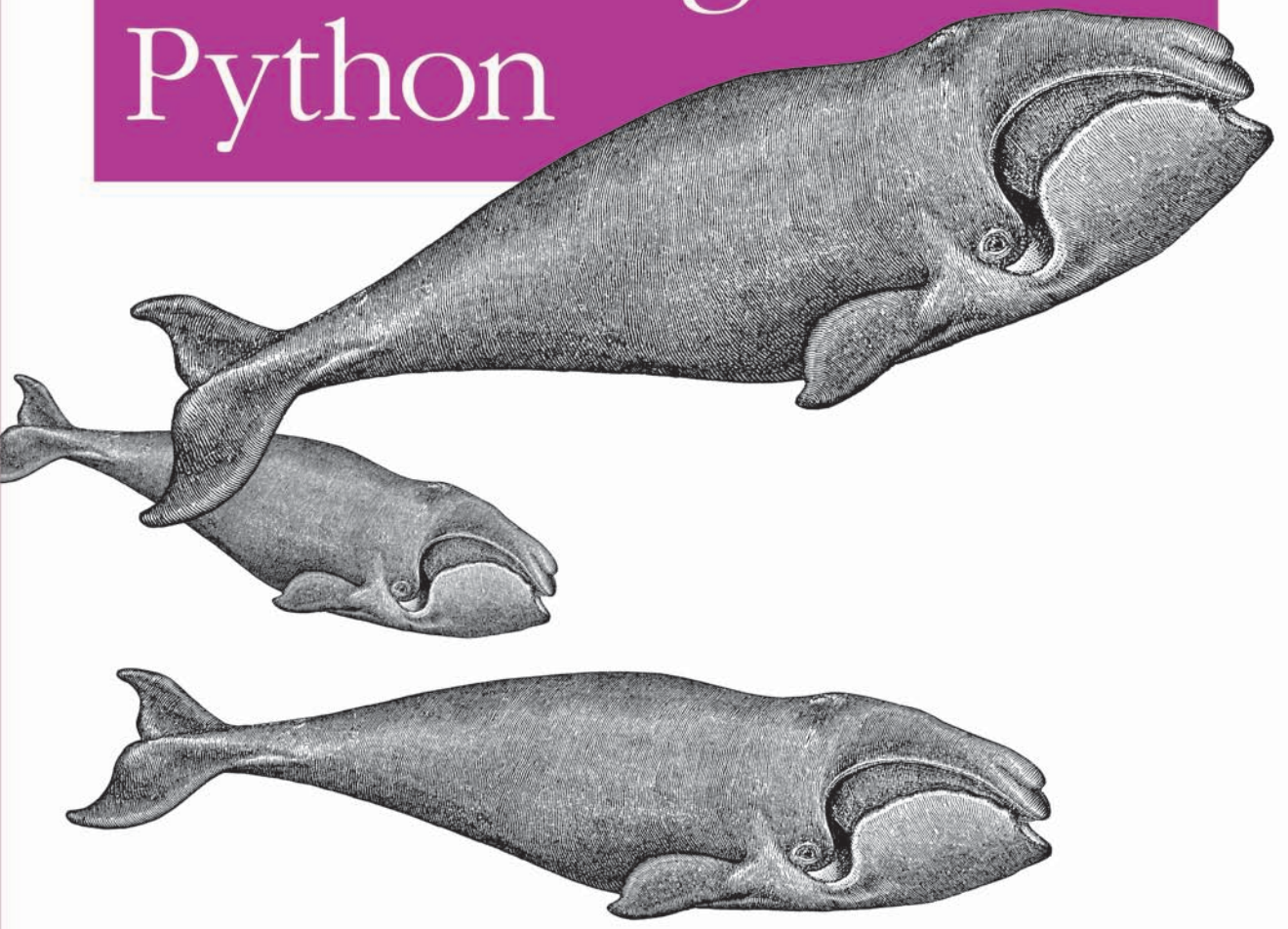
Analyzing Text with the Natural Language Toolkit

Natural Language Processing with Python



O'REILLY®

Steven Bird, Ewan Klein & Edward Loper

O'REILLY®

Natural Language Processing with Python

This book offers a highly accessible introduction to natural language processing, the field that supports a variety of language technologies, from predictive text and email filtering to automatic summarization and translation. With it, you'll learn how to write Python programs that work with large collections of unstructured text. You'll access richly annotated datasets using a comprehensive range of linguistic data structures, and you'll understand the main algorithms for analyzing the content and structure of written communication.

Packed with examples and exercises, *Natural Language Processing with Python* will help you:

- **Extract information from unstructured text, either to guess the topic or identify "named entities"**
- **Analyze linguistic structure in text, including parsing and semantic analysis**
- **Access popular linguistic databases, including WordNet and treebanks**
- **Integrate techniques drawn from fields as diverse as linguistics and artificial intelligence**

This book will help you gain practical skills in natural language processing using the Python programming language and the Natural Language Toolkit (NLTK) open source library. If you're interested in developing web applications, analyzing multilingual news sources, or documenting endangered languages—or if you're simply curious to have a programmer's perspective on how human language works—you'll find *Natural Language Processing with Python* both fascinating and immensely useful.

"Rarely does a book tackle such a difficult computing subject with such a clear approach and with such beautifully clean code.... This is the book from which to learn natural language processing."

—Ken Getz,
Senior Consultant,
MCW Technologies

Steven Bird is Associate Professor in the Department of Computer Science and Software Engineering at the University of Melbourne, and Senior Research Associate in the Linguistic Data Consortium at the University of Pennsylvania.

Ewan Klein is Professor of Language Technology in the School of Informatics at the University of Edinburgh.

Edward Loper recently completed a Ph.D. on machine learning for natural language processing at the University of Pennsylvania, and is now a researcher at BBN Technologies in Boston.

oreilly.com

US $44.99 **CAN $56.99**

ISBN: 978-0-596-51649-9

Natural Language Processing with Python

Natural Language Processing with Python

Steven Bird, Ewan Klein, and Edward Loper

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Natural Language Processing with Python
by Steven Bird, Ewan Klein, and Edward Loper

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Julie Steele
Production Editor: Loranah Dimant
Copyeditor: Genevieve d'Entremont
Proofreader: Loranah Dimant

Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
June 2009: First Edition.

ISBN: 978-0-596-51649-9

[M]

1244726609

Table of Contents

Preface

This is a book about Natural Language Processing. By "natural language" we mean a language that is used for everyday communication by humans; languages such as English, Hindi, or Portuguese. In contrast to artificial languages such as programming languages and mathematical notations, natural languages have evolved as they pass from generation to generation, and are hard to pin down with explicit rules. We will take Natural Language Processing—or NLP for short—in a wide sense to cover any kind of computer manipulation of natural language. At one extreme, it could be as simple as counting word frequencies to compare different writing styles. At the other extreme, NLP involves "understanding" complete human utterances, at least to the extent of being able to give useful responses to them.

Technologies based on NLP are becoming increasingly widespread. For example, phones and handheld computers support predictive text and handwriting recognition; web search engines give access to information locked up in unstructured text; machine translation allows us to retrieve texts written in Chinese and read them in Spanish. By providing more natural human-machine interfaces, and more sophisticated access to stored information, language processing has come to play a central role in the multilingual information society.

This book provides a highly accessible introduction to the field of NLP. It can be used for individual study or as the textbook for a course on natural language processing or computational linguistics, or as a supplement to courses in artificial intelligence, text mining, or corpus linguistics. The book is intensely practical, containing hundreds of fully worked examples and graded exercises.

The book is based on the Python programming language together with an open source library called the *Natural Language Toolkit* (NLTK). NLTK includes extensive software, data, and documentation, all freely downloadable from *http://www.nltk.org/*. Distributions are provided for Windows, Macintosh, and Unix platforms. We strongly encourage you to download Python and NLTK, and try out the examples and exercises along the way.

Audience

NLP is important for scientific, economic, social, and cultural reasons. NLP is experiencing rapid growth as its theories and methods are deployed in a variety of new language technologies. For this reason it is important for a wide range of people to have a working knowledge of NLP. Within industry, this includes people in human-computer interaction, business information analysis, and web software development. Within academia, it includes people in areas from humanities computing and corpus linguistics through to computer science and artificial intelligence. (To many people in academia, NLP is known by the name of "Computational Linguistics.")

This book is intended for a diverse range of people who want to learn how to write programs that analyze written language, regardless of previous programming experience:

New to programming?
: The early chapters of the book are suitable for readers with no prior knowledge of programming, so long as you aren't afraid to tackle new concepts and develop new computing skills. The book is full of examples that you can copy and try for yourself, together with hundreds of graded exercises. If you need a more general introduction to Python, see the list of Python resources at *http://docs.python.org/*.

New to Python?
: Experienced programmers can quickly learn enough Python using this book to get immersed in natural language processing. All relevant Python features are carefully explained and exemplified, and you will quickly come to appreciate Python's suitability for this application area. The language index will help you locate relevant discussions in the book.

Already dreaming in Python?
: Skim the Python examples and dig into the interesting language analysis material that starts in Chapter 1. You'll soon be applying your skills to this fascinating domain.

Emphasis

This book is a **practical** introduction to NLP. You will learn by example, write real programs, and grasp the value of being able to test an idea through implementation. If you haven't learned already, this book will teach you **programming**. Unlike other programming books, we provide extensive illustrations and exercises from NLP. The approach we have taken is also **principled**, in that we cover the theoretical underpinnings and don't shy away from careful linguistic and computational analysis. We have tried to be **pragmatic** in striking a balance between theory and application, identifying the connections and the tensions. Finally, we recognize that you won't get through this unless it is also **pleasurable**, so we have tried to include many applications and examples that are interesting and entertaining, and sometimes whimsical.

Note that this book is not a reference work. Its coverage of Python and NLP is selective, and presented in a tutorial style. For reference material, please consult the substantial quantity of searchable resources available at *http://python.org/* and *http://www.nltk.org/*.

This book is not an advanced computer science text. The content ranges from introductory to intermediate, and is directed at readers who want to learn how to analyze text using Python and the Natural Language Toolkit. To learn about advanced algorithms implemented in NLTK, you can examine the Python code linked from *http://www.nltk.org/*, and consult the other materials cited in this book.

What You Will Learn

By digging into the material presented here, you will learn:

- How simple programs can help you manipulate and analyze language data, and how to write these programs
- How key concepts from NLP and linguistics are used to describe and analyze language
- How data structures and algorithms are used in NLP
- How language data is stored in standard formats, and how data can be used to evaluate the performance of NLP techniques

Depending on your background, and your motivation for being interested in NLP, you will gain different kinds of skills and knowledge from this book, as set out in Table P-1.

Table P-1. Skills and knowledge to be gained from reading this book, depending on readers' goals and background

Goals	Background in arts and humanities	Background in science and engineering
Language analysis	Manipulating large corpora, exploring linguistic models, and testing empirical claims.	Using techniques in data modeling, data mining, and knowledge discovery to analyze natural language.
Language technology	Building robust systems to perform linguistic tasks with technological applications.	Using linguistic algorithms and data structures in robust language processing software.

Organization

The early chapters are organized in order of conceptual difficulty, starting with a practical introduction to language processing that shows how to explore interesting bodies of text using tiny Python programs (Chapters 1–3). This is followed by a chapter on structured programming (Chapter 4) that consolidates the programming topics scattered across the preceding chapters. After this, the pace picks up, and we move on to a series of chapters covering fundamental topics in language processing: tagging, classification, and information extraction (Chapters 5–7). The next three chapters look at

ways to parse a sentence, recognize its syntactic structure, and construct representations of meaning (Chapters 8–10). The final chapter is devoted to linguistic data and how it can be managed effectively (Chapter 11). The book concludes with an Afterword, briefly discussing the past and future of the field.

Within each chapter, we switch between different styles of presentation. In one style, natural language is the driver. We analyze language, explore linguistic concepts, and use programming examples to support the discussion. We often employ Python constructs that have not been introduced systematically, so you can see their purpose before delving into the details of how and why they work. This is just like learning idiomatic expressions in a foreign language: you're able to buy a nice pastry without first having learned the intricacies of question formation. In the other style of presentation, the programming language will be the driver. We'll analyze programs, explore algorithms, and the linguistic examples will play a supporting role.

Each chapter ends with a series of graded exercises, which are useful for consolidating the material. The exercises are graded according to the following scheme: ○ is for easy exercises that involve minor modifications to supplied code samples or other simple activities; ◑ is for intermediate exercises that explore an aspect of the material in more depth, requiring careful analysis and design; ● is for difficult, open-ended tasks that will challenge your understanding of the material and force you to think independently (readers new to programming should skip these).

Each chapter has a further reading section and an online "extras" section at *http://www.nltk.org/*, with pointers to more advanced materials and online resources. Online versions of all the code examples are also available there.

Why Python?

Python is a simple yet powerful programming language with excellent functionality for processing linguistic data. Python can be downloaded for free from *http://www.python.org/*. Installers are available for all platforms.

Here is a five-line Python program that processes *file.txt* and prints all the words ending in `ing`:

```
>>> for line in open("file.txt"):
...     for word in line.split():
...         if word.endswith('ing'):
...             print word
```

This program illustrates some of the main features of Python. First, whitespace is used to *nest* lines of code; thus the line starting with `if` falls inside the scope of the previous line starting with `for`; this ensures that the `ing` test is performed for each word. Second, Python is *object-oriented*; each variable is an entity that has certain defined attributes and methods. For example, the value of the variable `line` is more than a sequence of characters. It is a string object that has a "method" (or operation) called `split()` that

we can use to break a line into its words. To apply a method to an object, we write the object name, followed by a period, followed by the method name, i.e., `line.split()`. Third, methods have *arguments* expressed inside parentheses. For instance, in the example, `word.endswith('ing')` had the argument `'ing'` to indicate that we wanted words ending with *ing* and not something else. Finally—and most importantly—Python is highly readable, so much so that it is fairly easy to guess what this program does even if you have never written a program before.

We chose Python because it has a shallow learning curve, its syntax and semantics are transparent, and it has good string-handling functionality. As an interpreted language, Python facilitates interactive exploration. As an object-oriented language, Python permits data and methods to be encapsulated and re-used easily. As a dynamic language, Python permits attributes to be added to objects on the fly, and permits variables to be typed dynamically, facilitating rapid development. Python comes with an extensive standard library, including components for graphical programming, numerical processing, and web connectivity.

Python is heavily used in industry, scientific research, and education around the world. Python is often praised for the way it facilitates productivity, quality, and maintainability of software. A collection of Python success stories is posted at *http://www.python.org/about/success/*.

NLTK defines an infrastructure that can be used to build NLP programs in Python. It provides basic classes for representing data relevant to natural language processing; standard interfaces for performing tasks such as part-of-speech tagging, syntactic parsing, and text classification; and standard implementations for each task that can be combined to solve complex problems.

NLTK comes with extensive documentation. In addition to this book, the website at *http://www.nltk.org/* provides API documentation that covers every module, class, and function in the toolkit, specifying parameters and giving examples of usage. The website also provides many HOWTOs with extensive examples and test cases, intended for users, developers, and instructors.

Software Requirements

To get the most out of this book, you should install several free software packages. Current download pointers and instructions are available at *http://www.nltk.org/*.

Python
: The material presented in this book assumes that you are using Python version 2.4 or 2.5. We are committed to porting NLTK to Python 3.0 once the libraries that NLTK depends on have been ported.

NLTK
: The code examples in this book use NLTK version 2.0. Subsequent releases of NLTK will be backward-compatible.

NLTK-Data
: This contains the linguistic corpora that are analyzed and processed in the book.

NumPy (recommended)
: This is a scientific computing library with support for multidimensional arrays and linear algebra, required for certain probability, tagging, clustering, and classification tasks.

Matplotlib (recommended)
: This is a 2D plotting library for data visualization, and is used in some of the book's code samples that produce line graphs and bar charts.

NetworkX (optional)
: This is a library for storing and manipulating network structures consisting of nodes and edges. For visualizing semantic networks, also install the Graphviz library.

Prover9 (optional)
: This is an automated theorem prover for first-order and equational logic, used to support inference in language processing.

Natural Language Toolkit (NLTK)

NLTK was originally created in 2001 as part of a computational linguistics course in the Department of Computer and Information Science at the University of Pennsylvania. Since then it has been developed and expanded with the help of dozens of contributors. It has now been adopted in courses in dozens of universities, and serves as the basis of many research projects. Table P-2 lists the most important NLTK modules.

Table P-2. Language processing tasks and corresponding NLTK modules with examples of functionality

Language processing task	NLTK modules	Functionality
Accessing corpora	nltk.corpus	Standardized interfaces to corpora and lexicons
String processing	nltk.tokenize, nltk.stem	Tokenizers, sentence tokenizers, stemmers
Collocation discovery	nltk.collocations	t-test, chi-squared, point-wise mutual information
Part-of-speech tagging	nltk.tag	n-gram, backoff, Brill, HMM, TnT
Classification	nltk.classify, nltk.cluster	Decision tree, maximum entropy, naive Bayes, EM, k-means
Chunking	nltk.chunk	Regular expression, n-gram, named entity
Parsing	nltk.parse	Chart, feature-based, unification, probabilistic, dependency
Semantic interpretation	nltk.sem, nltk.inference	Lambda calculus, first-order logic, model checking
Evaluation metrics	nltk.metrics	Precision, recall, agreement coefficients
Probability and estimation	nltk.probability	Frequency distributions, smoothed probability distributions
Applications	nltk.app, nltk.chat	Graphical concordancer, parsers, WordNet browser, chatbots

Language processing task	NLTK modules	Functionality
Linguistic fieldwork	nltk.toolbox	Manipulate data in SIL Toolbox format

NLTK was designed with four primary goals in mind:

Simplicity
: To provide an intuitive framework along with substantial building blocks, giving users a practical knowledge of NLP without getting bogged down in the tedious house-keeping usually associated with processing annotated language data

Consistency
: To provide a uniform framework with consistent interfaces and data structures, and easily guessable method names

Extensibility
: To provide a structure into which new software modules can be easily accommodated, including alternative implementations and competing approaches to the same task

Modularity
: To provide components that can be used independently without needing to understand the rest of the toolkit

Contrasting with these goals are three non-requirements—potentially useful qualities that we have deliberately avoided. First, while the toolkit provides a wide range of functions, it is not encyclopedic; it is a toolkit, not a system, and it will continue to evolve with the field of NLP. Second, while the toolkit is efficient enough to support meaningful tasks, it is not highly optimized for runtime performance; such optimizations often involve more complex algorithms, or implementations in lower-level programming languages such as C or C++. This would make the software less readable and more difficult to install. Third, we have tried to avoid clever programming tricks, since we believe that clear implementations are preferable to ingenious yet indecipherable ones.

For Instructors

Natural Language Processing is often taught within the confines of a single-semester course at the advanced undergraduate level or postgraduate level. Many instructors have found that it is difficult to cover both the theoretical and practical sides of the subject in such a short span of time. Some courses focus on theory to the exclusion of practical exercises, and deprive students of the challenge and excitement of writing programs to automatically process language. Other courses are simply designed to teach programming for linguists, and do not manage to cover any significant NLP content. NLTK was originally developed to address this problem, making it feasible to cover a substantial amount of theory and practice within a single-semester course, even if students have no prior programming experience.

A significant fraction of any NLP syllabus deals with algorithms and data structures. On their own these can be rather dry, but NLTK brings them to life with the help of interactive graphical user interfaces that make it possible to view algorithms step-by-step. Most NLTK components include a demonstration that performs an interesting task without requiring any special input from the user. An effective way to deliver the materials is through interactive presentation of the examples in this book, entering them in a Python session, observing what they do, and modifying them to explore some empirical or theoretical issue.

This book contains hundreds of exercises that can be used as the basis for student assignments. The simplest exercises involve modifying a supplied program fragment in a specified way in order to answer a concrete question. At the other end of the spectrum, NLTK provides a flexible framework for graduate-level research projects, with standard implementations of all the basic data structures and algorithms, interfaces to dozens of widely used datasets (corpora), and a flexible and extensible architecture. Additional support for teaching using NLTK is available on the NLTK website.

We believe this book is unique in providing a comprehensive framework for students to learn about NLP in the context of learning to program. What sets these materials apart is the tight coupling of the chapters and exercises with NLTK, giving students—even those with no prior programming experience—a practical introduction to NLP. After completing these materials, students will be ready to attempt one of the more advanced textbooks, such as *Speech and Language Processing*, by Jurafsky and Martin (Prentice Hall, 2008).

This book presents programming concepts in an unusual order, beginning with a non-trivial data type—lists of strings—then introducing non-trivial control structures such as comprehensions and conditionals. These idioms permit us to do useful language processing from the start. Once this motivation is in place, we return to a systematic presentation of fundamental concepts such as strings, loops, files, and so forth. In this way, we cover the same ground as more conventional approaches, without expecting readers to be interested in the programming language for its own sake.

Two possible course plans are illustrated in Table P-3. The first one presumes an arts/humanities audience, whereas the second one presumes a science/engineering audience. Other course plans could cover the first five chapters, then devote the remaining time to a single area, such as text classification (Chapters 6 and 7), syntax (Chapters 8 and 9), semantics (Chapter 10), or linguistic data management (Chapter 11).

Table P-3. Suggested course plans; approximate number of lectures per chapter

Chapter	Arts and Humanities	Science and Engineering
Chapter 1, *Language Processing and Python*	2–4	2
Chapter 2, *Accessing Text Corpora and Lexical Resources*	2–4	2
Chapter 3, *Processing Raw Text*	2–4	2
Chapter 4, *Writing Structured Programs*	2–4	1–2

Chapter	Arts and Humanities	Science and Engineering
Chapter 5, *Categorizing and Tagging Words*	2–4	2–4
Chapter 6, *Learning to Classify Text*	0–2	2–4
Chapter 7, *Extracting Information from Text*	2	2–4
Chapter 8, *Analyzing Sentence Structure*	2–4	2–4
Chapter 9, *Building Feature-Based Grammars*	2–4	1–4
Chapter 10, *Analyzing the Meaning of Sentences*	1–2	1–4
Chapter 11, *Managing Linguistic Data*	1–2	1–4
Total	18–36	18–36

Conventions Used in This Book

The following typographical conventions are used in this book:

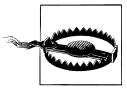
Bold
: Indicates new terms.

Italic
: Used within paragraphs to refer to linguistic examples, the names of texts, and URLs; also used for filenames and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, statements, and keywords; also used for program names.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context; also used for metavariables within program code examples.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example,

writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Natural Language Processing with Python*, by Steven Bird, Ewan Klein, and Edward Loper. Copyright 2009 Steven Bird, Ewan Klein, and Edward Loper, 978-0-596-51649-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596516499

The authors provide additional materials for each chapter via the NLTK website at:

http://www.nltk.org/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

The authors are indebted to the following people for feedback on earlier drafts of this book: Doug Arnold, Michaela Atterer, Greg Aumann, Kenneth Beesley, Steven Bethard, Ondrej Bojar, Chris Cieri, Robin Cooper, Grev Corbett, James Curran, Dan Garrette, Jean Mark Gawron, Doug Hellmann, Nitin Indurkhya, Mark Liberman, Peter Ljunglöf, Stefan Müller, Robin Munn, Joel Nothman, Adam Przepiorkowski, Brandon Rhodes, Stuart Robinson, Jussi Salmela, Kyle Schlansker, Rob Speer, and Richard Sproat. We are thankful to many students and colleagues for their comments on the class materials that evolved into these chapters, including participants at NLP and linguistics summer schools in Brazil, India, and the USA. This book would not exist without the members of the `nltk-dev` developer community, named on the NLTK website, who have given so freely of their time and expertise in building and extending NLTK.

We are grateful to the U.S. National Science Foundation, the Linguistic Data Consortium, an Edward Clarence Dyason Fellowship, and the Universities of Pennsylvania, Edinburgh, and Melbourne for supporting our work on this book.

We thank Julie Steele, Abby Fox, Loranah Dimant, and the rest of the O'Reilly team, for organizing comprehensive reviews of our drafts from people across the NLP and Python communities, for cheerfully customizing O'Reilly's production tools to accommodate our needs, and for meticulous copyediting work.

Finally, we owe a huge debt of gratitude to our partners, Kay, Mimo, and Jee, for their love, patience, and support over the many years that we worked on this book. We hope that our children—Andrew, Alison, Kirsten, Leonie, and Maaike—catch our enthusiasm for language and computation from these pages.

Royalties

Royalties from the sale of this book are being used to support the development of the Natural Language Toolkit.

Figure P-1. Edward Loper, Ewan Klein, and Steven Bird, Stanford, July 2007

CHAPTER 1

Language Processing and Python

It is easy to get our hands on millions of words of text. What can we do with it, assuming we can write some simple programs? In this chapter, we'll address the following questions:

1. What can we achieve by combining simple programming techniques with large quantities of text?
2. How can we automatically extract key words and phrases that sum up the style and content of a text?
3. What tools and techniques does the Python programming language provide for such work?
4. What are some of the interesting challenges of natural language processing?

This chapter is divided into sections that skip between two quite different styles. In the "computing with language" sections, we will take on some linguistically motivated programming tasks without necessarily explaining how they work. In the "closer look at Python" sections we will systematically review key programming concepts. We'll flag the two styles in the section titles, but later chapters will mix both styles without being so up-front about it. We hope this style of introduction gives you an authentic taste of what will come later, while covering a range of elementary concepts in linguistics and computer science. If you have basic familiarity with both areas, you can skip to Section 1.5; we will repeat any important points in later chapters, and if you miss anything you can easily consult the online reference material at *http://www.nltk.org/*. If the material is completely new to you, this chapter will raise more questions than it answers, questions that are addressed in the rest of this book.

1.1 Computing with Language: Texts and Words

We're all very familiar with text, since we read and write it every day. Here we will treat text as *raw data* for the programs we write, programs that manipulate and analyze it in a variety of interesting ways. But before we can do this, we have to get started with the Python interpreter.

Getting Started with Python

One of the friendly things about Python is that it allows you to type directly into the interactive **interpreter**—the program that will be running your Python programs. You can access the Python interpreter using a simple graphical interface called the Interactive DeveLopment Environment (IDLE). On a Mac you can find this under Applications→MacPython, and on Windows under All Programs→Python. Under Unix you can run Python from the shell by typing `idle` (if this is not installed, try typing `python`). The interpreter will print a blurb about your Python version; simply check that you are running Python 2.4 or 2.5 (here it is 2.5.1):

```
Python 2.5.1 (r251:54863, Apr 15 2008, 22:57:26)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you are unable to run the Python interpreter, you probably don't have Python installed correctly. Please visit *http://python.org/* for detailed instructions.

The `>>>` prompt indicates that the Python interpreter is now waiting for input. When copying examples from this book, don't type the "`>>>`" yourself. Now, let's begin by using Python as a calculator:

```
>>> 1 + 5 * 2 - 3
8
>>>
```

Once the interpreter has finished calculating the answer and displaying it, the prompt reappears. This means the Python interpreter is waiting for another instruction.

Your Turn: Enter a few more expressions of your own. You can use asterisk (*) for multiplication and slash (/) for division, and parentheses for bracketing expressions. Note that division doesn't always behave as you might expect—it does integer division (with rounding of fractions downwards) when you type `1/3` and "floating-point" (or decimal) division when you type `1.0/3.0`. In order to get the expected behavior of division (standard in Python 3.0), you need to type: `from __future__ import division`.

The preceding examples demonstrate how you can work interactively with the Python interpreter, experimenting with various expressions in the language to see what they do. Now let's try a non-sensical expression to see how the interpreter handles it:

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

This produced a **syntax error**. In Python, it doesn't make sense to end an instruction with a plus sign. The Python interpreter indicates the line where the problem occurred (line 1 of `<stdin>`, which stands for "standard input").

Now that we can use the Python interpreter, we're ready to start working with language data.

Getting Started with NLTK

Before going further you should install NLTK, downloadable for free from *http://www.nltk.org/*. Follow the instructions there to download the version required for your platform.

Once you've installed NLTK, start up the Python interpreter as before, and install the data required for the book by typing the following two commands at the Python prompt, then selecting the `book` collection as shown in Figure 1-1.

```
>>> import nltk
>>> nltk.download()
```

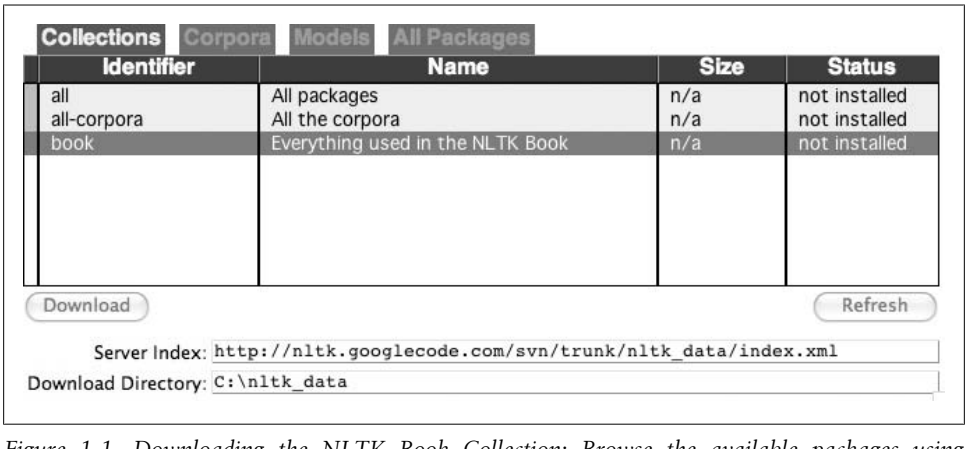


Figure 1-1. Downloading the NLTK Book Collection: Browse the available packages using `nltk.download()`. *The* ***Collections*** *tab on the downloader shows how the packages are grouped into sets, and you should select the line labeled* ***book*** *to obtain all data required for the examples and exercises in this book. It consists of about 30 compressed files requiring about 100Mb disk space. The full collection of data (i.e.,* ***all*** *in the downloader) is about five times this size (at the time of writing) and continues to expand.*

Once the data is downloaded to your machine, you can load some of it using the Python interpreter. The first step is to type a special command at the Python prompt, which

tells the interpreter to load some texts for us to explore: `from nltk.book import *`. This says "from NLTK's `book` module, load all items." The `book` module contains all the data you will need as you read this chapter. After printing a welcome message, it loads the text of several books (this will take a few seconds). Here's the command again, together with the output that you will see. Take care to get spelling and punctuation right, and remember that you don't type the `>>>`.

```
>>> from nltk.book import *
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
>>>
```

Any time we want to find out about these texts, we just have to enter their names at the Python prompt:

```
>>> text1
<Text: Moby Dick by Herman Melville 1851>
>>> text2
<Text: Sense and Sensibility by Jane Austen 1811>
>>>
```

Now that we can use the Python interpreter, and have some data to work with, we're ready to get started.

Searching Text

There are many ways to examine the context of a text apart from simply reading it. A concordance view shows us every occurrence of a given word, together with some context. Here we look up the word *monstrous* in *Moby Dick* by entering `text1` followed by a period, then the term `concordance`, and then placing `"monstrous"` in parentheses:

```
>>> text1.concordance("monstrous")
Building index...
Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came towards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r
ll over with a heathenish array of monstrous clubs and spears . Some were thick
d as you gazed , and wondered what monstrous cannibal and savage could ever hav
that has survived the flood ; most monstrous and most mountainous ! That Himmal
they might scout at Moby Dick as a monstrous fable , or still worse and more de
th of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall ere l
ing Scenes . In connexion with the monstrous pictures of whales , I am strongly
ere to enter upon those still more monstrous stories of them which are to be fo
```

```
ght have been rummaged out of this monstrous cabinet there is no telling . But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up dead u
>>>
```

Your Turn: Try searching for other words; to save re-typing, you might be able to use up-arrow, Ctrl-up-arrow, or Alt-p to access the previous command and modify the word being searched. You can also try searches on some of the other texts we have included. For example, search *Sense and Sensibility* for the word *affection*, using `text2.concordance("affection")`. Search the book of Genesis to find out how long some people lived, using: `text3.concordance("lived")`. You could look at `text4`, the *Inaugural Address Corpus*, to see examples of English going back to 1789, and search for words like *nation*, *terror*, *god* to see how these words have been used differently over time. We've also included `text5`, the *NPS Chat Corpus*: search this for unconventional words like *im*, *ur*, *lol*. (Note that this corpus is uncensored!)

Once you've spent a little while examining these texts, we hope you have a new sense of the richness and diversity of language. In the next chapter you will learn how to access a broader range of text, including text in languages other than English.

A concordance permits us to see words in context. For example, we saw that *monstrous* occurred in contexts such as *the ___ pictures* and *the ___ size*. What other words appear in a similar range of contexts? We can find out by appending the term `similar` to the name of the text in question, then inserting the relevant word in parentheses:

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

Observe that we get different results for different texts. Austen uses this word quite differently from Melville; for her, *monstrous* has positive connotations, and sometimes functions as an intensifier like the word *very*.

The term `common_contexts` allows us to examine just the contexts that are shared by two or more words, such as *monstrous* and *very*. We have to enclose these words by square brackets as well as parentheses, and separate them with a comma:

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

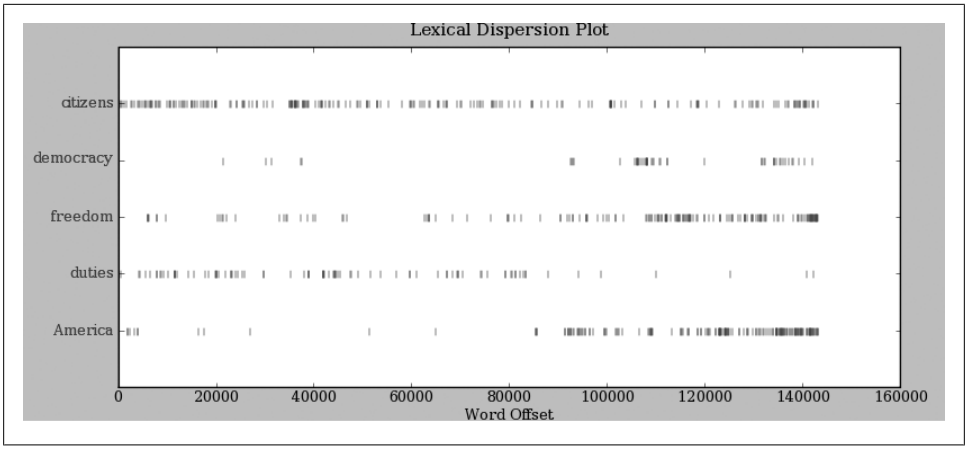


Figure 1-2. Lexical dispersion plot for words in U.S. Presidential Inaugural Addresses: This can be used to investigate changes in language use over time.

Your Turn: Pick another pair of words and compare their usage in two different texts, using the `similar()` and `common_contexts()` functions.

It is one thing to automatically detect that a particular word occurs in a text, and to display some words that appear in the same context. However, we can also determine the *location* of a word in the text: how many words from the beginning it appears. This positional information can be displayed using a **dispersion plot**. Each stripe represents an instance of a word, and each row represents the entire text. In Figure 1-2 we see some striking patterns of word usage over the last 220 years (in an artificial text constructed by joining the texts of the Inaugural Address Corpus end-to-end). You can produce this plot as shown below. You might like to try more words (e.g., *liberty*, *constitution*) and different texts. Can you predict the dispersion of a word before you view it? As before, take care to get the quotes, commas, brackets, and parentheses exactly right.

```
>>> text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])
>>>
```

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

Now, just for fun, let's try generating some random text in the various styles we have just seen. To do this, we type the name of the text followed by the term `generate`. (We need to include the parentheses, but there's nothing that goes between them.)

```
>>> text3.generate()
In the beginning of his brother is a hairy man , whose top may reach
unto heaven ; and ye shall sow the land of Egypt there was no bread in
all that he was taken out of the month , upon the earth . So shall thy
wages be ? And they made their father ; and Isaac was old , and kissed
him : and Laban with his cattle in the midst of the hands of Esau thy
first born , and Phichol the chief butler unto his son Isaac , she
>>>
```

Note that the first time you run this command, it is slow because it gathers statistics about word sequences. Each time you run it, you will get different output text. Now try generating random text in the style of an inaugural address or an Internet chat room. Although the text is random, it reuses common words and phrases from the source text and gives us a sense of its style and content. (What is lacking in this randomly generated text?)

When `generate` produces its output, punctuation is split off from the preceding word. While this is not correct formatting for English text, we do it to make clear that words and punctuation are independent of one another. You will learn more about this in Chapter 3.

Counting Vocabulary

The most obvious fact about texts that emerges from the preceding examples is that they differ in the vocabulary they use. In this section, we will see how to use the computer to count the words in a text in a variety of useful ways. As before, you will jump right in and experiment with the Python interpreter, even though you may not have studied Python systematically yet. Test your understanding by modifying the examples, and trying the exercises at the end of the chapter.

Let's begin by finding out the length of a text from start to finish, in terms of the words and punctuation symbols that appear. We use the term `len` to get the length of something, which we'll apply here to the book of Genesis:

```
>>> len(text3)
44764
>>>
```

So Genesis has 44,764 words and punctuation symbols, or "tokens." A **token** is the technical name for a sequence of characters—such as `hairy`, `his`, or `:)`—that we want to treat as a group. When we count the number of tokens in a text, say, the phrase *to be or not to be*, we are counting occurrences of these sequences. Thus, in our example phrase there are two occurrences of *to*, two of *be*, and one each of *or* and *not*. But there are only four distinct vocabulary items in this phrase. How many distinct words does the book of Genesis contain? To work this out in Python, we have to pose the question slightly differently. The vocabulary of a text is just the *set* of tokens that it uses, since in a set, all duplicates are collapsed together. In Python we can obtain the vocabulary

items of `text3` with the command: `set(text3)`. When you do this, many screens of words will fly past. Now try the following:

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

By wrapping `sorted()` around the Python expression `set(text3)` ❶, we obtain a sorted list of vocabulary items, beginning with various punctuation symbols and continuing with words starting with *A*. All capitalized words precede lowercase words. We discover the size of the vocabulary indirectly, by asking for the number of items in the set, and again we can use `len` to obtain this number ❷. Although it has 44,764 tokens, this book has only 2,789 distinct words, or "word types." A **word type** is the form or spelling of the word independently of its specific occurrences in a text—that is, the word considered as a unique item of vocabulary. Our count of 2,789 items will include punctuation symbols, so we will generally call these unique items **types** instead of word types.

Now, let's calculate a measure of the lexical richness of the text. The next example shows us that each word is used 16 times on average (we need to make sure Python uses floating-point division):

```
>>> from __future__ import division
>>> len(text3) / len(set(text3))
16.050197203298673
>>>
```

Next, let's focus on particular words. We can count how often a word occurs in a text, and compute what percentage of the text is taken up by a specific word:

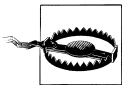
```
>>> text3.count("smote")
5
>>> 100 * text4.count('a') / len(text4)
1.4643016433938312
>>>
```

Your Turn: How many times does the word *lol* appear in `text5`? How much is this as a percentage of the total number of words in this text?

You may want to repeat such calculations on several texts, but it is tedious to keep retyping the formula. Instead, you can come up with your own name for a task, like "lexical_diversity" or "percentage", and associate it with a block of code. Now you only have to type a short name instead of one or more complete lines of Python code, and you can reuse it as often as you like. The block of code that does a task for us is

called a **function**, and we define a short name for our function with the keyword `def`. The next example shows how to define two new functions, `lexical_diversity()` and `percentage()`:

```
>>> def lexical_diversity(text): ❶
...     return len(text) / len(set(text)) ❷
...
>>> def percentage(count, total): ❸
...     return 100 * count / total
...
```



Caution!

The Python interpreter changes the prompt from `>>>` to `...` after encountering the colon at the end of the first line. The `...` prompt indicates that Python expects an **indented code block** to appear next. It is up to you to do the indentation, by typing four spaces or hitting the Tab key. To finish the indented block, just enter a blank line.

In the definition of `lexical diversity()` ❶, we specify a **parameter** labeled `text`. This parameter is a "placeholder" for the actual text whose lexical diversity we want to compute, and reoccurs in the block of code that will run when the function is used, in line ❷. Similarly, `percentage()` is defined to take two parameters, labeled `count` and `total` ❸.

Once Python knows that `lexical_diversity()` and `percentage()` are the names for specific blocks of code, we can go ahead and use these functions:

```
>>> lexical_diversity(text3)
16.050197203298673
>>> lexical_diversity(text5)
7.4200461589185629
>>> percentage(4, 5)
80.0
>>> percentage(text4.count('a'), len(text4))
1.4643016433938312
>>>
```

To recap, we use or **call** a function such as `lexical_diversity()` by typing its name, followed by an open parenthesis, the name of the text, and then a close parenthesis. These parentheses will show up often; their role is to separate the name of a task—such as `lexical_diversity()`—from the data that the task is to be performed on—such as `text3`. The data value that we place in the parentheses when we call a function is an **argument** to the function.

You have already encountered several functions in this chapter, such as `len()`, `set()`, and `sorted()`. By convention, we will always add an empty pair of parentheses after a function name, as in `len()`, just to make clear that what we are talking about is a function rather than some other kind of Python expression. Functions are an important concept in programming, and we only mention them at the outset to give newcomers

a sense of the power and creativity of programming. Don't worry if you find it a bit confusing right now.

Later we'll see how to use functions when tabulating data, as in Table 1-1. Each row of the table will involve the same computation but with different data, and we'll do this repetitive work using a function.

Table 1-1. Lexical diversity of various genres in the Brown Corpus

Genre	Tokens	Types	Lexical diversity
skill and hobbies	82345	11935	6.9
humor	21695	5017	4.3
fiction: science	14470	3233	4.5
press: reportage	100554	14394	7.0
fiction: romance	70022	8452	8.3
religion	39399	6373	6.2

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

You've seen some important elements of the Python programming language. Let's take a few moments to review them systematically.

Lists

What is a text? At one level, it is a sequence of symbols on a page such as this one. At another level, it is a sequence of chapters, made up of a sequence of sections, where each section is a sequence of paragraphs, and so on. However, for our purposes, we will think of a text as nothing more than a sequence of words and punctuation. Here's how we represent text in Python, in this case the opening sentence of *Moby Dick*:

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>>
```

After the prompt we've given a name we made up, `sent1`, followed by the equals sign, and then some quoted words, separated with commas, and surrounded with brackets. This bracketed material is known as a **list** in Python: it is how we store a text. We can inspect it by typing the name ❶. We can ask for its length ❷. We can even apply our own `lexical_diversity()` function to it ❸.

```
>>> sent1 ❶
['Call', 'me', 'Ishmael', '.']
>>> len(sent1) ❷
4
>>> lexical_diversity(sent1) ❸
1.0
>>>
```

Some more lists have been defined for you, one for the opening sentence of each of our texts, `sent2` ... `sent9`. We inspect two of them here; you can see the rest for yourself using the Python interpreter (if you get an error saying that `sent2` is not defined, you need to first type `from nltk.book import *`).

```
>>> sent2
['The', 'family', 'of', 'Dashwood', 'had', 'long',
'been', 'settled', 'in', 'Sussex', '.']
>>> sent3
['In', 'the', 'beginning', 'God', 'created', 'the',
'heaven', 'and', 'the', 'earth', '.']
>>>
```

Your Turn: Make up a few sentences of your own, by typing a name, equals sign, and a list of words, like this: `ex1 = ['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']`. Repeat some of the other Python operations we saw earlier in Section 1.1, e.g., `sorted(ex1)`, `len(set(ex1))`, `ex1.count('the')`.

A pleasant surprise is that we can use Python's addition operator on lists. Adding two lists ❶ creates a new list with everything from the first list, followed by everything from the second list:

```
>>> ['Monty', 'Python'] + ['and', 'the', 'Holy', 'Grail'] ❶
['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']
```

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

We don't have to literally type the lists either; we can use short names that refer to predefined lists.

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

What if we want to add a single item to a list? This is known as **appending**. When we `append()` to a list, the list itself is updated as a result of the operation.

```
>>> sent1.append("Some")
>>> sent1
['Call', 'me', 'Ishmael', '.', 'Some']
>>>
```

Indexing Lists

As we have seen, a text in Python is a list of words, represented using a combination of brackets and quotes. Just as with an ordinary page of text, we can count up the total number of words in `text1` with `len(text1)`, and count the occurrences in a text of a particular word—say, *heaven*—using `text1.count('heaven')`.

With some patience, we can pick out the 1st, 173rd, or even 14,278th word in a printed text. Analogously, we can identify the elements of a Python list by their order of occurrence in the list. The number that represents this position is the item's **index**. We instruct Python to show us the item that occurs at an index such as `173` in a text by writing the name of the text followed by the index inside square brackets:

```
>>> text4[173]
'awaken'
>>>
```

We can do the converse; given a word, find the index of when it first occurs:

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

Indexes are a common way to access the words of a text, or, more generally, the elements of any list. Python permits us to access sublists as well, extracting manageable pieces of language from large texts, a technique known as **slicing**.

```
>>> text5[16715:16735]
['U86', 'thats', 'why', 'something', 'like', 'gamefly', 'is', 'so', 'good',
'because', 'you', 'can', 'actually', 'play', 'a', 'full', 'game', 'without',
'buying', 'it']
>>> text6[1600:1625]
['We', "'", 're', 'an', 'anarcho', '-', 'syndicalist', 'commune', '.', 'We',
'take', 'it', 'in', 'turns', 'to', 'act', 'as', 'a', 'sort', 'of', 'executive',
'officer', 'for', 'the', 'week']
>>>
```

Indexes have some subtleties, and we'll explore these with the help of an artificial sentence:

```
>>> sent = ['word1', 'word2', 'word3', 'word4', 'word5',
...         'word6', 'word7', 'word8', 'word9', 'word10']
>>> sent[0]
'word1'
>>> sent[9]
'word10'
>>>
```

Notice that our indexes start from zero: `sent` element zero, written `sent[0]`, is the first word, `'word1'`, whereas `sent` element 9 is `'word10'`. The reason is simple: the moment Python accesses the content of a list from the computer's memory, it is already at the first element; we have to tell it how many elements forward to go. Thus, zero steps forward leaves it at the first element.

This practice of counting from zero is initially confusing, but typical of modern programming languages. You'll quickly get the hang of it if you've mastered the system of counting centuries where 19XY is a year in the 20th century, or if you live in a country where the floors of a building are numbered from 1, and so walking up *n-1* flights of stairs takes you to level *n*.

Now, if we accidentally use an index that is too large, we get an error:

```
>>> sent[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

This time it is not a syntax error, because the program fragment is syntactically correct. Instead, it is a **runtime error**, and it produces a `Traceback` message that shows the context of the error, followed by the name of the error, `IndexError`, and a brief explanation.

Let's take a closer look at slicing, using our artificial sentence again. Here we verify that the slice `5:8` includes `sent` elements at indexes 5, 6, and 7:

```
>>> sent[5:8]
['word6', 'word7', 'word8']
>>> sent[5]
'word6'
>>> sent[6]
'word7'
>>> sent[7]
'word8'
>>>
```

By convention, `m:n` means elements *m...n-1*. As the next example shows, we can omit the first number if the slice begins at the start of the list ❶, and we can omit the second number if the slice goes to the end ❷:

```
>>> sent[:3] ❶
['word1', 'word2', 'word3']
>>> text2[141525:] ❷
['among', 'the', 'merits', 'and', 'the', 'happiness', 'of', 'Elinor', 'and', 'Marianne',
',', 'let', 'it', 'not', 'be', 'ranked', 'as', 'the', 'least', 'considerable', ',',
'that', 'though', 'sisters', ',', 'and', 'living', 'almost', 'within', 'sight', 'of',
'each', 'other', ',', 'they', 'could', 'live', 'without', 'disagreement', 'between',
'themselves', ',', 'or', 'producing', 'coolness', 'between', 'their', 'husbands', '.',
'THE', 'END']
>>>
```

We can modify an element of a list by assigning to one of its index values. In the next example, we put `sent[0]` on the left of the equals sign ❶. We can also replace an entire slice with new material ❷. A consequence of this last change is that the list only has four elements, and accessing a later value generates an error ❸.

```
>>> sent[0] = 'First' ❶
>>> sent[9] = 'Last'
>>> len(sent)
10
>>> sent[1:9] = ['Second', 'Third'] ❷
>>> sent
['First', 'Second', 'Third', 'Last']
>>> sent[9] ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

Your Turn: Take a few minutes to define a sentence of your own and modify individual words and groups of words (slices) using the same methods used earlier. Check your understanding by trying the exercises on lists at the end of this chapter.

Variables

From the start of Section 1.1, you have had access to texts called `text1`, `text2`, and so on. It saved a lot of typing to be able to refer to a 250,000-word book with a short name like this! In general, we can make up names for anything we care to calculate. We did this ourselves in the previous sections, e.g., defining a **variable** `sent1`, as follows:

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>>
```

Such lines have the form: *variable = expression*. Python will evaluate the expression, and save its result to the variable. This process is called **assignment**. It does not generate any output; you have to type the variable on a line of its own to inspect its contents. The equals sign is slightly misleading, since information is moving from the right side to the left. It might help to think of it as a left-arrow. The name of the variable can be anything you like, e.g., `my_sent`, `sentence`, `xyzzy`. It must start with a letter, and can include numbers and underscores. Here are some examples of variables and assignments:

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

Remember that capitalized words appear before lowercase words in sorted lists.

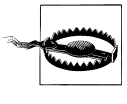
Notice in the previous example that we split the definition of `my_sent` over two lines. Python expressions can be split across multiple lines, so long as this happens within any kind of brackets. Python uses the `...` prompt to indicate that more input is expected. It doesn't matter how much indentation is used in these continuation lines, but some indentation usually makes them easier to read.

It is good to choose meaningful variable names to remind you—and to help anyone else who reads your Python code—what your code is meant to do. Python does not try to make sense of the names; it blindly follows your instructions, and does not object if you do something confusing, such as `one = 'two'` or `two = 3`. The only restriction is that a variable name cannot be any of Python's reserved words, such as `def`, `if`, `not`, and `import`. If you use a reserved word, Python will produce a syntax error:

```
>>> not = 'Camelot'
File "<stdin>", line 1
    not = 'Camelot'
        ^
SyntaxError: invalid syntax
>>>
```

We will often use variables to hold intermediate steps of a computation, especially when this makes the code easier to follow. Thus `len(set(text1))` could also be written:

```
>>> vocab = set(text1)
>>> vocab_size = len(vocab)
>>> vocab_size
19317
>>>
```



Caution!

Take care with your choice of names (or **identifiers**) for Python variables. First, you should start the name with a letter, optionally followed by digits (`0` to `9`) or letters. Thus, `abc23` is fine, but `23abc` will cause a syntax error. Names are case-sensitive, which means that `myVar` and `myvar` are distinct variables. Variable names cannot contain whitespace, but you can separate words using an underscore, e.g., `my_var`. Be careful not to insert a hyphen instead of an underscore: `my-var` is wrong, since Python interprets the `-` as a minus sign.

Strings

Some of the methods we used to access the elements of a list also work with individual words, or **strings**. For example, we can assign a string to a variable ❶, index a string ❷, and slice a string ❸.

```
>>> name = 'Monty' ❶
>>> name[0] ❷
'M'
>>> name[:4] ❸
'Mont'
>>>
```

We can also perform multiplication and addition with strings:

```
>>> name * 2
'MontyMonty'
>>> name + '!'
'Monty!'
>>>
```

We can join the words of a list to make a single string, or split a string into a list, as follows:

```
>>> ' '.join(['Monty', 'Python'])
'Monty Python'
>>> 'Monty Python'.split()
['Monty', 'Python']
>>>
```

We will come back to the topic of strings in Chapter 3. For the time being, we have two important building blocks—lists and strings—and are ready to get back to some language analysis.

1.3 Computing with Language: Simple Statistics

Let's return to our exploration of the ways we can bring our computational resources to bear on large quantities of text. We began this discussion in Section 1.1, and saw how to search for words in context, how to compile the vocabulary of a text, how to generate random text in the same style, and so on.

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

Before continuing further, you might like to check your understanding of the last section by predicting the output of the following code. You can use the interpreter to check whether you got it right. If you're not sure how to do this task, it would be a good idea to review the previous section before continuing further.

```
>>> saying = ['After', 'all', 'is', 'said', 'and', 'done',
...           'more', 'is', 'said', 'than', 'done']
>>> tokens = set(saying)
>>> tokens = sorted(tokens)
>>> tokens[-2:]
what output do you expect here?
>>>
```

Frequency Distributions

How can we automatically identify the words of a text that are most informative about the topic and genre of the text? Imagine how you might go about finding the 50 most frequent words of a book. One method would be to keep a tally for each vocabulary item, like that shown in Figure 1-3. The tally would need thousands of rows, and it would be an exceedingly laborious process—so laborious that we would rather assign the task to a machine.

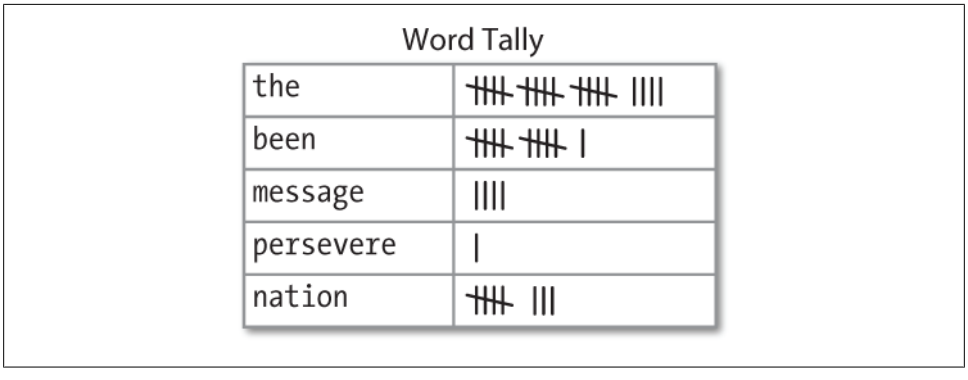


Figure 1-3. Counting words appearing in a text (a frequency distribution).

The table in Figure 1-3 is known as a **frequency distribution** , and it tells us the frequency of each vocabulary item in the text. (In general, it could count any kind of observable event.) It is a "distribution" since it tells us how the total number of word tokens in the text are distributed across the vocabulary items. Since we often need frequency distributions in language processing, NLTK provides built-in support for them. Let's use a `FreqDist` to find the 50 most frequent words of *Moby Dick*. Try to work out what is going on here, then read the explanation that follows.

```
>>> fdist1 = FreqDist(text1) ❶
>>> fdist1 ❷
<FreqDist with 260819 outcomes>
>>> vocabulary1 = fdist1.keys() ❸
>>> vocabulary1[:50] ❹
[',', 'the', '.', 'of', 'and', 'a', 'to', ';', 'in', 'that', "'", '-',
'his', 'it', 'I', 's', 'is', 'he', 'with', 'was', 'as', '"', 'all', 'for',
'this', '!', 'at', 'by', 'but', 'not', '--', 'him', 'from', 'be', 'on',
'so', 'whale', 'one', 'you', 'had', 'have', 'there', 'But', 'or', 'were',
'now', 'which', '?', 'me', 'like']
>>> fdist1['whale']
906
>>>
```

When we first invoke `FreqDist`, we pass the name of the text as an argument ❶. We can inspect the total number of words ("outcomes") that have been counted up ❷—260,819 in the case of *Moby Dick*. The expression `keys()` gives us a list of all the distinct types in the text ❸, and we can look at the first 50 of these by slicing the list ❹.

Your Turn: Try the preceding frequency distribution example for yourself, for `text2`. Be careful to use the correct parentheses and uppercase letters. If you get an error message `NameError: name 'FreqDist' is not defined`, you need to start your work with `from nltk.book import *`.

Do any words produced in the last example help us grasp the topic or genre of this text? Only one word, *whale*, is slightly informative! It occurs over 900 times. The rest of the words tell us nothing about the text; they're just English "plumbing." What proportion of the text is taken up with such words? We can generate a cumulative frequency plot for these words, using `fdist1.plot(50, cumulative=True)`, to produce the graph in Figure 1-4. These 50 words account for nearly half the book!

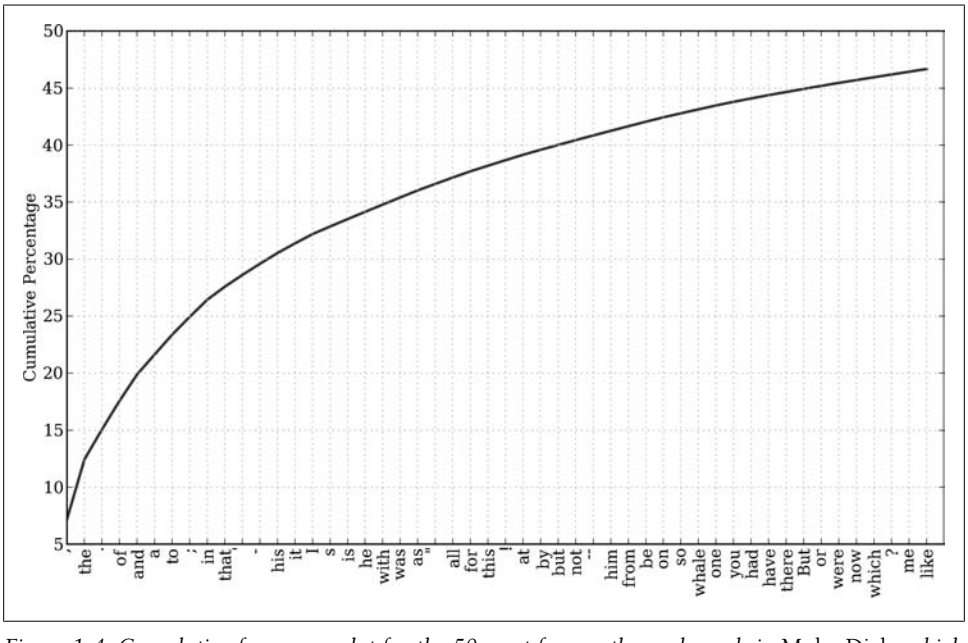


Figure 1-4. Cumulative frequency plot for the 50 most frequently used words in Moby Dick, *which account for nearly half of the tokens.*

If the frequent words don't help us, how about the words that occur once only, the so-called **hapaxes**? View them by typing `fdist1.hapaxes()`. This list contains *lexicographer*, *cetological*, *contraband*, *expostulations*, and about 9,000 others. It seems that there are too many rare words, and without seeing the context we probably can't guess what half of the hapaxes mean in any case! Since neither frequent nor infrequent words help, we need to try something else.

Fine-Grained Selection of Words

Next, let's look at the *long* words of a text; perhaps these will be more characteristic and informative. For this we adapt some notation from set theory. We would like to find the words from the vocabulary of the text that are more than 15 characters long. Let's call this property P, so that $P(w)$ is true if and only if w is more than 15 characters long. Now we can express the words of interest using mathematical set notation as shown in (1a). This means "the set of all w such that w is an element of V (the vocabulary) and w has property P."

(1) a. $\{w \mid w \in V \ \& \ P(w)\}$

b. `[w for w in V if p(w)]`

The corresponding Python expression is given in (1b). (Note that it produces a list, not a set, which means that duplicates are possible.) Observe how similar the two notations are. Let's go one more step and write executable Python code:

```
>>> V = set(text1)
>>> long_words = [w for w in V if len(w) > 15]
>>> sorted(long_words)
['CIRCUMNAVIGATION', 'Physiognomically', 'apprehensiveness', 'cannibalistically',
'characteristically', 'circumnavigating', 'circumnavigation', 'circumnavigations',
'comprehensiveness', 'hermaphroditical', 'indiscriminately', 'indispensableness',
'irresistibleness', 'physiognomically', 'preternaturalness', 'responsibilities',
'simultaneousness', 'subterraneousness', 'supernaturalness', 'superstitiousness',
'uncomfortableness', 'uncompromisedness', 'undiscriminating', 'uninterpenetratingly']
>>>
```

For each word `w` in the vocabulary `V`, we check whether `len(w)` is greater than 15; all other words will be ignored. We will discuss this syntax more carefully later.

Your Turn: Try out the previous statements in the Python interpreter, and experiment with changing the text and changing the length condition. Does it make an difference to your results if you change the variable names, e.g., using `[word for word in vocab if ...]`?

Let's return to our task of finding words that characterize a text. Notice that the long words in `text4` reflect its national focus—*constitutionally*, *transcontinental*—whereas those in `text5` reflect its informal content: *boooooooooooglyyyyyy* and *yuuuuuuuuuuuummmmmmmmmmmm*. Have we succeeded in automatically extracting words that typify a text? Well, these very long words are often hapaxes (i.e., unique) and perhaps it would be better to find *frequently occurring* long words. This seems promising since it eliminates frequent short words (e.g., *the*) and infrequent long words (e.g., *antiphilosophists*). Here are all words from the chat corpus that are longer than seven characters, that occur more than seven times:

```
>>> fdist5 = FreqDist(text5)
>>> sorted([w for w in set(text5) if len(w) > 7 and fdist5[w] > 7])
['#14-19teens', '#talkcity_adults', '((((((((((', '........', 'Question',
'actually', 'anything', 'computer', 'cute.-ass', 'everyone', 'football',
'innocent', 'listening', 'remember', 'seriously', 'something', 'together',
'tomorrow', 'watching']
>>>
```

Notice how we have used two conditions: `len(w) > 7` ensures that the words are longer than seven letters, and `fdist5[w] > 7` ensures that these words occur more than seven times. At last we have managed to automatically identify the frequently occurring content-bearing words of the text. It is a modest but important milestone: a tiny piece of code, processing tens of thousands of words, produces some informative output.

Collocations and Bigrams

A **collocation** is a sequence of words that occur together unusually often. Thus *red wine* is a collocation, whereas *the wine* is not. A characteristic of collocations is that they are resistant to substitution with words that have similar senses; for example, *maroon wine* sounds very odd.

To get a handle on collocations, we start off by extracting from a text a list of word pairs, also known as **bigrams**. This is easily accomplished with the function `bigrams()`:

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

Here we see that the pair of words *than-done* is a bigram, and we write it in Python as `('than', 'done')`. Now, collocations are essentially just frequent bigrams, except that we want to pay more attention to the cases that involve rare words. In particular, we want to find bigrams that occur more often than we would expect based on the frequency of individual words. The `collocations()` function does this for us (we will see how it works later):

```
>>> text4.collocations()
Building collocations list
United States; fellow citizens; years ago; Federal Government; General
Government; American people; Vice President; Almighty God; Fellow
citizens; Chief Magistrate; Chief Justice; God bless; Indian tribes;
public debt; foreign nations; political parties; State governments;
```

```
National Government; United Nations; public money
>>> text8.collocations()
Building collocations list
medium build; social drinker; quiet nights; long term; age open;
financially secure; fun times; similar interests; Age open; poss
rship; single mum; permanent relationship; slim build; seeks lady;
Late 30s; Photo pls; Vibrant personality; European background; ASIAN
LADY; country drives
>>>
```

The collocations that emerge are very specific to the genre of the texts. In order to find *red wine* as a collocation, we would need to process a much larger body of text.

Counting Other Things

Counting words is useful, but we can count other things too. For example, we can look at the distribution of word lengths in a text, by creating a `FreqDist` out of a long list of numbers, where each number is the length of the corresponding word in the text:

```
>>> [len(w) for w in text1] ❶
[1, 4, 4, 2, 6, 8, 4, 1, 9, 1, 1, 8, 2, 1, 4, 11, 5, 2, 1, 7, 6, 1, 3, 4, 5, 2, ...]
>>> fdist = FreqDist([len(w) for w in text1])  ❷
>>> fdist  ❸
<FreqDist with 260819 outcomes>
>>> fdist.keys()
[3, 1, 4, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20]
>>>
```

We start by deriving a list of the lengths of words in `text1` ❶, and the `FreqDist` then counts the number of times each of these occurs ❷. The result ❸ is a distribution containing a quarter of a million items, each of which is a number corresponding to a word token in the text. But there are only 20 distinct items being counted, the numbers 1 through 20, because there are only 20 different word lengths. I.e., there are words consisting of just 1 character, 2 characters, ..., 20 characters, but none with 21 or more characters. One might wonder how frequent the different lengths of words are (e.g., how many words of length 4 appear in the text, are there more words of length 5 than length 4, etc.). We can do this as follows:

```
>>> fdist.items()
[(3, 50223), (1, 47933), (4, 42345), (2, 38513), (5, 26597), (6, 17111), (7, 14399),
(8, 9966), (9, 6428), (10, 3528), (11, 1873), (12, 1053), (13, 567), (14, 177),
(15, 70), (16, 22), (17, 12), (18, 1), (20, 1)]
>>> fdist.max()
3
>>> fdist[3]
50223
>>> fdist.freq(3)
0.19255882431878046
>>>
```

From this we see that the most frequent word length is 3, and that words of length 3 account for roughly 50,000 (or 20%) of the words making up the book. Although we will not pursue it here, further analysis of word length might help us understand

differences between authors, genres, or languages. Table 1-2 summarizes the functions defined in frequency distributions.

Table 1-2. Functions defined for NLTK's frequency distributions

Example	Description
`fdist = FreqDist(samples)`	Create a frequency distribution containing the given samples
`fdist.inc(sample)`	Increment the count for this sample
`fdist['monstrous']`	Count of the number of times a given sample occurred
`fdist.freq('monstrous')`	Frequency of a given sample
`fdist.N()`	Total number of samples
`fdist.keys()`	The samples sorted in order of decreasing frequency
`for sample in fdist:`	Iterate over the samples, in order of decreasing frequency
`fdist.max()`	Sample with the greatest count
`fdist.tabulate()`	Tabulate the frequency distribution
`fdist.plot()`	Graphical plot of the frequency distribution
`fdist.plot(cumulative=True)`	Cumulative plot of the frequency distribution
`fdist1 < fdist2`	Test if samples in `fdist1` occur less frequently than in `fdist2`

Our discussion of frequency distributions has introduced some important Python concepts, and we will look at them systematically in Section 1.4.

1.4 Back to Python: Making Decisions and Taking Control

So far, our little programs have had some interesting qualities: the ability to work with language, and the potential to save human effort through automation. A key feature of programming is the ability of machines to make decisions on our behalf, executing instructions when certain conditions are met, or repeatedly looping through text data until some condition is satisfied. This feature is known as **control**, and is the focus of this section.

Conditionals

Python supports a wide range of operators, such as `<` and `>=`, for testing the relationship between values. The full set of these **relational operators** are shown in Table 1-3.

Table 1-3. Numerical comparison operators

Operator	Relationship
`<`	Less than
`<=`	Less than or equal to
`==`	Equal to (note this is two "="signs, not one)

Operator	Relationship
`!=`	Not equal to
`>`	Greater than
`>=`	Greater than or equal to

We can use these to select different words from a sentence of news text. Here are some examples—notice only the operator is changed from one line to the next. They all use `sent7`, the first sentence from `text7` (*Wall Street Journal*). As before, if you get an error saying that `sent7` is undefined, you need to first type: `from nltk.book import *`.

```
>>> sent7
['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the',
'board', 'as', 'a', 'nonexecutive', 'director', 'Nov.', '29', '.']
>>> [w for w in sent7 if len(w) < 4]
[',', '61', 'old', ',', 'the', 'as', 'a', '29', '.']
>>> [w for w in sent7 if len(w) <= 4]
[',', '61', 'old', ',', 'will', 'join', 'the', 'as', 'a', 'Nov.', '29', '.']
>>> [w for w in sent7 if len(w) == 4]
['will', 'join', 'Nov.']
>>> [w for w in sent7 if len(w) != 4]
['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'the', 'board',
'as', 'a', 'nonexecutive', 'director', '29', '.']
>>>
```

There is a common pattern to all of these examples: `[w for w in text if condition]`, where *`condition`* is a Python "test" that yields either true or false. In the cases shown in the previous code example, the condition is always a numerical comparison. However, we can also test various properties of words, using the functions listed in Table 1-4.

Table 1-4. Some word comparison operators

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

Here are some examples of these operators being used to select words from our texts: words ending with *-ableness*; words containing *gnt*; words having an initial capital; and words consisting entirely of digits.

```
>>> sorted([w for w in set(text1) if w.endswith('ableness')])
['comfortableness', 'honourableness', 'immutableness', 'indispensableness', ...]
>>> sorted([term for term in set(text4) if 'gnt' in term])
['Sovereignty', 'sovereignties', 'sovereignty']
>>> sorted([item for item in set(text6) if item.istitle()])
['A', 'Aaaaaaaaah', 'Aaaaaaaah', 'Aaaaaah', 'Aaaah', 'Aaaaugh', 'Aaagh', ...]
>>> sorted([item for item in set(sent7) if item.isdigit()])
['29', '61']
>>>
```

We can also create more complex conditions. If c is a condition, then `not` c is also a condition. If we have two conditions c_1 and c_2, then we can combine them to form a new condition using conjunction and disjunction: c_1 `and` c_2, c_1 `or` c_2.

Your Turn: Run the following examples and try to explain what is going on in each one. Next, try to make up some conditions of your own.

```
>>> sorted([w for w in set(text7) if '-' in w and 'index' in w])
>>> sorted([wd for wd in set(text3) if wd.istitle() and len(wd) > 10])
>>> sorted([w for w in set(sent7) if not w.islower()])
>>> sorted([t for t in set(text2) if 'cie' in t or 'cei' in t])
```

Operating on Every Element

In Section 1.3, we saw some examples of counting items other than words. Let's take a closer look at the notation we used:

```
>>> [len(w) for w in text1]
[1, 4, 4, 2, 6, 8, 4, 1, 9, 1, 1, 8, 2, 1, 4, 11, 5, 2, 1, 7, 6, 1, 3, 4, 5, 2, ...]
>>> [w.upper() for w in text1]
['[', 'MOBY', 'DICK', 'BY', 'HERMAN', 'MELVILLE', '1851', ']', 'ETYMOLOGY', '.', ...]
>>>
```

These expressions have the form `[f(w) for ...]` or `[w.f() for ...]`, where `f` is a function that operates on a word to compute its length, or to convert it to uppercase. For now, you don't need to understand the difference between the notations `f(w)` and `w.f()`. Instead, simply learn this Python idiom which performs the same operation on every element of a list. In the preceding examples, it goes through each word in `text1`, assigning each one in turn to the variable `w` and performing the specified operation on the variable.

The notation just described is called a "list comprehension." This is our first example of a Python idiom, a fixed notation that we use habitually without bothering to analyze each time. Mastering such idioms is an important part of becoming a fluent Python programmer.

Let's return to the question of vocabulary size, and apply the same idiom here:

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

```
>>> len(set(text1))
19317
>>> len(set([word.lower() for word in text1]))
17231
>>>
```

Now that we are not double-counting words like *This* and *this*, which differ only in capitalization, we've wiped 2,000 off the vocabulary count! We can go a step further and eliminate numbers and punctuation from the vocabulary count by filtering out any non-alphabetic items:

```
>>> len(set([word.lower() for word in text1 if word.isalpha()]))
16948
>>>
```

This example is slightly complicated: it lowercases all the purely alphabetic items. Perhaps it would have been simpler just to count the lowercase-only items, but this gives the wrong answer (why?).

Don't worry if you don't feel confident with list comprehensions yet, since you'll see many more examples along with explanations in the following chapters.

Nested Code Blocks

Most programming languages permit us to execute a block of code when a **conditional expression**, or `if` statement, is satisfied. We already saw examples of conditional tests in code like `[w for w in sent7 if len(w) < 4]`. In the following program, we have created a variable called `word` containing the string value `'cat'`. The `if` statement checks whether the test `len(word) < 5` is true. It is, so the body of the `if` statement is invoked and the `print` statement is executed, displaying a message to the user. Remember to indent the `print` statement by typing four spaces.

```
>>> word = 'cat'
>>> if len(word) < 5:
...     print 'word length is less than 5'
...   ❶
word length is less than 5
>>>
```

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

If we change the conditional test to `len(word) >= 5`, to check that the length of `word` is greater than or equal to `5`, then the test will no longer be true. This time, the body of the `if` statement will not be executed, and no message is shown to the user:

```
>>> if len(word) >= 5:
...   print 'word length is greater than or equal to 5'
...
>>>
```

An `if` statement is known as a **control structure** because it controls whether the code in the indented block will be run. Another control structure is the `for` loop. Try the following, and remember to include the colon and the four spaces:

```
>>> for word in ['Call', 'me', 'Ishmael', '.']:
...     print word
...
Call
me
Ishmael
.
>>>
```

This is called a loop because Python executes the code in circular fashion. It starts by performing the assignment `word = 'Call'`, effectively using the `word` variable to name the first item of the list. Then, it displays the value of `word` to the user. Next, it goes back to the `for` statement, and performs the assignment `word = 'me'` before displaying this new value to the user, and so on. It continues in this fashion until every item of the list has been processed.

Looping with Conditions

Now we can combine the `if` and `for` statements. We will loop over every item of the list, and print the item only if it ends with the letter *l*. We'll pick another name for the variable to demonstrate that Python doesn't try to make sense of variable names.

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>> for xyzzy in sent1:
...     if xyzzy.endswith('l'):
...         print xyzzy
...
Call
Ishmael
>>>
```

You will notice that `if` and `for` statements have a colon at the end of the line, before the indentation begins. In fact, all Python control structures end with a colon. The colon indicates that the current statement relates to the indented block that follows.

We can also specify an action to be taken if the condition of the `if` statement is not met. Here we see the `elif` (else if) statement, and the `else` statement. Notice that these also have colons before the indented code.

```
>>> for token in sent1:
...     if token.islower():
...         print token, 'is a lowercase word'
...     elif token.istitle():
...         print token, 'is a titlecase word'
...     else:
...         print token, 'is punctuation'
...
Call is a titlecase word
me is a lowercase word
```

```
Ishmael is a titlecase word
. is punctuation
>>>
```

As you can see, even with this small amount of Python knowledge, you can start to build multiline Python programs. It's important to develop such programs in pieces, testing that each piece does what you expect before combining them into a program. This is why the Python interactive interpreter is so invaluable, and why you should get comfortable using it.

Finally, let's combine the idioms we've been exploring. First, we create a list of *cie* and *cei* words, then we loop over each item and print it. Notice the comma at the end of the print statement, which tells Python to produce its output on a single line.

```
>>> tricky = sorted([w for w in set(text2) if 'cie' in w or 'cei' in w])
>>> for word in tricky:
...     print word,
ancient ceiling conceit conceited conceive conscience
conscientious conscientiously deceitful deceive ...
>>>
```

1.5 Automatic Natural Language Understanding

We have been exploring language bottom-up, with the help of texts and the Python programming language. However, we're also interested in exploiting our knowledge of language and computation by building useful language technologies. We'll take the opportunity now to step back from the nitty-gritty of code in order to paint a bigger picture of natural language processing.

At a purely practical level, we all need help to navigate the universe of information locked up in text on the Web. Search engines have been crucial to the growth and popularity of the Web, but have some shortcomings. It takes skill, knowledge, and some luck, to extract answers to such questions as: *What tourist sites can I visit between Philadelphia and Pittsburgh on a limited budget? What do experts say about digital SLR cameras? What predictions about the steel market were made by credible commentators in the past week?* Getting a computer to answer them automatically involves a range of language processing tasks, including information extraction, inference, and summarization, and would need to be carried out on a scale and with a level of robustness that is still beyond our current capabilities.

On a more philosophical level, a long-standing challenge within artificial intelligence has been to build intelligent machines, and a major part of intelligent behavior is understanding language. For many years this goal has been seen as too difficult. However, as NLP technologies become more mature, and robust methods for analyzing unrestricted text become more widespread, the prospect of natural language understanding has re-emerged as a plausible goal.

In this section we describe some language understanding technologies, to give you a sense of the interesting challenges that are waiting for you.

Word Sense Disambiguation

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

(2) a. *serve*: help with food or drink; hold an office; put ball into play
    b. *dish*: plate; course of a meal; communications device

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

(3) a. The lost children were found by the *searchers* (agentive)
    b. The lost children were found by the *mountain* (locative)
    c. The lost children were found by the *afternoon* (temporal)

Pronoun Resolution

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

(4) a. The thieves stole the paintings. They were subsequently *sold*.
    b. The thieves stole the paintings. They were subsequently *caught*.
    c. The thieves stole the paintings. They were subsequently *found*.

Answering this question involves finding the **antecedent** of the pronoun *they*, either thieves or paintings. Computational techniques for tackling this problem include **anaphora resolution**—identifying what a pronoun or noun phrase refers to—and

semantic role labeling—identifying how a noun phrase relates to the verb (as agent, patient, instrument, and so on).

Generating Language Output

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

(5) a. *Text:* ... The thieves stole the paintings. They were subsequently sold. ...
b. *Human:* Who or what was sold?
c. *Machine:* The paintings.

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

(6) a. The thieves stole the paintings. They were subsequently found.
b. Les voleurs ont volé les peintures. Ils ont été trouvés plus tard. (the thieves)
c. Les voleurs ont volé les peintures. Elles ont été trouvées plus tard. (the paintings)

In all of these examples, working out the sense of a word, the subject of a verb, and the antecedent of a pronoun are steps in establishing the meaning of a sentence, things we would expect a language understanding system to be able to do.

Machine Translation

For a long time now, machine translation (MT) has been the holy grail of language understanding, ultimately seeking to provide high-quality, idiomatic translation between any pair of languages. Its roots go back to the early days of the Cold War, when the promise of automatic translation led to substantial government sponsorship, and with it, the genesis of NLP itself.

Today, practical translation systems exist for particular pairs of languages, and some are integrated into web search engines. However, these systems have some serious shortcomings. We can explore them with the help of NLTK's "babelizer" (which is automatically loaded when you import this chapter's materials using `from nltk.book import *`). This program submits a sentence for translation into a specified language,

then submits the resulting sentence for translation back into English. It stops after 12 iterations, or if it receives a translation that was produced already (indicating a loop):

```
>>> babelize_shell()
NLTK Babelizer: type 'help' for a list of commands.
Babel> how long before the next flight to Alice Springs?
Babel> german
Babel> run
0> how long before the next flight to Alice Springs?
1> wie lang vor dem folgenden Flug zu Alice Springs?
2> how long before the following flight to Alice jump?
3> wie lang vor dem folgenden Flug zu Alice springen Sie?
4> how long before the following flight to Alice do you jump?
5> wie lang, bevor der folgende Flug zu Alice tun, Sie springen?
6> how long, before the following flight to Alice does, do you jump?
7> wie lang bevor der folgende Flug zu Alice tut, tun Sie springen?
8> how long before the following flight to Alice does, do you jump?
9> wie lang, bevor der folgende Flug zu Alice tut, tun Sie springen?
10> how long, before the following flight does to Alice, do do you jump?
11> wie lang bevor der folgende Flug zu Alice tut, Sie tun Sprung?
12> how long before the following flight does leap to Alice, does you?
```

Observe that the system correctly translates *Alice Springs* from English to German (in the line starting `1>`), but on the way back to English, this ends up as *Alice jump* (line `2`). The preposition *before* is initially translated into the corresponding German preposition *vor*, but later into the conjunction *bevor* (line 5). After line `5` the sentences become non-sensical (but notice the various phrasings indicated by the commas, and the change from *jump* to *leap*). The translation system did not recognize when a word was part of a proper name, and it misinterpreted the grammatical structure. The grammatical problems are more obvious in the following example. Did John find the pig, or did the pig find John?

```
>>> babelize_shell()
Babel> The pig that John found looked happy
Babel> german
Babel> run
0> The pig that John found looked happy
1> Das Schwein, das John fand, schaute gl?cklich
2> The pig, which found John, looked happy
```

Machine translation is difficult because a given word could have several possible translations (depending on its meaning), and because word order must be changed in keeping with the grammatical structure of the target language. Today these difficulties are being faced by collecting massive quantities of parallel texts from news and government websites that publish documents in two or more languages. Given a document in German and English, and possibly a bilingual dictionary, we can automatically pair up the sentences, a process called **text alignment**. Once we have a million or more sentence pairs, we can detect corresponding words and phrases, and build a model that can be used for translating new text.

Spoken Dialogue Systems

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

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

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

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

Dialogue systems give us an opportunity to mention the commonly assumed pipeline for NLP. Figure 1-5 shows the architecture of a simple dialogue system. Along the top of the diagram, moving from left to right, is a "pipeline" of some language understanding **components**. These map from speech input via syntactic parsing to some kind of meaning representation. Along the middle, moving from right to left, is the reverse pipeline of components for converting concepts to speech. These components make up the dynamic aspects of the system. At the bottom of the diagram are some representative bodies of static information: the repositories of language-related data that the processing components draw on to do their work.

Your Turn: For an example of a primitive dialogue system, try having a conversation with an NLTK chatbot. To see the available chatbots, run `nltk.chat.chatbots()`. (Remember to `import nltk` first.)

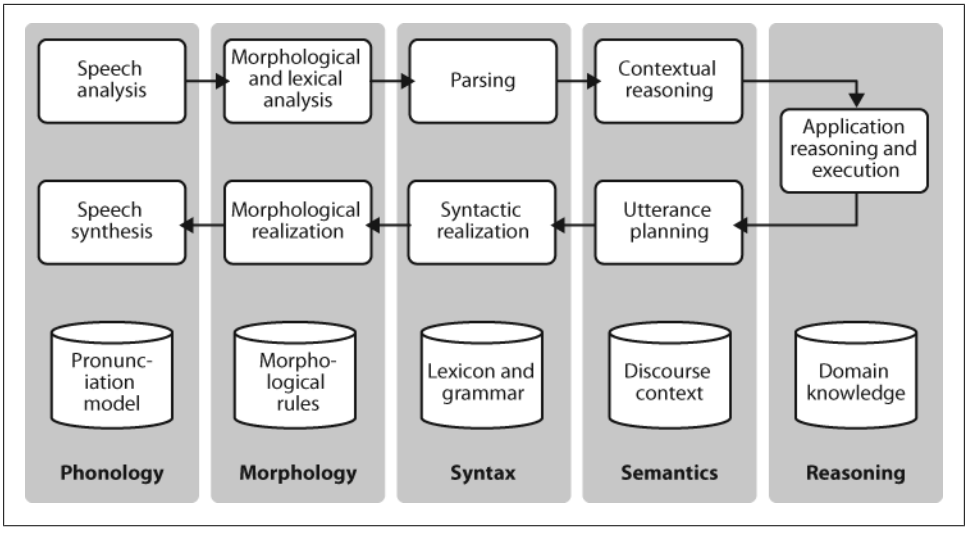


Figure 1-5. Simple pipeline architecture for a spoken dialogue system: Spoken input (top left) is analyzed, words are recognized, sentences are parsed and interpreted in context, application-specific actions take place (top right); a response is planned, realized as a syntactic structure, then to suitably inflected words, and finally to spoken output; different types of linguistic knowledge inform each stage of the process.

Textual Entailment

The challenge of language understanding has been brought into focus in recent years by a public "shared task" called Recognizing Textual Entailment (RTE). The basic scenario is simple. Suppose you want to find evidence to support the hypothesis: *Sandra Goudie was defeated by Max Purnell*, and that you have another short text that seems to be relevant, for example, *Sandra Goudie was first elected to Parliament in the 2002 elections, narrowly winning the seat of Coromandel by defeating Labour candidate Max Purnell and pushing incumbent Green MP Jeanette Fitzsimons into third place*. Does the text provide enough evidence for you to accept the hypothesis? In this particular case, the answer will be "No." You can draw this conclusion easily, but it is very hard to come up with automated methods for making the right decision. The RTE Challenges provide data that allow competitors to develop their systems, but not enough data for "brute force" machine learning techniques (a topic we will cover in Chapter 6). Consequently, some linguistic analysis is crucial. In the previous example, it is important for the system to note that *Sandra Goudie* names the person being defeated in the hypothesis, not the person doing the defeating in the text. As another illustration of the difficulty of the task, consider the following text-hypothesis pair:

(7) a. Text: David Golinkin is the editor or author of 18 books, and over 150 responsa, articles, sermons and books

b. Hypothesis: Golinkin has written 18 books

In order to determine whether the hypothesis is supported by the text, the system needs the following background knowledge: (i) if someone is an author of a book, then he/she has written that book; (ii) if someone is an editor of a book, then he/she has not written (all of) that book; (iii) if someone is editor or author of 18 books, then one cannot conclude that he/she is author of 18 books.

Limitations of NLP

Despite the research-led advances in tasks such as RTE, natural language systems that have been deployed for real-world applications still cannot perform common-sense reasoning or draw on world knowledge in a general and robust manner. We can wait for these difficult artificial intelligence problems to be solved, but in the meantime it is necessary to live with some severe limitations on the reasoning and knowledge capabilities of natural language systems. Accordingly, right from the beginning, an important goal of NLP research has been to make progress on the difficult task of building technologies that "understand language," using superficial yet powerful techniques instead of unrestricted knowledge and reasoning capabilities. Indeed, this is one of the goals of this book, and we hope to equip you with the knowledge and skills to build useful NLP systems, and to contribute to the long-term aspiration of building intelligent machines.

1.6 Summary

- Texts are represented in Python using lists: `['Monty', 'Python']`. We can use indexing, slicing, and the `len()` function on lists.
- A word "token" is a particular appearance of a given word in a text; a word "type" is the unique form of the word as a particular sequence of letters. We count word tokens using `len(text)` and word types using `len(set(text))`.
- We obtain the vocabulary of a text `t` using `sorted(set(t))`.
- We operate on each item of a text using `[f(x) for x in text]`.
- To derive the vocabulary, collapsing case distinctions and ignoring punctuation, we can write `set([w.lower() for w in text if w.isalpha()])`.
- We process each word in a text using a `for` statement, such as `for w in t:` or `for word in text:`. This must be followed by the colon character and an indented block of code, to be executed each time through the loop.
- We test a condition using an `if` statement: `if len(word) < 5:`. This must be followed by the colon character and an indented block of code, to be executed only if the condition is true.
- A frequency distribution is a collection of items along with their frequency counts (e.g., the words of a text and their frequency of appearance).

- A function is a block of code that has been assigned a name and can be reused. Functions are defined using the `def` keyword, as in `def mult(x, y)`; `x` and `y` are parameters of the function, and act as placeholders for actual data values.
- A function is called by specifying its name followed by one or more arguments inside parentheses, like this: `mult(3, 4)`, e.g., `len(text1)`.

1.7 Further Reading

This chapter has introduced new concepts in programming, natural language processing, and linguistics, all mixed in together. Many of them are consolidated in the following chapters. However, you may also want to consult the online materials provided with this chapter (at *http://www.nltk.org/*), including links to additional background materials, and links to online NLP systems. You may also like to read up on some linguistics and NLP-related concepts in Wikipedia (e.g., collocations, the Turing Test, the type-token distinction).

You should acquaint yourself with the Python documentation available at *http://docs.python.org/*, including the many tutorials and comprehensive reference materials linked there. A *Beginner's Guide to Python* is available at *http://wiki.python.org/moin/BeginnersGuide*. Miscellaneous questions about Python might be answered in the FAQ at *http://www.python.org/doc/faq/general/*.

As you delve into NLTK, you might want to subscribe to the mailing list where new releases of the toolkit are announced. There is also an NLTK-Users mailing list, where users help each other as they learn how to use Python and NLTK for language analysis work. Details of these lists are available at *http://www.nltk.org/*.

For more information on the topics covered in Section 1.5, and on NLP more generally, you might like to consult one of the following excellent books:

- Indurkhya, Nitin and Fred Damerau (eds., 2010) *Handbook of Natural Language Processing* (second edition), Chapman & Hall/CRC.
- Jurafsky, Daniel and James Martin (2008) *Speech and Language Processing* (second edition), Prentice Hall.
- Mitkov, Ruslan (ed., 2002) *The Oxford Handbook of Computational Linguistics*. Oxford University Press. (second edition expected in 2010).

The Association for Computational Linguistics is the international organization that represents the field of NLP. The ACL website hosts many useful resources, including: information about international and regional conferences and workshops; the *ACL Wiki* with links to hundreds of useful resources; and the *ACL Anthology*, which contains most of the NLP research literature from the past 50 years, fully indexed and freely downloadable.

Some excellent introductory linguistics textbooks are: (Finegan, 2007), (O'Grady et al., 2004), (OSU, 2007). You might like to consult *LanguageLog*, a popular linguistics blog with occasional posts that use the techniques described in this book.

1.8 Exercises

1. ○ Try using the Python interpreter as a calculator, and typing expressions like `12 / (4 + 1)`.
2. ○ Given an alphabet of 26 letters, there are 26 to the power 10, or `26 ** 10`, 10-letter strings we can form. That works out to `141167095653376L` (the `L` at the end just indicates that this is Python's long-number format). How many hundred-letter strings are possible?
3. ○ The Python multiplication operation can be applied to lists. What happens when you type `['Monty', 'Python'] * 20`, or `3 * sent1`?
4. ○ Review Section 1.1 on computing with language. How many words are there in `text2`? How many distinct words are there?
5. ○ Compare the lexical diversity scores for humor and romance fiction in Table 1-1. Which genre is more lexically diverse?
6. ○ Produce a dispersion plot of the four main protagonists in *Sense and Sensibility*: Elinor, Marianne, Edward, and Willoughby. What can you observe about the different roles played by the males and females in this novel? Can you identify the couples?
7. ○ Find the collocations in `text5`.
8. ○ Consider the following Python expression: `len(set(text4))`. State the purpose of this expression. Describe the two steps involved in performing this computation.
9. ○ Review Section 1.2 on lists and strings.
   a. Define a string and assign it to a variable, e.g., `my_string = 'My String'` (but put something more interesting in the string). Print the contents of this variable in two ways, first by simply typing the variable name and pressing Enter, then by using the `print` statement.
   b. Try adding the string to itself using `my_string + my_string`, or multiplying it by a number, e.g., `my_string * 3`. Notice that the strings are joined together without any spaces. How could you fix this?
10. ○ Define a variable `my_sent` to be a list of words, using the syntax `my_sent = ["My", "sent"]` (but with your own words, or a favorite saying).
    a. Use `' '.join(my_sent)` to convert this into a string.
    b. Use `split()` to split the string back into the list form you had to start with.
11. ○ Define several variables containing lists of words, e.g., `phrase1`, `phrase2`, and so on. Join them together in various combinations (using the plus operator) to form

whole sentences. What is the relationship between `len(phrase1 + phrase2)` and `len(phrase1) + len(phrase2)`?

12. ○ Consider the following two expressions, which have the same value. Which one will typically be more relevant in NLP? Why?
    a. `"Monty Python"[6:12]`
    b. `["Monty", "Python"][1]`
13. ○ We have seen how to represent a sentence as a list of words, where each word is a sequence of characters. What does `sent1[2][2]` do? Why? Experiment with other index values.
14. ○ The first sentence of `text3` is provided to you in the variable `sent3`. The index of *the* in `sent3` is 1, because `sent3[1]` gives us `'the'`. What are the indexes of the two other occurrences of this word in `sent3`?
15. ○ Review the discussion of conditionals in Section 1.4. Find all words in the Chat Corpus (`text5`) starting with the letter *b*. Show them in alphabetical order.
16. ○ Type the expression `range(10)` at the interpreter prompt. Now try `range(10, 20)`, `range(10, 20, 2)`, and `range(20, 10, -2)`. We will see a variety of uses for this built-in function in later chapters.
17. ◑ Use `text9.index()` to find the index of the word *sunset*. You'll need to insert this word as an argument between the parentheses. By a process of trial and error, find the slice for the complete sentence that contains this word.
18. ◑ Using list addition, and the `set` and `sorted` operations, compute the vocabulary of the sentences `sent1` ... `sent8`.
19. ◑ What is the difference between the following two lines? Which one will give a larger value? Will this be the case for other texts?

```
>>> sorted(set([w.lower() for w in text1]))
>>> sorted([w.lower() for w in set(text1)])
```

20. ◑ What is the difference between the following two tests: `w.isupper()` and `not w.islower()`?
21. ◑ Write the slice expression that extracts the last two words of `text2`.
22. ◑ Find all the four-letter words in the Chat Corpus (`text5`). With the help of a frequency distribution (`FreqDist`), show these words in decreasing order of frequency.
23. ◑ Review the discussion of looping with conditions in Section 1.4. Use a combination of `for` and `if` statements to loop over the words of the movie script for *Monty Python and the Holy Grail* (`text6`) and `print` all the uppercase words, one per line.
24. ◑ Write expressions for finding all words in `text6` that meet the following conditions. The result should be in the form of a list of words: `['word1', 'word2', ...]`.

a. Ending in *ize*

b. Containing the letter *z*

c. Containing the sequence of letters *pt*

d. All lowercase letters except for an initial capital (i.e., `titlecase`)

25. ◑ Define `sent` to be the list of words `['she', 'sells', 'sea', 'shells', 'by', 'the', 'sea', 'shore']`. Now write code to perform the following tasks:

    a. Print all words beginning with *sh*.

    b. Print all words longer than four characters

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

27. ◑ Define a function called `vocab_size(text)` that has a single parameter for the text, and which returns the vocabulary size of the text.

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

29. ◑ We have been using sets to store vocabularies. Try the following Python expression: `set(sent3) < set(text1)`. Experiment with this using different arguments to `set()`. What does it do? Can you think of a practical application for this?

CHAPTER 2

Accessing Text Corpora and Lexical Resources

Practical work in Natural Language Processing typically uses large bodies of linguistic data, or **corpora**. The goal of this chapter is to answer the following questions:

1. What are some useful text corpora and lexical resources, and how can we access them with Python?
2. Which Python constructs are most helpful for this work?
3. How do we avoid repeating ourselves when writing Python code?

This chapter continues to present programming concepts by example, in the context of a linguistic processing task. We will wait until later before exploring each Python construct systematically. Don't worry if you see an example that contains something unfamiliar; simply try it out and see what it does, and—if you're game—modify it by substituting some part of the code with a different text or word. This way you will associate a task with a programming idiom, and learn the hows and whys later.

2.1 Accessing Text Corpora

As just mentioned, a text corpus is a large body of text. Many corpora are designed to contain a careful balance of material in one or more genres. We examined some small text collections in Chapter 1, such as the speeches known as the US Presidential Inaugural Addresses. This particular corpus actually contains dozens of individual texts—one per address—but for convenience we glued them end-to-end and treated them as a single text. Chapter 1 also used various predefined texts that we accessed by typing `from book import *`. However, since we want to be able to work with other texts, this section examines a variety of text corpora. We'll see how to select individual texts, and how to work with them.

Gutenberg Corpus

NLTK includes a small selection of texts from the Project Gutenberg electronic text archive, which contains some 25,000 free electronic books, hosted at *http://www.gutenberg.org/*. We begin by getting the Python interpreter to load the NLTK package, then ask to see `nltk.corpus.gutenberg.fileids()`, the file identifiers in this corpus:

```
>>> import nltk
>>> nltk.corpus.gutenberg.fileids()
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt',
'blake-poems.txt', 'bryant-stories.txt', 'burgess-busterbrown.txt',
'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt',
'chesterton-thursday.txt', 'edgeworth-parents.txt', 'melville-moby_dick.txt',
'milton-paradise.txt', 'shakespeare-caesar.txt', 'shakespeare-hamlet.txt',
'shakespeare-macbeth.txt', 'whitman-leaves.txt']
```

Let's pick out the first of these texts—*Emma* by Jane Austen—and give it a short name, `emma`, then find out how many words it contains:

```
>>> emma = nltk.corpus.gutenberg.words('austen-emma.txt')
>>> len(emma)
192427
```

In Section 1.1, we showed how you could carry out concordancing of a text such as `text1` with the command `text1.concordance()`. However, this assumes that you are using one of the nine texts obtained as a result of doing `from nltk.book import *`. Now that you have started examining data from `nltk.corpus`, as in the previous example, you have to employ the following pair of statements to perform concordancing and other tasks from Section 1.1:

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

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

```
>>> from nltk.corpus import gutenberg
>>> gutenberg.fileids()
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', ...]
>>> emma = gutenberg.words('austen-emma.txt')
```

Let's write a short program to display other information about each text, by looping over all the values of `fileid` corresponding to the `gutenberg` file identifiers listed earlier and then computing statistics for each text. For a compact output display, we will make sure that the numbers are all integers, using `int()`.

```
>>> for fileid in gutenberg.fileids():
...     num_chars = len(gutenberg.raw(fileid)) ❶
...     num_words = len(gutenberg.words(fileid))
...     num_sents = len(gutenberg.sents(fileid))
```

```
...     num_vocab = len(set([w.lower() for w in gutenberg.words(fileid)]))
...     print int(num_chars/num_words), int(num_words/num_sents), int(num_words/num_vocab),
        fileid
...
4 21 26 austen-emma.txt
4 23 16 austen-persuasion.txt
4 24 22 austen-sense.txt
4 33 79 bible-kjv.txt
4 18 5 blake-poems.txt
4 17 14 bryant-stories.txt
4 17 12 burgess-busterbrown.txt
4 16 12 carroll-alice.txt
4 17 11 chesterton-ball.txt
4 19 11 chesterton-brown.txt
4 16 10 chesterton-thursday.txt
4 18 24 edgeworth-parents.txt
4 24 15 melville-moby_dick.txt
4 52 10 milton-paradise.txt
4 12 8 shakespeare-caesar.txt
4 13 7 shakespeare-hamlet.txt
4 13 6 shakespeare-macbeth.txt
4 35 12 whitman-leaves.txt
```

This program displays three statistics for each text: average word length, average sentence length, and the number of times each vocabulary item appears in the text on average (our lexical diversity score). Observe that average word length appears to be a general property of English, since it has a recurrent value of *4*. (In fact, the average word length is really *3*, not *4*, since the `num_chars` variable counts space characters.) By contrast average sentence length and lexical diversity appear to be characteristics of particular authors.

The previous example also showed how we can access the "raw" text of the book ❶, not split up into tokens. The `raw()` function gives us the contents of the file without any linguistic processing. So, for example, `len(gutenberg.raw('blake-poems.txt'))` tells us how many *letters* occur in the text, including the spaces between words. The `sents()` function divides the text up into its sentences, where each sentence is a list of words:

```
>>> macbeth_sentences = gutenberg.sents('shakespeare-macbeth.txt')
>>> macbeth_sentences
[['[', 'The', 'Tragedie', 'of', 'Macbeth', 'by', 'William', 'Shakespeare',
'1603', ']'], ['Actus', 'Primus', '.'], ...]
>>> macbeth_sentences[1037]
['Double', ',', 'double', ',', 'toile', 'and', 'trouble', ';',
'Fire', 'burne', ',', 'and', 'Cauldron', 'bubble']
>>> longest_len = max([len(s) for s in macbeth_sentences])
>>> [s for s in macbeth_sentences if len(s) == longest_len]
[['Doubtfull', 'it', 'stood', ',', 'As', 'two', 'spent', 'Swimmers', ',', 'that',
'doe', 'cling', 'together', ',', 'And', 'choake', 'their', 'Art', ':', 'The',
'mercilesse', 'Macdonwald', ...], ...]
```

Most NLTK corpus readers include a variety of access methods apart from `words()`, `raw()`, and `sents()`. Richer linguistic content is available from some corpora, such as part-of-speech tags, dialogue tags, syntactic trees, and so forth; we will see these in later chapters.

Web and Chat Text

Although Project Gutenberg contains thousands of books, it represents established literature. It is important to consider less formal language as well. NLTK's small collection of web text includes content from a Firefox discussion forum, conversations overheard in New York, the movie script of *Pirates of the Carribean*, personal advertisements, and wine reviews:

```
>>> from nltk.corpus import webtext
>>> for fileid in webtext.fileids():
...     print fileid, webtext.raw(fileid)[:65], '...'
...
firefox.txt Cookie Manager: "Don't allow sites that set removed cookies to se...
grail.txt SCENE 1: [wind] [clop clop clop] KING ARTHUR: Whoa there!  [clop...
overheard.txt White guy: So, do you have any plans for this evening? Asian girl...
pirates.txt PIRATES OF THE CARRIBEAN: DEAD MAN'S CHEST, by Ted Elliott & Terr...
singles.txt 25 SEXY MALE, seeks attrac older single lady, for discreet encoun...
wine.txt Lovely delicate, fragrant Rhone wine. Polished leather and strawb...
```

There is also a corpus of instant messaging chat sessions, originally collected by the Naval Postgraduate School for research on automatic detection of Internet predators. The corpus contains over 10,000 posts, anonymized by replacing usernames with generic names of the form "UserNNN", and manually edited to remove any other identifying information. The corpus is organized into 15 files, where each file contains several hundred posts collected on a given date, for an age-specific chatroom (teens, 20s, 30s, 40s, plus a generic adults chatroom). The filename contains the date, chatroom, and number of posts; e.g., `10-19-20s_706posts.xml` contains 706 posts gathered from the 20s chat room on 10/19/2006.

```
>>> from nltk.corpus import nps_chat
>>> chatroom = nps_chat.posts('10-19-20s_706posts.xml')
>>> chatroom[123]
['i', 'do', "n't", 'want', 'hot', 'pics', 'of', 'a', 'female', ',',
'I', 'can', 'look', 'in', 'a', 'mirror', '.']
```

Brown Corpus

The Brown Corpus was the first million-word electronic corpus of English, created in 1961 at Brown University. This corpus contains text from 500 sources, and the sources have been categorized by genre, such as *news*, *editorial*, and so on. Table 2-1 gives an example of each genre (for a complete list, see *http://icame.uib.no/brown/bcm-los.html*).

Table 2-1. Example document for each section of the Brown Corpus

ID	File	Genre	Description
A16	*ca16*	news	Chicago Tribune: *Society Reportage*
B02	*cb02*	editorial	Christian Science Monitor: *Editorials*
C17	*cc17*	reviews	Time Magazine: *Reviews*
D12	*cd12*	religion	Underwood: *Probing the Ethics of Realtors*
E36	*ce36*	hobbies	Norling: *Renting a Car in Europe*
F25	*cf25*	lore	Boroff: *Jewish Teenage Culture*
G22	*cg22*	belles_lettres	Reiner: *Coping with Runaway Technology*
H15	*ch15*	government	US Office of Civil and Defence Mobilization: *The Family Fallout Shelter*
J17	*cj19*	learned	Mosteller: *Probability with Statistical Applications*
K04	*ck04*	fiction	W.E.B. Du Bois: *Worlds of Color*
L13	*cl13*	mystery	Hitchens: *Footsteps in the Night*
M01	*cm01*	science_fiction	Heinlein: *Stranger in a Strange Land*
N14	*cn15*	adventure	Field: *Rattlesnake Ridge*
P12	*cp12*	romance	Callaghan: *A Passion in Rome*
R06	*cr06*	humor	Thurber: *The Future, If Any, of Comedy*

We can access the corpus as a list of words or a list of sentences (where each sentence is itself just a list of words). We can optionally specify particular categories or files to read:

```
>>> from nltk.corpus import brown
>>> brown.categories()
['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies',
'humor', 'learned', 'lore', 'mystery', 'news', 'religion', 'reviews', 'romance',
'science_fiction']
>>> brown.words(categories='news')
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
>>> brown.words(fileids=['cg22'])
['Does', 'our', 'society', 'have', 'a', 'runaway', ',', ...]
>>> brown.sents(categories=['news', 'editorial', 'reviews'])
[['The', 'Fulton', 'County'...], ['The', 'jury', 'further'...], ...]
```

The Brown Corpus is a convenient resource for studying systematic differences between genres, a kind of linguistic inquiry known as **stylistics**. Let's compare genres in their usage of modal verbs. The first step is to produce the counts for a particular genre. Remember to `import nltk` before doing the following:

```
>>> from nltk.corpus import brown
>>> news_text = brown.words(categories='news')
>>> fdist = nltk.FreqDist([w.lower() for w in news_text])
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> for m in modals:
...     print m + ':', fdist[m],
```

```
...
can: 94 could: 87 may: 93 might: 38 must: 53 will: 389
```

Your Turn: Choose a different section of the Brown Corpus, and adapt the preceding example to count a selection of *wh* words, such as *what*, *when*, *where*, *who* and *why*.

Next, we need to obtain counts for each genre of interest. We'll use NLTK's support for conditional frequency distributions. These are presented systematically in Section 2.2, where we also unpick the following code line by line. For the moment, you can ignore the details and just concentrate on the output.

```
>>> 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']
>>> cfd.tabulate(conditions=genres, samples=modals)
                  can could  may might must will
            news   93   86   66   38   50  389
       religion   82   59   78   12   54   71
        hobbies  268   58  131   22   83  264
science_fiction   16   49    4   12    8   16
        romance   74  193   11   51   45   43
          humor   16   30    8    8    9   13
```

Observe that the most frequent modal in the news genre is *will*, while the most frequent modal in the romance genre is *could*. Would you have predicted this? The idea that word counts might distinguish genres will be taken up again in Chapter 6.

Reuters Corpus

The Reuters Corpus contains 10,788 news documents totaling 1.3 million words. The documents have been classified into 90 topics, and grouped into two sets, called "training" and "test"; thus, the text with fileid `'test/14826'` is a document drawn from the test set. This split is for training and testing algorithms that automatically detect the topic of a document, as we will see in Chapter 6.

```
>>> from nltk.corpus import reuters
>>> reuters.fileids()
['test/14826', 'test/14828', 'test/14829', 'test/14832', ...]
>>> reuters.categories()
['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa',
'coconut', 'coconut-oil', 'coffee', 'copper', 'copra-cake', 'corn',
'cotton', 'cotton-oil', 'cpi', 'cpu', 'crude', 'dfl', 'dlr', ...]
```

Unlike the Brown Corpus, categories in the Reuters Corpus overlap with each other, simply because a news story often covers multiple topics. We can ask for the topics

covered by one or more documents, or for the documents included in one or more categories. For convenience, the corpus methods accept a single fileid or a list of fileids.

```
>>> reuters.categories('training/9865')
['barley', 'corn', 'grain', 'wheat']
>>> reuters.categories(['training/9865', 'training/9880'])
['barley', 'corn', 'grain', 'money-fx', 'wheat']
>>> reuters.fileids('barley')
['test/15618', 'test/15649', 'test/15676', 'test/15728', 'test/15871', ...]
>>> reuters.fileids(['barley', 'corn'])
['test/14832', 'test/14858', 'test/15033', 'test/15043', 'test/15106',
'test/15287', 'test/15341', 'test/15618', 'test/15618', 'test/15648', ...]
```

Similarly, we can specify the words or sentences we want in terms of files or categories. The first handful of words in each of these texts are the titles, which by convention are stored as uppercase.

```
>>> reuters.words('training/9865')[:14]
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', 'BIDS',
'DETAILED', 'French', 'operators', 'have', 'requested', 'licences', 'to', 'export']
>>> reuters.words(['training/9865', 'training/9880'])
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', ...]
>>> reuters.words(categories='barley')
['FRENCH', 'FREE', 'MARKET', 'CEREAL', 'EXPORT', ...]
>>> reuters.words(categories=['barley', 'corn'])
['THAI', 'TRADE', 'DEFICIT', 'WIDENS', 'IN', 'FIRST', ...]
```

Inaugural Address Corpus

In Section 1.1, we looked at the Inaugural Address Corpus, but treated it as a single text. The graph in Figure 1-2 used "word offset" as one of the axes; this is the numerical index of the word in the corpus, counting from the first word of the first address. However, the corpus is actually a collection of 55 texts, one for each presidential address. An interesting property of this collection is its time dimension:

```
>>> from nltk.corpus import inaugural
>>> inaugural.fileids()
['1789-Washington.txt', '1793-Washington.txt', '1797-Adams.txt', ...]
>>> [fileid[:4] for fileid in inaugural.fileids()]
['1789', '1793', '1797', '1801', '1805', '1809', '1813', '1817', '1821', ...]
```

Notice that the year of each text appears in its filename. To get the year out of the filename, we extracted the first four characters, using `fileid[:4]`.

Let's look at how the words *America* and *citizen* are used over time. The following code converts the words in the Inaugural corpus to lowercase using `w.lower()` ❶, then checks whether they start with either of the "targets" `america` or `citizen` using `startswith()` ❶. Thus it will count words such as *American's* and *Citizens*. We'll learn about conditional frequency distributions in Section 2.2; for now, just consider the output, shown in Figure 2-1.

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

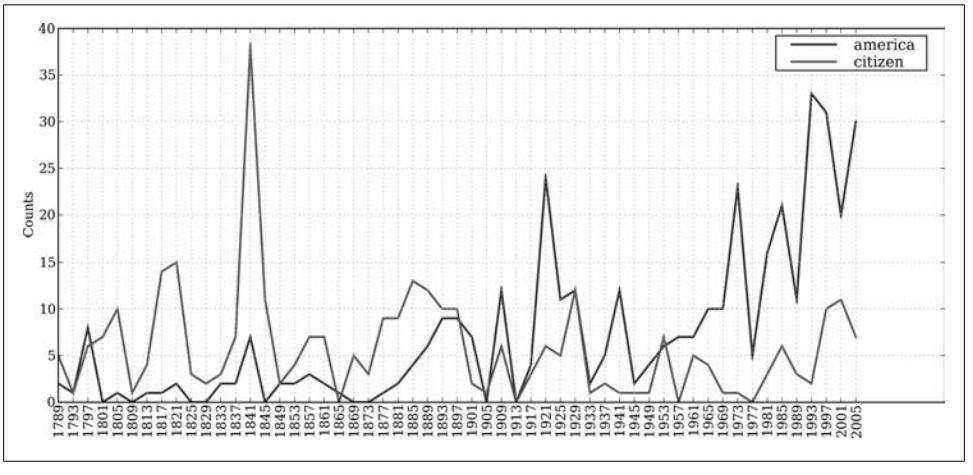


Figure 2-1. Plot of a conditional frequency distribution: All words in the Inaugural Address Corpus that begin with america *or* citizen *are counted; separate counts are kept for each address; these are plotted so that trends in usage over time can be observed; counts are not normalized for document length.*

Annotated Text Corpora

Many text corpora contain linguistic annotations, representing part-of-speech tags, named entities, syntactic structures, semantic roles, and so forth. NLTK provides convenient ways to access several of these corpora, and has data packages containing corpora and corpus samples, freely downloadable for use in teaching and research. Table 2-2 lists some of the corpora. For information about downloading them, see *http://www.nltk.org/data*. For more examples of how to access NLTK corpora, please consult the Corpus HOWTO at *http://www.nltk.org/howto*.

Table 2-2. Some of the corpora and corpus samples distributed with NLTK

Corpus	Compiler	Contents
Brown Corpus	Francis, Kucera	15 genres, 1.15M words, tagged, categorized
CESS Treebanks	CLiC-UB	1M words, tagged and parsed (Catalan, Spanish)
Chat-80 Data Files	Pereira & Warren	World Geographic Database
CMU Pronouncing Dictionary	CMU	127k entries
CoNLL 2000 Chunking Data	CoNLL	270k words, tagged and chunked

Corpus	Compiler	Contents
CoNLL 2002 Named Entity	CoNLL	700k words, POS and named entity tagged (Dutch, Spanish)
CoNLL 2007 Dependency Parsed Treebanks (selections)	CoNLL	150k words, dependency parsed (Basque, Catalan)
Dependency Treebank	Narad	Dependency parsed version of Penn Treebank sample
Floresta Treebank	Diana Santos et al.	9k sentences, tagged and parsed (Portuguese)
Gazetteer Lists	Various	Lists of cities and countries
Genesis Corpus	Misc web sources	6 texts, 200k words, 6 languages
Gutenberg (selections)	Hart, Newby, et al.	18 texts, 2M words
Inaugural Address Corpus	CSpan	U.S. Presidential Inaugural Addresses (1789–present)
Indian POS Tagged Corpus	Kumaran et al.	60k words, tagged (Bangla, Hindi, Marathi, Telugu)
MacMorpho Corpus	NILC, USP, Brazil	1M words, tagged (Brazilian Portuguese)
Movie Reviews	Pang, Lee	2k movie reviews with sentiment polarity classification
Names Corpus	Kantrowitz, Ross	8k male and female names
NIST 1999 Info Extr (selections)	Garofolo	63k words, newswire and named entity SGML markup
NPS Chat Corpus	Forsyth, Martell	10k IM chat posts, POS and dialogue-act tagged
Penn Treebank (selections)	LDC	40k words, tagged and parsed
PP Attachment Corpus	Ratnaparkhi	28k prepositional phrases, tagged as noun or verb modifiers
Proposition Bank	Palmer	113k propositions, 3,300 verb frames
Question Classification	Li, Roth	6k questions, categorized
Reuters Corpus	Reuters	1.3M words, 10k news documents, categorized
Roget's Thesaurus	Project Gutenberg	200k words, formatted text
RTE Textual Entailment	Dagan et al.	8k sentence pairs, categorized
SEMCOR	Rus, Mihalcea	880k words, POS and sense tagged
Senseval 2 Corpus	Pedersen	600k words, POS and sense tagged
Shakespeare texts (selections)	Bosak	8 books in XML format
State of the Union Corpus	CSpan	485k words, formatted text
Stopwords Corpus	Porter et al.	2,400 stopwords for 11 languages
Swadesh Corpus	Wiktionary	Comparative wordlists in 24 languages
Switchboard Corpus (selections)	LDC	36 phone calls, transcribed, parsed
TIMIT Corpus (selections)	NIST/LDC	Audio files and transcripts for 16 speakers
Univ Decl of Human Rights	United Nations	480k words, 300+ languages
VerbNet 2.1	Palmer et al.	5k verbs, hierarchically organized, linked to WordNet
Wordlist Corpus	OpenOffice.org et al.	960k words and 20k affixes for 8 languages
WordNet 3.0 (English)	Miller, Fellbaum	145k synonym sets

Corpora in Other Languages

NLTK comes with corpora for many languages, though in some cases you will need to learn how to manipulate character encodings in Python before using these corpora (see Section 3.3).

```
>>> nltk.corpus.cess_esp.words()
['El', 'grupo', 'estatal', 'Electricit\xe9_de_France', ...]
>>> nltk.corpus.floresta.words()
['Um', 'revivalismo', 'refrescante', 'O', '7_e_Meio', ...]
>>> nltk.corpus.indian.words('hindi.pos')
['\xe0\xa4\xaa\xe0\xa5\x82\xe0\xa4\xb0\xe0\xa5\x8d\xe0\xa4\xa3',
'\xe0\xa4\xaa\xe0\xa5\x8d\xe0\xa4\xb0\xe0\xa4\xa4\xe0\xa4\xbf\xe0\xa4\xac\xe0\xa4
\x82\xe0\xa4\xa7', ...]
>>> nltk.corpus.udhr.fileids()
['Abkhaz-Cyrillic+Abkh', 'Abkhaz-UTF8', 'Achehnese-Latin1', 'Achuar-Shiwiar-Latin1',
'Adja-UTF8', 'Afaan_Oromo_Oromiffa-Latin1', 'Afrikaans-Latin1', 'Aguaruna-Latin1',
'Akuapem_Twi-UTF8', 'Albanian_Shqip-Latin1', 'Amahuaca', 'Amahuaca-Latin1', ...]
>>> nltk.corpus.udhr.words('Javanese-Latin1')[11:]
[u'Saben', u'umat', u'manungsa', u'lair', u'kanthi', ...]
```

The last of these corpora, `udhr`, contains the Universal Declaration of Human Rights in over 300 languages. The fileids for this corpus include information about the character encoding used in the file, such as `UTF8` or `Latin1`. Let's use a conditional frequency distribution to examine the differences in word lengths for a selection of languages included in the `udhr` corpus. The output is shown in Figure 2-2 (run the program yourself to see a color plot). Note that `True` and `False` are Python's built-in Boolean values.

```
>>> from nltk.corpus import udhr
>>> languages = ['Chickasaw', 'English', 'German_Deutsch',
...     'Greenlandic_Inuktikut', 'Hungarian_Magyar', 'Ibibio_Efik']
>>> cfd = nltk.ConditionalFreqDist(
...           (lang, len(word))
...           for lang in languages
...           for word in udhr.words(lang + '-Latin1'))
>>> cfd.plot(cumulative=True)
```

Your Turn: Pick a language of interest in `udhr.fileids()`, and define a variable `raw_text = udhr.raw(`*`Language-Latin1`*`)`. Now plot a frequency distribution of the letters of the text using

```
nltk.FreqDist(raw_text).plot().
```

Unfortunately, for many languages, substantial corpora are not yet available. Often there is insufficient government or industrial support for developing language resources, and individual efforts are piecemeal and hard to discover or reuse. Some languages have no established writing system, or are endangered. (See Section 2.7 for suggestions on how to locate language resources.)

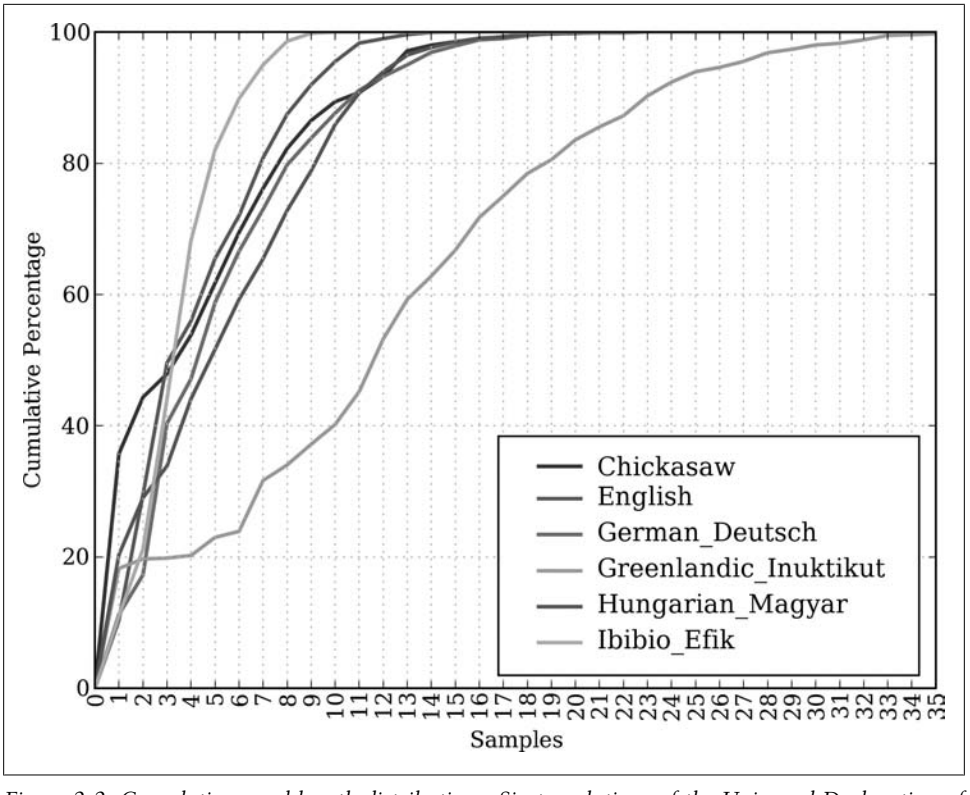


Figure 2-2. Cumulative word length distributions: Six translations of the Universal Declaration of Human Rights are processed; this graph shows that words having five or fewer letters account for about 80% of Ibibio text, 60% of German text, and 25% of Inuktitut text.

Text Corpus Structure

We have seen a variety of corpus structures so far; these are summarized in Figure 2-3. The simplest kind lacks any structure: it is just a collection of texts. Often, texts are grouped into categories that might correspond to genre, source, author, language, etc. Sometimes these categories overlap, notably in the case of topical categories, as a text can be relevant to more than one topic. Occasionally, text collections have temporal structure, news collections being the most common example.

NLTK's corpus readers support efficient access to a variety of corpora, and can be used to work with new corpora. Table 2-3 lists functionality provided by the corpus readers.

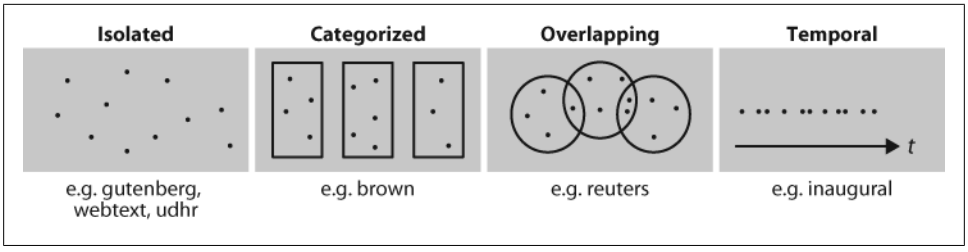


Figure 2-3. Common structures for text corpora: The simplest kind of corpus is a collection of isolated texts with no particular organization; some corpora are structured into categories, such as genre (Brown Corpus); some categorizations overlap, such as topic categories (Reuters Corpus); other corpora represent language use over time (Inaugural Address Corpus).

Table 2-3. Basic corpus functionality defined in NLTK: More documentation can be found using help(nltk.corpus.reader) and by reading the online Corpus HOWTO at http://www.nltk.org/howto.

Example	Description
`fileids()`	The files of the corpus
`fileids([categories])`	The files of the corpus corresponding to these categories
`categories()`	The categories of the corpus
`categories([fileids])`	The categories of the corpus corresponding to these files
`raw()`	The raw content of the corpus
`raw(fileids=[f1,f2,f3])`	The raw content of the specified files
`raw(categories=[c1,c2])`	The raw content of the specified categories
`words()`	The words of the whole corpus
`words(fileids=[f1,f2,f3])`	The words of the specified fileids
`words(categories=[c1,c2])`	The words of the specified categories
`sents()`	The sentences of the specified categories
`sents(fileids=[f1,f2,f3])`	The sentences of the specified fileids
`sents(categories=[c1,c2])`	The sentences of the specified categories
`abspath(fileid)`	The location of the given file on disk
`encoding(fileid)`	The encoding of the file (if known)
`open(fileid)`	Open a stream for reading the given corpus file
`root()`	The path to the root of locally installed corpus
`readme()`	The contents of the README file of the corpus

We illustrate the difference between some of the corpus access methods here:

```
>>> raw = gutenberg.raw("burgess-busterbrown.txt")
>>> raw[1:20]
'The Adventures of B'
>>> words = gutenberg.words("burgess-busterbrown.txt")
>>> words[1:20]
```

```
['The', 'Adventures', 'of', 'Buster', 'Bear', 'by', 'Thornton', 'W', '.',
'Burgess', '1920', ']', 'I', 'BUSTER', 'BEAR', 'GOES', 'FISHING', 'Buster',
'Bear']
>>> sents = gutenberg.sents("burgess-busterbrown.txt")
>>> sents[1:20]
[['I'], ['BUSTER', 'BEAR', 'GOES', 'FISHING'], ['Buster', 'Bear', 'yawned', 'as',
'he', 'lay', 'on', 'his', 'comfortable', 'bed', 'of', 'leaves', 'and', 'watched',
'the', 'first', 'early', 'morning', 'sunbeams', 'creeping', 'through', ...], ...]
```

Loading Your Own Corpus

If you have a your own collection of text files that you would like to access using the methods discussed earlier, you can easily load them with the help of NLTK's `PlaintextCorpusReader`. Check the location of your files on your file system; in the following example, we have taken this to be the directory */usr/share/dict*. Whatever the location, set this to be the value of `corpus_root` ❶. The second parameter of the `PlaintextCorpusReader` initializer ❷ can be a list of fileids, like `['a.txt', 'test/b.txt']`, or a pattern that matches all fileids, like `'[abc]/.*\.txt'` (see Section 3.4 for information about regular expressions).

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

As another example, suppose you have your own local copy of Penn Treebank (release 3), in `C:\corpora`. We can use the `BracketParseCorpusReader` to access this corpus. We specify the `corpus_root` to be the location of the parsed *Wall Street Journal* component of the corpus ❶, and give a `file_pattern` that matches the files contained within its subfolders ❷ (using forward slashes).

```
>>> from nltk.corpus import BracketParseCorpusReader
>>> corpus_root = r"C:\corpora\penntreebank\parsed\mrg\wsj" ❶
>>> file_pattern = r".*/wsj_.*\.mrg" ❷
>>> ptb = BracketParseCorpusReader(corpus_root, file_pattern)
>>> ptb.fileids()
['00/wsj_0001.mrg', '00/wsj_0002.mrg', '00/wsj_0003.mrg', '00/wsj_0004.mrg', ...]
>>> len(ptb.sents())
49208
>>> ptb.sents(fileids='20/wsj_2013.mrg')[19]
['The', '55-year-old', 'Mr.', 'Noriega', 'is', "n't", 'as', 'smooth', 'as', 'the',
'shah', 'of', 'Iran', ',', 'as', 'well-born', 'as', 'Nicaragua', "'s", 'Anastasio',
'Somoza', ',', 'as', 'imperial', 'as', 'Ferdinand', 'Marcos', 'of', 'the', 'Philippines',
'or', 'as', 'bloody', 'as', 'Haiti', "'s", 'Baby', Doc', 'Duvalier', '.']
```

2.2 Conditional Frequency Distributions

We introduced frequency distributions in Section 1.3. We saw that given some list `mylist` of words or other items, `FreqDist(mylist)` would compute the number of occurrences of each item in the list. Here we will generalize this idea.

When the texts of a corpus are divided into several categories (by genre, topic, author, etc.), we can maintain separate frequency distributions for each category. This will allow us to study systematic differences between the categories. In the previous section, we achieved this using NLTK's `ConditionalFreqDist` data type. A **conditional frequency distribution** is a collection of frequency distributions, each one for a different "condition." The condition will often be the category of the text. Figure 2-4 depicts a fragment of a conditional frequency distribution having just two conditions, one for news text and one for romance text.

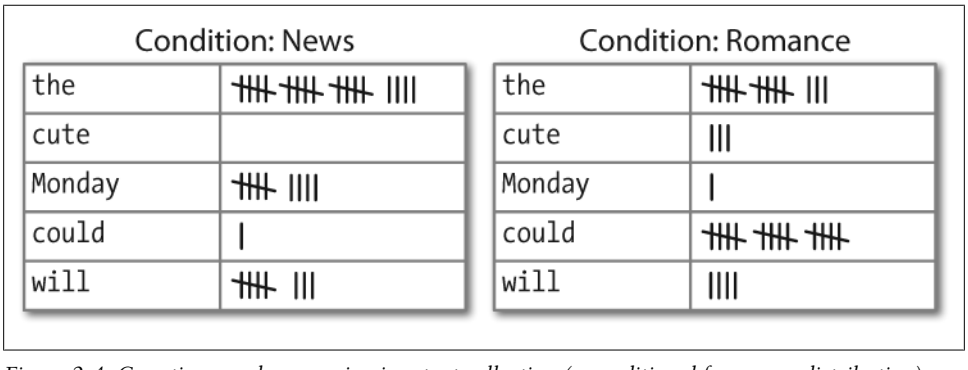


Figure 2-4. Counting words appearing in a text collection (a conditional frequency distribution).

Conditions and Events

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

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

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

Counting Words by Genre

In Section 2.1, we saw a conditional frequency distribution where the condition was the section of the Brown Corpus, and for each condition we counted words. Whereas `FreqDist()` takes a simple list as input, `ConditionalFreqDist()` takes a list of pairs.

```
>>> from nltk.corpus import brown
>>> cfd = nltk.ConditionalFreqDist(
...           (genre, word)
...           for genre in brown.categories()
...           for word in brown.words(categories=genre))
```

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

```
>>> genre_word = [(genre, word) ❶
...               for genre in ['news', 'romance'] ❷
...               for word in brown.words(categories=genre)] ❸
>>> len(genre_word)
170576
```

So, as we can see in the following code, pairs at the beginning of the list `genre_word` will be of the form (`'news'`, *word*) ❶, whereas those at the end will be of the form (`'roman ce'`, *word*) ❷.

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

We can now use this list of pairs to create a `ConditionalFreqDist`, and save it in a variable `cfd`. As usual, we can type the name of the variable to inspect it ❶, and verify it has two conditions ❷:

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

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

```
>>> cfd['news']
<FreqDist with 100554 outcomes>
>>> cfd['romance']
<FreqDist with 70022 outcomes>
>>> list(cfd['romance'])
[',', '.', 'the', 'and', 'to', 'a', 'of', '``', "''", 'was', 'I', 'in', 'he', 'had',
'?', 'her', 'that', 'it', 'his', 'she', 'with', 'you', 'for', 'at', 'He', 'on', 'him',
'said', '!', '--', 'be', 'as', ';', 'have', 'but', 'not', 'would', 'She', 'The', ...]
>>> cfd['romance']['could']
193
```

Plotting and Tabulating Distributions

Apart from combining two or more frequency distributions, and being easy to initialize, a `ConditionalFreqDist` provides some useful methods for tabulation and plotting.

The plot in Figure 2-1 was based on a conditional frequency distribution reproduced in the following code. The condition is either of the words *america* or *citizen* ❷, and the counts being plotted are the number of times the word occurred in a particular speech. It exploits the fact that the filename for each speech—for example, *1865-Lincoln.txt*—contains the year as the first four characters ❶. This code generates the pair `('america', '1865')` for every instance of a word whose lowercased form starts with *america*—such as *Americans*—in the file *1865-Lincoln.txt*.

```
>>> from nltk.corpus import inaugural
>>> cfd = nltk.ConditionalFreqDist(
...           (target, fileid[:4]) ❶
...           for fileid in inaugural.fileids()
...           for w in inaugural.words(fileid)
...           for target in ['america', 'citizen'] ❷
...           if w.lower().startswith(target))
```

The plot in Figure 2-2 was also based on a conditional frequency distribution, reproduced in the following code. This time, the condition is the name of the language, and the counts being plotted are derived from word lengths ❶. It exploits the fact that the filename for each language is the language name followed by `'-Latin1'` (the character encoding).

```
>>> from nltk.corpus import udhr
>>> languages = ['Chickasaw', 'English', 'German_Deutsch',
...     'Greenlandic_Inuktikut', 'Hungarian_Magyar', 'Ibibio_Efik']
>>> cfd = nltk.ConditionalFreqDist(
...           (lang, len(word)) ❶
...           for lang in languages
...           for word in udhr.words(lang + '-Latin1'))
```

In the `plot()` and `tabulate()` methods, we can optionally specify which conditions to display with a `conditions=` parameter. When we omit it, we get all the conditions. Similarly, we can limit the samples to display with a `samples=` parameter. This makes it possible to load a large quantity of data into a conditional frequency distribution, and then to explore it by plotting or tabulating selected conditions and samples. It also gives us full control over the order of conditions and samples in any displays. For example, we can tabulate the cumulative frequency data just for two languages, and for words less than 10 characters long, as shown next. We interpret the last cell on the top row to mean that 1,638 words of the English text have nine or fewer letters.

```
>>> cfd.tabulate(conditions=['English', 'German_Deutsch'],
...              samples=range(10), cumulative=True)
                  0    1    2    3    4    5    6    7    8    9
       English    0  185  525  883  997 1166 1283 1440 1558 1638
German_Deutsch    0  171  263  614  717  894 1013 1110 1213 1275
```

Your Turn: Working with the news and romance genres from the Brown Corpus, find out which days of the week are most newsworthy, and which are most romantic. Define a variable called `days` containing a list of days of the week, i.e., `['Monday', ...]`. Now tabulate the counts for these words using `cfd.tabulate(samples=days)`. Now try the same thing using `plot` in place of `tabulate`. You may control the output order of days with the help of an extra parameter: `conditions=['Monday', ...]`.

You may have noticed that the multiline expressions we have been using with conditional frequency distributions look like list comprehensions, but without the brackets. In general, when we use a list comprehension as a parameter to a function, like `set([w.lower for w in t])`, we are permitted to omit the square brackets and just write `set(w.lower() for w in t)`. (See the discussion of "generator expressions" in Section 4.2 for more about this.)

Generating Random Text with Bigrams

We can use a conditional frequency distribution to create a table of bigrams (word pairs, introduced in Section 1.3). The `bigrams()` function takes a list of words and builds a list of consecutive word pairs:

```
>>> sent = ['In', 'the', 'beginning', 'God', 'created', 'the', 'heaven',
...   'and', 'the', 'earth', '.']
>>> nltk.bigrams(sent)
[('In', 'the'), ('the', 'beginning'), ('beginning', 'God'), ('God', 'created'),
('created', 'the'), ('the', 'heaven'), ('heaven', 'and'), ('and', 'the'),
('the', 'earth'), ('earth', '.')]
```

In Example 2-1, we treat each word as a condition, and for each one we effectively create a frequency distribution over the following words. The function `generate_model()` contains a simple loop to generate text. When we call the function, we choose a word (such as `'living'`) as our initial context. Then, once inside the loop, we print the current value of the variable `word`, and reset `word` to be the most likely token in that context (using `max()`); next time through the loop, we use that word as our new context. As you can see by inspecting the output, this simple approach to text generation tends to get stuck in loops. Another method would be to randomly choose the next word from among the available words.

Example 2-1. Generating random text: This program obtains all bigrams from the text of the book of Genesis, then constructs a conditional frequency distribution to record which words are most likely to follow a given word; e.g., after the word living, *the most likely word is* creature; *the* `generate_model()` *function uses this data, and a seed word, to generate random text.*

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

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

```
>>> print cfd['living']
<FreqDist: 'creature': 7, 'thing': 4, 'substance': 2, ',': 1, '.': 1, 'soul': 1>
>>> generate_model(cfd, 'living')
living creature that he said , and the land of the land of the land
```

Conditional frequency distributions are a useful data structure for many NLP tasks. Their commonly used methods are summarized in Table 2-4.

Table 2-4. NLTK's conditional frequency distributions: Commonly used methods and idioms for defining, accessing, and visualizing a conditional frequency distribution of counters

Example	Description
`cfdist = ConditionalFreqDist(pairs)`	Create a conditional frequency distribution from a list of pairs
`cfdist.conditions()`	Alphabetically sorted list of conditions
`cfdist[condition]`	The frequency distribution for this condition
`cfdist[condition][sample]`	Frequency for the given sample for this condition
`cfdist.tabulate()`	Tabulate the conditional frequency distribution
`cfdist.tabulate(samples, conditions)`	Tabulation limited to the specified samples and conditions
`cfdist.plot()`	Graphical plot of the conditional frequency distribution
`cfdist.plot(samples, conditions)`	Graphical plot limited to the specified samples and conditions
`cfdist1 < cfdist2`	Test if samples in `cfdist1` occur less frequently than in `cfdist2`

2.3 More Python: Reusing Code

By this time you've probably typed and retyped a lot of code in the Python interactive interpreter. If you mess up when retyping a complex example, you have to enter it again. Using the arrow keys to access and modify previous commands is helpful but only goes so far. In this section, we see two important ways to reuse code: text editors and Python functions.

Creating Programs with a Text Editor

The Python interactive interpreter performs your instructions as soon as you type them. Often, it is better to compose a multiline program using a text editor, then ask Python to run the whole program at once. Using IDLE, you can do this by going to the File menu and opening a new window. Try this now, and enter the following one-line program:

```
print 'Monty Python'
```

Save this program in a file called *monty.py*, then go to the Run menu and select the command Run Module. (We'll learn what modules are shortly.) The result in the main IDLE window should look like this:

```
>>> ================================ RESTART ================================
>>>
Monty Python
>>>
```

You can also type `from monty import *` and it will do the same thing.

From now on, you have a choice of using the interactive interpreter or a text editor to create your programs. It is often convenient to test your ideas using the interpreter, revising a line of code until it does what you expect. Once you're ready, you can paste the code (minus any `>>>` or `...` prompts) into the text editor, continue to expand it, and finally save the program in a file so that you don't have to type it in again later. Give the file a short but descriptive name, using all lowercase letters and separating words with underscore, and using the *.py* filename extension, e.g., *monty_python.py*.

Important: Our inline code examples include the `>>>` and `...` prompts as if we are interacting directly with the interpreter. As they get more complicated, you should instead type them into the editor, without the prompts, and run them from the editor as shown earlier. When we provide longer programs in this book, we will leave out the prompts to remind you to type them into a file rather than using the interpreter. You can see this already in Example 2-1. Note that the example still includes a couple of lines with the Python prompt; this is the interactive part of the task where you inspect some data and invoke a function. Remember that all code samples like Example 2-1 are downloadable from *http://www.nltk.org/*.

Functions

Suppose that you work on analyzing text that involves different forms of the same word, and that part of your program needs to work out the plural form of a given singular noun. Suppose it needs to do this work in two places, once when it is processing some texts and again when it is processing user input.

Rather than repeating the same code several times over, it is more efficient and reliable to localize this work inside a **function**. A function is just a named block of code that performs some well-defined task, as we saw in Section 1.1. A function is usually defined to take some inputs, using special variables known as **parameters**, and it may produce a result, also known as a **return value**. We define a function using the keyword `def` followed by the function name and any input parameters, followed by the body of the function. Here's the function we saw in Section 1.1 (including the `import` statement that makes division behave as expected):

```
>>> from __future__ import division
>>> def lexical_diversity(text):
...     return len(text) / len(set(text))
```

We use the keyword `return` to indicate the value that is produced as output by the function. In this example, all the work of the function is done in the `return` statement. Here's an equivalent definition that does the same work using multiple lines of code. We'll change the parameter name from `text` to `my_text_data` to remind you that this is an arbitrary choice:

```
>>> def lexical_diversity(my_text_data):
...     word_count = len(my_text_data)
...     vocab_size = len(set(my_text_data))
...     diversity_score = word_count / vocab_size
...     return diversity_score
```

Notice that we've created some new variables inside the body of the function. These are **local variables** and are not accessible outside the function. So now we have defined a function with the name `lexical_diversity`. But just defining it won't produce any output! Functions do nothing until they are "called" (or "invoked").

Let's return to our earlier scenario, and actually define a simple function to work out English plurals. The function `plural()` in Example 2-2 takes a singular noun and generates a plural form, though it is not always correct. (We'll discuss functions at greater length in Section 4.4.)

Example 2-2. A Python function: This function tries to work out the plural form of any English noun; the keyword def (define) is followed by the function name, then a parameter inside parentheses, and a colon; the body of the function is the indented block of code; it tries to recognize patterns within the word and process the word accordingly; e.g., if the word ends with y, delete the y and add ies.

```
def plural(word):
    if word.endswith('y'):
        return word[:-1] + 'ies'
    elif word[-1] in 'sx' or word[-2:] in ['sh', 'ch']:
        return word + 'es'
    elif word.endswith('an'):
        return word[:-2] + 'en'
    else:
        return word + 's'
```

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

The `endswith()` function is always associated with a string object (e.g., `word` in Example 2-2). To call such functions, we give the name of the object, a period, and then the name of the function. These functions are usually known as **methods**.

Modules

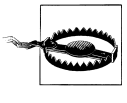
Over time you will find that you create a variety of useful little text-processing functions, and you end up copying them from old programs to new ones. Which file contains the latest version of the function you want to use? It makes life a lot easier if you can collect your work into a single place, and access previously defined functions without making copies.

To do this, save your function(s) in a file called (say) *textproc.py*. Now, you can access your work simply by importing it from the file:

```
>>> from textproc import plural
>>> plural('wish')
wishes
>>> plural('fan')
fen
```

Our plural function obviously has an error, since the plural of *fan* is *fans*. Instead of typing in a new version of the function, we can simply edit the existing one. Thus, at every stage, there is only one version of our plural function, and no confusion about which one is being used.

A collection of variable and function definitions in a file is called a Python **module**. A collection of related modules is called a **package**. NLTK's code for processing the Brown Corpus is an example of a module, and its collection of code for processing all the different corpora is an example of a package. NLTK itself is a set of packages, sometimes called a **library**.



Caution!

If you are creating a file to contain some of your Python code, do *not* name your file *nltk.py*: it may get imported in place of the "real" NLTK package. When it imports modules, Python first looks in the current directory (folder).

2.4 Lexical Resources

A lexicon, or lexical resource, is a collection of words and/or phrases along with associated information, such as part-of-speech and sense definitions. Lexical resources are secondary to texts, and are usually created and enriched with the help of texts. For example, if we have defined a text `my_text`, then `vocab = sorted(set(my_text))` builds the vocabulary of `my_text`, whereas `word_freq = FreqDist(my_text)` counts the frequency of each word in the text. Both `vocab` and `word_freq` are simple lexical resources. Similarly, a concordance like the one we saw in Section 1.1 gives us information about word usage that might help in the preparation of a dictionary. Standard terminology for lexicons is illustrated in Figure 2-5. A **lexical entry** consists of a **headword** (also known as a **lemma**) along with additional information, such as the part-of-speech and

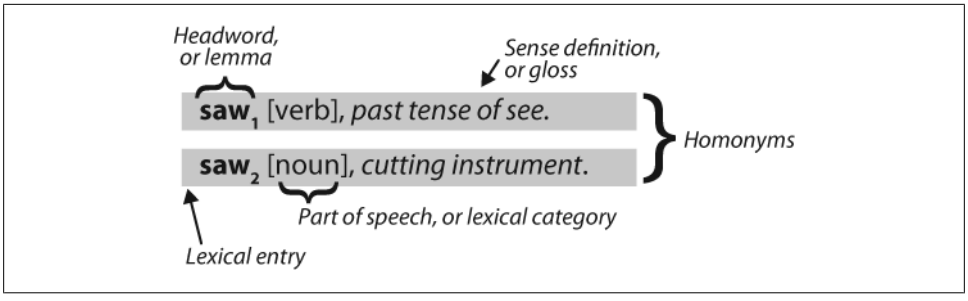


Figure 2-5. Lexicon terminology: Lexical entries for two lemmas having the same spelling (homonyms), providing part-of-speech and gloss information.

the sense definition. Two distinct words having the same spelling are called **homonyms**.

The simplest kind of lexicon is nothing more than a sorted list of words. Sophisticated lexicons include complex structure within and across the individual entries. In this section, we'll look at some lexical resources included with NLTK.

Wordlist Corpora

NLTK includes some corpora that are nothing more than wordlists. The Words Corpus is the */usr/dict/words* file from Unix, used by some spellcheckers. We can use it to find unusual or misspelled words in a text corpus, as shown in Example 2-3.

Example 2-3. Filtering a text: This program computes the vocabulary of a text, then removes all items that occur in an existing wordlist, leaving just the uncommon or misspelled words.

```
def unusual_words(text):
    text_vocab = set(w.lower() for w in text if w.isalpha())
    english_vocab = set(w.lower() for w in nltk.corpus.words.words())
    unusual = text_vocab.difference(english_vocab)
    return sorted(unusual)
```

```
>>> unusual_words(nltk.corpus.gutenberg.words('austen-sense.txt'))
['abbeyland', 'abhorrence', 'abominably', 'abridgement', 'accordant', 'accustomary',
'adieus', 'affability', 'affectedly', 'aggrandizement', 'alighted', 'allenham',
'amiably', 'annamaria', 'annuities', 'apologising', 'arbour', 'archness', ...]
>>> unusual_words(nltk.corpus.nps_chat.words())
['aaaaaaaaaaaaaaaaa', 'aaahhhh', 'abou', 'abourted', 'abs', 'ack', 'acros',
'actualy', 'adduser', 'addy', 'adoted', 'adreniline', 'ae', 'afe', 'affari', 'afk',
'agaibn', 'agurlwithbigguns', 'ahah', 'ahahah', 'ahahh', 'ahahha', 'ahem', 'ahh', ...]
```

There is also a corpus of **stopwords**, that is, high-frequency words such as *the*, *to*, and *also* that we sometimes want to filter out of a document before further processing. Stopwords usually have little lexical content, and their presence in a text fails to distinguish it from other texts.

```
>>> from nltk.corpus import stopwords
>>> stopwords.words('english')
['a', "a's", 'able', 'about', 'above', 'according', 'accordingly', 'across',
```

```
'actually', 'after', 'afterwards', 'again', 'against', "ain't", 'all', 'allow',
'allows', 'almost', 'alone', 'along', 'already', 'also', 'although', 'always', ...]
```

Let's define a function to compute what fraction of words in a text are *not* in the stopwords list:

```
>>> def content_fraction(text):
...     stopwords = nltk.corpus.stopwords.words('english')
...     content = [w for w in text if w.lower() not in stopwords]
...     return len(content) / len(text)
...
>>> content_fraction(nltk.corpus.reuters.words())
0.65997695393285261
```

Thus, with the help of stopwords, we filter out a third of the words of the text. Notice that we've combined two different kinds of corpus here, using a lexical resource to filter the content of a text corpus.

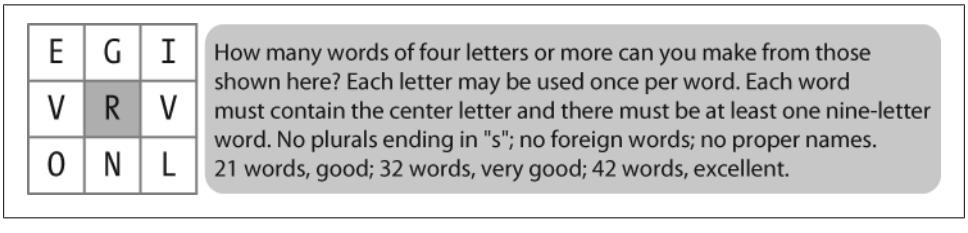


Figure 2-6. A word puzzle: A grid of randomly chosen letters with rules for creating words out of the letters; this puzzle is known as "Target."

A wordlist is useful for solving word puzzles, such as the one in Figure 2-6. Our program iterates through every word and, for each one, checks whether it meets the conditions. It is easy to check obligatory letter ❷ and length ❶ constraints (and we'll only look for words with six or more letters here). It is trickier to check that candidate solutions only use combinations of the supplied letters, especially since some of the supplied letters appear twice (here, the letter *v*). The `FreqDist` comparison method ❸ permits us to check that the frequency of each *letter* in the candidate word is less than or equal to the frequency of the corresponding letter in the puzzle.

```
>>> puzzle_letters = nltk.FreqDist('egivrvonl')
>>> obligatory = 'r'
>>> wordlist = nltk.corpus.words.words()
>>> [w for w in wordlist if len(w) >= 6 ❶
...                      and obligatory in w ❷
...                      and nltk.FreqDist(w) <= puzzle_letters] ❸
['glover', 'gorlin', 'govern', 'grovel', 'ignore', 'involver', 'lienor',
'linger', 'longer', 'lovering', 'noiler', 'overling', 'region', 'renvoi',
'revolving', 'ringle', 'roving', 'violer', 'virole']
```

One more wordlist corpus is the Names Corpus, containing 8,000 first names categorized by gender. The male and female names are stored in separate files. Let's find names that appear in both files, i.e., names that are ambiguous for gender:

```
>>> names = nltk.corpus.names
>>> names.fileids()
['female.txt', 'male.txt']
>>> male_names = names.words('male.txt')
>>> female_names = names.words('female.txt')
>>> [w for w in male_names if w in female_names]
['Abbey', 'Abbie', 'Abby', 'Addie', 'Adrian', 'Adrien', 'Ajay', 'Alex', 'Alexis',
'Alfie', 'Ali', 'Alix', 'Allie', 'Allyn', 'Andie', 'Andrea', 'Andy', 'Angel',
'Angie', 'Ariel', 'Ashley', 'Aubrey', 'Augustine', 'Austin', 'Averil', ...]
```

It is well known that names ending in the letter *a* are almost always female. We can see this and some other patterns in the graph in Figure 2-7, produced by the following code. Remember that `name[-1]` is the last letter of `name`.

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

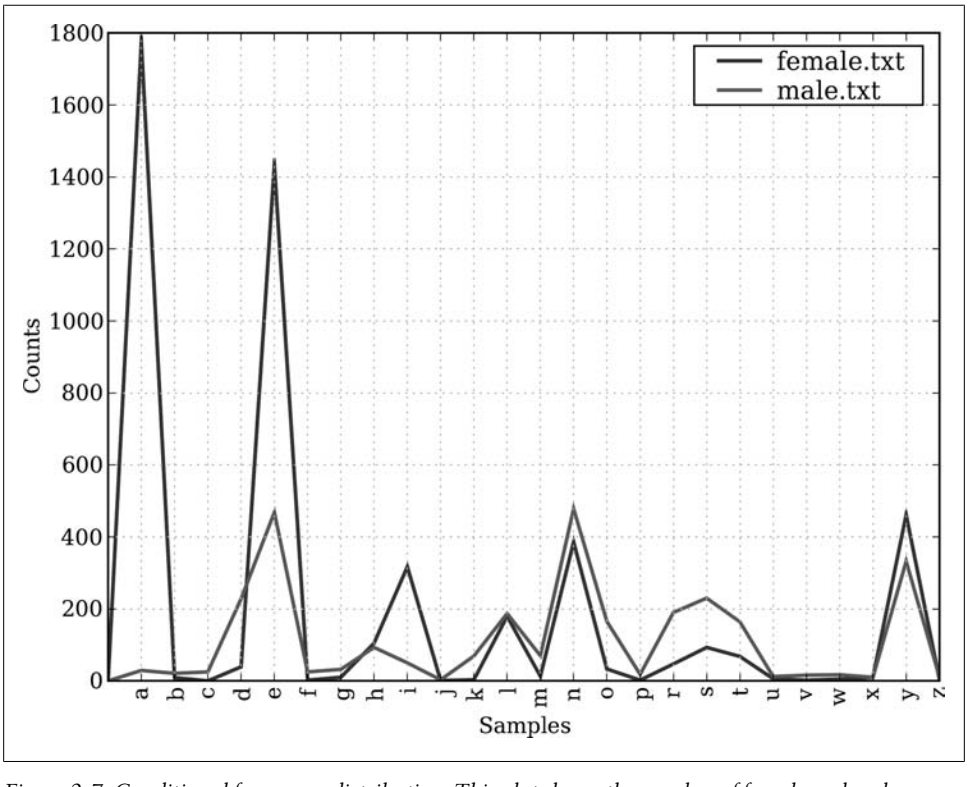


Figure 2-7. Conditional frequency distribution: This plot shows the number of female and male names ending with each letter of the alphabet; most names ending with a, e, *or* i *are female; names ending in* h *and* l *are equally likely to be male or female; names ending in* k, o, r, s, *and* t *are likely to be male.*

A Pronouncing Dictionary

A slightly richer kind of lexical resource is a table (or spreadsheet), containing a word plus some properties in each row. NLTK includes the CMU Pronouncing Dictionary for U.S. English, which was designed for use by speech synthesizers.

```
>>> entries = nltk.corpus.cmudict.entries()
>>> len(entries)
127012
>>> for entry in entries[39943:39951]:
...     print entry
...
('fir', ['F', 'ER1'])
('fire', ['F', 'AY1', 'ER0'])
('fire', ['F', 'AY1', 'R'])
('firearm', ['F', 'AY1', 'ER0', 'AA2', 'R', 'M'])
('firearm', ['F', 'AY1', 'R', 'AA2', 'R', 'M'])
('firearms', ['F', 'AY1', 'ER0', 'AA2', 'R', 'M', 'Z'])
('firearms', ['F', 'AY1', 'R', 'AA2', 'R', 'M', 'Z'])
('fireball', ['F', 'AY1', 'ER0', 'B', 'AO2', 'L'])
```

For each word, this lexicon provides a list of phonetic codes—distinct labels for each contrastive sound—known as *phones*. Observe that *fire* has two pronunciations (in U.S. English): the one-syllable `F AY1 R`, and the two-syllable `F AY1 ER0`. The symbols in the CMU Pronouncing Dictionary are from the *Arpabet*, described in more detail at *http://en.wikipedia.org/wiki/Arpabet*.

Each entry consists of two parts, and we can process these individually using a more complex version of the `for` statement. Instead of writing `for entry in entries:`, we replace `entry` with *two* variable names, `word, pron` ❶. Now, each time through the loop, `word` is assigned the first part of the entry, and `pron` is assigned the second part of the entry:

```
>>> for word, pron in entries: ❶
...     if len(pron) == 3: ❷
...         ph1, ph2, ph3 = pron ❸
...         if ph1 == 'P' and ph3 == 'T':
...             print word, ph2,
...
pait EY1 pat AE1 pate EY1 patt AE1 peart ER1 peat IY1 peet IY1 peete IY1 pert ER1
pet EH1 pete IY1 pett EH1 piet IY1 piette IY1 pit IH1 pitt IH1 pot AA1 pote OW1
pott AA1 pout AW1 puett UW1 purt ER1 put UH1 putt AH1
```

The program just shown scans the lexicon looking for entries whose pronunciation consists of three phones ❷. If the condition is true, it assigns the contents of `pron` to three new variables: `ph1`, `ph2`, and `ph3`. Notice the unusual form of the statement that does that work ❸.

Here's another example of the same `for` statement, this time used inside a list comprehension. This program finds all words whose pronunciation ends with a syllable sounding like *nicks*. You could use this method to find rhyming words.

```
>>> syllable = ['N', 'IH0', 'K', 'S']
>>> [word for word, pron in entries if pron[-4:] == syllable]
["atlantic's", 'audiotronics', 'avionics', 'beatniks', 'calisthenics', 'centronics',
'chetniks', "clinic's", 'clinics', 'conics', 'cynics', 'diasonics', "dominic's",
'ebonics', 'electronics', "electronics'", 'endotronics', "endotronics'", 'enix', ...]
```

Notice that the one pronunciation is spelled in several ways: *nics*, *niks*, *nix*, and even *ntic's* with a silent *t*, for the word *atlantic's*. Let's look for some other mismatches between pronunciation and writing. Can you summarize the purpose of the following examples and explain how they work?

```
>>> [w for w, pron in entries if pron[-1] == 'M' and w[-1] == 'n']
['autumn', 'column', 'condemn', 'damn', 'goddamn', 'hymn', 'solemn']
>>> sorted(set(w[:2] for w, pron in entries if pron[0] == 'N' and w[0] != 'n'))
['gn', 'kn', 'mn', 'pn']
```

The phones contain digits to represent primary stress (`1`), secondary stress (`2`), and no stress (`0`). As our final example, we define a function to extract the stress digits and then scan our lexicon to find words having a particular stress pattern.

```
>>> def stress(pron):
...     return [char for phone in pron for char in phone if char.isdigit()]
>>> [w for w, pron in entries if stress(pron) == ['0', '1', '0', '2', '0']]
['abbreviated', 'abbreviating', 'accelerated', 'accelerating', 'accelerator',
'accentuated', 'accentuating', 'accommodated', 'accommodating', 'accommodative',
'accumulated', 'accumulating', 'accumulative', 'accumulator', 'accumulators', ...]
>>> [w for w, pron in entries if stress(pron) == ['0', '2', '0', '1', '0']]
['abbreviation', 'abbreviations', 'abomination', 'abortifacient', 'abortifacients',
'academicians', 'accommodation', 'accommodations', 'accreditation', 'accreditations',
'accumulation', 'accumulations', 'acetylcholine', 'acetylcholine', 'adjudication', ...]
```

A subtlety of this program is that our user-defined function `stress()` is invoked inside the condition of a list comprehension. There is also a doubly nested `for` loop. There's a lot going on here, and you might want to return to this once you've had more experience using list comprehensions.

We can use a conditional frequency distribution to help us find minimally contrasting sets of words. Here we find all the *p* words consisting of three sounds ❷, and group them according to their first and last sounds ❶.

```
>>> p3 = [(pron[0]+'-'+pron[2], word) ❶
...       for (word, pron) in entries
...       if pron[0] == 'P' and len(pron) == 3] ❷
>>> cfd = nltk.ConditionalFreqDist(p3)
>>> for template in cfd.conditions():
...     if len(cfd[template]) > 10:
...         words = cfd[template].keys()
...         wordlist = ' '.join(words)
...         print template, wordlist[:70] + "..."
...
P-CH perch puche poche peach petsche poach pietsch putsch pautsch piche pet...
```

```
P-K pik peek pic pique paque polk perc poke perk pac pock poch purk pak pa...
P-L pil poehl pille pehl pol pall pohl pahl paul perl pale paille perle po...
P-N paine payne pon pain pin pawn pinn pun pine paign pen pyne pane penn p...
P-P pap paap pipp paup pape pup pep poop pop pipe paape popp pip peep pope...
P-R paar poor par poore pear pare pour peer pore parr por pair porr pier...
P-S pearse piece posts pasts peace perce pos pers pace puss pesce pass pur...
P-T pot puett pit pete putt pat purt pet peart pott pett pait pert pote pa...
P-Z pays p.s pao's pais paws p.'s pas pez paz pei's pose poise peas paiz p...
```

Rather than iterating over the whole dictionary, we can also access it by looking up particular words. We will use Python's dictionary data structure, which we will study systematically in Section 5.3. We look up a dictionary by specifying its name, followed by a **key** (such as the word `'fire'`) inside square brackets ❶.

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

If we try to look up a non-existent key ❷, we get a `KeyError`. This is similar to what happens when we index a list with an integer that is too large, producing an `IndexError`. The word *blog* is missing from the pronouncing dictionary, so we tweak our version by assigning a value for this key ❸ (this has no effect on the NLTK corpus; next time we access it, *blog* will still be absent).

We can use any lexical resource to process a text, e.g., to filter out words having some lexical property (like nouns), or mapping every word of the text. For example, the following text-to-speech function looks up each word of the text in the pronunciation dictionary:

```
>>> text = ['natural', 'language', 'processing']
>>> [ph for w in text for ph in prondict[w][0]]
['N', 'AE1', 'CH', 'ER0', 'AH0', 'L', 'L', 'AE1', 'NG', 'G', 'W', 'AH0', 'JH',
'P', 'R', 'AA1', 'S', 'EH0', 'S', 'IH0', 'NG']
```

Comparative Wordlists

Another example of a tabular lexicon is the **comparative wordlist**. NLTK includes so-called **Swadesh wordlists**, lists of about 200 common words in several languages. The languages are identified using an ISO 639 two-letter code.

```
>>> from nltk.corpus import swadesh
>>> swadesh.fileids()
['be', 'bg', 'bs', 'ca', 'cs', 'cu', 'de', 'en', 'es', 'fr', 'hr', 'it', 'la', 'mk',
'nl', 'pl', 'pt', 'ro', 'ru', 'sk', 'sl', 'sr', 'sw', 'uk']
>>> swadesh.words('en')
['I', 'you (singular), thou', 'he', 'we', 'you (plural)', 'they', 'this', 'that',
```

```
'here', 'there', 'who', 'what', 'where', 'when', 'how', 'not', 'all', 'many', 'some',
'few', 'other', 'one', 'two', 'three', 'four', 'five', 'big', 'long', 'wide', ...]
```

We can access cognate words from multiple languages using the `entries()` method, specifying a list of languages. With one further step we can convert this into a simple dictionary (we'll learn about `dict()` in Section 5.3).

```
>>> fr2en = swadesh.entries(['fr', 'en'])
>>> fr2en
[('je', 'I'), ('tu, vous', 'you (singular), thou'), ('il', 'he'), ...]
>>> translate = dict(fr2en)
>>> translate['chien']
'dog'
>>> translate['jeter']
'throw'
```

We can make our simple translator more useful by adding other source languages. Let's get the German-English and Spanish-English pairs, convert each to a dictionary using `dict()`, then *update* our original `translate` dictionary with these additional mappings:

```
>>> de2en = swadesh.entries(['de', 'en'])    # German-English
>>> es2en = swadesh.entries(['es', 'en'])    # Spanish-English
>>> translate.update(dict(de2en))
>>> translate.update(dict(es2en))
>>> translate['Hund']
'dog'
>>> translate['perro']
'dog'
```

We can compare words in various Germanic and Romance languages:

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

Shoebox and Toolbox Lexicons

Perhaps the single most popular tool used by linguists for managing data is *Toolbox*, previously known as *Shoebox* since it replaces the field linguist's traditional shoebox full of file cards. Toolbox is freely downloadable from *http://www.sil.org/computing/toolbox/*.

A Toolbox file consists of a collection of entries, where each entry is made up of one or more fields. Most fields are optional or repeatable, which means that this kind of lexical resource cannot be treated as a table or spreadsheet.

Here is a dictionary for the Rotokas language. We see just the first entry, for the word *kaa*, meaning "to gag":

```
>>> from nltk.corpus import toolbox
>>> toolbox.entries('rotokas.dic')
[('kaa', [('ps', 'V'), ('pt', 'A'), ('ge', 'gag'), ('tkp', 'nek i pas'),
('dcsv', 'true'), ('vx', '1'), ('sc', '???'), ('dt', '29/Oct/2005'),
('ex', 'Apoka ira kaaroi aioa-ia reoreopaoro.'),
('xp', 'Kaikai i pas long nek bilong Apoka bikos em i kaikai na toktok.'),
('xe', 'Apoka is gagging from food while talking.')]), ...]
```

Entries consist of a series of attribute-value pairs, such as `('ps', 'V')` to indicate that the part-of-speech is `'V'` (verb), and `('ge', 'gag')` to indicate that the gloss-into-English is `'gag'`. The last three pairs contain an example sentence in Rotokas and its translations into Tok Pisin and English.

The loose structure of Toolbox files makes it hard for us to do much more with them at this stage. XML provides a powerful way to process this kind of corpus, and we will return to this topic in Chapter 11.

The Rotokas language is spoken on the island of Bougainville, Papua New Guinea. This lexicon was contributed to NLTK by Stuart Robinson. Rotokas is notable for having an inventory of just 12 phonemes (contrastive sounds); see *http://en.wikipedia.org/wiki/Rotokas_language*

2.5 WordNet

WordNet is a semantically oriented dictionary of English, similar to a traditional thesaurus but with a richer structure. NLTK includes the English WordNet, with 155,287 words and 117,659 synonym sets. We'll begin by looking at synonyms and how they are accessed in WordNet.

Senses and Synonyms

Consider the sentence in (1a). If we replace the word *motorcar* in (1a) with *automobile*, to get (1b), the meaning of the sentence stays pretty much the same:

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

Since everything else in the sentence has remained unchanged, we can conclude that the words *motorcar* and *automobile* have the same meaning, i.e., they are **synonyms**. We can explore these words with the help of WordNet:

```
>>> from nltk.corpus import wordnet as wn
>>> wn.synsets('motorcar')
[Synset('car.n.01')]
```

Thus, *motorcar* has just one possible meaning and it is identified as `car.n.01`, the first noun sense of *car*. The entity `car.n.01` is called a **synset**, or "synonym set," a collection of synonymous words (or "lemmas"):

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

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

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

Although definitions help humans to understand the intended meaning of a synset, the *words* of the synset are often more useful for our programs. To eliminate ambiguity, we will identify these words as `car.n.01.automobile`, `car.n.01.motorcar`, and so on. This pairing of a synset with a word is called a lemma. We can get all the lemmas for a given synset ❶, look up a particular lemma ❷, get the synset corresponding to a lemma ❸, and get the "name" of a lemma ❹:

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

Unlike the words *automobile* and *motorcar*, which are unambiguous and have one synset, the word *car* is ambiguous, having five synsets:

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

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

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

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

The WordNet Hierarchy

WordNet synsets correspond to abstract concepts, and they don't always have corresponding words in English. These concepts are linked together in a hierarchy. Some concepts are very general, such as *Entity*, *State*, *Event*; these are called **unique beginners** or root synsets. Others, such as *gas guzzler* and *hatchback*, are much more specific. A small portion of a concept hierarchy is illustrated in Figure 2-8.

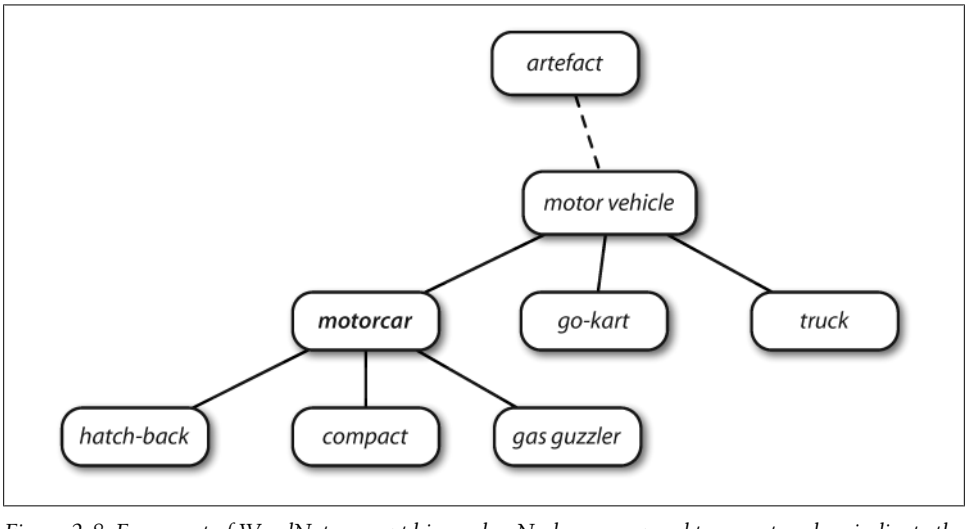


Figure 2-8. Fragment of WordNet concept hierarchy: Nodes correspond to synsets; edges indicate the hypernym/hyponym relation, i.e., the relation between superordinate and subordinate concepts.

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

```
>>> motorcar = wn.synset('car.n.01')
>>> types_of_motorcar = motorcar.hyponyms()
>>> types_of_motorcar[26]
Synset('ambulance.n.01')
>>> sorted([lemma.name for synset in types_of_motorcar for lemma in synset.lemmas])
['Model_T', 'S.U.V.', 'SUV', 'Stanley_Steamer', 'ambulance', 'beach_waggon',
'beach_wagon', 'bus', 'cab', 'compact', 'compact_car', 'convertible',
'coupe', 'cruiser', 'electric', 'electric_automobile', 'electric_car',
'estate_car', 'gas_guzzler', 'hack', 'hardtop', 'hatchback', 'heap',
'horseless_carriage', 'hot-rod', 'hot_rod', 'jalopy', 'jeep', 'landrover',
'limo', 'limousine', 'loaner', 'minicar', 'minivan', 'pace_car', 'patrol_car',
```

```
'phaeton', 'police_car', 'police_cruiser', 'prowl_car', 'race_car', 'racer',
'racing_car', 'roadster', 'runabout', 'saloon', 'secondhand_car', 'sedan',
'sport_car', 'sport_utility', 'sport_utility_vehicle', 'sports_car', 'squad_car',
'station_waggon', 'station_wagon', 'stock_car', 'subcompact', 'subcompact_car',
'taxi', 'taxicab', 'tourer', 'touring_car', 'two-seater', 'used-car', 'waggon',
'wagon']
```

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

```
>>> motorcar.hypernyms()
[Synset('motor_vehicle.n.01')]
>>> paths = motorcar.hypernym_paths()
>>> len(paths)
2
>>> [synset.name for synset in paths[0]]
['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01',
'instrumentality.n.03', 'container.n.01', 'wheeled_vehicle.n.01',
'self-propelled_vehicle.n.01', 'motor_vehicle.n.01', 'car.n.01']
>>> [synset.name for synset in paths[1]]
['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01',
'instrumentality.n.03', 'conveyance.n.03', 'vehicle.n.01', 'wheeled_vehicle.n.01',
'self-propelled_vehicle.n.01', 'motor_vehicle.n.01', 'car.n.01']
```

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

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

Your Turn: Try out NLTK's convenient graphical WordNet browser: `nltk.app.wordnet()`. Explore the WordNet hierarchy by following the hypernym and hyponym links.

More Lexical Relations

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

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

```
>>> wn.synset('tree.n.01').member_holonyms()
[Synset('forest.n.01')]
```

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

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

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

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

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

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

You can see the lexical relations, and the other methods defined on a synset, using `dir()`. For example, try `dir(wn.synset('harmony.n.02'))`.

Semantic Similarity

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

Recall that each synset has one or more hypernym paths that link it to a root hypernym such as `entity.n.01`. Two synsets linked to the same root may have several hypernyms in common (see Figure 2-8). If two synsets share a very specific hypernym—one that is low down in the hypernym hierarchy—they must be closely related.

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

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

```
>>> wn.synset('baleen_whale.n.01').min_depth()
14
>>> wn.synset('whale.n.02').min_depth()
13
>>> wn.synset('vertebrate.n.01').min_depth()
8
>>> wn.synset('entity.n.01').min_depth()
0
```

Similarity measures have been defined over the collection of WordNet synsets that incorporate this insight. For example, `path_similarity` assigns a score in the range `0`–`1` based on the shortest path that connects the concepts in the hypernym hierarchy (`-1` is returned in those cases where a path cannot be found). Comparing a synset with itself will return `1`. Consider the following similarity scores, relating *right whale* to *minke whale*, *orca*, *tortoise*, and *novel*. Although the numbers won't mean much, they decrease as we move away from the semantic space of sea creatures to inanimate objects.

```
>>> right.path_similarity(minke)
0.25
>>> right.path_similarity(orca)
0.16666666666666666
>>> right.path_similarity(tortoise)
0.07692307692307693
>>> right.path_similarity(novel)
0.043478260869565216
```

Several other similarity measures are available; you can type `help(wn)` for more information. NLTK also includes VerbNet, a hierarchical verb lexicon linked to WordNet. It can be accessed with `nltk.corpus.verbnet`.

2.6 Summary

- A text corpus is a large, structured collection of texts. NLTK comes with many corpora, e.g., the Brown Corpus, `nltk.corpus.brown`.
- Some text corpora are categorized, e.g., by genre or topic; sometimes the categories of a corpus overlap each other.
- A conditional frequency distribution is a collection of frequency distributions, each one for a different condition. They can be used for counting word frequencies, given a context or a genre.
- Python programs more than a few lines long should be entered using a text editor, saved to a file with a *.py* extension, and accessed using an `import` statement.
- Python functions permit you to associate a name with a particular block of code, and reuse that code as often as necessary.
- Some functions, known as "methods," are associated with an object, and we give the object name followed by a period followed by the method name, like this: `x.funct(y)`, e.g., `word.isalpha()`.
- To find out about some variable `v`, type `help(v)` in the Python interactive interpreter to read the help entry for this kind of object.
- WordNet is a semantically oriented dictionary of English, consisting of synonym sets—or synsets—and organized into a network.
- Some functions are not available by default, but must be accessed using Python's `import` statement.

2.7 Further Reading

Extra materials for this chapter are posted at *http://www.nltk.org/*, including links to freely available resources on the Web. The corpus methods are summarized in the Corpus HOWTO, at *http://www.nltk.org/howto*, and documented extensively in the online API documentation.

Significant sources of published corpora are the *Linguistic Data Consortium* (LDC) and the *European Language Resources Agency* (ELRA). Hundreds of annotated text and speech corpora are available in dozens of languages. Non-commercial licenses permit the data to be used in teaching and research. For some corpora, commercial licenses are also available (but for a higher fee).

These and many other language resources have been documented using OLAC Metadata, and can be searched via the OLAC home page at *http://www.language-archives.org/*. *Corpora List* (see *http://gandalf.aksis.uib.no/corpora/sub.html*) is a mailing list for discussions about corpora, and you can find resources by searching the list archives or posting to the list. The most complete inventory of the world's languages is *Ethnologue*, *http://www.ethnologue.com/*. Of 7,000 languages, only a few dozen have substantial digital resources suitable for use in NLP.

This chapter has touched on the field of **Corpus Linguistics**. Other useful books in this area include (Biber, Conrad, & Reppen, 1998), (McEnery, 2006), (Meyer, 2002), (Sampson & McCarthy, 2005), and (Scott & Tribble, 2006). Further readings in quantitative data analysis in linguistics are: (Baayen, 2008), (Gries, 2009), and (Woods, Fletcher, & Hughes, 1986).

The original description of WordNet is (Fellbaum, 1998). Although WordNet was originally developed for research in psycholinguistics, it is now widely used in NLP and Information Retrieval. WordNets are being developed for many other languages, as documented at *http://www.globalwordnet.org/*. For a study of WordNet similarity measures, see (Budanitsky & Hirst, 2006).

Other topics touched on in this chapter were phonetics and lexical semantics, and we refer readers to Chapters 7 and 20 of (Jurafsky & Martin, 2008).

2.8 Exercises

1. ○ Create a variable `phrase` containing a list of words. Experiment with the operations described in this chapter, including addition, multiplication, indexing, slicing, and sorting.
2. ○ Use the corpus module to explore `austen-persuasion.txt`. How many word tokens does this book have? How many word types?
3. ○ Use the Brown Corpus reader `nltk.corpus.brown.words()` or the Web Text Corpus reader `nltk.corpus.webtext.words()` to access some sample text in two different genres.
4. ○ Read in the texts of the *State of the Union* addresses, using the `state_union` corpus reader. Count occurrences of `men`, `women`, and `people` in each document. What has happened to the usage of these words over time?
5. ○ Investigate the holonym-meronym relations for some nouns. Remember that there are three kinds of holonym-meronym relation, so you need to use `member_meronyms()`, `part_meronyms()`, `substance_meronyms()`, `member_holonyms()`, `part_holonyms()`, and `substance_holonyms()`.
6. ○ In the discussion of comparative wordlists, we created an object called `translate`, which you could look up using words in both German and Italian in order

to get corresponding words in English. What problem might arise with this approach? Can you suggest a way to avoid this problem?

7. ○ According to Strunk and White's *Elements of Style*, the word *however*, used at the start of a sentence, means "in whatever way" or "to whatever extent," and not "nevertheless." They give this example of correct usage: *However you advise him, he will probably do as he thinks best.* (*http://www.bartleby.com/141/strunk3.html*) Use the concordance tool to study actual usage of this word in the various texts we have been considering. See also the *LanguageLog* posting "Fossilized prejudices about 'however'" at *http://itre.cis.upenn.edu/~myl/languagelog/archives/001913.html*.
8. ◑ Define a conditional frequency distribution over the Names Corpus that allows you to see which *initial* letters are more frequent for males versus females (see Figure 2-7).
9. ◑ Pick a pair of texts and study the differences between them, in terms of vocabulary, vocabulary richness, genre, etc. Can you find pairs of words that have quite different meanings across the two texts, such as *monstrous* in *Moby Dick* and in *Sense and Sensibility*?
10. ◑ Read the BBC News article: "UK's Vicky Pollards 'left behind'" at *http://news.bbc.co.uk/1/hi/education/6173441.stm*. The article gives the following statistic about teen language: "the top 20 words used, including yeah, no, but and like, account for around a third of all words." How many word types account for a third of all word tokens, for a variety of text sources? What do you conclude about this statistic? Read more about this on *LanguageLog*, at *http://itre.cis.upenn.edu/~myl/languagelog/archives/003993.html*.
11. ◑ Investigate the table of modal distributions and look for other patterns. Try to explain them in terms of your own impressionistic understanding of the different genres. Can you find other closed classes of words that exhibit significant differences across different genres?
12. ◑ The CMU Pronouncing Dictionary contains multiple pronunciations for certain words. How many distinct words does it contain? What fraction of words in this dictionary have more than one possible pronunciation?
13. ◑ What percentage of noun synsets have no hyponyms? You can get all noun synsets using `wn.all_synsets('n')`.
14. ◑ Define a function `supergloss(s)` that takes a synset `s` as its argument and returns a string consisting of the concatenation of the definition of `s`, and the definitions of all the hypernyms and hyponyms of `s`.
15. ◑ Write a program to find all words that occur at least three times in the Brown Corpus.
16. ◑ Write a program to generate a table of lexical diversity scores (i.e., token/type ratios), as we saw in Table 1-1. Include the full set of Brown Corpus genres

(`nltk.corpus.brown.categories()`). Which genre has the lowest diversity (greatest number of tokens per type)? Is this what you would have expected?

17. ◑ Write a function that finds the 50 most frequently occurring words of a text that are not stopwords.
18. ◑ Write a program to print the 50 most frequent bigrams (pairs of adjacent words) of a text, omitting bigrams that contain stopwords.
19. ◑ Write a program to create a table of word frequencies by genre, like the one given in Section 2.1 for modals. Choose your own words and try to find words whose presence (or absence) is typical of a genre. Discuss your findings.
20. ◑ Write a function `word_freq()` that takes a word and the name of a section of the Brown Corpus as arguments, and computes the frequency of the word in that section of the corpus.
21. ◑ Write a program to guess the number of syllables contained in a text, making use of the CMU Pronouncing Dictionary.
22. ◑ Define a function `hedge(text)` that processes a text and produces a new version with the word `'like'` between every third word.
23. ● **Zipf's Law**: Let $f(w)$ be the frequency of a word w in free text. Suppose that all the words of a text are ranked according to their frequency, with the most frequent word first. Zipf's Law states that the frequency of a word type is inversely proportional to its rank (i.e., $f \times r = k$, for some constant k). For example, the 50th most common word type should occur three times as frequently as the 150th most common word type.
    a. Write a function to process a large text and plot word frequency against word rank using `pylab.plot`. Do you confirm Zipf's law? (Hint: it helps to use a logarithmic scale.) What is going on at the extreme ends of the plotted line?
    b. Generate random text, e.g., using `random.choice("abcdefg ")`, taking care to include the space character. You will need to `import random` first. Use the string concatenation operator to accumulate characters into a (very) long string. Then tokenize this string, generate the Zipf plot as before, and compare the two plots. What do you make of Zipf's Law in the light of this?
24. ● Modify the text generation program in Example 2-1 further, to do the following tasks:

a. Store the *n* most likely words in a list `words`, then randomly choose a word from the list using `random.choice()`. (You will need to `import random` first.)

b. Select a particular genre, such as a section of the Brown Corpus or a Genesis translation, one of the Gutenberg texts, or one of the Web texts. Train the model on this corpus and get it to generate random text. You may have to experiment with different start words. How intelligible is the text? Discuss the strengths and weaknesses of this method of generating random text.

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

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

26. • What is the branching factor of the noun hypernym hierarchy? I.e., for every noun synset that has hyponyms—or children in the hypernym hierarchy—how many do they have on average? You can get all noun synsets using `wn.all_synsets('n')`.

27. • The polysemy of a word is the number of senses it has. Using WordNet, we can determine that the noun *dog* has seven senses with `len(wn.synsets('dog', 'n'))`. Compute the average polysemy of nouns, verbs, adjectives, and adverbs according to WordNet.

28. • Use one of the predefined similarity measures to score the similarity of each of the following pairs of words. Rank the pairs in order of decreasing similarity. How close is your ranking to the order given here, an order that was established experimentally by (Miller & Charles, 1998): car-automobile, gem-jewel, journey-voyage, boy-lad, coast-shore, asylum-madhouse, magician-wizard, midday-noon, furnace-stove, food-fruit, bird-cock, bird-crane, tool-implement, brother-monk, lad-brother, crane-implement, journey-car, monk-oracle, cemetery-woodland, food-rooster, coast-hill, forest-graveyard, shore-woodland, monk-slave, coast-forest, lad-wizard, chord-smile, glass-magician, rooster-voyage, noon-string.

CHAPTER 3
Processing Raw Text

The most important source of texts is undoubtedly the Web. 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.)

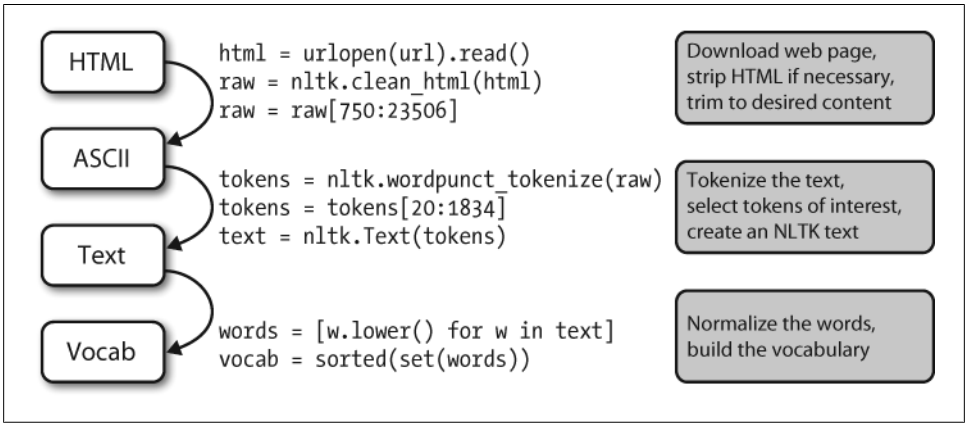


Figure 3-1. The processing pipeline: We open a URL and read its HTML content, remove the markup and select a slice of characters; this is then tokenized and optionally converted into an `nltk.Text` *object; we can also lowercase all the words and extract the vocabulary.*

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'
```

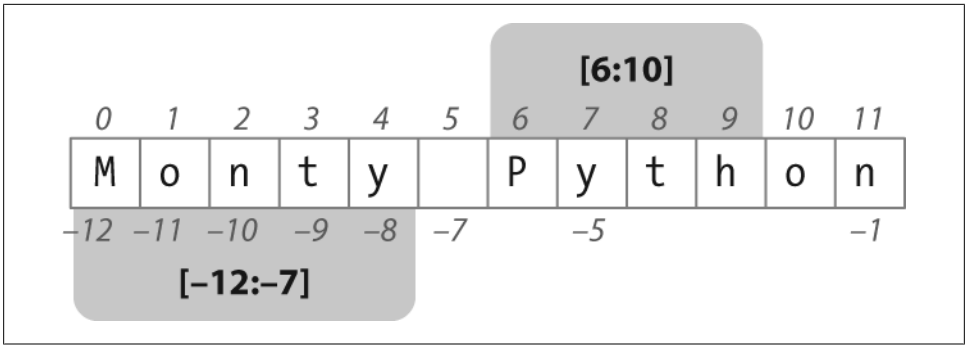


Figure 3-2. String slicing: The string Monty Python *is shown along with its positive and negative indexes; two substrings are selected using "slice" notation. The slice* `[m,n]` *contains the characters from position m through n-1.*

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.

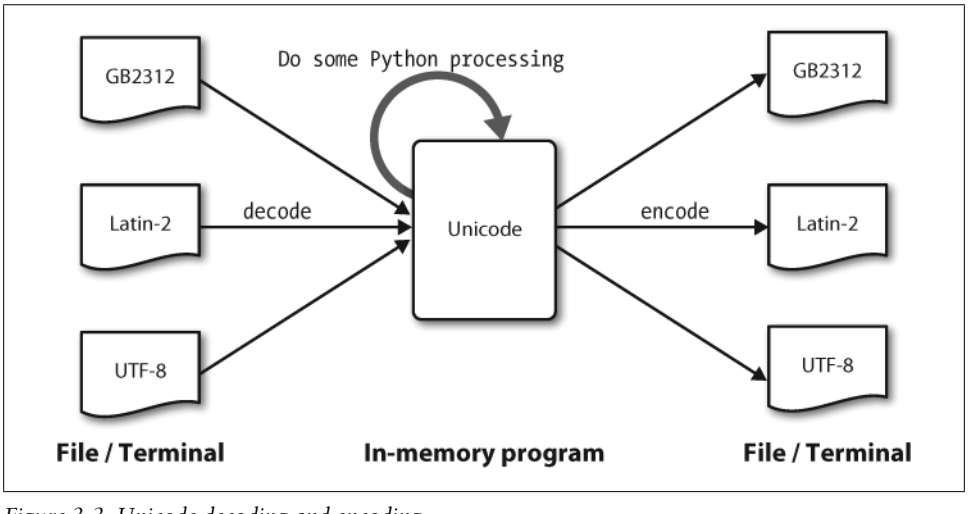


Figure 3-3. Unicode decoding and encoding.

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`».

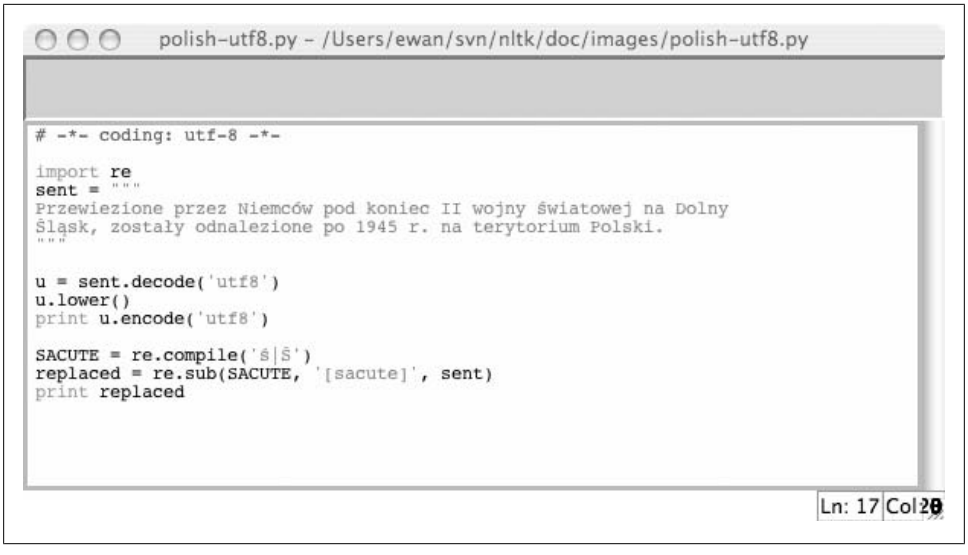


Figure 3-4. Unicode and IDLE: UTF-8 encoded string literals in the IDLE editor; this requires that an appropriate font is set in IDLE's preferences; here we have chosen Courier CE.

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.

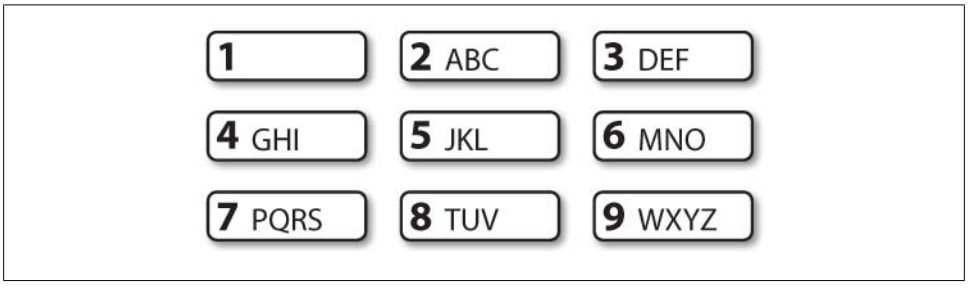


Figure 3-5. T9: Text on 9 keys.

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 = "0000000000000001000000000010000000000000000100000000000"
>>> seg2 = "0100100100100001001001000010100100010010000100010010000"
>>> 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.

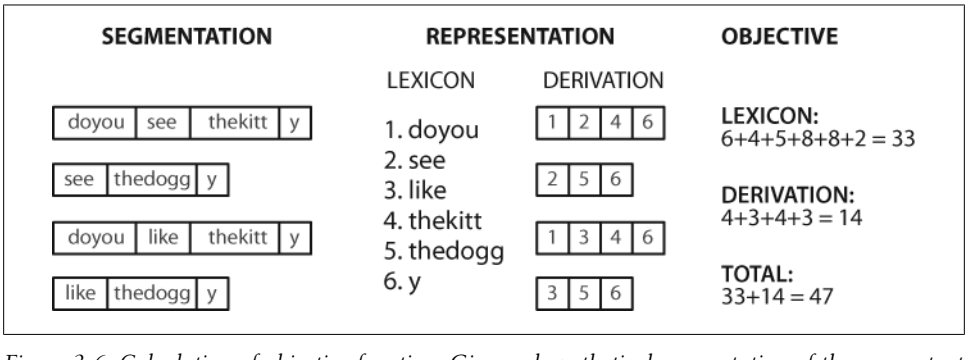


Figure 3-6. Calculation of objective function: Given a hypothetical segmentation of the source text (on the left), derive a lexicon and a derivation table that permit the source text to be reconstructed, then total up the number of characters used by each lexical item (including a boundary marker) and 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 = "0000000000000001000000000010000000000000001000000000"
>>> 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']
'0000100100000001001000000010000100010000000100010000000'
```

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.

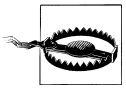
```
>>> output_file = open('output.txt', 'w')
>>> words = set(nltk.corpus.genesis.words('english-kjv.txt'))
>>> for word in sorted(words):
...     output_file.write(word + "\n")
```

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.

- If a regular expression string includes a backslash, you should tell Python not to preprocess the string, by using a raw string with an `r` prefix: `r'regexp'`.
- When backslash is used before certain characters, e.g., `\n`, this takes on a special meaning (newline character); however, when backslash is used before regular expression wildcards and operators, e.g., `\.`, `\|`, `\$`, these characters *lose* their special meaning and are matched literally.
- A string formatting expression `template % arg_tuple` consists of a format string `template` that contains conversion specifiers like `%-6s` and `%0.2d`.

3.11 Further Reading

Extra materials for this chapter are posted at *http://www.nltk.org/*, including links to freely available resources on the Web. Remember to consult the Python reference materials at *http://docs.python.org/*. (For example, this documentation covers "universal newline support," explaining how to work with the different newline conventions used by various operating systems.)

For more examples of processing words with NLTK, see the tokenization, stemming, and corpus HOWTOs at *http://www.nltk.org/howto*. Chapters 2 and 3 of (Jurafsky & Martin, 2008) contain more advanced material on regular expressions and morphology. For more extensive discussion of text processing with Python, see (Mertz, 2003). For information about normalizing non-standard words, see (Sproat et al., 2001).

There are many references for regular expressions, both practical and theoretical. For an introductory tutorial to using regular expressions in Python, see Kuchling's *Regular Expression HOWTO*, *http://www.amk.ca/python/howto/regex/*. For a comprehensive and detailed manual in using regular expressions, covering their syntax in most major programming languages, including Python, see (Friedl, 2002). Other presentations include Section 2.1 of (Jurafsky & Martin, 2008), and Chapter 3 of (Mertz, 2003).

There are many online resources for Unicode. Useful discussions of Python's facilities for handling Unicode are:

- PEP-100 *http://www.python.org/dev/peps/pep-0100/*
- Jason Orendorff, *Unicode for Programmers*, *http://www.jorendorff.com/articles/unicode/*
- A. M. Kuchling, *Unicode HOWTO*, *http://www.amk.ca/python/howto/unicode*
- Frederik Lundh, *Python Unicode Objects*, *http://effbot.org/zone/unicode-objects.htm*
- Joel Spolsky, *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*, *http://www.joelonsoftware.com/articles/Unicode.html*

The problem of tokenizing Chinese text is a major focus of SIGHAN, the ACL Special Interest Group on Chinese Language Processing (*http://sighan.org/*). Our method for segmenting English text follows (Brent & Cartwright, 1995); this work falls in the area of language acquisition (Niyogi, 2006).

Collocations are a special case of multiword expressions. A **multiword expression** is a small phrase whose meaning and other properties cannot be predicted from its words alone, e.g., *part-of-speech* (Baldwin & Kim, 2010).

Simulated annealing is a heuristic for finding a good approximation to the optimum value of a function in a large, discrete search space, based on an analogy with annealing in metallurgy. The technique is described in many Artificial Intelligence texts.

The approach to discovering hyponyms in text using search patterns like *x and other ys* is described by (Hearst, 1992).

3.12 Exercises

1. ○ Define a string `s = 'colorless'`. Write a Python statement that changes this to "colourless" using only the slice and concatenation operations.
2. ○ We can use the slice notation to remove morphological endings on words. For example, `'dogs'[:-1]` removes the last character of `dogs`, leaving `dog`. Use slice notation to remove the affixes from these words (we've inserted a hyphen to indicate the affix boundary, but omit this from your strings): `dish-es`, `run-ning`, `nation-ality`, `un-do`, `pre-heat`.
3. ○ We saw how we can generate an `IndexError` by indexing beyond the end of a string. Is it possible to construct an index that goes too far to the left, before the start of the string?
4. ○ We can specify a "step" size for the slice. The following returns every second character within the slice: `monty[6:11:2]`. It also works in the reverse direction: `monty[10:5:-2]`. Try these for yourself, and then experiment with different step values.
5. ○ What happens if you ask the interpreter to evaluate `monty[::-1]`? Explain why this is a reasonable result.
6. ○ Describe the class of strings matched by the following regular expressions:
   a. `[a-zA-Z]+`
   b. `[A-Z][a-z]*`
   c. `p[aeiou]{,2}t`
   d. `\d+(\.\d+)?`
   e. `([^aeiou][aeiou][^aeiou])*`
   f. `\w+|[^\w\s]+`

   Test your answers using `nltk.re_show()`.

7. ○ Write regular expressions to match the following classes of strings:
   a. A single determiner (assume that *a*, *an*, and *the* are the only determiners)
   b. An arithmetic expression using integers, addition, and multiplication, such as `2*3+8`
8. ○ Write a utility function that takes a URL as its argument, and returns the contents of the URL, with all HTML markup removed. Use `urllib.urlopen` to access the contents of the URL, e.g.:

```
raw_contents = urllib.urlopen('http://www.nltk.org/').read()
```

9. ○ Save some text into a file *corpus.txt*. Define a function `load(f)` that reads from the file named in its sole argument, and returns a string containing the text of the file.
   a. Use `nltk.regexp_tokenize()` to create a tokenizer that tokenizes the various kinds of punctuation in this text. Use one multiline regular expression inline comments, using the verbose flag `(?x)`.
   b. Use `nltk.regexp_tokenize()` to create a tokenizer that tokenizes the following kinds of expressions: monetary amounts; dates; names of people and organizations.
10. ○ Rewrite the following loop as a list comprehension:

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

11. ○ Define a string `raw` containing a sentence of your own choosing. Now, split `raw` on some character other than space, such as `'s'`.
12. ○ Write a `for` loop to print out the characters of a string, one per line.
13. ○ What is the difference between calling `split` on a string with no argument and one with `' '` as the argument, e.g., `sent.split()` versus `sent.split(' ')`? What happens when the string being split contains tab characters, consecutive space characters, or a sequence of tabs and spaces? (In IDLE you will need to use `'\t'` to enter a tab character.)
14. ○ Create a variable `words` containing a list of words. Experiment with `words.sort()` and `sorted(words)`. What is the difference?
15. ○ Explore the difference between strings and integers by typing the following at a Python prompt: `"3" * 7` and `3 * 7`. Try converting between strings and integers using `int("3")` and `str(3)`.
16. ○ Earlier, we asked you to use a text editor to create a file called *test.py*, containing the single line `monty = 'Monty Python'`. If you haven't already done this (or can't find the file), go ahead and do it now. Next, start up a new session with the Python

interpreter, and enter the expression `monty` at the prompt. You will get an error from the interpreter. Now, try the following (note that you have to leave off the *.py* part of the filename):

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

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

17. ○ What happens when the formatting strings `%6s` and `%-6s` are used to display strings that are longer than six characters?
18. ◑ Read in some text from a corpus, tokenize it, and print the list of all *wh*-word types that occur. (*wh*-words in English are used in questions, relative clauses, and exclamations: *who*, *which*, *what*, and so on.) Print them in order. Are any words duplicated in this list, because of the presence of case distinctions or punctuation?
19. ◑ Create a file consisting of words and (made up) frequencies, where each line consists of a word, the space character, and a positive integer, e.g., `fuzzy 53`. Read the file into a Python list using `open(filename).readlines()`. Next, break each line into its two fields using `split()`, and convert the number into an integer using `int()`. The result should be a list of the form: `[['fuzzy', 53], ...]`.
20. ◑ Write code to access a favorite web page and extract some text from it. For example, access a weather site and extract the forecast top temperature for your town or city today.
21. ◑ Write a function `unknown()` that takes a URL as its argument, and returns a list of unknown words that occur on that web page. In order to do this, extract all substrings consisting of lowercase letters (using `re.findall()`) and remove any items from this set that occur in the Words Corpus (`nltk.corpus.words`). Try to categorize these words manually and discuss your findings.
22. ◑ Examine the results of processing the URL *http://news.bbc.co.uk/* using the regular expressions suggested above. You will see that there is still a fair amount of non-textual data there, particularly JavaScript commands. You may also find that sentence breaks have not been properly preserved. Define further regular expressions that improve the extraction of text from this web page.
23. ◑ Are you able to write a regular expression to tokenize text in such a way that the word *don't* is tokenized into *do* and *n't*? Explain why this regular expression won't work: «`n't|\w+`».
24. ◑ Try to write code to convert text into *hAck3r*, using regular expressions and substitution, where `e` → `3`, `i` → `1`, `o` → `0`, `l` → `|`, `s` → `5`, `.` → `5w33t!`, `ate` → `8`. Normalize the text to lowercase before converting it. Add more substitutions of your own. Now try to map `s` to two different values: `$` for word-initial `s`, and `5` for word-internal `s`.

25. ◑ *Pig Latin* is a simple transformation of English text. Each word of the text is converted as follows: move any consonant (or consonant cluster) that appears at the start of the word to the end, then append *ay*, e.g., *string* → *ingstray*, *idle* → *idleay* (see http://en.wikipedia.org/wiki/Pig_Latin).
    a. Write a function to convert a word to Pig Latin.
    b. Write code that converts text, instead of individual words.
    c. Extend it further to preserve capitalization, to keep `qu` together (so that `quiet` becomes `ietquay`, for example), and to detect when `y` is used as a consonant (e.g., `yellow`) versus a vowel (e.g., `style`).
26. ◑ Download some text from a language that has vowel harmony (e.g., Hungarian), extract the vowel sequences of words, and create a vowel bigram table.
27. ◑ Python's `random` module includes a function `choice()` which randomly chooses an item from a sequence; e.g., `choice("aehh ")` will produce one of four possible characters, with the letter `h` being twice as frequent as the others. Write a generator expression that produces a sequence of 500 randomly chosen letters drawn from the string `"aehh "`, and put this expression inside a call to the `''.join()` function, to concatenate them into one long string. You should get a result that looks like uncontrolled sneezing or maniacal laughter: `he haha ee heheeh eha`. Use `split()` and `join()` again to normalize the whitespace in this string.
28. ◑ Consider the numeric expressions in the following sentence from the MedLine Corpus: *The corresponding free cortisol fractions in these sera were 4.53 +/- 0.15% and 8.16 +/- 0.23%, respectively*. Should we say that the numeric expression *4.53 +/- 0.15%* is three words? Or should we say that it's a single compound word? Or should we say that it is actually *nine* words, since it's read "four point five three, plus or minus fifteen percent"? Or should we say that it's not a "real" word at all, since it wouldn't appear in any dictionary? Discuss these different possibilities. Can you think of application domains that motivate at least two of these answers?
29. ◑ Readability measures are used to score the reading difficulty of a text, for the purposes of selecting texts of appropriate difficulty for language learners. Let us define μ_w to be the average number of letters per word, and μ_s to be the average number of words per sentence, in a given text. The Automated Readability Index (ARI) of the text is defined to be: $4.71\ \mu_w + 0.5\ \mu_s - 21.43$. Compute the ARI score for various sections of the Brown Corpus, including section `f` (popular lore) and `j` (learned). Make use of the fact that `nltk.corpus.brown.words()` produces a sequence of words, whereas `nltk.corpus.brown.sents()` produces a sequence of sentences.
30. ◑ Use the Porter Stemmer to normalize some tokenized text, calling the stemmer on each word. Do the same thing with the Lancaster Stemmer, and see if you observe any differences.
31. ◑ Define the variable `saying` to contain the list `['After', 'all', 'is', 'said', 'and', 'done', ',', 'more', 'is', 'said', 'than', 'done', '.']`. Process the list

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

32. ◑ Define a variable `silly` to contain the string: `'newly formed bland ideas are inexpressible in an infuriating way'`. (This happens to be the legitimate interpretation that bilingual English-Spanish speakers can assign to Chomsky's famous nonsense phrase *colorless green ideas sleep furiously*, according to Wikipedia). Now write code to perform the following tasks:
    a. Split `silly` into a list of strings, one per word, using Python's `split()` operation, and save this to a variable called `bland`.
    b. Extract the second letter of each word in `silly` and join them into a string, to get `'eoldrnnnna'`.
    c. Combine the words in `bland` back into a single string, using `join()`. Make sure the words in the resulting string are separated with whitespace.
    d. Print the words of `silly` in alphabetical order, one per line.
33. ◑ The `index()` function can be used to look up items in sequences. For example, `'inexpressible'.index('e')` tells us the index of the first position of the letter `e`.
    a. What happens when you look up a substring, e.g., `'inexpressible'.index('re')`?
    b. Define a variable `words` containing a list of words. Now use `words.index()` to look up the position of an individual word.
    c. Define a variable `silly` as in Exercise 32. Use the `index()` function in combination with list slicing to build a list `phrase` consisting of all the words up to (but not including) `in` in `silly`.
34. ◑ Write code to convert nationality adjectives such as *Canadian* and *Australian* to their corresponding nouns *Canada* and *Australia* (see *http://en.wikipedia.org/wiki/List_of_adjectival_forms_of_place_names*).
35. ◑ Read the LanguageLog post on phrases of the form *as best as p can* and *as best p can*, where *p* is a pronoun. Investigate this phenomenon with the help of a corpus and the `findall()` method for searching tokenized text described in Section 3.5. The post is at *http://itre.cis.upenn.edu/~myl/languagelog/archives/002733.html*.
36. ◑ Study the *lolcat* version of the book of Genesis, accessible as `nltk.corpus.genesis.words('lolcat.txt')`, and the rules for converting text into *lolspeak* at *http://www.lolcatbible.com/index.php?title=How_to_speak_lolcat*. Define regular expressions to convert English words into corresponding lolspeak words.
37. ◑ Read about the `re.sub()` function for string substitution using regular expressions, using `help(re.sub)` and by consulting the further readings for this chapter. Use `re.sub` in writing code to remove HTML tags from an HTML file, and to normalize whitespace.

38. ● An interesting challenge for tokenization is words that have been split across a linebreak. E.g., if *long-term* is split, then we have the string `long-\nterm`.
    a. Write a regular expression that identifies words that are hyphenated at a linebreak. The expression will need to include the `\n` character.
    b. Use `re.sub()` to remove the `\n` character from these words.
    c. How might you identify words that should not remain hyphenated once the newline is removed, e.g., `'encyclo-\npedia'`?
39. ● Read the Wikipedia entry on *Soundex*. Implement this algorithm in Python.
40. ● Obtain raw texts from two or more genres and compute their respective reading difficulty scores as in the earlier exercise on reading difficulty. E.g., compare ABC Rural News and ABC Science News (`nltk.corpus.abc`). Use Punkt to perform sentence segmentation.
41. ● Rewrite the following nested loop as a nested list comprehension:

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

42. ● Use WordNet to create a semantic index for a text collection. Extend the concordance search program in Example 3-1, indexing each word using the offset of its first synset, e.g., `wn.synsets('dog')[0].offset` (and optionally the offset of some of its ancestors in the hypernym hierarchy).
43. ● With the help of a multilingual corpus such as the Universal Declaration of Human Rights Corpus (`nltk.corpus.udhr`), along with NLTK's frequency distribution and rank correlation functionality (`nltk.FreqDist`, `nltk.spearman_correlation`), develop a system that guesses the language of a previously unseen text. For simplicity, work with a single character encoding and just a few languages.
44. ● Write a program that processes a text and discovers cases where a word has been used with a novel sense. For each word, compute the WordNet similarity between all synsets of the word and all synsets of the words in its context. (Note that this is a crude approach; doing it well is a difficult, open research problem.)
45. ● Read the article on normalization of non-standard words (Sproat et al., 2001), and implement a similar system for text normalization.

CHAPTER 4
Writing Structured Programs

By now you will have a sense of the capabilities of the Python programming language for processing natural language. However, if you're new to Python or to programming, you may still be wrestling with Python and not feel like you are in full control yet. In this chapter we'll address the following questions:

1. How can you write well-structured, readable programs that you and others will be able to reuse easily?
2. How do the fundamental building blocks work, such as loops, functions, and assignment?
3. What are some of the pitfalls with Python programming, and how can you avoid them?

Along the way, you will consolidate your knowledge of fundamental programming constructs, learn more about using features of the Python language in a natural and concise way, and learn some useful techniques in visualizing natural language data. As before, this chapter contains many examples and exercises (and as before, some exercises introduce new material). Readers new to programming should work through them carefully and consult other introductions to programming if necessary; experienced programmers can quickly skim this chapter.

In the other chapters of this book, we have organized the programming concepts as dictated by the needs of NLP. Here we revert to a more conventional approach, where the material is more closely tied to the structure of the programming language. There's not room for a complete presentation of the language, so we'll just focus on the language constructs and idioms that are most important for NLP.

4.1 Back to the Basics

Assignment

Assignment would seem to be the most elementary programming concept, not deserving a separate discussion. However, there are some surprising subtleties here. Consider the following code fragment:

```
>>> foo = 'Monty'
>>> bar = foo ❶
>>> foo = 'Python' ❷
>>> bar
'Monty'
```

This behaves exactly as expected. When we write `bar = foo` in the code ❶, the value of `foo` (the string `'Monty'`) is assigned to `bar`. That is, `bar` is a **copy** of `foo`, so when we overwrite `foo` with a new string `'Python'` on line ❷, the value of `bar` is not affected.

However, assignment statements do not always involve making copies in this way. Assignment always copies the value of an expression, but a value is not always what you might expect it to be. In particular, the "value" of a structured object such as a list is actually just a *reference* to the object. In the following example, ❶ assigns the reference of `foo` to the new variable `bar`. Now when we modify something inside `foo` on line ❷, we can see that the contents of `bar` have also been changed.

```
>>> foo = ['Monty', 'Python']
>>> bar = foo ❶
>>> foo[1] = 'Bodkin' ❷
>>> bar
['Monty', 'Bodkin']
```

The line `bar = foo` ❶ does not copy the contents of the variable, only its "object reference." To understand what is going on here, we need to know how lists are stored in the computer's memory. In Figure 4-1, we see that a list `foo` is a reference to an object stored at location 3133 (which is itself a series of pointers to other locations holding strings). When we assign `bar = foo`, it is just the object reference 3133 that gets copied. This behavior extends to other aspects of the language, such as parameter passing (Section 4.4).

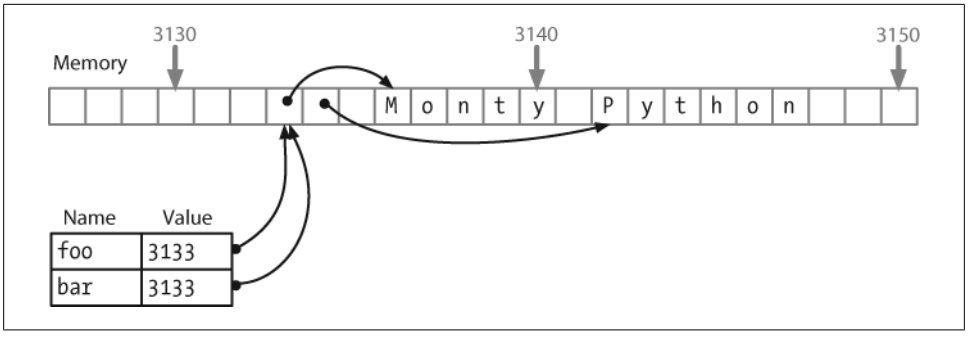


Figure 4-1. List assignment and computer memory: Two list objects `foo` *and* `bar` *reference the same location in the computer's memory; updating* `foo` *will also modify* `bar`*, and vice versa.*

Let's experiment some more, by creating a variable `empty` holding the empty list, then using it three times on the next line.

```
>>> empty = []
>>> nested = [empty, empty, empty]
>>> nested
[[], [], []]
>>> nested[1].append('Python')
>>> nested
[['Python'], ['Python'], ['Python']]
```

Observe that changing one of the items inside our nested list of lists changed them all. This is because each of the three elements is actually just a reference to one and the same list in memory.

Your Turn: Use multiplication to create a list of lists: `nested = [[]] * 3`. Now modify one of the elements of the list, and observe that all the elements are changed. Use Python's `id()` function to find out the numerical identifier for any object, and verify that `id(nested[0])`, `id(nested[1])`, and `id(nested[2])` are all the same.

Now, notice that when we assign a new value to one of the elements of the list, it does not propagate to the others:

```
>>> nested = [[]] * 3
>>> nested[1].append('Python')
>>> nested[1] = ['Monty']
>>> nested
[['Python'], ['Monty'], ['Python']]
```

We began with a list containing three references to a single empty list object. Then we modified that object by appending `'Python'` to it, resulting in a list containing three references to a single list object `['Python']`. Next, we *overwrote* one of those references with a reference to a new object `['Monty']`. This last step modified one of the three object references inside the nested list. However, the `['Python']` object wasn't changed,

and is still referenced from two places in our nested list of lists. It is crucial to appreciate this difference between modifying an object via an object reference and overwriting an object reference.

Important: To copy the items from a list `foo` to a new list `bar`, you can write `bar = foo[:]`. This copies the object references inside the list. To copy a structure without copying any object references, use `copy.deepcopy()`.

Equality

Python provides two ways to check that a pair of items are the same. The `is` operator tests for object identity. We can use it to verify our earlier observations about objects. First, we create a list containing several copies of the same object, and demonstrate that they are not only identical according to `==`, but also that they are one and the same object:

```
>>> size = 5
>>> python = ['Python']
>>> snake_nest = [python] * size
>>> snake_nest[0] == snake_nest[1] == snake_nest[2] == snake_nest[3] == snake_nest[4]
True
>>> snake_nest[0] is snake_nest[1] is snake_nest[2] is snake_nest[3] is snake_nest[4]
True
```

Now let's put a new python in this nest. We can easily show that the objects are not all identical:

```
>>> import random
>>> position = random.choice(range(size))
>>> snake_nest[position] = ['Python']
>>> snake_nest
[['Python'], ['Python'], ['Python'], ['Python'], ['Python']]
>>> snake_nest[0] == snake_nest[1] == snake_nest[2] == snake_nest[3] == snake_nest[4]
True
>>> snake_nest[0] is snake_nest[1] is snake_nest[2] is snake_nest[3] is snake_nest[4]
False
```

You can do several pairwise tests to discover which position contains the interloper, but the `id()` function makes detection is easier:

```
>>> [id(snake) for snake in snake_nest]
[513528, 533168, 513528, 513528, 513528]
```

This reveals that the second item of the list has a distinct identifier. If you try running this code snippet yourself, expect to see different numbers in the resulting list, and don't be surprised if the interloper is in a different position.

Having two kinds of equality might seem strange. However, it's really just the type-token distinction, familiar from natural language, here showing up in a programming language.

Conditionals

In the condition part of an `if` statement, a non-empty string or list is evaluated as true, while an empty string or list evaluates as false.

```
>>> mixed = ['cat', '', ['dog'], []]
>>> for element in mixed:
...     if element:
...         print element
...
cat
['dog']
```

That is, we *don't* need to say `if len(element) > 0:` in the condition.

What's the difference between using `if...elif` as opposed to using a couple of `if` statements in a row? Well, consider the following situation:

```
>>> animals = ['cat', 'dog']
>>> if 'cat' in animals:
...     print 1
... elif 'dog' in animals:
...     print 2
...
1
```

Since the `if` clause of the statement is satisfied, Python never tries to evaluate the `elif` clause, so we never get to print out `2`. By contrast, if we replaced the `elif` by an `if`, then we would print out both `1` and `2`. So an `elif` clause potentially gives us more information than a bare `if` clause; when it evaluates to true, it tells us not only that the condition is satisfied, but also that the condition of the main `if` clause was *not* satisfied.

The functions `all()` and `any()` can be applied to a list (or other sequence) to check whether all or any items meet some condition:

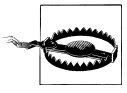
```
>>> sent = ['No', 'good', 'fish', 'goes', 'anywhere', 'without', 'a', 'porpoise', '.']
>>> all(len(w) > 4 for w in sent)
False
>>> any(len(w) > 4 for w in sent)
True
```

4.2 Sequences

So far, we have seen two kinds of sequence object: strings and lists. Another kind of sequence is called a **tuple**. Tuples are formed with the comma operator ❶, and typically enclosed using parentheses. We've actually seen them in the previous chapters, and sometimes referred to them as "pairs," since there were always two members. However, tuples can have any number of members. Like lists and strings, tuples can be indexed ❷ and sliced ❸, and have a length ❹.

```
>>> t = 'walk', 'fem', 3 ❶
>>> t
('walk', 'fem', 3)
```

```
>>> t[0] ❷
'walk'
>>> t[1:] ❸
('fem', 3)
>>> len(t) ❹
```



Caution!

Tuples are constructed using the comma operator. Parentheses are a more general feature of Python syntax, designed for grouping. A tuple containing the single element `'snark'` is defined by adding a trailing comma, like this: `'snark',`. The empty tuple is a special case, and is defined using empty parentheses `()`.

Let's compare strings, lists, and tuples directly, and do the indexing, slice, and length operation on each type:

```
>>> raw = 'I turned off the spectroroute'
>>> text = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> pair = (6, 'turned')
>>> raw[2], text[3], pair[1]
('t', 'the', 'turned')
>>> raw[-3:], text[-3:], pair[-3:]
('ute', ['off', 'the', 'spectroroute'], (6, 'turned'))
>>> len(raw), len(text), len(pair)
(29, 5, 2)
```

Notice in this code sample that we computed multiple values on a single line, separated by commas. These comma-separated expressions are actually just tuples—Python allows us to omit the parentheses around tuples if there is no ambiguity. When we print a tuple, the parentheses are always displayed. By using tuples in this way, we are implicitly aggregating items together.

Your Turn: Define a set, e.g., using `set(text)`, and see what happens when you convert it to a list or iterate over its members.

Operating on Sequence Types

We can iterate over the items in a sequence `s` in a variety of useful ways, as shown in Table 4-1.

Table 4-1. Various ways to iterate over sequences

Python expression	Comment
`for item in s`	Iterate over the items of `s`
`for item in sorted(s)`	Iterate over the items of `s` in order
`for item in set(s)`	Iterate over unique elements of `s`

Python expression	Comment
`for item in reversed(s)`	Iterate over elements of `s` in reverse
`for item in set(s).difference(t)`	Iterate over elements of `s` not in `t`
`for item in random.shuffle(s)`	Iterate over elements of `s` in random order

The sequence functions illustrated in Table 4-1 can be combined in various ways; for example, to get unique elements of `s` sorted in reverse, use `reversed(sorted(set(s)))`.

We can convert between these sequence types. For example, `tuple(s)` converts any kind of sequence into a tuple, and `list(s)` converts any kind of sequence into a list. We can convert a list of strings to a single string using the `join()` function, e.g., `':'.join(words)`.

Some other objects, such as a `FreqDist`, can be converted into a sequence (using `list()`) and support iteration:

```
>>> raw = 'Red lorry, yellow lorry, red lorry, yellow lorry.'
>>> text = nltk.word_tokenize(raw)
>>> fdist = nltk.FreqDist(text)
>>> list(fdist)
['lorry', ',', 'yellow', '.', 'Red', 'red']
>>> for key in fdist:
...     print fdist[key],
...
4 3 2 1 1 1
```

In the next example, we use tuples to re-arrange the contents of our list. (We can omit the parentheses because the comma has higher precedence than assignment.)

```
>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> words[2], words[3], words[4] = words[3], words[4], words[2]
>>> words
['I', 'turned', 'the', 'spectroroute', 'off']
```

This is an idiomatic and readable way to move items inside a list. It is equivalent to the following traditional way of doing such tasks that does not use tuples (notice that this method needs a temporary variable `tmp`).

```
>>> tmp = words[2]
>>> words[2] = words[3]
>>> words[3] = words[4]
>>> words[4] = tmp
```

As we have seen, Python has sequence functions such as `sorted()` and `reversed()` that rearrange the items of a sequence. There are also functions that modify the *structure* of a sequence, which can be handy for language processing. Thus, `zip()` takes the items of two or more sequences and "zips" them together into a single list of pairs. Given a sequence `s`, `enumerate(s)` returns pairs consisting of an index and the item at that index.

```
>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> tags = ['noun', 'verb', 'prep', 'det', 'noun']
>>> zip(words, tags)
```

```
[('I', 'noun'), ('turned', 'verb'), ('off', 'prep'),
('the', 'det'), ('spectroroute', 'noun')]
>>> list(enumerate(words))
[(0, 'I'), (1, 'turned'), (2, 'off'), (3, 'the'), (4, 'spectroroute')]
```

For some NLP tasks it is necessary to cut up a sequence into two or more parts. For instance, we might want to "train" a system on 90% of the data and test it on the remaining 10%. To do this we decide the location where we want to cut the data ❶, then cut the sequence at that location ❷.

```
>>> text = nltk.corpus.nps_chat.words()
>>> cut = int(0.9 * len(text)) ❶
>>> training_data, test_data = text[:cut], text[cut:] ❷
>>> text == training_data + test_data ❸
True
>>> len(training_data) / len(test_data) ❹
9
```

We can verify that none of the original data is lost during this process, nor is it duplicated ❸. We can also verify that the ratio of the sizes of the two pieces is what we intended ❹.

Combining Different Sequence Types

Let's combine our knowledge of these three sequence types, together with list comprehensions, to perform the task of sorting the words in a string by their length.

```
>>> words = 'I turned off the spectroroute'.split() ❶
>>> wordlens = [(len(word), word) for word in words] ❷
>>> wordlens.sort() ❸
>>> ' '.join(w for (_, w) in wordlens) ❹
'I off the turned spectroroute'
```

Each of the preceding lines of code contains a significant feature. A simple string is actually an object with methods defined on it, such as `split()` ❶. We use a list comprehension to build a list of tuples ❷, where each tuple consists of a number (the word length) and the word, e.g., `(3, 'the')`. We use the `sort()` method ❸ to sort the list in place. Finally, we discard the length information and join the words back into a single string ❹. (The underscore ❹ is just a regular Python variable, but we can use underscore by convention to indicate that we will not use its value.)

We began by talking about the commonalities in these sequence types, but the previous code illustrates important differences in their roles. First, strings appear at the beginning and the end: this is typical in the context where our program is reading in some text and producing output for us to read. Lists and tuples are used in the middle, but for different purposes. A list is typically a sequence of objects all having the *same type*, of *arbitrary length*. We often use lists to hold sequences of words. In contrast, a tuple is typically a collection of objects of *different types*, of *fixed length*. We often use a tuple to hold a **record**, a collection of different **fields** relating to some entity. This distinction between the use of lists and tuples takes some getting used to, so here is another example:

```
>>> lexicon = [
...     ('the', 'det', ['Di:', 'D@']),
...     ('off', 'prep', ['Qf', 'O:f'])
... ]
```

Here, a lexicon is represented as a list because it is a collection of objects of a single type—lexical entries—of no predetermined length. An individual entry is represented as a tuple because it is a collection of objects with different interpretations, such as the orthographic form, the part-of-speech, and the pronunciations (represented in the SAMPA computer-readable phonetic alphabet; see *http://www.phon.ucl.ac.uk/home/sampa/*). Note that these pronunciations are stored using a list. (Why?)

A good way to decide when to use tuples versus lists is to ask whether the interpretation of an item depends on its position. For example, a tagged token combines two strings having different interpretations, and we choose to interpret the first item as the token and the second item as the tag. Thus we use tuples like this: `('grail', 'noun')`. A tuple of the form `('noun', 'grail')` would be non-sensical since it would be a word `noun` tagged `grail`. In contrast, the elements of a text are all tokens, and position is not significant. Thus we use lists like this: `['venetian', 'blind']`. A list of the form `['blind', 'venetian']` would be equally valid. The linguistic meaning of the words might be different, but the interpretation of list items as tokens is unchanged.

The distinction between lists and tuples has been described in terms of usage. However, there is a more fundamental difference: in Python, lists are **mutable**, whereas tuples are **immutable**. In other words, lists can be modified, whereas tuples cannot. Here are some of the operations on lists that do in-place modification of the list:

```
>>> lexicon.sort()
>>> lexicon[1] = ('turned', 'VBD', ['t3:nd', 't3`nd'])
>>> del lexicon[0]
```

Your Turn: Convert `lexicon` to a tuple, using `lexicon = tuple(lexicon)`, then try each of the operations, to confirm that none of them is permitted on tuples.

Generator Expressions

We've been making heavy use of list comprehensions, for compact and readable processing of texts. Here's an example where we tokenize and normalize a text:

```
>>> text = '''"When I use a word," Humpty Dumpty said in rather a scornful tone,
... "it means just what I choose it to mean - neither more nor less."'''
>>> [w.lower() for w in nltk.word_tokenize(text)]
['"', 'when', 'i', 'use', 'a', 'word', ',', '"', 'humpty', 'dumpty', 'said', ...]
```

Suppose we now want to process these words further. We can do this by inserting the preceding expression inside a call to some other function ❶, but Python allows us to omit the brackets ❷.

```
>>> max([w.lower() for w in nltk.word_tokenize(text)]) ❶
'word'
>>> max(w.lower() for w in nltk.word_tokenize(text)) ❷
'word'
```

The second line uses a **generator expression**. This is more than a notational convenience: in many language processing situations, generator expressions will be more efficient. In ❶, storage for the list object must be allocated before the value of `max()` is computed. If the text is very large, this could be slow. In ❷, the data is streamed to the calling function. Since the calling function simply has to find the maximum value—the word that comes latest in lexicographic sort order—it can process the stream of data without having to store anything more than the maximum value seen so far.

4.3 Questions of Style

Programming is as much an art as a science. The undisputed "bible" of programming, a 2,500 page multivolume work by Donald Knuth, is called *The Art of Computer Programming*. Many books have been written on *Literate Programming*, recognizing that humans, not just computers, must read and understand programs. Here we pick up on some issues of programming style that have important ramifications for the readability of your code, including code layout, procedural versus declarative style, and the use of loop variables.

Python Coding Style

When writing programs you make many subtle choices about names, spacing, comments, and so on. When you look at code written by other people, needless differences in style make it harder to interpret the code. Therefore, the designers of the Python language have published a style guide for Python code, available at *http://www.python.org/dev/peps/pep-0008/*. The underlying value presented in the style guide is *consistency*, for the purpose of maximizing the readability of code. We briefly review some of its key recommendations here, and refer readers to the full guide for detailed discussion with examples.

Code layout should use four spaces per indentation level. You should make sure that when you write Python code in a file, you avoid tabs for indentation, since these can be misinterpreted by different text editors and the indentation can be messed up. Lines should be less than 80 characters long; if necessary, you can break a line inside parentheses, brackets, or braces, because Python is able to detect that the line continues over to the next line, as in the following examples:

```
>>> cv_word_pairs = [(cv, w) for w in rotokas_words
...                          for cv in re.findall('[ptksvr][aeiou]', w)]
```

```
>>> cfd = nltk.ConditionalFreqDist(
...           (genre, word)
...           for genre in brown.categories()
...           for word in brown.words(categories=genre))

>>> ha_words = ['aaahhhh', 'ah', 'ahah', 'ahahah', 'ahh', 'ahhahahaha',
...             'ahhh', 'ahhhh', 'ahhhhhh', 'ahhhhhhhhhhhhh', 'ha',
...             'haaa', 'hah', 'haha', 'hahaaa', 'hahah', 'hahaha']
```

If you need to break a line outside parentheses, brackets, or braces, you can often add extra parentheses, and you can always add a backslash at the end of the line that is broken:

```
>>> if (len(syllables) > 4 and len(syllables[2]) == 3 and
...    syllables[2][2] in [aeiou] and syllables[2][3] == syllables[1][3]):
...     process(syllables)
>>> if len(syllables) > 4 and len(syllables[2]) == 3 and \
...    syllables[2][2] in [aeiou] and syllables[2][3] == syllables[1][3]:
...     process(syllables)
```

Typing spaces instead of tabs soon becomes a chore. Many programming editors have built-in support for Python, and can automatically indent code and highlight any syntax errors (including indentation errors). For a list of Python-aware editors, please see *http://wiki.python.org/moin/PythonEditors*.

Procedural Versus Declarative Style

We have just seen how the same task can be performed in different ways, with implications for efficiency. Another factor influencing program development is *programming style*. Consider the following program to compute the average length of words in the Brown Corpus:

```
>>> tokens = nltk.corpus.brown.words(categories='news')
>>> count = 0
>>> total = 0
>>> for token in tokens:
...     count += 1
...     total += len(token)
>>> print total / count
4.2765382469
```

In this program we use the variable `count` to keep track of the number of tokens seen, and `total` to store the combined length of all words. This is a low-level style, not far removed from machine code, the primitive operations performed by the computer's CPU. The two variables are just like a CPU's registers, accumulating values at many intermediate stages, values that are meaningless until the end. We say that this program is written in a *procedural* style, dictating the machine operations step by step. Now consider the following program that computes the same thing:

```
>>> total = sum(len(t) for t in tokens)
>>> print total / len(tokens)
4.2765382469
```

The first line uses a generator expression to sum the token lengths, while the second line computes the average as before. Each line of code performs a complete, meaningful task, which can be understood in terms of high-level properties like: "`total` is the sum of the lengths of the tokens." Implementation details are left to the Python interpreter. The second program uses a built-in function, and constitutes programming at a more abstract level; the resulting code is more declarative. Let's look at an extreme example:

```
>>> word_list = []
>>> len_word_list = len(word_list)
>>> i = 0
>>> while i < len(tokens):
...     j = 0
...     while j < len_word_list and word_list[j] < tokens[i]:
...         j += 1
...     if j == 0 or tokens[i] != word_list[j]:
...         word_list.insert(j, tokens[i])
...         len_word_list += 1
...     i += 1
```

The equivalent declarative version uses familiar built-in functions, and its purpose is instantly recognizable:

```
>>> word_list = sorted(set(tokens))
```

Another case where a loop counter seems to be necessary is for printing a counter with each line of output. Instead, we can use `enumerate()`, which processes a sequence `s` and produces a tuple of the form `(i, s[i])` for each item in `s`, starting with `(0, s[0])`. Here we enumerate the keys of the frequency distribution, and capture the integer-string pair in the variables `rank` and `word`. We print `rank+1` so that the counting appears to start from `1`, as required when producing a list of ranked items.

```
>>> fd = nltk.FreqDist(nltk.corpus.brown.words())
>>> cumulative = 0.0
>>> for rank, word in enumerate(fd):
...     cumulative += fd[word] * 100 / fd.N()
...     print "%3d %6.2f%% %s" % (rank+1, cumulative, word)
...     if cumulative > 25:
...         break
...
  1   5.40% the
  2  10.42% ,
  3  14.67% .
  4  17.78% of
  5  20.19% and
  6  22.40% to
  7  24.29% a
  8  25.97% in
```

It's sometimes tempting to use loop variables to store a maximum or minimum value seen so far. Let's use this method to find the longest word in a text.

```
>>> text = nltk.corpus.gutenberg.words('milton-paradise.txt')
>>> longest = ''
>>> for word in text:
...     if len(word) > len(longest):
...         longest = word
>>> longest
'unextinguishable'
```

However, a more transparent solution uses two list comprehensions, both having forms that should be familiar by now:

```
>>> maxlen = max(len(word) for word in text)
>>> [word for word in text if len(word) == maxlen]
['unextinguishable', 'transubstantiate', 'inextinguishable', 'incomprehensible']
```

Note that our first solution found the first word having the longest length, while the second solution found *all* of the longest words (which is usually what we would want). Although there's a theoretical efficiency difference between the two solutions, the main overhead is reading the data into main memory; once it's there, a second pass through the data is effectively instantaneous. We also need to balance our concerns about program efficiency with programmer efficiency. A fast but cryptic solution will be harder to understand and maintain.

Some Legitimate Uses for Counters

There are cases where we still want to use loop variables in a list comprehension. For example, we need to use a loop variable to extract successive overlapping n-grams from a list:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> n = 3
>>> [sent[i:i+n] for i in range(len(sent)-n+1)]
[['The', 'dog', 'gave'],
 ['dog', 'gave', 'John'],
 ['gave', 'John', 'the'],
 ['John', 'the', 'newspaper']]
```

It is quite tricky to get the range of the loop variable right. Since this is a common operation in NLP, NLTK supports it with functions `bigrams(text)` and `trigrams(text)`, and a general-purpose `ngrams(text, n)`.

Here's an example of how we can use loop variables in building multidimensional structures. For example, to build an array with *m* rows and *n* columns, where each cell is a set, we could use a nested list comprehension:

```
>>> m, n = 3, 7
>>> array = [[set() for i in range(n)] for j in range(m)]
>>> array[2][5].add('Alice')
>>> pprint.pprint(array)
[[set([]), set([]), set([]), set([]), set([]), set([]), set([])],
 [set([]), set([]), set([]), set([]), set([]), set([]), set([])],
 [set([]), set([]), set([]), set([]), set([]), set(['Alice']), set([])]]
```

Observe that the loop variables `i` and `j` are not used anywhere in the resulting object; they are just needed for a syntactically correct `for` statement. As another example of this usage, observe that the expression `['very' for i in range(3)]` produces a list containing three instances of `'very'`, with no integers in sight.

Note that it would be incorrect to do this work using multiplication, for reasons concerning object copying that were discussed earlier in this section.

```
>>> array = [[set()] * n] * m
>>> array[2][5].add(7)
>>> pprint.pprint(array)
[[set([7]), set([7]), set([7]), set([7]), set([7]), set([7]), set([7])],
 [set([7]), set([7]), set([7]), set([7]), set([7]), set([7]), set([7])],
 [set([7]), set([7]), set([7]), set([7]), set([7]), set([7]), set([7])]]
```

Iteration is an important programming device. It is tempting to adopt idioms from other languages. However, Python offers some elegant and highly readable alternatives, as we have seen.

4.4 Functions: The Foundation of Structured Programming

Functions provide an effective way to package and reuse program code, as already explained in Section 2.3. For example, suppose we find that we often want to read text from an HTML file. This involves several steps: opening the file, reading it in, normalizing whitespace, and stripping HTML markup. We can collect these steps into a function, and give it a name such as `get_text()`, as shown in Example 4-1.

Example 4-1. Read text from a file.

```
import re
def get_text(file):
    """Read text from a file, normalizing whitespace and stripping HTML markup."""
    text = open(file).read()
    text = re.sub('\s+', ' ', text)
    text = re.sub(r'<.*?>', ' ', text)
    return text
```

Now, any time we want to get cleaned-up text from an HTML file, we can just call `get_text()` with the name of the file as its only argument. It will return a string, and we can assign this to a variable, e.g., `contents = get_text("test.html")`. Each time we want to use this series of steps, we only have to call the function.

Using functions has the benefit of saving space in our program. More importantly, our choice of name for the function helps make the program *readable*. In the case of the preceding example, whenever our program needs to read cleaned-up text from a file we don't have to clutter the program with four lines of code; we simply need to call `get_text()`. This naming helps to provide some "semantic interpretation"—it helps a reader of our program to see what the program "means."

Notice that this example function definition contains a string. The first string inside a function definition is called a **docstring**. Not only does it document the purpose of the function to someone reading the code, it is accessible to a programmer who has loaded the code from a file:

```
>>> help(get_text)
Help on function get_text:

get_text(file)
    Read text from a file, normalizing whitespace
    and stripping HTML markup.
```

We have seen that functions help to make our work reusable and readable. They also help make it *reliable*. When we reuse code that has already been developed and tested, we can be more confident that it handles a variety of cases correctly. We also remove the risk of forgetting some important step or introducing a bug. The program that calls our function also has increased reliability. The author of that program is dealing with a shorter program, and its components behave transparently.

To summarize, as its name suggests, a function captures functionality. It is a segment of code that can be given a meaningful name and which performs a well-defined task. Functions allow us to abstract away from the details, to see a bigger picture, and to program more effectively.

The rest of this section takes a closer look at functions, exploring the mechanics and discussing ways to make your programs easier to read.

Function Inputs and Outputs

We pass information to functions using a function's parameters, the parenthesized list of variables and constants following the function's name in the function definition. Here's a complete example:

```
>>> def repeat(msg, num):  ❶
...     return ' '.join([msg] * num)
>>> monty = 'Monty Python'
>>> repeat(monty, 3) ❷
'Monty Python Monty Python Monty Python'
```

We first define the function to take two parameters, `msg` and `num` ❶. Then, we call the function and pass it two arguments, `monty` and `3` ❷; these arguments fill the "placeholders" provided by the parameters and provide values for the occurrences of `msg` and `num` in the function body.

It is not necessary to have any parameters, as we see in the following example:

```
>>> def monty():
...     return "Monty Python"
>>> monty()
'Monty Python'
```

A function usually communicates its results back to the calling program via the `return` statement, as we have just seen. To the calling program, it looks as if the function call had been replaced with the function's result:

```
>>> repeat(monty(), 3)
'Monty Python Monty Python Monty Python'
>>> repeat('Monty Python', 3)
'Monty Python Monty Python Monty Python'
```

A Python function is not required to have a return statement. Some functions do their work as a side effect, printing a result, modifying a file, or updating the contents of a parameter to the function (such functions are called "procedures" in some other programming languages).

Consider the following three sort functions. The third one is dangerous because a programmer could use it without realizing that it had modified its input. In general, functions should modify the contents of a parameter (`my_sort1()`), or return a value (`my_sort2()`), but not both (`my_sort3()`).

```
>>> def my_sort1(mylist):      # good: modifies its argument, no return value
...     mylist.sort()
>>> def my_sort2(mylist):      # good: doesn't touch its argument, returns value
...     return sorted(mylist)
>>> def my_sort3(mylist):      # bad: modifies its argument and also returns it
...     mylist.sort()
...     return mylist
```

Parameter Passing

Back in Section 4.1, you saw that assignment works on values, but that the value of a structured object is a *reference* to that object. The same is true for functions. Python interprets function parameters as values (this is known as **call-by-value**). In the following code, `set_up()` has two parameters, both of which are modified inside the function. We begin by assigning an empty string to `w` and an empty dictionary to `p`. After calling the function, `w` is unchanged, while `p` is changed:

```
>>> def set_up(word, properties):
...     word = 'lolcat'
...     properties.append('noun')
...     properties = 5
...
>>> w = ''
>>> p = []
>>> set_up(w, p)
>>> w
''
>>> p
['noun']
```

Notice that `w` was not changed by the function. When we called `set_up(w, p)`, the value of `w` (an empty string) was assigned to a new variable `word`. Inside the function, the value

of `word` was modified. However, that change did not propagate to `w`. This parameter passing is identical to the following sequence of assignments:

```
>>> w = ''
>>> word = w
>>> word = 'lolcat'
>>> w
''
```

Let's look at what happened with the list `p`. When we called `set_up(w, p)`, the value of `p` (a reference to an empty list) was assigned to a new local variable `properties`, so both variables now reference the same memory location. The function modifies `properties`, and this change is also reflected in the value of `p`, as we saw. The function also assigned a new value to properties (the number `5`); this did not modify the contents at that memory location, but created a new local variable. This behavior is just as if we had done the following sequence of assignments:

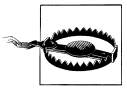
```
>>> p = []
>>> properties = p
>>> properties.append['noun']
>>> properties = 5
>>> p
['noun']
```

Thus, to understand Python's call-by-value parameter passing, it is enough to understand how assignment works. Remember that you can use the `id()` function and `is` operator to check your understanding of object identity after each statement.

Variable Scope

Function definitions create a new local **scope** for variables. When you assign to a new variable inside the body of a function, the name is defined only within that function. The name is not visible outside the function, or in other functions. This behavior means you can choose variable names without being concerned about collisions with names used in your other function definitions.

When you refer to an existing name from within the body of a function, the Python interpreter first tries to resolve the name with respect to the names that are local to the function. If nothing is found, the interpreter checks whether it is a global name within the module. Finally, if that does not succeed, the interpreter checks whether the name is a Python built-in. This is the so-called **LGB rule** of name resolution: local, then global, then built-in.



Caution!

A function can create a new global variable, using the `global` declaration. However, this practice should be avoided as much as possible. Defining global variables inside a function introduces dependencies on context and limits the portability (or reusability) of the function. In general you should use parameters for function inputs and return values for function outputs.

Checking Parameter Types

Python does not force us to declare the type of a variable when we write a program, and this permits us to define functions that are flexible about the type of their arguments. For example, a tagger might expect a sequence of words, but it wouldn't care whether this sequence is expressed as a list, a tuple, or an iterator (a new sequence type that we'll discuss later).

However, often we want to write programs for later use by others, and want to program in a defensive style, providing useful warnings when functions have not been invoked correctly. The author of the following `tag()` function assumed that its argument would always be a string.

```
>>> def tag(word):
...     if word in ['a', 'the', 'all']:
...         return 'det'
...     else:
...         return 'noun'
...
>>> tag('the')
'det'
>>> tag('knight')
'noun'
>>> tag(["'Tis", 'but', 'a', 'scratch']) ❶
'noun'
```

The function returns sensible values for the arguments `'the'` and `'knight'`, but look what happens when it is passed a list ❶—it fails to complain, even though the result which it returns is clearly incorrect. The author of this function could take some extra steps to ensure that the `word` parameter of the `tag()` function is a string. A naive approach would be to check the type of the argument using `if not type(word) is str`, and if `word` is not a string, to simply return Python's special empty value, `None`. This is a slight improvement, because the function is checking the type of the argument, and trying to return a "special" diagnostic value for the wrong input. However, it is also dangerous because the calling program may not detect that `None` is intended as a "special" value, and this diagnostic return value may then be propagated to other parts of the program with unpredictable consequences. This approach also fails if the word is a Unicode string, which has type `unicode`, not `str`. Here's a better solution, using an `assert` statement together with Python's `basestring` type that generalizes over both `unicode` and `str`.

```
>>> def tag(word):
...     assert isinstance(word, basestring), "argument to tag() must be a string"
...     if word in ['a', 'the', 'all']:
...         return 'det'
...     else:
...         return 'noun'
```

If the `assert` statement fails, it will produce an error that cannot be ignored, since it halts program execution. Additionally, the error message is easy to interpret. Adding

assertions to a program helps you find logical errors, and is a kind of **defensive programming**. A more fundamental approach is to document the parameters to each function using docstrings, as described later in this section.

Functional Decomposition

Well-structured programs usually make extensive use of functions. When a block of program code grows longer than 10–20 lines, it is a great help to readability if the code is broken up into one or more functions, each one having a clear purpose. This is analogous to the way a good essay is divided into paragraphs, each expressing one main idea.

Functions provide an important kind of abstraction. They allow us to group multiple actions into a single, complex action, and associate a name with it. (Compare this with the way we combine the actions of *go* and *bring back* into a single more complex action *fetch*.) When we use functions, the main program can be written at a higher level of abstraction, making its structure transparent, as in the following:

```
>>> data = load_corpus()
>>> results = analyze(data)
>>> present(results)
```

Appropriate use of functions makes programs more readable and maintainable. Additionally, it becomes possible to reimplement a function—replacing the function's body with more efficient code—without having to be concerned with the rest of the program.

Consider the `freq_words` function in Example 4-2. It updates the contents of a frequency distribution that is passed in as a parameter, and it also prints a list of the *n* most frequent words.

Example 4-2. Poorly designed function to compute frequent words.

```
def freq_words(url, freqdist, n):
    text = nltk.clean_url(url)
    for word in nltk.word_tokenize(text):
        freqdist.inc(word.lower())
    print freqdist.keys()[:n]
```

```
>>> constitution = "http://www.archives.gov/national-archives-experience" \
...                "/charters/constitution_transcript.html"
>>> fd = nltk.FreqDist()
>>> freq_words(constitution, fd, 20)
['the', 'of', 'charters', 'bill', 'constitution', 'rights', ',',
'declaration', 'impact', 'freedom', '-', 'making', 'independence']
```

This function has a number of problems. The function has two side effects: it modifies the contents of its second parameter, and it prints a selection of the results it has computed. The function would be easier to understand and to reuse elsewhere if we initialize the `FreqDist()` object inside the function (in the same place it is populated), and if we moved the selection and display of results to the calling program. In Example 4-3 we **refactor** this function, and simplify its interface by providing a single `url` parameter.

Example 4-3. Well-designed function to compute frequent words.

```
def freq_words(url):
    freqdist = nltk.FreqDist()
    text = nltk.clean_url(url)
    for word in nltk.word_tokenize(text):
        freqdist.inc(word.lower())
    return freqdist
```

```
>>> fd = freq_words(constitution)
>>> print fd.keys()[:20]
['the', 'of', 'charters', 'bill', 'constitution', 'rights', ',',
'declaration', 'impact', 'freedom', '-', 'making', 'independence']
```

Note that we have now simplified the work of `freq_words` to the point that we can do its work with three lines of code:

```
>>> words = nltk.word_tokenize(nltk.clean_url(constitution))
>>> fd = nltk.FreqDist(word.lower() for word in words)
>>> fd.keys()[:20]
['the', 'of', 'charters', 'bill', 'constitution', 'rights', ',',
'declaration', 'impact', 'freedom', '-', 'making', 'independence']
```

Documenting Functions

If we have done a good job at decomposing our program into functions, then it should be easy to describe the purpose of each function in plain language, and provide this in the docstring at the top of the function definition. This statement should not explain how the functionality is implemented; in fact, it should be possible to reimplement the function using a different method without changing this statement.

For the simplest functions, a one-line docstring is usually adequate (see Example 4-1). You should provide a triple-quoted string containing a complete sentence on a single line. For non-trivial functions, you should still provide a one-sentence summary on the first line, since many docstring processing tools index this string. This should be followed by a blank line, then a more detailed description of the functionality (see *http://www.python.org/dev/peps/pep-0257/* for more information on docstring conventions).

Docstrings can include a **doctest block**, illustrating the use of the function and the expected output. These can be tested automatically using Python's `docutils` module. Docstrings should document the type of each parameter to the function, and the return type. At a minimum, that can be done in plain text. However, note that NLTK uses the "epytext" markup language to document parameters. This format can be automatically converted into richly structured API documentation (see *http://www.nltk.org/*), and includes special handling of certain "fields," such as `@param`, which allow the inputs and outputs of functions to be clearly documented. Example 4-4 illustrates a complete docstring.

Example 4-4. Illustration of a complete docstring, consisting of a one-line summary, a more detailed explanation, a doctest example, and epytext markup specifying the parameters, types, return type, and exceptions.

```
def accuracy(reference, test):
    """
    Calculate the fraction of test items that equal the corresponding reference items.

    Given a list of reference values and a corresponding list of test values,
    return the fraction of corresponding values that are equal.
    In particular, return the fraction of indexes
    {0<i<=len(test)} such that C{test[i] == reference[i]}.

    >>> accuracy(['ADJ', 'N', 'V', 'N'], ['N', 'N', 'V', 'ADJ'])
    0.5

@param reference: An ordered list of reference values.
@type reference: C{list}
@param test: A list of values to compare against the corresponding
    reference values.
@type test: C{list}
@rtype: C{float}
@raise ValueError: If C{reference} and C{length} do not have the
    same length.
"""

if len(reference) != len(test):
    raise ValueError("Lists must have the same length.")
num_correct = 0
for x, y in izip(reference, test):
    if x == y:
        num_correct += 1
return float(num_correct) / len(reference)
```

4.5 Doing More with Functions

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

Functions As Arguments

So far the arguments we have passed into functions have been simple objects, such as strings, or structured objects, such as lists. Python also lets us pass a function as an argument to another function. Now we can abstract out the operation, and apply a *different operation* on the *same data*. As the following examples show, we can pass the built-in function `len()` or a user-defined function `last_letter()` as arguments to another function:

```
>>> sent = ['Take', 'care', 'of', 'the', 'sense', ',', 'and', 'the',
...         'sounds', 'will', 'take', 'care', 'of', 'themselves', '.']
>>> def extract_property(prop):
...     return [prop(word) for word in sent]
...
```

```
>>> extract_property(len)
[4, 4, 2, 3, 5, 1, 3, 3, 6, 4, 4, 4, 2, 10, 1]
>>> def last_letter(word):
...     return word[-1]
>>> extract_property(last_letter)
['e', 'e', 'f', 'e', 'e', ',', 'd', 'e', 's', 'l', 'e', 'e', 'f', 's', '.']
```

The objects `len` and `last_letter` can be passed around like lists and dictionaries. Notice that parentheses are used after a function name only if we are invoking the function; when we are simply treating the function as an object, these are omitted.

Python provides us with one more way to define functions as arguments to other functions, so-called **lambda expressions**. Supposing there was no need to use the `last_letter()` function in multiple places, and thus no need to give it a name. Let's suppose we can equivalently write the following:

```
>>> extract_property(lambda w: w[-1])
['e', 'e', 'f', 'e', 'e', ',', 'd', 'e', 's', 'l', 'e', 'e', 'f', 's', '.']
```

Our next example illustrates passing a function to the `sorted()` function. When we call the latter with a single argument (the list to be sorted), it uses the built-in comparison function `cmp()`. However, we can supply our own sort function, e.g., to sort by decreasing length.

```
>>> sorted(sent)
[',', '.', 'Take', 'and', 'care', 'care', 'of', 'of', 'sense', 'sounds',
'take', 'the', 'the', 'themselves', 'will']
>>> sorted(sent, cmp)
[',', '.', 'Take', 'and', 'care', 'care', 'of', 'of', 'sense', 'sounds',
'take', 'the', 'the', 'themselves', 'will']
>>> sorted(sent, lambda x, y: cmp(len(y), len(x)))
['themselves', 'sounds', 'sense', 'Take', 'care', 'will', 'take', 'care',
'the', 'and', 'the', 'of', 'of', ',', '.']
```

Accumulative Functions

These functions start by initializing some storage, and iterate over input to build it up, before returning some final object (a large structure or aggregated result). A standard way to do this is to initialize an empty list, accumulate the material, then return the list, as shown in function `search1()` in Example 4-5.

Example 4-5. Accumulating output into a list.

```
def search1(substring, words):
    result = []
    for word in words:
        if substring in word:
            result.append(word)
    return result

def search2(substring, words):
    for word in words:
        if substring in word:
            yield word
```

```
print "search1:"
for item in search1('zz', nltk.corpus.brown.words()):
    print item
print "search2:"
for item in search2('zz', nltk.corpus.brown.words()):
    print item
```

The function `search2()` is a generator. The first time this function is called, it gets as far as the `yield` statement and pauses. The calling program gets the first word and does any necessary processing. Once the calling program is ready for another word, execution of the function is continued from where it stopped, until the next time it encounters a `yield` statement. This approach is typically more efficient, as the function only generates the data as it is required by the calling program, and does not need to allocate additional memory to store the output (see the earlier discussion of generator expressions).

Here's a more sophisticated example of a generator which produces all permutations of a list of words. In order to force the `permutations()` function to generate all its output, we wrap it with a call to `list()` ❶.

```
>>> def permutations(seq):
...     if len(seq) <= 1:
...         yield seq
...     else:
...         for perm in permutations(seq[1:]):
...             for i in range(len(perm)+1):
...                 yield perm[:i] + seq[0:1] + perm[i:]
...
>>> list(permutations(['police', 'fish', 'buffalo'])) ❶
[['police', 'fish', 'buffalo'], ['fish', 'police', 'buffalo'],
 ['fish', 'buffalo', 'police'], ['police', 'buffalo', 'fish'],
 ['buffalo', 'police', 'fish'], ['buffalo', 'fish', 'police']]
```

The `permutations` function uses a technique called recursion, discussed later in Section 4.7. The ability to generate permutations of a set of words is useful for creating data to test a grammar (Chapter 8).

Higher-Order Functions

Python provides some higher-order functions that are standard features of functional programming languages such as Haskell. We illustrate them here, alongside the equivalent expression using list comprehensions.

Let's start by defining a function `is_content_word()` which checks whether a word is from the open class of content words. We use this function as the first parameter of `filter()`, which applies the function to each item in the sequence contained in its second parameter, and retains only the items for which the function returns `True`.

```
>>> def is_content_word(word):
...     return word.lower() not in ['a', 'of', 'the', 'and', 'will', ',', '.']
>>> sent = ['Take', 'care', 'of', 'the', 'sense', ',', 'and', 'the',
...         'sounds', 'will', 'take', 'care', 'of', 'themselves', '.']
>>> filter(is_content_word, sent)
['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
>>> [w for w in sent if is_content_word(w)]
['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
```

Another higher-order function is `map()`, which applies a function to every item in a sequence. It is a general version of the `extract_property()` function we saw earlier in this section. Here is a simple way to find the average length of a sentence in the news section of the Brown Corpus, followed by an equivalent version with list comprehension calculation:

```
>>> lengths = map(len, nltk.corpus.brown.sents(categories='news'))
>>> sum(lengths) / len(lengths)
21.7508111616
>>> lengths = [len(w) for w in nltk.corpus.brown.sents(categories='news'))]
>>> sum(lengths) / len(lengths)
21.7508111616
```

In the previous examples, we specified a user-defined function `is_content_word()` and a built-in function `len()`. We can also provide a lambda expression. Here's a pair of equivalent examples that count the number of vowels in each word.

```
>>> map(lambda w: len(filter(lambda c: c.lower() in "aeiou", w)), sent)
[2, 2, 1, 1, 2, 0, 1, 1, 2, 1, 2, 2, 1, 3, 0]
>>> [len([c for c in w if c.lower() in "aeiou"]) for w in sent]
[2, 2, 1, 1, 2, 0, 1, 1, 2, 1, 2, 2, 1, 3, 0]
```

The solutions based on list comprehensions are usually more readable than the solutions based on higher-order functions, and we have favored the former approach throughout this book.

Named Arguments

When there are a lot of parameters it is easy to get confused about the correct order. Instead we can refer to parameters by name, and even assign them a default value just in case one was not provided by the calling program. Now the parameters can be specified in any order, and can be omitted.

```
>>> def repeat(msg='<empty>', num=1):
...     return msg * num
>>> repeat(num=3)
'<empty><empty><empty>'
>>> repeat(msg='Alice')
'Alice'
>>> repeat(num=5, msg='Alice')
'AliceAliceAliceAliceAlice'
```

These are called **keyword arguments**. If we mix these two kinds of parameters, then we must ensure that the unnamed parameters precede the named ones. It has to be this

way, since unnamed parameters are defined by position. We can define a function that takes an arbitrary number of unnamed and named parameters, and access them via an in-place list of arguments `*args` and an in-place dictionary of keyword arguments `**kwargs`.

```
>>> def generic(*args, **kwargs):
...     print args
...     print kwargs
...
>>> generic(1, "African swallow", monty="python")
(1, 'African swallow')
{'monty': 'python'}
```

When `*args` appears as a function parameter, it actually corresponds to all the unnamed parameters of the function. As another illustration of this aspect of Python syntax, consider the `zip()` function, which operates on a variable number of arguments. We'll use the variable name `*song` to demonstrate that there's nothing special about the name `*args`.

```
>>> song = [['four', 'calling', 'birds'],
...         ['three', 'French', 'hens'],
...         ['two', 'turtle', 'doves']]
>>> zip(song[0], song[1], song[2])
[('four', 'three', 'two'), ('calling', 'French', 'turtle'), ('birds', 'hens', 'doves')]
>>> zip(*song)
[('four', 'three', 'two'), ('calling', 'French', 'turtle'), ('birds', 'hens', 'doves')]
```

It should be clear from this example that typing `*song` is just a convenient shorthand, and equivalent to typing out `song[0], song[1], song[2]`.

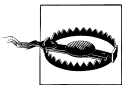
Here's another example of the use of keyword arguments in a function definition, along with three equivalent ways to call the function:

```
>>> def freq_words(file, min=1, num=10):
...     text = open(file).read()
...     tokens = nltk.word_tokenize(text)
...     freqdist = nltk.FreqDist(t for t in tokens if len(t) >= min)
...     return freqdist.keys()[:num]
>>> fw = freq_words('ch01.rst', 4, 10)
>>> fw = freq_words('ch01.rst', min=4, num=10)
>>> fw = freq_words('ch01.rst', num=10, min=4)
```

A side effect of having named arguments is that they permit optionality. Thus we can leave out any arguments where we are happy with the default value: `freq_words('ch01.rst', min=4)`, `freq_words('ch01.rst', 4)`. Another common use of optional arguments is to permit a flag. Here's a revised version of the same function that reports its progress if a `verbose` flag is set:

```
>>> def freq_words(file, min=1, num=10, verbose=False):
...     freqdist = FreqDist()
...     if trace: print "Opening", file
...     text = open(file).read()
...     if trace: print "Read in %d characters" % len(file)
...     for word in nltk.word_tokenize(text):
```

```
...         if len(word) >= min:
...             freqdist.inc(word)
...             if trace and freqdist.N() % 100 == 0: print "."
...     if trace: print
...     return freqdist.keys()[:num]
```



Caution!

Take care not to use a mutable object as the default value of a parameter. A series of calls to the function will use the same object, sometimes with bizarre results, as we will see in the discussion of debugging later.

4.6 Program Development

Programming is a skill that is acquired over several years of experience with a variety of programming languages and tasks. Key high-level abilities are *algorithm design* and its manifestation in *structured programming*. Key low-level abilities include familiarity with the syntactic constructs of the language, and knowledge of a variety of diagnostic methods for trouble-shooting a program which does not exhibit the expected behavior.

This section describes the internal structure of a program module and how to organize a multi-module program. Then it describes various kinds of error that arise during program development, what you can do to fix them and, better still, to avoid them in the first place.

Structure of a Python Module

The purpose of a program module is to bring logically related definitions and functions together in order to facilitate reuse and abstraction. Python modules are nothing more than individual *.py* files. For example, if you were working with a particular corpus format, the functions to read and write the format could be kept together. Constants used by both formats, such as field separators, or a `EXTN = ".inf"` filename extension, could be shared. If the format was updated, you would know that only one file needed to be changed. Similarly, a module could contain code for creating and manipulating a particular data structure such as syntax trees, or code for performing a particular processing task such as plotting corpus statistics.

When you start writing Python modules, it helps to have some examples to emulate. You can locate the code for any NLTK module on your system using the `__file__` variable:

```
>>> nltk.metrics.distance.__file__
'/usr/lib/python2.5/site-packages/nltk/metrics/distance.pyc'
```

This returns the location of the compiled *.pyc* file for the module, and you'll probably see a different location on your machine. The file that you will need to open is the corresponding *.py* source file, and this will be in the same directory as the *.pyc* file.

Alternatively, you can view the latest version of this module on the Web at *http://code.google.com/p/nltk/source/browse/trunk/nltk/nltk/metrics/distance.py*.

Like every other NLTK module, *distance.py* begins with a group of comment lines giving a one-line title of the module and identifying the authors. (Since the code is distributed, it also includes the URL where the code is available, a copyright statement, and license information.) Next is the module-level docstring, a triple-quoted multiline string containing information about the module that will be printed when someone types `help(nltk.metrics.distance)`.

```
# Natural Language Toolkit: Distance Metrics
#
# Copyright (C) 2001-2009 NLTK Project
# Author: Edward Loper <edloper@gradient.cis.upenn.edu>
#         Steven Bird <sb@csse.unimelb.edu.au>
#         Tom Lippincott <tom@cs.columbia.edu>
# URL: <http://www.nltk.org/>
# For license information, see LICENSE.TXT
#

"""
Distance Metrics.

Compute the distance between two items (usually strings).
As metrics, they must satisfy the following three requirements:

1. d(a, a) = 0
2. d(a, b) >= 0
3. d(a, c) <= d(a, b) + d(b, c)
"""
```

After this comes all the import statements required for the module, then any global variables, followed by a series of function definitions that make up most of the module. Other modules define "classes," the main building blocks of object-oriented programming, which falls outside the scope of this book. (Most NLTK modules also include a `demo()` function, which can be used to see examples of the module in use.)

Some module variables and functions are only used within the module. These should have names beginning with an underscore, e.g., `_helper()`, since this will hide the name. If another module imports this one, using the idiom: `from module import *`, these names will not be imported. You can optionally list the externally accessible names of a module using a special built-in variable like this: `__all__ = ['edit_distance', 'jaccard_distance']`.

Multimodule Programs

Some programs bring together a diverse range of tasks, such as loading data from a corpus, performing some analysis tasks on the data, then visualizing it. We may already

have stable modules that take care of loading data and producing visualizations. Our work might involve coding up the analysis task, and just invoking functions from the existing modules. This scenario is depicted in Figure 4-2.

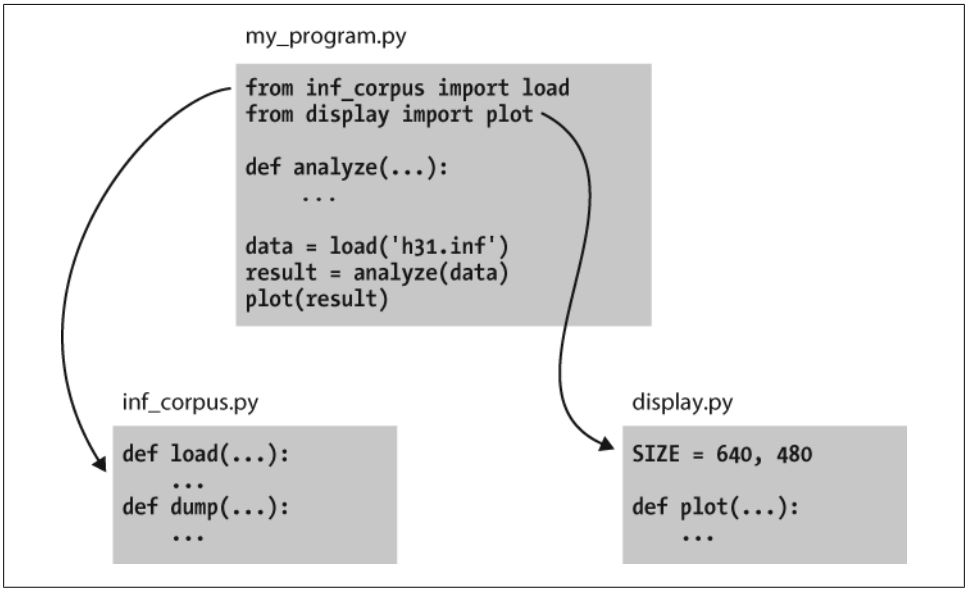


Figure 4-2. Structure of a multimodule program: The main program my_program.py *imports functions from two other modules; unique analysis tasks are localized to the main program, while common loading and visualization tasks are kept apart to facilitate reuse and abstraction.*

By dividing our work into several modules and using `import` statements to access functions defined elsewhere, we can keep the individual modules simple and easy to maintain. This approach will also result in a growing collection of modules, and make it possible for us to build sophisticated systems involving a hierarchy of modules. Designing such systems well is a complex software engineering task, and beyond the scope of this book.

Sources of Error

Mastery of programming depends on having a variety of problem-solving skills to draw upon when the program doesn't work as expected. Something as trivial as a misplaced symbol might cause the program to behave very differently. We call these "bugs" because they are tiny in comparison to the damage they can cause. They creep into our code unnoticed, and it's only much later when we're running the program on some new data that their presence is detected. Sometimes, fixing one bug only reveals another, and we get the distinct impression that the bug is on the move. The only reassurance we have is that bugs are spontaneous and not the fault of the programmer.

Flippancy aside, debugging code is hard because there are so many ways for it to be faulty. Our understanding of the input data, the algorithm, or even the programming language, may be at fault. Let's look at examples of each of these.

First, the input data may contain some unexpected characters. For example, WordNet synset names have the form `tree.n.01`, with three components separated using periods. The NLTK WordNet module initially decomposed these names using `split('.')`. However, this method broke when someone tried to look up the word *PhD*, which has the synset name `ph.d..n.01`, containing four periods instead of the expected two. The solution was to use `rsplit('.', 2)` to do at most two splits, using the rightmost instances of the period, and leaving the `ph.d.` string intact. Although several people had tested the module before it was released, it was some weeks before someone detected the problem (see *http://code.google.com/p/nltk/issues/detail?id=297*).

Second, a supplied function might not behave as expected. For example, while testing NLTK's interface to WordNet, one of the authors noticed that no synsets had any antonyms defined, even though the underlying database provided a large quantity of antonym information. What looked like a bug in the WordNet interface turned out to be a misunderstanding about WordNet itself: antonyms are defined for lemmas, not for synsets. The only "bug" was a misunderstanding of the interface (see *http://code.google.com/p/nltk/issues/detail?id=98*).

Third, our understanding of Python's semantics may be at fault. It is easy to make the wrong assumption about the relative scope of two operators. For example, `"%s.%s.%02d" % "ph.d.", "n", 1` produces a runtime error `TypeError: not enough arguments for format string`. This is because the percent operator has higher precedence than the comma operator. The fix is to add parentheses in order to force the required scope. As another example, suppose we are defining a function to collect all tokens of a text having a given length. The function has parameters for the text and the word length, and an extra parameter that allows the initial value of the result to be given as a parameter:

```
>>> def find_words(text, wordlength, result=[]):
...     for word in text:
...         if len(word) == wordlength:
...             result.append(word)
...     return result
>>> find_words(['omg', 'teh', 'lolcat', 'sitted', 'on', 'teh', 'mat'], 3) ❶
['omg', 'teh', 'teh', 'mat']
>>> find_words(['omg', 'teh', 'lolcat', 'sitted', 'on', 'teh', 'mat'], 2, ['ur']) ❷
['ur', 'on']
>>> find_words(['omg', 'teh', 'lolcat', 'sitted', 'on', 'teh', 'mat'], 3) ❸
['omg', 'teh', 'teh', 'mat', 'omg', 'teh', 'teh', 'mat']
```

The first time we call `find_words()` ❶, we get all three-letter words as expected. The second time we specify an initial value for the result, a one-element list `['ur']`, and as expected, the result has this word along with the other two-letter word in our text. Now, the next time we call `find_words()` ❸ we use the same parameters as in ❶, but we get a different result! Each time we call `find_words()` with no third parameter, the

result will simply extend the result of the previous call, rather than start with the empty result list as specified in the function definition. The program's behavior is not as expected because we incorrectly assumed that the default value was created at the time the function was invoked. However, it is created just once, at the time the Python interpreter loads the function. This one list object is used whenever no explicit value is provided to the function.

Debugging Techniques

Since most code errors result from the programmer making incorrect assumptions, the first thing to do when you detect a bug is to *check your assumptions*. Localize the problem by adding `print` statements to the program, showing the value of important variables, and showing how far the program has progressed.

If the program produced an "exception"—a runtime error—the interpreter will print a **stack trace**, pinpointing the location of program execution at the time of the error. If the program depends on input data, try to reduce this to the smallest size while still producing the error.

Once you have localized the problem to a particular function or to a line of code, you need to work out what is going wrong. It is often helpful to recreate the situation using the interactive command line. Define some variables, and then copy-paste the offending line of code into the session and see what happens. Check your understanding of the code by reading some documentation and examining other code samples that purport to do the same thing that you are trying to do. Try explaining your code to someone else, in case she can see where things are going wrong.

Python provides a **debugger** which allows you to monitor the execution of your program, specify line numbers where execution will stop (i.e., **breakpoints**), and step through sections of code and inspect the value of variables. You can invoke the debugger on your code as follows:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.myfunction()')
```

It will present you with a prompt `(Pdb)` where you can type instructions to the debugger. Type `help` to see the full list of commands. Typing `step` (or just `s`) will execute the current line and stop. If the current line calls a function, it will enter the function and stop at the first line. Typing `next` (or just `n`) is similar, but it stops execution at the next line in the current function. The `break` (or `b`) command can be used to create or list breakpoints. Type `continue` (or `c`) to continue execution as far as the next breakpoint. Type the name of any variable to inspect its value.

We can use the Python debugger to locate the problem in our `find_words()` function. Remember that the problem arose the second time the function was called. We'll start by calling the function without using the debugger ❶, using the smallest possible input. The second time, we'll call it with the debugger ❷.

```
>>> import pdb
>>> find_words(['cat'], 3) ❶
['cat']
>>> pdb.run("find_words(['dog'], 3)") ❷
> <string>(1)<module>()
(Pdb) step
--Call--
> <stdin>(1)find_words()
(Pdb) args
text = ['dog']
wordlength = 3
result = ['cat']
```

Here we typed just two commands into the debugger: `step` took us inside the function, and `args` showed the values of its arguments (or parameters). We see immediately that `result` has an initial value of `['cat']`, and not the empty list as expected. The debugger has helped us to localize the problem, prompting us to check our understanding of Python functions.

Defensive Programming

In order to avoid some of the pain of debugging, it helps to adopt some defensive programming habits. Instead of writing a 20-line program and then testing it, build the program bottom-up out of small pieces that are known to work. Each time you combine these pieces to make a larger unit, test it carefully to see that it works as expected. Consider adding `assert` statements to your code, specifying properties of a variable, e.g., `assert(isinstance(text, list))`. If the value of the `text` variable later becomes a string when your code is used in some larger context, this will raise an `AssertionError` and you will get immediate notification of the problem.

Once you think you've found the bug, view your solution as a hypothesis. Try to predict the effect of your bugfix before re-running the program. If the bug isn't fixed, don't fall into the trap of blindly changing the code in the hope that it will magically start working again. Instead, for each change, try to articulate a hypothesis about what is wrong and why the change will fix the problem. Then undo the change if the problem was not resolved.

As you develop your program, extend its functionality, and fix any bugs, it helps to maintain a suite of test cases. This is called **regression testing**, since it is meant to detect situations where the code "regresses"—where a change to the code has an unintended side effect of breaking something that used to work. Python provides a simple regression-testing framework in the form of the `doctest` module. This module searches a file of code or documentation for blocks of text that look like an interactive Python session, of the form you have already seen many times in this book. It executes the Python commands it finds, and tests that their output matches the output supplied in the original file. Whenever there is a mismatch, it reports the expected and actual values. For details, please consult the `doctest` documentation at

http://docs.python.org/library/doctest.html. Apart from its value for regression testing, the `doctest` module is useful for ensuring that your software documentation stays in sync with your code.

Perhaps the most important defensive programming strategy is to set out your code clearly, choose meaningful variable and function names, and simplify the code wherever possible by decomposing it into functions and modules with well-documented interfaces.

4.7 Algorithm Design

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

A major part of algorithmic problem solving is selecting or adapting an appropriate algorithm for the problem at hand. Sometimes there are several alternatives, and choosing the best one depends on knowledge about how each alternative performs as the size of the data grows. Whole books are written on this topic, and we only have space to introduce some key concepts and elaborate on the approaches that are most prevalent in natural language processing.

The best-known strategy is known as **divide-and-conquer**. We attack a problem of size *n* by dividing it into two problems of size *n*/2, solve these problems, and combine their results into a solution of the original problem. For example, suppose that we had a pile of cards with a single word written on each card. We could sort this pile by splitting it in half and giving it to two other people to sort (they could do the same in turn). Then, when two sorted piles come back, it is an easy task to merge them into a single sorted pile. See Figure 4-3 for an illustration of this process.

Another example is the process of looking up a word in a dictionary. We open the book somewhere around the middle and compare our word with the current page. If it's earlier in the dictionary, we repeat the process on the first half; if it's later, we use the second half. This search method is called *binary search* since it splits the problem in half at every step.

In another approach to algorithm design, we attack a problem by transforming it into an instance of a problem we already know how to solve. For example, in order to detect duplicate entries in a list, we can **pre-sort** the list, then scan through it once to check whether any adjacent pairs of elements are identical.

Recursion

The earlier examples of sorting and searching have a striking property: to solve a problem of size *n*, we have to break it in half and then work on one or more problems of size *n*/2. A common way to implement such methods uses **recursion**. We define a function *f*, which simplifies the problem, and *calls itself* to solve one or more easier

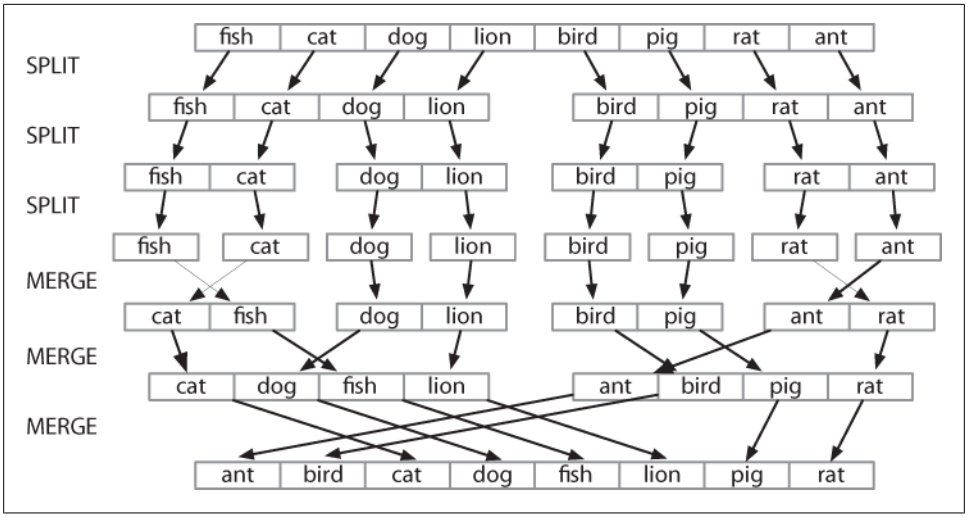


Figure 4-3. Sorting by divide-and-conquer: To sort an array, we split it in half and sort each half (recursively); we merge each sorted half back into a whole list (again recursively); this algorithm is known as "Merge Sort."

instances of the same problem. It then combines the results into a solution for the original problem.

For example, suppose we have a set of *n* words, and want to calculate how many different ways they can be combined to make a sequence of words. If we have only one word (n=1), there is just one way to make it into a sequence. If we have a set of two words, there are two ways to put them into a sequence. For three words there are six possibilities. In general, for *n* words, there are $n \times n\text{-}1 \times \ldots \times 2 \times 1$ ways (i.e., the factorial of *n*). We can code this up as follows:

```
>>> def factorial1(n):
...     result = 1
...     for i in range(n):
...         result *= (i+1)
...     return result
```

However, there is also a recursive algorithm for solving this problem, based on the following observation. Suppose we have a way to construct all orderings for n-1 distinct words. Then for each such ordering, there are *n* places where we can insert a new word: at the start, the end, or any of the n-2 boundaries between the words. Thus we simply multiply the number of solutions found for n-1 by the value of *n*. We also need the **base case**, to say that if we have a single word, there's just one ordering. We can code this up as follows:

```
>>> def factorial2(n):
...     if n == 1:
...         return 1
...     else:
...         return n * factorial2(n-1)
```

These two algorithms solve the same problem. One uses iteration while the other uses recursion. We can use recursion to navigate a deeply nested object, such as the WordNet hypernym hierarchy. Let's count the size of the hypernym hierarchy rooted at a given synset *s*. We'll do this by finding the size of each hyponym of *s*, then adding these together (we will also add 1 for the synset itself). The following function `size1()` does this work; notice that the body of the function includes a recursive call to `size1()`:

```
>>> def size1(s):
...     return 1 + sum(size1(child) for child in s.hyponyms())
```

We can also design an iterative solution to this problem which processes the hierarchy in layers. The first layer is the synset itself ❶, then all the hyponyms of the synset, then all the hyponyms of the hyponyms. Each time through the loop it computes the next layer by finding the hyponyms of everything in the last layer ❸. It also maintains a total of the number of synsets encountered so far ❷.

```
>>> def size2(s):
...     layer = [s] ❶
...     total = 0
...     while layer:
...         total += len(layer) ❷
...         layer = [h for c in layer for h in c.hyponyms()] ❸
...     return total
```

Not only is the iterative solution much longer, it is harder to interpret. It forces us to think procedurally, and keep track of what is happening with the `layer` and `total` variables through time. Let's satisfy ourselves that both solutions give the same result. We'll use a new form of the import statement, allowing us to abbreviate the name `wordnet` to `wn`:

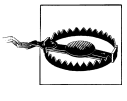
```
>>> from nltk.corpus import wordnet as wn
>>> dog = wn.synset('dog.n.01')
>>> size1(dog)
190
>>> size2(dog)
190
```

As a final example of recursion, let's use it to *construct* a deeply nested object. A **letter trie** is a data structure that can be used for indexing a lexicon, one letter at a time. (The name is based on the word re*trie*val.) For example, if `trie` contained a letter trie, then `trie['c']` would be a smaller trie which held all words starting with *c*. Example 4-6 demonstrates the recursive process of building a trie, using Python dictionaries (Section 5.3). To insert the word *chien* (French for *dog*), we split off the *c* and recursively insert *hien* into the sub-trie `trie['c']`. The recursion continues until there are no letters remaining in the word, when we store the intended value (in this case, the word *dog*).

Example 4-6. Building a letter trie: A recursive function that builds a nested dictionary structure; each level of nesting contains all words with a given prefix, and a sub-trie containing all possible continuations.

```
def insert(trie, key, value):
    if key:
        first, rest = key[0], key[1:]
        if first not in trie:
            trie[first] = {}
        insert(trie[first], rest, value)
    else:
        trie['value'] = value
```

```
>>> trie = nltk.defaultdict(dict)
>>> insert(trie, 'chat', 'cat')
>>> insert(trie, 'chien', 'dog')
>>> insert(trie, 'chair', 'flesh')
>>> insert(trie, 'chic', 'stylish')
>>> trie = dict(trie)              # for nicer printing
>>> trie['c']['h']['a']['t']['value']
'cat'
>>> pprint.pprint(trie)
{'c': {'h': {'a': {'t': {'value': 'cat'}},
                   {'i': {'r': {'value': 'flesh'}}},
             'i': {'e': {'n': {'value': 'dog'}}}
                   {'c': {'value': 'stylish'}}}}}
```



Caution!

Despite the simplicity of recursive programming, it comes with a cost. Each time a function is called, some state information needs to be pushed on a stack, so that once the function has completed, execution can continue from where it left off. For this reason, iterative solutions are often more efficient than recursive solutions.

Space-Time Trade-offs

We can sometimes significantly speed up the execution of a program by building an auxiliary data structure, such as an index. The listing in Example 4-7 implements a simple text retrieval system for the Movie Reviews Corpus. By indexing the document collection, it provides much faster lookup.

Example 4-7. A simple text retrieval system.

```
def raw(file):
    contents = open(file).read()
    contents = re.sub(r'<.*?>', ' ', contents)
    contents = re.sub('\s+', ' ', contents)
    return contents

def snippet(doc, term): # buggy
    text = ' '*30 + raw(doc) + ' '*30
    pos = text.index(term)
    return text[pos-30:pos+30]
```

```
print "Building Index..."
files = nltk.corpus.movie_reviews.abspaths()
idx = nltk.Index((w, f) for f in files for w in raw(f).split())

query = ''
while query != "quit":
    query = raw_input("query> ")
    if query in idx:
        for doc in idx[query]:
            print snippet(doc, query)
    else:
        print "Not found"
```

A more subtle example of a space-time trade-off involves replacing the tokens of a corpus with integer identifiers. We create a vocabulary for the corpus, a list in which each word is stored once, then invert this list so that we can look up any word to find its identifier. Each document is preprocessed, so that a list of words becomes a list of integers. Any language models can now work with integers. See the listing in Example 4-8 for an example of how to do this for a tagged corpus.

Example 4-8. Preprocess tagged corpus data, converting all words and tags to integers.

```
def preprocess(tagged_corpus):
    words = set()
    tags = set()
    for sent in tagged_corpus:
        for word, tag in sent:
            words.add(word)
            tags.add(tag)
    wm = dict((w,i) for (i,w) in enumerate(words))
    tm = dict((t,i) for (i,t) in enumerate(tags))
    return [[(wm[w], tm[t]) for (w,t) in sent] for sent in tagged_corpus]
```

Another example of a space-time trade-off is maintaining a vocabulary list. If you need to process an input text to check that all words are in an existing vocabulary, the vocabulary should be stored as a set, not a list. The elements of a set are automatically indexed, so testing membership of a large set will be much faster than testing membership of the corresponding list.

We can test this claim using the `timeit` module. The `Timer` class has two parameters: a statement that is executed multiple times, and setup code that is executed once at the beginning. We will simulate a vocabulary of 100,000 items using a list ❶ or set ❷ of integers. The test statement will generate a random item that has a 50% chance of being in the vocabulary ❸.

```
>>> from timeit import Timer
>>> vocab_size = 100000
>>> setup_list = "import random; vocab = range(%d)" % vocab_size ❶
>>> setup_set = "import random; vocab = set(range(%d))" % vocab_size ❷
>>> statement = "random.randint(0, %d) in vocab" % vocab_size * 2 ❸
>>> print Timer(statement, setup_list).timeit(1000)
2.78092288971
>>> print Timer(statement, setup_set).timeit(1000)
0.0037260055542
```

Performing 1,000 list membership tests takes a total of 2.8 seconds, whereas the equivalent tests on a set take a mere 0.0037 seconds, or three orders of magnitude faster!

Dynamic Programming

Dynamic programming is a general technique for designing algorithms which is widely used in natural language processing. The term "programming" is used in a different sense to what you might expect, to mean planning or scheduling. Dynamic programming is used when a problem contains overlapping subproblems. Instead of computing solutions to these subproblems repeatedly, we simply store them in a lookup table. In the remainder of this section, we will introduce dynamic programming, but in a rather different context to syntactic parsing.

Pingala was an Indian author who lived around the 5th century B.C., and wrote a treatise on Sanskrit prosody called the *Chandas Shastra*. Virahanka extended this work around the 6th century A.D., studying the number of ways of combining short and long syllables to create a meter of length *n*. Short syllables, marked *S*, take up one unit of length, while long syllables, marked *L*, take two. Pingala found, for example, that there are five ways to construct a meter of length 4: $V_4 = \{LL, SSL, SLS, LSS, SSSS\}$. Observe that we can split V_4 into two subsets, those starting with *L* and those starting with *S*, as shown in (1).

(1) V_4 =
    LL, LSS
      i.e. L prefixed to each item of V_2 = {L, SS}
    SSL, SLS, SSSS
      i.e. S prefixed to each item of V_3 = {SL, LS, SSS}

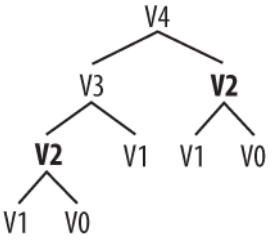
With this observation, we can write a little recursive function called `virahanka1()` to compute these meters, shown in Example 4-9. Notice that, in order to compute V_4 we first compute V_3 and V_2. But to compute V_3, we need to first compute V_2 and V_1. This **call structure** is depicted in (2).

Example 4-9. Four ways to compute Sanskrit meter: (i) iterative, (ii) bottom-up dynamic programming, (iii) top-down dynamic programming, and (iv) built-in memoization.

```
def virahanka1(n):
    if n == 0:
        return [""]
    elif n == 1:
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka1(n-1)]
        l = ["L" + prosody for prosody in virahanka1(n-2)]
        return s + l

def virahanka2(n):
    lookup = [[""], ["S"]]
    for i in range(n-1):
        s = ["S" + prosody for prosody in lookup[i+1]]
        l = ["L" + prosody for prosody in lookup[i]]
        lookup.append(s + l)
    return lookup[n]

def virahanka3(n, lookup={0:[""], 1:["S"]}):
    if n not in lookup:
        s = ["S" + prosody for prosody in virahanka3(n-1)]
        l = ["L" + prosody for prosody in virahanka3(n-2)]
        lookup[n] = s + l
    return lookup[n]

from nltk import memoize
@memoize
def virahanka4(n):
    if n == 0:
        return [""]
    elif n == 1:
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka4(n-1)]
        l = ["L" + prosody for prosody in virahanka4(n-2)]
        return s + l
```

```
>>> virahanka1(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka2(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka3(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
>>> virahanka4(4)
['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
```

(2)



As you can see, V_2 is computed twice. This might not seem like a significant problem, but it turns out to be rather wasteful as n gets large: to compute V_{20} using this recursive technique, we would compute V_2 4,181 times; and for V_{40} we would compute V_2 63,245,986 times! A much better alternative is to store the value of V_2 in a table and look it up whenever we need it. The same goes for other values, such as V_3 and so on. Function `virahanka2()` implements a dynamic programming approach to the problem. It works by filling up a table (called `lookup`) with solutions to *all* smaller instances of the problem, stopping as soon as we reach the value we're interested in. At this point we read off the value and return it. Crucially, each subproblem is only ever solved once.

Notice that the approach taken in `virahanka2()` is to solve smaller problems on the way to solving larger problems. Accordingly, this is known as the **bottom-up** approach to dynamic programming. Unfortunately it turns out to be quite wasteful for some applications, since it may compute solutions to sub-problems that are never required for solving the main problem. This wasted computation can be avoided using the **top-down** approach to dynamic programming, which is illustrated in the function `virahanka3()` in Example 4-9. Unlike the bottom-up approach, this approach is recursive. It avoids the huge wastage of `virahanka1()` by checking whether it has previously stored the result. If not, it computes the result recursively and stores it in the table. The last step is to return the stored result. The final method, in `virahanka4()`, is to use a Python "decorator" called `memoize`, which takes care of the housekeeping work done by `virahanka3()` without cluttering up the program. This "memoization" process stores the result of each previous call to the function along with the parameters that were used. If the function is subsequently called with the same parameters, it returns the stored result instead of recalculating it. (This aspect of Python syntax is beyond the scope of this book.)

This concludes our brief introduction to dynamic programming. We will encounter it again in Section 8.4.

4.8 A Sample of Python Libraries

Python has hundreds of third-party libraries, specialized software packages that extend the functionality of Python. NLTK is one such library. To realize the full power of Python programming, you should become familiar with several other libraries. Most of these will need to be manually installed on your computer.

Matplotlib

Python has some libraries that are useful for visualizing language data. The Matplotlib package supports sophisticated plotting functions with a MATLAB-style interface, and is available from *http://matplotlib.sourceforge.net/*.

So far we have focused on textual presentation and the use of formatted print statements to get output lined up in columns. It is often very useful to display numerical data in graphical form, since this often makes it easier to detect patterns. For example, in Example 3-5, we saw a table of numbers showing the frequency of particular modal verbs in the Brown Corpus, classified by genre. The program in Example 4-10 presents the same information in graphical format. The output is shown in Figure 4-4 (a color figure in the graphical display).

Example 4-10. Frequency of modals in different sections of the Brown Corpus.

```
colors = 'rgbcmyk' # red, green, blue, cyan, magenta, yellow, black
def bar_chart(categories, words, counts):
    "Plot a bar chart showing counts for each word by category"
    import pylab
    ind = pylab.arange(len(words))
    width = 1 / (len(categories) + 1)
    bar_groups = []
    for c in range(len(categories)):
        bars = pylab.bar(ind+c*width, counts[categories[c]], width,
                         color=colors[c % len(colors)])
        bar_groups.append(bars)
    pylab.xticks(ind+width, words)
    pylab.legend([b[0] for b in bar_groups], categories, loc='upper left')
    pylab.ylabel('Frequency')
    pylab.title('Frequency of Six Modal Verbs by Genre')
    pylab.show()
```

```
>>> genres = ['news', 'religion', 'hobbies', 'government', 'adventure']
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> cfdist = nltk.ConditionalFreqDist(
...              (genre, word)
...              for genre in genres
...              for word in nltk.corpus.brown.words(categories=genre)
...              if word in modals)
...
>>> counts = {}
>>> for genre in genres:
...     counts[genre] = [cfdist[genre][word] for word in modals]
>>> bar_chart(genres, modals, counts)
```

From the bar chart it is immediately obvious that *may* and *must* have almost identical relative frequencies. The same goes for *could* and *might*.

It is also possible to generate such data visualizations on the fly. For example, a web page with form input could permit visitors to specify search parameters, submit the form, and see a dynamically generated visualization. To do this we have to specify the

Agg backend for matplotlib, which is a library for producing raster (pixel) images ❶. Next, we use all the same PyLab methods as before, but instead of displaying the result on a graphical terminal using pylab.show(), we save it to a file using pylab.savefig() ❷. We specify the filename and dpi, then print HTML markup that directs the web browser to load the file.

```
>>> import matplotlib
>>> matplotlib.use('Agg') ❶
>>> pylab.savefig('modals.png') ❷
>>> print 'Content-Type: text/html'
>>> print
>>> print '<html><body>'
>>> print '<img src="modals.png"/>'
>>> print '</body></html>'
```

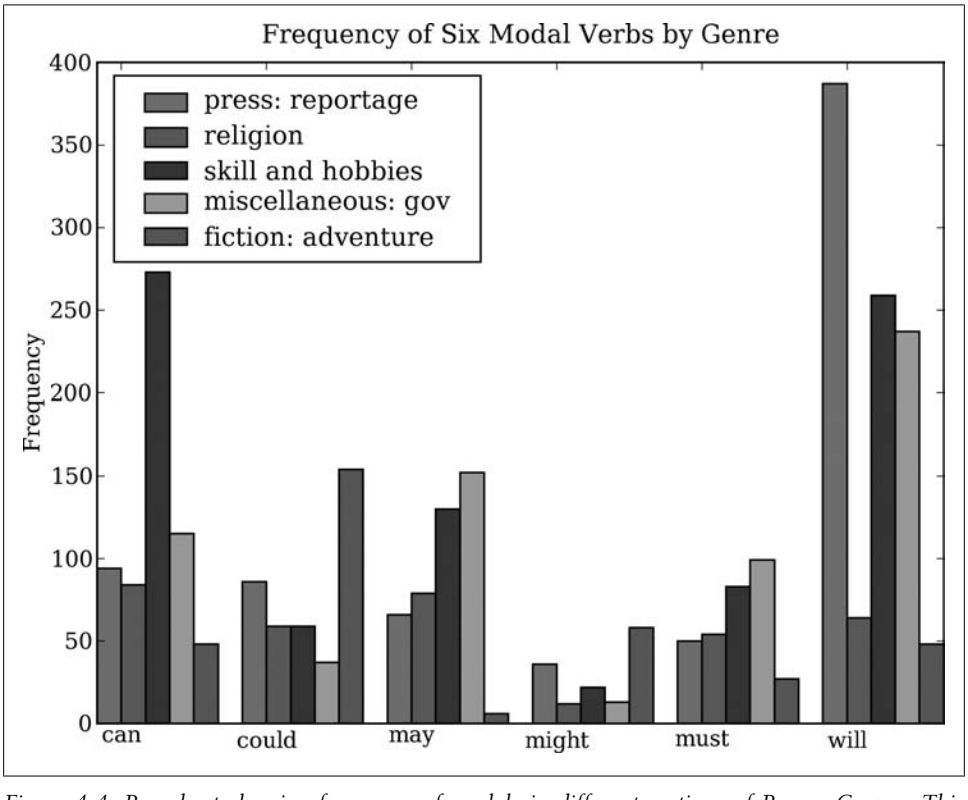


Figure 4-4. Bar chart showing frequency of modals in different sections of Brown Corpus: This visualization was produced by the program in Example 4-10.

NetworkX

The NetworkX package is for defining and manipulating structures consisting of nodes and edges, known as **graphs**. It is available from *https://networkx.lanl.gov/*. NetworkX

can be used in conjunction with Matplotlib to visualize networks, such as WordNet (the semantic network we introduced in Section 2.5). The program in Example 4-11 initializes an empty graph ❸ and then traverses the WordNet hypernym hierarchy adding edges to the graph ❶. Notice that the traversal is recursive ❷, applying the programming technique discussed in Section 4.7. The resulting display is shown in Figure 4-5.

Example 4-11. Using the NetworkX and Matplotlib libraries.

```
import networkx as nx
import matplotlib
from nltk.corpus import wordnet as wn

def traverse(graph, start, node):
    graph.depth[node.name] = node.shortest_path_distance(start)
    for child in node.hyponyms():
        graph.add_edge(node.name, child.name) ❶
        traverse(graph, start, child) ❷

def hyponym_graph(start):
    G = nx.Graph() ❸
    G.depth = {}
    traverse(G, start, start)
    return G

def graph_draw(graph):
    nx.draw_graphviz(graph,
         node_size = [16 * graph.degree(n) for n in graph],
         node_color = [graph.depth[n] for n in graph],
         with_labels = False)
    matplotlib.pyplot.show()
```

```
>>> dog = wn.synset('dog.n.01')
>>> graph = hyponym_graph(dog)
>>> graph_draw(graph)
```

CSV

Language analysis work often involves data tabulations, containing information about lexical items, the participants in an empirical study, or the linguistic features extracted from a corpus. Here's a fragment of a simple lexicon, in CSV format:

```
sleep, sli:p, v.i, a condition of body and mind ...
walk, wo:k, v.intr, progress by lifting and setting down each foot ...
wake, weik, intrans, cease to sleep
```

We can use Python's CSV library to read and write files stored in this format. For example, we can open a CSV file called *lexicon.csv* ❶ and iterate over its rows ❷:

```
>>> import csv
>>> input_file = open("lexicon.csv", "rb") ❶
>>> for row in csv.reader(input_file): ❷
...     print row
['sleep', 'sli:p', 'v.i', 'a condition of body and mind ...']
```

```
['walk', 'wo:k', 'v.intr', 'progress by lifting and setting down each foot ...']
['wake', 'weik', 'intrans', 'cease to sleep']
```

Each row is just a list of strings. If any fields contain numerical data, they will appear as strings, and will have to be converted using `int()` or `float()`.

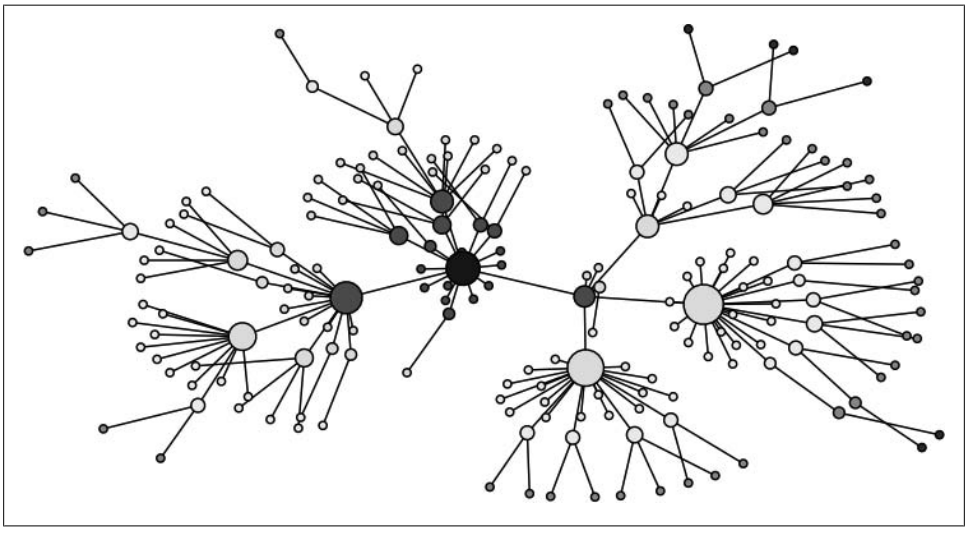


Figure 4-5. Visualization with NetworkX and Matplotlib: Part of the WordNet hypernym hierarchy is displayed, starting with dog.n.01 (the darkest node in the middle); node size is based on the number of children of the node, and color is based on the distance of the node from dog.n.01; this visualization was produced by the program in Example 4-11.

NumPy

The NumPy package provides substantial support for numerical processing in Python. NumPy has a multidimensional array object, which is easy to initialize and access:

```
>>> from numpy import array
>>> cube = array([ [[0,0,0], [1,1,1], [2,2,2]],
...                [[3,3,3], [4,4,4], [5,5,5]],
...                [[6,6,6], [7,7,7], [8,8,8]] ])
>>> cube[1,1,1]
4
>>> cube[2].transpose()
array([[6, 7, 8],
       [6, 7, 8],
       [6, 7, 8]])
>>> cube[2,1:]
array([[7, 7, 7],
       [8, 8, 8]])
```

NumPy includes linear algebra functions. Here we perform singular value decomposition on a matrix, an operation used in **latent semantic analysis** to help identify implicit concepts in a document collection:

```
>>> from numpy import linalg
>>> a=array([[4,0], [3,-5]])
>>> u,s,vt = linalg.svd(a)
>>> u
array([[-0.4472136 , -0.89442719],
       [-0.89442719,  0.4472136 ]])
>>> s
array([ 6.32455532,  3.16227766])
>>> vt
array([[-0.70710678,  0.70710678],
       [-0.70710678, -0.70710678]])
```

NLTK's clustering package `nltk.cluster` makes extensive use of NumPy arrays, and includes support for *k*-means clustering, Gaussian EM clustering, group average agglomerative clustering, and dendogram plots. For details, type `help(nltk.cluster)`.

Other Python Libraries

There are many other Python libraries, and you can search for them with the help of the Python Package Index at *http://pypi.python.org/*. Many libraries provide an interface to external software, such as relational databases (e.g., `mysql-python`) and large document collections (e.g., `PyLucene`). Many other libraries give access to file formats such as PDF, MSWord, and XML (`pypdf`, `pywin32`, `xml.etree`), RSS feeds (e.g., `feedparser`), and electronic mail (e.g., `imaplib`, `email`).

4.9 Summary

- Python's assignment and parameter passing use object references; e.g., if `a` is a list and we assign `b = a`, then any operation on `a` will modify `b`, and vice versa.
- The `is` operation tests whether two objects are identical internal objects, whereas `==` tests whether two objects are equivalent. This distinction parallels the type-token distinction.
- Strings, lists, and tuples are different kinds of sequence object, supporting common operations such as indexing, slicing, `len()`, `sorted()`, and membership testing using `in`.
- We can write text to a file by opening the file for writing

  ```
  ofile = open('output.txt', 'w'
  ```

  then adding content to the file `ofile.write("Monty Python")`, and finally closing the file `ofile.close()`.
- A declarative programming style usually produces more compact, readable code; manually incremented loop variables are usually unnecessary. When a sequence must be enumerated, use `enumerate()`.
- Functions are an essential programming abstraction: key concepts to understand are parameter passing, variable scope, and docstrings.

- A function serves as a namespace: names defined inside a function are not visible outside that function, unless those names are declared to be global.
- Modules permit logically related material to be localized in a file. A module serves as a namespace: names defined in a module—such as variables and functions—are not visible to other modules, unless those names are imported.
- Dynamic programming is an algorithm design technique used widely in NLP that stores the results of previous computations in order to avoid unnecessary recomputation.

4.10 Further Reading

This chapter has touched on many topics in programming, some specific to Python, and some quite general. We've just scratched the surface, and you may want to read more about these topics, starting with the further materials for this chapter available at *http://www.nltk.org/*.

The Python website provides extensive documentation. It is important to understand the built-in functions and standard types, described at *http://docs.python.org/library/functions.html* and *http://docs.python.org/library/stdtypes.html*. We have learned about generators and their importance for efficiency; for information about iterators, a closely related topic, see *http://docs.python.org/library/itertools.html*. Consult your favorite Python book for more information on such topics. An excellent resource for using Python for multimedia processing, including working with sound files, is (Guzdial, 2005).

When using the online Python documentation, be aware that your installed version might be different from the version of the documentation you are reading. You can easily check what version you have, with `import sys; sys.version`. Version-specific documentation is available at *http://www.python.org/doc/versions/*.

Algorithm design is a rich field within computer science. Some good starting points are (Harel, 2004), (Levitin, 2004), and (Knuth, 2006). Useful guidance on the practice of software development is provided in (Hunt & Thomas, 2000) and (McConnell, 2004).

4.11 Exercises

1. ○ Find out more about sequence objects using Python's help facility. In the interpreter, type `help(str)`, `help(list)`, and `help(tuple)`. This will give you a full list of the functions supported by each type. Some functions have special names flanked with underscores; as the help documentation shows, each such function corresponds to something more familiar. For example `x.__getitem__(y)` is just a long-winded way of saying `x[y]`.
2. ○ Identify three operations that can be performed on both tuples and lists. Identify three list operations that cannot be performed on tuples. Name a context where using a list instead of a tuple generates a Python error.

3. ○ Find out how to create a tuple consisting of a single item. There are at least two ways to do this.
4. ○ Create a list `words = ['is', 'NLP', 'fun', '?']`. Use a series of assignment statements (e.g., `words[1] = words[2]`) and a temporary variable `tmp` to transform this list into the list `['NLP', 'is', 'fun', '!']`. Now do the same transformation using tuple assignment.
5. ○ Read about the built-in comparison function `cmp`, by typing `help(cmp)`. How does it differ in behavior from the comparison operators?
6. ○ Does the method for creating a sliding window of n-grams behave correctly for the two limiting cases: $n = 1$ and $n =$ `len(sent)`?
7. ○ We pointed out that when empty strings and empty lists occur in the condition part of an `if` clause, they evaluate to `False`. In this case, they are said to be occurring in a Boolean context. Experiment with different kinds of non-Boolean expressions in Boolean contexts, and see whether they evaluate as `True` or `False`.
8. ○ Use the inequality operators to compare strings, e.g., `'Monty' < 'Python'`. What happens when you do `'Z' < 'a'`? Try pairs of strings that have a common prefix, e.g., `'Monty' < 'Montague'`. Read up on "lexicographical sort" in order to understand what is going on here. Try comparing structured objects, e.g., `('Monty', 1) < ('Monty', 2)`. Does this behave as expected?
9. ○ Write code that removes whitespace at the beginning and end of a string, and normalizes whitespace between words to be a single-space character.
   a. Do this task using `split()` and `join()`.
   b. Do this task using regular expression substitutions.
10. ○ Write a program to sort words by length. Define a helper function `cmp_len` which uses the `cmp` comparison function on word lengths.
11. ◑ Create a list of words and store it in a variable `sent1`. Now assign `sent2 = sent1`. Modify one of the items in `sent1` and verify that `sent2` has changed.
    a. Now try the same exercise, but instead assign `sent2 = sent1[:]`. Modify `sent1` again and see what happens to `sent2`. Explain.
    b. Now define `text1` to be a list of lists of strings (e.g., to represent a text consisting of multiple sentences). Now assign `text2 = text1[:]`, assign a new value to one of the words, e.g., `text1[1][1] = 'Monty'`. Check what this did to `text2`. Explain.
    c. Load Python's `deepcopy()` function (i.e., `from copy import deepcopy`), consult its documentation, and test that it makes a fresh copy of any object.
12. ◑ Initialize an n-by-m list of lists of empty strings using list multiplication, e.g., `word_table = [[''] * n] * m`. What happens when you set one of its values, e.g., `word_table[1][2] = "hello"`? Explain why this happens. Now write an expression using `range()` to construct a list of lists, and show that it does not have this problem.

13. ◑ Write code to initialize a two-dimensional array of sets called `word_vowels` and process a list of words, adding each word to `word_vowels[l][v]` where `l` is the length of the word and `v` is the number of vowels it contains.
14. ◑ Write a function `novel10(text)` that prints any word that appeared in the last 10% of a text that had not been encountered earlier.
15. ◑ Write a program that takes a sentence expressed as a single string, splits it, and counts up the words. Get it to print out each word and the word's frequency, one per line, in alphabetical order.
16. ◑ Read up on Gematria, a method for assigning numbers to words, and for mapping between words having the same number to discover the hidden meaning of texts (*http://en.wikipedia.org/wiki/Gematria*, *http://essenes.net/gemcal.htm*).
    a. Write a function `gematria()` that sums the numerical values of the letters of a word, according to the letter values in `letter_vals`:

    ```
    >>> letter_vals = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':80, 'g':3, 'h':8,
    ... 'i':10, 'j':10, 'k':20, 'l':30, 'm':40, 'n':50, 'o':70, 'p':80, 'q':100,
    ... 'r':200, 's':300, 't':400, 'u':6, 'v':6, 'w':800, 'x':60, 'y':10, 'z':7}
    ```

    b. Process a corpus (e.g., `nltk.corpus.state_union`) and for each document, count how many of its words have the number 666.
    c. Write a function `decode()` to process a text, randomly replacing words with their Gematria equivalents, in order to discover the "hidden meaning" of the text.
17. ◑ Write a function `shorten(text, n)` to process a text, omitting the *n* most frequently occurring words of the text. How readable is it?
18. ◑ Write code to print out an index for a lexicon, allowing someone to look up words according to their meanings (or their pronunciations; whatever properties are contained in the lexical entries).
19. ◑ Write a list comprehension that sorts a list of WordNet synsets for proximity to a given synset. For example, given the synsets `minke_whale.n.01`, `orca.n.01`, `novel.n.01`, and `tortoise.n.01`, sort them according to their `path_distance()` from `right_whale.n.01`.
20. ◑ Write a function that takes a list of words (containing duplicates) and returns a list of words (with no duplicates) sorted by decreasing frequency. E.g., if the input list contained 10 instances of the word `table` and 9 instances of the word `chair`, then `table` would appear before `chair` in the output list.
21. ◑ Write a function that takes a text and a vocabulary as its arguments and returns the set of words that appear in the text but not in the vocabulary. Both arguments can be represented as lists of strings. Can you do this in a single line, using `set.difference()`?
22. ◑ Import the `itemgetter()` function from the `operator` module in Python's standard library (i.e., `from operator import itemgetter`). Create a list `words` containing sev-

eral words. Now try calling: `sorted(words, key=itemgetter(1))`, and `sorted(words, key=itemgetter(-1))`. Explain what `itemgetter()` is doing.

23. ◑ Write a recursive function `lookup(trie, key)` that looks up a key in a trie, and returns the value it finds. Extend the function to return a word when it is uniquely determined by its prefix (e.g., `vanguard` is the only word that starts with `vang-`, so `lookup(trie, 'vang')` should return the same thing as `lookup(trie, 'vanguard')`).
24. ◑ Read up on "keyword linkage" (Chapter 5 of (Scott & Tribble, 2006)). Extract keywords from NLTK's Shakespeare Corpus and using the NetworkX package, plot keyword linkage networks.
25. ◑ Read about string edit distance and the Levenshtein Algorithm. Try the implementation provided in `nltk.edit_dist()`. In what way is this using dynamic programming? Does it use the bottom-up or top-down approach? (See also *http://norvig.com/spell-correct.html*.)
26. ◑ The Catalan numbers arise in many applications of combinatorial mathematics, including the counting of parse trees (Section 8.6). The series can be defined as follows: $C_0 = 1$, and $C_{n+1} = \Sigma_{0..n} (C_i C_{n-i})$.
    a. Write a recursive function to compute nth Catalan number C_n.
    b. Now write another function that does this computation using dynamic programming.
    c. Use the `timeit` module to compare the performance of these functions as n increases.
27. ● Reproduce some of the results of (Zhao & Zobel, 2007) concerning authorship identification.
28. ● Study gender-specific lexical choice, and see if you can reproduce some of the results of *http://www.clintoneast.com/articles/words.php*.
29. ● Write a recursive function that pretty prints a trie in alphabetically sorted order, for example:

```
chair: 'flesh'
---t: 'cat'
--ic: 'stylish'
---en: 'dog'
```

30. ● With the help of the trie data structure, write a recursive function that processes text, locating the uniqueness point in each word, and discarding the remainder of each word. How much compression does this give? How readable is the resulting text?
31. ● Obtain some raw text, in the form of a single, long string. Use Python's `textwrap` module to break it up into multiple lines. Now write code to add extra spaces between words, in order to justify the output. Each line must have the same width, and spaces must be approximately evenly distributed across each line. No line can begin or end with a space.

32. ● Develop a simple extractive summarization tool, that prints the sentences of a document which contain the highest total word frequency. Use `FreqDist()` to count word frequencies, and use `sum` to sum the frequencies of the words in each sentence. Rank the sentences according to their score. Finally, print the *n* highest-scoring sentences in document order. Carefully review the design of your program, especially your approach to this double sorting. Make sure the program is written as clearly as possible.
33. ● Develop your own `NgramTagger` class that inherits from NLTK's class, and which encapsulates the method of collapsing the vocabulary of the tagged training and testing data that was described in Chapter 5. Make sure that the unigram and default backoff taggers have access to the full vocabulary.
34. ● Read the following article on semantic orientation of adjectives. Use the NetworkX package to visualize a network of adjectives with edges to indicate same versus different semantic orientation (see *http://www.aclweb.org/anthology/P97-1023*).
35. ● Design an algorithm to find the "statistically improbable phrases" of a document collection (see *http://www.amazon.com/gp/search-inside/sipshelp.html*).
36. ● Write a program to implement a brute-force algorithm for discovering word squares, a kind of $n \times n$: crossword in which the entry in the nth row is the same as the entry in the nth column. For discussion, see *http://itre.cis.upenn.edu/~myl/languagelog/archives/002679.html*.

CHAPTER 5

Categorizing and Tagging Words

Back in elementary school you learned the difference between nouns, verbs, adjectives, and adverbs. These "word classes" are not just the idle invention of grammarians, but are useful categories for many language processing tasks. As we will see, they arise from simple analysis of the distribution of words in text. The goal of this chapter is to answer the following questions:

1. What are lexical categories, and how are they used in natural language processing?
2. What is a good Python data structure for storing words and their categories?
3. How can we automatically tag each word of a text with its word class?

Along the way, we'll cover some fundamental techniques in NLP, including sequence labeling, n-gram models, backoff, and evaluation. These techniques are useful in many areas, and tagging gives us a simple context in which to present them. We will also see how tagging is the second step in the typical NLP pipeline, following tokenization.

The process of classifying words into their **parts-of-speech** and labeling them accordingly is known as **part-of-speech tagging**, **POS tagging**, or simply **tagging**. Parts-of-speech are also known as **word classes** or **lexical categories**. The collection of tags used for a particular task is known as a **tagset**. Our emphasis in this chapter is on exploiting tags, and tagging text automatically.

5.1 Using a Tagger

A part-of-speech tagger, or **POS tagger**, processes a sequence of words, and attaches a part of speech tag to each word (don't forget to `import nltk`):

```
>>> text = nltk.word_tokenize("And now for something completely different")
>>> nltk.pos_tag(text)
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'),
('completely', 'RB'), ('different', 'JJ')]
```

Here we see that *and* is `CC`, a coordinating conjunction; *now* and *completely* are `RB`, or adverbs; *for* is `IN`, a preposition; *something* is `NN`, a noun; and *different* is `JJ`, an adjective.

NLTK provides documentation for each tag, which can be queried using the tag, e.g., `nltk.help.upenn_tagset('RB')`, or a regular expression, e.g., `nltk.help.upenn_brown_tagset('NN.*')`. Some corpora have README files with tagset documentation; see `nltk.name.readme()`, substituting in the name of the corpus.

Let's look at another example, this time including some homonyms:

```
>>> text = nltk.word_tokenize("They refuse to permit us to obtain the refuse permit")
>>> nltk.pos_tag(text)
[('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'),
('to', 'TO'), ('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'), ('permit', 'NN')]
```

Notice that *refuse* and *permit* both appear as a present tense verb (`VBP`) and a noun (`NN`). E.g., *refUSE* is a verb meaning "deny," while *REFuse* is a noun meaning "trash" (i.e., they are not homophones). Thus, we need to know which word is being used in order to pronounce the text correctly. (For this reason, text-to-speech systems usually perform POS tagging.)

Your Turn: Many words, like *ski* and *race*, can be used as nouns or verbs with no difference in pronunciation. Can you think of others? Hint: think of a commonplace object and try to put the word *to* before it to see if it can also be a verb, or think of an action and try to put *the* before it to see if it can also be a noun. Now make up a sentence with both uses of this word, and run the POS tagger on this sentence.

Lexical categories like "noun" and part-of-speech tags like `NN` seem to have their uses, but the details will be obscure to many readers. You might wonder what justification there is for introducing this extra level of information. Many of these categories arise from superficial analysis of the distribution of words in text. Consider the following analysis involving *woman* (a noun), *bought* (a verb), *over* (a preposition), and *the* (a determiner). The `text.similar()` method takes a word w, finds all contexts $w_1w\ w_2$, then finds all words w' that appear in the same context, i.e. $w_1w'w_2$.

```
>>> text = nltk.Text(word.lower() for word in nltk.corpus.brown.words())
>>> text.similar('woman')
Building word-context index...
man time day year car moment world family house country child boy
state job way war girl place room word
>>> text.similar('bought')
made said put done seen had found left given heard brought got been
was set told took in felt that
>>> text.similar('over')
in on to of and for with from at by that into as up out down through
is all about
>>> text.similar('the')
a his this their its her an that our any all one these my in your no
some other and
```

Observe that searching for *woman* finds nouns; searching for *bought* mostly finds verbs; searching for *over* generally finds prepositions; searching for *the* finds several determiners. A tagger can correctly identify the tags on these words in the context of a sentence, e.g., *The woman bought over $150,000 worth of clothes*.

A tagger can also model our knowledge of unknown words; for example, we can guess that *scrobbling* is probably a verb, with the root *scrobble*, and likely to occur in contexts like *he was scrobbling*.

5.2 Tagged Corpora

Representing Tagged Tokens

By convention in NLTK, a tagged token is represented using a tuple consisting of the token and the tag. We can create one of these special tuples from the standard string representation of a tagged token, using the function `str2tuple()`:

```
>>> tagged_token = nltk.tag.str2tuple('fly/NN')
>>> tagged_token
('fly', 'NN')
>>> tagged_token[0]
'fly'
>>> tagged_token[1]
'NN'
```

We can construct a list of tagged tokens directly from a string. The first step is to tokenize the string to access the individual `word/tag` strings, and then to convert each of these into a tuple (using `str2tuple()`).

```
>>> sent = '''
... The/AT grand/JJ jury/NN commented/VBD on/IN a/AT number/NN of/IN
... other/AP topics/NNS ,/, AMONG/IN them/PPO the/AT Atlanta/NP and/CC
... Fulton/NP-tl County/NN-tl purchasing/VBG departments/NNS which/WDT it/PPS
... said/VBD ``/`` ARE/BER well/QL operated/VBN and/CC follow/VB generally/RB
... accepted/VBN practices/NNS which/WDT inure/VB to/IN the/AT best/JJT
... interest/NN of/IN both/ABX governments/NNS ''/'' ./.
... '''
>>> [nltk.tag.str2tuple(t) for t in sent.split()]
[('The', 'AT'), ('grand', 'JJ'), ('jury', 'NN'), ('commented', 'VBD'),
('on', 'IN'), ('a', 'AT'), ('number', 'NN'), ... ('.', '.')]
```

Reading Tagged Corpora

Several of the corpora included with NLTK have been **tagged** for their part-of-speech. Here's an example of what you might see if you opened a file from the Brown Corpus with a text editor:

> The/at Fulton/np-tl County/nn-tl Grand/jj-tl Jury/nn-tl said/vbd Friday/nr an/at investigation/nn of/in Atlanta's/np$ recent/jj primary/nn election/nn produced/vbd / no/at evidence/nn "/" that/cs any/dti irregularities/nns took/vbd place/nn ./.

Other corpora use a variety of formats for storing part-of-speech tags. NLTK's corpus readers provide a uniform interface so that you don't have to be concerned with the different file formats. In contrast with the file extract just shown, the corpus reader for the Brown Corpus represents the data as shown next. Note that part-of-speech tags have been converted to uppercase; this has become standard practice since the Brown Corpus was published.

```
>>> nltk.corpus.brown.tagged_words()
[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ...]
>>> nltk.corpus.brown.tagged_words(simplify_tags=True)
[('The', 'DET'), ('Fulton', 'N'), ('County', 'N'), ...]
```

Whenever a corpus contains tagged text, the NLTK corpus interface will have a `tagged_words()` method. Here are some more examples, again using the output format illustrated for the Brown Corpus:

```
>>> print nltk.corpus.nps_chat.tagged_words()
[('now', 'RB'), ('im', 'PRP'), ('left', 'VBD'), ...]
>>> nltk.corpus.conll2000.tagged_words()
[('Confidence', 'NN'), ('in', 'IN'), ('the', 'DT'), ...]
>>> nltk.corpus.treebank.tagged_words()
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ...]
```

Not all corpora employ the same set of tags; see the tagset help functionality and the `readme()` methods mentioned earlier for documentation. Initially we want to avoid the complications of these tagsets, so we use a built-in mapping to a simplified tagset:

```
>>> nltk.corpus.brown.tagged_words(simplify_tags=True)
[('The', 'DET'), ('Fulton', 'NP'), ('County', 'N'), ...]
>>> nltk.corpus.treebank.tagged_words(simplify_tags=True)
[('Pierre', 'NP'), ('Vinken', 'NP'), (',', ','), ...]
```

Tagged corpora for several other languages are distributed with NLTK, including Chinese, Hindi, Portuguese, Spanish, Dutch, and Catalan. These usually contain non-ASCII text, and Python always displays this in hexadecimal when printing a larger structure such as a list.

```
>>> nltk.corpus.sinica_treebank.tagged_words()
[('\xe4\xb8\x80', 'Neu'), ('\xe5\x8f\x8b\xe6\x83\x85', 'Nad'), ...]
>>> nltk.corpus.indian.tagged_words()
[('\xe0\xa6\xae\xe0\xa6\xb9\xe0\xa6\xbf\xe0\xa6\xb7\xe0\xa7\x87\xe0\xa6\xb0', 'NN'),
('\xe0\xa6\xb8\xe0\xa6\xa8\xe0\xa7\x8d\xe0\xa6\xa4\xe0\xa6\xbe\xe0\xa6\xa8', 'NN'),
...]
>>> nltk.corpus.mac_morpho.tagged_words()
[('Jersei', 'N'), ('atinge', 'V'), ('m\xe9dia', 'N'), ...]
>>> nltk.corpus.conll2002.tagged_words()
[('Sao', 'NC'), ('Paulo', 'VMI'), ('(', 'Fpa'), ...]
>>> nltk.corpus.cess_cat.tagged_words()
[('El', 'da0ms0'), ('Tribunal_Suprem', 'np0000o'), ...]
```

If your environment is set up correctly, with appropriate editors and fonts, you should be able to display individual strings in a human-readable way. For example, Figure 5-1 shows data accessed using `nltk.corpus.indian`.

If the corpus is also segmented into sentences, it will have a `tagged_sents()` method that divides up the tagged words into sentences rather than presenting them as one big list. This will be useful when we come to developing automatic taggers, as they are trained and tested on lists of sentences, not words.

A Simplified Part-of-Speech Tagset

Tagged corpora use many different conventions for tagging words. To help us get started, we will be looking at a simplified tagset (shown in Table 5-1).

Table 5-1. Simplified part-of-speech tagset

Tag	Meaning	Examples
ADJ	adjective	*new, good, high, special, big, local*
ADV	adverb	*really, already, still, early, now*
CNJ	conjunction	*and, or, but, if, while, although*
DET	determiner	*the, a, some, most, every, no*
EX	existential	*there, there's*
FW	foreign word	*dolce, ersatz, esprit, quo, maitre*
MOD	modal verb	*will, can, would, may, must, should*
N	noun	*year, home, costs, time, education*
NP	proper noun	*Alison, Africa, April, Washington*
NUM	number	*twenty-four, fourth, 1991, 14:24*
PRO	pronoun	*he, their, her, its, my, I, us*
P	preposition	*on, of, at, with, by, into, under*
TO	the word *to*	*to*
UH	interjection	*ah, bang, ha, whee, hmpf, oops*
V	verb	*is, has, get, do, make, see, run*
VD	past tense	*said, took, told, made, asked*
VG	present participle	*making, going, playing, working*
VN	past participle	*given, taken, begun, sung*
WH	*wh* determiner	*who, which, when, what, where, how*

```
Bangla: ক ুঁড়েঘরগ ুল ি/'NN' আক ার/'NN' ব াংল ার/'NNP' ব া/'CC' ভ ারতব ে/'NNP' ?/None
ন য/'JJ' ?/None এ চলব ে/'NN' প রচল তি/'JJ' ক ুঁড়ে/'NN' ঘর/'NN' নয়/'VM' ি/'SYM'
Hindi: प ाकिस्त ान/'NNP' की/'PREP' प ूर्व/'JJ' प्रध ानम ंत्री/'NN' बेनजीर/'NNPC' भ ुट्टो/'NNP'
पर/'PREP' लगे/'VFM' भ्रष्ट ाच ार/'NN' के/'PREP' आरोपों/'NN' के/'PREP' ख िल ाफ/'PREP' भ ुट्टो/'NNP'
द्व ारा/'PREP' द ायर/'NVB' की/'VFM' गई/'VAUX' य ाचिक ा/'NN' की/'PREP' स ुनव ाई/'NN'
म ंगलव ार/'NN' को/'PREP' वकीलों/'NN' की/'PREP' हड़त ाल/'NN' के/'PREP' क ारण/'PREP'
स्थ गित/'JVB' कर/'VFM' दी/'VAUX' गई/'VAUX' ।/'PUNC'
Marathi: ग ामीण/'JJ' जिल्ह ाध्यक्ष/'NN' ब ाळ स ाहेब/'NNPC' भ ोसले/'NNP' य ांच्य ा/'PRP' ?/None
ध्यक्षतेख ाली/'NN' पक्ष ाची/'NN' आज/'NN' ब ैठ?/None क/'NN' झ ाली/'VM' ./'SYM'
Telugu: ఖ ండ ాలు/'NN' మ ంచి/'PREP' మ ద్దత ు/'VJJ' ప ల్లెల/'NN' మ /'PREP' స ొమ్మ ు/'NN'
```

Figure 5-1. POS tagged data from four Indian languages: Bangla, Hindi, Marathi, and Telugu.

Let's see which of these tags are the most common in the news category of the Brown Corpus:

```
>>> from nltk.corpus import brown
>>> brown_news_tagged = brown.tagged_words(categories='news', simplify_tags=True)
>>> tag_fd = nltk.FreqDist(tag for (word, tag) in brown_news_tagged)
>>> tag_fd.keys()
['N', 'P', 'DET', 'NP', 'V', 'ADJ', ',', '.', 'CNJ', 'PRO', 'ADV', 'VD', ...]
```

Your Turn: Plot the frequency distribution just shown using `tag_fd.plot(cumulative=True)`. What percentage of words are tagged using the first five tags of the above list?

We can use these tags to do powerful searches using a graphical POS-concordance tool `nltk.app.concordance()`. Use it to search for any combination of words and POS tags, e.g., `N N N N`, `hit/VD`, `hit/VN`, or `the ADJ man`.

Nouns

Nouns generally refer to people, places, things, or concepts, e.g., *woman, Scotland, book, intelligence*. Nouns can appear after determiners and adjectives, and can be the subject or object of the verb, as shown in Table 5-2.

Table 5-2. Syntactic patterns involving some nouns

Word	After a determiner	Subject of the verb
woman	*the* woman who I saw yesterday ...	the woman *sat* down
Scotland	*the* Scotland I remember as a child ...	Scotland *has* five million people
book	*the* book I bought yesterday ...	this book *recounts* the colonization of Australia
intelligence	*the* intelligence displayed by the child ...	Mary's intelligence *impressed* her teachers

The simplified noun tags are `N` for common nouns like *book*, and `NP` for proper nouns like *Scotland*.

Let's inspect some tagged text to see what parts-of-speech occur before a noun, with the most frequent ones first. To begin with, we construct a list of bigrams whose members are themselves word-tag pairs, such as `(('The', 'DET'), ('Fulton', 'NP'))` and `(('Fulton', 'NP'), ('County', 'N'))`. Then we construct a `FreqDist` from the tag parts of the bigrams.

```
>>> word_tag_pairs = nltk.bigrams(brown_news_tagged)
>>> list(nltk.FreqDist(a[1] for (a, b) in word_tag_pairs if b[1] == 'N'))
['DET', 'ADJ', 'N', 'P', 'NP', 'NUM', 'V', 'PRO', 'CNJ', '.', ',', 'VG', 'VN', ...]
```

This confirms our assertion that nouns occur after determiners and adjectives, including numeral adjectives (tagged as `NUM`).

Verbs

Verbs are words that describe events and actions, e.g., *fall* and *eat*, as shown in Table 5-3. In the context of a sentence, verbs typically express a relation involving the referents of one or more noun phrases.

Table 5-3. Syntactic patterns involving some verbs

Word	Simple	With modifiers and adjuncts (italicized)
fall	Rome fell	Dot com stocks *suddenly* fell *like a stone*
eat	Mice eat cheese	John ate the pizza *with gusto*

What are the most common verbs in news text? Let's sort all the verbs by frequency:

```
>>> wsj = nltk.corpus.treebank.tagged_words(simplify_tags=True)
>>> word_tag_fd = nltk.FreqDist(wsj)
>>> [word + "/" + tag for (word, tag) in word_tag_fd if tag.startswith('V')]
['is/V', 'said/VD', 'was/VD', 'are/V', 'be/V', 'has/V', 'have/V', 'says/V',
'were/VD', 'had/VD', 'been/VN', "'s/V", 'do/V', 'say/V', 'make/V', 'did/VD',
'rose/VD', 'does/V', 'expected/VN', 'buy/V', 'take/V', 'get/V', 'sell/V',
'help/V', 'added/VD', 'including/VG', 'according/VG', 'made/VN', 'pay/V', ...]
```

Note that the items being counted in the frequency distribution are word-tag pairs. Since words and tags are paired, we can treat the word as a condition and the tag as an event, and initialize a conditional frequency distribution with a list of condition-event pairs. This lets us see a frequency-ordered list of tags given a word:

```
>>> cfd1 = nltk.ConditionalFreqDist(wsj)
>>> cfd1['yield'].keys()
['V', 'N']
>>> cfd1['cut'].keys()
['V', 'VD', 'N', 'VN']
```

We can reverse the order of the pairs, so that the tags are the conditions, and the words are the events. Now we can see likely words for a given tag:

```
>>> cfd2 = nltk.ConditionalFreqDist((tag, word) for (word, tag) in wsj)
>>> cfd2['VN'].keys()
['been', 'expected', 'made', 'compared', 'based', 'priced', 'used', 'sold',
'named', 'designed', 'held', 'fined', 'taken', 'paid', 'traded', 'said', ...]
```

To clarify the distinction between `VD` (past tense) and `VN` (past participle), let's find words that can be both `VD` and `VN`, and see some surrounding text:

```
>>> [w for w in cfd1.conditions() if 'VD' in cfd1[w] and 'VN' in cfd1[w]]
['Asked', 'accelerated', 'accepted', 'accused', 'acquired', 'added', 'adopted', ...]
>>> idx1 = wsj.index(('kicked', 'VD'))
>>> wsj[idx1-4:idx1+1]
[('While', 'P'), ('program', 'N'), ('trades', 'N'), ('swiftly', 'ADV'),
('kicked', 'VD')]
>>> idx2 = wsj.index(('kicked', 'VN'))
>>> wsj[idx2-4:idx2+1]
[('head', 'N'), ('of', 'P'), ('state', 'N'), ('has', 'V'), ('kicked', 'VN')]
```

In this case, we see that the past participle of *kicked* is preceded by a form of the auxiliary verb *have*. Is this generally true?

Your Turn: Given the list of past participles specified by `cfd2['VN'].keys()`, try to collect a list of all the word-tag pairs that immediately precede items in that list.

Adjectives and Adverbs

Two other important word classes are **adjectives** and **adverbs**. Adjectives describe nouns, and can be used as modifiers (e.g., *large* in *the large pizza*), or as predicates (e.g., *the pizza is large*). English adjectives can have internal structure (e.g., *fall+ing* in *the falling stocks*). Adverbs modify verbs to specify the time, manner, place, or direction of the event described by the verb (e.g., *quickly* in *the stocks fell quickly*). Adverbs may also modify adjectives (e.g., *really* in *Mary's teacher was really nice*).

English has several categories of closed class words in addition to prepositions, such as **articles** (also often called **determiners**) (e.g., *the*, *a*), **modals** (e.g., *should*, *may*), and **personal pronouns** (e.g., *she*, *they*). Each dictionary and grammar classifies these words differently.

Your Turn: If you are uncertain about some of these parts-of-speech, study them using `nltk.app.concordance()`, or watch some of the *Schoolhouse Rock!* grammar videos available at YouTube, or consult Section 5.9.

Unsimplified Tags

Let's find the most frequent nouns of each noun part-of-speech type. The program in Example 5-1 finds all tags starting with `NN`, and provides a few example words for each one. You will see that there are many variants of `NN`; the most important contain `$` for possessive nouns, `S` for plural nouns (since plural nouns typically end in *s*), and `P` for proper nouns. In addition, most of the tags have suffix modifiers: `-NC` for citations, `-HL` for words in headlines, and `-TL` for titles (a feature of Brown tags).

Example 5-1. Program to find the most frequent noun tags.

```
def findtags(tag_prefix, tagged_text):
    cfd = nltk.ConditionalFreqDist((tag, word) for (word, tag) in tagged_text
                                   if tag.startswith(tag_prefix))
    return dict((tag, cfd[tag].keys()[:5]) for tag in cfd.conditions())
```

```
>>> tagdict = findtags('NN', nltk.corpus.brown.tagged_words(categories='news'))
>>> for tag in sorted(tagdict):
...     print tag, tagdict[tag]
...
NN ['year', 'time', 'state', 'week', 'man']
NN$ ["year's", "world's", "state's", "nation's", "company's"]
NN$-HL ["Golf's", "Navy's"]
NN$-TL ["President's", "University's", "League's", "Gallery's", "Army's"]
NN-HL ['cut', 'Salary', 'condition', 'Question', 'business']
NN-NC ['eva', 'ova', 'aya']
NN-TL ['President', 'House', 'State', 'University', 'City']
NN-TL-HL ['Fort', 'City', 'Commissioner', 'Grove', 'House']
NNS ['years', 'members', 'people', 'sales', 'men']
NNS$ ["children's", "women's", "men's", "janitors'", "taxpayers'"]
NNS$-HL ["Dealers'", "Idols'"]
NNS$-TL ["Women's", "States'", "Giants'", "Officers'", "Bombers'"]
NNS-HL ['years', 'idols', 'Creations', 'thanks', 'centers']
NNS-TL ['States', 'Nations', 'Masters', 'Rules', 'Communists']
NNS-TL-HL ['Nations']
```

When we come to constructing part-of-speech taggers later in this chapter, we will use the unsimplified tags.

Exploring Tagged Corpora

Let's briefly return to the kinds of exploration of corpora we saw in previous chapters, this time exploiting POS tags.

Suppose we're studying the word *often* and want to see how it is used in text. We could ask to see the words that follow *often*:

```
>>> brown_learned_text = brown.words(categories='learned')
>>> sorted(set(b for (a, b) in nltk.ibigrams(brown_learned_text) if a == 'often'))
[',', '.', 'accomplished', 'analytically', 'appear', 'apt', 'associated', 'assuming',
'became', 'become', 'been', 'began', 'call', 'called', 'carefully', 'chose', ...]
```

However, it's probably more instructive use the `tagged_words()` method to look at the part-of-speech tag of the following words:

```
>>> brown_lrnd_tagged = brown.tagged_words(categories='learned', simplify_tags=True)
>>> tags = [b[1] for (a, b) in nltk.ibigrams(brown_lrnd_tagged) if a[0] == 'often']
>>> fd = nltk.FreqDist(tags)
>>> fd.tabulate()
  VN    V   VD  DET  ADJ  ADV    P  CNJ    ,   TO   VG   WH  VBZ    .
  15   12    8    5    5    4    4    3    3    1    1    1    1    1
```

Notice that the most high-frequency parts-of-speech following *often* are verbs. Nouns never appear in this position (in this particular corpus).

Next, let's look at some larger context, and find words involving particular sequences of tags and words (in this case "`<Verb> to <Verb>`"). In Example 5-2, we consider each three-word window in the sentence ❶, and check whether they meet our criterion ❷. If the tags match, we print the corresponding words ❸.

Example 5-2. Searching for three-word phrases using POS tags.

```
from nltk.corpus import brown
def process(sentence):
    for (w1,t1), (w2,t2), (w3,t3) in nltk.trigrams(sentence): ❶
        if (t1.startswith('V') and t2 == 'TO' and t3.startswith('V')): ❷
            print w1, w2, w3 ❸
```

```
>>> for tagged_sent in brown.tagged_sents():
...     process(tagged_sent)
...
combined to achieve
continue to place
serve to protect
wanted to wait
allowed to place
expected to become
...
```

Finally, let's look for words that are highly ambiguous as to their part-of-speech tag. Understanding why such words are tagged as they are in each context can help us clarify the distinctions between the tags.

```
>>> brown_news_tagged = brown.tagged_words(categories='news', simplify_tags=True)
>>> data = nltk.ConditionalFreqDist((word.lower(), tag)
...                                 for (word, tag) in brown_news_tagged)
>>> for word in data.conditions():
...     if len(data[word]) > 3:
...         tags = data[word].keys()
...         print word, ' '.join(tags)
...
best ADJ ADV NP V
better ADJ ADV V DET
close ADV ADJ V N
cut V N VN VD
even ADV DET ADJ V
grant NP N V -
hit V VD VN N
lay ADJ V NP VD
left VD ADJ N VN
```

```
like CNJ V ADJ P -
near P ADV ADJ DET
open ADJ V N ADV
past N ADJ DET P
present ADJ ADV V N
read V VN VD NP
right ADJ N DET ADV
second NUM ADV DET N
set VN V VD N -
that CNJ V WH DET
```

Your Turn: Open the POS concordance tool `nltk.app.concordance()` and load the complete Brown Corpus (simplified tagset). Now pick some of the words listed at the end of the previous code example and see how the tag of the word correlates with the context of the word. E.g., search for `near` to see all forms mixed together, `near/ADJ` to see it used as an adjective, `near N` to see just those cases where a noun follows, and so forth.

5.3 Mapping Words to Properties Using Python Dictionaries

As we have seen, a tagged word of the form `(word, tag)` is an association between a word and a part-of-speech tag. Once we start doing part-of-speech tagging, we will be creating programs that assign a tag to a word, the tag which is most likely in a given context. We can think of this process as **mapping** from words to tags. The most natural way to store mappings in Python uses the so-called **dictionary** data type (also known as an **associative array** or **hash array** in other programming languages). In this section, we look at dictionaries and see how they can represent a variety of language information, including parts-of-speech.

Indexing Lists Versus Dictionaries

A text, as we have seen, is treated in Python as a list of words. An important property of lists is that we can "look up" a particular item by giving its index, e.g., `text1[100]`. Notice how we specify a number and get back a word. We can think of a list as a simple kind of table, as shown in Figure 5-2.

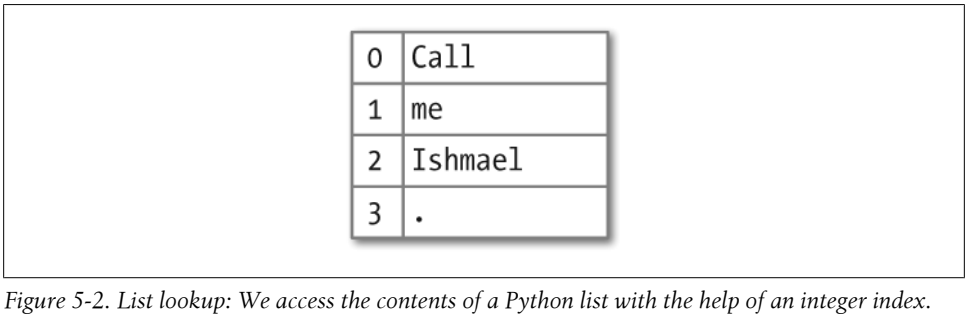


Figure 5-2. List lookup: We access the contents of a Python list with the help of an integer index.

Contrast this situation with frequency distributions (Section 1.3), where we specify a word and get back a number, e.g., `fdist['monstrous']`, which tells us the number of times a given word has occurred in a text. Lookup using words is familiar to anyone who has used a dictionary. Some more examples are shown in Figure 5-3.

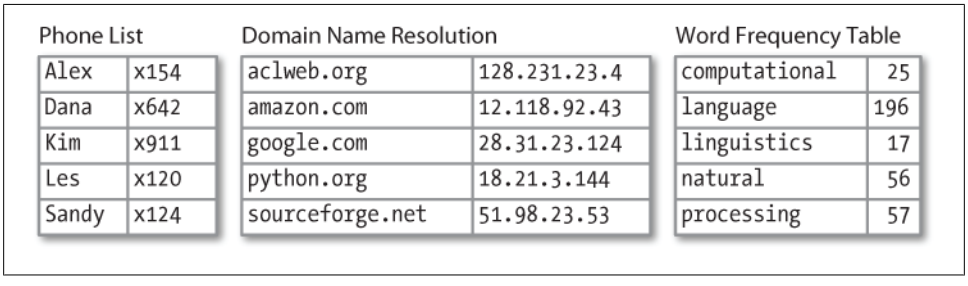


Figure 5-3. Dictionary lookup: we access the entry of a dictionary using a key such as someone's name, a web domain, or an English word; other names for dictionary are map, hashmap, hash, and associative array.

In the case of a phonebook, we look up an entry using a *name* and get back a number. When we type a domain name in a web browser, the computer looks this up to get back an IP address. A word frequency table allows us to look up a word and find its frequency in a text collection. In all these cases, we are mapping from names to numbers, rather than the other way around as with a list. In general, we would like to be able to map between arbitrary types of information. Table 5-4 lists a variety of linguistic objects, along with what they map.

Table 5-4. Linguistic objects as mappings from keys to values

Linguistic object	Maps from	Maps to
Document Index	Word	List of pages (where word is found)
Thesaurus	Word sense	List of synonyms
Dictionary	Headword	Entry (part-of-speech, sense definitions, etymology)
Comparative Wordlist	Gloss term	Cognates (list of words, one per language)
Morph Analyzer	Surface form	Morphological analysis (list of component morphemes)

Most often, we are mapping from a "word" to some structured object. For example, a document index maps from a word (which we can represent as a string) to a list of pages (represented as a list of integers). In this section, we will see how to represent such mappings in Python.

Dictionaries in Python

Python provides a **dictionary** data type that can be used for mapping between arbitrary types. It is like a conventional dictionary, in that it gives you an efficient way to look things up. However, as we see from Table 5-4, it has a much wider range of uses.

To illustrate, we define `pos` to be an empty dictionary and then add four entries to it, specifying the part-of-speech of some words. We add entries to a dictionary using the familiar square bracket notation:

```
>>> pos = {}
>>> pos
{}
>>> pos['colorless'] = 'ADJ' ❶
>>> pos
{'colorless': 'ADJ'}
>>> pos['ideas'] = 'N'
>>> pos['sleep'] = 'V'
>>> pos['furiously'] = 'ADV'
>>> pos ❷
{'furiously': 'ADV', 'ideas': 'N', 'colorless': 'ADJ', 'sleep': 'V'}
```

So, for example, ❶ says that the part-of-speech of *colorless* is adjective, or more specifically, that the **key** `'colorless'` is assigned the **value** `'ADJ'` in dictionary `pos`. When we inspect the value of `pos` ❷ we see a set of key-value pairs. Once we have populated the dictionary in this way, we can employ the keys to retrieve values:

```
>>> pos['ideas']
'N'
>>> pos['colorless']
'ADJ'
```

Of course, we might accidentally use a key that hasn't been assigned a value.

```
>>> pos['green']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'green'
```

This raises an important question. Unlike lists and strings, where we can use `len()` to work out which integers will be legal indexes, how do we work out the legal keys for a dictionary? If the dictionary is not too big, we can simply inspect its contents by evaluating the variable `pos`. As we saw earlier in line ❷, this gives us the key-value pairs. Notice that they are not in the same order they were originally entered; this is because dictionaries are not sequences but mappings (see Figure 5-3), and the keys are not inherently ordered.

Alternatively, to just find the keys, we can either convert the dictionary to a list ❶ or use the dictionary in a context where a list is expected, as the parameter of `sorted()` ❷ or in a `for` loop ❸.

```
>>> list(pos) ❶
['ideas', 'furiously', 'colorless', 'sleep']
>>> sorted(pos) ❷
['colorless', 'furiously', 'ideas', 'sleep']
>>> [w for w in pos if w.endswith('s')] ❸
['colorless', 'ideas']
```

When you type `list(pos)`, you might see a different order to the one shown here. If you want to see the keys in order, just sort them.

As well as iterating over all keys in the dictionary with a `for` loop, we can use the `for` loop as we did for printing lists:

```
>>> for word in sorted(pos):
...     print word + ":", pos[word]
...
colorless: ADJ
furiously: ADV
sleep: V
ideas: N
```

Finally, the dictionary methods `keys()`, `values()`, and `items()` allow us to access the keys, values, and key-value pairs as separate lists. We can even sort tuples ❶, which orders them according to their first element (and if the first elements are the same, it uses their second elements).

```
>>> pos.keys()
['colorless', 'furiously', 'sleep', 'ideas']
>>> pos.values()
['ADJ', 'ADV', 'V', 'N']
>>> pos.items()
[('colorless', 'ADJ'), ('furiously', 'ADV'), ('sleep', 'V'), ('ideas', 'N')]
>>> for key, val in sorted(pos.items()): ❶
...     print key + ":", val
...
colorless: ADJ
furiously: ADV
ideas: N
sleep: V
```

We want to be sure that when we look something up in a dictionary, we get only one value for each key. Now suppose we try to use a dictionary to store the fact that the word *sleep* can be used as both a verb and a noun:

```
>>> pos['sleep'] = 'V'
>>> pos['sleep']
'V'
>>> pos['sleep'] = 'N'
>>> pos['sleep']
'N'
```

Initially, `pos['sleep']` is given the value `'V'`. But this is immediately overwritten with the new value, `'N'`. In other words, there can be only one entry in the dictionary for `'sleep'`. However, there is a way of storing multiple values in that entry: we use a list value, e.g., `pos['sleep'] = ['N', 'V']`. In fact, this is what we saw in Section 2.4 for the CMU Pronouncing Dictionary, which stores multiple pronunciations for a single word.

Defining Dictionaries

We can use the same key-value pair format to create a dictionary. There are a couple of ways to do this, and we will normally use the first:

```
>>> pos = {'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}
>>> pos = dict(colorless='ADJ', ideas='N', sleep='V', furiously='ADV')
```

Note that dictionary keys must be immutable types, such as strings and tuples. If we try to define a dictionary using a mutable key, we get a `TypeError`:

```
>>> pos = {['ideas', 'blogs', 'adventures']: 'N'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

Default Dictionaries

If we try to access a key that is not in a dictionary, we get an error. However, it's often useful if a dictionary can automatically create an entry for this new key and give it a default value, such as zero or the empty list. Since Python 2.5, a special kind of dictionary called a `defaultdict` has been available. (It is provided as `nltk.defaultdict` for the benefit of readers who are using Python 2.4.) In order to use it, we have to supply a parameter which can be used to create the default value, e.g., `int`, `float`, `str`, `list`, `dict`, `tuple`.

```
>>> frequency = nltk.defaultdict(int)
>>> frequency['colorless'] = 4
>>> frequency['ideas']
0
>>> pos = nltk.defaultdict(list)
>>> pos['sleep'] = ['N', 'V']
>>> pos['ideas']
[]
```

These default values are actually functions that convert other objects to the specified type (e.g., `int("2")`, `list("2")`). When they are called with no parameter—say, `int()`, `list()`—they return `0` and `[]` respectively.

The preceding examples specified the default value of a dictionary entry to be the default value of a particular data type. However, we can specify any default value we like, simply by providing the name of a function that can be called with no arguments to create the required value. Let's return to our part-of-speech example, and create a dictionary whose default value for any entry is `'N'` ❶. When we access a non-existent entry ❷, it is automatically added to the dictionary ❸.

```
>>> pos = nltk.defaultdict(lambda: 'N') ❶
>>> pos['colorless'] = 'ADJ'
>>> pos['blog'] ❷
'N'
```

```
>>> pos.items()
[('blog', 'N'), ('colorless', 'ADJ')] ❸
```

This example used a *lambda expression*, introduced in Section 4.4. This lambda expression specifies no parameters, so we call it using parentheses with no arguments. Thus, the following definitions of `f` and `g` are equivalent:

```
>>> f = lambda: 'N'
>>> f()
'N'
>>> def g():
...     return 'N'
>>> g()
'N'
```

Let's see how default dictionaries could be used in a more substantial language processing task. Many language processing tasks—including tagging—struggle to correctly process the hapaxes of a text. They can perform better with a fixed vocabulary and a guarantee that no new words will appear. We can preprocess a text to replace low-frequency words with a special "out of vocabulary" token, `UNK`, with the help of a default dictionary. (Can you work out how to do this without reading on?)

We need to create a default dictionary that maps each word to its replacement. The most frequent *n* words will be mapped to themselves. Everything else will be mapped to `UNK`.

```
>>> alice = nltk.corpus.gutenberg.words('carroll-alice.txt')
>>> vocab = nltk.FreqDist(alice)
>>> v1000 = list(vocab)[:1000]
>>> mapping = nltk.defaultdict(lambda: 'UNK')
>>> for v in v1000:
...     mapping[v] = v
...
>>> alice2 = [mapping[v] for v in alice]
>>> alice2[:100]
['UNK', 'Alice', "'", 's', 'Adventures', 'in', 'Wonderland', 'by', 'UNK', 'UNK',
'UNK', 'UNK', 'CHAPTER', 'I', '.', 'UNK', 'the', 'Rabbit', '-', 'UNK', 'Alice',
'was', 'beginning', 'to', 'get', 'very', 'tired', 'of', 'sitting', 'by', 'her',
'sister', 'on', 'the', 'bank', ',', 'and', 'of', 'having', 'nothing', 'to', 'do',
':', 'once', 'or', 'twice', 'she', 'had', 'UNK', 'into', 'the', 'book', 'her',
'sister', 'was', 'UNK', ',', 'but', 'it', 'had', 'no', 'pictures', 'or', 'UNK',
'in', 'it', ',', "'", 'and', 'what', 'is', 'the', 'use', 'of', 'a', 'book', ",'",
'thought', 'Alice', "'", 'without', 'pictures', 'or', 'conversation', "?'", ...]
>>> len(set(alice2))
1001
```

Incrementally Updating a Dictionary

We can employ dictionaries to count occurrences, emulating the method for tallying words shown in Figure 1-3. We begin by initializing an empty `defaultdict`, then process each part-of-speech tag in the text. If the tag hasn't been seen before, it will have a zero

count by default. Each time we encounter a tag, we increment its count using the `+=` operator (see Example 5-3).

Example 5-3. Incrementally updating a dictionary, and sorting by value.

```
>>> counts = nltk.defaultdict(int)
>>> from nltk.corpus import brown
>>> for (word, tag) in brown.tagged_words(categories='news'):
...     counts[tag] += 1
...
>>> counts['N']
22226
>>> list(counts)
['FW', 'DET', 'WH', "''", 'VBZ', 'VB+PPO', "'", ')', 'ADJ', 'PRO', '*', '-', ...]

>>> from operator import itemgetter
>>> sorted(counts.items(), key=itemgetter(1), reverse=True)
[('N', 22226), ('P', 10845), ('DET', 10648), ('NP', 8336), ('V', 7313), ...]
>>> [t for t, c in sorted(counts.items(), key=itemgetter(1), reverse=True)]
['N', 'P', 'DET', 'NP', 'V', 'ADJ', ',', '.', 'CNJ', 'PRO', 'ADV', 'VD', ...]
```

The listing in Example 5-3 illustrates an important idiom for sorting a dictionary by its values, to show words in decreasing order of frequency. The first parameter of `sorted()` is the items to sort, which is a list of tuples consisting of a POS tag and a frequency. The second parameter specifies the sort key using a function `itemgetter()`. In general, `itemgetter(n)` returns a function that can be called on some other sequence object to obtain the *n*th element:

```
>>> pair = ('NP', 8336)
>>> pair[1]
8336
>>> itemgetter(1)(pair)
8336
```

The last parameter of `sorted()` specifies that the items should be returned in reverse order, i.e., decreasing values of frequency.

There's a second useful programming idiom at the beginning of Example 5-3, where we initialize a `defaultdict` and then use a `for` loop to update its values. Here's a schematic version:

```
>>> my_dictionary = nltk.defaultdict(function to create default value)
>>> for item in sequence:
...      my_dictionary[item_key] is updated with information about item
```

Here's another instance of this pattern, where we index words according to their last two letters:

```
>>> last_letters = nltk.defaultdict(list)
>>> words = nltk.corpus.words.words('en')
>>> for word in words:
...     key = word[-2:]
...     last_letters[key].append(word)
...
```

```
>>> last_letters['ly']
['abactinally', 'abandonedly', 'abasedly', 'abashedly', 'abashlessly', 'abbreviately',
'abdominally', 'abhorrently', 'abidingly', 'abiogenetically', 'abiologically', ...]
>>> last_letters['zy']
['blazy', 'bleezy', 'blowzy', 'boozy', 'breezy', 'bronzy', 'buzzy', 'Chazy', ...]
```

The following example uses the same pattern to create an anagram dictionary. (You might experiment with the third line to get an idea of why this program works.)

```
>>> anagrams = nltk.defaultdict(list)
>>> for word in words:
...     key = ''.join(sorted(word))
...     anagrams[key].append(word)
...
>>> anagrams['aeilnrt']
['entrail', 'latrine', 'ratline', 'reliant', 'retinal', 'trenail']
```

Since accumulating words like this is such a common task, NLTK provides a more convenient way of creating a `defaultdict(list)`, in the form of `nltk.Index()`:

```
>>> anagrams = nltk.Index((''.join(sorted(w)), w) for w in words)
>>> anagrams['aeilnrt']
['entrail', 'latrine', 'ratline', 'reliant', 'retinal', 'trenail']
```

`nltk.Index` is a `defaultdict(list)` with extra support for initialization. Similarly, `nltk.FreqDist` is essentially a `defaultdict(int)` with extra support for initialization (along with sorting and plotting methods).

Complex Keys and Values

We can use default dictionaries with complex keys and values. Let's study the range of possible tags for a word, given the word itself and the tag of the previous word. We will see how this information can be used by a POS tagger.

```
>>> pos = nltk.defaultdict(lambda: nltk.defaultdict(int))
>>> brown_news_tagged = brown.tagged_words(categories='news', simplify_tags=True)
>>> for ((w1, t1), (w2, t2)) in nltk.ibigrams(brown_news_tagged): ❶
...     pos[(t1, w2)][t2] += 1 ❷
...
>>> pos[('DET', 'right')] ❸
defaultdict(<type 'int'>, {'ADV': 3, 'ADJ': 9, 'N': 3})
```

This example uses a dictionary whose default value for an entry is a dictionary (whose default value is `int()`, i.e., zero). Notice how we iterated over the bigrams of the tagged corpus, processing a pair of word-tag pairs for each iteration ❶. Each time through the loop we updated our `pos` dictionary's entry for `(t1, w2)`, a tag and its *following* word ❷. When we look up an item in `pos` we must specify a compound key ❸, and we get back a dictionary object. A POS tagger could use such information to decide that the word *right*, when preceded by a determiner, should be tagged as `ADJ`.

Inverting a Dictionary

Dictionaries support efficient lookup, so long as you want to get the value for any key. If `d` is a dictionary and `k` is a key, we type `d[k]` and immediately obtain the value. Finding a key given a value is slower and more cumbersome:

```
>>> counts = nltk.defaultdict(int)
>>> for word in nltk.corpus.gutenberg.words('milton-paradise.txt'):
...     counts[word] += 1
...
>>> [key for (key, value) in counts.items() if value == 32]
['brought', 'Him', 'virtue', 'Against', 'There', 'thine', 'King', 'mortal',
'every', 'been']
```

If we expect to do this kind of "reverse lookup" often, it helps to construct a dictionary that maps values to keys. In the case that no two keys have the same value, this is an easy thing to do. We just get all the key-value pairs in the dictionary, and create a new dictionary of value-key pairs. The next example also illustrates another way of initializing a dictionary `pos` with key-value pairs.

```
>>> pos = {'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}
>>> pos2 = dict((value, key) for (key, value) in pos.items())
>>> pos2['N']
'ideas'
```

Let's first make our part-of-speech dictionary a bit more realistic and add some more words to `pos` using the dictionary `update()` method, to create the situation where multiple keys have the same value. Then the technique just shown for reverse lookup will no longer work (why not?). Instead, we have to use `append()` to accumulate the words for each part-of-speech, as follows:

```
>>> pos.update({'cats': 'N', 'scratch': 'V', 'peacefully': 'ADV', 'old': 'ADJ'})
>>> pos2 = nltk.defaultdict(list)
>>> for key, value in pos.items():
...     pos2[value].append(key)
...
>>> pos2['ADV']
['peacefully', 'furiously']
```

Now we have inverted the `pos` dictionary, and can look up any part-of-speech and find all words having that part-of-speech. We can do the same thing even more simply using NLTK's support for indexing, as follows:

```
>>> pos2 = nltk.Index((value, key) for (key, value) in pos.items())
>>> pos2['ADV']
['peacefully', 'furiously']
```

A summary of Python's dictionary methods is given in Table 5-5.

Table 5-5. Python's dictionary methods: A summary of commonly used methods and idioms involving dictionaries

Example	Description
`d = {}`	Create an empty dictionary and assign it to d
`d[key] = value`	Assign a value to a given dictionary key
`d.keys()`	The list of keys of the dictionary
`list(d)`	The list of keys of the dictionary
`sorted(d)`	The keys of the dictionary, sorted
`key in d`	Test whether a particular key is in the dictionary
`for key in d`	Iterate over the keys of the dictionary
`d.values()`	The list of values in the dictionary
`dict([(k1,v1), (k2,v2), ...])`	Create a dictionary from a list of key-value pairs
`d1.update(d2)`	Add all items from d2 to d1
`defaultdict(int)`	A dictionary whose default value is zero

5.4 Automatic Tagging

In the rest of this chapter we will explore various ways to automatically add part-of-speech tags to text. We will see that the tag of a word depends on the word and its context within a sentence. For this reason, we will be working with data at the level of (tagged) sentences rather than words. We'll begin by loading the data we will be using.

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
```

The Default Tagger

The simplest possible tagger assigns the same tag to each token. This may seem to be a rather banal step, but it establishes an important baseline for tagger performance. In order to get the best result, we tag each word with the most likely tag. Let's find out which tag is most likely (now using the unsimplified tagset):

```
>>> tags = [tag for (word, tag) in brown.tagged_words(categories='news')]
>>> nltk.FreqDist(tags).max()
'NN'
```

Now we can create a tagger that tags everything as NN.

```
>>> raw = 'I do not like green eggs and ham, I do not like them Sam I am!'
>>> tokens = nltk.word_tokenize(raw)
>>> default_tagger = nltk.DefaultTagger('NN')
>>> default_tagger.tag(tokens)
[('I', 'NN'), ('do', 'NN'), ('not', 'NN'), ('like', 'NN'), ('green', 'NN'),
('eggs', 'NN'), ('and', 'NN'), ('ham', 'NN'), (',', 'NN'), ('I', 'NN'),
```

```
('do', 'NN'), ('not', 'NN'), ('like', 'NN'), ('them', 'NN'), ('Sam', 'NN'),
('I', 'NN'), ('am', 'NN'), ('!', 'NN')]
```

Unsurprisingly, this method performs rather poorly. On a typical corpus, it will tag only about an eighth of the tokens correctly, as we see here:

```
>>> default_tagger.evaluate(brown_tagged_sents)
0.13089484257215028
```

Default taggers assign their tag to every single word, even words that have never been encountered before. As it happens, once we have processed several thousand words of English text, most new words will be nouns. As we will see, this means that default taggers can help to improve the robustness of a language processing system. We will return to them shortly.

The Regular Expression Tagger

The regular expression tagger assigns tags to tokens on the basis of matching patterns. For instance, we might guess that any word ending in *ed* is the past participle of a verb, and any word ending with *'s* is a possessive noun. We can express these as a list of regular expressions:

```
>>> patterns = [
...     (r'.*ing$', 'VBG'),               # gerunds
...     (r'.*ed$', 'VBD'),                # simple past
...     (r'.*es$', 'VBZ'),                # 3rd singular present
...     (r'.*ould$', 'MD'),               # modals
...     (r'.*\'s$', 'NN$'),               # possessive nouns
...     (r'.*s$', 'NNS'),                 # plural nouns
...     (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  # cardinal numbers
...     (r'.*', 'NN')                     # nouns (default)
... ]
```

Note that these are processed in order, and the first one that matches is applied. Now we can set up a tagger and use it to tag a sentence. After this step, it is correct about a fifth of the time.

```
>>> regexp_tagger = nltk.RegexpTagger(patterns)
>>> regexp_tagger.tag(brown_sents[3])
[('``', 'NN'), ('Only', 'NN'), ('a', 'NN'), ('relative', 'NN'), ('handful', 'NN'),
('of', 'NN'), ('such', 'NN'), ('reports', 'NNS'), ('was', 'NNS'), ('received', 'VBD'),
("''", 'NN'), (',', 'NN'), ('the', 'NN'), ('jury', 'NN'), ('said', 'NN'), (',', 'NN'),
('``', 'NN'), ('considering', 'VBG'), ('the', 'NN'), ('widespread', 'NN'), ...]
>>> regexp_tagger.evaluate(brown_tagged_sents)
0.20326391789486245
```

The final regular expression «`.*`» is a catch-all that tags everything as a noun. This is equivalent to the default tagger (only much less efficient). Instead of respecifying this as part of the regular expression tagger, is there a way to combine this tagger with the default tagger? We will see how to do this shortly.

Your Turn: See if you can come up with patterns to improve the performance of the regular expression tagger just shown. (Note that Section 6.1 describes a way to partially automate such work.)

The Lookup Tagger

A lot of high-frequency words do not have the NN tag. Let's find the hundred most frequent words and store their most likely tag. We can then use this information as the model for a "lookup tagger" (an NLTK UnigramTagger):

```
>>> fd = nltk.FreqDist(brown.words(categories='news'))
>>> cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
>>> most_freq_words = fd.keys()[:100]
>>> likely_tags = dict((word, cfd[word].max()) for word in most_freq_words)
>>> baseline_tagger = nltk.UnigramTagger(model=likely_tags)
>>> baseline_tagger.evaluate(brown_tagged_sents)
0.45578495136941344
```

It should come as no surprise by now that simply knowing the tags for the 100 most frequent words enables us to tag a large fraction of tokens correctly (nearly half, in fact). Let's see what it does on some untagged input text:

```
>>> sent = brown.sents(categories='news')[3]
>>> baseline_tagger.tag(sent)
[('``', '``'), ('Only', None), ('a', 'AT'), ('relative', None),
('handful', None), ('of', 'IN'), ('such', None), ('reports', None),
('was', 'BEDZ'), ('received', None), ("''", "''"), (',', ','),
('the', 'AT'), ('jury', None), ('said', 'VBD'), (',', ','),
('``', '``'), ('considering', None), ('the', 'AT'), ('widespread', None),
('interest', None), ('in', 'IN'), ('the', 'AT'), ('election', None),
(',', ','), ('the', 'AT'), ('number', None), ('of', 'IN'),
('voters', None), ('and', 'CC'), ('the', 'AT'), ('size', None),
('of', 'IN'), ('this', 'DT'), ('city', None), ("''", "''"), ('.', '.')]
```

Many words have been assigned a tag of None, because they were not among the 100 most frequent words. In these cases we would like to assign the default tag of NN. In other words, we want to use the lookup table first, and if it is unable to assign a tag, then use the default tagger, a process known as **backoff** (Section 5.5). We do this by specifying one tagger as a parameter to the other, as shown next. Now the lookup tagger will only store word-tag pairs for words other than nouns, and whenever it cannot assign a tag to a word, it will invoke the default tagger.

```
>>> baseline_tagger = nltk.UnigramTagger(model=likely_tags,
...                                      backoff=nltk.DefaultTagger('NN'))
```

Let's put all this together and write a program to create and evaluate lookup taggers having a range of sizes (Example 5-4).

Example 5-4. Lookup tagger performance with varying model size.

```
def performance(cfd, wordlist):
    lt = dict((word, cfd[word].max()) for word in wordlist)
    baseline_tagger = nltk.UnigramTagger(model=lt, backoff=nltk.DefaultTagger('NN'))
    return baseline_tagger.evaluate(brown.tagged_sents(categories='news'))

def display():
    import pylab
    words_by_freq = list(nltk.FreqDist(brown.words(categories='news')))
    cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
    sizes = 2 ** pylab.arange(15)
    perfs = [performance(cfd, words_by_freq[:size]) for size in sizes]
    pylab.plot(sizes, perfs, '-bo')
    pylab.title('Lookup Tagger Performance with Varying Model Size')
    pylab.xlabel('Model Size')
    pylab.ylabel('Performance')
    pylab.show()
```

```
>>> display()
```

Observe in Figure 5-4 that performance initially increases rapidly as the model size grows, eventually reaching a plateau, when large increases in model size yield little improvement in performance. (This example used the `pylab` plotting package, discussed in Section 4.8.)

Evaluation

In the previous examples, you will have noticed an emphasis on accuracy scores. In fact, evaluating the performance of such tools is a central theme in NLP. Recall the processing pipeline in Figure 1-5; any errors in the output of one module are greatly multiplied in the downstream modules.

We evaluate the performance of a tagger relative to the tags a human expert would assign. Since we usually don't have access to an expert and impartial human judge, we make do instead with **gold standard** test data. This is a corpus which has been manually annotated and accepted as a standard against which the guesses of an automatic system are assessed. The tagger is regarded as being correct if the tag it guesses for a given word is the same as the gold standard tag.

Of course, the humans who designed and carried out the original gold standard annotation were only human. Further analysis might show mistakes in the gold standard, or may eventually lead to a revised tagset and more elaborate guidelines. Nevertheless, the gold standard is by definition "correct" as far as the evaluation of an automatic tagger is concerned.

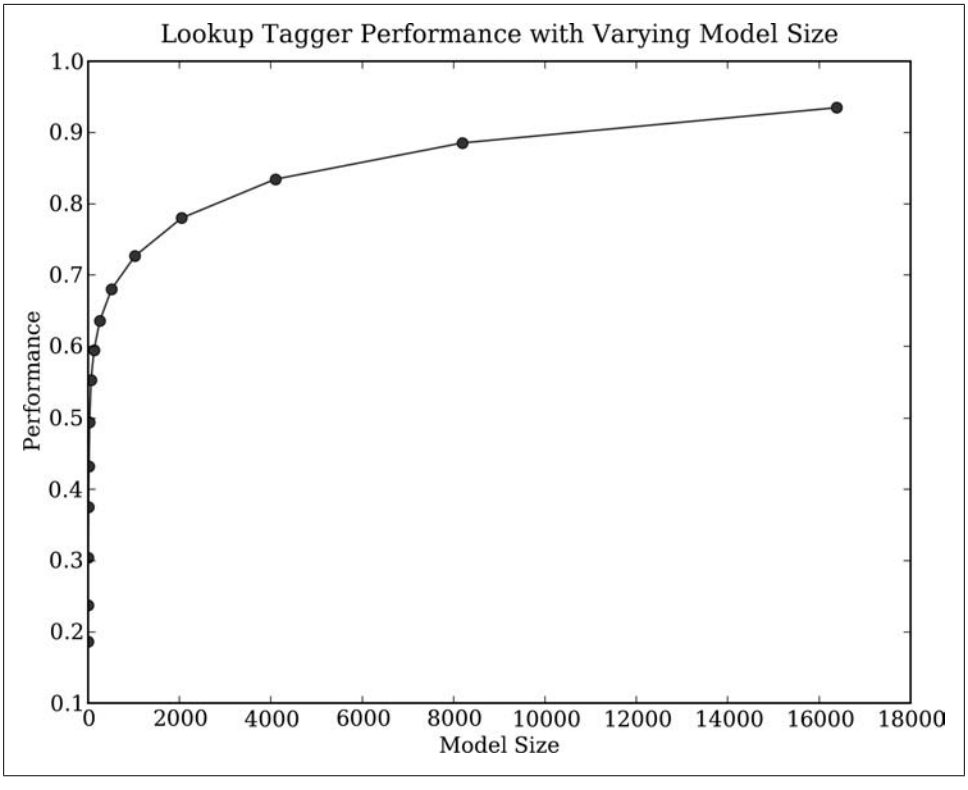


Figure 5-4. Lookup tagger

Developing an annotated corpus is a major undertaking. Apart from the data, it generates sophisticated tools, documentation, and practices for ensuring high-quality annotation. The tagsets and other coding schemes inevitably depend on some theoretical position that is not shared by all. However, corpus creators often go to great lengths to make their work as theory-neutral as possible in order to maximize the usefulness of their work. We will discuss the challenges of creating a corpus in Chapter 11.

5.5 N-Gram Tagging

Unigram Tagging

Unigram taggers are based on a simple statistical algorithm: for each token, assign the tag that is most likely for that particular token. For example, it will assign the tag `JJ` to any occurrence of the word *frequent*, since *frequent* is used as an adjective (e.g., *a frequent word*) more often than it is used as a verb (e.g., *I frequent this cafe*). A unigram tagger behaves just like a lookup tagger (Section 5.4), except there is a more convenient

technique for setting it up, called **training**. In the following code sample, we train a unigram tagger, use it to tag a sentence, and then evaluate:

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> unigram_tagger = nltk.UnigramTagger(brown_tagged_sents)
>>> unigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments', 'NNS'),
('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace', 'NN'), ('type', 'NN'),
(',', ','), ('being', 'BEG'), ('on', 'IN'), ('the', 'AT'), ('ground', 'NN'),
('floor', 'NN'), ('so', 'QL'), ('that', 'CS'), ('entrance', 'NN'), ('is', 'BEZ'),
('direct', 'JJ'), ('.', '.')]
>>> unigram_tagger.evaluate(brown_tagged_sents)
0.9349006503968017
```

We **train** a `UnigramTagger` by specifying tagged sentence data as a parameter when we initialize the tagger. The training process involves inspecting the tag of each word and storing the most likely tag for any word in a dictionary that is stored inside the tagger.

Separating the Training and Testing Data

Now that we are training a tagger on some data, we must be careful not to test it on the same data, as we did in the previous example. A tagger that simply memorized its training data and made no attempt to construct a general model would get a perfect score, but would be useless for tagging new text. Instead, we should split the data, training on 90% and testing on the remaining 10%:

```
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> size
4160
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
>>> unigram_tagger = nltk.UnigramTagger(train_sents)
>>> unigram_tagger.evaluate(test_sents)
0.81202033290142528
```

Although the score is worse, we now have a better picture of the usefulness of this tagger, i.e., its performance on previously unseen text.

General N-Gram Tagging

When we perform a language processing task based on unigrams, we are using one item of context. In the case of tagging, we consider only the current token, in isolation from any larger context. Given such a model, the best we can do is tag each word with its *a priori* most likely tag. This means we would tag a word such as *wind* with the same tag, regardless of whether it appears in the context *the wind* or *to wind*.

An **n-gram tagger** is a generalization of a unigram tagger whose context is the current word together with the part-of-speech tags of the *n*-1 preceding tokens, as shown in Figure 5-5. The tag to be chosen, t_n, is circled, and the context is shaded in grey. In the example of an n-gram tagger shown in Figure 5-5, we have *n*=3; that is, we consider

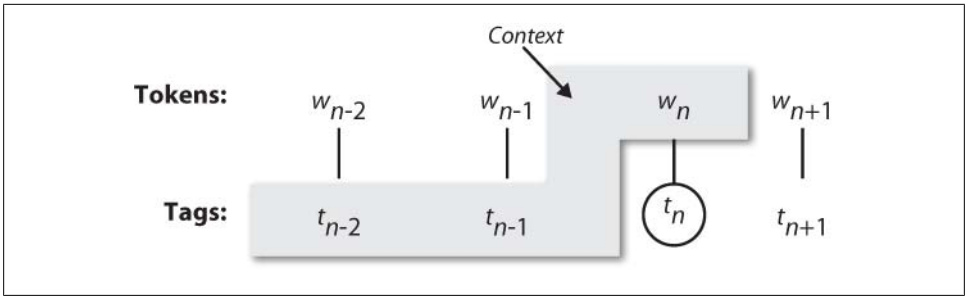


Figure 5-5. Tagger context.

the tags of the two preceding words in addition to the current word. An n-gram tagger picks the tag that is most likely in the given context.

A 1-gram tagger is another term for a unigram tagger: i.e., the context used to tag a token is just the text of the token itself. 2-gram taggers are also called *bigram taggers*, and 3-gram taggers are called *trigram taggers*.

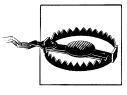
The `NgramTagger` class uses a tagged training corpus to determine which part-of-speech tag is most likely for each context. Here we see a special case of an n-gram tagger, namely a bigram tagger. First we train it, then use it to tag untagged sentences:

```
>>> bigram_tagger = nltk.BigramTagger(train_sents)
>>> bigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments', 'NNS'),
('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace', 'NN'),
('type', 'NN'), (',', ','), ('being', 'BEG'), ('on', 'IN'), ('the', 'AT'),
('ground', 'NN'), ('floor', 'NN'), ('so', 'CS'), ('that', 'CS'),
('entrance', 'NN'), ('is', 'BEZ'), ('direct', 'JJ'), ('.', '.')]
>>> unseen_sent = brown_sents[4203]
>>> bigram_tagger.tag(unseen_sent)
[('The', 'AT'), ('population', 'NN'), ('of', 'IN'), ('the', 'AT'), ('Congo', 'NP'),
('is', 'BEZ'), ('13.5', None), ('million', None), (',', None), ('divided', None),
('into', None), ('at', None), ('least', None), ('seven', None), ('major', None),
('``', None), ('culture', None), ('clusters', None), ("''", None), ('and', None),
('innumerable', None), ('tribes', None), ('speaking', None), ('400', None),
('separate', None), ('dialects', None), ('.', None)]
```

Notice that the bigram tagger manages to tag every word in a sentence it saw during training, but does badly on an unseen sentence. As soon as it encounters a new word (i.e., *13.5*), it is unable to assign a tag. It cannot tag the following word (i.e., *million*), even if it was seen during training, simply because it never saw it during training with a `None` tag on the previous word. Consequently, the tagger fails to tag the rest of the sentence. Its overall accuracy score is very low:

```
>>> bigram_tagger.evaluate(test_sents)
0.10276088906608193
```

As *n* gets larger, the specificity of the contexts increases, as does the chance that the data we wish to tag contains contexts that were not present in the training data. This is known as the *sparse data* problem, and is quite pervasive in NLP. As a consequence, there is a trade-off between the accuracy and the coverage of our results (and this is related to the **precision/recall trade-off** in information retrieval).



Caution!

N-gram taggers should not consider context that crosses a sentence boundary. Accordingly, NLTK taggers are designed to work with lists of sentences, where each sentence is a list of words. At the start of a sentence, t_{n-1} and preceding tags are set to `None`.

Combining Taggers

One way to address the trade-off between accuracy and coverage is to use the more accurate algorithms when we can, but to fall back on algorithms with wider coverage when necessary. For example, we could combine the results of a bigram tagger, a unigram tagger, and a default tagger, as follows:

1. Try tagging the token with the bigram tagger.
2. If the bigram tagger is unable to find a tag for the token, try the unigram tagger.
3. If the unigram tagger is also unable to find a tag, use a default tagger.

Most NLTK taggers permit a backoff tagger to be specified. The backoff tagger may itself have a backoff tagger:

```
>>> t0 = nltk.DefaultTagger('NN')
>>> t1 = nltk.UnigramTagger(train_sents, backoff=t0)
>>> t2 = nltk.BigramTagger(train_sents, backoff=t1)
>>> t2.evaluate(test_sents)
0.84491179108940495
```

Your Turn: Extend the preceding example by defining a `TrigramTagger` called `t3`, which backs off to `t2`.

Note that we specify the backoff tagger when the tagger is initialized so that training can take advantage of the backoff tagger. Thus, if the bigram tagger would assign the same tag as its unigram backoff tagger in a certain context, the bigram tagger discards the training instance. This keeps the bigram tagger model as small as possible. We can further specify that a tagger needs to see more than one instance of a context in order to retain it. For example, `nltk.BigramTagger(sents, cutoff=2, backoff=t1)` will discard contexts that have only been seen once or twice.

Tagging Unknown Words

Our approach to tagging unknown words still uses backoff to a regular expression tagger or a default tagger. These are unable to make use of context. Thus, if our tagger encountered the word *blog*, not seen during training, it would assign it the same tag, regardless of whether this word appeared in the context *the blog* or *to blog*. How can we do better with these unknown words, or **out-of-vocabulary** items?

A useful method to tag unknown words based on context is to limit the vocabulary of a tagger to the most frequent *n* words, and to replace every other word with a special word *UNK* using the method shown in Section 5.3. During training, a unigram tagger will probably learn that *UNK* is usually a noun. However, the n-gram taggers will detect contexts in which it has some other tag. For example, if the preceding word is *to* (tagged `TO`), then *UNK* will probably be tagged as a verb.

Storing Taggers

Training a tagger on a large corpus may take a significant time. Instead of training a tagger every time we need one, it is convenient to save a trained tagger in a file for later reuse. Let's save our tagger `t2` to a file *t2.pkl*:

```
>>> from cPickle import dump
>>> output = open('t2.pkl', 'wb')
>>> dump(t2, output, -1)
>>> output.close()
```

Now, in a separate Python process, we can load our saved tagger:

```
>>> from cPickle import load
>>> input = open('t2.pkl', 'rb')
>>> tagger = load(input)
>>> input.close()
```

Now let's check that it can be used for tagging:

```
>>> text = """The board's action shows what free enterprise
...     is up against in our complex maze of regulatory laws ."""
>>> tokens = text.split()
>>> tagger.tag(tokens)
[('The', 'AT'), ("board's", 'NN$'), ('action', 'NN'), ('shows', 'NNS'),
('what', 'WDT'), ('free', 'JJ'), ('enterprise', 'NN'), ('is', 'BEZ'),
('up', 'RP'), ('against', 'IN'), ('in', 'IN'), ('our', 'PP$'), ('complex', 'JJ'),
('maze', 'NN'), ('of', 'IN'), ('regulatory', 'NN'), ('laws', 'NNS'), ('.', '.')]
```

Performance Limitations

What is the upper limit to the performance of an n-gram tagger? Consider the case of a trigram tagger. How many cases of part-of-speech ambiguity does it encounter? We can determine the answer to this question empirically:

```
>>> cfd = nltk.ConditionalFreqDist(
...            ((x[1], y[1], z[0]), z[1])
...            for sent in brown_tagged_sents
...            for x, y, z in nltk.trigrams(sent))
>>> ambiguous_contexts = [c for c in cfd.conditions() if len(cfd[c]) > 1]
>>> sum(cfd[c].N() for c in ambiguous_contexts) / cfd.N()
0.049297702068029296
```

Thus, 1 out of 20 trigrams is ambiguous. Given the current word and the previous two tags, in 5% of cases there is more than one tag that could be legitimately assigned to the current word according to the training data. Assuming we always pick the most likely tag in such ambiguous contexts, we can derive a lower bound on the performance of a trigram tagger.

Another way to investigate the performance of a tagger is to study its mistakes. Some tags may be harder than others to assign, and it might be possible to treat them specially by pre- or post-processing the data. A convenient way to look at tagging errors is the **confusion matrix**. It charts expected tags (the gold standard) against actual tags generated by a tagger:

```
>>> test_tags = [tag for sent in brown.sents(categories='editorial')
...                  for (word, tag) in t2.tag(sent)]
>>> gold_tags = [tag for (word, tag) in brown.tagged_words(categories='editorial')]
>>> print nltk.ConfusionMatrix(gold, test)
```

Based on such analysis we may decide to modify the tagset. Perhaps a distinction between tags that is difficult to make can be dropped, since it is not important in the context of some larger processing task.

Another way to analyze the performance bound on a tagger comes from the less than 100% agreement between human annotators.

In general, observe that the tagging process collapses distinctions: e.g., lexical identity is usually lost when all personal pronouns are tagged `PRP`. At the same time, the tagging process introduces new distinctions and removes ambiguities: e.g., *deal* tagged as `VB` or `NN`. This characteristic of collapsing certain distinctions and introducing new distinctions is an important feature of tagging which facilitates classification and prediction. When we introduce finer distinctions in a tagset, an n-gram tagger gets more detailed information about the left-context when it is deciding what tag to assign to a particular word. However, the tagger simultaneously has to do more work to classify the current token, simply because there are more tags to choose from. Conversely, with fewer distinctions (as with the simplified tagset), the tagger has less information about context, and it has a smaller range of choices in classifying the current token.

We have seen that ambiguity in the training data leads to an upper limit in tagger performance. Sometimes more context will resolve the ambiguity. In other cases, however, as noted by (Abney, 1996), the ambiguity can be resolved only with reference to syntax or to world knowledge. Despite these imperfections, part-of-speech tagging has played a central role in the rise of statistical approaches to natural language processing. In the early 1990s, the surprising accuracy of statistical taggers was a striking

demonstration that it was possible to solve one small part of the language understanding problem, namely part-of-speech disambiguation, without reference to deeper sources of linguistic knowledge. Can this idea be pushed further? In Chapter 7, we will see that it can.

Tagging Across Sentence Boundaries

An n-gram tagger uses recent tags to guide the choice of tag for the current word. When tagging the first word of a sentence, a trigram tagger will be using the part-of-speech tag of the previous two tokens, which will normally be the last word of the previous sentence and the sentence-ending punctuation. However, the lexical category that closed the previous sentence has no bearing on the one that begins the next sentence.

To deal with this situation, we can train, run, and evaluate taggers using lists of tagged sentences, as shown in Example 5-5.

Example 5-5. N-gram tagging at the sentence level.

```
brown_tagged_sents = brown.tagged_sents(categories='news')
brown_sents = brown.sents(categories='news')

size = int(len(brown_tagged_sents) * 0.9)
train_sents = brown_tagged_sents[:size]
test_sents = brown_tagged_sents[size:]

t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(train_sents, backoff=t0)
t2 = nltk.BigramTagger(train_sents, backoff=t1)
```

```
>>> t2.evaluate(test_sents)
0.84491179108940495
```

5.6 Transformation-Based Tagging

A potential issue with n-gram taggers is the size of their n-gram table (or language model). If tagging is to be employed in a variety of language technologies deployed on mobile computing devices, it is important to strike a balance between model size and tagger performance. An n-gram tagger with backoff may store trigram and bigram tables, which are large, sparse arrays that may have hundreds of millions of entries.

A second issue concerns context. The only information an n-gram tagger considers from prior context is tags, even though words themselves might be a useful source of information. It is simply impractical for n-gram models to be conditioned on the identities of words in the context. In this section, we examine Brill tagging, an inductive tagging method which performs very well using models that are only a tiny fraction of the size of n-gram taggers.

Brill tagging is a kind of *transformation-based learning*, named after its inventor. The general idea is very simple: guess the tag of each word, then go back and fix the mistakes.

In this way, a Brill tagger successively transforms a bad tagging of a text into a better one. As with n-gram tagging, this is a *supervised learning* method, since we need annotated training data to figure out whether the tagger's guess is a mistake or not. However, unlike n-gram tagging, it does not count observations but compiles a list of transformational correction rules.

The process of Brill tagging is usually explained by analogy with painting. Suppose we were painting a tree, with all its details of boughs, branches, twigs, and leaves, against a uniform sky-blue background. Instead of painting the tree first and then trying to paint blue in the gaps, it is simpler to paint the whole canvas blue, then "correct" the tree section by over-painting the blue background. In the same fashion, we might paint the trunk a uniform brown before going back to over-paint further details with even finer brushes. Brill tagging uses the same idea: begin with broad brush strokes, and then fix up the details, with successively finer changes. Let's look at an example involving the following sentence:

(1) The President said he will ask Congress to increase grants to states for vocational rehabilitation.

We will examine the operation of two rules: (a) replace `NN` with `VB` when the previous word is `TO`; (b) replace `TO` with `IN` when the next tag is `NNS`. Table 5-6 illustrates this process, first tagging with the unigram tagger, then applying the rules to fix the errors.

Table 5-6. Steps in Brill tagging

Phrase	to	increase	grants	to	states	for	vocational	rehabilitation
Unigram	TO	*NN*	NNS	*TO*	NNS	IN	JJ	NN
Rule 1		*VB*						
Rule 2				*IN*				
Output	TO	VB	NNS	IN	NNS	IN	JJ	NN
Gold	TO	VB	NNS	IN	NNS	IN	JJ	NN

In this table, we see two rules. All such rules are generated from a template of the following form: "replace T_1 with T_2 in the context C." Typical contexts are the identity or the tag of the preceding or following word, or the appearance of a specific tag within two to three words of the current word. During its training phase, the tagger guesses values for T_1, T_2, and C, to create thousands of candidate rules. Each rule is scored according to its net benefit: the number of incorrect tags that it corrects, less the number of correct tags it incorrectly modifies.

Brill taggers have another interesting property: the rules are linguistically interpretable. Compare this with the n-gram taggers, which employ a potentially massive table of n-grams. We cannot learn much from direct inspection of such a table, in comparison to the rules learned by the Brill tagger. Example 5-6 demonstrates NLTK's Brill tagger.

Example 5-6. Brill tagger demonstration: The tagger has a collection of templates of the form X → Y if the preceding word is Z; the variables in these templates are instantiated to particular words and tags to create "rules"; the score for a rule is the number of broken examples it corrects minus the number of correct cases it breaks; apart from training a tagger, the demonstration displays residual errors.

```
>>> nltk.tag.brill.demo()
Training Brill tagger on 80 sentences...
Finding initial useful rules...
    Found 6555 useful rules.

           B      |
   S   F   r   O  |        Score = Fixed - Broken
   c   i   o   t  |  R     Fixed = num tags changed incorrect -> correct
   o   x   k   h  |  u     Broken = num tags changed correct -> incorrect
   r   e   e   e  |  l     Other = num tags changed incorrect -> incorrect
   e   d   n   r  |  e
------------------+-------------------------------------------------------
  12  13   1   4  | NN -> VB if the tag of the preceding word is 'TO'
   8   9   1  23  | NN -> VBD if the tag of the following word is 'DT'
   8   8   0   9  | NN -> VBD if the tag of the preceding word is 'NNS'
   6   9   3  16  | NN -> NNP if the tag of words i-2...i-1 is '-NONE-'
   5   8   3   6  | NN -> NNP if the tag of the following word is 'NNP'
   5   6   1   0  | NN -> NNP if the text of words i-2...i-1 is 'like'
   5   5   0   3  | NN -> VBN if the text of the following word is '*-1'
   ...
>>> print(open("errors.out").read())
             left context |    word/test->gold     | right context
--------------------------+-----------------------+--------------------------
                          |      Then/NN->RB       | ,/, in/IN the/DT guests/N
, in/IN the/DT guests/NNS |       '/VBD->POS       | honor/NN ,/, the/DT speed
'/POS honor/NN ,/, the/DT |    speedway/JJ->NN     | hauled/VBD out/RP four/CD
NN ,/, the/DT speedway/NN |     hauled/NN->VBD     | out/RP four/CD drivers/NN
DT speedway/NN hauled/VBD |      out/NNP->RP       | four/CD drivers/NNS ,/, c
dway/NN hauled/VBD out/RP |      four/NNP->CD      | drivers/NNS ,/, crews/NNS
hauled/VBD out/RP four/CD |    drivers/NNP->NNS    | ,/, crews/NNS and/CC even
P four/CD drivers/NNS ,/, |     crews/NN->NNS      | and/CC even/RB the/DT off
NNS and/CC even/RB the/DT |    official/NNP->JJ    | Indianapolis/NNP 500/CD a
                          |     After/VBD->IN      | the/DT race/NN ,/, Fortun
ter/IN the/DT race/NN ,/, |    Fortune/IN->NNP     | 500/CD executives/NNS dro
s/NNS drooled/VBD like/IN |  schoolboys/NNP->NNS   | over/IN the/DT cars/NNS a
olboys/NNS over/IN the/DT |      cars/NN->NNS      | and/CC drivers/NNS ./.
```

5.7 How to Determine the Category of a Word

Now that we have examined word classes in detail, we turn to a more basic question: how do we decide what category a word belongs to in the first place? In general, linguists use morphological, syntactic, and semantic clues to determine the category of a word.

Morphological Clues

The internal structure of a word may give useful clues as to the word's category. For example, *-ness* is a suffix that combines with an adjective to produce a noun, e.g., *happy* → *happiness*, *ill* → *illness*. So if we encounter a word that ends in *-ness*, this is very likely to be a noun. Similarly, *-ment* is a suffix that combines with some verbs to produce a noun, e.g., *govern* → *government* and *establish* → *establishment*.

English verbs can also be morphologically complex. For instance, the **present participle** of a verb ends in *-ing*, and expresses the idea of ongoing, incomplete action (e.g., *falling*, *eating*). The *-ing* suffix also appears on nouns derived from verbs, e.g., *the falling of the leaves* (this is known as the **gerund**).

Syntactic Clues

Another source of information is the typical contexts in which a word can occur. For example, assume that we have already determined the category of nouns. Then we might say that a syntactic criterion for an adjective in English is that it can occur immediately before a noun, or immediately following the words *be* or *very*. According to these tests, *near* should be categorized as an adjective:

(2) a. the near window
    b. The end is (very) near.

Semantic Clues

Finally, the meaning of a word is a useful clue as to its lexical category. For example, the best-known definition of a noun is semantic: "the name of a person, place, or thing." Within modern linguistics, semantic criteria for word classes are treated with suspicion, mainly because they are hard to formalize. Nevertheless, semantic criteria underpin many of our intuitions about word classes, and enable us to make a good guess about the categorization of words in languages with which we are unfamiliar. For example, if all we know about the Dutch word *verjaardag* is that it means the same as the English word *birthday*, then we can guess that *verjaardag* is a noun in Dutch. However, some care is needed: although we might translate *zij is vandaag jarig* as *it's her birthday today*, the word *jarig* is in fact an adjective in Dutch, and has no exact equivalent in English.

New Words

All languages acquire new lexical items. A list of words recently added to the Oxford Dictionary of English includes *cyberslacker*, *fatoush*, *blamestorm*, *SARS*, *cantopop*, *bupkis*, *noughties*, *muggle*, and *robata*. Notice that all these new words are nouns, and this is reflected in calling nouns an **open class**. By contrast, prepositions are regarded as a **closed class**. That is, there is a limited set of words belonging to the class (e.g., *above*, *along*, *at*, *below*, *beside*, *between*, *during*, *for*, *from*, *in*, *near*, *on*, *outside*, *over*,

past, *through*, *towards*, *under*, *up*, *with*), and membership of the set only changes very gradually over time.

Morphology in Part-of-Speech Tagsets

Common tagsets often capture some **morphosyntactic** information, that is, information about the kind of morphological markings that words receive by virtue of their syntactic role. Consider, for example, the selection of distinct grammatical forms of the word *go* illustrated in the following sentences:

(3) a. *Go* away!
    b. He sometimes *goes* to the cafe.
    c. All the cakes have *gone*.
    d. We *went* on the excursion.

Each of these forms—*go*, *goes*, *gone*, and *went*—is morphologically distinct from the others. Consider the form *goes*. This occurs in a restricted set of grammatical contexts, and requires a third person singular subject. Thus, the following sentences are ungrammatical.

(4) a. *They sometimes *goes* to the cafe.
    b. *I sometimes *goes* to the cafe.

By contrast, *gone* is the past participle form; it is required after *have* (and cannot be replaced in this context by *goes*), and cannot occur as the main verb of a clause.

(5) a. *All the cakes have *goes*.
    b. *He sometimes *gone* to the cafe.

We can easily imagine a tagset in which the four distinct grammatical forms just discussed were all tagged as `VB`. Although this would be adequate for some purposes, a more fine-grained tagset provides useful information about these forms that can help other processors that try to detect patterns in tag sequences. The Brown tagset captures these distinctions, as summarized in Table 5-7.

Table 5-7. Some morphosyntactic distinctions in the Brown tagset

Form	Category	Tag
go	base	VB
goes	third singular present	VBZ
gone	past participle	VBN
going	gerund	VBG
went	simple past	VBD

In addition to this set of verb tags, the various forms of the verb *to be* have special tags: `be/BE`, `being/BEG`, `am/BEM`, `are/BER`, `is/BEZ`, `been/BEN`, `were/BED`, and `was/BEDZ` (plus extra tags for negative forms of the verb). All told, this fine-grained tagging of verbs means that an automatic tagger that uses this tagset is effectively carrying out a limited amount of **morphological analysis**.

Most part-of-speech tagsets make use of the same basic categories, such as noun, verb, adjective, and preposition. However, tagsets differ both in how finely they divide words into categories, and in how they define their categories. For example, *is* might be tagged simply as a verb in one tagset, but as a distinct form of the *lexeme be* in another tagset (as in the Brown Corpus). This variation in tagsets is unavoidable, since part-of-speech tags are used in different ways for different tasks. In other words, there is no one "right way" to assign tags, only more or less useful ways depending on one's goals.

5.8 Summary

- Words can be grouped into classes, such as nouns, verbs, adjectives, and adverbs. These classes are known as lexical categories or parts-of-speech. Parts-of-speech are assigned short labels, or tags, such as `NN` and `VB`.
- The process of automatically assigning parts-of-speech to words in text is called part-of-speech tagging, POS tagging, or just tagging.
- Automatic tagging is an important step in the NLP pipeline, and is useful in a variety of situations, including predicting the behavior of previously unseen words, analyzing word usage in corpora, and text-to-speech systems.
- Some linguistic corpora, such as the Brown Corpus, have been POS tagged.
- A variety of tagging methods are possible, e.g., default tagger, regular expression tagger, unigram tagger, and n-gram taggers. These can be combined using a technique known as backoff.
- Taggers can be trained and evaluated using tagged corpora.
- Backoff is a method for combining models: when a more specialized model (such as a bigram tagger) cannot assign a tag in a given context, we back off to a more general model (such as a unigram tagger).
- Part-of-speech tagging is an important, early example of a sequence classification task in NLP: a classification decision at any one point in the sequence makes use of words and tags in the local context.
- A dictionary is used to map between arbitrary types of information, such as a string and a number: `freq['cat'] = 12`. We create dictionaries using the brace notation: `pos = {}`, `pos = {'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}`.
- N-gram taggers can be defined for large values of *n*, but once *n* is larger than 3, we usually encounter the sparse data problem; even with a large quantity of training data, we see only a tiny fraction of possible contexts.

- Transformation-based tagging involves learning a series of repair rules of the form "change tag *s* to tag *t* in context *c*," where each rule fixes mistakes and possibly introduces a (smaller) number of errors.

5.9 Further Reading

Extra materials for this chapter are posted at *http://www.nltk.org/*, including links to freely available resources on the Web. For more examples of tagging with NLTK, please see the Tagging HOWTO at *http://www.nltk.org/howto*. Chapters 4 and 5 of (Jurafsky & Martin, 2008) contain more advanced material on n-grams and part-of-speech tagging. Other approaches to tagging involve machine learning methods (Chapter 6). In Chapter 7, we will see a generalization of tagging called *chunking* in which a contiguous sequence of words is assigned a single tag.

For tagset documentation, see `nltk.help.upenn_tagset()` and `nltk.help.brown_tagset()`. Lexical categories are introduced in linguistics textbooks, including those listed in Chapter 1 of this book.

There are many other kinds of tagging. Words can be tagged with directives to a speech synthesizer, indicating which words should be emphasized. Words can be tagged with sense numbers, indicating which sense of the word was used. Words can also be tagged with morphological features. Examples of each of these kinds of tags are shown in the following list. For space reasons, we only show the tag for a single word. Note also that the first two examples use XML-style tags, where elements in angle brackets enclose the word that is tagged.

Speech Synthesis Markup Language (W3C SSML)
: `That is a <emphasis>big</emphasis> car!`

SemCor: Brown Corpus tagged with WordNet senses
: `Space in any <wf pos="NN" lemma="form" wnsn="4">form</wf> is completely measured by the three dimensions.` (Wordnet form/nn sense 4: "shape, form, configuration, contour, conformation")

Morphological tagging, from the Turin University Italian Treebank
: `E' italiano , come progetto e realizzazione , il primo (PRIMO ADJ ORDIN MSING) porto turistico dell' Albania .`

Note that tagging is also performed at higher levels. Here is an example of dialogue act tagging, from the NPS Chat Corpus (Forsyth & Martell, 2007) included with NLTK. Each turn of the dialogue is categorized as to its communicative function:

```
Statement  User117 Dude..., I wanted some of that
ynQuestion User120 m I missing something?
Bye        User117 I'm gonna go fix food, I'll be back later.
System     User122 JOIN
System     User2   slaps User122 around a bit with a large trout.
Statement  User121 18/m pm me if u tryin to chat
```

5.10 Exercises

1. ○ Search the Web for "spoof newspaper headlines," to find such gems as: *British Left Waffles on Falkland Islands*, and *Juvenile Court to Try Shooting Defendant*. Manually tag these headlines to see whether knowledge of the part-of-speech tags removes the ambiguity.
2. ○ Working with someone else, take turns picking a word that can be either a noun or a verb (e.g., *contest*); the opponent has to predict which one is likely to be the most frequent in the Brown Corpus. Check the opponent's prediction, and tally the score over several turns.
3. ○ Tokenize and tag the following sentence: *They wind back the clock, while we chase after the wind*. What different pronunciations and parts-of-speech are involved?
4. ○ Review the mappings in Table 5-4. Discuss any other examples of mappings you can think of. What type of information do they map from and to?
5. ○ Using the Python interpreter in interactive mode, experiment with the dictionary examples in this chapter. Create a dictionary `d`, and add some entries. What happens whether you try to access a non-existent entry, e.g., `d['xyz']`?
6. ○ Try deleting an element from a dictionary `d`, using the syntax `del d['abc']`. Check that the item was deleted.
7. ○ Create two dictionaries, `d1` and `d2`, and add some entries to each. Now issue the command `d1.update(d2)`. What did this do? What might it be useful for?
8. ○ Create a dictionary `e`, to represent a single lexical entry for some word of your choice. Define keys such as `headword`, `part-of-speech`, `sense`, and `example`, and assign them suitable values.
9. ○ Satisfy yourself that there are restrictions on the distribution of *go* and *went*, in the sense that they cannot be freely interchanged in the kinds of contexts illustrated in (3), Section 5.7.
10. ○ Train a unigram tagger and run it on some new text. Observe that some words are not assigned a tag. Why not?
11. ○ Learn about the affix tagger (type `help(nltk.AffixTagger)`). Train an affix tagger and run it on some new text. Experiment with different settings for the affix length and the minimum word length. Discuss your findings.
12. ○ Train a bigram tagger with no backoff tagger, and run it on some of the training data. Next, run it on some new data. What happens to the performance of the tagger? Why?
13. ○ We can use a dictionary to specify the values to be substituted into a formatting string. Read Python's library documentation for formatting strings (*http://docs.py*

thon.org/lib/typesseq-strings.html) and use this method to display today's date in two different formats.

14. ◑ Use `sorted()` and `set()` to get a sorted list of tags used in the Brown Corpus, removing duplicates.
15. ◑ Write programs to process the Brown Corpus and find answers to the following questions:
    a. Which nouns are more common in their plural form, rather than their singular form? (Only consider regular plurals, formed with the *-s* suffix.)
    b. Which word has the greatest number of distinct tags? What are they, and what do they represent?
    c. List tags in order of decreasing frequency. What do the 20 most frequent tags represent?
    d. Which tags are nouns most commonly found after? What do these tags represent?
16. ◑ Explore the following issues that arise in connection with the lookup tagger:
    a. What happens to the tagger performance for the various model sizes when a backoff tagger is omitted?
    b. Consider the curve in Figure 5-4; suggest a good size for a lookup tagger that balances memory and performance. Can you come up with scenarios where it would be preferable to minimize memory usage, or to maximize performance with no regard for memory usage?
17. ◑ What is the upper limit of performance for a lookup tagger, assuming no limit to the size of its table? (Hint: write a program to work out what percentage of tokens of a word are assigned the most likely tag for that word, on average.)
18. ◑ Generate some statistics for tagged data to answer the following questions:
    a. What proportion of word types are always assigned the same part-of-speech tag?
    b. How many words are ambiguous, in the sense that they appear with at least two tags?
    c. What percentage of word *tokens* in the Brown Corpus involve these ambiguous words?
19. ◑ The `evaluate()` method works out how accurately the tagger performs on this text. For example, if the supplied tagged text was `[('the', 'DT'), ('dog', 'NN')]` and the tagger produced the output `[('the', 'NN'), ('dog', 'NN')]`, then the score would be `0.5`. Let's try to figure out how the evaluation method works:
    a. A tagger `t` takes a list of words as input, and produces a list of tagged words as output. However, `t.evaluate()` is given correctly tagged text as its only parameter. What must it do with this input before performing the tagging?

b. Once the tagger has created newly tagged text, how might the `evaluate()` method go about comparing it with the original tagged text and computing the accuracy score?

c. Now examine the source code to see how the method is implemented. Inspect `nltk.tag.api.__file__` to discover the location of the source code, and open this file using an editor (be sure to use the *api.py* file and not the compiled *api.pyc* binary file).

20. ◑ Write code to search the Brown Corpus for particular words and phrases according to tags, to answer the following questions:

a. Produce an alphabetically sorted list of the distinct words tagged as `MD`.

b. Identify words that can be plural nouns or third person singular verbs (e.g., *deals*, *flies*).

c. Identify three-word prepositional phrases of the form IN + DET + NN (e.g., *in the lab*).

d. What is the ratio of masculine to feminine pronouns?

21. ◑ In Table 3-1, we saw a table involving frequency counts for the verbs *adore*, *love*, *like*, and *prefer*, and preceding qualifiers such as *really*. Investigate the full range of qualifiers (Brown tag `QL`) that appear before these four verbs.

22. ◑ We defined the `regexp_tagger` that can be used as a fall-back tagger for unknown words. This tagger only checks for cardinal numbers. By testing for particular prefix or suffix strings, it should be possible to guess other tags. For example, we could tag any word that ends with *-s* as a plural noun. Define a regular expression tagger (using `RegexpTagger()`) that tests for at least five other patterns in the spelling of words. (Use inline documentation to explain the rules.)

23. ◑ Consider the regular expression tagger developed in the exercises in the previous section. Evaluate the tagger using its `accuracy()` method, and try to come up with ways to improve its performance. Discuss your findings. How does objective evaluation help in the development process?

24. ◑ How serious is the sparse data problem? Investigate the performance of n-gram taggers as *n* increases from 1 to 6. Tabulate the accuracy score. Estimate the training data required for these taggers, assuming a vocabulary size of 10^5 and a tagset size of 10^2.

25. ◑ Obtain some tagged data for another language, and train and evaluate a variety of taggers on it. If the language is morphologically complex, or if there are any orthographic clues (e.g., capitalization) to word classes, consider developing a regular expression tagger for it (ordered after the unigram tagger, and before the default tagger). How does the accuracy of your tagger(s) compare with the same taggers run on English data? Discuss any issues you encounter in applying these methods to the language.

26. ◑ Example 5-4 plotted a curve showing change in the performance of a lookup tagger as the model size was increased. Plot the performance curve for a unigram tagger, as the amount of training data is varied.
27. ◑ Inspect the confusion matrix for the bigram tagger `t2` defined in Section 5.5, and identify one or more sets of tags to collapse. Define a dictionary to do the mapping, and evaluate the tagger on the simplified data.
28. ◑ Experiment with taggers using the simplified tagset (or make one of your own by discarding all but the first character of each tag name). Such a tagger has fewer distinctions to make, but much less information on which to base its work. Discuss your findings.
29. ◑ Recall the example of a bigram tagger which encountered a word it hadn't seen during training, and tagged the rest of the sentence as `None`. It is possible for a bigram tagger to fail partway through a sentence even if it contains no unseen words (even if the sentence was used during training). In what circumstance can this happen? Can you write a program to find some examples of this?
30. ◑ Preprocess the Brown News data by replacing low-frequency words with *UNK*, but leaving the tags untouched. Now train and evaluate a bigram tagger on this data. How much does this help? What is the contribution of the unigram tagger and default tagger now?
31. ◑ Modify the program in Example 5-4 to use a logarithmic scale on the *x*-axis, by replacing `pylab.plot()` with `pylab.semilogx()`. What do you notice about the shape of the resulting plot? Does the gradient tell you anything?
32. ◑ Consult the documentation for the Brill tagger demo function, using `help(nltk.tag.brill.demo)`. Experiment with the tagger by setting different values for the parameters. Is there any trade-off between training time (corpus size) and performance?
33. ◑ Write code that builds a dictionary of dictionaries of sets. Use it to store the set of POS tags that can follow a given word having a given POS tag, i.e., $\text{word}_i \rightarrow \text{tag}_i \rightarrow \text{tag}_{i+1}$.
34. ● There are 264 distinct words in the Brown Corpus having exactly three possible tags.
    a. Print a table with the integers 1..10 in one column, and the number of distinct words in the corpus having 1..10 distinct tags in the other column.
    b. For the word with the greatest number of distinct tags, print out sentences from the corpus containing the word, one for each possible tag.
35. ● Write a program to classify contexts involving the word *must* according to the tag of the following word. Can this be used to discriminate between the epistemic and deontic uses of *must*?
36. ● Create a regular expression tagger and various unigram and n-gram taggers, incorporating backoff, and train them on part of the Brown Corpus.

a. Create three different combinations of the taggers. Test the accuracy of each combined tagger. Which combination works best?

b. Try varying the size of the training corpus. How does it affect your results?

37. ● Our approach for tagging an unknown word has been to consider the letters of the word (using `RegexpTagger()`), or to ignore the word altogether and tag it as a noun (using `nltk.DefaultTagger()`). These methods will not do well for texts having new words that are not nouns. Consider the sentence *I like to blog on Kim's blog*. If *blog* is a new word, then looking at the previous tag (`TO` versus `NP$`) would probably be helpful, i.e., we need a default tagger that is sensitive to the preceding tag.

    a. Create a new kind of unigram tagger that looks at the tag of the previous word, and ignores the current word. (The best way to do this is to modify the source code for `UnigramTagger()`, which presumes knowledge of object-oriented programming in Python.)

    b. Add this tagger to the sequence of backoff taggers (including ordinary trigram and bigram taggers that look at words), right before the usual default tagger.

    c. Evaluate the contribution of this new unigram tagger.

38. ● Consider the code in Section 5.5, which determines the upper bound for accuracy of a trigram tagger. Review Abney's discussion concerning the impossibility of exact tagging (Abney, 2006). Explain why correct tagging of these examples requires access to other kinds of information than just words and tags. How might you estimate the scale of this problem?

39. ● Use some of the estimation techniques in `nltk.probability`, such as *Lidstone* or *Laplace* estimation, to develop a statistical tagger that does a better job than n-gram backoff taggers in cases where contexts encountered during testing were not seen during training.

40. ● Inspect the diagnostic files created by the Brill tagger `rules.out` and `errors.out`. Obtain the demonstration code by accessing the source code (at *http://www.nltk.org/code*) and create your own version of the Brill tagger. Delete some of the rule templates, based on what you learned from inspecting `rules.out`. Add some new rule templates which employ contexts that might help to correct the errors you saw in `errors.out`.

41. ● Develop an n-gram backoff tagger that permits "anti-n-grams" such as `["the", "the"]` to be specified when a tagger is initialized. An anti-n-gram is assigned a count of zero and is used to prevent backoff for this n-gram (e.g., to avoid estimating P(*the* | *the*) as just P(*the*)).

42. ● Investigate three different ways to define the split between training and testing data when developing a tagger using the Brown Corpus: genre (`category`), source (`fileid`), and sentence. Compare their relative performance and discuss which method is the most legitimate. (You might use n-fold cross validation, discussed in Section 6.3, to improve the accuracy of the evaluations.)