CHAPTER 3
Processing Raw Text

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

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

1. How can we write programs to access text from local files and from the Web, in order to get hold of an unlimited range of language material?
2. How can we split documents up into individual words and punctuation symbols, so we can carry out the same kinds of analysis we did with text corpora in earlier chapters?
3. How can we write programs to produce formatted output and save it in a file?

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

Important: From this chapter onwards, our program samples will assume you begin your interactive session or your program with the following import statements:

```
>>> from __future__ import division
>>> import nltk, re, pprint
```

3.1 Accessing Text from the Web and from Disk

Electronic Books

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

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

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

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

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

The variable `raw` contains a string with 1,176,831 characters. (We can see that it is a string, using `type(raw)`.) This is the raw content of the book, including many details we are not interested in, such as whitespace, line breaks, and blank lines. Notice the `\r` and `\n` in the opening line of the file, which is how Python displays the special carriage return and line-feed characters (the file must have been created on a Windows machine). For our language processing, we want to break up the string into words and punctuation, as we saw in Chapter 1. This step is called **tokenization**, and it produces our familiar structure, a list of words and punctuation.

```
>>> tokens = nltk.word_tokenize(raw)
>>> type(tokens)
<type 'list'>
>>> len(tokens)
255809
>>> tokens[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime', 'and', 'Punishment', ',', 'by']
```

Notice that NLTK was needed for tokenization, but not for any of the earlier tasks of opening a URL and reading it into a string. If we now take the further step of creating an NLTK text from this list, we can carry out all of the other linguistic processing we saw in Chapter 1, along with the regular list operations, such as slicing:

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

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

```
>>> raw.find("PART I")
5303
>>> raw.rfind("End of Project Gutenberg's Crime")
1157681
>>> raw = raw[5303:1157681] ❶
>>> raw.find("PART I")
0
```

The `find()` and `rfind()` ("reverse find") methods help us get the right index values to use for slicing the string ❶. We overwrite `raw` with this slice, so now it begins with "PART I" and goes up to (but not including) the phrase that marks the end of the content.

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

Dealing with HTML

Much of the text on the Web is in the form of HTML documents. You can use a web browser to save a page as text to a local file, then access this as described in the later section on files. However, if you're going to do this often, it's easiest to get Python to do the work directly. The first step is the same as before, using `urlopen`. For fun we'll

pick a BBC News story called "Blondes to die out in 200 years," an urban legend passed along by the BBC as established scientific fact:

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

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

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

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

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

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

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

Processing Search Engine Results

The Web can be thought of as a huge corpus of unannotated text. Web search engines provide an efficient means of searching this large quantity of text for relevant linguistic examples. The main advantage of search engines is size: since you are searching such a large set of documents, you are more likely to find any linguistic pattern you are interested in. Furthermore, you can make use of very specific patterns, which would match only one or two examples on a smaller example, but which might match tens of thousands of examples when run on the Web. A second advantage of web search engines is that they are very easy to use. Thus, they provide a very convenient tool for quickly checking a theory, to see if it is reasonable. See Table 3-1 for an example.

Table 3-1. Google hits for collocations: The number of hits for collocations involving the words absolutely *or* definitely, *followed by one of* adore, love, like, *or* prefer. *(Liberman, in LanguageLog, 2005)*

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

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

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

Processing RSS Feeds

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

```
>>> import feedparser
>>> llog = feedparser.parse("http://languagelog.ldc.upenn.edu/nll/?feed=atom")
>>> llog['feed']['title']
u'Language Log'
>>> len(llog.entries)
15
>>> post = llog.entries[2]
>>> post.title
u"He's My BF"
>>> content = post.content[0].value
>>> content[:70]
u'<p>Today I was chatting with three of our visiting graduate students f'
>>> nltk.word_tokenize(nltk.html_clean(content))
>>> nltk.word_tokenize(nltk.clean_html(llog.entries[2].content[0].value))
[u'Today', u'I', u'was', u'chatting', u'with', u'three', u'of', u'our', u'visiting',
u'graduate', u'students', u'from', u'the', u'PRC', u'.', u'Thinking', u'that', u'I',
```

```
u'was', u'being', u'au', u'courant', u',', u'I', u'mentioned', u'the', u'expression',
u'DUI4XIANG4', u'\u5c0d\u8c61', u'("', u'boy', u'/', u'girl', u'friend', u'"', ...]
```

Note that the resulting strings have a `u` prefix to indicate that they are Unicode strings (see Section 3.3). With some further work, we can write programs to create a small corpus of blog posts, and use this as the basis for our NLP work.

Reading Local Files

In order to read a local file, we need to use Python's built-in `open()` function, followed by the `read()` method. Supposing you have a file *document.txt*, you can load its contents like this:

```
>>> f = open('document.txt')
>>> raw = f.read()
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Extracting Text from PDF, MSWord, and Other Binary Formats

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

Capturing User Input

Sometimes we want to capture the text that a user inputs when she is interacting with our program. To prompt the user to type a line of input, call the Python function `raw_input()`. After saving the input to a variable, we can manipulate it just as we have done for other strings.

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

The NLP Pipeline

Figure 3-1 summarizes what we have covered in this section, including the process of building a vocabulary that we saw in Chapter 1. (One step, normalization, will be discussed in Section 3.6.)

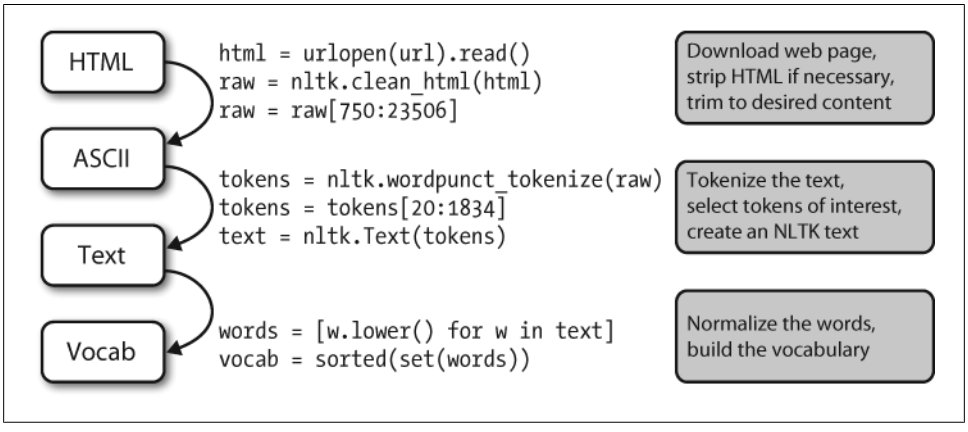


Figure 3-1. The processing pipeline: We open a URL and read its HTML content, remove the markup and select a slice of characters; this is then tokenized and optionally converted into an `nltk.Text` *object; we can also lowercase all the words and extract the vocabulary.*

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

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

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

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

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

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

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

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

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

In the next section, we examine strings more closely and further explore the relationship between strings and lists.

3.2 Strings: Text Processing at the Lowest Level

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

Basic Operations with Strings

Strings are specified using single quotes ❶ or double quotes ❷, as shown in the following code example. If a string contains a single quote, we must backslash-escape the quote ❸ so Python knows a literal quote character is intended, or else put the string in double quotes ❷. Otherwise, the quote inside the string ❹ will be interpreted as a close quote, and the Python interpreter will report a syntax error:

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

Sometimes strings go over several lines. Python provides us with various ways of entering them. In the next example, a sequence of two strings is joined into a single string. We need to use backslash ❶ or parentheses ❷ so that the interpreter knows that the statement is not complete after the first line.

```
>>> couplet = "Shall I compare thee to a Summer's day?"\
...           "Thou are more lovely and more temperate:" ❶
>>> print couplet
Shall I compare thee to a Summer's day?Thou are more lovely and more temperate:
>>> couplet = ("Rough winds do shake the darling buds of May,"
...           "And Summer's lease hath all too short a date:") ❷
>>> print couplet
Rough winds do shake the darling buds of May,And Summer's lease hath all too short a date:
```

Unfortunately these methods do not give us a newline between the two lines of the sonnet. Instead, we can use a triple-quoted string as follows:

```
>>> couplet = """Shall I compare thee to a Summer's day?
... Thou are more lovely and more temperate:"""
>>> print couplet
Shall I compare thee to a Summer's day?
Thou are more lovely and more temperate:
>>> couplet = '''Rough winds do shake the darling buds of May,
... And Summer's lease hath all too short a date:'''
>>> print couplet
Rough winds do shake the darling buds of May,
And Summer's lease hath all too short a date:
```

Now that we can define strings, we can try some simple operations on them. First let's look at the `+` operation, known as **concatenation** ❶. It produces a new string that is a copy of the two original strings pasted together end-to-end. Notice that concatenation doesn't do anything clever like insert a space between the words. We can even multiply strings ❷:

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

Your Turn: Try running the following code, then try to use your understanding of the string `+` and `*` operations to figure out how it works. Be careful to distinguish between the string `' '`, which is a single whitespace character, and `''`, which is the empty string.

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2, 1]
>>> b = [' ' * 2 * (7 - i) + 'very' * i for i in a]
>>> for line in b:
...     print b
```

We've seen that the addition and multiplication operations apply to strings, not just numbers. However, note that we cannot use subtraction or division with strings:

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

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

Printing Strings

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

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

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

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

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

Accessing Individual Characters

As we saw in Section 1.2 for lists, strings are indexed, starting from zero. When we index a string, we get one of its characters (or letters). A single character is nothing special—it's just a string of length `1`.

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

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

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

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

```
>>> monty[-1] ❶
'n'
>>> monty[5]
' '
>>> monty[-7]
' '
```

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

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

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

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

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

Accessing Substrings

A substring is any continuous section of a string that we want to pull out for further processing. We can easily access substrings using the same slice notation we used for lists (see Figure 3-2). For example, the following code accesses the substring starting at index `6`, up to (but not including) index `10`:

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

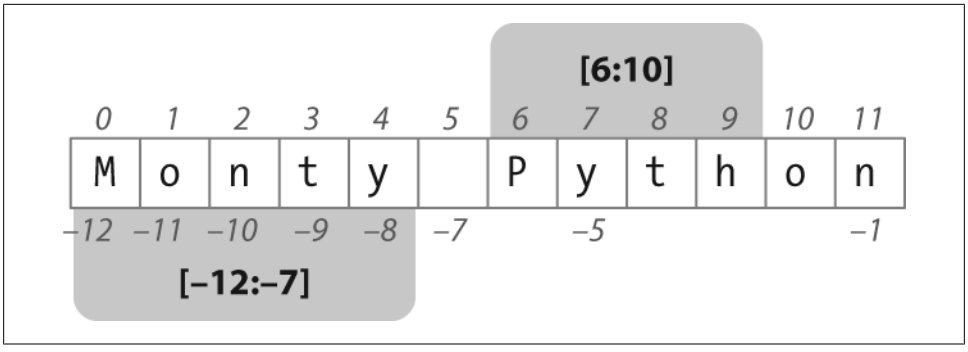


Figure 3-2. String slicing: The string Monty Python *is shown along with its positive and negative indexes; two substrings are selected using "slice" notation. The slice* `[m,n]` *contains the characters from position m through n-1.*

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

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

```
>>> monty[-12:-7]
'Monty'
```

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

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

We test if a string contains a particular substring using the `in` operator, as follows:

```
>>> phrase = 'And now for something completely different'
>>> if 'thing' in phrase:
...     print 'found "thing"'
found "thing"
```

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

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

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

More Operations on Strings

Python has comprehensive support for processing strings. A summary, including some operations we haven't seen yet, is shown in Table 3-2. For more information on strings, type `help(str)` at the Python prompt.

Table 3-2. Useful string methods: Operations on strings in addition to the string tests shown in Table 1-4; all methods produce a new string or list

Method	Functionality
`s.find(t)`	Index of first instance of string `t` inside `s` (`-1` if not found)
`s.rfind(t)`	Index of last instance of string `t` inside `s` (`-1` if not found)
`s.index(t)`	Like `s.find(t)`, except it raises `ValueError` if not found
`s.rindex(t)`	Like `s.rfind(t)`, except it raises `ValueError` if not found
`s.join(text)`	Combine the words of the text into a string using `s` as the glue
`s.split(t)`	Split `s` into a list wherever a `t` is found (whitespace by default)
`s.splitlines()`	Split `s` into a list of strings, one per line
`s.lower()`	A lowercased version of the string `s`
`s.upper()`	An uppercased version of the string `s`
`s.titlecase()`	A titlecased version of the string `s`
`s.strip()`	A copy of `s` without leading or trailing whitespace
`s.replace(t, u)`	Replace instances of `t` with `u` inside `s`

The Difference Between Lists and Strings

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

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

When we open a file for reading into a Python program, we get a string corresponding to the contents of the whole file. If we use a `for` loop to process the elements of this string, all we can pick out are the individual characters—we don't get to choose the granularity. By contrast, the elements of a list can be as big or small as we like: for example, they could be paragraphs, sentences, phrases, words, characters. So lists have the advantage that we can be flexible about the elements they contain, and correspondingly flexible about any downstream processing. Consequently, one of the first things we are likely to do in a piece of NLP code is tokenize a string into a list of strings (Section 3.7). Conversely, when we want to write our results to a file, or to a terminal, we will usually format them as a string (Section 3.9).

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

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

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

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

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

Your Turn: Consolidate your knowledge of strings by trying some of the exercises on strings at the end of this chapter.

3.3 Text Processing with Unicode

Our programs will often need to deal with different languages, and different character sets. The concept of "plain text" is a fiction. If you live in the English-speaking world you probably use ASCII, possibly without realizing it. If you live in Europe you might use one of the extended Latin character sets, containing such characters as "ø" for Danish and Norwegian, "ő" for Hungarian, "ñ" for Spanish and Breton, and "ň" for Czech and Slovak. In this section, we will give an overview of how to use Unicode for processing texts that use non-ASCII character sets.

What Is Unicode?

Unicode supports over a million characters. Each character is assigned a number, called a **code point**. In Python, code points are written in the form `\u`*XXXX*, where *XXXX* is the number in four-digit hexadecimal form.

Within a program, we can manipulate Unicode strings just like normal strings. However, when Unicode characters are stored in files or displayed on a terminal, they must be encoded as a stream of bytes. Some encodings (such as ASCII and Latin-2) use a single byte per code point, so they can support only a small subset of Unicode, enough for a single language. Other encodings (such as UTF-8) use multiple bytes and can represent the full range of Unicode characters.

Text in files will be in a particular encoding, so we need some mechanism for translating it into Unicode—translation into Unicode is called **decoding**. Conversely, to write out Unicode to a file or a terminal, we first need to translate it into a suitable encoding—this translation out of Unicode is called **encoding**, and is illustrated in Figure 3-3.

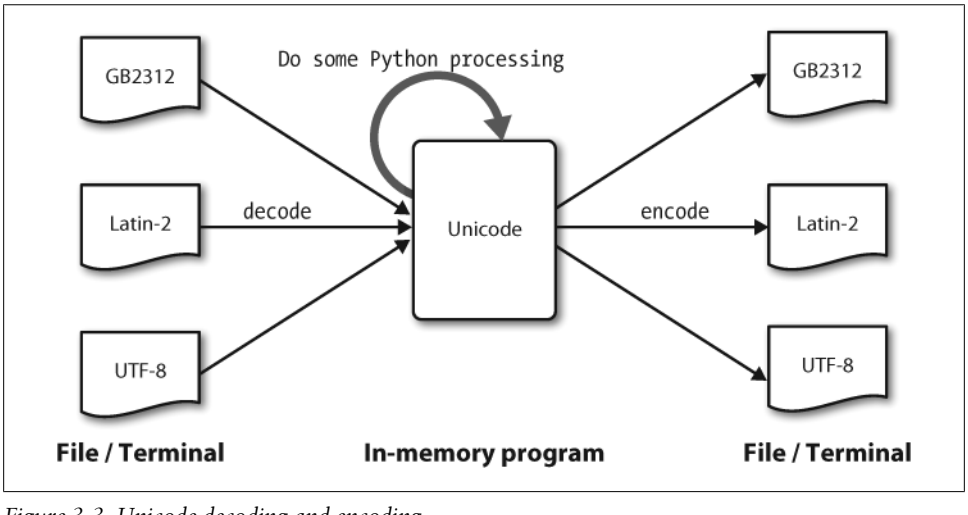


Figure 3-3. Unicode decoding and encoding.

From a Unicode perspective, characters are abstract entities that can be realized as one or more **glyphs**. Only glyphs can appear on a screen or be printed on paper. A font is a mapping from characters to glyphs.

~~Extracting Encoded Text from Files~~

~~Let's assume that we have a small text file, and that we know how it is encoded. For example, *polish-lat2.txt*, as the name suggests, is a snippet of Polish text (from the Polish Wikipedia; see *http://pl.wikipedia.org/wiki/Biblioteka_Pruska*). This file is encoded as Latin-2, also known as ISO-8859-2. The function `nltk.data.find()` locates the file for us.~~

```
>>> path = nltk.data.find('corpora/unicode_samples/polish-lat2.txt')
```

The Python `codecs` module provides functions to read encoded data into Unicode strings, and to write out Unicode strings in encoded form. The `codecs.open()` function takes an encoding parameter to specify the encoding of the file being read or written. So let's import the `codecs` module, and call it with the encoding `'latin2'` to open our Polish file as Unicode:

```
>>> import codecs
>>> f = codecs.open(path, encoding='latin2')
```

For a list of encoding parameters allowed by `codecs`, see *http://docs.python.org/lib/standard-encodings.html*. Note that we can write Unicode-encoded data to a file using `f = codecs.open(path, 'w', encoding='utf-8')`.

Text read from the file object `f` will be returned in Unicode. As we pointed out earlier, in order to view this text on a terminal, we need to encode it, using a suitable encoding. The Python-specific encoding `unicode_escape` is a dummy encoding that converts all non-ASCII characters into their `\u`*XXXX* representations. Code points above the ASCII 0–127 range but below 256 are represented in the two-digit form `\x`*XX*.

```
>>> for line in f:
...     line = line.strip()
...     print line.encode('unicode_escape')
Pruska Biblioteka Pa\u0144stwowa. Jej dawne zbiory znane pod nazw\u0105
"Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez
Niemc\xf3w pod koniec II wojny \u015bwiatowej na Dolny \u015al\u0105sk, zosta\u0142y
odnalezione po 1945 r. na terytorium Polski. Trafi\u0142y do Biblioteki
Jagiello\u0144skiej w Krakowie, obejmuj\u0105 ponad 500 tys. zabytkowych
archiwali\xf3w, m.in. manuskrypty Goethego, Mozarta, Beethovena, Bacha.
```

The first line in this output illustrates a Unicode escape string preceded by the `\u` escape string, namely `\u0144`. The relevant Unicode character will be displayed on the screen as the glyph `ń`. In the third line of the preceding example, we see `\xf3`, which corresponds to the glyph ó, and is within the 128–255 range.

In Python, a Unicode string literal can be specified by preceding an ordinary string literal with a `u`, as in `u'hello'`. Arbitrary Unicode characters are defined using the `\u`*XXXX* escape sequence inside a Unicode string literal. We find the integer ordinal of a character using `ord()`. For example:

```
>>> ord('a')
97
```

The hexadecimal four-digit notation for 97 is 0061, so we can define a Unicode string literal with the appropriate escape sequence:

```
>>> a = u'\u0061'
>>> a
u'a'
>>> print a
a
```

Notice that the Python `print` statement is assuming a default encoding of the Unicode character, namely ASCII. However, ń is outside the ASCII range, so cannot be printed unless we specify an encoding. In the following example, we have specified that `print` should use the `repr()` of the string, which outputs the UTF-8 escape sequences (of the form \x*XX*) rather than trying to render the glyphs.

```
>>> nacute = u'\u0144'
>>> nacute
u'\u0144'
>>> nacute_utf = nacute.encode('utf8')
>>> print repr(nacute_utf)
'\xc5\x84'
```

If your operating system and locale are set up to render UTF-8 encoded characters, you ought to be able to give the Python command `print nacute_utf` and see ń on your screen.

There are many factors determining what glyphs are rendered on your screen. If you are sure that you have the correct encoding, but your Python code is still failing to produce the glyphs you expected, you should also check that you have the necessary fonts installed on your system.

The module `unicodedata` lets us inspect the properties of Unicode characters. In the following example, we select all characters in the third line of our Polish text outside the ASCII range and print their UTF-8 escaped value, followed by their code point integer using the standard Unicode convention (i.e., prefixing the hex digits with `U+`), followed by their Unicode name.

```
>>> import unicodedata
>>> lines = codecs.open(path, encoding='latin2').readlines()
>>> line = lines[2]
>>> print line.encode('unicode_escape')
Niemc\xf3w pod koniec II wojny \u015bwiatowej na Dolny \u015al\u0105sk, zosta\u0142y\n
>>> for c in line:
...     if ord(c) > 127:
...         print '%r U+%04x %s' % (c.encode('utf8'), ord(c), unicodedata.name(c))
'\xc3\xb3' U+00f3 LATIN SMALL LETTER O WITH ACUTE
'\xc5\x9b' U+015b LATIN SMALL LETTER S WITH ACUTE
'\xc5\x9a' U+015a LATIN CAPITAL LETTER S WITH ACUTE
'\xc4\x85' U+0105 LATIN SMALL LETTER A WITH OGONEK
'\xc5\x82' U+0142 LATIN SMALL LETTER L WITH STROKE
```

If you replace the `%r` (which yields the `repr()` value) by `%s` in the format string of the preceding code sample, and if your system supports UTF-8, you should see an output like the following:

```
ó U+00f3 LATIN SMALL LETTER O WITH ACUTE
ś U+015b LATIN SMALL LETTER S WITH ACUTE
Ś U+015a LATIN CAPITAL LETTER S WITH ACUTE
```

```
ą U+0105 LATIN SMALL LETTER A WITH OGONEK
ł U+0142 LATIN SMALL LETTER L WITH STROKE
```

Alternatively, you may need to replace the encoding `'utf8'` in the example by `'latin2'`, again depending on the details of your system.

The next examples illustrate how Python string methods and the `re` module accept Unicode strings.

```
>>> line.find(u'zosta\u0142y')
54
>>> line = line.lower()
>>> print line.encode('unicode_escape')
niemc\xf3w pod koniec ii wojny \u015bwiatowej na dolny \u015bl\u0105sk, zosta\u0142y\n
>>> import re
>>> m = re.search(u'\u015b\w*', line)
>>> m.group()
u'\u015bwiatowej'
```

NLTK tokenizers allow Unicode strings as input, and correspondingly yield Unicode strings as output.

```
>>> nltk.word_tokenize(line)
[u'niemc\xf3w', u'pod', u'koniec', u'ii', u'wojny', u'\u015bwiatowej',
u'na', u'dolny', u'\u015bl\u0105sk', u'zosta\u0142y']
```

Using Your Local Encoding in Python

If you are used to working with characters in a particular local encoding, you probably want to be able to use your standard methods for inputting and editing strings in a Python file. In order to do this, you need to include the string `'# -*- coding: <coding> -*-'` as the first or second line of your file. Note that `<coding>` has to be a string like `'latin-1'`, `'big5'`, or `'utf-8'` (see Figure 3-4).

Figure 3-4 also illustrates how regular expressions can use encoded strings.

3.4 Regular Expressions for Detecting Word Patterns

Many linguistic processing tasks involve pattern matching. For example, we can find words ending with *ed* using `endswith('ed')`. We saw a variety of such "word tests" in Table 1-4. Regular expressions give us a more powerful and flexible method for describing the character patterns we are interested in.

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

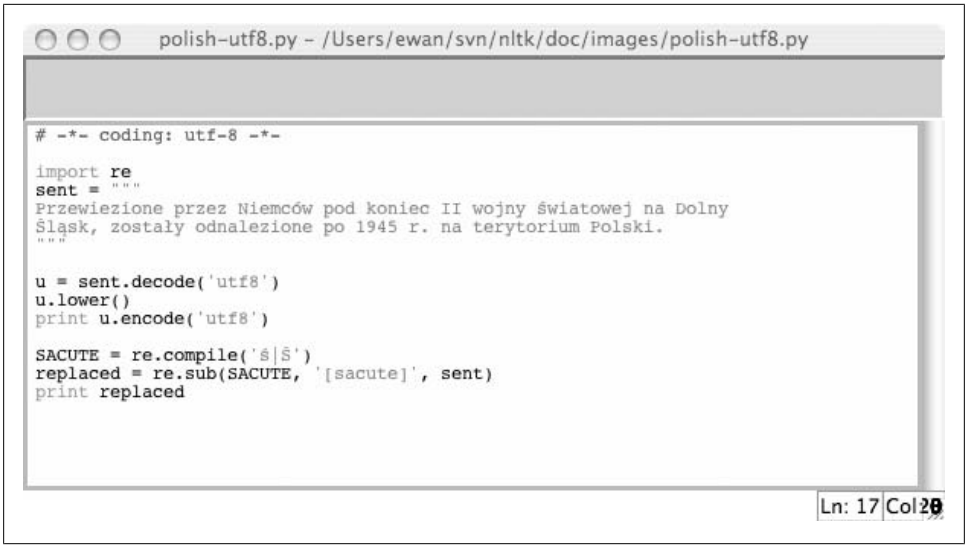


Figure 3-4. Unicode and IDLE: UTF-8 encoded string literals in the IDLE editor; this requires that an appropriate font is set in IDLE's preferences; here we have chosen Courier CE.

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

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

Using Basic Metacharacters

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

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

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

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

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

Finally, the ? symbol specifies that the previous character is optional. Thus «`^e-?mail $`» will match both *email* and *e-mail*. We could count the total number of occurrences of this word (in either spelling) in a text using `sum(1 for w in text if re.search('^e-?mail$', w))`.

Ranges and Closures

The **T9** system is used for entering text on mobile phones (see Figure 3-5). Two or more words that are entered with the same sequence of keystrokes are known as **textonyms**. For example, both *hole* and *golf* are entered by pressing the sequence 4653. What other words could be produced with the same sequence? Here we use the regular expression «`^[ghi][mno][jlk][def]$`»:

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

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

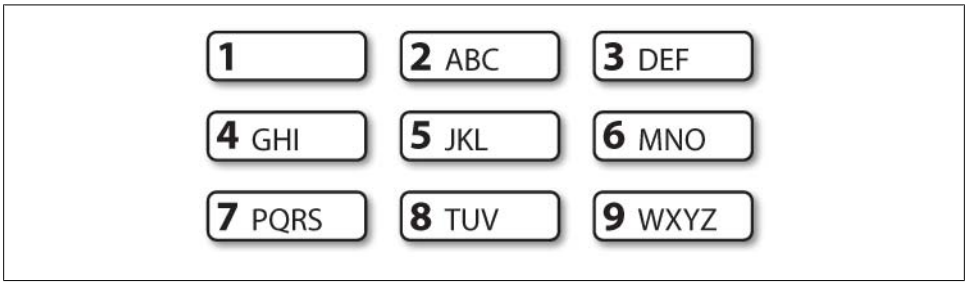


Figure 3-5. T9: Text on 9 keys.

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

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

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

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

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

Here are some more examples of regular expressions being used to find tokens that match a particular pattern, illustrating the use of some new symbols: \, {}, (), and |.

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

Your Turn: Study the previous examples and try to work out what the \, {}, (), and | notations mean before you read on.

You probably worked out that a backslash means that the following character is deprived of its special powers and must literally match a specific character in the word. Thus, while `.` is special, `\.` only matches a period. The braced expressions, like `{3,5}`, specify the number of repeats of the previous item. The pipe character indicates a choice between the material on its left or its right. Parentheses indicate the scope of an operator, and they can be used together with the pipe (or disjunction) symbol like this: «`w(i|e|ai|oo)t`», matching *wit*, *wet*, *wait*, and *woot*. It is instructive to see what happens when you omit the parentheses from the last expression in the example, and search for «`ed|ing$`».

The metacharacters we have seen are summarized in Table 3-3.

Table 3-3. Basic regular expression metacharacters, including wildcards, ranges, and closures

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

To the Python interpreter, a regular expression is just like any other string. If the string contains a backslash followed by particular characters, it will interpret these specially. For example, `\b` would be interpreted as the backspace character. In general, when using regular expressions containing backslash, we should instruct the interpreter not to look inside the string at all, but simply to pass it directly to the `re` library for processing. We do this by prefixing the string with the letter `r`, to indicate that it is a **raw string**. For example, the raw string `r'\band\b'` contains two `\b` symbols that are interpreted by the `re` library as matching word boundaries instead of backspace characters. If you get into the habit of using `r'...'` for regular expressions—as we will do from now on—you will avoid having to think about these complications.

3.5 Useful Applications of Regular Expressions

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

Extracting Word Pieces

The `re.findall()` ("find all") method finds all (non-overlapping) matches of the given regular expression. Let's find all the vowels in a word, then count them:

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

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

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

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

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

Doing More with Word Pieces

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

It is sometimes noted that English text is highly redundant, and it is still easy to read when word-internal vowels are left out. For example, *declaration* becomes *dclrtn*, and *inalienable* becomes *inlnble*, retaining any initial or final vowel sequences. The regular expression in our next example matches initial vowel sequences, final vowel sequences, and all consonants; everything else is ignored. This three-way disjunction is processed left-to-right, and if one of the three parts matches the word, any later parts of the regular expression are ignored. We use `re.findall()` to extract all the matching pieces, and `''.join()` to join them together (see Section 3.9 for more about the join operation).

```
>>> regexp = r'^[AEIOUaeiou]+|[AEIOUaeiou]+$|[^AEIOUaeiou]'
>>> def compress(word):
...     pieces = re.findall(regexp, word)
...     return ''.join(pieces)
...
>>> english_udhr = nltk.corpus.udhr.words('English-Latin1')
>>> print nltk.tokenwrap(compress(w) for w in english_udhr[:75])
Unvrsl Dclrtn of Hmn Rghts Prmble Whrs rcgntn of the inhrnt dgnty and
of the eql and inlnble rghts of all mmbrs of the hmn fmly is the fndtn
of frdm , jstce and pce in the wrld , Whrs dsrgrd and cntmpt fr hmn
rghts hve rsltd in brbrs acts whch hve outrgd the cnscnce of mnknd ,
and the advnt of a wrld in whch hmn bngs shll enjy frdm of spch and
```

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

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

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

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

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

This program processes each word `w` in turn, and for each one, finds every substring that matches the regular expression «`[ptksvr][aeiou]`». In the case of the word *kasuari*, it finds *ka*, *su*, and *ri*. Therefore, the `cv_word_pairs` list will contain `('ka', 'ka`

suari'), ('su', 'kasuari'), and ('ri', 'kasuari'). One further step, using nltk.Index(), converts this into a useful index.

Finding Word Stems

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> def stem(word):
...     regexp = r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$'
...     stem, suffix = re.findall(regexp, word)[0]
...     return stem
...
>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords
... is no basis for a system of government.  Supreme executive power derives from
... a mandate from the masses, not from some farcical aquatic ceremony."""
>>> tokens = nltk.word_tokenize(raw)
>>> [stem(t) for t in tokens]
['DENNIS', ':', 'Listen', ',', 'strange', 'women', 'ly', 'in', 'pond',
'distribut', 'sword', 'i', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',
'.', 'Supreme', 'execut', 'power', 'deriv', 'from', 'a', 'mandate', 'from',
'the', 'mass', ',', 'not', 'from', 'some', 'farcical', 'aquatic', 'ceremony', '.']
```

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

Searching Tokenized Text

You can use a special kind of regular expression for searching across multiple words in a text (where a text is a list of tokens). For example, `"<a> <man>"` finds all instances of *a man* in the text. The angle brackets are used to mark token boundaries, and any whitespace between the angle brackets is ignored (behaviors that are unique to NLTK's `findall()` method for texts). In the following example, we include `<.*>` ❶, which will match any single token, and enclose it in parentheses so only the matched word (e.g., *monied*) and not the matched phrase (e.g., *a monied man*) is produced. The second example finds three-word phrases ending with the word *bro* ❷. The last example finds sequences of three or more words starting with the letter *l* ❸.

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

```
>>> chat.findall(r"<l.*>{3,}") ❸
lol lol lol; lmao lol lol; lol lol lol; la la la la la; la la la; la
la la; lovely lol lol love; lol lol lol.; la la la; la la la
```

Your Turn: Consolidate your understanding of regular expression patterns and substitutions using `nltk.re_show(p, s)`, which annotates the string `s` to show every place where pattern *p* was matched, and `nltk.app.nemo()`, which provides a graphical interface for exploring regular expressions. For more practice, try some of the exercises on regular expressions at the end of this chapter.

It is easy to build search patterns when the linguistic phenomenon we're studying is tied to particular words. In some cases, a little creativity will go a long way. For instance, searching a large text corpus for expressions of the form *x and other ys* allows us to discover hypernyms (see Section 2.5):

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

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

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

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

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

3.6 Normalizing Text

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

```
>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords
... is no basis for a system of government.  Supreme executive power derives from
... a mandate from the masses, not from some farcical aquatic ceremony."""
>>> tokens = nltk.word_tokenize(raw)
```

Stemmers

NLTK includes several off-the-shelf stemmers, and if you ever need a stemmer, you should use one of these in preference to crafting your own using regular expressions, since NLTK's stemmers handle a wide range of irregular cases. The Porter and Lancaster stemmers follow their own rules for stripping affixes. Observe that the Porter stemmer correctly handles the word *lying* (mapping it to *lie*), whereas the Lancaster stemmer does not.

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

Stemming is not a well-defined process, and we typically pick the stemmer that best suits the application we have in mind. The Porter Stemmer is a good choice if you are indexing some texts and want to support search using alternative forms of words (illustrated in Example 3-1, which uses *object-oriented* programming techniques that are outside the scope of this book, string formatting techniques to be covered in Section 3.9, and the `enumerate()` function to be explained in Section 4.2).

Example 3-1. Indexing a text using a stemmer.

```
class IndexedText(object):

    def __init__(self, stemmer, text):
        self._text = text
        self._stemmer = stemmer
```

```
        self._index = nltk.Index((self._stem(word), i)
                                 for (i, word) in enumerate(text))

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

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

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

Lemmatization

The WordNet lemmatizer removes affixes only if the resulting word is in its dictionary. This additional checking process makes the lemmatizer slower than the stemmers just mentioned. Notice that it doesn't handle *lying*, but it converts *women* to *woman*.

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

The WordNet lemmatizer is a good choice if you want to compile the vocabulary of some texts and want a list of valid lemmas (or lexicon headwords).

Another normalization task involves identifying **non-standard words**, including numbers, abbreviations, and dates, and mapping any such tokens to a special vocabulary. For example, every decimal number could be mapped to a single token `0.0`, and every acronym could be mapped to `AAA`. This keeps the vocabulary small and improves the accuracy of many language modeling tasks.

3.7 Regular Expressions for Tokenizing Text

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

Simple Approaches to Tokenization

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

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

We could split this raw text on whitespace using `raw.split()`. To do the same using a regular expression, it is not enough to match any space characters in the string ❶, since this results in tokens that contain a `\n` newline character; instead, we need to match any number of spaces, tabs, or newlines ❷:

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

The regular expression «`[ \t\n]+`» matches one or more spaces, tabs (`\t`), or newlines (`\n`). Other whitespace characters, such as carriage return and form feed, should really be included too. Instead, we will use a built-in `re` abbreviation, `\s`, which means any whitespace character. The second statement in the preceding example can be rewritten as `re.split(r'\s+', raw)`.

Important: Remember to prefix regular expressions with the letter `r` (meaning "raw"), which instructs the Python interpreter to treat the string literally, rather than processing any backslashed characters it contains.

Splitting on whitespace gives us tokens like `'(not'` and `'herself,'`. An alternative is to use the fact that Python provides us with a character class `\w` for word characters, equivalent to `[a-zA-Z0-9_]`. It also defines the complement of this class, `\W`, i.e., all

characters other than letters, digits, or underscore. We can use `\W` in a simple regular expression to split the input on anything *other* than a word character:

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

Observe that this gives us empty strings at the start and the end (to understand why, try doing `'xx'.split('x')`). With `re.findall(r'\w+', raw)`, we get the same tokens, but without the empty strings, using a pattern that matches the words instead of the spaces. Now that we're matching the words, we're in a position to extend the regular expression to cover a wider range of cases. The regular expression «`\w+|\S\w*`» will first try to match any sequence of word characters. If no match is found, it will try to match any *non*-whitespace character (`\S` is the complement of `\s`) followed by further word characters. This means that punctuation is grouped with any following letters (e.g., *'s*) but that sequences of two or more punctuation characters are separated.

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

Let's generalize the `\w+` in the preceding expression to permit word-internal hyphens and apostrophes: «`\w+([-']\w+)*`». This expression means `\w+` followed by zero or more instances of `[-']\w+`; it would match *hot-tempered* and *it's*. (We need to include `?:` in this expression for reasons discussed earlier.) We'll also add a pattern to match quote characters so these are kept separate from the text they enclose.

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

The expression in this example also included «`[-.(]+`», which causes the double hyphen, ellipsis, and open parenthesis to be tokenized separately.

Table 3-4 lists the regular expression character class symbols we have seen in this section, in addition to some other useful symbols.

Table 3-4. Regular expression symbols

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

Symbol	Function
\D	Any non-digit character (equivalent to [^0-9])
\s	Any whitespace character (equivalent to [\t\n\r\f\v]
\S	Any non-whitespace character (equivalent to [^ \t\n\r\f\v])
\w	Any alphanumeric character (equivalent to [a-zA-Z0-9_])
\W	Any non-alphanumeric character (equivalent to [^a-zA-Z0-9_])
\t	The tab character
\n	The newline character

NLTK's Regular Expression Tokenizer

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

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

When using the verbose flag, you can no longer use `' '` to match a space character; use `\s` instead. The `regexp_tokenize()` function has an optional `gaps` parameter. When set to `True`, the regular expression specifies the gaps between tokens, as with `re.split()`.

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

Further Issues with Tokenization

Tokenization turns out to be a far more difficult task than you might have expected. No single solution works well across the board, and we must decide what counts as a token depending on the application domain.

When developing a tokenizer it helps to have access to raw text which has been manually tokenized, in order to compare the output of your tokenizer with high-quality (or

"gold-standard") tokens. The NLTK corpus collection includes a sample of Penn Treebank data, including the raw *Wall Street Journal* text (`nltk.corpus.treebank_raw.raw()`) and the tokenized version (`nltk.corpus.treebank.words()`).

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

3.8 Segmentation

This section discusses more advanced concepts, which you may prefer to skip on the first time through this chapter.

Tokenization is an instance of a more general problem of **segmentation**. In this section, we will look at two other instances of this problem, which use radically different techniques to the ones we have seen so far in this chapter.

Sentence Segmentation

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

```
>>> len(nltk.corpus.brown.words()) / len(nltk.corpus.brown.sents())
20.250994070456922
```

In other cases, the text is available only as a stream of characters. Before tokenizing the text into words, we need to segment it into sentences. NLTK facilitates this by including the Punkt sentence segmenter (Kiss & Strunk, 2006). Here is an example of its use in segmenting the text of a novel. (Note that if the segmenter's internal data has been updated by the time you read this, you will see different output.)

```
>>> sent_tokenizer=nltk.data.load('tokenizers/punkt/english.pickle')
>>> text = nltk.corpus.gutenberg.raw('chesterton-thursday.txt')
>>> sents = sent_tokenizer.tokenize(text)
>>> pprint.pprint(sents[171:181])
['"Nonsense!',
 '" said Gregory, who was very rational when anyone else\nattempted paradox.',
 '"Why do all the clerks and navvies in the\nrailway trains look so sad and tired,...',
 'I will\ntell you.',
 'It is because they know that the train is going right.',
 'It\nis because they know that whatever place they have taken a ticket\nfor that ...',
 'It is because after they have\npassed Sloane Square they know that the next stat...',
 'Oh, their wild rapture!',
 'oh,\ntheir eyes like stars and their souls again in Eden, if the next\nstation w...'
 '"\n\n"It is you who are unpoetical," replied the poet Syme.']
```

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

Sentence segmentation is difficult because a period is used to mark abbreviations, and some periods simultaneously mark an abbreviation and terminate a sentence, as often happens with acronyms like *U.S.A.*

For another approach to sentence segmentation, see Section 6.2.

Word Segmentation

For some writing systems, tokenizing text is made more difficult by the fact that there is no visual representation of word boundaries. For example, in Chinese, the three-character string: 爱国人 (ai4 "love" [verb], guo3 "country", ren2 "person") could be tokenized as 爱国 / 人, "country-loving person," or as 爱 / 国人, "love country-person."

A similar problem arises in the processing of spoken language, where the hearer must segment a continuous speech stream into individual words. A particularly challenging version of this problem arises when we don't know the words in advance. This is the problem faced by a language learner, such as a child hearing utterances from a parent. Consider the following artificial example, where word boundaries have been removed:

(1) a. doyouseethekitty
    b. seethedoggy
    c. doyoulikethekitty
    d. likethedoggy

Our first challenge is simply to represent the problem: we need to find a way to separate text content from the segmentation. We can do this by annotating each character with a boolean value to indicate whether or not a word-break appears after the character (an idea that will be used heavily for "chunking" in Chapter 7). Let's assume that the learner is given the utterance breaks, since these often correspond to extended pauses. Here is a possible representation, including the initial and target segmentations:

```
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "000000000000000100000000001000000000000000001000000000000"
>>> seg2 = "010010010010000100100100001010010001001000010001010000"
```

Observe that the segmentation strings consist of zeros and ones. They are one character shorter than the source text, since a text of length *n* can be broken up in only *n*–1 places. The `segment()` function in Example 3-2 demonstrates that we can get back to the original segmented text from its representation.

Example 3-2. Reconstruct segmented text from string representation: `seg1` *and* `seg2` *represent the initial and final segmentations of some hypothetical child-directed speech; the* `segment()` *function can use them to reproduce the segmented text.*

```
def segment(text, segs):
    words = []
    last = 0
    for i in range(len(segs)):
        if segs[i] == '1':
            words.append(text[last:i+1])
            last = i+1
    words.append(text[last:])
    return words
```

```
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "000000000000000100000000001000000000000000100000000000"
>>> seg2 = "010010010010000100100100001010010001001000010001010000"
>>> segment(text, seg1)
['doyouseethekitty', 'seethedoggy', 'doyoulikethekitty', 'likethedoggy']
>>> segment(text, seg2)
['do', 'you', 'see', 'the', 'kitty', 'see', 'the', 'doggy', 'do', 'you',
 'like', 'the', kitty', 'like', 'the', 'doggy']
```

Now the segmentation task becomes a search problem: find the bit string that causes the text string to be correctly segmented into words. We assume the learner is acquiring words and storing them in an internal lexicon. Given a suitable lexicon, it is possible to reconstruct the source text as a sequence of lexical items. Following (Brent & Cartwright, 1995), we can define an **objective function**, a scoring function whose value we will try to optimize, based on the size of the lexicon and the amount of information needed to reconstruct the source text from the lexicon. We illustrate this in Figure 3-6.

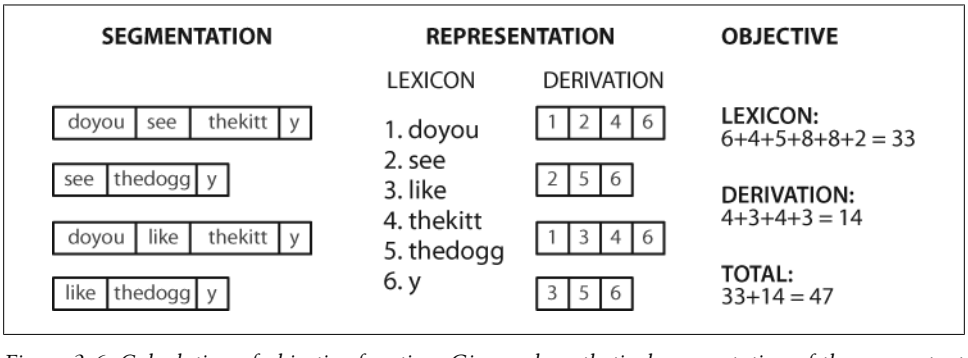


Figure 3-6. Calculation of objective function: Given a hypothetical segmentation of the source text (on the left), derive a lexicon and a derivation table that permit the source text to be reconstructed, then total up the number of characters used by each lexical item (including a boundary marker) and each derivation, to serve as a score of the quality of the segmentation; smaller values of the score indicate a better segmentation.

It is a simple matter to implement this objective function, as shown in Example 3-3.

Example 3-3. Computing the cost of storing the lexicon and reconstructing the source text.

```
def evaluate(text, segs):
    words = segment(text, segs)
    text_size = len(words)
    lexicon_size = len(' '.join(list(set(words))))
    return text_size + lexicon_size
```

```
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "0000000000000001000000000010000000000000000100000000000"
>>> seg2 = "0100100100100001001001000010100100010010000100010010000"
>>> seg3 = "0000100100000011001000000110000100010000001100010000001"
>>> segment(text, seg3)
['doyou', 'see', 'thekitt', 'y', 'see', 'thedogg', 'y', 'doyou', 'like',
 'thekitt', 'y', 'like', 'thedogg', 'y']
>>> evaluate(text, seg3)
46
>>> evaluate(text, seg2)
47
>>> evaluate(text, seg1)
63
```

The final step is to search for the pattern of zeros and ones that maximizes this objective function, shown in Example 3-4. Notice that the best segmentation includes "words" like *thekitty*, since there's not enough evidence in the data to split this any further.

Example 3-4. Non-deterministic search using simulated annealing: Begin searching with phrase segmentations only; randomly perturb the zeros and ones proportional to the "temperature"; with each iteration the temperature is lowered and the perturbation of boundaries is reduced.

```
from random import randint

def flip(segs, pos):
    return segs[:pos] + str(1-int(segs[pos])) + segs[pos+1:]

def flip_n(segs, n):
    for i in range(n):
        segs = flip(segs, randint(0,len(segs)-1))
    return segs

def anneal(text, segs, iterations, cooling_rate):
    temperature = float(len(segs))
    while temperature > 0.5:
        best_segs, best = segs, evaluate(text, segs)
        for i in range(iterations):
            guess = flip_n(segs, int(round(temperature)))
            score = evaluate(text, guess)
            if score < best:
                best, best_segs = score, guess
        score, segs = best, best_segs
        temperature = temperature / cooling_rate
        print evaluate(text, segs), segment(text, segs)
```

```
    print
    return segs

>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "0000000000000001000000000010000000000000000100000000000"
>>> anneal(text, seg1, 5000, 1.2)
60 ['doyouseetheki', 'tty', 'see', 'thedoggy', 'doyouliketh', 'ekittylike', 'thedoggy']
58 ['doy', 'ouseetheki', 'ttysee', 'thedoggy', 'doy', 'o', 'ulikethekittylike', 'thedoggy']
56 ['doyou', 'seetheki', 'ttysee', 'thedoggy', 'doyou', 'liketh', 'ekittylike', 'thedoggy']
54 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'likethekittylike', 'thedoggy']
53 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 'thedoggy']
51 ['doyou', 'seethekittysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 'thedoggy']
42 ['doyou', 'see', 'thekitty', 'see', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 'thedoggy']
'000010010000000100100000001000010001000000010010000000'
```

With enough data, it is possible to automatically segment text into words with a reasonable degree of accuracy. Such methods can be applied to tokenization for writing systems that don't have any visual representation of word boundaries.

3.9 Formatting: From Lists to Strings

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

From Lists to Strings

The simplest kind of structured object we use for text processing is lists of words. When we want to output these to a display or a file, we must convert these lists into strings. To do this in Python we use the `join()` method, and specify the string to be used as the "glue":

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

So `' '.join(silly)` means: take all the items in `silly` and concatenate them as one big string, using `' '` as a spacer between the items. I.e., `join()` is a method of the string that you want to use as the glue. (Many people find this notation for `join()` counter-intuitive.) The `join()` method only works on a list of strings—what we have been calling a text—a complex type that enjoys some privileges in Python.

Strings and Formats

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

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

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

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

Formatted output typically contains a combination of variables and pre-specified strings. For example, given a frequency distribution `fdist`, we could do:

```
>>> fdist = nltk.FreqDist(['dog', 'cat', 'dog', 'cat', 'dog', 'snake', 'dog', 'cat'])
>>> for word in fdist:
...     print word, '->', fdist[word], ';',
dog -> 4 ; cat -> 3 ; snake -> 1 ;
```

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

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

To understand what is going on here, let's test out the string formatting expression on its own. (By now this will be your usual method of exploring new syntax.)

```
>>> '%s->%d;' % ('cat', 3)
'cat->3;'
>>> '%s->%d;' % 'cat'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format string
```

The special symbols `%s` and `%d` are placeholders for strings and (decimal) integers. We can embed these inside a string, then use the `%` operator to combine them. Let's unpack this code further, in order to see this behavior up close:

```
>>> '%s->' % 'cat'
'cat->'
>>> '%d' % 3
'3'
>>> 'I want a %s right now' % 'coffee'
'I want a coffee right now'
```

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

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

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

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

The `%s` and `%d` symbols are called **conversion specifiers**. They start with the `%` character and end with a conversion character such as `s` (for string) or `d` (for decimal integer) The string containing conversion specifiers is called a **format string**. We combine a format string with the `%` operator and a tuple of values to create a complete string formatting expression.

Lining Things Up

So far our formatting strings generated output of arbitrary width on the page (or screen), such as `%s` and `%d`. We can specify a width as well, such as `%6s`, producing a string that is padded to width 6. It is right-justified by default ❶, but we can include a minus sign to make it left-justified ❷. In case we don't know in advance how wide a displayed value should be, the width value can be replaced with a star in the formatting string, then specified using a variable ❸.

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

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

```
>>> count, total = 3205, 9375
>>> "accuracy for %d words: %2.4f%%" % (total, 100 * count / total)
'accuracy for 9375 words: 34.1867%'
```

An important use of formatting strings is for tabulating data. Recall that in Section 2.1 we saw data being tabulated from a conditional frequency distribution. Let's perform the tabulation ourselves, exercising full control of headings and column widths, as shown in Example 3-5. Note the clear separation between the language processing work, and the tabulation of results.

Example 3-5. Frequency of modals in different sections of the Brown Corpus.

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

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

Recall from the listing in Example 3-1 that we used a formatting string `"%*s"`. This allows us to specify the width of a field using a variable.

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

We could use this to automatically customize the column to be just wide enough to accommodate all the words, using `width = max(len(w) for w in words)`. Remember that the comma at the end of print statements adds an extra space, and this is sufficient to prevent the column headings from running into each other.

Writing Results to a File

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

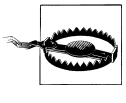
```
>>> output_file = open('output.txt', 'w')
>>> words = set(nltk.corpus.genesis.words('english-kjv.txt'))
>>> for word in sorted(words):
...     output_file.write(word + "\n")
```

Your Turn: What is the effect of appending `\n` to each string before we write it to the file? If you're using a Windows machine, you may want to use `word + "\r\n"` instead. What happens if we do

```
output_file.write(word)
```

When we write non-text data to a file, we must convert it to a string first. We can do this conversion using formatting strings, as we saw earlier. Let's write the total number of words to our file, before closing it.

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



Caution!

You should avoid filenames that contain space characters, such as *output file.txt*, or that are identical except for case distinctions, e.g., *Output.txt* and *output.TXT*.

Text Wrapping

When the output of our program is text-like, instead of tabular, it will usually be necessary to wrap it so that it can be displayed conveniently. Consider the following output, which overflows its line, and which uses a complicated `print` statement:

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

We can take care of line wrapping with the help of Python's `textwrap` module. For maximum clarity we will separate each step onto its own line:

```
>>> from textwrap import fill
>>> format = '%s (%d),'
```

```
>>> pieces = [format % (word, len(word)) for word in saying]
>>> output = ' '.join(pieces)
>>> wrapped = fill(output)
>>> print wrapped
After (5), all (3), is (2), said (4), and (3), done (4), , (1), more
(4), is (2), said (4), than (4), done (4), . (1),
```

Notice that there is a linebreak between `more` and its following number. If we wanted to avoid this, we could redefine the formatting string so that it contained no spaces (e.g., `'%s_(%d),'`), then instead of printing the value of `wrapped`, we could print `wrapped.replace('_', ' ')`.

3.10 Summary

- In this book we view a text as a list of words. A "raw text" is a potentially long string containing words and whitespace formatting, and is how we typically store and visualize a text.
- A string is specified in Python using single or double quotes: `'Monty Python'`, `"Monty Python"`.
- The characters of a string are accessed using indexes, counting from zero: `'Monty Python'[0]` gives the value `M`. The length of a string is found using `len()`.
- Substrings are accessed using slice notation: `'Monty Python'[1:5]` gives the value `onty`. If the start index is omitted, the substring begins at the start of the string; if the end index is omitted, the slice continues to the end of the string.
- Strings can be split into lists: `'Monty Python'.split()` gives `['Monty', 'Python']`. Lists can be joined into strings: `'/'.join(['Monty', 'Python'])` gives `'Monty/Python'`.
- We can read text from a file `f` using `text = open(f).read()`. We can read text from a URL `u` using `text = urlopen(u).read()`. We can iterate over the lines of a text file using `for line in open(f)`.
- Texts found on the Web may contain unwanted material (such as headers, footers, and markup), that need to be removed before we do any linguistic processing.
- Tokenization is the segmentation of a text into basic units—or tokens—such as words and punctuation. Tokenization based on whitespace is inadequate for many applications because it bundles punctuation together with words. NLTK provides an off-the-shelf tokenizer `nltk.word_tokenize()`.
- Lemmatization is a process that maps the various forms of a word (such as *appeared*, *appears*) to the canonical or citation form of the word, also known as the lexeme or lemma (e.g., *appear*).
- Regular expressions are a powerful and flexible method of specifying patterns. Once we have imported the `re` module, we can use `re.findall()` to find all substrings in a string that match a pattern.