paths back to a unique beginner. Synsets have a method `tree()` which produces a nested list structure.

```
>>> from pprint import pprint
>>> pprint(N['car'][0].tree(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 any repeats eliminated. Both of these functions take an optional `depth` argument which permit us to limit the number of steps to take, which is important when using unbounded relations like `SIMILAR`. Table 3.1 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 hypernym 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 3.1: 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. `of word in V`. 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.

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

3.4.3 WordNet Similarity

It is often useful to be able to tell whether two lexical concepts are **semantically related**. For example, in order to check whether a particular instance of the word *bank* means *financial institution*, we can count the number of nearby words that are semantically related to this sense. Using WordNet, we can investigate whether semantic relatedness can be expressed in terms of the graph structure of the concept hierarchy. More specifically, we would expect that the semantic relatedness of two concepts correlates with the length of the path between them. The `wordnet` package includes a variety of measures which 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.

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

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

3.4.4 Exercises

1. ☼ Familiarize yourself with the Wordnet interface, by reading the documentation available via `help(wordnet)`.

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

3. ◑ Write a program to score the similarity of two nouns as the depth of their first common hypernym.

4. ★ 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.

3.5 Simple Statistics with Tokens

3.5.1 Example: Stylistics

So far, we've seen how to count the number of tokens or types in a document. But it's much more interesting to look at *which* tokens or types appear in a document. We can use a Python dictionary to count the number of occurrences of each word type in a document:

```
>>> counts = {}
>>> for word in text.split():
...     if word not in counts:
...         counts[word] = 0
...     counts[word] += 1
```

The first statement, `counts = {}`, initializes the dictionary, while the next four lines successively add entries to it and increment the count each time we encounter a new token of a given type. To view the contents of the dictionary, we can iterate over its keys and print each entry (here just for the first 10 entries):

```
>>> for word in sorted(counts)[:10]:
...     print counts[word], word
1 $1.1
2 $130
1 $36
1 $45
1 $490
1 $5
1 $62.625,
```

```
1 $620
1 $63
2 $7
```

We can also print the number of times that a specific word we're interested in appeared:

```
>>> print counts['might']
3
```

Applying this same approach to document collections that are categorized by genre, we can learn something about the patterns of word usage in those genres. For example, Table 3.2 was constructed by counting the number of times various modal words appear in different genres in the Brown 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 3.2: 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 3.3. (Genres with lower diversity have a higher number of tokens per type.)

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 3.3: Word Types and Tokens in Brown Corpus, by Genre

We can carry out a variety of interesting explorations simply by counting words. In fact, the field of *Corpus Linguistics* focuses almost exclusively on creating and interpreting such tables of word counts. So far, our method for identifying word tokens has been a little primitive, and we have not been able to separate punctuation from the words. We will take up this issue in the next section.

3.5.2 Example: 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 3.2: Lexical Dispersion

As you can see, *Elinor* and *Marianne* appear rather uniformly throughout the text, while *Edward* and *Willoughby* tend to appear separately. Here is the program that generated the above plot. [NB. Requires NLTK-Lite 0.6.7].

```
>>> from nltk_lite.corpora import gutenberg
>>> from nltk_lite.draw import dispersion
>>> words = ['Elinor', 'Marianne', 'Edward', 'Willoughby']
>>> dispersion.plot(gutenberg.raw('austen-sense'), words)
```

3.5.3 Frequency Distributions

We can do more sophisticated counting using *frequency distributions*. Abstractly, a frequency distribution is a record of the number of times each *outcome* of an *experiment* has occurred. For instance, a frequency distribution could be used to record the frequency of each word in a document (where the "experiment" is examining a word, and the "outcome" is the word's type). Frequency distributions are generally created by repeatedly running an experiment, and incrementing the count for a sample every time it is an outcome of the experiment. The following program produces a frequency distribution that records how often each word type occurs in a text. It increments a separate counter for each word, and prints the most frequently occurring word:

```
>>> from nltk_lite.probability import FreqDist
>>> from nltk_lite.corpora import genesis
>>> fd = FreqDist()
>>> for token in genesis.raw():
...     fd.inc(token)
>>> fd.max()
'the'
```

Once we construct a frequency distribution that records the outcomes of an experiment, we can use it to examine a number of interesting properties of the experiment. Some of these properties are summarized in Table 3.4.

Name	Sample	Description
Count	`fd.count('the')`	number of times a given sample occurred
Frequency	`fd.freq('the')`	frequency of a given sample
N	`fd.N()`	number of samples
Samples	`fd.samples()`	list of distinct samples recorded

Name	Sample	Description
Max	`fd.max()`	sample with the greatest number of outcomes

Table 3.4: Frequency Distribution Module

We can also use a `FreqDist` 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 length_dist(text):
...     fd = FreqDist()                          # initialize frequency distribution
...     for token in genesis.raw(text):          # for each token
...         fd.inc(len(token))                   # found a word with this length
...     for i in range(1,15):                    # for each length from 1 to 14
...         print "%2d" % int(100*fd.freq(i)), # print the percentage of words with this le
...     print
```

Now we can call `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.

```
>>> length_dist('english-kjv')
 2 14 28 21 13  7  5  2  2  0  0  0  0  0
>>> length_dist('finnish')
 0  9  6 10 16 16 12  9  6  3  2  2  1  0
```

3.5.4 Conditional Frequency Distributions

A *condition* specifies the context in which an experiment is performed. Often, we are interested in the effect that conditions have on the outcome for an experiment. A *conditional frequency distribution* is a collection of frequency distributions for the same experiment, run under different conditions. For example, we might want to examine how the distribution of a word's length (the outcome) is affected by the word's initial letter (the condition).

```
>>> from nltk_lite.corpora import genesis
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
>>> for text in genesis.items:
...     for word in genesis.raw(text):
...         cfdist[word[0]].inc(len(word))
```

To plot the results, we construct a list of points, where the x coordinate is the word length, and the y coordinate is the frequency with which that word length is used:

```
>>> for cond in cfdist.conditions():
...     wordlens = cfdist[cond].samples()
...     wordlens.sort()
...     points = [(i, cfdist[cond].freq(i)) for i in wordlens]
```

We can plot these points using the `Plot` function defined in `nltk_lite.draw.plot`, as follows:

```
>>> Plot(points).mainloop()
```

3.5.5 Predicting the Next Word

Conditional frequency distributions are often used for prediction. *Prediction* is the problem of deciding a likely outcome for a given run of an experiment. The decision of which outcome to predict is usually based on the context in which the experiment is performed. For example, we might try to predict a word's text (outcome), based on the text of the word that it follows (context).

To predict the outcomes of an experiment, we first examine a representative *training corpus*, where the context and outcome for each run of the experiment are known. When presented with a new run of the experiment, we simply choose the outcome that occurred most frequently for the experiment's context.

We can use a `ConditionalFreqDist` to find the most frequent occurrence for each context. First, we record each outcome in the training corpus, using the context that the experiment was run under as the condition. Then, we can access the frequency distribution for a given context with the indexing operator, and use the `max()` method to find the most likely outcome.

We will now use a `ConditionalFreqDist` to predict the most likely next word in a text. To begin, we load a corpus from a text file, and create an empty `ConditionalFreqDist`:

```
>>> from nltk_lite.corpora import genesis
>>> from nltk_lite.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()
```

We then examine each token in the corpus, and increment the appropriate sample's count. We use the variable `prev` to record the previous word.

```
>>> prev = None
>>> for word in genesis.raw():
...     cfdist[prev].inc(word)
...     prev = word
```

> **Note**
>
> Sometimes the context for an experiment is unavailable, or does not exist. For example, the first token in a text does not follow any word. In these cases, we must decide what context to use. For this example, we use `None` as the context for the first token. Another option would be to discard the first token.

Once we have constructed a conditional frequency distribution for the training corpus, we can use it to find the most likely word for any given context. For example, taking the word *living* as our context, we can inspect all the words that occurred in that context.

```
>>> word = 'living'
>>> cfdist[word].samples()
['creature,', 'substance', 'soul.', 'thing', 'thing,', 'creature']
```

We can set up a simple loop to generate text: we set an initial context, picking the most likely token in that context as our next word, and then using that word as our new context:

```
>>> word = 'living'
>>> for i in range(20):
...     print word,
...     word = cfdist[word].max()
living creature that he said, I will not be a wife of the land
of the land of the land
```

This simple approach to text generation tends to get stuck in loops, as demonstrated by the text generated above. A more advanced approach would be to randomly choose each word, with more frequent words chosen more often.

3.5.6 Exercises

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

2. ☼ Use the `Plot` function defined in `nltk_lite.draw.plot` plot word-initial character against word length, as discussed in this section.

3. ◑ 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.

4. ◑ **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 the `nltk_lite.draw.plot` module. Do you confirm Zipf's law? (Hint: it helps to use logarithmic axes, by including `scale='log'` as a second argument to `Plot()`). 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?

5. ◑ **Predicting the next word**: The word prediction program we saw in this chapter quickly gets stuck in a cycle. Modify the program to choose the next word randomly, from a list of the *n* most likely words in the given context. (Hint: store the *n* most likely words in a list `lwords` then randomly choose a word from the list using `random.choice()`.)

   a) Select a particular genre, such as a section of the Brown Corpus, or a genesis translation, or one of the Gutenberg texts. Train your system 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.

   b) Try the same approach with different genres, and with different amounts of training data. What do you observe?

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

6. ◑ **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?

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

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

3.6 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. Other kinds of tokenization, such as sentence tokenization, are left for the exercises. 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.

3.7 Further Reading