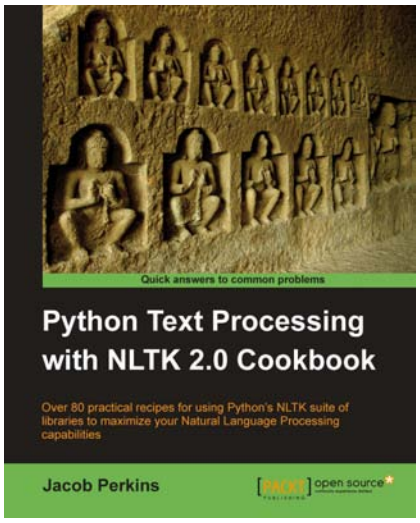
Python Text Processing with NLTK 2.0 Cookbook

Jacob Perkins


Quick answers to common problems
Python Text Processing with NLTK 2.0 Cookbook
Over 80 practical recipes for using Python's NLTK suite of libraries to maximize your Natural Language Processing capabilities
Jacob Perkins
[PACKT] open source

Chapter No.3
"Creating Custom Corpora"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Creating Custom Corpora"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Jacob Perkins has been an avid user of open source software since high school, when he first built his own computer and didn't want to pay for Windows. At one point he had five operating systems installed, including Red Hat Linux, OpenBSD, and BeOS.

While at Washington University in St. Louis, Jacob took classes in Spanish and poetry writing, and worked on an independent study project that eventually became his Master's project: WUGLE—a GUI for manipulating logical expressions. In his free time, he wrote the Gnome2 version of Seahorse (a GUI for encryption and key management), which has since been translated into over a dozen languages and is included in the default Gnome distribution.

After receiving his MS in Computer Science, Jacob tried to start a web development studio with some friends, but since no one knew anything about web development, it didn't work out as planned. Once he'd actually learned about web development, he went off and co-founded another company called Weotta, which sparked his interest in Machine Learning and Natural Language Processing.

Jacob is currently the CTO/Chief Hacker for Weotta and blogs about what he's learned along the way at http://streamhacker.com/. He is also applying this knowledge to produce text processing APIs and demos at http://text-processing.com/. This book is a synthesis of his knowledge on processing text using Python, NLTK, and more.

Thanks to my parents for all their support, even when they don't understand what I'm doing; Grant for sparking my interest in Natural Language Processing; Les for inspiring me to program when I had no desire to; Arnie for all the algorithm discussions; and the whole Wernick family for feeding me such good food whenever I come over.

Python Text Processing with NLTK 2.0 Cookbook

Natural Language Processing is used everywhere—in search engines, spell checkers, mobile phones, computer games, and even in your washing machine. Python's Natural Language Toolkit (NLTK) suite of libraries has rapidly emerged as one of the most efficient tools for Natural Language Processing. You want to employ nothing less than the best techniques in Natural Language Processing—and this book is your answer.

Python Text Processing with NLTK 2.0 Cookbook is your handy and illustrative guide, which will walk you through all the Natural Language Processing techniques in a step-by-step manner. It will demystify the advanced features of text analysis and text mining using the comprehensive NLTK suite.

This book cuts short the preamble and lets you dive right into the science of text processing with a practical hands-on approach.

Get started off with learning tokenization of text. Receive an overview of WordNet and how to use it. Learn the basics as well as advanced features of stemming and lemmatization. Discover various ways to replace words with simpler and more common (read: more searched) variants. Create your own corpora and learn to create custom corpus readers for data stored in MongoDB. Use and manipulate POS taggers. Transform and normalize parsed chunks to produce a canonical form without changing their meaning. Dig into feature extraction and text classification. Learn how to easily handle huge amounts of data without any loss in efficiency or speed.

This book will teach you all that and beyond, in a hands-on learn-by-doing manner. Make yourself an expert in using the NLTK for Natural Language Processing with t
his handy companion.

What This Book Covers

Chapter 1, Tokenizing Text and WordNet Basics, covers the basics of tokenizing text and using WordNet.

Chapter 2, Replacing and Correcting Words, discusses various word replacement and correction techniques. The recipes cover the gamut of linguistic compression, spelling correction, and text normalization.

Chapter 3, Creating Custom Corpora, covers how to use corpus readers and create custom corpora. At the same time, it explains how to use the existing corpus data that comes with NLTK.

Chapter 4, Part-of-Speech Tagging, explains the process of converting a sentence, in the form of a list of words, into a list of tuples. It also explains taggers, which are trainable.

Chapter 5, Extracting Chunks, explains the process of extracting short phrases from a part-of-speech tagged sentence. It uses Penn Treebank corpus for basic training and testing chunk extraction, and the CoNLL 2000 corpus as it has a simpler and more flexible format that supports multiple chunk types.

Chapter 6, Transforming Chunks and Trees, shows you how to do various transforms on both chunks and trees. The functions detailed in these recipes modify data, as opposed to learning from it.

Chapter 7, Text Classification, describes a way to categorize documents or pieces of text and, by examining the word usage in a piece of text, classifiers decide what class label should be assigned to it.

Chapter 8, Distributed Processing and Handling Large Datasets, discusses how to use execnet to do parallel and distributed processing with NLTK. It also explains how to use the Redis data structure server/database to store frequency distributions.

Chapter 9, Parsing Specific Data, covers parsing specific kinds of data, focusing primarily on dates, times, and HTML.

Appendix, Penn Treebank Part-of-Speech Tags, lists a table of all the part-of-speech tags that occur in the treebank corpus distributed with NLTK.

3
Creating Custom Corpora

In this chapter, we will cover:

- Setting up a custom corpus
- Creating a word list corpus
- Creating a part-of-speech tagged word corpus
- Creating a chunked phrase corpus
- Creating a categorized text corpus
- Creating a categorized chunk corpus reader
- Lazy corpus loading
- Creating a custom corpus view
- Creating a MongoDB backed corpus reader
- Corpus editing with file locking

Introduction

In this chapter, we'll cover how to use corpus readers and create custom corpora. At the same time, you'll learn how to use the existing corpus data that comes with NLTK. This information is essential for future chapters when we'll need to access the corpora as training data. We'll also cover creating custom corpus readers, which can be used when your corpus is not in a file format that NLTK already recognizes, or if your corpus is not in files at all, but instead is located in a database such as MongoDB.

Setting up a custom corpus

A **corpus** is a collection of text documents, and **corpora** is the plural of corpus. So a *custom corpus* is really just a bunch of text files in a directory, often alongside many other directories of text files.

Getting ready

You should already have the NLTK data package installed, following the instructions at `http://www.nltk.org/data`. We'll assume that the data is installed to `C:\nltk_data` on Windows, and `/usr/share/nltk_data` on Linux, Unix, or Mac OS X.

How to do it...

NLTK defines a list of data directories, or **paths**, in `nltk.data.path`. Our custom corpora must be within one of these paths so it can be found by NLTK. So as not to conflict with the official data package, we'll create a custom `nltk_data` directory in our home directory. Here's some Python code to create this directory and verify that it is in the list of known paths specified by `nltk.data.path`:

```
>>> import os, os.path
>>> path = os.path.expanduser('~/nltk_data')
>>> if not os.path.exists(path):
...     os.mkdir(path)
>>> os.path.exists(path)
True
>>> import nltk.data
>>> path in nltk.data.path
True
```

If the last line, `path in nltk.data.path`, is `True`, then you should now have a `nltk_data` directory in your home directory. The path should be `%UserProfile%\nltk_data` on Windows, or `~/nltk_data` on Unix, Linux, or Mac OS X. For simplicity, I'll refer to the directory as `~/nltk_data`.

If the last line does not return `True`, try creating the `nltk_data` directory manually in your home directory, then verify that the absolute path is in `nltk.data.path`. It's essential to ensure that this directory exists and is in `nltk.data.path` before continuing. Once you have your `nltk_data` directory, the convention is that corpora reside in a `corpora` subdirectory. Create this `corpora` directory within the `nltk_data` directory, so that the path is `~/nltk_data/corpora`. Finally, we'll create a subdirectory in `corpora` to hold our custom corpus. Let's call it `cookbook`, giving us the full path of `~/nltk_data/corpora/cookbook`.

Now we can create a simple *word list* file and make sure it loads. In *Chapter 2*, *Replacing and Correcting Words*, *Spelling correction with Enchant* recipe, we created a word list file called `mywords.txt`. Put this file into `~/nltk_data/corpora/cookbook/`. Now we can use `nltk.data.load()` to load the file.

```
>>> import nltk.data
>>> nltk.data.load('corpora/cookbook/mywords.txt', format='raw')
'nltk\n'
```

[We need to specify `format='raw'` since `nltk.data.load()` doesn't know how to interpret `.txt` files. As we'll see, it does know how to interpret a number of other file formats.]

How it works...

The `nltk.data.load()` function recognizes a number of formats, such as `'raw'`, `'pickle'`, and `'yaml'`. If no format is specified, then it tries to guess the format based on the file's extension. In the previous case, we have a `.txt` file, which is not a recognized extension, so we have to specify the `'raw'` format. But if we used a file that ended in `.yaml`, then we would not need to specify the format.

Filenames passed in to `nltk.data.load()` can be *absolute* or *relative* paths. Relative paths must be relative to one of the paths specified in `nltk.data.path`. The file is found using `nltk.data.find(path)`, which searches all known paths combined with the relative path. Absolute paths do not require a search, and are used as is.

There's more...

For most corpora access, you won't actually need to use `nltk.data.load`, as that will be handled by the `CorpusReader` classes covered in the following recipes. But it's a good function to be familiar with for loading `.pickle` files and `.yaml` files, plus it introduces the idea of putting all of your data files into a path known by NLTK.

Loading a YAML file

If you put the `synonyms.yaml` file from the *Chapter 2*, *Replacing and Correcting Words*, *Replacing synonyms* recipe, into `~/nltk_data/corpora/cookbook` (next to `mywords.txt`), you can use `nltk.data.load()` to load it without specifying a format.

```
>>> import nltk.data
>>> nltk.data.load('corpora/cookbook/synonyms.yaml')
{'bday': 'birthday'}
```

This assumes that PyYAML is installed. If not, you can find download and installation instructions at `http://pyyaml.org/wiki/PyYAML`.

See also

In the next recipes, we'll cover various corpus readers, and then in the *Lazy corpus loading* recipe, we'll use the `LazyCorpusLoader`, which expects corpus data to be in a `corpora` subdirectory of one of the paths specified by `nltk.data.path`.

Creating a word list corpus

The `WordListCorpusReader` is one of the simplest `CorpusReader` classes. It provides access to a file containing a list of words, one word per line. In fact, you've already used it when we used the `stopwords` corpus in the *Filtering stopwords in a tokenized sentence* and *Discovering word collocations* recipes in *Chapter 1*, *Tokenizing Text and WordNet Basics*.

Getting ready

We need to start by creating a word list file. This could be a single column CSV file, or just a normal text file with one word per line. Let's create a file named `wordlist` that looks like this:

```
nltk
corpus
corpora
wordnet
```

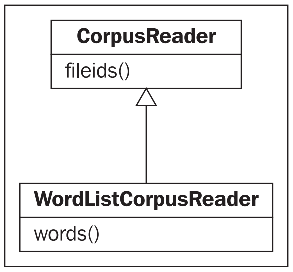
How to do it...

Now we can instantiate a `WordListCorpusReader` that will produce a list of words from our file. It takes two arguments: the directory path containing the files, and a list of filenames. If you open the Python console in the same directory as the files, then `'.'` can be used as the directory path. Otherwise, you must use a directory path such as: `'nltk_data/corpora/cookbook'`.

```
>>> from nltk.corpus.reader import WordListCorpusReader
>>> reader = WordListCorpusReader('.', ['wordlist'])
>>> reader.words()
['nltk', 'corpus', 'corpora', 'wordnet']
>>> reader.fileids()
['wordlist']
```

How it works...

`WordListCorpusReader` inherits from `CorpusReader`, which is a common base class for all corpus readers. `CorpusReader` does all the work of identifying which files to read, while `WordListCorpus` reads the files and tokenizes each line to produce a list of words. Here's an inheritance diagram:



When you call the `words()` function, it calls `nltk.tokenize.line_tokenize()` on the raw file data, which you can access using the `raw()` function.

```
>>> reader.raw()
'nltk\ncorpus\ncorpora\nwordnet\n'
>>> from nltk.tokenize import line_tokenize
>>> line_tokenize(reader.raw())
['nltk', 'corpus', 'corpora', 'wordnet']
```

There's more...

The `stopwords` corpus is a good example of a multi-file `WordListCorpusReader`. In *Chapter 1*, *Tokenizing Text and WordNet Basics*, in the *Filtering stopwords in a tokenized sentence* recipe, we saw that it had one word list file for each language, and you could access the words for that language by calling `stopwords.words(fileid)`. If you want to create your own multi-file word list corpus, this is a great example to follow.

Names corpus

Another word list corpus that comes with NLTK is the `names` corpus. It contains two files: `female.txt` and `male.txt`, each containing a list of a few thousand common first names organized by gender.

```
>>> from nltk.corpus import names
>>> names.fileids()
['female.txt', 'male.txt']
>>> len(names.words('female.txt'))
5001
```

```
>>> len(names.words('male.txt'))
2943
```

English words

NLTK also comes with a large list of English words. There's one file with 850 `basic` words, and another list with over 200,000 known English words.

```
>>> from nltk.corpus import words
>>> words.fileids()
['en', 'en-basic']
>>> len(words.words('en-basic'))
850
>>> len(words.words('en'))
234936
```

See also

In *Chapter 1*, *Tokenizing Text and WordNet Basics*, the *Filtering stopwords in a tokenized sentence* recipe, has more details on using the `stopwords` corpus. In the following recipes, we'll cover more advanced corpus file formats and corpus reader classes.

Creating a part-of-speech tagged word corpus

Part-of-speech tagging is the process of identifying the part-of-speech tag for a word. Most of the time, a *tagger* must first be trained on a *training corpus*. How to train and use a tagger is covered in detail in *Chapter 4*, *Part-of-Speech Tagging*, but first we must know how to create and use a training corpus of part-of-speech tagged words.

Getting ready

The simplest format for a tagged corpus is of the form "word/tag". Following is an excerpt from the `brown` corpus:

```
The/at-tl expense/nn and/cc time/nn involved/vbn are/ber astronomical/
jj ./.
```

Each word has a *tag* denoting its part-of-speech. For example, `nn` refers to a noun, while a tag that starts with `vb` is a verb.

How to do it...

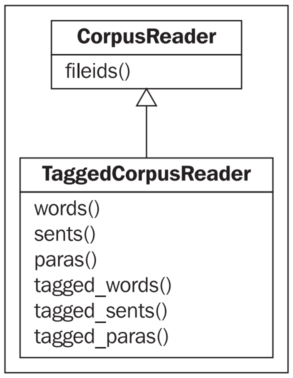
If you were to put the previous excerpt into a file called `brown.pos`, you could then create a `TaggedCorpusReader` and do the following:

```
>>> from nltk.corpus.reader import TaggedCorpusReader
>>> reader = TaggedCorpusReader('.', r'.*\.pos')
>>> reader.words()
['The', 'expense', 'and', 'time', 'involved', 'are', ...]
>>> reader.tagged_words()
[('The', 'AT-TL'), ('expense', 'NN'), ('and', 'CC'), …]
>>> reader.sents()
[['The', 'expense', 'and', 'time', 'involved', 'are', 'astronomical',
'.']]
>>> reader.tagged_sents()
[[('The', 'AT-TL'), ('expense', 'NN'), ('and', 'CC'), ('time', 'NN'),
('involved', 'VBN'), ('are', 'BER'), ('astronomical', 'JJ'), ('.',
'.')]]
>>> reader.paras()
[[['The', 'expense', 'and', 'time', 'involved', 'are', 'astronomical',
'.']]]
>>> reader.tagged_paras()
[[[('The', 'AT-TL'), ('expense', 'NN'), ('and', 'CC'), ('time', 'NN'),
('involved', 'VBN'), ('are', 'BER'), ('astronomical', 'JJ'), ('.',
'.')]]]
```

How it works...

This time, instead of naming the file explicitly, we use a regular expression, `r'.*\.pos'`, to match all files whose name ends with `.pos`. We could have done the same thing as we did with the `WordListCorpusReader`, and pass `['brown.pos']` as the second argument, but this way you can see how to include multiple files in a corpus without naming each one explicitly.

TaggedCorpusReader provides a number of methods for extracting text from a corpus. First, you can get a list of all words, or a list of tagged tokens. A **tagged token** is simply a tuple of (word, tag). Next, you can get a list of every sentence, and also every tagged sentence, where the sentence is itself a list of words or tagged tokens. Finally, you can get a list of paragraphs, where each paragraph is a list of sentences, and each sentence is a list of words or tagged tokens. Here's an inheritance diagram listing all the major methods:



There's more...

The functions demonstrated in the previous diagram all depend on *tokenizers* for splitting the text. TaggedCorpusReader tries to have good defaults, but you can customize them by passing in your own tokenizers at initialization time.

Customizing the word tokenizer

The default word tokenizer is an instance of nltk.tokenize.WhitespaceTokenizer. If you want to use a different tokenizer, you can pass that in as word_tokenizer.

```
>>> from nltk.tokenize import SpaceTokenizer
>>> reader = TaggedCorpusReader('.', r'.*\.pos', word_
tokenizer=SpaceTokenizer())
>>> reader.words()
['The', 'expense', 'and', 'time', 'involved', 'are', ...]
```

Customizing the sentence tokenizer

The default sentence tokenizer is an instance of `nltk.tokenize.RegexpTokenize` with `'\n'` to identify the gaps. It assumes that each sentence is on a line all by itself, and individual sentences do not have line breaks. To customize this, you can pass in your own tokenizer as `sent_tokenizer`.

```
>>> from nltk.tokenize import LineTokenizer
>>> reader = TaggedCorpusReader('.', r'.*\.pos', sent_
tokenizer=LineTokenizer())
>>> reader.sents()
[['The', 'expense', 'and', 'time', 'involved', 'are', 'astronomical',
'.']]
```

Customizing the paragraph block reader

Paragraphs are assumed to be split by blank lines. This is done with the default `para_block_reader`, which is `nltk.corpus.reader.util.read_blankline_block`. There are a number of other block reader functions in `nltk.corpus.reader.util`, whose purpose is to read blocks of text from a *stream*. Their usage will be covered in more detail in the later recipe, *Creating a custom corpus view*, where we'll create a custom corpus reader.

Customizing the tag separator

If you don't want to use `'/'` as the word/tag separator, you can pass an alternative string to `TaggedCorpusReader` for `sep`. The default is `sep='/'`, but if you want to split words and tags with `'|'`, such as 'word|tag', then you should pass in `sep='|'`.

Simplifying tags with a tag mapping function

If you'd like to somehow transform the part-of-speech tags, you can pass in a `tag_mapping_function` at initialization, then call one of the `tagged_*` functions with `simplify_tags=True`. Here's an example where we lowercase each tag:

```
>>> reader = TaggedCorpusReader('.', r'.*\.pos', tag_mapping_
function=lambda t: t.lower())
>>> reader.tagged_words(simplify_tags=True)
[('The', 'at-tl'), ('expense', 'nn'), ('and', 'cc'), …]
```

Calling `tagged_words()` without `simplify_tags=True` would produce the same result as if you did not pass in a `tag_mapping_function`.

There are also a number of tag simplification functions defined in `nltk.tag.simplify`. These can be useful for reducing the number of different part-of-speech tags.

```
>>> from nltk.tag import simplify
>>> reader = TaggedCorpusReader('.', r'.*\.pos', tag_mapping_
function=simplify.simplify_brown_tag)
>>> reader.tagged_words(simplify_tags=True)
```

```
[('The', 'DET'), ('expense', 'N'), ('and', 'CNJ'), ...]
>>> reader = TaggedCorpusReader('.', r'.*\.pos', tag_mapping_
function=simplify.simplify_tag)
>>> reader.tagged_words(simplify_tags=True)
[('The', 'A'), ('expense', 'N'), ('and', 'C'), ...]
```

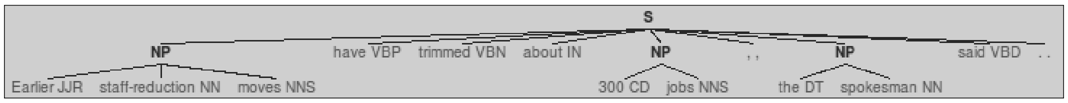
See also

Chapter 4, *Part-of-Speech Tagging* will cover part-of-speech tags and tagging in much more detail. And for more on tokenizers, see the first three recipes of *Chapter 1*, *Tokenizing Text and WordNet Basics*.

In the next recipe, we'll create a *chunked phrase* corpus, where each phrase is also part-of-speech tagged.

Creating a chunked phrase corpus

A **chunk** is a short phrase within a sentence. If you remember sentence diagrams from grade school, they were a tree-like representation of phrases within a sentence. This is exactly what chunks are: *sub-trees within a sentence tree*, and they will be covered in much more detail in *Chapter 5*, *Extracting Chunks*. Following is a sample sentence tree with three noun phrase (**NP**) chunks shown as sub-trees.



This recipe will cover how to create a corpus with sentences that contain chunks.

Getting ready

Here is an excerpt from the tagged `treebank` corpus. It has part-of-speech tags, as in the previous recipe, but it also has square brackets for denoting chunks. This is the same sentence as in the previous tree diagram, but in text form:

```
[Earlier/JJR staff-reduction/NN moves/NNS] have/VBP trimmed/VBN about/
IN [300/CD jobs/NNS] ,/, [the/DT spokesman/NN] said/VBD ./.
```

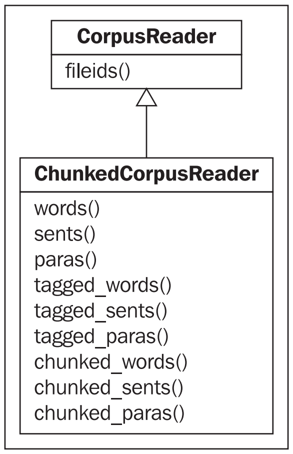
In this format, every chunk is a *noun phrase*. Words that are not within brackets are part of the sentence tree, but are not part of any noun phrase sub-tree.

How to do it...

Put this excerpt into a file called `treebank.chunk`, and then do the following:

```
>>> from nltk.corpus.reader import ChunkedCorpusReader
>>> reader = ChunkedCorpusReader('.', r'.*\.chunk')
>>> reader.chunked_words()
[Tree('NP', [('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves',
'NNS')]), ('have', 'VBP'), ...]
>>> reader.chunked_sents()
[Tree('S', [Tree('NP', [('Earlier', 'JJR'), ('staff-reduction', 'NN'),
('moves', 'NNS')]), ('have', 'VBP'), ('trimmed', 'VBN'), ('about',
'IN'), Tree('NP', [('300', 'CD'), ('jobs', 'NNS')]), (',', ','),
Tree('NP', [('the', 'DT'), ('spokesman', 'NN')]), ('said', 'VBD'),
('.', '.')])]
>>> reader.chunked_paras()
[[Tree('S', [Tree('NP', [('Earlier', 'JJR'), ('staff-reduction',
'NN'), ('moves', 'NNS')]), ('have', 'VBP'), ('trimmed', 'VBN'),
('about', 'IN'), Tree('NP', [('300', 'CD'), ('jobs', 'NNS')]), (',',
','), Tree('NP', [('the', 'DT'), ('spokesman', 'NN')]), ('said',
'VBD'), ('.', '.')])]]
```

The `ChunkedCorpusReader` provides the same methods as the `TaggedCorpusReader` for getting tagged tokens, along with three new methods for getting chunks. Each chunk is represented as an instance of `nltk.tree.Tree`. Sentence level trees look like `Tree('S', [...])` while noun phrase trees look like `Tree('NP', [...])`. In `chunked_sents()`, you get a list of sentence trees, with each noun-phrase as a sub-tree of the sentence. In `chunked_words()`, you get a list of noun phrase trees alongside tagged tokens of words that were not in a chunk. Here's an inheritance diagram listing the major methods:



You can draw a `Tree` by calling the `draw()` method. Using the corpus reader defined earlier, you could do `reader.chunked_sents()[0].draw()` to get the same sentence tree diagram shown at the beginning of this recipe.

How it works...

`ChunkedCorpusReader` is similar to the `TaggedCorpusReader` from the last recipe. It has the same default `sent_tokenizer` and `para_block_reader`, but instead of a `word_tokenizer`, it uses a `str2chunktree()` function. The default is `nltk.chunk.util.tagstr2tree()`, which parses a sentence string containing bracketed chunks into a sentence tree, with each chunk as a noun phrase sub-tree. Words are split by whitespace, and the default word/tag separator is `'/'`. If you want to customize the chunk parsing, then you can pass in your own function for `str2chunktree()`.

There's more...

An alternative format for denoting chunks is called IOB tags. **IOB** tags are similar to part-of-speech tags, but provide a way to denote the inside, outside, and beginning of a chunk. They also have the benefit of allowing multiple different chunk phrase types, not just noun phrases. Here is an excerpt from the `conll2000` corpus. Each word is on its own line with a part-of-speech tag followed by an IOB tag.

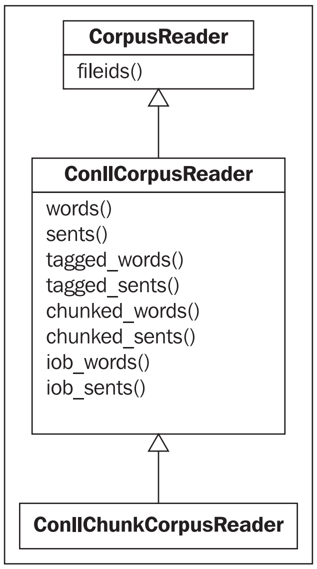
```
Mr. NNP B-NP
Meador NNP I-NP
had VBD B-VP
been VBN I-VP
executive JJ B-NP
vice NN I-NP
president NN I-NP
of IN B-PP
Balcor NNP B-NP
. . O
```

`B-NP` denotes the beginning of a noun phrase, while `I-NP` denotes that the word is inside of the current noun phrase. `B-VP` and `I-VP` denote the beginning and inside of a verb phrase. `O` ends the sentence.

To read a corpus using the IOB format, you must use the `ConllChunkCorpusReader`. Each sentence is separated by a blank line, but there is no separation for paragraphs. This means that the `para_*` methods are not available. If you put the previous IOB example text into a file named `conll.iob`, you can create and use a `ConllChunkCorpusReader` with the code we are about to see. The third argument to `ConllChunkCorpusReader` should be a tuple or list specifying the types of chunks in the file, which in this case is `('NP', 'VP', 'PP')`.

```
>>> from nltk.corpus.reader import ConllChunkCorpusReader
>>> conllreader = ConllChunkCorpusReader('.', r'.*\.iob', ('NP',
'VP', 'PP'))
>>> conllreader.chunked_words()
[Tree('NP', [('Mr.', 'NNP'), ('Meador', 'NNP')]), Tree('VP',
[('had', 'VBD'), ('been', 'VBN')]), ...]
>>> conllreader.chunked_sents()
[Tree('S', [Tree('NP', [('Mr.', 'NNP'), ('Meador', 'NNP')]),
Tree('VP', [('had', 'VBD'), ('been', 'VBN')]), Tree('NP',
[('executive', 'JJ'), ('vice', 'NN'), ('president', 'NN')]),
Tree('PP', [('of', 'IN')]), Tree('NP', [('Balcor', 'NNP')]), ('.',
'.')])]
>>> conllreader.iob_words()
[('Mr.', 'NNP', 'B-NP'), ('Meador', 'NNP', 'I-NP'), ...]
>>> conllreader.iob_sents()
[[('Mr.', 'NNP', 'B-NP'), ('Meador', 'NNP', 'I-NP'), ('had',
'VBD', 'B-VP'), ('been', 'VBN', 'I-VP'), ('executive', 'JJ', 'B-
NP'), ('vice', 'NN', 'I-NP'), ('president', 'NN', 'I-NP'), ('of',
'IN', 'B-PP'), ('Balcor', 'NNP', 'B-NP'), ('.', '.', 'O')]]
```

The previous code also shows the `iob_words()` and `iob_sents()` methods, which return lists of three tuples of `(word, pos, iob)`. The inheritance diagram for `ConllChunkCorpusReader` looks like the following, with most of the methods implemented by its superclass, `ConllCorpusReader`:



Tree leaves

When it comes to chunk trees, the leaves of a tree are the tagged tokens. So if you want to get a list of all the tagged tokens in a tree, call the `leaves()` method.

```
>>> reader.chunked_words()[0].leaves()
[('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves', 'NNS')]
>>> reader.chunked_sents()[0].leaves()
[('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves', 'NNS'),
('have', 'VBP'), ('trimmed', 'VBN'), ('about', 'IN'), ('300',
'CD'), ('jobs', 'NNS'), (',', ','), ('the', 'DT'), ('spokesman',
'NN'), ('said', 'VBD'), ('.', '.')]
>>> reader.chunked_paras()[0][0].leaves()
[('Earlier', 'JJR'), ('staff-reduction', 'NN'), ('moves', 'NNS'),
('have', 'VBP'), ('trimmed', 'VBN'), ('about', 'IN'), ('300',
'CD'), ('jobs', 'NNS'), (',', ','), ('the', 'DT'), ('spokesman',
'NN'), ('said', 'VBD'), ('.', '.')]
```

Treebank chunk corpus

The `nltk.corpus.treebank_chunk` corpus uses `ChunkedCorpusReader` to provide part-of-speech tagged words and noun phrase chunks of Wall Street Journal headlines. NLTK comes with a 5% sample from the Penn Treebank Project. You can find out more at `http://www.cis.upenn.edu/~treebank/home.html`.

CoNLL2000 corpus

CoNLL stands for the **Conference on Computational Natural Language Learning**. For the year 2000 conference, a shared task was undertaken to produce a corpus of chunks based on the Wall Street Journal corpus. In addition to noun phrases (NP), it also contains verb phrases (VP) and prepositional phrases (PP). This chunked corpus is available as `nltk.corpus.conll2000`, which is an instance of `ConllChunkCorpusReader`. You can read more at `http://www.cnts.ua.ac.be/conll2000/chunking/`.

See also

Chapter 5, Extracting Chunks will cover chunk extraction in detail. Also see the previous recipe for details on getting tagged tokens from a corpus reader.

Creating a categorized text corpus

If you have a large corpus of text, you may want to categorize it into separate sections. The brown corpus, for example, has a number of different categories.

```
>>> from nltk.corpus import brown
>>> brown.categories()
```

```
['adventure', 'belles_lettres', 'editorial', 'fiction',
'government', 'hobbies', 'humor', 'learned', 'lore', 'mystery',
'news', 'religion', 'reviews', 'romance', 'science_fiction']
```

In this recipe, we'll learn how to create our own categorized text corpus.

Getting ready

The easiest way to categorize a corpus is to have one file for each category. Following are two excerpts from the `movie_reviews` corpus:

`movie_pos.txt`

```
the thin red line is flawed but it provokes .
```

`movie_neg.txt`

```
a big-budget and glossy production can not make up for a lack of
spontaneity that permeates their tv show .
```

With these two files, we'll have two categories: `pos` and `neg`.

How to do it...

We'll use the `CategorizedPlaintextCorpusReader`, which inherits from both `PlaintextCorpusReader` and `CategorizedCorpusReader`. These two superclasses require three arguments: the root directory, the `fileids`, and a category specification.

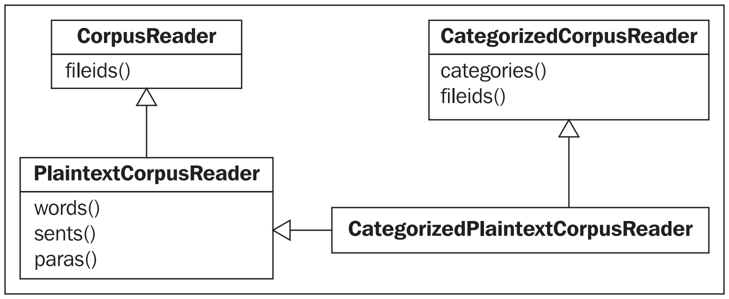
```
>>> from nltk.corpus.reader import
CategorizedPlaintextCorpusReader
>>> reader = CategorizedPlaintextCorpusReader('.', r'movie_.*\.
txt', cat_pattern=r'movie_(\w+)\.txt')
>>> reader.categories()
['neg', 'pos']
>>> reader.fileids(categories=['neg'])
['movie_neg.txt']
>>> reader.fileids(categories=['pos'])
['movie_pos.txt']
```

How it works...

The first two arguments to `CategorizedPlaintextCorpusReader` are the root directory and `fileids`, which are passed on to the `PlaintextCorpusReader` to read in the files. The `cat_pattern` keyword argument is a regular expression for extracting the category names from the `fileids`. In our case, the category is the part of the `fileid` after `movie_` and before `.txt`. **The category must be surrounded by grouping parenthesis**.

`cat_pattern` is passed to `CategorizedCorpusReader`, which overrides the common corpus reader functions such as `fileids()`, `words()`, `sents()`, and `paras()` to accept a `categories` keyword argument. This way, you could get all the pos sentences by calling `reader.sents(categories=['pos'])`. `CategorizedCorpusReader` also provides the `categories()` function, which returns a list of all known categories in the corpus.

`CategorizedPlaintextCorpusReader` is an example of using multiple-inheritance to join methods from multiple superclasses, as shown in the following diagram:



There's more...

Instead of `cat_pattern`, you could pass in a `cat_map`, which is a dictionary mapping a `fileid` to a list of category labels.

```
>>> reader = CategorizedPlaintextCorpusReader('.', r'movie_.*\.
txt', cat_map={'movie_pos.txt': ['pos'], 'movie_neg.txt':
['neg']})
>>> reader.categories()
['neg', 'pos']
```

Category file

A third way of specifying categories is to use the `cat_file` keyword argument to specify a filename containing a mapping of `fileid` to category. For example, the `brown` corpus has a file called `cats.txt` that looks like this:

```
ca44 news
cb01 editorial
```

The `reuters` corpus has files in multiple categories, and its `cats.txt` looks like this:

```
test/14840 rubber coffee lumber palm-oil veg-oil
test/14841 wheat grain
```

Categorized tagged corpus reader

The `brown` corpus reader is actually an instance of `CategorizedTaggedCorpusReader`, which inherits from `CategorizedCorpusReader` and `TaggedCorpusReader`. Just like in `CategorizedPlaintextCorpusReader`, it overrides all the methods of `TaggedCorpusReader` to allow a `categories` argument, so you can call `brown.tagged_sents(categories=['news'])` to get all the tagged sentences from the news category. You can use the `CategorizedTaggedCorpusReader` just like `CategorizedPlaintextCorpusReader` for your own categorized and tagged text corpora.

Categorized corpora

The `movie_reviews` corpus reader is an instance of `CategorizedPlaintextCorpusReader`, as is the `reuters` corpus reader. But where the `movie_reviews` corpus only has two categories (`neg` and `pos`), `reuters` has 90 categories. These corpora are often used for training and evaluating classifiers, which will be covered in *Chapter 7*, *Text Classification*.

See also

In the next recipe, we'll create a subclass of `CategorizedCorpusReader` and `ChunkedCorpusReader` for reading a categorized chunk corpus. Also see *Chapter 7*, *Text Classification* in which we use categorized text for classification.

Creating a categorized chunk corpus reader

NLTK provides a `CategorizedPlaintextCorpusReader` and `CategorizedTaggedCorpusReader`, but there's no categorized corpus reader for chunked corpora. So in this recipe, we're going to make one.

Getting ready

Refer to the earlier recipe, *Creating a chunked phrase corpus*, for an explanation of ChunkedCorpusReader, and to the previous recipe for details on CategorizedPlaintextCorpusReader and CategorizedTaggedCorpusReader, both of which inherit from CategorizedCorpusReader.

How to do it...

We'll create a class called CategorizedChunkedCorpusReader that inherits from both CategorizedCorpusReader and ChunkedCorpusReader. It is heavily based on the CategorizedTaggedCorpusReader, and also provides three additional methods for getting categorized chunks. The following code is found in catchunked.py:

```
from nltk.corpus.reader import CategorizedCorpusReader,
ChunkedCorpusReader

class CategorizedChunkedCorpusReader(CategorizedCorpusReader,
ChunkedCorpusReader):
  def __init__(self, *args, **kwargs):
    CategorizedCorpusReader.__init__(self, kwargs)
    ChunkedCorpusReader.__init__(self, *args, **kwargs)

  def _resolve(self, fileids, categories):
    if fileids is not None and categories is not None:
      raise ValueError('Specify fileids or categories, not both')
    if categories is not None:
      return self.fileids(categories)
    else:
      return fileids
```

All of the following methods call the corresponding function in ChunkedCorpusReader with the value returned from _resolve(). We'll start with the plain text methods.

```
  def raw(self, fileids=None, categories=None):
    return ChunkedCorpusReader.raw(self, self._resolve(fileids,
categories))

  def words(self, fileids=None, categories=None):
    return ChunkedCorpusReader.words(self, self._resolve(fileids,
categories))

  def sents(self, fileids=None, categories=None):
    return ChunkedCorpusReader.sents(self, self._resolve(fileids,
categories))

  def paras(self, fileids=None, categories=None):
        return ChunkedCorpusReader.paras(self, self._resolve(fileids, 
categories))
```

Next comes the tagged text methods.

```
    def tagged_words(self, fileids=None, categories=None, simplify_
tags=False):
        return ChunkedCorpusReader.tagged_words(
          self, self._resolve(fileids, categories), simplify_tags)

    def tagged_sents(self, fileids=None, categories=None, simplify_
tags=False):
        return ChunkedCorpusReader.tagged_sents(
          self, self._resolve(fileids, categories), simplify_tags)

    def tagged_paras(self, fileids=None, categories=None, simplify_
tags=False):
        return ChunkedCorpusReader.tagged_paras(
          self, self._resolve(fileids, categories), simplify_tags)
```

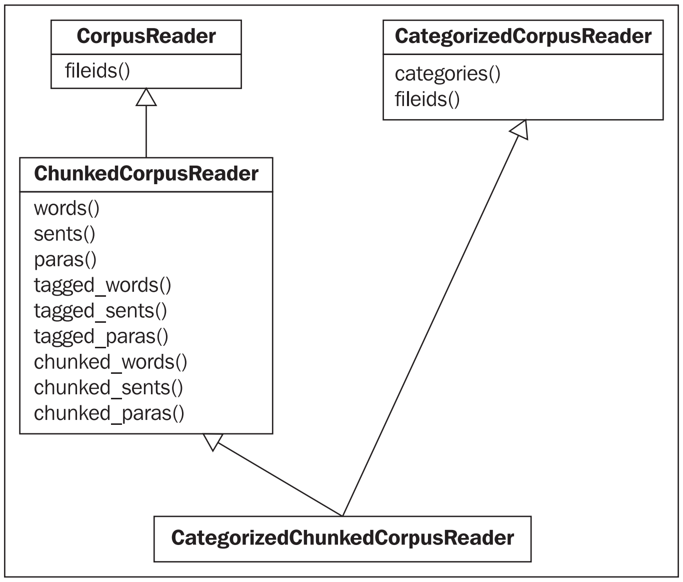
And finally, the chunked methods, which is what we've really been after.

```
    def chunked_words(self, fileids=None, categories=None):
      return ChunkedCorpusReader.chunked_words(
        self, self._resolve(fileids, categories))

    def chunked_sents(self, fileids=None, categories=None):
      return ChunkedCorpusReader.chunked_sents(
        self, self._resolve(fileids, categories))

    def chunked_paras(self, fileids=None, categories=None):
      return ChunkedCorpusReader.chunked_paras(
        self, self._resolve(fileids, categories))
```

All these methods together give us a complete `CategorizedChunkedCorpusReader`.

How it works...

CategorizedChunkedCorpusReader overrides all the ChunkedCorpusReader methods to take a categories argument for locating fileids. These fileids are found with the internal _resolve() function. This _resolve() function makes use of CategorizedCorpusReader.fileids() to return fileids for a given list of categories. If no categories are given, _resolve() just returns the given fileids, which could be None, in which case all files are read. The initialization of both CategorizedCorpusReader and ChunkedCorpusReader is what makes this all possible. If you look at the code for CategorizedTaggedCorpusReader, you'll see it's very similar. The inheritance diagram looks like this:



Here's some example code for using the treebank corpus. All we're doing is making categories out of the fileids, but the point is that you could use the same techniques to create your own categorized chunk corpus.

```
>>> import nltk.data
>>> from catchunked import CategorizedChunkedCorpusReader
>>> path = nltk.data.find('corpora/treebank/tagged')
>>> reader = CategorizedChunkedCorpusReader(path, r'wsj_.*\.pos',
cat_pattern=r'wsj_(.*)\.pos')
>>> len(reader.categories()) == len(reader.fileids())
True
>>> len(reader.chunked_sents(categories=['0001']))
16
```

We use `nltk.data.find()` to search the data directories to get a `FileSystemPathPointer` to the `treebank` corpus. All the `treebank` tagged files start with `wsj_` followed by a number, and end with `.pos`. The previous code turns that file number into a category.

There's more...

As covered in the *Creating a chunked phrase corpus* recipe, there's an alternative format and reader for a chunk corpus using IOB tags. To have a categorized corpus of IOB chunks, we have to make a new corpus reader.

Categorized Conll chunk corpus reader

Here's a subclass of `CategorizedCorpusReader` and `ConllChunkReader` called `CategorizedConllChunkCorpusReader`. It overrides all methods of `ConllCorpusReader` that take a `fileids` argument, so the methods can also take a `categories` argument. The `ConllChunkCorpusReader` is just a small subclass of `ConllCorpusReader` that handles initialization; most of the work is done in `ConllCorpusReader`. This code can also be found in `catchunked.py`.

```
from nltk.corpus.reader import CategorizedCorpusReader,
ConllCorpusReader, ConllChunkCorpusReader

class CategorizedConllChunkCorpusReader(CategorizedCorpusReader,
ConllChunkCorpusReader):
  def __init__(self, *args, **kwargs):
    CategorizedCorpusReader.__init__(self, kwargs)
    ConllChunkCorpusReader.__init__(self, *args, **kwargs)

  def _resolve(self, fileids, categories):
    if fileids is not None and categories is not None:
      raise ValueError('Specify fileids or categories, not both')
    if categories is not None:
      return self.fileids(categories)
    else:
      return fileids
```

All the following methods call the corresponding method of `ConllCorpusReader` with the value returned from `_resolve()`. We'll start with the plain text methods.

```
  def raw(self, fileids=None, categories=None):
    return ConllCorpusReader.raw(self, self._resolve(fileids,
categories))

  def words(self, fileids=None, categories=None):
    return ConllCorpusReader.words(self, self._resolve(fileids, 
categories))

  def sents(self, fileids=None, categories=None):
    return ConllCorpusReader.sents(self, self._resolve(fileids, 
categories))
```

The `ConllCorpusReader` does not recognize paragraphs, so there are no `*_paras()` methods. Next are the tagged and chunked methods.

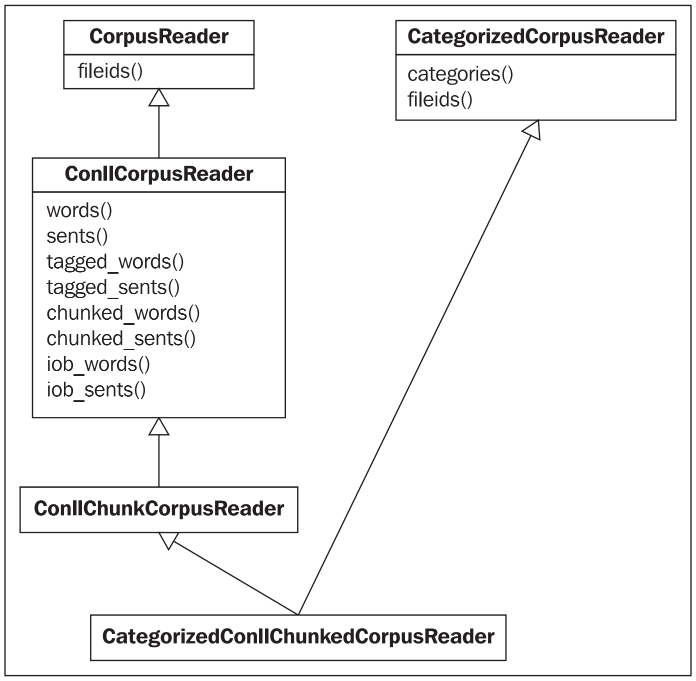
```
  def tagged_words(self, fileids=None, categories=None):
    return ConllCorpusReader.tagged_words(self, self._
resolve(fileids, categories))

  def tagged_sents(self, fileids=None, categories=None):
    return ConllCorpusReader.tagged_sents(self, self._
resolve(fileids, categories))

  def chunked_words(self, fileids=None, categories=None, chunk_
types=None):
    return ConllCorpusReader.chunked_words(
      self, self._resolve(fileids, categories), chunk_types)

  def chunked_sents(self, fileids=None, categories=None, chunk_
types=None):
    return ConllCorpusReader.chunked_sents(
      self, self._resolve(fileids, categories), chunk_types)
```

For completeness, we must override the following methods of the `ConllCorpusReader`:

```
  def parsed_sents(self, fileids=None, categories=None, pos_in_
tree=None):
    return ConllCorpusReader.parsed_sents(
      self, self._resolve(fileids, categories), pos_in_tree)

  def srl_spans(self, fileids=None, categories=None):
    return ConllCorpusReader.srl_spans(self, self._
resolve(fileids, categories))

  def srl_instances(self, fileids=None, categories=None, pos_in_
tree=None, flatten=True):
    return ConllCorpusReader.srl_instances(
      self, self._resolve(fileids, categories), pos_in_tree, 
flatten)
```

```
  def iob_words(self, fileids=None, categories=None):
    return ConllCorpusReader.iob_words(self, self._
resolve(fileids, categories))

  def iob_sents(self, fileids=None, categories=None):
    return ConllCorpusReader.iob_sents(self, self._
resolve(fileids, categories))
```

The inheritance diagram for this class is as follows:



Following is some example code using the `conll2000` corpus. Like with `treebank`, we're using the `fileids` for categories. The `ConllChunkCorpusReader` requires a third argument to specify the `chunk_types`. These `chunk_types` are used to parse the IOB tags. As you learned in the *Creating a chunked phrase corpus* recipe, the `conll2000` corpus recognizes three chunk types:

- NP for noun phrases
- VP for verb phrases
- PP for prepositional phrases

```
>>> import nltk.data
>>> from catchunked import CategorizedConllChunkCorpusReader
>>> path = nltk.data.find('corpora/conll2000')
>>> reader = CategorizedConllChunkCorpusReader(path, r'.*\.txt',
('NP','VP','PP'), cat_pattern=r'(.*)\.txt')
>>> reader.categories()
['test', 'train']
>>> reader.fileids()
['test.txt', 'train.txt']
>>> len(reader.chunked_sents(categories=['test']))
2012
```

See also

In the *Creating a chunked phrase corpus* recipe in this chapter, we covered both the `ChunkedCorpusReader` and `ConllChunkCorpusReader`. And in the previous recipe, we covered `CategorizedPlaintextCorpusReader` and `CategorizedTaggedCorpusReader`, which share the same superclass used by `CategorizedChunkedCorpusReader` and `CategorizedConllChunkReader`—`CategorizedCorpusReader`.

Lazy corpus loading

Loading a corpus reader can be an expensive operation due to the number of files, file sizes, and various initialization tasks. And while you'll often want to specify a corpus reader in a common module, you don't always need to access it right away. To speed up module import time when a corpus reader is defined, NLTK provides a `LazyCorpusLoader` class that can transform itself into your actual corpus reader as soon as you need it. This way, you can define a corpus reader in a common module without it slowing down module loading.

How to do it...

`LazyCorpusLoader` requires two arguments: the `name` of the corpus and the corpus reader class, plus any other arguments needed to initialize the corpus reader class.

The `name` argument specifies the root directory name of the corpus, which must be within a `corpora` subdirectory of one of the paths in `nltk.data.path`. See the first recipe of this chapter, *Setting up a custom corpus*, for more details on `nltk.data.path`.

For example, if you have a custom corpora named `cookbook` in your local `nltk_data` directory, its path would be `~/nltk_data/corpora/cookbook`. You'd then pass `'cookbook'` to `LazyCorpusLoader` as the `name`, and `LazyCorpusLoader` will look in `~/nltk_data/corpora` for a directory named `'cookbook'`.

The second argument to `LazyCorpusLoader` is `reader_cls`, which should be the name of a subclass of `CorpusReader`, such as `WordListCorpusReader`. You will also need to pass in any other arguments required by the `reader_cls` for initialization. This will be demonstrated as follows, using the same `wordlist` file we created in the earlier recipe, *Creating a word list corpus*. The third argument to `LazyCorpusLoader` is the list of filenames and `fileids` that will be passed in to `WordListCorpusReader` at initialization.

```
>>> from nltk.corpus.util import LazyCorpusLoader
>>> from nltk.corpus.reader import WordListCorpusReader
>>> reader = LazyCorpusLoader('cookbook', WordListCorpusReader,
['wordlist'])
>>> isinstance(reader, LazyCorpusLoader)
True
>>> reader.fileids()
['wordlist']
>>> isinstance(reader, LazyCorpusLoader)
False
>>> isinstance(reader, WordListCorpusReader)
True
```

How it works...

`LazyCorpusLoader` stores all the arguments given, but otherwise does nothing until you try to access an attribute or method. This way initialization is very fast, eliminating the overhead of loading the corpus reader immediately. As soon as you do access an attribute or method, it does the following:

1. Calls `nltk.data.find('corpora/%s' % name)` to find the corpus data root directory.
2. Instantiate the corpus reader class with the root directory and any other arguments.
3. Transforms itself into the corpus reader class.

So in the previous example code, before we call `reader.fileids()`, `reader` is an instance of `LazyCorpusLoader`, but after the call, `reader` is an instance of `WordListCorpusReader`.

There's more...

All of the corpora included with NLTK and defined in `nltk.corpus` are initially an instance of `LazyCorpusLoader`. Here's some code from `nltk.corpus` defining the `treebank` corpora.

```
treebank = LazyCorpusLoader(

    'treebank/combined', BracketParseCorpusReader, r'wsj_.*\.mrg',
    tag_mapping_function=simplify_wsj_tag)
treebank_chunk = LazyCorpusLoader(
    'treebank/tagged', ChunkedCorpusReader, r'wsj_.*\.pos',
    sent_tokenizer=RegexpTokenizer(r'(?<=/\.)\s*(?![^\[]*\])',
gaps=True),
    para_block_reader=tagged_treebank_para_block_reader)
treebank_raw = LazyCorpusLoader(
    'treebank/raw', PlaintextCorpusReader, r'wsj_.*')
```

As you can see, any number of additional arguments can be passed through by `LazyCorpusLoader` to its `reader_cls`.

Creating a custom corpus view

A **corpus view** is a class wrapper around a corpus file that reads in blocks of tokens as needed. Its purpose is to provide a *view* into a file without reading the whole file at once (since corpus files can often be quite large). If the corpus readers included by NLTK already meet all your needs, then you do not have to know anything about corpus views. But, if you have a custom file format that needs special handling, this recipe will show you how to create and use a custom corpus view. The main corpus view class is `StreamBackedCorpusView`, which opens a single file as a *stream*, and maintains an internal cache of blocks it has read.

Blocks of tokens are read in with a *block reader* function. A **block** can be any piece of text, such as a paragraph or a line, and **tokens** are parts of a block, such as individual words. In the *Creating a part-of-speech tagged word corpus* recipe, we discussed the default `para_block_reader` function of the `TaggedCorpusReader`, which reads lines from a file until it finds a blank line, then returns those lines as a single paragraph token. The actual block reader function is: `nltk.corpus.reader.util.read_blankline_block`. `TaggedCorpusReader` passes this block reader function into a `TaggedCorpusView` whenever it needs to read blocks from a file. `TaggedCorpusView` is a subclass of `StreamBackedCorpusView` that knows to split paragraphs of "word/tag" into `(word, tag)` tuples.

How to do it...

We'll start with the simple case of a plain text file with a heading that should be ignored by the corpus reader. Let's make a file called `heading_text.txt` that looks like this:

```
A simple heading
Here is the actual text for the corpus.
Paragraphs are split by blanklines.
This is the 3rd paragraph.
```

Normally we'd use the `PlaintextCorpusReader` but, by default, it will treat `A simple heading` as the first paragraph. To ignore this heading, we need to subclass the `PlaintextCorpusReader` so we can override its `CorpusView` class variable with our own `StreamBackedCorpusView` subclass. This code is found in `corpus.py`.

```
from nltk.corpus.reader import PlaintextCorpusReader
from nltk.corpus.reader.util import StreamBackedCorpusView

class IgnoreHeadingCorpusView(StreamBackedCorpusView):
  def __init__(self, *args, **kwargs):
    StreamBackedCorpusView.__init__(self, *args, **kwargs)
    # open self._stream
    self._open()
    # skip the heading block
    self.read_block(self._stream)
    # reset the start position to the current position in the
stream
    self._filepos = [self._stream.tell()]

class IgnoreHeadingCorpusReader(PlaintextCorpusReader):
  CorpusView = IgnoreHeadingCorpusView
```

To demonstrate that this works as expected, here's the code showing that the default `PlaintextCorpusReader` finds four paragraphs, while our `IgnoreHeadingCorpusReader` only has three paragraphs.

```
>>> from nltk.corpus.reader import PlaintextCorpusReader
>>> plain = PlaintextCorpusReader('.', ['heading_text.txt'])
>>> len(plain.paras())
4
>>> from corpus import IgnoreHeadingCorpusReader
>>> reader = IgnoreHeadingCorpusReader('.', ['heading_text.txt'])
>>> len(reader.paras())
3
```

How it works...

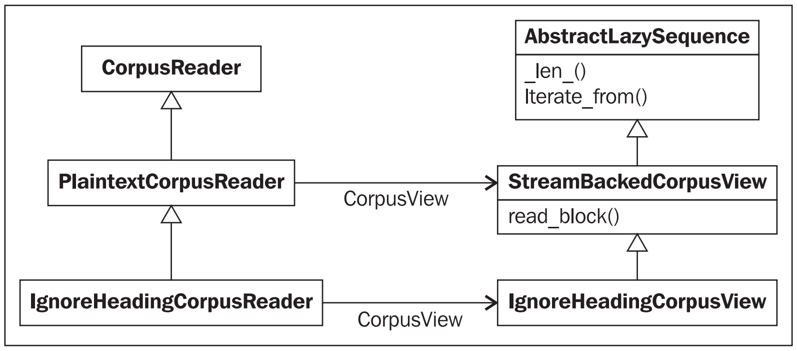
The `PlaintextCorpusReader` by design has a `CorpusView` class variable that can be overridden by subclasses. So we do just that, and make our `IgnoreHeadingCorpusView` the `CorpusView`.

Most corpus readers do not have a `CorpusView` class variable because they require very specific corpus views.

The `IgnoreHeadingCorpusView` is a subclass of `StreamBackedCorpusView` that does the following on initialization:

1. Open the file using `self._open()`. This function is defined by `StreamBackedCorpusView`, and sets the internal instance variable `self._stream` to the opened file.
2. Read one block with `read_blankline_block()`, which will read the heading as a paragraph, and move the stream's file position forward to the next block.
3. Reset the start file position to the current position of `self._stream`. `self._filepos` is an internal index of where each block is in the file.

Here's a diagram illustrating the relationships between the classes:



There's more...

Corpus views can get a lot fancier and more complicated, but the core concept is the same: read *blocks* from a `stream` to return a list of *tokens*. There are a number of block readers provided in `nltk.corpus.reader.util`, but you can always create your own. If you do want to define your own block reader function, then you have two choices on how to implement it:

1. Define it as a separate function and pass it in to `StreamBackedCorpusView` as `block_reader`. This is a good option if your block reader is fairly simple, reusable, and doesn't require any outside variables or configuration.

2. Subclass `StreamBackedCorpusView` and override the `read_block()` method. This is what many custom corpus views do because the block reading is highly specialized and requires additional functions and configuration, usually provided by the corpus reader when the corpus view is initialized.

Block reader functions

Following is a survey of most of the included block readers in `nltk.corpus.reader.util`. Unless otherwise noted, each block reader function takes a single argument: the `stream` to read from.

- `read_whitespace_block()` will read 20 lines from the stream, splitting each line into tokens by whitespace.
- `read_wordpunct_block()` reads 20 lines from the stream, splitting each line using `nltk.tokenize.wordpunct_tokenize()`.
- `read_line_block()` reads 20 lines from the stream and returns them as a list, with each line as a token.
- `read_blankline_block()` will read lines from the stream until it finds a blank line. It will then return a single token of all lines found combined into a single string.
- `read_regexp_block()` takes two additional arguments, which must be regular expressions that can be passed to `re.match()`: a `start_re` and `end_re`. `start_re` matches the starting line of a block, and `end_re` matches the ending line of the block. `end_re` defaults to `None`, in which case the block will end as soon as a new `start_re` match is found. The return value is a single token of all lines in the block joined into a single string.

Pickle corpus view

If you want to have a corpus of pickled objects, you can use the `PickleCorpusView`, a subclass of `StreamBackedCorpusView` found in `nltk.corpus.reader.util`. A file consists of blocks of pickled objects, and can be created with the `PickleCorpusView.write()` class method, which takes a sequence of objects and an output file, then pickles each object using `pickle.dump()` and writes it to the file. It overrides the `read_block()` method to return a list of unpickled objects from the stream, using `pickle.load()`.

Concatenated corpus view

Also found in `nltk.corpus.reader.util` is the `ConcatenatedCorpusView`. This class is useful if you have multiple files that you want a corpus reader to treat as a single file. A `ConcatenatedCorpusView` is created by giving it a list of `corpus_views`, which are then iterated over as if they were a single view.

See also

The concept of block readers was introduced in the *Creating a part-of-speech tagged word corpus* recipe in this chapter.

Creating a MongoDB backed corpus reader

All the corpus readers we've dealt with so far have been file-based. That is in part due to the design of the `CorpusReader` base class, and also the assumption that most corpus data will be in text files. But sometimes you'll have a bunch of data stored in a database that you want to access and use just like a text file corpus. In this recipe, we'll cover the case where you have documents in MongoDB, and you want to use a particular field of each document as your block of text.

Getting ready

MongoDB is a document-oriented database that has become a popular alternative to relational databases such as MySQL. The installation and setup of MongoDB is outside the scope of this book, but you can find instructions at `http://www.mongodb.org/display/DOCS/Quickstart`.

You'll also need to install PyMongo, a Python driver for MongoDB. You should be able to do this with either `easy_install` or `pip`, by doing `sudo easy_install pymongo` or `sudo pip install pymongo`.

The code in the *How to do it...* section assumes that your database is on `localhost` port `27017`, which is the MongoDB default configuration, and that you'll be using the `test` database with a collection named `corpus` that contains documents with a `text` field. Explanations for these arguments are available in the PyMongo documentation at `http://api.mongodb.org/python/`.

How to do it...

Since the `CorpusReader` class assumes you have a file-based corpus, we can't directly subclass it. Instead, we're going to emulate both the `StreamBackedCorpusView` and `PlaintextCorpusReader`. `StreamBackedCorpusView` is a subclass of `nltk.util.AbstractLazySequence`, so we'll subclass `AbstractLazySequence` to create a MongoDB view, and then create a new class that will use the view to provide functionality similar to the `PlaintextCorpusReader`. This code is found in `mongoreader.py`.

```
import pymongo
from nltk.data import LazyLoader
from nltk.tokenize import TreebankWordTokenizer
from nltk.util import AbstractLazySequence, LazyMap,
LazyConcatenation
class MongoDBLazySequence(AbstractLazySequence):
  def __init__(self, host='localhost', port=27017, db='test',
collection='corpus', field='text'):
    self.conn = pymongo.Connection(host, port)
    self.collection = self.conn[db][collection]
    self.field = field
  def __len__(self):
    return self.collection.count()
  def iterate_from(self, start):
    f = lambda d: d.get(self.field, '')
    return iter(LazyMap(f, self.collection.find(fields=[self.
field], skip=start)))
class MongoDBCorpusReader(object):
  def __init__(self, word_tokenizer=TreebankWordTokenizer(),
         sent_tokenizer=LazyLoader('tokenizers/punkt/english.
pickle'),
         **kwargs):
    self._seq = MongoDBLazySequence(**kwargs)
    self._word_tokenize = word_tokenizer.tokenize
    self._sent_tokenize = sent_tokenizer.tokenize
  def text(self):
    return self._seq
  def words(self):
    return LazyConcatenation(LazyMap(self._word_tokenize, self.
text()))
  def sents(self):
    return LazyConcatenation(LazyMap(self._sent_tokenize, self.
text()))
```

How it works...

`AbstractLazySequence` is an abstract class that provides read-only, on-demand iteration. Subclasses must implement the `__len__()` and `iterate_from(start)` methods, while it provides the rest of the list and iterator emulation methods. By creating the `MongoDBLazySequence` subclass as our view, we can iterate over documents in the MongoDB collection on-demand, without keeping all the documents in memory. `LazyMap` is a lazy version of Python's built-in `map()` function, and is used in `iterate_from()` to transform the document into the specific field that we're interested in. It's also a subclass of `AbstractLazySequence`.

The `MongoDBCorpusReader` creates an internal instance of `MongoDBLazySequence` for iteration, then defines the word and sentence tokenization methods. The `text()` method simply returns the instance of `MongoDBLazySequence`, which results in a lazily evaluated list of each text field. The `words()` method uses `LazyMap` and `LazyConcatenation` to return a lazily evaluated list of all words, while the `sents()` method does the same for sentences. The `sent_tokenizer` is loaded on demand with `LazyLoader`, which is a wrapper around `nltk.data.load()`, analogous to `LazyCorpusLoader`. `LazyConcatentation` is a subclass of `AbstractLazySequence` too, and produces a flat list from a given list of lists (each list may also be lazy). In our case, we're concatenating the results of `LazyMap` to ensure we don't return nested lists.

There's more...

All of the parameters are configurable. For example, if you had a db named `website`, with a `collection` named `comments`, whose documents had a `field` called `comment`, you could create a `MongoDBCorpusReader` as follows:

```
>>> reader = MongoDBCorpusReader(db='website',
collection='comments', field='comment')
```

You can also pass in custom instances for `word_tokenizer` and `sent_tokenizer`, as long as the objects implement the `nltk.tokenize.TokenizerI` interface by providing a `tokenize(text)` method.

See also

Corpus views were covered in the previous recipe, and tokenization was covered in *Chapter 1, Tokenizing Text and WordNet Basics*.

Corpus editing with file locking

Corpus readers and views are all read-only, but there may be times when you want to add to or edit the corpus files. However, modifying a corpus file while other processes are using it, such as through a corpus reader, can lead to dangerous undefined behavior. This is where file locking comes in handy.

Getting ready

You must install the `lockfile` library using `sudo easy_install lockfile` or `sudo pip install lockfile`. This library provides cross-platform file locking, and so will work on Windows, Unix/Linux, Mac OX, and more. You can find detailed documentation on `lockfile` at `http://packages.python.org/lockfile/`.

For the following code to work, you must also have Python 2.6. Versions 2.4 and earlier do not support the `with` keyword.

How to do it...

Here are two file editing functions: `append_line()` and `remove_line()`. Both try to acquire an *exclusive lock* on the file before updating it. An **exclusive lock** means that these functions will wait until no other process is reading from or writing to the file. Once the lock is acquired, any other process that tries to access the file will have to wait until the lock is released. This way, modifying the file will be safe and not cause any undefined behavior in other processes. These functions can be found in `corpus.py`.

```
import lockfile, tempfile, shutil

def append_line(fname, line):
  with lockfile.FileLock(fname):
    fp = open(fname, 'a+')
    fp.write(line)
    fp.write('\n')
    fp.close()

def remove_line(fname, line):

  with lockfile.FileLock(fname):
    tmp = tempfile.TemporaryFile()
    fp = open(fname, 'r+')
    # write all lines from orig file, except if matches given line
    for l in fp:
      if l.strip() != line:
        tmp.write(l)
    # reset file pointers so entire files are copied
    fp.seek(0)
    tmp.seek(0)
    # copy tmp into fp, then truncate to remove trailing line(s)
    shutil.copyfileobj(tmp, fp)
    fp.truncate()
    fp.close()
    tmp.close()
```

The lock acquiring and releasing happens transparently when you do `with lockfile.FileLock(fname)`.

[Instead of using `with lockfile.FileLock(fname)`, you can also get a lock by calling `lock = lockfile.FileLock(fname)`, then call `lock.acquire()` to acquire the lock, and `lock.release()` to release the lock. This alternative usage is compatible with Python 2.4.]

How it works...

You can use these functions as follows:

```
    >>> from corpus import append_line, remove_line
    >>> append_line('test.txt', 'foo')
    >>> remove_line('test.txt', 'foo')
```

In `append_line()`, a lock is acquired, the file is opened in *append mode*, the text is written along with an end-of-line character, and then the file is closed, releasing the lock.

[A lock acquired by `lockfile` only protects the file from other processes that also use `lockfile`. In other words, just because your Python process has a lock with `lockfile`, doesn't mean a non-Python process can't modify the file. For this reason, it's best to only use `lockfile` with files that will not be edited by any non-Python processes, or Python processes that do not use `lockfile`.]

The `remove_line()` function is a bit more complicated. Because we're removing a line and not a specific section of the file, we need to iterate over the file to find each instance of the line to remove. The easiest way to do this while writing the changes back to the file, is to use a `TemporaryFile` to hold the changes, then copy that file back into the original file using `shutil.copyfileobj()`.

These functions are best suited for a word list corpus, or some other corpus type with presumably unique lines, that may be edited by multiple people at about the same time, such as through a web interface. Using these functions with a more document-oriented corpus such as `brown`, `treebank`, or `conll2000`, is probably a bad idea.