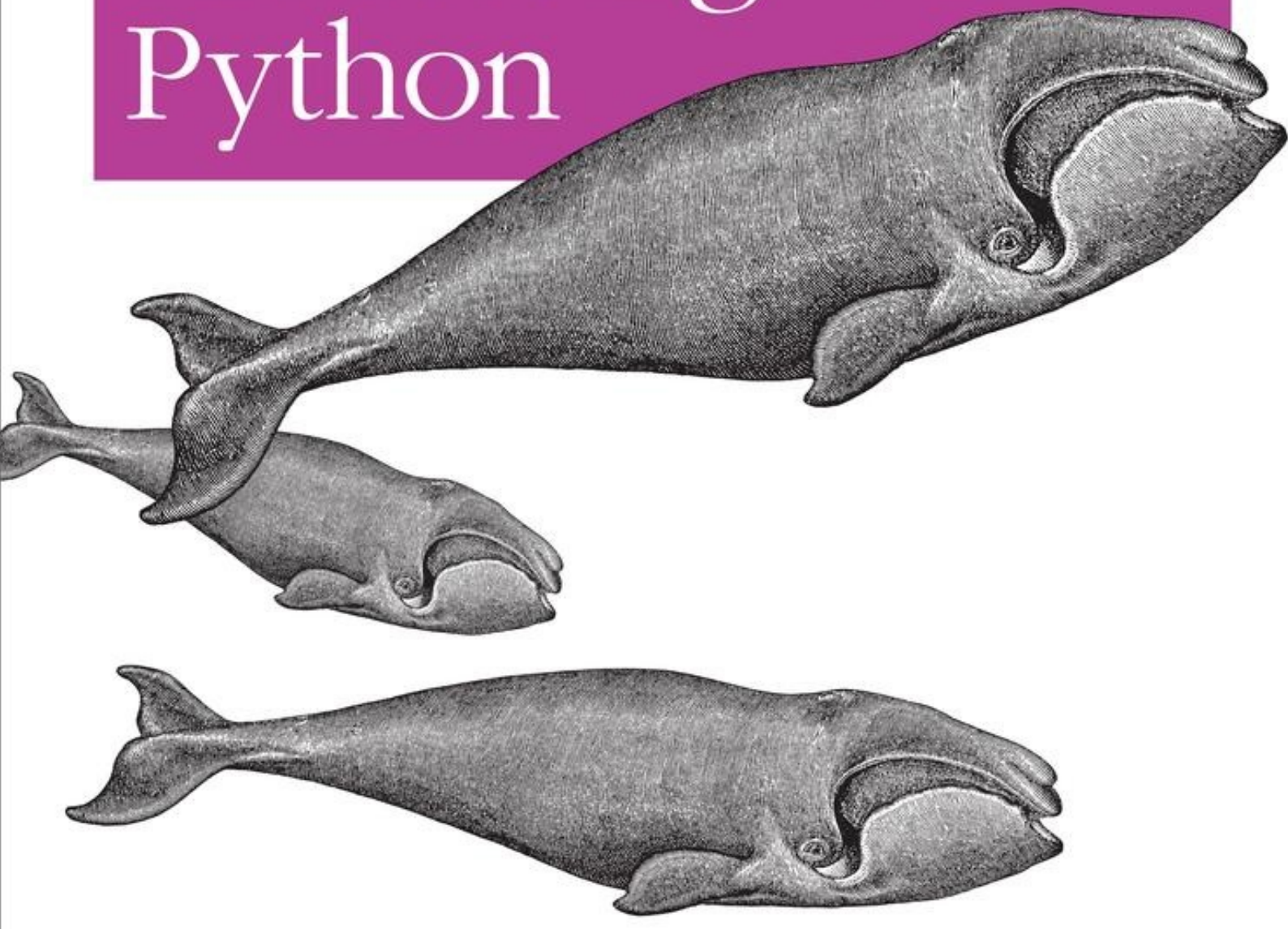
Analyzing Text with the Natural Language Toolkit

Natural Language Processing with Python



Steven Bird, Ewan Klein & Edward Loper

O'REILLY®

Natural Language Processing with Python

Steven Bird

Ewan Klein

Edward Loper

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Special Upgrade Offer

If you purchased this ebook directly from oreilly.com, you have the following benefits:

- DRM-free ebooks—use your ebooks across devices without restrictions or limitations
- Multiple formats—use on your laptop, tablet, or phone
- Lifetime access, with free updates
- Dropbox syncing—your files, anywhere

If you purchased this ebook from another retailer, you can upgrade your ebook to take advantage of a
these benefits for just $4.99. Click here to access your ebook upgrade.

Please note that upgrade offers are not available from sample content.

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 cove 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 handhe 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 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 i 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 exemplifie 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 cove 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 presente 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 1.

Table 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 i 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 languag 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 min 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/, wit 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 c 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 variabl 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 writ 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 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, a 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 an re-used easily. As a dynamic language, Python permits attributes to be added to objects on the fly, a 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 oft praised for the way it facilitates productivity, quality, and maintainability of software. A collection o 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 th 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 downloa 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 sample that produce line graphs and bar charts.

NetworkX (optional)

This is a library for storing and manipulating network structures with 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 inferenc

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 i dozens of universities, and serves as the basis of many research projects. Table 2 lists the most important NLTK modules.

Table 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
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 practica 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 encyclopedi 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 belie 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 t cover both the theoretical and practical sides of the subject in such a short span of time. Some course 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 designe 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 fo 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 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 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 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 f 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.

> **TIP**
>
> This icon signifies a tip, suggestion, or general note.

> **CAUTION**
>
> 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 reproducin 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, Ewa 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 t 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 mean 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, se 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 Cier 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.

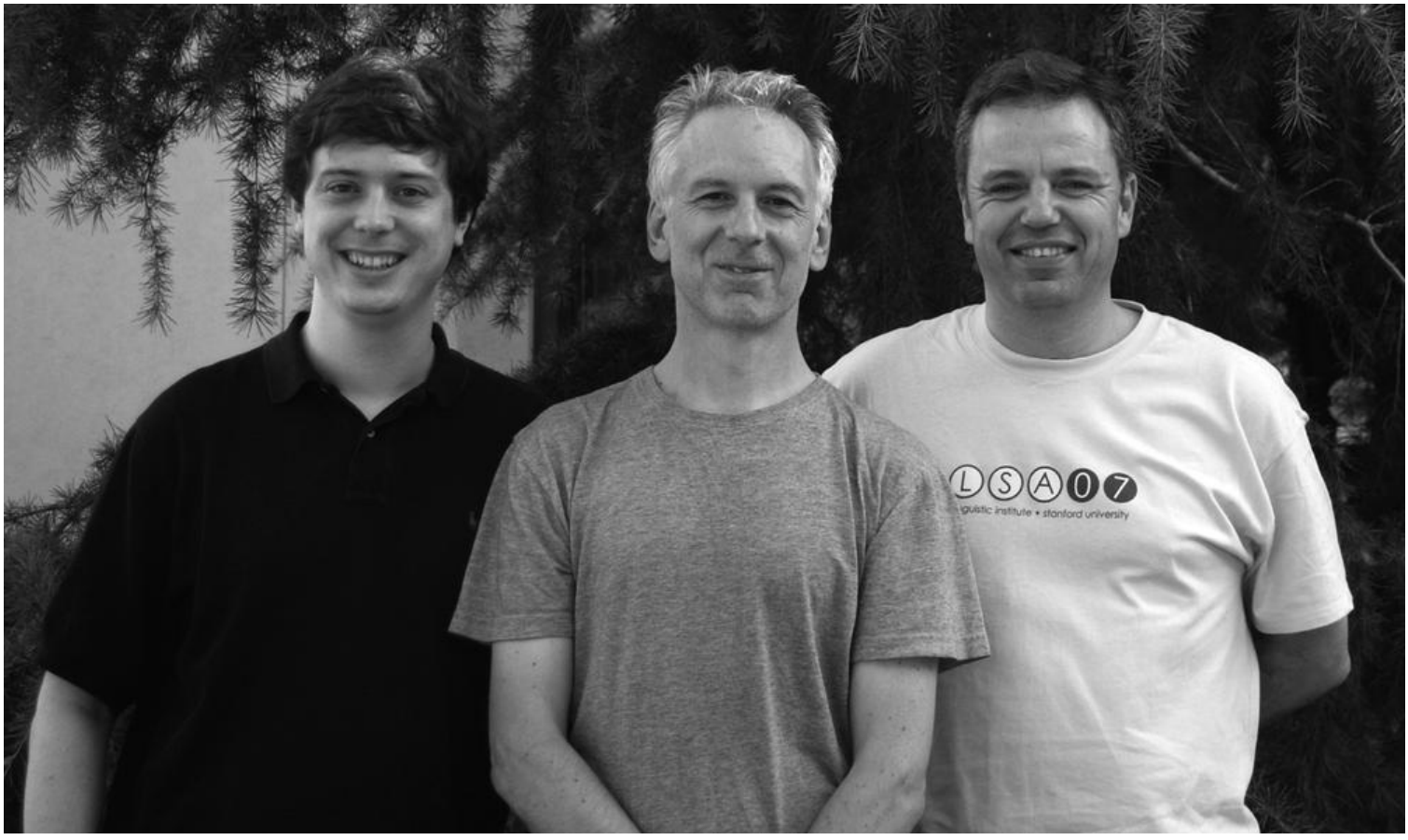


Figure 1. Edward Loper, Ewan Klein, and Steven Bird, Stanford, July 2007

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 writ 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 a 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 Automatic Natural Language Understanding; 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.

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.
>>>
```

> **NOTE**
>
> 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.

> **NOTE**
>
> **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 stand 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 fo 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()
```

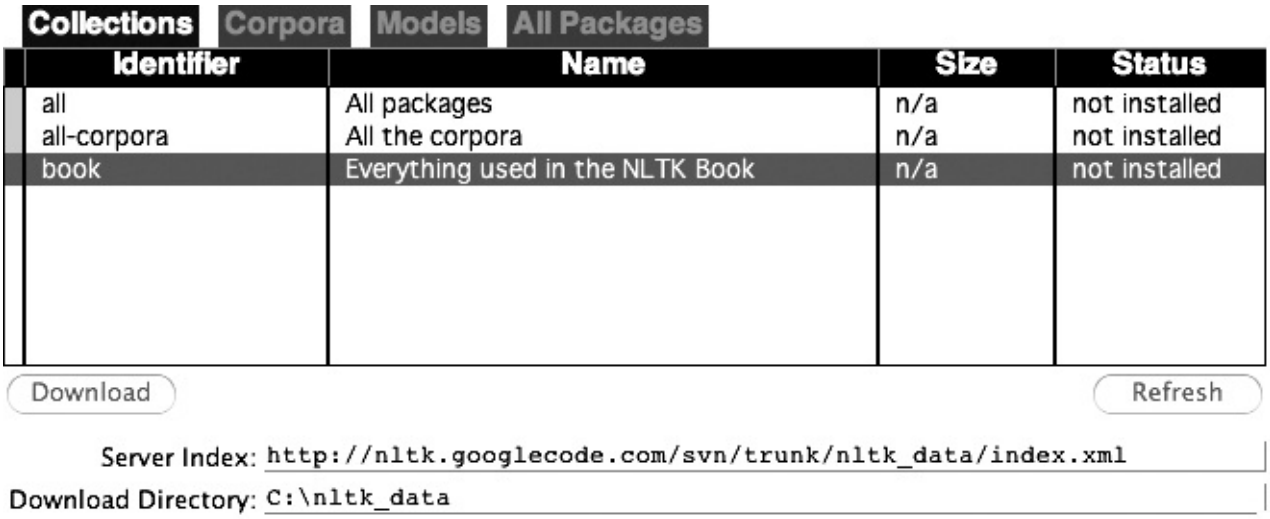


Figure 1-1. Downloading the NLTK Book Collection: Browse the available packages using `nltk.download()`. The Collections tab on the downloader shows how the packages are grouped into sets, and you should select the line labeled book to obtain all data required for the examples and exercises in this book. 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, loa all items." The `book` module contains all the data you will need as you read this chapter. After printin 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
>>>
```

NOTE

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
>>>
```

NOTE

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 g the quotes, commas, brackets, and parentheses exactly right.

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

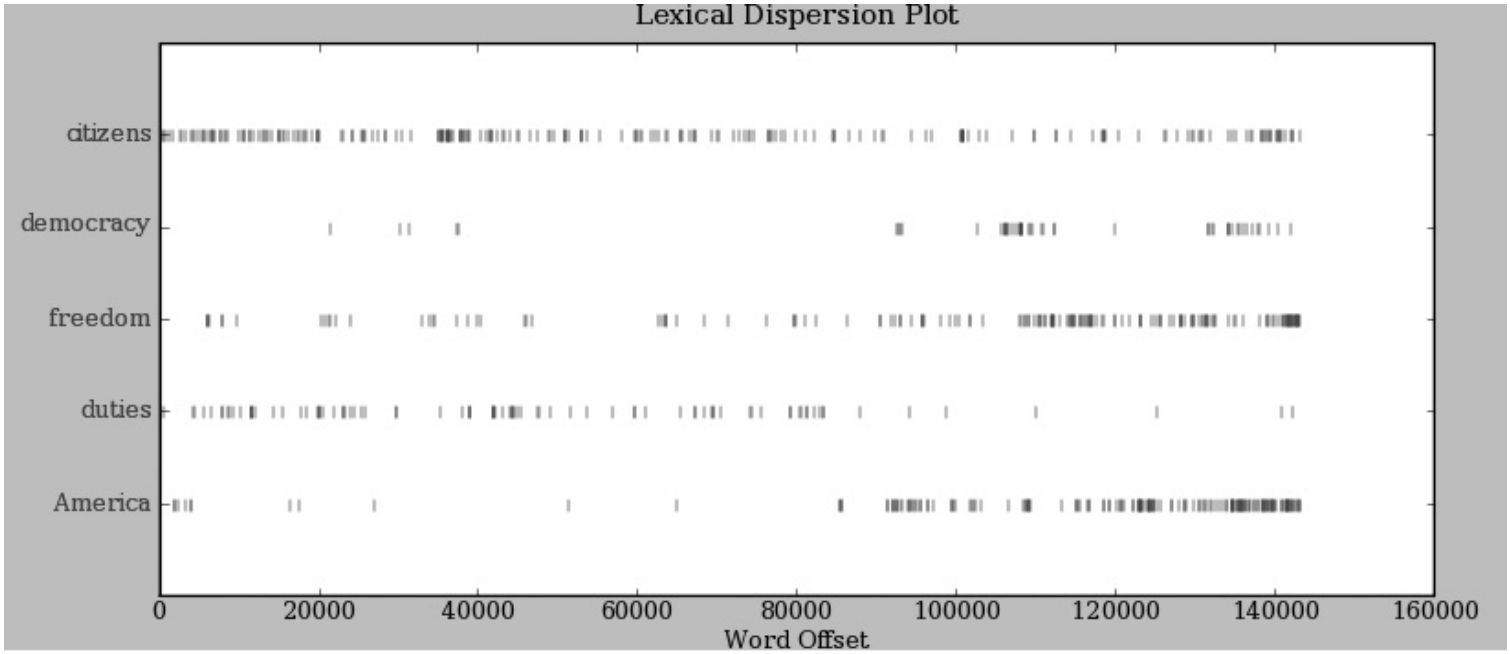


Figure 1-2. Lexical dispersion plot for words in U.S. Presidential Inaugural Addresses: This can be used to investigate changes i language use over time.

> **NOTE**
>
> **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 d 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?)

> **NOTE**
>
> 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 understandin 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 fo a sequence of characters—such as `hairy`, `his`, or `:)`—that we want to treat as a group. When we cou the number of tokens in a text, say, the phrase *to be or not to be*, we are counting occurrences of thes 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—tha 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 eac 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
>>>
```

> **NOTE**
>
> **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 o 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 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 t 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 a 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 tha 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 fin 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.